Efficient Mobility Pattern Stream Matching on Mobile Devices

Simona-Claudia Florescu[1] and **Michael Mock**[1] and **Christine Körner**[1] and **Michael May**[1]

Abstract. The increasing amount of mobile phones that are equipped with localization technology offers a great opportunity for the collection of mobility data. This data can be used for detecting mobility patterns. Matching mobility patterns in streams of spatio-temporal events implies a trade-off between efficiency and pattern complexity. Existing work deals either with low expressive patterns, which can be evaluated efficiently, or with very complex patterns on powerful machines. We propose an approach which solves the trade-off and is able to match flexible and sufficiently complex patterns while delivering a good performance on a resource-constrained mobile device. The supported patterns include full regular expressions as well as relative and absolute time constraints. We present the definition of our pattern language and the implementation and performance evaluation of the pattern matching on a mobile device, using a hierarchy of filters which continuously process the GPS input stream.

1 INTRODUCTION

The analysis of mobility behavior based on GPS-tracks has become a popular field of research [15, 7, 4, 16, 18]. In the context of the European LIFT [10] project, we aim at the on-line monitoring of global non-linear phenomena from massively distributed streams of data. In the mobility domain such global phenomena are, for example, mass events or changes in traffic flows. The basic approach of LIFT technology for the reduction of communication overhead is to build local mobility models on each device and to communicate only significant changes to a central coordinator, which is computing the global model. This paper presents an approach for building the local mobility model efficiently on a mobile device.

Mobility patterns such as used in [4] and [7] are an appropriate way of modeling spatio-temporal mobility behavior. Powerful specialized database systems such as [16] allow to retrieve patterns from spatio-temporal data using complex pattern queries, in which spatial and temporal conditions can be freely combined. Providing this flexibility for pattern definitions for building local mobility models on a mobile device would surely exceed the computational power of such devices. Patterns expressed by regular expressions only, but not supporting queries over travel times (as in [4]) might have a better chance of being efficiently implemented on a mobile device. The same holds for the work of [7], which allows queries over travel times but supports sequential patterns only. Our approach of building mobility models is based on the notion of *visits* as being formally introduced in [12, 11]. Patterns are build as regular expression over *visits*, and time constraints are applied to complete patterns. For achieving an efficient implementation on the mobile device, we spread the task of pattern matching over a filter hierarchy that is fed with the stream of GPS input data: Firstly, a *VisitEventFilter* detects whether a certain location is being visited and, if so, forwards a *visit event* to a *PatternFilter*, which can handle arbitrary regular expressions (including Kleene closure) over visit events. Lastly, a *TimeConstraintFilter* is used to check any expression over the travel time for the complete pattern. By this approach, we can use standard deterministic automatons for implementing matching of regular expressions and can perform efficient time constraint checking in constant time. The remainder of this paper is structured as follows: in the next section, we present our approach, *mobility pattern matching over streams* containing the pattern definition language and details of the implementation of the pattern matching algorithm. Section 3 contains the performance and scalability evaluation and Section 4 discusses related work. The last section, conclusions, provides a short summary, improvement suggestions and future work.

[1] Fraunhofer Institute for Intelligent Analysis and Information Systems, Germany, email:simona.florescu@gmail.com, firstname.lastname@iais.fraunhofer.de

2 MOBILITY PATTERN MATCHING

Figure 1 describes our general approach for building local and global mobility models. As described in [11], our mobility model is based on counts of occurrences of events, whereby an event represents the occurrence of a specific predefined spatio-temporal behavior in the observed GPS track. The *local mobility model* represents the behavior of a specific user and is locally computed on the device itself, whereas the *global model* is build by aggregating all local models on a single node (global coordinator). LIFT technology is used to reduce the amount of communication needed for maintaining the global model correct over time. The basic approach thereby is to define a so-called *SafeZone*, in which the local model can safely vary without notifying the global coordinator [17]. In this paper, we focus on the question whether the input for generating the local model can be computed efficiently on a mobile device, i.e., the gray shaded part in Figure 1. Being able to compute a model locally is a prerequisite for applying LIFT technology for communication reduction. The local mobility model is computed by processing the stream of GPS updates as provided by an *Android Location Provider* through a hierarchy of filters (see [8] for the details of filter interface definitions). At the first level, the *VisitEventFilter* detects whether the device stays for a pre-defined minimum time in one of the pre-defined locations, which are stored in the local location database. If so, a *visit event* is generated, which will be processed at the next layer in the hierarchy, the *PatternFilter*. This Filter takes list of predefined patterns (regular expressions over visits) as input and matches the incoming *visit events* against these patterns. In case of a match, a *pattern event*

is forwarded to the next filter. At the last filter level, the *TimeConstraintFilter*, the time constraints for the matched pattern are validated. If they are fulfilled, the respective pattern frequency count is increased. The input for our implementation consists therefore of: (1) an infinite stream of GPS-sensed location updates, (2) a given set of interesting locations to be monitored, (3) a set of patterns with the set of interesting locations as domain, as depicted in the figure below, Figure 1.

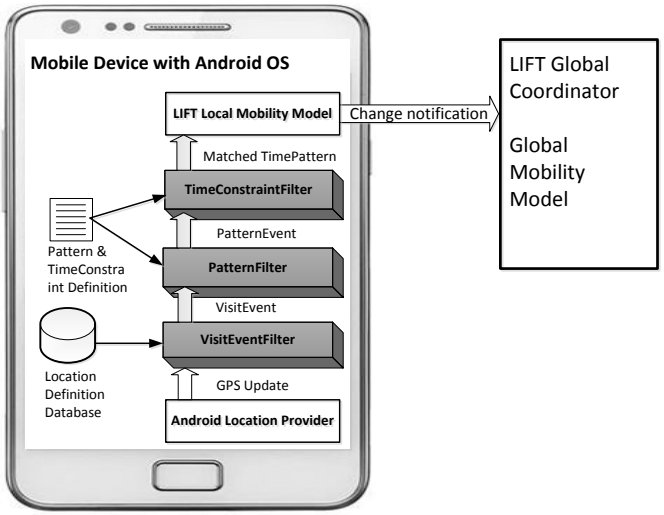

Figure 1. Filter hierarchy and data flow of the approach.

2.1 Pattern Matching Language

We propose a pattern language based on regular expressions. We define the language for the three main levels of our approach: ***VisitEventFilter***, ***PatternFilter*** and ***TimeConstraintFilter***.

Firstly, we define a ***location***, of which the input data for the *VisitEventFilter* consists. A location is defined by an *id*, a *type*, a *spatial extend* and the minimum *stay time* at the location. Note that this definition allows for overlapping locations (for example: a location for a specific attraction inside the location "Amusement Park") as well as for monitoring complete regions by dividing a region in a spatial grid of locations. In our implementation we consider rectangular spatial shapes, therefore we define the four coordinates of the bounding boxes. The id is a unique identification for the location and the type of location (e.g. cinema, fast-food, school) is coded for shortness purposes with two digits. The minimum stay time defines the time period that an encounter with a location must last in order to become a visit.[2] A location is defined as:

$$location := id, type, x_{min}, x_{max}, y_{min}, y_{max}, minStay \quad (1)$$

We represent a *visit event*, generated by the *VisitEventFilter* with the following attributes: location identification, location type, entry and exit time (in milliseconds):

$$visitEvent := (id, type, entryTime, exitTime) \quad (2)$$

We define a *visitExpression* as being the concatenation of the *id*, *type* and the *stayTime* using the within separator ",":

$$visitExpression := concat(id, type, stayTime, sep = ",") \quad (3)$$

The stay time is the difference between the exit and entry time. Similarly to [4] it is expressed as a sequence of repeated time units t so that it can be matched by regular expressions. This enables pattern queries like "a stay time of at least 5 and at most 20 minutes". The duration of a time unit depends on the required accuracy and can be set to e.g. one minute. An example of a visit expression is: ***1,01,tttt*** which represents a visit event with location 1, of type 01 (here code for cinema) and a stay time of 4 time units.

A ***pattern*** consists of (1) a regular expression of one or several *visitExpressions* and (2) a *timeConstraint* - containing absolute and relative constraints:

$$pattern := (regex(visitExpression+), timeConstraint) \quad (4)$$

$$timeConstraint := ([f_{exit}], [op_f], [l_{entry}], [op_l], [lc], [rc]) \quad (5)$$

The regular expression is defined according to the regular language specified in [14] on the alphabet of *visitExpressions* (Definition 3). Several expressions are hereby separated by a semicolon. In a digital format we represent a pattern in XML (Extended-Markup-Language). Figure 2 shows an example of a mobility pattern. The pattern's XML representation is shown in the code snippet below Figure 2. The part of the pattern containing the regular expression is

```
1,01,t{0,4};@;2,02,t{4,8};
```

It denotes a visit to location 1 (of type 01) for up to 4 time units followed by an arbitrary number of visits to unspecified locations (expressed by @[3]), followed by a visit to location 2 (of type 02) for a stay time between 4 and 8 time units.

The ***time constraint*** of the pattern definition (Definition 5) is checked by the ***TimeConstraintFilter*** and contains absolute and relative constraints. The **absolute time constraints** are: (1) f_{exit} - the constraint on the first event exit time in milliseconds; (2) op_f - the operator applied to f_{exit} with the following possible values: 0 for less, 1 for equals, 2 for greater than, - for none; (3) l_{entry} - the constraint on the last event entry time in milliseconds and (4) op_l - the operator applied to l_{entry} with the same values as op_f. In the example given below no absolute time constraints are specified but, for example, the values: $f_{exit} = 1,000,000$ and $op_1 = 2$ would impose on the first visit event that its *exitTime* must be greater than 1,000,000. The **relative time constraints** are: (5) *lc* - the left constraint for the pattern duration in milliseconds and (6) *rc* - the right constraint for the whole pattern in milliseconds e.g. $lc < l_{entry} - f_{exit} < rc$. For the given example the relative time constraint is: $0 < visitEvent_{id=2}.entryTime - visitEvent_{id=1}.exitTime < 7,200,000$.

Below, the XML for the pattern depicted in Figure 2 is shown containing the pattern id, the regular expression of the sequence of visits as well as the time constraints.

[2] Both the bounding box radius and minimum stay time are defined application-depend, depending on the location type (e.g. for bigger locations we set the minimum stay time higher) in order to distinguish between *passing by* and *visiting*.

[3] In [14] the JAVA library *automaton* specifies a regular language implementation where the symbol '@' represents any sequence of characters

Figure 2. Mobility pattern example: a visit to location 1 followed by an arbitrary number of visits to intermediate locations followed by a visit to location 2 with a maximum time period of 2 hours (7,200,000 milliseconds) between the first and last visit

```
<?xml version='1.0'?>
<PatternList>
       <Pattern>
         <id>1</id>
         <regex>1,01,t{0,4};@;2,02,t{4,8};</regex>
         <tc>
                <f_exit>-</f_exit>
                <op_f>-</op_f>
                <l_entry>-</l_entry>
                <op_l>-</op_l>
                <lc>0</lc>
                <rc>7200000</rc>
         </tc>
       </Pattern>
</PatternList>
```

2.2 Pattern Matching Algorithm

Our approach consists of several filters implemented in an embedded Android application: a *VisitEventFilter*, a *PatternFilter* and the *TimeConstraintFilter* which return a pattern distribution from a GPS stream input, see Figure 1 and Section 1.

The first filter, the ***VisitEventFilter***, receives as input the stream of GPS coordinates and a set of locations stored in a local SQLite database and generates visit events. In order to do so it checks whether there is a spatial match between the coordinates and the input of locations, described in Section 2.1 *Pattern Matching Language*. The input database contains the tables, which are joined by their id:

$Locations1(id, x_{min}, y_{min}, y_{max}, y_{max})$
$Locations2(id, latitude, longitude, name, type, min\_stay)$

The main steps of the visit filtering approach are: firstly, the database is queried to retrieve the ids of the locations in which the current position is in. As we are restricting the current implementation to rectangular locations, this can be achieved by a first query such as:

SELECT Id FROM $Locations1$ *WHERE* $x \geq x_{min}$ *and* $x \leq x_{max}$ *and* $y \geq y_{min}$ *and* $y \leq y_{max}$

followed by another query on the *Locations2* table for retrieving the rest of the location information (Definition 1).

Secondly, we are maintaining a list of *entered* locations. Whenever we detect that the current GPS point is no longer in a specific one of those entered locations, we check whether we have been staying at least a time of *minStay* within that location and, if so, generate a visit event for that location. In any case, the location is removed from the list of *entered* locations. The complete algorithm can be found in [6].

In the ***PatternFilter*** we model the patterns using *deterministic finite automata* [9]. We instantiate an automaton for each of the parsed patterns from the XML input. In the *PatternMatcher* class we create and model automata using the JAVA library *automaton* [14] to match the regular expressions specified in the first part of Definition 4.

The class structure of the pattern matching algorithm consists of:

- A *PatternFilter* class, which instantiates in its constructor a list of *PatternMatcher* objects by calling the *PatternReader* class. The *PatternFilter* maintains the list of all patterns. It receives visit events in its *update* function and generates and forwards pattern events to the next filter, the *TimeConstraintFilter*.
- A *PatternMatcher* class where an automaton is modeled. In the constructor the variables needed for saving the *automaton* data structure are initialized as well as a pattern event object for storing the information of the matched pattern.
- The *PatternReader* reads and converts a pattern from XML to an object of type *PatternMatcher*.
- The filter class *TimeConstraintFilter* checks if the time constraints are fulfilled for a received pattern event. In its constructor, it reads and parses the time constraints for each pattern into a *TimeConstraint* object.

In the *PatternMatcher* constructor, provided in the pseudocode of Class 1, the field `automaton` represents an object of type *RunAutomaton*, defined in the JAVA library *automaton*. The fields for defining the state of the automaton are `actualState` and `isInitial`. The `patternEvent` is an object of type `Event` which stores the properties of the generated pattern event for each match. The logic of the *PatternMatcher* is contained in the functions *processVisit* and *reset* shown in the pseudocode of Class 1. The `processVisit` function receives a `visitEvent` as a parameter, generated by the previous filter (the *VisitEventFilter*), and returns a boolean value of true if the visit event completes the matching of a given pattern and false if not. The *processVisit* function generates the `visitExpression` which has the structure given in Definition 3. The `stepThrough` function runs through the automaton with each character from the `visitExpression` as transition and returns the step obtained after the run. A value for `step` of -1 means that the matching failed. Any other value means that the automaton advances in another state, changing variable `actualState`. In this case and if the `isInitial` was `true`, the `patternEvent` stores the `exitTime` of the visit event. If the `visitExpression` could not be matched, the function `reset` is called. There, the `patternEvent` attributes are set to null and the state of the automaton is set to initial. Another check in the function is whether the `automaton` has reached the final state. In this case the `patternEvent` stores the `entryTime` of the visit event since this is its last visit event matched in the pattern.

The pseudocode of the *PatternFilter* is shown in Class 2. In the constructor a list of *PatternMatcher* objects is generated, one for each given pattern. Further, the *PatternReader* class, which reads and parses the XML input patterns, is called. For each pattern string the *PatternReader* instantiates a *PatternMatcher* by calling its constructor as shown in Class 1. In the `update` function of the *PatternFilter*, for each new visit event update, all the existing *PatternMatcher* objects from the `patternMatcherList` are traversed and called to execute the `processVisit` function, shown in Class 1. If the returned value from `processVisit` is true, then the matched pattern event, `patternEvent` is forwarded to the next filter.

Finally, in the ***TimeConstraintFilter*** the matched pattern time constraints are checked, if any are provided. The time constraints are de-

Class 1 PATTERNMATCHER

Fields:

$automaton$	- deterministic automaton for the pattern
$actualState$	- the actual state of the automaton
$patternEvent$	- an *Event* object for a matched pattern
$isInitial$	- boolean value for initial state of automaton

Constructor: $PatternMatcher(id, regex)$

id	- pattern id
$regex$	- regular expression

```
1: this.patternEvent.id ← id
2: this.automaton ← new RunAutomaton(regex)
   // using DFA java library from [14]
3: this.reset()
```

reset()

```
4: this.patternEvent.entryTime ← null
5: this.patternEvent.exitTime ← null
6: this.actualState ← (this.automaton.getInitialState())
7: this.isInitial ← true
```

boolean processVisit(visitEvent)

```
8: visitExpression ← makeVisitExpression(visitEvent)
9: step ← automaton.stepThrough(actualState, visitExpression)
   // stepThrough returns the state reached by inputting all characters of the visitExpression to the automaton starting with actualState
10: if step ≠ −1 then // step is -1 for a mismatch
11:     if this.automaton.isInitial then
12:         this.patternEvent.entryTime ← visitEvent.exitTime
13:         this.isInitial ← false
14:     end if
15:     actualState ← step
16: else // the visitExpression could not be entirely matched
17:     this.reset()
18: end if
    // check whether the automaton has reached the end state
19: if automaton.isFinal() then
20:     this.patternEvent.exitTime ← visitEvent.entryTime
21:     this.reset()
        // automaton is set on the initial state and all variables are reinitialized
22:     return true
23: else
24:     return false
25: end if
```

Class 2 PATTERNFILTER

Fields:

$patternMatcherList$	- a list of objects of type *PatternMatcher*

Constructor: $PatternFilter()$

```
1: reader = newPatternReader()
2: this.patternMatcherList ← reader.parseAutomaton(input_file)
   // instantiate list of PatternMatcher
```

update(visitEvent)

```
3: for PatternMatcher ∈ patternMatcherList do
4:     if PatternMatcher.processVisit(visitEvent) then
5:         forward(PatternMatcher.patternEvent)
6:     end if
7: end for
```

fined in Section 2.1 (*Pattern Matching Language*) and relate to the first and last event in the matched pattern, respectively. The *TimeConstraintFilter* constructor instantiates a *TimeConstraint* object for each pattern. When invoked with a pattern id of the incoming pattern event, it checks the existing time constraints for the respective pattern id. The generated event at this level is a matched time pattern.

3 PERFORMANCE EVALUATION

Our performance evaluation for checking the potential of the application in practice is based on synthetic data. The tests were run on a Samsung Galaxy SII GT-I9100, operating Android Gingerbread, version 2.3.3. The GPS stream data consists of synthetically generated coordinates. In addition, we retrieved 800,000 points of interest (POI) from the geo-service OSM [13] for the location set. The POIs were obtained for Germany and are of 15 different types. The generated patterns are formulated similarly to the example pattern in Section 2.1 and Figure 2, i.e. they specify the first and last location and allow an arbitrary number of visit events in between. In addition, all patterns possess time constraints. Our artificial GPS data is generated such that 40% of all points lead to a match with the location database and generate a *visitEvent*. For performance evaluations of the *PatternFilter* 2,6% of the input *visitEvents* complete a pattern match. Finally, each pattern match is checked in the *TimeConstraintFilter*.

#Locations	10	100	1K	10K	100K	500K	800K
Run-time (ms)	0.72	0.69	0.79	0.82	0.75	0.73	0.75
Stddev run	0.90	0.71	0.71	0.81	0.78	0.66	0.79
DBQuery (ms)	0.35	0.33	0.39	0.42	0.35	0.37	0.36
Stddev DBQuery	0.60	0.46	0.54	0.67	0.38	0.54	0.55

Table 1. The run-time values for the *VisitEventFilter*.

#Patterns	100	1K	10K	100K	300K
Run-time (ms)	1.2	17.1	157.0	1498.4	4397.3
Stddev.	0.08	6.5	7.1	24.0	129.6
Start-up (ms)	266.3	275.1	239.2	499.9	1341.8
Stddev.	201.6	225.1	48.2	9.3	15.3
Heap size (MB)	2.67	2.68	2.75	2.76	2.76
Memory (MB)	5	10	10	12	24

Table 2. The run-time values for the *PatternFilter*

Firstly, we measured the run-time for the ***VisitEventFilter***. We varied the size of the location set from 10 to 800,000. Table 1 shows the obtained performance results. We distinguish between the database query (DBQuery) and the entire run-time (Run-time) measured before forwarding a visit event, therefore the DBQuery run-time is included in the run-time value. Table 1 shows that the run-time of the *VisitEventFilter* is nearly constant when varying the number of locations in the underlying database. We ascribe this behavior to caching effects in the SQLite database, taking into account that the database is opened only once and that all queries are read-only. Ongoing experiments and code analysis showed that the time needed for the database part in the *VisitEventFilter* depends on the number of overlapping locations in which a given GPS point resides. In the synthetic GPS tracks used in the evaluation, the GPS points match to one (non-overlapping) location only. All visit events are detected in a time well below one second, which is the time difference between two GPS location updates in worst case. In addition, the number of monitored locations will be geographically limited in practice. For example, when restricting the collected POI (points-of-interest) to the city of Cologne, we obtained a set of about 20,000 locations.

Secondly, we evaluated the run-time of the ***PatternFilter*** under an increasing number of checked patterns with an average matching percentage of 2,6%. The results are shown in Table 2. The table also contains the reading and parsing time for all patterns (Start-up), which takes place in the *PatternFilter* constructor. The run-time for the pattern matching increases linearly with the number of tested patterns (Run-time), which corresponds to the main loop executed in the *PatternFilter.update* method. Furthermore, for each of the instances of the *PatternFilter* we show the heap size and the allocated memory. The values were captured for about three runs as delivered by the Android debugger class. The results for the heap size are relatively constant and show for the memory allocations fair results, since the high number of 300,000 patterns use about 24MB out of 64MB.

The ***TimeConstraintFilter*** has a constant run-time, since it only retrieves the time constraints for the respective pattern id from a *HashMap*, and the time constraint validation is trivial. The run-time

for the *TimeConstraintFilter* is approximately 10 milliseconds per pattern event.

4 RELATED WORK

Table 3 gives an overview on related work in the field of spatio-temporal mobility analysis. The criteria on which we compare related work are: (1) stay time (Stay) - patterns with conditions on minimum and maximum stay time, (2) travel time (TT) - the time span between two locations in the pattern, (3) the time constraints (TC) - time constraints on the full pattern, i.e. on first and last location in the pattern, (4) full regular expressions (FullRE) - supporting all the expression options from regular expressions i.e. Kleene closure (+,*), negation, conjunction, disjunction, and (5) predicates - additional complex conditions on the pattern (e.g. an attribute should have an incrementing value) and (6) Stream - whether the approach is applied on-line (on stream data) or later, off-line. Although our matched patterns are not the most complex, our approach is the first one successfully tested on a resource-constrained device.

Approach	Stay	TT	TC	FullRE	Pred.	Stream
T-patterns [7]	-	✓	-	-	-	-
Mob. patterns [4]	✓	-	-	✓	-	-
SASE [18]	✓	-	✓	✓	✓	✓
SASE$^+$ [1]	✓	-	-	✓	✓	✓
Cayuga [2]	-	-	✓	✓	✓	✓
STPQ [16]	✓	✓	✓	✓	✓	-
Our Approach	✓	-	✓	✓	-	✓

Table 3. Comparison with related work

The function `addProximityAlert` provided by the Android `LocationManager` [3] performs a similar task as our *VisitEventFilter*. Comparative experiments between both classes showed that the Android function can register proximity alerts for only less than 30,000 locations, compared to our approach, which has been tested for up to 800,000 locations.

5 CONCLUSION

In this paper we have shown that the detection of state-of-the art complex mobility patterns can be implemented on a resource-constrained environment such as a mobile device. Our experiments show that the pattern matching can process the matching of 800,000 locations and up to 10,000 complex patterns in much less than one second. For handling more locations or more patterns, measures can be taken to reduce the number of GPS position updates by configuring the Android Location Provider appropriately or by adding intermediate GPS smoothing filters. For example, for a frequency of 5 seconds per position update request, our application can efficiently scale up to at least 800,000 locations and over 300,000 complex patterns.

We consider a few improvements for future work. We have initial measurements of battery consumption which are promising, but need to be investigated in detail. The *VisitEventFilter* performs very fast (see Table 1). This results from using rectangular locations only, which allows to search for locations with the simple query shown in Section 2.2 efficiently. Specific applications may, however, require more complex location geometries. For the *PatternFilter*, run-times mainly depend of the loop executed over the set of patterns. Here, we can explore parallelism of the underlying multi-core hardware and we can apply optimizations from the area of *complex event processing*, see [5]. Furthermore, the handling of travel times (in addition to pattern time constraints) will be investigated by the repeated hierarchical composition of our *PatternFilter* and *TimeConstraint* filters. Lastly, we will evaluate the performance of our approach on real-world data collected from 70 users in the city of Cologne, Germany.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 255951 (LIFT Project).

REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, 'Efficient pattern matching over event streams', in *Proc. of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pp. 147–160, New York, NY, USA, (2008). ACM.

[2] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White, 'Cayuga : A General Purpose Event Monitoring System', *Publish*, 412–422, (2007).

[3] Android developer website. www.developer.android.com, Last accessed: April 2012.

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

[5] O. Etzion and P. Niblett, *Event Processing in Action*, Manning Publications Company, 2010.

[6] S.-C. Florescu, 'Efficient retrieval of mobility patterns on mobile devices', *RWTH Aachen, Germany*, (August 2012). To be submitted.

[7] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, 'Trajectory pattern mining', in *Proc. of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pp. 330–339, New York, NY, USA, (2007). ACM.

[8] M. Hoffmann, 'A simulation environment for distributed stream analysis', *Master Thesis, Univ. of Appl. Sc. Bonn-Rhein-Sieg*, (2011).

[9] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, volume 2, Addison-Wesley, 1979.

[10] LIFT (Using Local Inference in Massively Distributed Systems). http://www.lift-eu.org.

[11] C. Körner, *Modeling Visit Potential of Geographic Locations Based on Mobility Data*, Phd thesis, University of Bonn, 2012.

[12] C. Körner, D. Hecker, M. May, and S. Wrobel, 'Visit potential: A common vocabulary for the analysis of entity-location interactions in mobility applications', in *Geospatial Thinking*, Lecture Notes in Geoinformation and Cartography, 79–95, Springer, (2010).

[13] Open Street Maps. www.osm.org, Last accessed: December 2011.

[14] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[15] D. J. Patterson, L. Liao, K. Gajos, M. Collier, N. Livic, K. Olson, S. Wang, D. Fox, and H. Kautz, 'Opportunity knocks: a system to provide cognitive assistance with transportation services', in *Ubicomp*, pp. 433–450, (2004).

[16] M. A. Sakr and R. Hartmut Güting, 'Spatiotemporal pattern queries', *GeoInformatica*, **15**(3), 497–540, (2011).

[17] I. Sharfman, A. Schuster, and D. Keren, 'A geometric approach to monitoring threshold functions over distributed data streams', in *Proc. of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pp. 301–312, New York, NY, USA, (2006). ACM.

[18] E. Wu, 'High-performance complex event processing over streams', in *Proc. of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pp. 407–418, (2006).