INFORMATIK
BERICHTE

369 – 12/2013

Symbolic Trajectories

Ralf Hartmut Güting, Fabio Valdés, Maria Luisa Damiani

Fakultät für Mathematik und Informatik
D-58084 Hagen

Symbolic Trajectories*

Ralf Hartmut Güting[†] Fabio Valdés[†] Maria Luisa Damiani[‡]

Abstract

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, e.g., by data mining techniques. A question is how the resulting "meaningful" trajectories can be represented or further queried.

In this paper, we propose a very simple generic model called *symbolic trajectory* to capture a wide range of 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, behaviours of animals, cinemas within 2 kms distance, etc.

Besides the model, the core technical contribution of the paper 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, as well as 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.

1 Introduction

Due to the wide-spread use of GPS-enabled devices such as smartphones or 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 [46, 19].

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 time stamps, corresponding to the way data are recorded by devices. At a higher level of abstraction it is a

*This work was partially supported by the EU COST Action IC0903 "Knowledge Discovery from Moving Objects" (MOVE).

[†]Databases for New Applications, FernUniversität Hagen, 58084 Hagen, Germany, {rhg, fabio.valdes}@fernuni-hagen.de

[‡]Department of Computer Science, University of Milan, 39, Via Comelico, Milan 20135, Italy, damiani@di.unimi.it

continuous function from time into 2D space that may be represented by an abstract data type moving point [18].

A lot of efforts has gone into data mining on large sets of trajectories [17]. Whereas the general goal is to discover any kind of interesting phenomena on such data sets, 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/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 she is walking, going by bicycle or using a bus.

Clearly it is necessary to somehow represent such findings and there is a corresponding trend to consider "semantic trajectories". The work of Spaccapietra et al. [35] suggests to view a trajectory as a sequence of stops and moves. Further work in [36] generalizes the model and defines a semantic trajectory as an annotated trajectory where annotations have a type, e.g., stop/move, and regard the whole trajectory or parts of it.

However, the models proposed under the label "semantic trajectories" are designed as fairly complex extended entity relationship models that need to be mapped to corresponding sets of relations. They are not suitable for simple and elegant querying.

In contrast, in this paper we propose a simple and flexible generic model to represent any kind of semantic information that one might want to associate with a trajectory. Due to its simplicity it lends itself to elegant and expressive query formulation. We call this model a *symbolic trajectory.*

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 $< (i_1, l_1), ..., (i_n, l_n) >$ 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) >
```

A symbolic trajectory can express any time dependent symbolic information that can be derived from a geometric trajectory (sometimes called a *raw trajectory* in the literature). The symbolic information can be computed from the movement itself or be obtained by relating the geometric trajectory to its environment, e.g., static geometries or other moving objects. Hence it is an abstraction that captures certain aspects of a precise geometric trajectory. Here are some examples of possible interesting aspects:

- Names of roads traversed by a vehicle, obtained by map matching
- Cell identifiers of a cellular network
- Cardinal directions such as *north*, *northwest*, ...
- Speed profiles: *slow*, *moderate*, *fast*
- For animals: *at home range*, *migrating*, *stopover*, ...
- Indoor navigation: triples of (building, floor, room)
- Names of countries traversed on a long distance trip
- Activities of a tourist
- For animals: *grazing*, *resting*, *at feeding station*, ...
- Animal movement: *valley*, *level ground*, *mountain range*, *descending*, *climbing*
- Transportation mode: *car*, *bus*, *taxi*, *bicycle*, ...

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

- Current weather at moving entity: *sun*, *rain*, *snow*, *ice*, ...

We follow the framework of [18] and represent a symbolic trajectory as a 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 (for example **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(Trip: mpoint, RoadName: mlabel, Speed: mlabel)
```

where the road name is obtained from map matching and the speed from a classification of speeds. We call this a *hybrid* or *multi-dimensional trajectory*. It is of particular interest to manipulate symbolic trajectories together with their related geometric trajectories. An example is shown in Section 5.2.

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.

The core technical contribution of this paper 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.

Here are some simple examples:

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

This pattern matches symbolic trajectories containg 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.

The main contributions of the paper are the following:

- We introduce the concept of symbolic trajectories as a generic representation of meanings for trajectories.

- Symbolic trajectories 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. 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 refers not only to labels but 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.
- Rigorous formal definitions for the syntax and semantics of pattern matching and rewriting are provided.
- Efficient implementation of the language is presented in detail using non-deterministic finite automata.
- 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.

The rest of the paper is structured as follows: Section 2 reviews the concepts from [18] that are later needed for the definition of symbolic trajectories in Section 3. Section 4 presents the pattern language including formal definitions of syntax and semantics. Section 5 presents as examples two ways of using symbolic trajectories to represent personal trips; it addresses also the construction of symbolic from raw trajectories. 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 [14] 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 to future work.

2 Preliminaries

The model of symbolic trajectories proposed in this paper fits into and extends a comprehensive framework for representing and querying moving objects in databases [15, 18, 16]. In this section we briefly review the essential concepts of this framework needed later.

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*.

Formally, a system of types and operations is a (many-sorted) algebra. It consists of a signature which provides sorts and operations, defining for each operation the argument sorts and the result sort. A signature defines a set of terms. To define the semantics, one needs to assign carrier sets to the sorts and functions to the operations that are mappings on the respective carrier sets. The signature together with carrier sets and functions defines the algebra.

In the framework discussed, data types are built from some basic types and type constructors. The type system is itself described by a signature. In this signature, the sorts are so-called *kinds* and the operations are *type constructors*. The terms of the signature are exactly the available types of the type system. For example, consider a signature

int, *real*, *bool*:		$\rightarrow$	BASE
array:	BASE	$\rightarrow$	ARRAY

It has kinds BASE and ARRAY and type constructors *int*, *real*, *bool*, and *array*. The types defined are the terms of the signature, namely, *int*, *real*, *bool*, *array*(*int*), *array*(*real*), *array*(*bool*). Note that basic types are just type constructors without arguments.

The type system defined in [18] for moving objects is shown in Figure 1.

Type Constructor	Signature		
int, *real*, *string*, *bool*		$\rightarrow$	BASE
point, *points*, *line*, *region*		$\rightarrow$	SPATIAL
instant		$\rightarrow$	TIME
moving, *intime*	BASE $\cup$ SPATIAL	$\rightarrow$	TEMPORAL
range	BASE $\cup$ TIME	$\rightarrow$	RANGE

Figure 1: Type system defined in [18]

We first explain the type system informally. It has some basic standard types and some spatial data types. Type *instant* represents the continuous domain of time. Type constructor *moving* provides for a given static type a corresponding time dependent type. The *intime* constructor yields for a static type α a type whose values are pairs of an *instant* and a value of type α. The *range* constructor provides for a given type another type whose values are finite sets of disjoint intervals over the domain of α.

To provide formally the semantics of the data types, one needs to define their domains, or carrier sets. An important distinction introduced in [15, 18, 16] is that between an abstract model and a discrete model. In an *abstract model*, the domain may be defined in terms of infinite sets. Such a model is conceptually simple, but not directly implementable. In contrast, in a *discrete model*, the possible values of a data type must be defined in terms of finite representations. These can be mapped to data structures in the implementation.

For example, a $\underline{region}$ data type can be defined in an abstract model as a regular closed subset of the Euclidean plane, whereas in a discrete model it would be defined as a collection of disjoint polygons each of which may have polygonal holes.

Reference [18] defines an abstract model of data types and operations for moving objects whereas [16] provides a corresponding discrete model.

Some notations used in defining semantics of types are A_α and D_α to denote the carrier set of type α in the abstract and discrete model, respectively. When a carrier set A_α contains an undefined value $\perp$, then the notation $\bar{A}_\alpha$ refers to the carrier set without the undefined value, i.e., $\bar{A}_\alpha = A_\alpha \setminus \{\perp\}$. With these notations, the carrier set of the $\underline{moving}$ type constructor is defined as follows.[2]

Definition 2.1 Given a data type α to which type constructor $\underline{moving}$ is applicable, the carrier set of type $\underline{moving}(\alpha)$ is

$$A_{\underline{moving}(\alpha)} := \{f | f : A_{\underline{instant}} \to \bar{A}_\alpha \text{ is a partial function}\}$$

□

Note that the abstract model disregards completely the issue of how such functions can be represented. A function $f : A_{\underline{instant}} \to A_\alpha$ is simply an infinite set of pairs from $A_{\underline{instant}} \times A_\alpha$.

The discrete model of [16] provides finite representations for all the types of the abstract model. For types $\underline{moving}(\alpha)$ the so-called sliced representation is introduced. That means, to represent a function of time, the time domain is cut into disjoint time intervals (*slices*) such that within each slice the development can be represented by some simple function of time. "Simple" actually means finitely representable. In other words, the function for a slice can be described by a few parameters rather than an infinite set of pairs. Figure 2 illustrates the sliced representation for a $\underline{moving}(\underline{real})$ and a $\underline{moving}(\underline{point})$.

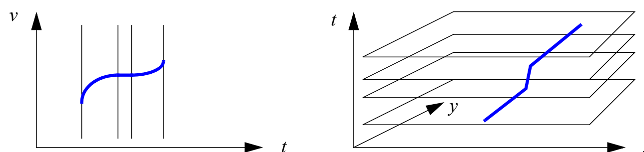


Figure 2: Sliced representations for $\underline{moving}(\underline{real})$ and $\underline{moving}(\underline{point})$

The representation of a single slice, consisting of the time interval and the function description, is called a *unit*.

For the given data types, a comprehensive set of operations is defined. Most of them are generic and applicable to many of the available data types. Two examples are

deftime:	$\underline{moving}(\alpha)$	$\to$	$\underline{periods}$
atinstant:	$\underline{moving}(\alpha) \times \underline{instant}$	$\to$	$\underline{intime}(\alpha)$

Here type $\underline{periods}$ is just an abbreviation of $\underline{range}(\underline{instant})$. Hence operation **deftime** returns the set of time intervals when a moving object is defined.

These two example operations are generic because they range over the types generated by type constructor $\underline{moving}$. A second technique to define generic operations used in [18] introduces so-called *spaces*, where a space contains *point types* and *point set types*. Essentially, a value of a point type is a single element of some domain ("space") and a point set type has a set of values from this domain. Two example spaces are *Int* and *2D*. The *Int* space has types $\underline{int}$ and

[2] The definition in [18] has an additional condition requesting that such a function has only a finite number of continuous components, omitted here.

range(*int*). The first can represent a single *int* value and the second a set of *int* values. Hence *int* is the point type and *range*(*int*) the point set type of space *Int*. For space *2D*, *point* is the point type and *points*, *line*, *region* are the point set types (as *line* and *region* are viewed as infinite sets of points from the 2D plane).

Generic operations are then defined to range over point and point set types of any space. Here π denotes a point type and σ a point set type. In defining semantics of operations, u and v (respectively U and V) refer to the first and second argument if it is of a point type (respectively a point set type). Two operations defined in this way are

Type Constructor	Signature			Semantics
inside:	$\pi \times \sigma$	$\rightarrow$	*bool*	$u \in V$
union:	$\sigma \times \sigma$	$\rightarrow$	σ	$U \cup V$

One signature covered by this definition is **inside**: *point* $\times$ *line* $\rightarrow$ *bool*.

Finally, *lifting* is introduced as a mechanism to make static and related time dependent operations consistent. Lifting means that for a given static operation each of the arguments may become time dependent (i.e., replacing in the signature type α by *moving*(α)) which makes the result time dependent as well. Furthermore the semantics of the lifted operation is derived using the semantics of the static operation for every instant of time. Hence by lifting we also have an operation **inside**: *moving*(*point*) $\times$ *line* $\rightarrow$ *moving*(*bool*).

See [18] for further details and the complete definition of types and operations.

3 Symbolic Trajectories

A symbolic trajectory is intended to capture some time dependent "symbolic" property of a moving object. Such a property can be derived from the movement itself or it can relate the movement to some environment (i.e., the space in which the object moves). As such, it is an abstraction that captures certain aspects of a precise geometric trajectory. Some interesting aspects (called *profiles*) are the following:

- Road names: the sequence of road names of roads traversed, obtained by map matching techniques.
- Speed profile: symbols characterizing the speed such as *fast*, *moderate*, *slow*, or a classification into ranges such as *0-20*, *20-50*, *50-80*, ...
- Personal locations: *home*, *work*, *friend 1*, *friend 2*, *cinema*, *bookstore*, ...
- Travel modes: *walk*, *bicycle*, *car*, *bus*, ...
- Activities: *sleeping*, *eating*, *watching TV*, ...
- Places traversed: the sequence of districts of a city passed, or of the cells of a mobile phone network.

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

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

where *instant* is the data type representing the continuous domain of time, and *label* is the type of labels. Clearly a symbolic trajectory can be represented by a type *moving*(*label*).

There are three further variants: a time dependent *set of labels*, a time dependent *place*, or a time dependent *set of places*. A place is a pair consisting of a label and an integer such as (cinema, 114) where the integer component is a reference into 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.

Data Types We introduce four data types label, labels, place, and places defined as follows.

Definition 3.1 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 [18].

We integrate the new types into the type system of [18] as shown in Figure 3, using a new kind SYMBOLIC.

Type Constructor	Signature	
int, real, string, bool		→ BASE
label, labels, place, places		→ SYMBOLIC
point, points, line, region		→ SPATIAL
instant		→ TIME
moving, intime	BASE ∪ SYMBOLIC ∪ SPATIAL	→ TEMPORAL
range	BASE ∪ TIME	→ RANGE

Figure 3: Extended type system

The three further kinds of symbolic trajectories correspond to data types moving(labels), moving(place), and moving(places). For them as well as for type moving(label) the semantics is already given in Definition 2.1. We also use abbreviations mlabel, mlabels, mplace, mplaces.

The purpose of the set types is to allow one to associate with a moving object not just one but several properties or places. We can introduce 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.

Discrete Model So far we have defined symbolic trajectories within the abstract model, viewing them as functions of time. The discrete model follows from [16] by analogy to the representation of types moving(α) with a discrete domain D_α such as moving(int) or moving(bool). Effectively, a symbolic trajectory is represented at the discrete level as a sequence of units where each unit consists of a time interval and a value from the respective type (label, labels, place, places). 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 $u_1...u_n$ for $n \geq 0$ where u_i is a unit, or as a sequence of pairs $< (i_1, l_1), ..., (i_n, l_n) >$ where i_j is a time interval and l_j a label (respectively set of labels, etc.). Time intervals are represented as four-tuples (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 [16] 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 a about a quarter of an hour on January 17, 2013.

In the sequel we follow a convention for representing nested list structures used in LISP and in the SECONDO system (which is the environment for implementation described below). There a list $< a_1, a_2, ..., a_n >$ is represented as $(a_1\ a_2\ ...\ a_n)$ where each element is either an atom or a list. Further, we may omit the boolean flags left-closed and right-closed because they are usually not interesting. The example trajectory then looks as follows:

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

Operations on Symbolic Trajectories Due to the integration into the model of [18] we have already inherited some generic operations. We make one further step by recognizing that *label* and *labels* as well as *place* and *places* form spaces as shown in Figure 4.

Space	Point Type	Point Set Type
Label	*label*	*labels*
Place	*place*	*places*

Figure 4: New types viewed as point or point set types

This means that all operations defined on spaces are inherited. These are also subject to lifting, hence the time dependent versions are defined as well.

For example, the desired set operations on symbolic trajectories are already formally defined in [18] (as **union**, **intersection**, **minus**).

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

Example 3.1 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 Figure 5):

- 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.

```
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 am?

```
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.

[3] Operations **sometimes** and **theInstant** are not defined in [18]. **sometimes** is shown to be a derived operation in [19], Exercise 4.5. **theInstant** is available in the implementation in the SECONDO system.

```
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 3.1 are shown in Figure 5 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.

Operator	Signature			Syntax
passes:	moving(label) × label	→	bool	_ # _
at:	moving(label) × label	→	moving(label)	_ # _
deftime:	moving(label)	→	periods	# (_)
atinstant:	moving(label) × instant	→	intime(label)	_ # _
val:	intime(label)	→	label	# (_)
intersection:	moving(label) × moving(label)	→	moving(label)	# (_, _)
=:	moving(label) × moving(label)	→	moving(bool)	_ # _
sometimes:	moving(bool)	→	bool	# (_)
theInstant:	int × int × int × int × int	→	instant	# (_,_,_,_,_)

Figure 5: Operations used in queries of Example 3.1

4 Pattern Matching and Rewriting

The generic operations inherited for symbolic trajectories from [18] already permit expressive queries. Nevertheless we are now interested in making the language even more expressive by providing a 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 mlabel. It is straightforward to extend this to the other three types of symbolic trajectories.

4.1 Patterns

In Section 3 we have seen that a symbolic trajectory can be represented as a nested list of the form

```
( ((s1 e1 lc1 rc1) l1) ... ((sn en lcn rcn) 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*, *left closed* and *right closed*.

A *pattern* describes such a list with some desired structure or contents by approximating this notation. As the left closed and right closed flags are not of interest, we can first simplify the notation and represent the trajectory as

```
((s1 e1) l1) ... ((sn en) ln) or
(t1 l1) ... (tn ln)
```

A pattern might look as follows:

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

Here a pair in parentheses denotes a unit pattern, i.e., 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 <u>*mlabel*</u> 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 4.1 A *simple pattern* is a sequence of *simple pattern elements* $< p_1, ..., p_n >$ 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 4.5.

Unit patterns and sequence patterns of the forms `*` and `+` are called *atomic pattern elements*. This notion is relevant in the implementation (Section 6).

Definition 4.2 (pattern matching)

(i) *Unit Patterns.* Let $u = (u_t, u_l)$ be a unit of type <u>*ulabel*</u>.

- `(` 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 $U =< u_1, ..., u_n >, n \geq 0$ be a sequence of units, each u_i of type <u>*ulabel*</u>.

- `*` 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 :⇔ there exists a partitioning of U into subsequences $U_1...U_m, m \geq 1$ such that $U = U_1 \circ ... \circ U_m$ and $\forall i \in \{1, ..., m\},\ p$ matches U_i.
- [p]* matches U :⇔ [p]+ matches $U \vee n = 0$.
- [p]? matches U :⇔ [p] matches $U \vee n = 0$.

(iii) Let $U =< u_1, ..., u_n >, n \geq 0$ be a sequence of units, each u_i of type <u>*ulabel*</u>. Let $P = p_1...p_m$ be a simple pattern.

P matches U :⇔ there exists a partitioning of U into subsequences $U_1...U_m$ such that $U = U_1 \circ ... \circ U_m$ and $\forall i \in \{1, ..., m\},\ p_i$ matches U_i.

□

Example 4.1 In Section 3 we have seen an example trajectory in a database of personal trips:

```
( ( (2013-01-17-9:02:30 2013-01-17-9:05:51) "Queen Anne St")
  ( (2013-01-17-9:05:51 2013-01-17-9:10:16) "Wimpole St")
  ...
  ( (2013-01-17-9:18:44 2013-01-17-9:20:10) "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 above. □

4.2 Variables

We now add variables to patterns. Their purpose is twofold: (i) to allow us to specify further conditions on subsequences matched by pattern elements, and (ii) 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) *
```

The above pattern has 5 elements. If the pattern matches an <u>*mlabel*</u>, 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 4.3 Let V be a domain of variable names. A *pattern* is a sequence of pattern elements $P =< e_1, ..., e_n >$ 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 $< p_1, ..., p_n >$ and *var*(P) denotes the set of variables occurring in P. □

Definition 4.4 A *binding* is a set of pairs $B = \{(v_1, U_1), ..., (v_k, U_k)\}$ where each pair consists of a variable v_i and a sequence of units U_i of type $\underline{ulabel}$. All variables are distinct. $var(B) = \{v_1, ..., v_k\}$ denotes the set of variables occurring in B. □

Definition 4.5 (pattern matching with binding) Let $U =< u_1, ..., u_n >, n \geq 0$ be a sequence of units, each u_i of type $\underline{ulabel}$. Let $P =< e_1, ..., e_m >$ be a pattern and $simple(P) = < p_1, ..., p_m >$.

P matches U with binding B $:\Leftrightarrow$ there exists a partitioning of U into subsequences $U_1...U_m$ such that $U = U_1 \circ ... \circ U_m$ and $\forall i \in \{1, ..., m\},\ p_i$ 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 4.6 (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 $\underline{label}$ with value u_l
- *time* of type $\underline{periods}$ with value u_t
- *start* of type $\underline{instant}$ with value s
- *end* of type $\underline{instant}$ with value e
- *leftclosed* of type $\underline{bool}$ with value lc
- *rightclosed* of type $\underline{bool}$ with value rc

(ii) v is a sequence variable and $U =< (t_1, l_1), ..., (t_n, l_n) >,\ n \geq 1$, with $t_i = (s_i, e_i, lc_i, rc_i)$. Then v has attributes

- *labels* of type $\underline{labels}$ with value $\{l_1, ..., l_n\}$
- *time* of type $\underline{periods}$ with value $\bigcup_{i=1}^{n} t_i$
- *card* of type $\underline{int}$ with value n
- *start* of type $\underline{instant}$ with value s_1
- *end* of type $\underline{instant}$ with value e_n
- *leftclosed* of type $\underline{bool}$ with value lc_1
- *rightclosed* of type $\underline{bool}$ 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) but all values are undefined, e.g. $v.labels = \perp$.

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 4.2 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 above 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
```

□

4.3 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 4.3 For example, we can restrict the previous round trip query to trips taking no more than twenty 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 one minute. We can also find round trips starting and ending at arbitrary streets 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 4.7 (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 $\sigma \in \Sigma_{wt}$, $w = t_1 ... t_n$ are the argument types and t is the result type. The operator's evaluation function is $f_\sigma : dom(t_1) \times ... \times dom(t_n) \to dom(t)$.

□

Definition 4.8 (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 $m \geq 0, e_1, ..., e_m$ are expressions of types $t_1, ..., t_m$, respectively, and $\sigma \in \Sigma_{t_1,...,t_m,t} \Rightarrow$ $\sigma(e_1, ..., e_m)$ is an expression of type t, and
$eval(\sigma(e_1, ..., e_m), B) = f_\sigma(eval(e_1, B), ..., eval(e_m, B))$.

□

Definition 4.9 A *pattern with conditions* is a pair (P, C) where P is a pattern and C a set of expressions of type *bool* over P. □

Definition 4.10 (pattern matching for patterns with conditions) Let $U = \ < u_1, ..., u_n >, n \geq 0$ 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 $\forall c \in C : eval(c, B) = true$.
□

4.4 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) Z * (_ c) * => X Y
```

For any *mlabel* value matching this pattern, the rule returns an *mlabel* value consisting just of the first two units. Similarly, the rule

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

returns the sequence matched by Z. The result is in any case of type *mlabel* even if only a single unit variable is mentioned in the result pattern as in

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

Obviously variables that are used neither in conditions nor in result patterns can be omitted, hence the above three rules could be simplified to

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

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, which 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 4.4 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*. For example, 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.

We now formalize rewrite rules and their semantics.

Definition 4.11 A *rule* is a four-tuple $R = (P, C, RP, A)$ where

(i) (P, C) is a pattern with conditions (C may be empty).

(ii) RP, called the *result pattern*, is a sequence of variables $RP =< Y_1, ..., Y_l >$ such that

- $\forall i \in \{1, ..., l\}, Y_i \in var(P) \cup V$ (V the domain of variables),
- $i \neq j \Rightarrow Y_i \neq Y_j$ (all variables are distinct)
- $P =< e_1, ..., e_n > \wedge (\exists i, j, u, v : e_i = (Y_u, p_i) \wedge e_j = (Y_v, p_j) \wedge i < j) \Rightarrow u < v$ (variables from the pattern occur in the result sequence in the same order).

(iii) A, called the set of *assignments*, is $A = \{a_1, ..., a_k\}$ such that

- $\forall i \in \{1, ..., k\}$, a_i, called an *assignment*, is a triple $a_i = (Y, attr, e)$, $Y \in \{Y_1, ..., Y_l\}$, Y is a unit variable, *attr* is an attribute of a unit variable of type t, and e is an expression over P of type t.
- $(X_u, attr_u, e_u) \in A \wedge (X_v, attr_v, e_v) \in A \wedge (X_u, attr_u) = (X_v, attr_v) \Rightarrow e_u = e_v$ (no two assignments to the same attribute).
- $(X_u, time, e_u) \in A \Rightarrow \forall (X_u, attr, e) \in A : attr \neq start \wedge attr \neq end \wedge attr \neq leftclosed \wedge attr \neq leftclosed$ (no assignments to overlapping attributes).

We denote by $A(X)$ the subset of A for variable X.

□

We define the semantics of a rule in three steps: (i) what does it mean to apply assignments to a given unit, (ii) checking if a sequence of units is a correct <u>*mlabel*</u> value, and (iii) defining the set of symbolic trajectories resulting from applying a rule to a given trajectory.

Definition 4.12 (application of assignments to a unit) Let $u = ((s, e, lc, rc), l)$ be a unit of type <u>*ulabel*</u> and B a binding.

(i) Let $a = (X, attr, expr)$ be an assignment. The *application of a to u with binding B*, denoted $assign(u, a, B)$, is defined as

- $attr = label \Rightarrow assign(u, a, B) = ((s, e, lc, rc), eval(expr, B))$
- $attr = time \Rightarrow assign(u, a, B) = (eval(expr, B), l)$
- $attr = start \Rightarrow assign(u, a, B) = ((eval(expr, B), e, lc, rc), l)$
- $attr = end \Rightarrow assign(u, a, B) = ((s, eval(expr, B), lc, rc), l)$
- $attr = leftclosed \Rightarrow assign(u, a, B) = ((s, e, eval(expr, B), rc), l)$
- $attr = rightclosed \Rightarrow assign(u, a, B) = ((s, e, lc, eval(expr, B)), l)$

(ii) Let $A(X) = \{a_1, ..., a_k\}$ be a set of assignments to attributes of variable X. The *application of $A(X)$ to u with binding B* is

$$assign(u, A(X), B) = assign(...assign(assign(u, a_1, B), a_2, B)...a_k, B)$$

□

Definition 4.13 (correctness check for a sequence of units) Let $U =< u_1, ..., u_n >$, $n \geq 0$ be a sequence of units, each u_i of type <u>*ulabel*</u>. The function *correct_mlabel* checks whether U forms a correct <u>*mlabel*</u> value, defined as

$$correct_mlabel(U) = \begin{cases} \{U\} & \text{if all unit time intervals are disjoint} \\ & \text{and units are ordered by time} \\ \emptyset & \text{otherwise} \end{cases}$$

□

Definition 4.14 (application of a rule to a symbolic trajectory) Let $R = (P, C, RP, A)$ be a rule and $U =< u_1, ..., u_n >, n \geq 0$ be a sequence of units, each u_i of type *ulabel*. The application of R to U, denoted $apply(R, U)$, is a set of trajectories defined as follows.

Let $\mathcal{B} = \{B \mid (P, C)$ matches U with binding $B\}$. Let $RP =< Y_1, ..., Y_l >$.

$$apply(R, U) = \bigcup_{B \in \mathcal{B}} correct_mlabel(seq(Y_1, B) \circ ... \circ seq(Y_l, B))$$

where $\forall i \in \{1, ..., l\}$

$$seq(Y_i, B) = \begin{cases} V & \text{if } (Y_i, V) \in B,\ Y_i \text{ a sequence variable} \\ assign(u, A(Y_i), B) & \text{if } (Y_i, < u >) \in B,\ Y_i \text{ a unit variable} \\ assign(u_\perp, A(Y_i), B) & \text{if } Y_i \notin var(B) \end{cases}$$

Here $u_\perp$ denotes the undefined *ulabel* unit where only the defaults for lc, rc have been set, that is, $u_\perp = ((\perp, \perp, T, F), \perp)$.

□

Definition 4.14 can be summarized as follows. For a given pattern with conditions, we determine all possible bindings. For a given binding, an output sequence of units is constructed as the concatenation of subsequences, one for each variable in the result pattern. For a sequence variable, this subsequence is simply the sequence found in the binding. For a unit variable, it is the unit resulting from applying all existing assignments to the unit of the binding. For any new variable, it is the unit one obtains from applying existing assignments to an undefined unit. Finally, the resulting sequence of units is returned, if it is a correct *mlabel* value, and discarded, otherwise.

4.5 Unit Patterns in More Detail

Now that the general structure and semantics of our language for pattern matching and rewriting is 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 4.1. 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, 2010-07-05, respectively.
- an hour, minute or second on a particular day, e.g. 2010-07-05-14:30
- a range of dates, e.g. 2010∼2011, 2010-07∼2011-03
- a range of times, e.g. 2010-07-05-14:30∼2010-07-09-14
- a halfopen range, e.g. 2005-05∼ or ∼2010-12-06
- a day of the week, i.e. one of {sunday, monday, tuesday, ..., saturday}
- a month of the year, i.e. 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 $\{label_1, ..., label_n\}$
- 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.

4.6 Embedding Patterns into the Algebra

4.6.1 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* $\rightarrow$ *pattern* `_ #`

The operator is responsible for parsing the pattern or rule, checking for correctness, and converting it to a corresponding data structure.

Example 4.5 We can store our example rewrite rule as a value of type *pattern* by the command:

```
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 non-deterministic finite automaton, see Section 6) occurs only once.

matches:	*mlabel* $\times$ (*pattern* | *text*)	$\rightarrow$	*bool*	`_ # _`
rewrite:	*mlabel* $\times$ (*pattern* | *text*)	$\rightarrow$	*set*(*mlabel*)	`#( _ , _ )`

The **matches** operator returns true if the pattern matches the *mlabel* value (Definition 4.10). **rewrite** returns[4] 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 4.14). If the pattern does not match, the result set is empty.

[4]The result is specified here as *set*(*mlabel*). Technically, in the implementation in SECONDO, sets are passed between operations in pipelined mode which is expressed by a type constructor *stream*. Hence the actual result type in the implementation is *stream*(*mlabel*). The same holds for further operations defined in this section: In the implementation, the *set* type constructor is replaced by *stream*.

Example 4.6 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 4.5:

```
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. □

4.6.2 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:

- 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 auxiliary operation **toclassifier** is provided to construct 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: $\underline{classifier} \times \underline{mlabel} \rightarrow \underline{set}(\underline{text})$ `#( _ , _ )`

Example 4.7 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
```

□

5 Application Examples

In this section we show two different ways of deriving symbolic trajectories from geometric trajectories and related queries. Both examples are based on a collection of recorded trips of a person (as before in the paper). As mentioned in the introduction, the main purpose of this work is not to support the management of personal trips — the techniques presented are suitable to represent derived symbolic information for any kind of moving objects and to query and classify them efficiently. But the domain of personal trips is easy to understand for everybody and therefore suitable for examples. In contrast, describing interesting patterns in the movement of roe deer, for example, is a topic for biological experts.

5.1 Personal Trips Based on Locations

Suppose we have a relation with trajectories collected for the trips of a person with schema

```
Traces(No: int, Trip: mpoint)
```

The entire movement has been split temporally by months; i.e. each trip covers one month. There exists also a relation with places the person visits with schema

```
Locations(Name: label, Area: region)
```

Some places the person visits are

```
Home, Work, Church, MusicLesson, Tennis, John, Gabi, CityParking, ...
```

Each of these places is defined as a small region around the respective location. Beyond manual construction of such regions it is also possible to import addresses from a personal address book, map them to geographic coordinates (e.g. using some service by Google) and then compute a small circle around the location. There is also an entry `None` containing the difference region of some large box containing all trips and the other entries in table *Locations*, i.e., the area of `None` is $Box - \bigcup_{l \in Locations,\ l.Name \neq None} l.Area$.

5.1.1 Constructing Symbolic Trajectories

Symbolic trajectories are now constructed by intersecting geometric trajectories with these location areas. We assume that location areas are disjoint so that a moving object can at any given time be only within one of the areas. Then an $\underline{mlabel}$ suffices for representation (if there were overlapping areas one would use type $\underline{mlabels}$). As a result, the symbolic trajectory for the geometric trajectory shown in Figure 6 will be

Figure 6: Geometric trajectory passing through locations

```
(... Home) (... None) (... Forest) (... None) (... Work)
```

Symbolic trajectories can be constructed as shown in the following, using a few operations that are already available in the framework of [18] and its SECONDO implementation. Operations used are:

passes:	*mpoint* × *region*	→	*bool*	`_ # _`
at:	*mpoint* × *region*	→	*mpoint*	`_ # _`
components:	*mpoint*	→	*set*(*mpoint*)	`# ( _ )`
deftime:	*mpoint*	→	*periods*	`# ( _ )`
the_unit:	*periods* × *label*	→	*ulabel*	`# ( _ )`
makemvalue:	*set*(*ulabel*)	→	*mlabel*	`# ( _ )`

In SQL the construction may be expressed as follows:

```
let R1 =
  select t.No as No, l.Name as Name, components(t.trip at l.Area) as Piece
  from Traces as t, Locations as l
  where t.Trip passes l.Area;
```

The query finds pairs of moving point $t.Trip$ and region $l.Area$ such that $t.Trip$ passes through $l.Area$. By the **at** operation, the trip is reduced to the parts when it is inside the area (it may enter and leave more than once). The **components** operation splits the remaining trip into elements such that each element has a single continuous time interval. Such a fragment moving point is put into a result tuple, together with the number of the trip and the name of the region.

```
let R2 = select No, the_unit(deftime(Piece), Name) as ULabel from R1;
```

For each tuple of the previous relation, one unit of an *mlabel* is constructed based on the time interval (obtained by **deftime**) and the name of the location.

```
let Trips = select No, makemvalue(Piece) as Trip from R2 groupby No;
```

Finally, trips are grouped by number and the *mlabel* value is constructed from the units through operation **makemvalue**. Hence as a result we have a relation with symbolic trajectories with schema:

Trips(No: int, Trip: *mlabel*)

describing the trips of the person.

5.1.2 Some Queries

Based on this representation, we formulate queries:

Example 5.1 Find parts of trips that went from Home to CityParking and back Home, without visiting other places.

```
select No, rewrite(Trip, '*  X [(_ Home) () (_ CityParking) () (_ Home)]  * => X'
  as PartialTrip
from Trips
```

□

Example 5.2 Determine for every month (trip) how often the person went to work.

```
select No, count(rewrite(Trip, 'X [(_ Home) * (_ Work)] => X')) as Cnt
from Trips
```

□

Example 5.3 Were there any places visited on the way from Home to Work? Which?

```
select No, rewrite(Trip,
  '(_ Home) * X () * (_ Work) // X.Label # "None" => X') as Visit
from Trip
```

□

5.2 Personal Trips Based on Road Names

The second example uses symbolic trajectories containing road names obtained from map matching as in the earlier examples of Section 4. This example is based on real data collected over several months by one of the authors. All steps described below have been implemented in SECONDO.

5.2.1 Constructing Symbolic Trajectories

Here the construction consists of the two steps:

- Construct a road network from OpenStreetMap data.
- Match geometric trajectories to the road network.

The construction of the road network is beyond the scope of this paper but the interested reader can find the SECONDO script performing this task at [10] The result needed for map matching is a relation Edges with schema

```
Edges(Source: int, Target: int, SourcePos: point, TargetPos: point, Curve: line,
  RoadName: text, RoadType: text, MaxSpeed: text)
```

Here *Source* and *Target* identify nodes of the road network graph; a tuple of *Edges* represents a directed edge. Further attributes include locations of the source and target node, the geometry of the road along this edge, and textual information such as the road name. The relation is stored in a B-tree ordered first by *Source*, then by *Target* fields. This representation permits efficient retrieval of successors of a node needed in navigation tasks.

Furthermore, an R-tree index *EdgeIndex_Box_rtree* indexing the bounding boxes of the *Curve* attributes and a relation *EdgeIndex* connecting the R-tree with relation *Edges* are provided.

Input for the map matching is a relation constructed from the GPS observations with schema

```
Trips(TrackId: int, Trip: mpoint, Traj: line)
```

These trips are constructed in such a way that a new trip starts whenever a break in the observations of more than 5 minutes occurs.

Map matching itself is performed by an operation **omapmatchmht**. It takes as arguments the two relations *Edges* and *EdgeIndex* and the R-tree *EdgeIndex_Box_rtree* as well as one geometric trajectory of type *mpoint*. It returns the sequence of edges from *Edges* obtained by map matching. Each edge tuple is extended by fields *StartTime* and *EndTime* describing when the moving point entered and left the edge. The operator uses the so-called "multiple hypothesis technique". It was implemented by Matthias Roth [31] based on algorithms proposed in [28, 34].

The SECONDO command matching the trips from relation *Trips* is

```
let MatchedTrips = Trips feed addcounter[No, 1]
  extend[Matched:
    omapmatchmht(Edges, EdgeIndex_Box_rtree, EdgeIndex, .Trip) aconsume]
  consume
```

It adds to each tuple from *Trips* a counter value *No* and a subrelation *Matched* containing the sequence of extended edges delivered by **omapmatchmht**. Hence the result is a nested relation *MatchedTrips*.

The next query computes from this representation the symbolic trajectories.

```
let SymTrips = MatchedTrips feed
  extend[SymTrip: compress(
    .Matched afeed extend[SymUnit:
      the_unit(tolabel(.RoadName), .StartTime, .EndTime, TRUE, FALSE)]
    makemvalue[SymUnit])]
  remove[Matched]
  consume
```

For each tuple of *MatchedTrips*, the subrelation *Matched* is processed. For each tuple of *Matched*, a unit of type *ulabel* is constructed, using the road name for the label and the *StartTime* and *EndTime* fields to define the time interval. This unit is appended to the tuple in field *SymUnit*. Finally, the **makemvalue** operator collects from each arriving tuple the *SymUnit* value and puts them all together into a single *mlabel* value.

Further, operation **compress** is applied to the *mlabel* value which is then stored in field *SymTrip*. The **compress** operation merges adjacent fields of a symbolic trajectory that have adjacent time intervals and the same label value. This is necessary to obtain each road traversed only once rather than with a label for each edge between two road intersections.

Finally, the subrelation *Matched* is removed as it is not needed any more.

The resulting relation SymTrips has schema

```
SymTrips(TrackId: int, Trip: mpoint, Traj: line, No: int, SymTrip: mstring)
```

Note that it contains in each tuple together its moving point *Trip*, the spatial projection *Traj* and the derived symbolic form *SymTrip*. Indeed it is usually interesting to consider a symbolic trajectory not in isolation but together with the geometric trajectory it describes.

5.2.2 Some Queries

Example 5.4 Find round trips starting and ending at "Alte Teichstraße".

```
let roundtrip = '(_ "Alte Teichstraße") + (_ "Alte Teichstraße")'

select * from symtrips where symtrip matches roundtrip
```

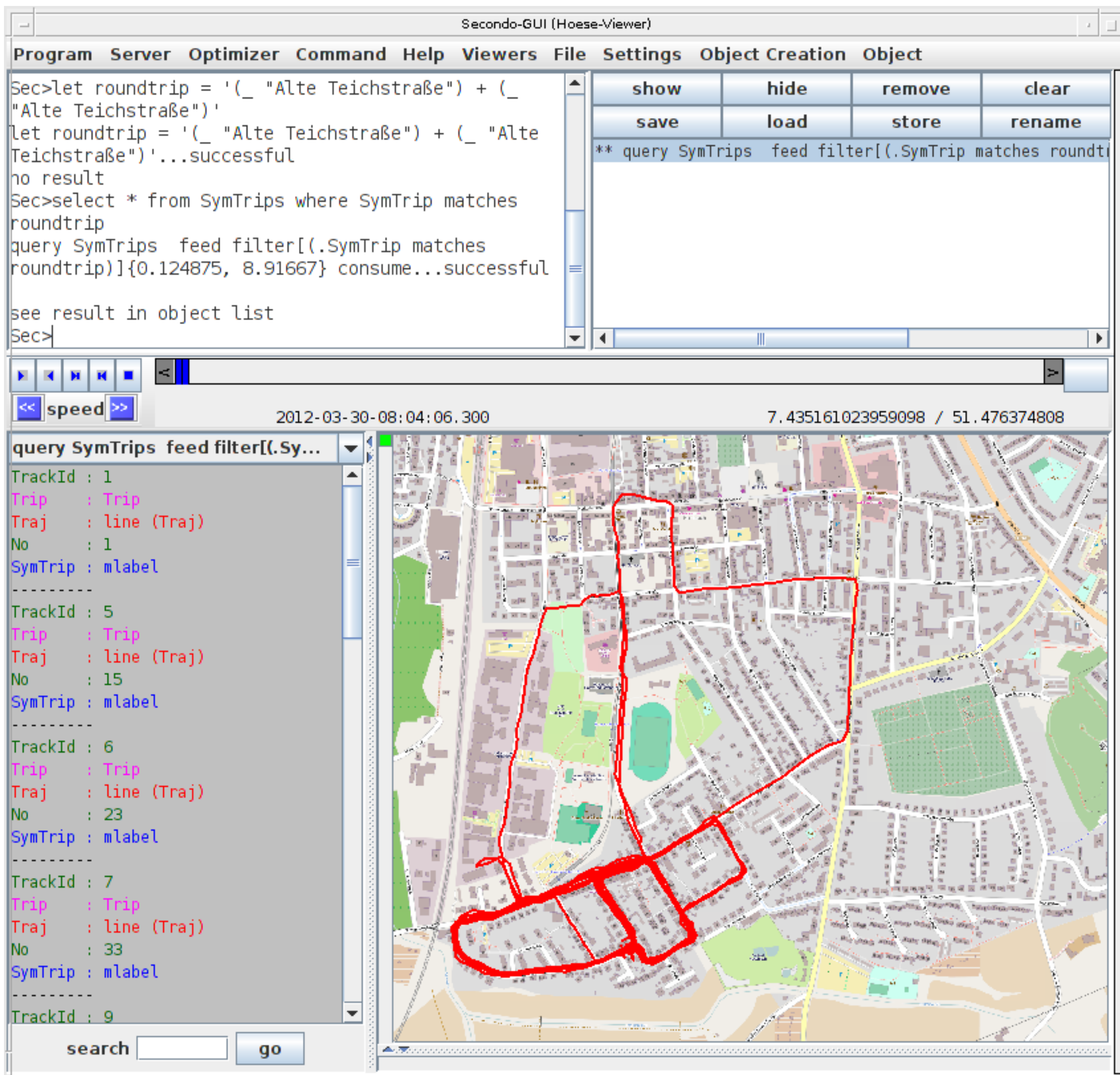


Figure 7: Round trips starting and ending at “Alte Teichstraße”

The result is shown in Figure 7. □

Example 5.5 Find trips from home to work, i.e., from Alte Teichstraße to Universitätsstraße.

```
let hometowork = '(_ "Alte Teichstraße") + (_ "Universitätsstraße")' ;
select * from symtrips where symtrip matches hometowork
```

□

Example 5.6 Find trips from home to work that use the highway “Sauerlandlinie”.

```
let hometowork2 =
  '(_ "Alte Teichstraße") * (_ "Sauerlandlinie") * (_ "Universitätsstraße")' ;
select * from symtrips where symtrip matches hometowork2
```

□

The results of the last two queries are shown in Figure 8. The first query has retrieved both routes shown in blue and yellow. The second returns only the blue routes (on top of the yellow of the first query).

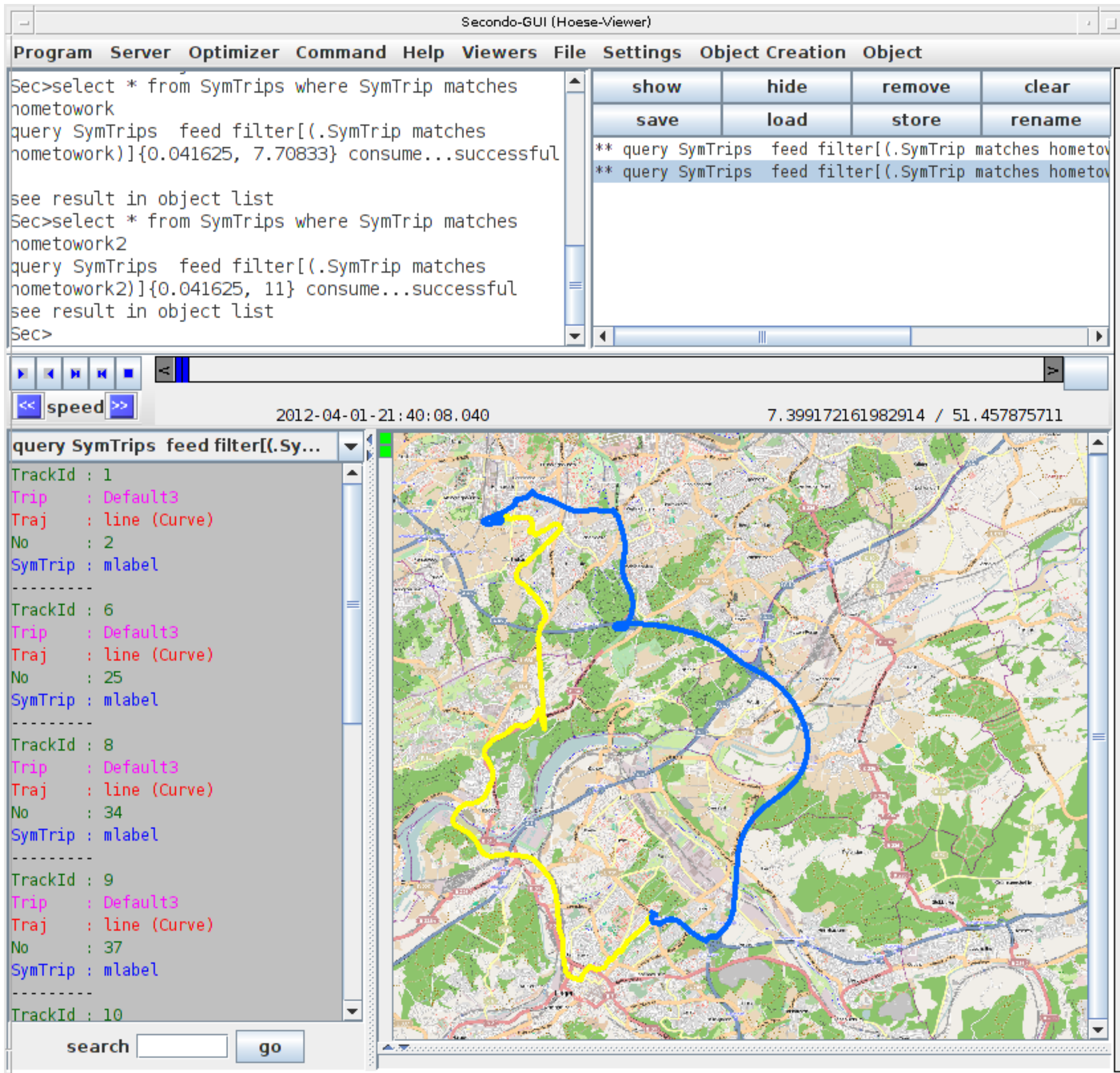


Figure 8: Trips from home to work

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 data structures and 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. However, the first subsection of this chapter considers the special parsing process and the applied data structures.

For a better understanding of the proposed concepts, we adopt the symbolic trajectory from Example 4.2 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, all of them 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 containing the temporal information of the trip before passing `Queen Anne Street` or `Welbeck St` (if existing) and the label `start of trip`. As we will show later, there are three different result trajectories.

The respective unit items are addressed by m_i (the ith unit of M_0, starting from 0), p_i (the ith atomic pattern element of P_0 and P_1, starting from 0), while the sizes of M_0 (number of units) and P_0 (number of atomic pattern elements) are denoted by $|M_0|$ and $|P_0|$, respectively.

6.1 Data Structures

In the following, we discuss the most important classes of the implementation. Most of the data are stored during the parsing process that is conducted at the beginning of each operator's execution. Further structures will be discussed when they are used.

Class Atomic Pattern Element For every atomic pattern element, the input parser stores a variable (type string), a set of time intervals (strings), a set of labels (strings) and a wildcard (either *no*, *star*, or *plus*). Processing the atomic pattern element p_1, an object holding a variable `Y`, a set of two temporal strings `thursday` and `morning`, a set of labels containing only `Queen Anne St`, and the wildcard value *no* is created. Note that the instance containing p_2 is also assigned the variable `Y`.

Class Condition Each condition object holds its original text (string) and a vector of variables (string) and attribute codes (integer; cf. Definition 4.6). For the condition from P_0, we record the input `(Y.end - X.start) < 20 * minute`, the variables `Y` and `X` and the types *end* and *start* (encoded as 3 and 2, respectively). Further data being kept for evaluating the condition will be detailed later.

For a more efficient matching procedure, we distinguish between regular and easy conditions. In contrast to the former, which are evaluated in a separate process requiring complex preparation after the basic pattern matching, the latter are checked during the matching itself. If a condition contains only one variable, and if this variable refers to a unit pattern, the condition is easy. Note that this class is applied for easy conditions as well as for general conditions.

Class Assignment For each variable in the results section, an assignment instance is created. It contains the variable (string) and its position in the results (0 for `A` and 1 for `Y` in P_1) and a boolean being true if and only if the variable occurs in the pattern elements (*false* for `A`, *true* for `Y` in P_1). In addition, the assignment commands (one for each attribute) are stored in an array of strings, along with the corresponding variables and attributes of the right assignment side which are kept in an array of suitable vectors. We record the assignments in P_1 in the string array's *time* and *label* slot, respectively, and push the information from `X.time` into the time vector and the string `start of trip` into the label vector of `A`. This class also records evaluation data.

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

Class Pattern The only instance of this class holds the outcome of the parsing process. More precisely, it contains a vector of atomic pattern elements, one vector of conditions and one of easy conditions, and a vector of assignments. Considering the input P_1, the atomic pattern element vector contains four elements, the condition vector contains one element, the easy condition vector remains empty, and the assignment vector holds two elements.

Moreover, the transition function δ of a nondeterministic finite automaton (NFA), forming the basis of the matching process and being modeled as a vector of mappings from integer to integer, is kept here. A mapping element $i \mapsto s_k$ at vector position s_j represents a transition from state s_j to state s_k, requiring a unit matching the atomic pattern element p_i. In the following, a transition t is denoted as $s_j \xrightarrow{i} s_k$, where s_j, i, and s_k are referred to as $t.source$, $t.atom$, and $t.target$, respectively. Moreover, $t.elem$ denotes the number of the pattern element that the atomic pattern element $t.atom$ belongs to. Finally, $\delta[s]$ is the set of transitions going out from the state s. Regarding P_0 or P_1, the set $\delta[0]$ equals $\{0 \xrightarrow{0} 0, 0 \xrightarrow{1} 1, 0 \xrightarrow{2} 1\}$, while $\delta[2]$ is empty. The complete transition function for P_0 is presented in Figure 9 (right).

Along with the NFA, this class holds a set of integers to represent the final states.

Class Match This class also has a unique instance keeping a pointer to a pattern and another to a moving label. In order to compute the bindings of units to variables, which is required for the condition evaluation and the rewriting of a trajectory, a two-dimensional array of integer sets is used. The latter is detailed in 6.2.3.

Class Classifier Since the operator **classify** processes arbitrarily many patterns, the corresponding NFAs have to be stored together for a simultaneous matching. For reasons of efficiency, we combine all NFAs to a single one by appending the vectors and translating the mappings' targets.

6.2 Algorithms

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

6.2.1 Parsing

The translation of the input string into instances and attributes of the abovementioned classes 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 atomic pattern elements, conditions, and assignments are stored as described in Section 6.1. During this process, we create a new string `regEx` containing only those regular expression parts and an integer for each of the atomic pattern elements, 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 [2, p. 159] to convert a regular expression to an NFA, resulting in the NFA depicted in Figure 9 (left).

The computation cost of the parsing phase is linear in the number of atomic pattern elements, thus it hardly affects the overall runtime.

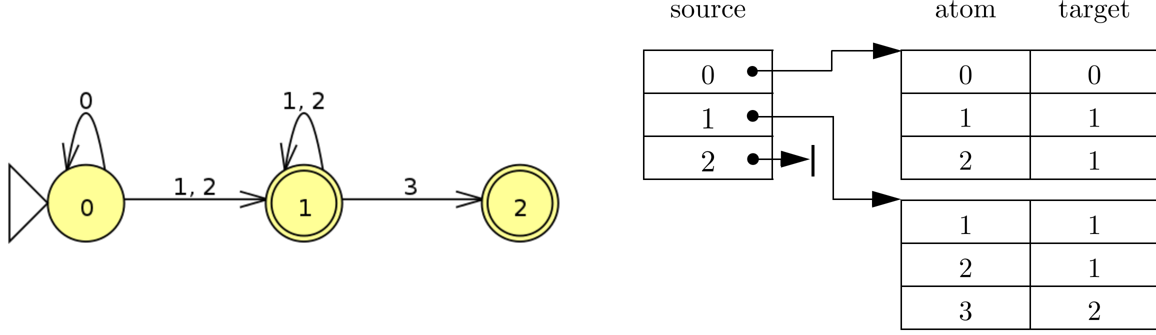


Figure 9: The NFA after parsing P_0 in graphical form (left) and as a vector of mappings (right)

6.2.2 Matching Without Conditions

In the following, we detail how the NFA transitions are applied for the matching process, in case there are no conditions.

Algorithm 1: *matchesWithoutCondition*

Input: P – a pattern with p atomic pattern elements;
including an NFA δ with n states and a set of final states F;
M – an mlabel of size m.

Output: *true*, if a final state of the NFA is active after processing the mlabel; *false* otherwise.

1 let $S = \{0\}$;
2 **for** $i = 0$ **to** $m - 1$ **do** // loop over trajectory
3   $T = \emptyset$;
4   **foreach** $s \in S$ **do** $T = T \cup \delta[s]$; // collect possible transitions
5   **if** $T = \emptyset$ **then return** *false*;
6   $S = \emptyset$;
7   **foreach** $t \in T$ **do** // loop over possible transitions
8   **if** $match(m_i, p_{t.atom})$ **then** $S = S \cup t.target$;
9 **return** $(S \cap F \neq \emptyset)$;

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 from each element of 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 performing those transitions from T whose corresponding atomic pattern element matches the current unit m_i. More exactly, the function *match* on the one hand compares the user-specified information from the atomic pattern element to the unit (cf. Definition 4.2), and on the other hand checks whether the easy conditions corresponding to the atomic pattern element 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. In the following, we denote $\delta[s]$ as δ_s.

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 atomic pattern elements 0 and 1, the states 0 and 1 become active, i.e., $S = \{0, 1\}$.

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, 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 there is a final state which is active after the loop over the symbolic trajectory 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 atomic pattern elements 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 atomic pattern element. 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.

6.2.3 Matching with Conditions

If processing a pattern with conditions, the latter have to be evaluated if and only if the sequence of pattern elements 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 4.4). For the conditions to be verified, we have to find one binding which 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.

Recording the Matching History Before the outer loop starts, the two-dimensional array of integer sets A from the Match class, 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 4.3). Now consider line 8 of Algorithm 1. In addition to the command after **then**, as long as $i < m - 1$ holds, we collect all feasible transitions T' from $t.target$, where $t \in T$, and insert $t'.elem$, the number of the pattern element containing the atomic pattern element $t'.atom$, where $t' \in T'$, into the set $A_{i,t.elem}$, i.e.,

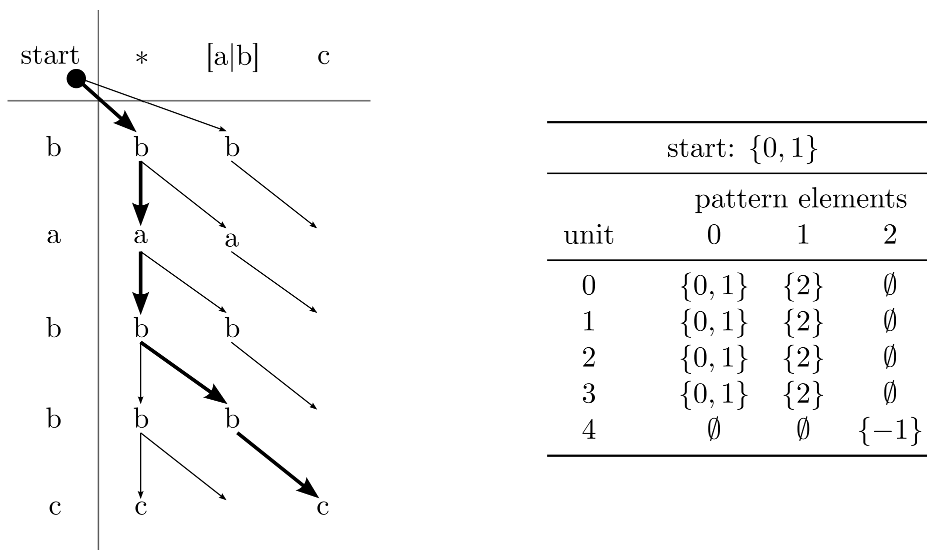
$$\textbf{if } (i < m-1) \textbf{ then } A_{i,t.elem} = A_{i,t.elem} \cup \{t'.elem | t' \in T', t'.source = t.target, t \in T\}.$$

In other words, all possible successive pattern element numbers are collected for each atomic pattern element matching a unit, so the elements of such a set represent pointers to matching candidates for the next unit. During the final iteration, i.e., 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 lower case letter and an atomic pattern element is either such a single letter or a `+` or a `*`, the two latter 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 10 from top to bottom. Since the first

atomic pattern element 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 10, 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 stands for 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.



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	$\emptyset$	$\emptyset$	$\{-1\}$

Figure 10: The recorded matching history for $M =$ `babbc` and $P =$ `* [a|b] c` as a graph (left) and as a table (right)

Now we return to the continuous example M_0 and P_0, for which in Table 1 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 1 now is a combination of the representations of the left and right parts of Figure 10.

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

Computation of Bindings A binding is implemented as a mapping from a string to a pair of integers. 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

Table 1: The recorded 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 $\{0,1\}$	QA $\{1,2\}$	$\emptyset$
Wim	1	Wim $\{0,1\}$	$\emptyset$	$\emptyset$
WelW	2	WelW $\{0,1\}$	$\emptyset$	$\emptyset$
WelSt	3	WelSt $\{0,1\}$	WelSt $\{1,2\}$	$\emptyset$
QA	4	QA $\emptyset$	QA $\{-1\}$	QA $\{-1\}$

suitable binding is found, and the condition evaluation is completely processed, *true* is returned. Details concerning the evaluation of conditions are discussed below.

Algorithm 3 starts at a certain position in the two-dimensional array A and recursively tries to find a path through A which 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 abovementioned binding modifications are undone if no success is reported (lines 13-15).

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 a binding is found that fulfills every condition; *false* otherwise.

1 **foreach** $j \in \{t.elem | t \in \delta_0\}$ **do**
2   **if** *bindingExists*$(P, 0, j, B)$ **then return** *true*; // abort if successful
3 **return** *false*;

Subsequently, we apply this procedure to our continuous example. In Table 2, 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 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 2. The next step consists in checking whether this binding fulfills every condition.

As the results of the condition evaluation are unpredictable, we have to determine the compu-

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 atomic pattern element 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++;               // extend existing binding
5      else                                             // add new variable to binding
6          B(v) = (i, i);
7          inserted = true;
8  if A_{i,j} contains −1 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--;
16 return false;
```

tation 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 the next paragraph. The worst case arises given a maximal number of transitions from each state (e.g., this holds for a pattern of the form `* * ... *`) 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 $O(m^{e-1})$.

Evaluation of Conditions As mentioned in Subsection 6.1, each condition object contains data supporting the evaluation. More precisely, during the parsing process, for every condition we create a SECONDO operator tree including 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 <u>*string*</u>, <u>*periods*</u>, <u>*instant*</u>, <u>*bool*</u>, <u>*int*</u>, and <u>*labels*</u>. Consequently, for the condition `Y.end - X.start < 20 * minute` from P_0, two pointers to <u>*instant*</u> 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 <u>*bool*</u>.

In the following, we detail the function *conditionsMatch* which is invoked in line 9 of Algo-

Table 2: Execution of algorithm 3 for P_0 and M_0

i	j	B at call	B updated	final	proceed	undo B
0	1	$\emptyset$	$\{\mathtt{Y} \mapsto [0,0]\}$	no	yes	no
1	2	$\{\mathtt{Y} \mapsto [0,0]\}$	$\{\mathtt{Y} \mapsto [0,0], \mathtt{Z} \mapsto [1,1]\}$	no	no	yes
1	1	$\{\mathtt{Y} \mapsto [0,0]\}$	$\{\mathtt{Y} \mapsto [0,1]\}$	no	no	yes
0	0	$\emptyset$	$\{\mathtt{X} \mapsto [0,0]\}$	no	yes	no
1	1	$\{\mathtt{X} \mapsto [0,0]\}$	$\{\mathtt{X} \mapsto [0,0], \mathtt{Y} \mapsto [1,1]\}$	no	no	yes
1	0	$\{\mathtt{X} \mapsto [0,0]\}$	$\{\mathtt{X} \mapsto [0,1]\}$	no	yes	no
2	1	$\{\mathtt{X} \mapsto [0,1]\}$	$\{\mathtt{X} \mapsto [0,1], \mathtt{Y} \mapsto [2,2]\}$	no	no	yes
2	0	$\{\mathtt{X} \mapsto [0,1]\}$	$\{\mathtt{X} \mapsto [0,2]\}$	no	yes	no
3	1	$\{\mathtt{X} \mapsto [0,2]\}$	$\{\mathtt{X} \mapsto [0,2], \mathtt{Y} \mapsto [3,3]\}$	no	yes	no
4	2	$\{\mathtt{X} \mapsto [0,2], \mathtt{Y} \mapsto [3,3]\}$	$\{\mathtt{X} \mapsto [0,2], \mathtt{Y} \mapsto [3,3], \mathtt{Z} \mapsto [4,4]\}$	yes	no	no

rithm 3. It loops over the conditions, returning *false* in case of a negative evaluation result, and *true* if all conditions are fulfilled. Before a single 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 4.6), the binding B combined with the symbolic trajectory M provide 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 $\{\mathtt{X} \mapsto [0,2], \mathtt{Y} \mapsto [3,3], \mathtt{Z} \mapsto [4,4]\}$. The <u>*instant*</u> pointers' targets are set to these values, and Secondo can execute the condition as a query, returning a result of type <u>*bool*</u>. In our case, the result is *true*, since the difference of the two instants is less than 20 minutes, 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 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.

6.2.4 Requirements for a Linear Runtime of matches

After the execution of Algorithm 2, the operator **matches** terminates after 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 non-trivial pattern specifications, consider the formula's non-linear 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. At the same time, no *time* or *labels* attribute may be used along with a sequence variable in any condition, for the sake of a constant value assignment cost.

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 6.2.2, the computation cost of **matches** is always linear in m for a pattern without conditions.

6.2.5 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 (cf. Subsection 6.1). Hence, discovering one binding which 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, there is an outer 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. This is done similarly to 6.2.3, 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 which fulfills the condition is found like in Table 2, while the position data i and j are stored on a stack. For every backtracking action, i.e., when i does not increase from one line to the next, the current binding is reset according to Algorithm 3, lines 13-15. The binding $\{\texttt{X} \mapsto [0,2], \texttt{Y} \mapsto [3,3], \texttt{Z} \mapsto [4,4]\}$ 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 2, backtracking only one level, i.e., $i = 3$ and $j = 2$, and choosing the element 1 leads to the next binding $\{\texttt{X} \mapsto [0,2], \texttt{Y} \mapsto [3,4]\}$. 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`. Finally, 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") )
```

is the consequence of the binding $\{\texttt{X} \mapsto [0,3], \texttt{Y} \mapsto [4,4]\}$, obtained by backtracking until $i = 2$, $j = 0$.

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.

6.2.6 Matching a Stream of Symbolic Trajectories

While the operator **matches** processes exactly one trajectory, for several applications it is suitable to compare k symbolic trajectories $M_0, \ldots, M_{k-1}$ ($k \in \mathbb{N}$) to a single pattern P in an efficient way. The straightforward approach, i.e., executing **matches** repeatedly on a text pattern, is less-than-ideal since P has to be parsed k times. It is better to first convert the pattern text into a value of type <u>*pattern*</u> and then to use **matches** with this argument, as described in Section 4.6.1.

An alternative solution is provided by the operator **filtermatches** which takes a stream of tuples containing symbolic trajectories and a pattern text and returns a stream of the tuples with matching symbolic trajectories. At the beginning, an NFA is produced from the pattern text. Subsequently, exactly those trajectories passing the matching process are copied to the output stream. This operator can be used by the query optimizer to translate a **matches** predicate on a text argument.

For example, let $M_0, \ldots, M_{k-1}$ ($k \in \mathbb{N}$) be the trajectories (geographic as well as symbolic) of a person during a long period. In order to display only those trajectories belonging to a short round trip (less than 20 minutes, starting and ending at a certain location) on a map, **filtermatches** can be applied [37].

6.2.7 Classification of a Symbolic Trajectory

The purpose of the operator **classify** is to distribute a set of symbolic trajectories into not necessarily disjoint subsets, so-called categories. Along with the trajectory collection, a set of patterns (with or without conditions) have to be specified, where each pattern must be annotated with a category description.

First, the operator reads in 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 of the patterns. Let $n_0, n_1, \ldots, n_{l-1}$ be the number of states for each of the l patterns. Hence, the multi-automaton 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.

6.2.8 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 the following, we introduce the concept of a trajectory index. Since a symbolic trajectory M is ordered by the time intervals of its units, an additional structure is necessary in order to conduct operations like deciding whether a certain label occurs in M and/or finding a certain label inside M in constant time. Due to the necessity to handle a set of symbolic trajectories, a trajectory index has to contain information not only about one trajectory but about arbitrarily many.

Hence, we implemented the operator **createtrie** which processes a relation containing an attribute of type <u>*mlabel*</u>, storing the tuple identifier(s) and unit position(s) of each label 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.

Now we briefly present the use of such an index for matching a pattern against a set of symbolic trajectories. The operator **indexmatches**, which computes the same result as **filtermatches**, is passed a relation R containing an *mlabel* attribute, the name of that attribute, the name of the index object, and a pattern object (or text).

If the pattern has conditions or regular expression items, the index is currently not applied, and the symbolic trajectories matching this pattern are determined as in 6.2.6, considering each (tuple of the relation containing a) trajectory in turn. Otherwise, with the help of an index, it suffices to follow NFA transitions until a final state is active and to track which trajectories are active at which positions. If a label is found in a unit pattern, it is looked up in the index – which happens in constant time, assuming labels have constant length –, and only a few units in a few trajectories remain active. Under convenient circumstances, i.e., if the pattern contains only wildcards and unit patterns with label specifications, the runtime complexity of **indexmatches** is in $O(t \cdot |R|)$, where t is the number of possible transitions of the NFA belonging to the pattern.

An additional operator named **indexclassify** calculates the same result as **classify** with the help of a trajectory index. Similarly to the previous paragraph, its runtime amounts to $O(t' \cdot |R|)$, where t' is the number of possible transitions of the multi-automaton created from the pattern set.

A trajectory index that contains time information and is applicable for general patterns is a subject of future work.

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 3.3 GHz processor running openSUSE 11.4, with 8 GBytes of main memory. In the first part, we present runtime graphs of the operators **matches**, **rewrite**, **filtermatches**, **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 [14], a benchmark for spatio-temporal database management sytems.

All runtimes were computed by running each query four times and taking the median value of the durations. The executed queries (patterns) as well as the precise elapsed times are listed in Appendix B.

7.1 Experiments with a Synthetic Dataset

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 subsection represent trips through the city of Dortmund, Germany, in randomized but valid order, more exactly, the labels correspond to the names of the 12 main districts, and the labels of two consecutive units are adjacent to each other on the map. As the time intervals are irrelevant for the runtime, each unit has a duration of half an hour, and each symbolic trajectory starts at `2012-01-01-00:00:00`. A relation containing arbitrarily many symbolic trajectories, each of which is a random trip through Dortmund of a user-defined size, can be produced by the operator **createmlrelation**.

7.1.1 matches

In Figure 11, we present the performance evaluation of the operator **matches** with respect to an increasing symbolic trajectory and different patterns. The left diagram refers to patterns without conditions, while each of the patterns on the right hand side contains at least one condition.

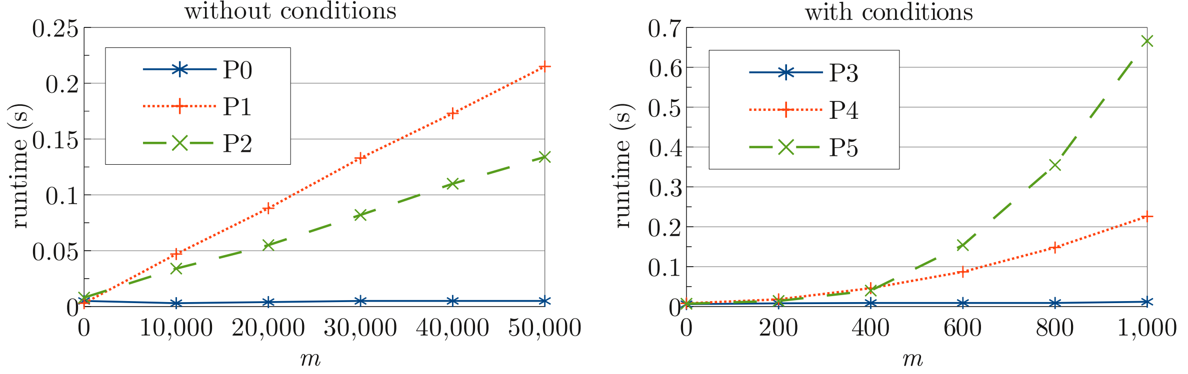


Figure 11: Runtime of the operator **matches** with (left) and without (right) conditions

The left plot visualizes 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 pattern P0 causes an early mismatch (after the second unit), its computation cost is constant. For P1 and P2, we observe a difference in the slopes, caused by the higher number of states that are active in every iteration of the matching process.

The patterns P4 and P5 on the right-hand diagram confirm the worst case runtime complexity of algorithm 2. Since the condition of P5 is never fulfilled, every possible binding has to be computed and evaluated, and the runtime is quadratic in m since P5 contains three wildcards. Due to the same reason, also P4 causes quadratic computation cost. However, the slope of the corresponding curve is below that of P5, since only a few bindings have to be computed and checked until a true configuration is found. For an easy condition, no two-dimensional array and no binding is required, thus the runtime of P3 is linear in m. Note that the trajectories applied on the right hand side are considerably shorter than on the left, otherwise the runtime graphs for P3 and P4 would have been hardly distinguishable.

7.1.2 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.

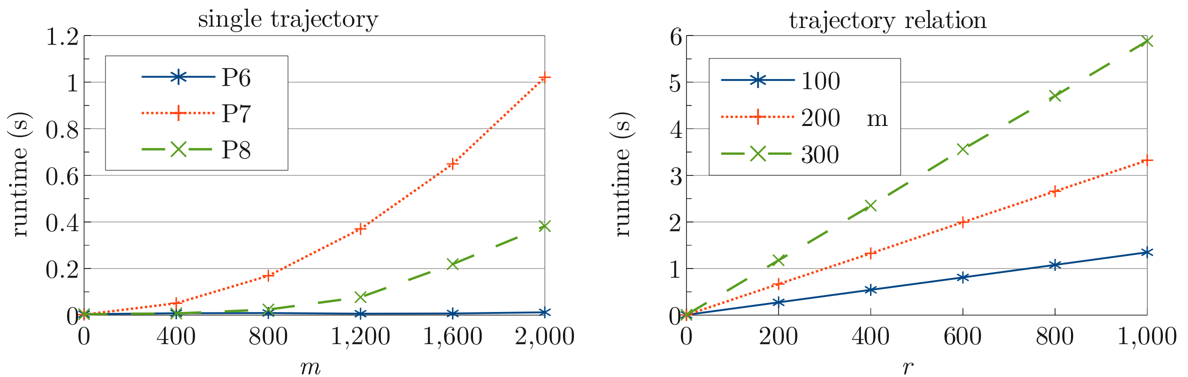


Figure 12: Runtime of the operator **rewrite** for a single trajectory (left) and a trajectory relation (right, conducted with pattern P9)

As expected, the graph resulting from the pattern P8 in the left hand diagram of Figure 12

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 (that is, $m + 1$ 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 a *mlabel* attribute. We varied the number of tuples r 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 filtermatches and indexmatches

We proceed to a series of experiments matching a single pattern against a stream or relation of symbolic trajectories. In the following, the benefit of applying a trajectory index is analyzed in contrast to the execution without index support.

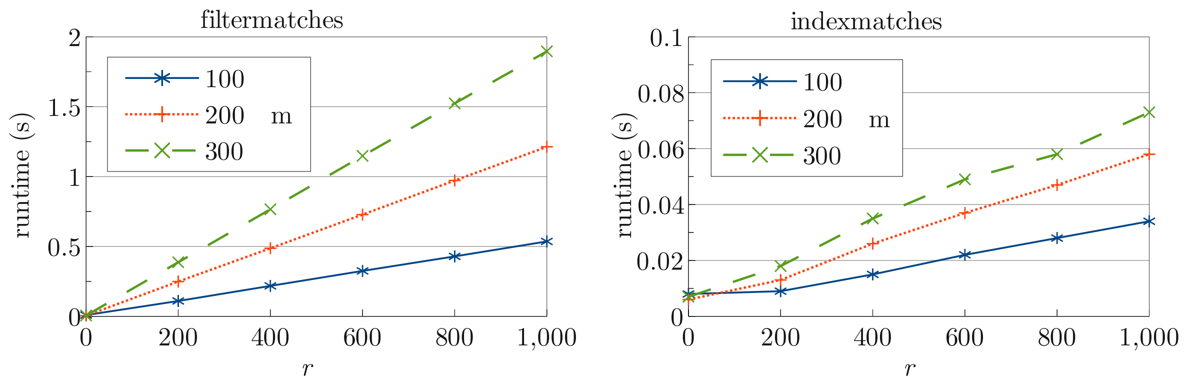


Figure 13: Runtime of the operators **filtermatches** and **indexmatches**

The linear shape of the functions depicted in the left plot of Figure 13 is as expected. Theoretically, the runtime of the operator **filtermatches** should also be proportional to m, which is not exactly the case. The slight overhead is due to internal SECONDO memory administration reasons, thus not related to the implementation.

In the right diagram, we present the massive influence of a trajectory relation index, reducing the runtimes by a factor of up to 26. The computation cost is still (almost) linear in r, since the operator **indexmatches** has to administrate the active units for each of the r trajectories. However, the runtime is less than proportional to m, since the trajectories are not completely processed.

7.1.4 classify and indexclassify

The final test, whose results are depicted in Figure 14, 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 three simple patterns.

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,

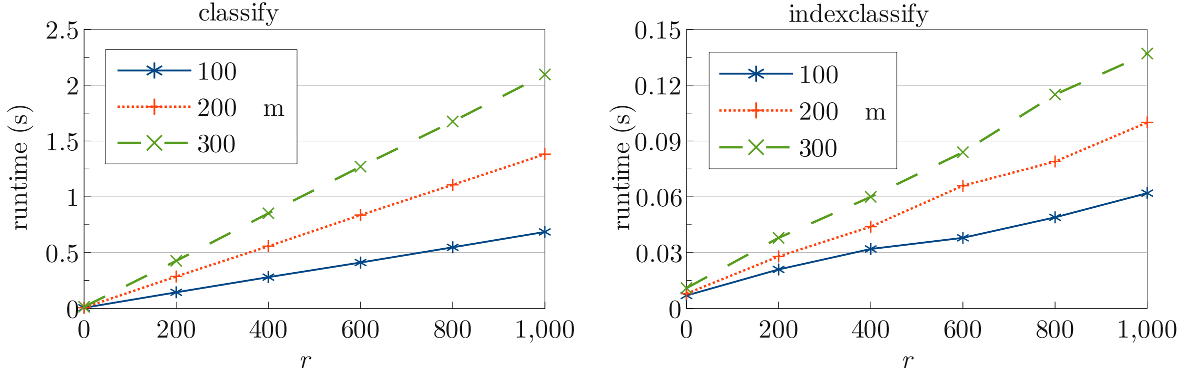


Figure 14: Runtime of the operator `classify`

since only Algorithm 1 is executed. Hence, the construction of the multi-automaton, 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, i.e., by a factor of approximately 15. Similar to the situation in 7.1.3, the computation cost 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 a *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 from the dataset and conducted the experiments with the remaining 145,000 trips.

In the following, we present six different approaches of computing which of the trajectories passed through certain streets. The first uses classical techniques, the remaining five are based on symbolic trajectories. More exactly, each of the subsequent methods is applied to compute how many of the trajectories passed

1. `Bundesallee`
2. `Bundesallee` and `Pariser Platz`
3. `Bundesallee`, `Pariser Platz`, and `Unter den Linden`

in the specified order. For each case, arbitrarily many streets may be passed before, between, and after the respective street names. The trajectories resulting from the third schedule are depicted in Figure 15.

7.2.1 Evaluation Using Raw Trajectories

Approach 1: Moving Points and a Geometric Index In order to solve the three tasks in a reasonably efficient way based on raw trajectories, a remarkable preparation effort was required. First, we created two B-trees over the streets relation for fast access to the routes themselves (datatype *line*) and to their bounding boxes. Subsequently, an R-tree containing the bounding boxes of the 56 million unit point segments was built, consuming more than 37

Figure 15: The streets of BerlinMOD (thin gray), the three specified streets (thick gray), and the filtered trajectories (black). Since `Pariser Platz` and `Unter den Linden` are adjacent, they appear as one street (center of the image).

minutes. Regarding the first task, we collected all trajectories having at least one unit whose bounding box intersects the bounding box of the street `Bundesallee`, which is done in less than 4 seconds applying the R-tree. The resulting trips had to be processed further, i.e., we checked whether they contain a unit whose start and end point are located in a small radius around the line corresponding to `Bundesallee`. This tolerance is necessary due to numerical issues in geometric algorithms and – for actually recorded data – to inaccurate GPS signals. After nearly 9 minutes, task 1 is solved.

For the second task, a temporal order had to be queried, so we modified the refinement function to not only decide whether a certain street is passed or not, but to return the time instant at which it is first visited. Another function was introduced deciding whether a moving point passes a certain street after a specified instant of time. With these tools, we were able to formulate a query providing the solution. A similar, even longer query was applied to solve the third task. Both were processed in less than one minute, which was clearly faster than for task 1, since the number of result candidates provided by the R-tree was notably smaller (1,000 vs. 16,000).

7.2.2 Evaluation Using Symbolic Trajectories

Construction of Symbolic Trajectories Since the conversion of an *mpoint* into an *mlabel* is a complex procedure, we decided to apply Parallel SECONDO [27], available from the Web site

[26], on a 12 node cluster for this task. After distributing the streets and trips data, we executed a query matching the center point of each unit point to the nearest street segment and creating a unit from the associated time interval and the street's name. The cluster accomplished this task in a little less than three hours.

The resulting 145,000 moving labels are clearly shorter than the raw trips, since the units of the latter have a duration of only two seconds, while in a symbolic trajectory, an additional unit is necessary only if the street name changes. More precisely, the memory consumption of the 145,000 moving labels amounts to 282 MBytes, whereas the associated moving points require 10.8 GBytes. A summary of the achieved runtimes is presented in Table 3.

Approach 2: A Linear Scan The first task was solved elegantly with the help of the **passes** operator in only three seconds. For a symbolic trajectory and a label, the predicate **passes** is true if and only if the label occurs at least once in the symbolic trajectory.

Similar to Approach 1, we created a function considering the temporal order for solving the tasks 2 and 3, consuming less than 5 seconds each. This approach is neither user-friendly nor expressive, since the complexity of the query itself (and of the auxiliary functions) significantly increases with any additional requirement.

Approach 3: Spatiotemporal Pattern Queries With the help of spatiotemporal pattern queries [33], available in the SECONDO distribution, we conducted the desired filter steps. For the second and the third task, the constraints were defined applying the operator **inside**. As spatiotemporal patterns require as least two *mbool* predicates, the first task cannot be solved with this technique in a sensible way.

The time consumption of the respective queries ranged from 8 to 9 seconds.

Approach 4: Working with an Inverted File For a faster processing, we created an inverted file that returns the tuple id and the unit position for each occurrence of a street name. This step was completed after 75 seconds. In order to solve the first task, we merely had to look for `Bundesallee` inside the inverted file and to count the different tuple ids. For the temporal order in the two other tasks, a hashjoin and a comparison of unit positions was necessary. The efficiency of this approach (runtimes clearly below 1 second) makes it suitable for filtering large trajectory collections before using more sophisticated techniques.

Approach 5: Pattern Matching In this approach, the operator **filtermatches** (see 6.2.6 for details) was applied. Without any index support, the operator processes all trajectories in 6.6 seconds. As described before, the influence of the number of atomic pattern elements is vanishingly low compared to that of the number of processed units.

Approach 6: Pattern Matching with a Trajectory Index Finally, a trajectory index (cf. 6.2.8) was created for the collection of symbolic trajectories. This operation took slightly more than 3 minutes, leading to a further reduction of the runtime by a factor of approx. 6. This benefit is inferior compared to the results of 7.1.3, since the number of trajectories causes a greater proportion of the computation cost of **indexmatches** than their length.

7.2.3 Summary of BerlinMOD Experiments

Considering Table 3, it is obvious that using symbolic instead of raw trajectories offers substantial advantages regarding the execution of queries as well as the disk space occupation and the creation of indexes. For all these quantities, gains in efficiency of several orders of magnitude can be observed.

Table 3: Result overview after filtering 145,000 BerlinMOD trajectories

Applied Approach	Runtime (seconds) consumed for Index	Task 1	Task 2	Task 3
Based on raw geometric data (occupying 10.8 GBytes)				
1: moving points, geometric index	2249.7	521.5	53.5	58.4
Based on symbolic trajectories (occupying 282 MBytes)				
2: linear scan		3.0	4.5	4.6
3: spatiotemporal pattern queries		8.1	8.0	9.2
4: inverted file	74.8	0.1	0.2	0.5
5: pattern matching		6.6	6.6	6.6
6: pattern matching, trajectory index	185.7	1.1	1.1	1.1

Among the methods based on symbolic trajectories one should compare the techniques without index (2, 3, 5) and the index-based methods (4, 6) separately. In the first case, one can see that the powerful pattern matching engine achieves similar efficiency as the simple linear scan or the more complex method of spatio-temporal pattern queries. For the index-based methods, the inverted file technique is a bit faster than using the trajectory index. At the same time, one can observe that query formulation is much more difficult and lengthy with the other techniques (see the queries in Appendix B.2) and simple and elegant with the pattern language. Moreover, the comparison captures only very simple queries that can be handled by the other techniques whereas the pattern language is vastly more powerful, enabling the use of regular expressions, temporal constraints, and expressive conditions, all very inconvenient or even impossible to express with basic techniques.

8 Related Work

The notion of symbolic trajectory relates to diverse research areas. In what follows we tie in our work with major research streams focusing, in particular, on pattern matching of sequences and semantic trajectories.

Pattern matching of sequences. To a first approximation, symbolic trajectories can be seen as sequences of symbols, i.e. strings. Pattern matching over strings is a well-known problem in the literature [20]. Given an alphabet Γ, a pattern is, in the simplest case, a finite string defined over Γ, otherwise it is a regular expression denoting a set of strings of potentially infinite length [29]. Regular expressions are extremely popular in e.g. programming languages and operating systems. For example, the syntax that we have chosen to represent our patterns has been loosely inspired by the Mathematica programming language.[6] The algorithms searching for text matching regular expressions typically employ finite automata, i.e. NFA or DFA, and a have running time linear in the size of the text [29]; more efficient solutions can be obtained by using an indexing mechanism over text [5]. These techniques are also employed in our pattern matching engine. Yet there is a significant difference between the notion of symbolic trajectory and that of string. In particular symbolic trajectories have a dual dimension, i.e. temporal and

[6]http://blog.wolfram.com/2008/11/18/surprise-mathematica-70-released-today/

textual where the temporal data not only serves to force ordering in the sequence as in time series, but complements the textual information. In addition the symbolic trajectory data model is embedded into a database system. System design is thus integral part of the solution.

In the database literature, regular expressions are key for querying event streams [32, 11, 1]. An event is an occurrence of interest at a point in time, that has a type and a set of attribute values; an event stream is a temporally ordered sequence of events that can be acquired one at a time or being archived. Query languages over event streams include for example SQL-TS [32] and DejaVu [11]. Despite their syntactic variations, these languages share many features for pattern matching that we can find in our language such as the use of variables, conditions and the Kleene operator. Events, however, have a different representation and purpose, while the query languages do not support full-fledged regular expressions.

Closely related to our work is the research on mobility pattern matching in spatio-temporal databases. The basis for the present work originates from [13, 12]. In particular, du Mouza and Rigaux introduce the notion of mobility pattern over trajectories defined in a discrete space (i.e. the space is not dense) where each region is identified by a symbol. An object's trajectory is defined by the sequence of symbols denoting the successive zones crossed by the object. The trajectories obtained in this way are interrogated using a pattern matching language supporting variables. Moreover the pattern matching engine relies on an NFA. The expressivity 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 on variables.

The assumption of a discrete reference space is also at the basis of the work by Vieira et al. [38, 39]. The idea is again to support pattern matching over trajectories, but in this case trajectories are geometric and not symbolic, i.e. a sample query is to find the (geometric) trajectories that go from region A to region B. The pattern language is rich and includes not only symbols and variables but also conditions, such as spatial and temporal conditions. This approach however makes strong assumptions on the underlying reference space, i.e. the space is partitioned, which seems quite restrictive in practice.

A different and recent line of research regards the querying of hybrid data, i.e. textual and spatial, where the text is for example a keyword describing a point of interest shared in a geo-social network, e.g. a place check-in. Current research is however focused on the efficient processing of conventional queries over textual-spatial objects such as nearest neighbor queries [7, 9], while trajectories are currently overlooked.

Symbolic trajectories are not restricted to movement in space but can employ any set of labels. Moreover the mobility patterns are expressed in terms of regular expressions with variables denoting symbols, time intervals and subsequences, whilst the pattern language is embedded into a database which offers a rich and extensible repertoire of data types and operations 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 a set of trajectories through multi-pattern 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.

Semantic trajectories. The second stream of related research focuses on the semantic enrichment of trajectories. Increasingly, advanced applications need to capture and annotate the meaning of the movement and not simply its evolution in space and time. This calls for enhanced representation models. For example, in diverse applications collecting movement data for semantic location recognition and recommendation purposes e.g. [25, 45, 8, 44, 6], trajectories are annotated with the name of the places in which the moving objects stay possibly for a significant amount of time. Another form of annotation regards the transportation modes, for example walking, bus, and alike, as in [23, 43]. Sequences of generic activities are represented

in e.g. [40, 42]. In general, these annotations can be obtained in different ways, i.e. through analytical techniques, or be specified by the user or be directly acquired from e.g. sensors as in [21, 22]. Annotations can also be used in place of the geometric data to obtain compressed representations of e.g. GPS trajectories as in [30].

The definition of a conceptual trajectory model is the first step in the direction of a general framework for enhanced trajectories representation. The early model proposed by Spaccapietra et al. [35] is centered on the key concepts of stop and move. This conceptualization is at the basis of the numerous trajectory data mining techniques following the early work in [3](see [36]). Generalized versions of this first model are presented in [4, 41, 36]. All of these works rely on some notion of semantic trajectory. For example in [36], a semantic trajectory is defined as a raw trajectory enhanced with annotations and episodes. Annotations are descriptions attached to the whole trajectory or parts of it, while an episode is a sub-trajectory resulting from a trajectory segmentation. For example stop and moves are two different types of episode. The definition of semantic trajectory is however not univocal. Moreover, the data model is only defined at a conceptual level. This means that the data management issues, i.e., how to interrogate large amounts of semantic trajectories, are ignored. This is exactly where symbolic trajectories fit into. The symbolic trajectory data model is defined at the logical level, formally defined and embedded in a moving object database to provide support for the efficient access possibly but not exclusively in a spatio-temporal context.

9 Conclusions

Capturing and representing the meaning of movement is a challenging issue that calls for novel solutions. This work presents a comprehensive framework for the generalized representation of movement in a symbolic space. Inspired by the concept of semantic trajectory, the work departs from the mere conceptualization to present a rigorous and rich data model embedded into a database system which significantly advances existing approaches.

We have defined the four data types *mlabel*, *mlabels*, *mplace*, and *mplaces*, but not explored the latter three further, focusing instead on the pattern matching and rewriting language. Future work will develop the possibilities of these types further such as "abstracting" moving labels based on classification hierarchies, using hierarchical structures for nested geometries (e.g., buildings) referred to by *mplaces*, efficiently managing repositories of geometries, and so forth.

Another interesting venue for research are more general indexes supporting pattern matching also for specifications of time intervals and patterns with conditions.

Symbolic trajectories have great application potential. As sequences of temporally annotated symbols, symbolic trajectories can be used in a variety of domains, not necessarily related to the geo-spatial context. For example, symbolic trajectories can be obtained from time series, e.g., health monitoring data, by applying techniques such as [24]. This opens up interesting opportunities for time series analysis. Discovering the full potential of the data model in non-spatial domains is a major challenge for future research.

As we have seen, however, symbolic trajectories are primarily motivated by the need of overcoming the limitations of the classic geometric trajectory data model in a geo-spatial context. We have also seen that the symbolic and the spatio-temporal dimensions of trajectories can coexist. For example, in SECONDO the movement of an entity can be simply described by two attributes one of type *mpoint* (i.e., the geometric trajectory) and one of type *mlabel* (i.e., the symbolic trajectory). In certain circumstances, however, a tighter integration of the two dimensions might be desirable to enable more efficient query processing and more powerful queries. Combining the symbolic and more in general the textual dimension with the spatio-temporal dimension paves the way to challenging research opportunities that, to our knowledge, have not been explored yet.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. of the SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, 2008.

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

[3] L. O. Alvares, V. Bogorny, B. Kuijpers, J. de Macedo, B. Moelans, and A. Vaisman. A model for enriching trajectories with semantic geographical information. In *Proc. ACM GIS*, page 22, 2007.

[4] G. Andrienko, N. Andrienko, and M. Heurich. An event-based conceptual model for context-aware movement analysis. *International Journal of Geographical Information Science*, 25(9):1347–1370, Sept. 2011.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[6] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and preference-aware recommendation using sparse geo-social networking data. In *Proc. of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 199–208, 2012.

[7] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *Proc. of the VLDB Endowment*, 6(1):1–12, Nov. 2012.

[8] X. Cao, G. Cong, and C. S. Jensen. Mining significant semantic locations from gps data. *Proc. of the VLDB Endowment*, 3(1-2):1009–1020, Sept. 2010.

[9] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *Proc. of the VLDB Endowment*, 6(3):217–228, Jan. 2013.

[10] *Creating an Ordered Relation Graph Representation from OpenStreetMap Import. Secondo Script.* `http://dna.fernuni-hagen.de/Secondo.html/OrderedRelationGraphFromFullOSMImport.SEC` (2013).

[11] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: Declarative pattern matching over live and archived streams of events. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1023–1026, 2009.

[12] C. du Mouza and P. Rigaux. Multi-scale classification of moving objects trajectories. In *Proc. of the 16th International Conference on Scientific and Statistical Database Management*, pages 307–316, 2004.

[13] C. du Mouza and P. Rigaux. Mobility patterns. *GeoInformatica*, 9(4):297–319, 2005.

[14] C. Düntgen, T. Behr, and R. H. Güting. Berlinmod: A benchmark for moving object databases. *VLDB Journal*, 18(6):1335–1368, 2009.

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

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

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

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

[19] R. H. Güting and M. Schneider. *Moving Objects Databases.* Morgan Kaufmann, 2005.

[20] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley-Longman, 2001.

[21] H. Hu and D. L. Lee. Semantic location modeling for location navigation in mobile environment. In *Mobile Data Management'04*, pages 52–61, 2004.

[22] C. Jensen, H. Lu, and B. Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD'09*, pages 208–227, 2009.

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

[24] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proc. of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 2–11, 2003.

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

[26] J. Lu. Parallel Secondo. `http://dna.fernuni-hagen.de/Secondo.html/ParallelSecondo/index.html`, 2013.

[27] J. Lu and R. H. Güting. Parallel secondo: Boosting database engines with hadoop. *International Conference on Parallel and Distributed Systems*, 0:738–743, 2012.

[28] F. Marchal, J. Hackney, and K. W. Axhausen. Efficient map matching of large global positioning system data sets: Tests on speed-monitoring experiment in Zürich. *Transportation Research Record: Journal of the Transportation Research Board*, 1935(1):93–100, 2005.

[29] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings.* Cambridge University Press, 2002.

[30] K.-F. Richter, F. Schmid, and P. Laube. Semantic trajectory compression: Representing urban movement in a nutshell. *Journal of Spatial Information Science*, 4(1):3–30, 2012.

[31] M. Roth. Map Matching von GPS-gestützten Positionsdaten im erweiterbaren Datenbanksystem SECONDO (Map matching of gps-based position data in the extensible DBMS SECONDO). Bachelor Thesis, Fakultät für Mathematik und Informatik, FernUniversität Hagen, 2012.

[32] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 71–81, 2001.

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

[34] N. Schüssler and K. W. Axhausen. Map-matching of gps traces on high-resolution navigation networks using the multiple hypothesis technique (mht). Arbeitsberichte Verkehrs- und Raumplanung, 568, IVT, ETH Zürich, 2009.

[35] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. de Macedo, F. Porto, and C. Vangenot. A conceptual view on trajectories. *Data & Knowledge Engineering*, 65:126–146, April 2008.

[36] 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. Semantic Trajectories Modeling and Analysis. *ACM Computing Surveys*, 45(4):42:1–42:32, 2013.

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

[38] M. R. Vieira, P. Bakalov, and V. J. Tsotras. Querying trajectories using flexible patterns. In *Proc. of the Conference on EDBT*, pages 406–417, 2010.

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

[40] Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, and K. Aberer. Semitri: A framework for semantic annotation of heterogeneous trajectories. In *International Conference on Extending Database Technology, EDBT'11*, pages 259–270, 2011.

[41] Z. Yan, D. Chakraborty, C. Parent, S. Spaccapietra, and K. Aberer. Semantic trajectories: Mobility data computation and annotation. *ACM Transactions on Intelligent Systems and Technology*, 4(3):49:1–49:38, July 2013.

[42] K. Zheng, Y. Yang, S. Shang, and N. J. Yuan. Towards efficient search for activity trajectories. In *Proc. of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 230–241, 2013.

[43] Y. Zheng, L. Liu, L. Wang, and X. Xie. Learning transportation mode from raw gps data for geographic applications on the web. In *Proc. of the 17th International Conference on World Wide Web*, WWW '08, pages 247–256, 2008.

[44] Y. Zheng and X. Xie. Learning travel recommendations from user-generated gps traces. *ACM Transactions on Intelligent Systems and Technology*, 2(1):2:1–2:29, Jan. 2011.

[45] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *Proc. of the 18th International Conference on World Wide Web (WWW '09)*, pages 791–800, 2009.

[46] Y. Zheng and X. Zhou, editors. *Computing with Spatial Trajectories.* Springer, 2011.

A Pattern Language

In the following, we present the details of our pattern language. A pattern consists of up to four components. These are pattern elements, conditions, results, and assignments.

A.1 Atomic Pattern Elements

The first part of a pattern comprises arbitrary many atomic pattern elements which are aligned in temporal order (aside from regular expression structures). Before we define the contents of an atomic pattern element, basic temporal concepts have to be introduced.

Let i be an instant. Then i has the form y-m-d-h:min:$s.ms$, where the year y has at least four digits and may be negative, the month m, the day d, the hour h and the minute min have their standard ranges and two digits each, and the millisecond ms has three digits.

Let T be a set of temporal periods, and let i_1 and i_2 be two instants. An element of T, that is, a period, can be entered in one of the following ways:

- $i_1 \sim i_2$. As the tilde symbol stands for "until", the instant on the left of it must not be greater than the one on the right;
- $i_1 \sim$, meaning the interval from the specified instant until eternity;
- $\sim i_1$, meaning the interval from the beginning of time until the specified instant;
- i_1, meaning the interval from the beginning of i_1 until its end;
- the name of a SECONDO database object of type *periods* or *interval*;
- a semantic date, i.e., a weekday (monday, ..., sunday), a daytime (morning, afternoon, evening, night), or a month (january, ..., december), each of which can be considered as an infinite period or a periodically repeating interval.

In each of the first four options, the instants may be abbreviated, resulting in different interpretations:

- $y_1 \sim y_2$-m_2-d_2 is an abbreviation for y_1-01-01-00:00:00.000 $\sim y_2$-m_2-d_2-23:59:59.999;
- y-$m \sim$ is the shortened form of y-m-01-00:00:00.000 $\sim$;
- $\sim y$-m-d-h abbreviates $\sim y$-m-d-h-59:59.999;
- y-m-d is short for y-m-d-00:00:00.000 $\sim y$-m-d-23:59:59.999.

Moreover, for the first three cases, the user may omit the year, the month, and the day in a period specification. The single times may be abbreviated to hours and minutes and are interpreted as daytimes. Hence, they are treated like semantic dates.

If the user specification is the name of a database object of type *periods* or *interval* as well as a semantic date, the latter has the higher priority.

Let L be a set of labels each of which is entered as a string. Each atomic pattern element may contain arbitrary many time and label specifications. If T or L have more than one element, they are interpreted as a logical conjunction (for T) or disjunction (for L), respectively, and have to be input as a set of comma-separated components between curly brackets. Otherwise, the brackets may be omitted.

Applying the recent definitions, each atomic pattern element has one of the following forms:

- `(T L)`, matching exactly one unit `(t l)` if and only if $t \subset t'$ for every $t' \in T$ and $l \in L$;
- `+`, matching any non-empty sequence of units;
- `*`, matching any sequence of units.

If no time and/or label specification is desired, the respective part has to be replaced by an underscore. The empty atomic pattern element `(_ _)` may be abbreviated by `()`.

The user may assign a unique variable – starting with a capital letter followed by arbitrary many letters and/or digits – to each pattern element by prepending it to the element. We discuss the use of variables in the following subsections.

A.2 Conditions

The pattern elements are followed by two slashes and any number of comma-separated conditions. A condition is any expression of boolean type which can be evaluated by SECONDO and contains at least one expression of the form *v.attr*, where *v* is one of the variables bound to a pattern element and *attr* is one of the condition attributes {label, time, start, end, leftclosed, rightclosed, card, labels}.

Note that a condition is invalid if the data type of one of the chosen attributes is incompatible with the rest of the expression. Apart from that, the use of the label attribute in a condition is only allowed if the corresponding variable refers to a unit pattern, whereas card or labels may only be used in combination with a sequence variable.

While an expression of the form *v*.label is a *string*, *v*.labels has its own data type called *labels*. For the latter, SECONDO provides the operator `contains` which checks whether a *string* is an element of a *labels* collection. The other data types (*string* for label, *periods* for time, *instant* for start and end, *bool* for leftclosed and rightclosed, and *int* for card) can be processed in numerous ways.

A.3 Results

The specification of results is mandatory for the **rewrite** operator. The result section is separated from the conditions by an arrow (`=>`) and consists of arbitrary many variables. However, there are several restrictions. For those variables attached to pattern elements, they have to be arranged in the same order as they are in the pattern elements section. New variables may be positioned freely, but it is necessary to assign a label and a time interval to them, otherwise the input is rejected. In addition, all result variables have to be unique.

A.4 Assignments

Finally, the user may assign additional information to the trajectory parts extracted by the three previous sections. More exactly, this process is optional for variables occurring in the first section and mandatory for the rest of them. The assignment part consists of arbitrary many comma-separated assignments. Each of these starts with an expression of the form *v.attr* and the assignment symbol `:=`, where *v* is one of the variables from the results part and *attr* is one of the assignment attributes {label, time, start, end, leftclosed, rightclosed}. Note that assigning a cardinality or a set of labels is not reasonable and therefore not allowed, i.e., expressions of the form *v*.card or *v*.labels may not occur on the left side of the assignment symbol. Variables referring to sequence patterns cannot be assigned new values at all.

On the right side of the assignment symbol, the user has to provide an expression which may contain any number of *v.attr* terms (see above) and must be evaluable by SECONDO. Moreover, the resulting data type of the right side has to correspond to the data type of the attribute on the left side. By default, the boolean values leftclosed and rightclosed are set to *true* and *false*, respectively, so it is not necessary to manipulate them for new variables.

A time interval has to be assigned to a variable introduced in the results section by specifying either a start and an end assignment or a single time assignment.

Finally, it has to be mentioned that in case of an invalid result (e.g., due to overlapping time intervals), the respective symbolic trajectory is removed from the output stream.

B Experimental Results

In this section, the results of the above experiments are presented in tabular form. Again, all runtimes are displayed as seconds, and m and r represent the number of units inside a symbolic trajectory and the number of symbolic trajectories, respectively.

B.1 Experiments with a Synthetic Dataset

B.1.1 matches

Applied patterns:

P0 `(2012-01-01 "Innenstadt-Ost") (_ "Hörde")`

P1 `() [* | (_ "Hombruch")]+ () *`

P2 `[(_ "Aplerbeck") | [(_ "Scharnhorst") (january _)]* | (2012-04 "Eving")]* [(_ "Scharnhorst")]+ ()`

P3 `{X * Y (2012-01 {"Innenstadt-West", "Innenstadt-Nord", "Innenstadt-Ost"}) Z * // Y.leftclosed = TRUE`

P4 `A () B [* | (thursday _)]+ C () D * // B.card > D.card`

P5 `X + Y (_ "Innenstadt-Ost") * (_ "Hörde") Z + // X.start > Z.end`

Table 4: Operator **matches**; without conditions and with conditions

without conditions				with conditions			
m	P0	P1	P2	m	P3	P4	P5
0	0.005	0.003	0.008	0	0.006	0.008	0.007
10,000	0.003	0.047	0.034	400	0.008	0.018	0.014
20,000	0.004	0.088	0.055	800	0.009	0.046	0.04
30,000	0.005	0.133	0.082	1,200	0.009	0.087	0.154
40,000	0.005	0.173	0.11	1,600	0.009	0.148	0.355
50,000	0.005	0.215	0.134	2,000	0.012	0.226	0.666

B.1.2 rewrite

Applied patterns:

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`

B.1.3 filtermatches and indexmatches

Applied pattern:

```
* (_ "Aplerbeck") * (_ "Hörde") * (_ "Eving") *
```

Table 5: Operator **rewrite**; with a single trajectory and a trajectory relation

single trajectory				trajectory relation, P9			
						m	
m	P6	P7	P8	r	100	200	300
0	0.004	0.002	0.004	0	0.005	0.003	0.007
400	0.008	0.051	0.007	200	0.274	0.667	1.178
800	0.009	0.169	0.024	400	0.543	1.326	2.351
1,200	0.006	0.37	0.077	600	0.809	1.994	3.561
1,600	0.007	0.649	0.219	800	1.078	2.659	4.707
2,000	0.012	1.021	0.382	1,000	1.348	3.324	5.883

Table 6: Operators **filtermatches** and **indexmatches**

		m				m	
r	100	200	300	r	100	200	300
0	0.009	0.005	0.006	0	0.008	0.006	0.007
200	0.11	0.249	0.387	200	0.009	0.013	0.018
400	0.218	0.487	0.766	400	0.015	0.026	0.035
600	0.325	0.728	1.148	600	0.022	0.037	0.049
800	0.429	0.972	1.524	800	0.028	0.047	0.058
1,000	0.536	1.213	1.896	1,000	0.034	0.058	0.073

B.1.4 classify and indexclassify

Applied descriptions and patterns:

- `start at Hörde, end at Brackel`
  `(_ "Hörde") * (_ "Brackel")`
- `start at Innenstadt-West, Lütgendortmund`
  `(_ "Innenstadt-West") (_ "Lütgendortmund") *`
- `Aplerbeck before Lütgendortmund before Eving`
  `* (_ "Aplerbeck") * (_ "Lütgendortmund") * (_ "Eving") *`

Table 7: Operators **classify** and **indexclassify**

		m				m	
r	100	200	300	r	100	200	300
0	0.007	0.008	0.015	0	0.007	0.008	0.011
200	0.145	0.287	0.427	200	0.021	0.028	0.038
400	0.281	0.558	0.853	400	0.032	0.044	0.06
600	0.412	0.837	1.271	600	0.038	0.066	0.084
800	0.548	1.108	1.675	800	0.049	0.079	0.115
1,000	0.686	1.382	2.096	1,000	0.062	0.1	0.137

B.2 Experiments with BerlinMOD Data

Approach 1: Moving Points and a Geometric Index

```
let strassen2 = strassen feed sortby[Name]
    groupby[Name; Line: group feed projecttransformstream[GeoData]
    collect_line[TRUE]] consume;

let strassen2_Name_btree = strassen2 feed addid createbtree[Name];

let getbb = fun(SName: string)
    bbox(strassen2_Name_btree strassen2 exactmatch[SName] extract[Bbox]);

let units_rtree_2d = SymTrips feed addid projectextendstream[TID; Bbox: units(.MP)]
    replaceAttr[Bbox: rectproject(bbox(.Bbox), 1, 2)] sortby[Bbox]
    bulkloadrtree[Bbox];
# 2249.7 seconds

let filterendpoints = fun(MPoint: mpoint, SName: string)
    units(MPoint) transformstream
    projectextend[; Sp: val(initial(.Elem)), Ep: val(final(.Elem))]
    filter[distance(.Sp, strassen_Name_btree strassen2
        exactmatch[SName] extract[Line]) < 0.03]
    filter[distance(.Ep, strassen_Name_btree strassen2
        exactmatch[SName] extract[Line]) < 0.03] head[1] count > 0;

query units_rtree_2d windowintersectsS[getbb("Bundesallee")]
    sortby[Id] rdup SymTrips gettuples
    filter[filterendpoints(.MP, "Bundesallee")] count;
# 521.5 seconds

let getfirsttimeMP = fun(MPoint: mpoint, SName: string)
    units(MPoint) transformstream
    extend[Sp: val(initial(.Elem)), Ep: val(final(.Elem))]
    filter[distance(.Sp, strassen_Name_btree strassen2
        exactmatch[SName] extract[Line]) < 0.03]
    filter[distance(.Ep, strassen_Name_btree strassen2
        exactmatch[SName] extract[Line]) < 0.03]
    head[1] projectextend[; End: inst(final(.Elem))] extract[End];

let filterendpointsafter = fun(MPoint: mpoint, SName: string, After: instant)
    units(MPoint) transformstream
    filter[inst(initial(.Elem)) >= After]
    projectextend[; Sp: val(initial(.Elem)), Ep: val(final(.Elem))]
    filter[distance(.Sp, strassen_Name_btree strassen2
        exactmatch[SName] extract[Line]) < 0.03]
    filter[distance(.Ep, strassen_Name_btree strassen2
        exactmatch[SName] extract[Line]) < 0.03] head[1] count > 0;

query units_rtree_2d windowintersectsS[getbb("Bundesallee")] sortby[Id] rdup {a}
    units_rtree_2d windowintersectsS[getbb("Pariser Platz")] sortby[Id] rdup {b}
    hashjoin[Id_a, Id_b, 999997] project[Id_a] SymTrips gettuples
    filter[filterendpoints(.MP, "Bundesallee")]
    extend[FirstEnd: getfirsttimeMP(.MP, "Bundesallee")]
    filter[filterendpointsafter(.MP, "Pariser Platz", .FirstEnd)] count;
# 53.5 seconds

query units_rtree_2d windowintersectsS[getbb("Bundesallee")] sortby[Id] rdup {a}
    units_rtree_2d windowintersectsS[getbb("Pariser Platz")] sortby[Id] rdup {b}
```

```
    hashjoin[Id_a, Id_b, 999997] {c}
    units_rtree_2d windowintersectsS[getbb("Unter den Linden")] sortby[Id] rdup {d}
    hashjoin[Id_a_c, Id_d, 999997] project[Id_a_c] SymTrips gettuples
    filter[filterendpoints(.MP, "Bundesallee")]
    extend[FirstEnd: getfirsttimeMP(.MP, "Bundesallee")]
    filter[filterendpointsafter(.MP, "Pariser Platz", .FirstEnd)]
    extend[SecondEnd: getfirsttimeMP(.MP, "Pariser Platz")]
    filter[filterendpointsafter(.MP, "Unter den Linden", .SecondEnd)] count;
# 58.4 seconds
```

Construction of Symbolic Trajectories

```
let CLUSTER_SIZE = 12;

let strassen_dup_dlo = strassen feed
  intstream(1, CLUSTER_SIZE) namedtransformstream[DSNo] product
  spread[;DSNo, CLUSTER_SIZE, FALSE;]
  hadoopMap[DLO,TRUE; . consume];

let strassen_GeoData_rtree_dup_dlo = strassen_dup_dlo hadoopMap[DLO,
    TRUE; . feed addtupleid sortby[GeoData] bulkloadrtree[GeoData]];

let Trips_Tripid_dlf = "Trips" ffeed['';;] spread[;TripId, TRUE;];

let SymTrips = Trips_Tripid_dlf hadoopMap[ DLF, TRUE; . extend[ML: units(.MP)
    transformstream projectextend[; Sp: val(initial(.Elem)),
                                    Ep: val(final(.Elem)),
                                    Si: inst(initial(.Elem)),
                                    Ei: inst(final(.Elem))]
    projectextend[Si, Ei; Cp: (.Sp + .Ep) scale[0.5]]
    projectextend[Si, Ei; N: para(strassen_GeoData_rtree_dup_dlo)
                  para(strassen_dup_dlo) distancescan3[.Cp,1] extract[Name]]
    projectextend[; U : the_unit(tolabel(.N),.Si,.Ei,TRUE,FALSE)]
    makemvalue[U]]] collect[] consume;
# 176 minutes, 48 seconds
```

Approach 2: A Linear Scan

```
query SymTrips feed filter[.ML passes tolabel("Bundesallee")] count;
# 3.0 seconds

let getfirsttime = fun(MLabel: mlabel, Name: string)
    units(MLabel) transformstream
    filter[tostring(val(initial(.Elem))) = Name]
    head[1] projectextend[; End: inst(final(.Elem))] extract[End];

let mlpassesnameafter = fun(MLabel: mlabel, Name: string, After: instant)
    units(MLabel) transformstream
    filter[inst(initial(.Elem)) >= After]
    filter[tostring(val(initial(.Elem))) = Name] head[1] count > 0;

query SymTrips feed filter[.ML passes tolabel("Bundesallee")]
    extend[FirstEnd: getfirsttime(.ML, "Bundesallee")]
    filter[mlpassesnameafter(.ML, "Pariser Platz", .FirstEnd)] count;
# 4.5 seconds

query SymTrips feed filter[.ML passes tolabel("Bundesallee")]
    extend[FirstEnd: getfirsttime(.ML, "Bundesallee")]
```

```
    filter[mlpassesnameafter(.ML, "Pariser Platz", .FirstEnd)]
    extend[SecondEnd: getfirsttime(.ML, "Pariser Platz")]
    filter[mlpassesnameafter(.ML, "Unter den Linden", .SecondEnd)] count;
# 4.6 seconds
```

Approach 3: Spatiotemporal Pattern Queries

```
query SymTrips feed filter[. stpattern[
    BA: .ML inside tolabel("Bundesallee"),
    PP: .ML inside tolabel("Pariser Platz");
    stconstraint("BA", "PP", vec("aabb", "aa.bb"))]] count;
# 8.0 seconds

query SymTrips feed filter[. stpattern[
    BA: .ML inside tolabel("Bundesallee"),
    PP: .ML inside tolabel("Pariser Platz"),
    UdL: .ML inside tolabel("Unter den Linden");
    stconstraint("BA", "PP", vec("aabb", "aa.bb")),
    stconstraint("PP", "UdL", vec("aabb", "aa.bb"))]] count;
# 9.2 seconds
```

Approach 4: Working with an Inverted File

```
let mltotext = fun(MLabel: mlabel)
    units(MLabel) transformstream
    projectextend[; Elem: totext(val(initial(.Elem)))]
    aggregateB[Elem; fun(a: text, b: text) a+'@'+b; ''];

let inv = SymTrips feed addid extend[Text: mltotext(.ML)]
    createInvFile[Text, TID, FALSE, 1, '', "@"];
# 74.8 seconds

query inv searchWord["Bundesallee"] groupby[Tid; Count: group count] count;
# 0.1 seconds

query inv searchWord["Bundesallee"] {a} inv searchWord["Pariser Platz"] {b}
    hashjoin[Tid_a, Tid_b, 999997] filter[.WordPos_a < .WordPos_b] count;
# 0.2 seconds

query (inv searchWord["Bundesallee"] {a} inv searchWord["Pariser Platz"] {b}
    hashjoin[Tid_a, Tid_b, 999997] filter[.WordPos_a < .WordPos_b])
    inv searchWord["Unter den Linden"] {d}
    hashjoin[Tid_a, Tid_d, 999997] filter[.WordPos_b < .WordPos_d] count;
# 0.5 seconds
```

Approach 5: Pattern Matching

```
query SymTrips feed filtermatches[ML, '* (_ "Bundesallee") *'] count;
# 6.6 seconds

query SymTrips feed filtermatches[ML,
    '* (_ "Bundesallee") * (_ "Pariser Platz") *'] count;
# 6.6 seconds

query SymTrips feed filtermatches[ML,
    '* (_ "Bundesallee") * (_ "Pariser Platz") * (_ "Unter den Linden") *'] count;
# 6.6 seconds
```

Approach 6: Pattern Matching with a Trajectory Index

```
let SymTripsIndex = SymTrips createtrie[ML];
# 185.7 seconds

query SymTrips indexmatches[ML, SymTripsIndex, '* (_ "Bundesallee") *'] count;
# 1.1 seconds

query SymTrips indexmatches[ML, SymTripsIndex,
    '* (_ "Bundesallee") * (_ "Pariser Platz") *'] count;
# 1.1 seconds

query SymTrips indexmatches[ML, SymTripsIndex,
    '* (_ "Bundesallee") * (_ "Pariser Platz") * (_ "Unter den Linden") *'] count;
# 1.1 seconds
```

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[352] Güting, R. H., Behr, T., Xu, J.:
Efficient k-Nearest Neighbor Search on Moving Object Trajectories

[353] Bauer, A., Dillhage, R., Hertling, P., Ko K.I., Rettinger, R.:
CCA 2009 Sixth International Conference on Computability and Complexity in Analysis

[354] Beierle, C., Kern-Isberner, G.:
Relational Approaches to Knowledge Representation and Learning

[355] Sakr, M.A., Güting, R.H.:
Spatiotemporal Pattern Queries

[356] Güting, R. H., Behr, T., Düntgen, C.:
SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations

[357] Düntgen, C., Behr, T., Güting, R.H.:
Assessing Representations for Moving Object Histories

[358] Sakr, M.A., Güting, R.H.:
Group Spatiotemporal Pattern Queries

[359] Hartrumpf, S., Helbig, H., vor der Brück, T. , Eichhorn, C.:
SemDupl: Semantic Based Duplicate Identification

[360] Xu, J., Güting, R.H.:
A Generic Data Model for Moving Objects

[361] Beierle, C., Kern-Isberner, G.:
Evolving Knowledge in Theory and Application: 3rd Workshop on Dynamics of Knowledge and Belief, DKB 2011

[362] Xu, J., Güting, R.H.:
GMOBench: A Benchmark for Generic Moving Objects

[363] Finthammer, M.:
A Generalized Iterative Scaling Algorithm for Maximum Entropy Reasoning in Relational Probabilistic Conditional Logic Under Aggregation Semantics

[364] Güting, R.H., Behr, T., Düntgen, C.:
Book Chapter: Trajectory Databases

[365] Paul, A.; Rettinger, R.; Weihrauch, K.:
CCA 2012 Ninth International Conference on Computability and Complexity in Analysis (extended abstracts)

[366] Lu, J., Güting, R.H.:
Simple and Efficient Coupling of a Hadoop With a Database Engine

[367] Hoyrup, M., Ko, K., Rettinger, R., Zhong, N.:
CCA 2013 Tenth International Conference on Computability and Complexity in Analysis

[368] Beierle, C., Kern-Isberner, G.:
4th Workshop Dynamics of Knowledge and Belief (DKB-2013)