Semantic Trajectories in Mobile Workforce Management Applications *

Nieves R. Brisaboa, Miguel R. Luaces, Cristina Martínez Pérez, and Ángeles S. Places

Universidade da Coruña
Laboratorio de Bases de Datos
A Coruña, Spain
{brisaboa,luaces,cristina.martinez,asplaces}@udc.es

Abstract. As a consequence of the competition between different manufacturers, current smartphones improve their features continuously and they currently include many sensors. *Mobile Workforce Management (MWM)* is an industrial process that would benefit highly from the information captured by the sensors of mobile devices. However, there are some problems that prevent MWM software from using this information: i) the abstraction level of the activities currently identified is too low (e.g., *moving* instead of *performing an inspection on a client*, or *stopped* instead of *loading a truck in the facility of a client*; ii) research work focuses on using geographic information algorithms on GPS data, or machine learning algorithms on sensor data, but there is little research on combining both types of data; and iii) context information extracted from geographic information providers or MWM software is rarely used.

In this paper, we present a new methodology to turn raw data collected from the sensors of mobile devices into trajectories annotated with semantic activities of a high level of abstraction. The methodology is based on *activity taxonomies* that can be adapted easily to the needs of any company. The activity taxonomies describe the expected values for each of the variables that are collected in the system using predicates defined in a *pattern specification language*. We also present the functional architecture of a module that combines context information retrieved from MWM software and geographic information providers with data from the sensors of the mobile device of the worker to annotate their trajectories and that can be easily integrated in MWM systems and in the workflow of any company.

Keywords: semantic trajectories, mobile workforce management, sensor data, geographic information systems

* Funded by MINECO (PGE & FEDER) [TIN2013-46238-C4-3-R, TIN2013-46801-C4-3-R, TIN2013-47090-C3-3-P, TIN2015-69951-R]; CDTI and MINECO [Ref. IDI-20141259, Ref. ITC-20151305, Ref. ITC-20151247]; Xunta de Galicia (co-funded by FEDER) [GRC2013/053].

1 Background and Motivation

The use of mobile devices has grown incessantly in the last decade and more than 60% of the people in advanced economies report owning a smartphone [7]. Furthermore, the capabilities of mobile devices such as smartphones, tablet computers, and wearable devices have increased continuously. Their computing power is similar to the one of a desktop computer from the last decade, and they include multiple sensors that can be used to measure different variables such as the geographic position using a GPS receiver, the user activity using an accelerometer, or the surrounding environment using a thermometer. An industrial process that would benefit especially from the information collected using mobile devices is *Mobile Workforce Management (MWM)*. MWM systems are used by companies to manage and optimize the task scheduling of their workers (e.g., ensuring that the company has the lowest number of active employees at any time of the day) and to improve the performance of their business processes (e.g., detecting which tasks are costly for the company). As an example, it would be very beneficial for a company that collects waste materials in rural areas to detect if the workers spend a long time in the activity *refueling* to help deciding whether it is good idea to hire someone who does this work at night.

This type of studies are impossible to carry out without the historical movement data of the workers and a procedure to detect and analyze, at a high level of abstraction, the activities they were performing and how much time they required. If the MWM system can detect what has really happened on a working day and compare it to what was scheduled, it can generate useful information to manage the business processes and to detect the critical points. However, current MWM systems are not using the information that can be collected by mobile devices. At most, they are using the location services to record the time when a worker arrived at the client facility, but they do not use the full range of data collected by the sensors of the mobile devices to answer queries such as *was the worker at a traffic jam that made him arrive late?* or *how much time is wasted walking inside a client facility in each visit?* because the data is massive and complex.

The interest in coherently organizing and labeling sequences of spatio-temporal points to handle starts with the seminal works [1,10,11] where the idea of assigning semantic keywords to the raw data obtained from a GPS sensor is presented for the first time. However, these works assume that the semantic keywords are obtained from other sources and not from the GPS data itself. That is, the system analyzes the GPS data sequence and organizes it into coherent sets but, instead of labelling them, it presents them to be annotated by the user. From then on, a new line of research was born in the scientific community dedicated to semantically annotating moving objects, and the center of interest has shifted from raw data obtained from the GPS sensor, to greaters level of abstraction oriented to the needs of concrete applications. There has been lately much research work in the field of *semantic trajectories* addressing the problems of collecting, modeling, storing and analyzing these data. A semantic trajectory is a set of episodes, each one consisting of a start time, an end time, and a semantic key-

word. A good summary of the research and the challenges in the field can be found in [6]. However, the research on semantic trajectories is focusing on detecting low-level activities such as *stopped*, *moving*, *walking*, or *running* instead of high-level activities such as *stuck in a traffic jam on the way to a client facility*, or *loading the truck in the facility of a client*. The reason of this mismatch in the abstraction level is that research on semantic trajectories is aimed at the general case and there is a limited usage of context information, whereas in a MWM system the context information (e.g., the schedule of the worker) is extremely important.

Some of the techniques proposed in the research field of semantic trajectories have focused on the use of machine learning techniques for the detection of activities[2,8,9]. However, these techniques cannot be applied in MWM systems because they require training data, which is very is expensive to collect. For example, in an MWM system this would mean that employees must annotate for each task they perform during the day the start and end time, which is an expensive and cumbersome process. In addition, there will be a large margin of error because it is very difficult for an employee to perform his work and at the same time to collect the sample data for the training process. The employee will easily forget to tag the start or end of an activity and therefore, the sample data set would be wrong. Finally, this could cause the employees to make mistakes in their real tasks. Hence, the annotation of semantic trajectories using machine learning techniques do seem appropriate.

Other research papers have focused on modeling, representing, processing, querying, and mining object trajectories in their raw state. These lines of research define query languages and model the paths of moving objects so that specific patterns can be extracted later using query languages. In [3] the raw data is divided in *stop* and *move* episodes [10], and relevant geographic information from the application domain is assigned to each episode. Then, the authors define a semantic trajectory data mining query language (ST-DMQL) to extract knowledge (data mining) from the raw data using queries and movement patterns. The authors of [5] propose the idea that a trajectory is formed by segments of different levels of abstraction according to different criteria (e.g., stops and moves versus the transport system used). In [4] the authors define a *symbolic trajectory* as a set of units composed by a time interval plus a value that can be of type *label*, *labels*, *place* or *places*. The authors also define a query language with operators to filter and retrieve those symbolic trajectories. Even though all these aproaches can be used to efficiently query trajectory data, they are all techniques of a low level of abstraction and they are hardly integrable in an MWM system because it would require an untrained company manager to write complex queries.

Finally, in addition to the problems mentioned above, all the solutions are generic and do not assume any additional knowledge beyond the data of the trajectory. However, in the case of the integration with an MWM system, it is necessary to keep in mind that additional information is known and must be taken into account.

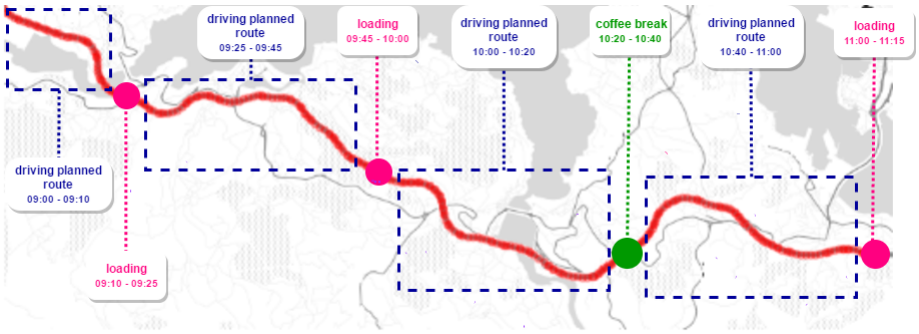

Fig. 1. Trajectory annotated with semantic activities

In this paper, we present the functional architecture of a module for MWM systems that can be used to collect the information captured by the sensors of the mobile devices, to analyze and annotate the trajectory with high-level activities using context information from the MWM system and geographic information providers, and to provide the semantic trajectory information back to the MWM system in order to support business intelligence processes. The rest of this paper is organized as follows. Section 2 describes the annotation methodology and the activity taxonomies that form the basis of the methodology. Then, Section 3 describes the activity pattern specification language that is used on the taxonomy to decide which activity is performed. Finally, Section 4 presents the functional architecture of the working system and Section 5 presents the conclusions and future work.

2 Annotation Methodology

The purpose of the annotation process is to divide the *raw data* captured by the mobile device during the working day into a collection of activities, each one consisting of a start time, an end time and a label that describes the activity performed by the worker. For example, Figure 1 shows an annotated trajectory in which the the worker was performing the activity *driving on a planned route* between 09:00 and 09:10.

The raw data used to build and annotate the trajectory is retrieved from three different sources: *the MWM system*, *the sensors of the mobile device*, and *the geographical information of the domain*. Regarding the *MWM system*, the conceptual model of the Figure 2 shows the minimum information required by the annotation process. First, it is necessary to know the information of all the workers of the company (represented in the class `Worker`). In addition, we must know all the locations to which the employees have to go in their work day (represented in the class `Location`). Each location is described by the name of the client, the name and the geographic position of the location, and a boolean

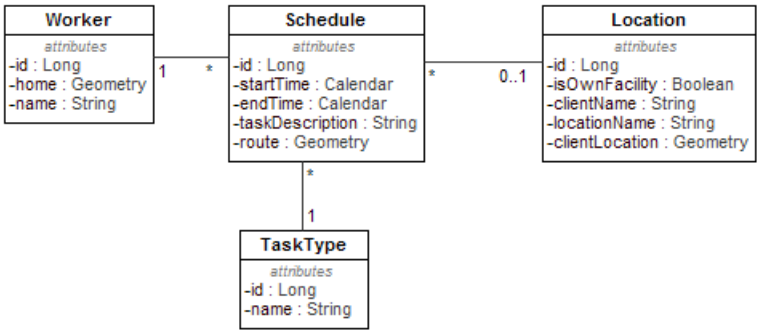

Fig. 2. Minimum information required from the MWM system

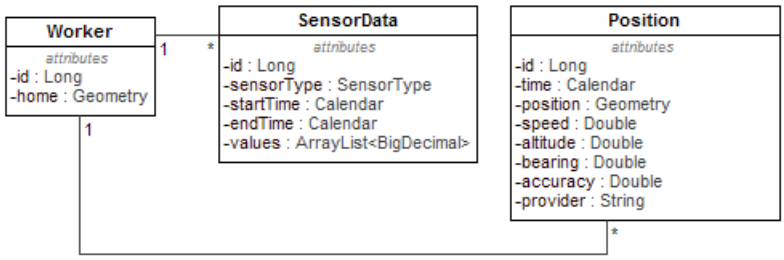

Fig. 3. Raw data collected from the mobile device

value to indicate whether the location is a facility of our own company or it is a facility of a client. Finally, the daily schedule of each worker is represented by the `Schedule` class. This class describes, for each task that must be performed by each worker, the scheduled start and end time, the type of task (from a catalog represented by `TaskType`), a description of the task, and the planned route that the employee must follow to reach the location.

Figure 3 shows a conceptual model of the information collected by the sensors of the mobile device. Each worker (represented by the `Worker` class) is related to all the values collected by the sensors. The sensor information is represented by the `SensorData` class, which stores the sensor type, the time interval when the value was collected, and the value itself. In addition, we keep all the locations of the workers (represented by the `Position` class) storing the variables collected by the location sensor: the time instant when the value is registered, the geographic location, the speed, the height, the heading, the precision and the provider of the location.

In addition to the information provided by the MWM software and the information collected by the sensors of the mobile device, the annotation process needs the geographic information of the domain to compare the location of the

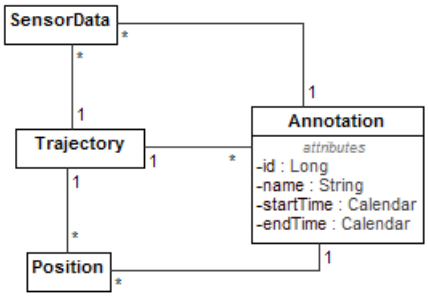

Fig. 4. Annotated trajectory

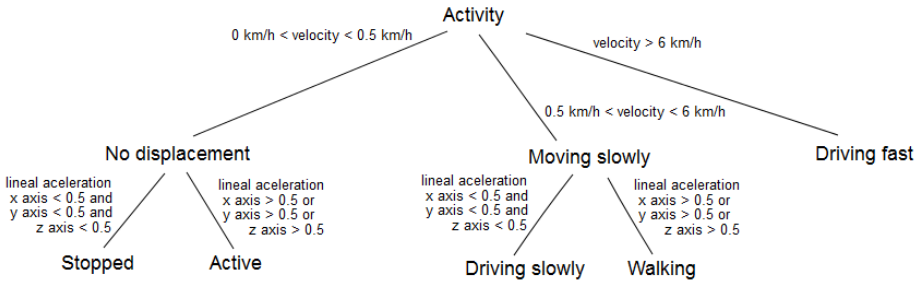

Fig. 5. Example of an activity taxonomy

worker with elements of the context such as the road network, or specific points of interest of the domain such as gas stations. This information is considered external to the annotation system, but it is used in the annotation process and the system retrieves. There are two possible ways to retrieve this information: the first by accessing a local database using SQL queries, and the second by querying OpenStreetMap using the Overpass API[1]. In both cases, the result is a set of geographic objects that can be used in the evaluation of the activity pattern.

The conceptual model in Figure 4 shows the expected result of the annotation process. A raw trajectory is a set of raw data represented by the `SensorData` and `Position` classes. An annotated trajectory consists of a set of annotations (represented by the class `Annotation`. Each annotation stores the activity name, the start and end time of the activity, and it references the raw data that compose the activity.

The basis of the annotation process is the *activity taxonomy*. For each of the worker types in the company an activity taxonomy must be defined that

[1] `https://overpass-api.de/`

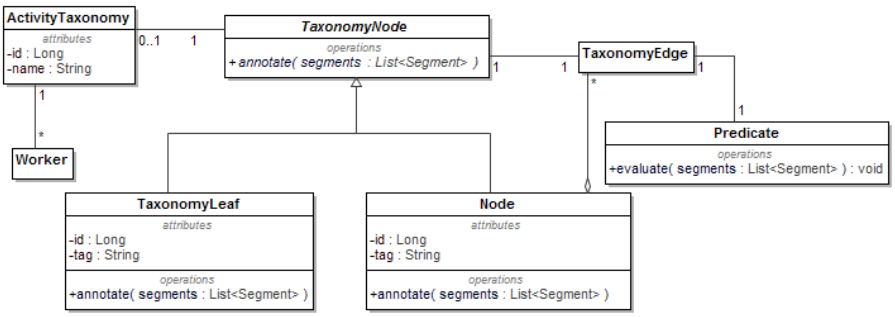

Fig. 6. Catalog of activity taxonomies

includes all the activities that can be performed by the worker type and that includes the rules for deciding what activity is performed at each time interval. The Figure 5 shows an example of an activity taxonomy. The leaf nodes of the taxonomy are the activities that can be performed by a worker (e.g., *stopped*, *active*, *driving slowly*, *walking*, *driving fast*). Each intermediate node is formed by a set of edges. Each taxonomy edge is tagged with an *activity pattern* that represents the expected values for the variables that are collected in the system (e.g., $0km/h < velocity < 0.5km/h$).

The activity taxonomies for each worker are stored in a catalog whose conceptual model is shown in Figure 6. In this catalog, each worker in the company (represented by the `Worker` class) is associated with an activity taxonomy represented by the `Taxonomy` class. The definition of the predicate associated to each edge of the taxonomy is done by means of a specification language that is described in the Section 3 and that is represented by the Pattern Specification class. Finally, a taxonomy can be associated with more than one employee since the activities identified for all employees of the company in the same category will be the same.

The procedure for annotating the trajectories is described in Algorithm 1. The first step is to obtain all the workers of the company. Then, for each worker, the activity taxonomy corresponding to the worker type is retrieved. After that, the raw data is retrieved for the worker. As shown in Figure 3, the sensor data consists of a collection of `SensorData` objects, each of them consisting of a time interval and a sensor value. Similarly, the location data consists of a collection of `Position` objects, each of them consisting of a timestamp and the GPS variables. However, these data may not be *aligned* in the sense that the start and end times of the intervals are not necessarily the same for all sensor types, and the location timestamps may not coincide with any time interval. Therefore, it is necessary a step in the algorithm to homogenize the time intervals. The result is a collection of `TrajectorySegment` objects, each of them consisting of a time interval, a sensor value for each sensor type, a `Position`.

Algorithm 1 Algorithm to annotate trajectories

```
function annotateTrajectories(currentDate: Timestamp)
    workers ← retrieveWorkers()
    for all aWorker ∈ workers do
        ▷ Retrieve the activity taxonomy
        activityTaxonomy ← retrieveActivityTaxonomy(aWorker)
        rootNode ← activityTaxonomy.getRootNode()
        ▷ Retrieve and segment the trajectory
        rawTrajectory ← retrieveTrajectoryData(aWorker, currentDate)
        segmentedTrajectory ← segmentTrajectory(rawTrajectory)
        ▷ Annotate the trajectory (see Algorithm 2)
        rawAnnotation ← annotate(segmentedTrajectory, rootNode)
        ▷ Aggregate contiguous segments of the same activity
        annotation ← aggregateSegments(rawAnnotation)
        ▷ Store the annotated trajectory
        for all segment ∈ annotation do
            annotate(rawTrajectory, segment)
        end for
    end for
end function
```

Algorithm 2 Algorithm to evaluate a trajectory in a taxonomy node

```
function annotate(segmentedTrajectory: List of segments, node: TaxonomyNode)
    if node.getType() ≠ "leaf" then
        ▷ It is an intermediate node
        edges ← note.getTaxonomyEdges()
        for all anEdge ∈ edges do
            ▷ The predicate of the edge is evaluated
            evaluation ← anEdge.getPredicate().evaluate(segmentedTrajectory)
            ▷ The segments that evaluate to true are passed to the child node
            trueSegments ← removeTrueSegments(evaluation, segmentedTrajectory)
            annotate(trueSegments, anEdge.getChildNode())
            ▷ Only the segments that evaluate to false are kept in segmentedTrajectory
            removeFalseSegments(evaluation, segmentedTrajectory)
        end for
        ▷ The remaining segments are annotated as undefined
        annotate(segmentedTrajectory, undefined)
    else
        ▷ It is a leaf node
        for all segment ∈ segmentedTrajectory do
            ▷ All segments are annotated with the activity of the node
            segment.activity ← node.getActivityName()
        end for
    end if
end function
```

In the next step, the activity taxonomy is evaluated against the segmented trajectory. Algorithm 2 shows the algorithm used for the evaluation. The process starts at the root of the taxonomy. The first taxonomy edge is retrieved and its activity pattern is evaluated against the trajectory segments. All segments that evaluate to true are passed to the taxonomy edge child node to be evaluated recursively. This recursive procedure ends when a taxomy leaf node is reached. In this case, all the segments received are annotated using the activity name in the leaf node. All the segments that do not evaluate to true in an internal node are evaluated against the next taxonomy edge. The segments that remain after all taxonomy edges are annotated as *undefined*.

After evaluating the activity taxonomy against the segmented trajectory it may happen that many segments contiguous in time are annotated with the same activity. The next step of Algorithm 1 aggregates all these segments to a single one. The final step of the algorithm stores the annotation in a database following the conceptual model described in Figure 4.

3 Pattern specification language

As described in Section 2, each edge of the activity taxonomy is associated with a predicate that is used to evaluate each trajectory segment and decide the concrete activity that is used to annotate the segment. Each predicate is defined using a *pattern specification language* that describes the expected values for each of the variables that are collected in the system. The language consists of seven different types of predicates that receive a trajectory represented as a list of segments and return the result of evaluating the predicate as a list of boolean value annotated with a time interval. The different types of predicates are the following and are grouped into two levels, primary level and composite level, this last level allows to combine simple predicates to build relationships between them.

Primary predicates

- `SensorPredicate`: It returns true if the sensor values in the segment satisfies a comparison operator.
  - `sensorname`: The name of the device sensor used.
  - `operator`: A list of comparison operators used to check the sensor value against the threshold (one operator for each sensor dimension). Each operator may be one of $<, >, =, \leq, \geq, \neq$.
  - `threshold`: A list of threshold values used in the comparison (one value for each sensor dimension).
  - `isOr`: A boolean value determining whether it is enough that one sensor dimension fulfills the comparison (true), or all sensor dimensions are required to fulfill the comparison (false).
  - `time`: The minimum time interval that the sensor value must not fulfill the comparison in order to render the predicate false. This attribute

is used to that short changes in the sensor values. turn the predicate false (e.g., a sudden and short movement of the device should not be considered relevant if the device has been static for a long period of time).

- `GPSPredicate`: It returns true if the GPS position in the segment satisfies a comparison operator.
  - `gpsattribute`: The GPS attribute used in the predicate. It may be one of *speed*, *altitude*, *bearing*, *precission*, or *location provider*.
  - `operator`: A comparison operator used to check the GPS against the threshold. The operator may be one of $<, >, =, \leq, \geq, \neq$.
  - `threshold`: A threshold value used in the comparison.
  - `time`: The minimum time interval that the GPS attribute must not fulfill the comparison in order to render the predicate false.
- `SpatialPredicate`: It evaluates whether the GPS position satisfies a spatial relationship with the context geographical information.
  - `operator`: The spatial relationship predicate used to compare the GPS position against the collection of spatial features. It may be any of the predicates defined by the Open Geospatial Consortium Simple Features Specification (i.e., *equals*, *disjoint*, *overlaps*, *touches*, *within*, *contains*, *intersects*).
  - `features`: A list of spatial features retrieved from the context geographic information.
  - `time`: The minimum time interval that the GPS position must not fulfill the spatial predicate in order to render the predicate false.
- `SchedulePredicate`: It returns true if the schedule information in the MWM for the worker satisfies a comparison operator.
  - `operator`: The comparison operator used between the task name provided in the predicate and the task scheduled in the MWM. It may be one of *equals*, *distinct*, *like*, *any* (returns true as long as there is an scheduled task).
  - `taskname`: The name of the task. It may use SQL-like wildcards.

Composite predicates

- `LogicPredicate`: It allows to combine different predicates through logical operators.
  - `operator`: The logic operator used to combine the child predicates. It may be one of *and*, *or* or *not*.
  - `childpredicates`: A list of child predicates to be evaluated and combined.
- `DecisionPredicate`: It can be used to create a decision tree.
  - `condition`: The predicate that is evaluated and that is used to decide which value must be returned.
  - `isTrue`: The predicate that is returned when the condition predicate returns true.

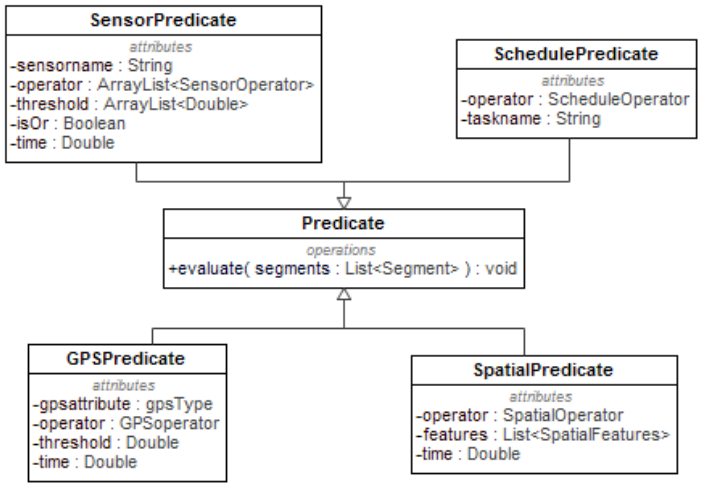

Fig. 7. Predicate types

  - `isFalse`: The predicate that is returned when the condition predicate returns false.
  - `isUndefined`: The predicate that is returned when the condition predicate returns undefined.
- `ConstantPredicate`: It returns a constant value (either true, false or undefined) regardless of the segment values. It is useful as a child predicate of a `DecisionPredicate`.
  - `value`: The value that is always returned by the predicate. It may be either *true*, *true*, or *undefined*.

Figures 7 and 8 show a conceptual model of these predicates. Each predicate is a specialization of the abstract class `Predicate`, which defines a method to evaluate the predicate on a list of trajectory segments. Each subclass of `Predicate` defines specific attributes for the concrete predicate and it overrides the method evaluate to implement a diferent evaluation procedure. Finally, `LogicPredicate` and `DecisionPredicate` are special because they require a set of child predicates (the arguments of the logical operator in the first case, the decision predicate and the predicates for each decision result).

Figure 9 shows in a conceptual way the result of evaluating some predicates. The top part of the figure shows the GPS positions of the trajectory and a spatial feature used for the evaluation. The bottom part of the figure shows the results of the predicate evaluation. The horizontal axis represents time, the vertical axis represents the result of the evaluation of the predicate using a the value zero if the evaluation is false, the value one if the evaluation is false, and no value if the evaluation is undefined. The topmost predicate, *within client facility*, is a `SpatialPredicate` that returns true when the GPS position is within the spatial

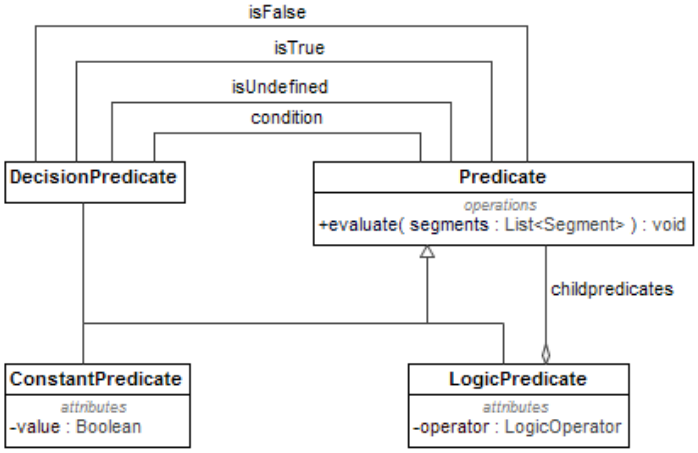

Fig. 8. Predicate types

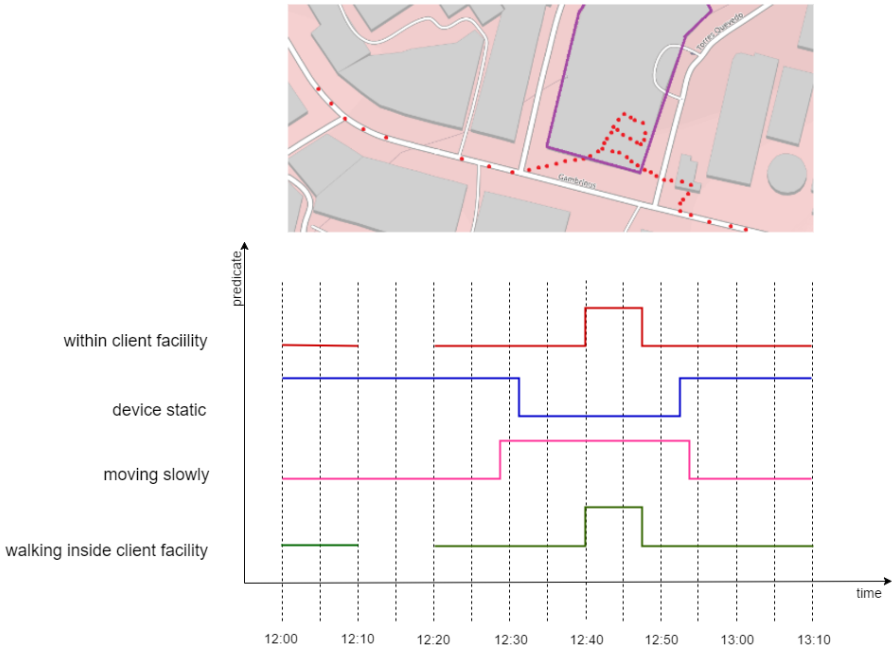

Fig. 9. Evaluation of the predicate *walking inside client facility*

feature of the client facility (i.e., between 12:40 and 12:47) and false otherwise. It also returns *undefined* from 12:10 to 12:20 because there are no GPS data. The second predicate, *device static*, is a `SensorPredicate` that returns true when the values returned by the linear accelerometer of the device are below 1 m/s^2 (hence, the device is relatively static and the user is not walking or running). The following predicate, *moving slowly*, is defined using a `GPSPredicate` that returns true when the speed recorded by the GPS is less than 10 km/h. Finally, the predicate *walking inside client facility* is defined using a `LogicPredicate` that combines the previous predicates using the logical operator *and* and negating the predicate *device static* as follows:

$$\text{walking inside client facility} = \text{within client facility} \wedge \\ \wedge \neg \text{ device static } \wedge \text{ moving slowly}$$

4 System architecture

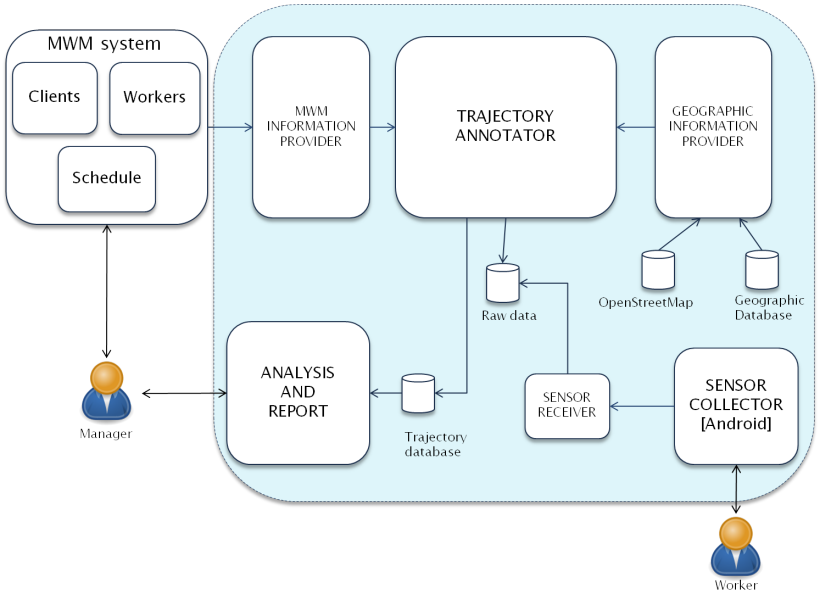

Fig. 10. Functional architecture

Figure 10 shows the functional architecture of the system. On the left side of the figure we show the MWM system with the minimal information that we expect to retrieve: information regarding the clients of the company, the workers of the company, and their daily schedules. The right side of the figure shows the

system that we have built, which is composed of a module that is deployed as an Android application on the worker mobile devices (i.e., *Sensor Collector*) and a collection of server-side modules (i.e., *Sensor Receiver*, *Geographic Information Provider*, *MWM Information Provider*, *Trajectory Annotator*, and *Analysis and Report*). The modules in the architecture are used by three different user roles: *workers* uses the *Sensor Collector* in their mobile devices, *company managers* review the information computed by the *Trajectory Annotator* using the *Analysis and Report* module, and *analysts* configure the different modules to suit the company needs (e.g., they define the activity taxonomies, define the geographic data sources, or configure the *Sensor Collector* sampling rate).

The Android sensor framework supports 13 types of sensors. An application monitoring sensor events decides the desired sampling rate ranging from 5 events per second to real-time (i.e., as fast as possible). Each sensor event consists of a timestamp, the reading accuracy and the sensor's data as an array of float values. The meaning of each element in the array depends on the sensor type, but typically each value represents the measurement in a specific dimension. For example, a linear acceleration sensor returns an acceleration measure on each of the three axes. Furthermore, each sensor reports about its capabilities, such as its maximum range, manufacturer, power requirements, and more importantly, its resolution.

The *Sensor Collector* module is an Android application that collects information from the devices of the workers and sends it to the server-side. The module can be configured to select which sensor types will be captured and the sampling rate used. In order to reduce the volume of data that must be stored in the mobile device and transmitted over the network, instead of storing every sampled value from the sensor, the data is stored aggregated in *sensor segments* that consist of a time interval (i.e., start time and end time) and a sensor event value array. When the Sensor Collector module receives a new sensor event, if the difference between the new sensor value and the value in the current sensor segment is smaller than the sensor resolution (i.e., the sensor event can be considered noise) the sensor event is discarded. Otherwise, the current sensor segment end time is set to the current timestamp and a new sensor segment is started.

The location sensor works differently in Android because the developer must configure a distance and a time threshold that must be exceeded in order to receive a location event. Therefore, the Android framework does the work of avoiding excessive location events. For example, a developer may indicate that a location event must only occur if the distance from the last position is larger than 10 meters and the time passed is larger than 10 seconds. Hence, considering that the data rate from the location sensor is lower than the data rate from the other sensors, the *Sensor Collector* does not have to discard data from the location sensor.

Given that sending data through the mobile network is expensive in terms of battery life, and considering that the worker may not be connected at all times, the *Sensor Collector* uses two data queues to optimize the network usage. When

a sensor event occurs, the data is stored in the *ready for packing* queue. Even though this task is executed frequently, it is simple and it does not consume much battery. At regular time intervals (e.g., every five minutes, although this time interval is configurable by the developer) all elements in the *ready for packing* queue are removed, compressed in a single data package using the ZIP file format, and stored in the *ready for sending* queue. A different process retrieves data packets from the *ready for sending* queue and tries to deliver them to the server-side. If the sending process is sucessfull, the data packets are removed from the queue. Otherwise, the data packets in the queue to retry sending them later. These two strategies allow the *Sensor Collector* to save battery and to be resistant to network problems.

The sensor information is received in the server-side by the *Sensor Receiver*, which is a simple REST web service that receives the sensor data packets and stores them in the *raw data repository*, whose conceptual data model can be seen in Figure 3.

The *Geographic Information Provider* retrieves geographic information from different data sources and makes it available to the trajectory annotator. This module currently supports two different types of geographic data sources: *spatial databases* (i.e., databases that support the Simple Features for SQL standard) and *OpenStreetMap* (i.e., using the Overpass API). A developer defines a geographic data source in this module providing a name for the data source and the query that must be executed (i.e., a JDBC connection string and a SQL query for spatial database, or an overpass query for OpenStreetMap). The spatial features in the geographic data sources are available to the trajectory annotator as a named list that can be used in the predicates.

The *MWM Information Provider* retrieves the information from the MWM system. The minimum information we expect to retrieve from the MWM system is described in the conceptual model of Figure 2. This module is the only one that must be adapted for different MWM systems from different vendors. In some cases it may be a simple proxy that retrieves information from the web services provided by the MWM system, but in other cases it may be a complex module that retrieves information from the MWM data repositories.

The *Trajectory Annotator* module uses the raw data collected from the device sensors, the information retrieved from the MWM system, the context geographic information, and the activity taxonomies to annotate the trajectories of the workers and store them in the *trajectory database*, as described in Section 2.

Finally, the *Analysis and Report* module uses the annotated trajectories to provide useful information to the company managers, such as delays on the schedules, that can be used to improve future schedules.

5 Conclusions

We have presented in this paper a new methodology to turn raw data collected from the sensors of mobile devices into trajectories annotated with semantic activities of a high level of abstraction. The methodology is based on the concept

of *activity taxonomies* that make the system is highly flexible because they can be adapted easily to the needs of any company. Furthermore, the activity taxonomies describe the expected values for each of the variables that are collected in the system using predicates defined in a *pattern specification language*, which is very expressive and takes into account not only the raw sensor data but also data retrieved from a *MWM system*, and from *domain-related context geographical information*. Finally, we describe the functional architecture of a module that can be easily integrated in MWM systems and in the workflow of any company.

As future work, we are finishing the implementation of all the modules and we plan to do a full-fledged experimental evaluation with two real companies in the context of a research project.

References

1. Baglioni, M., Macedo, J., Renso, C., Wachowicz, M.: An Ontology-Based Approach for the Semantic Modelling and Reasoning on Trajectories, pp. 344–353. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
2. Bayat, A., Pomplun, M., Tran, D.A.: A study on human activity recognition using accelerometer data from smartphones. Procedia Computer Science 34, 450 – 457 (2014), the 11th International Conference on Mobile Systems and Pervasive Computing (MobiSPC'14)
3. Bogorny, V., Kuijpers, B., Alvares, L.O.: St-dmql: A semantic trajectory data mining query language. International Journal of Geographical Information Science 23(10), 1245–1276 (2009)
4. Güting, R.H., Valdés, F., Damiani, M.L.: Symbolic trajectories. ACM Trans. Spatial Algorithms Syst. 1(2), 7:1–7:51 (Jul 2015)
5. Ilarri, S., Stojanovic, D., Ray, C.: Semantic management of moving objects. Expert Syst. Appl. 42(3), 1418–1435 (Feb 2015)
6. Parent, C., Spaccapietra, S., Renso, C., Andrienko, G., Andrienko, N., Bogorny, V., Damiani, M.L., Gkoulalas-Divanis, A., Macedo, J., Pelekis, N., Theodoridis, Y., Yan, Z.: Semantic trajectories modeling and analysis. ACM Comput. Surv. 45(4), 42:1–42:32 (Aug 2013)
7. Pew Research Center: Smartphone ownership and internet usage continues to climb in emerging economies. `http://www.pewglobal.org/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies/`, (Accessed on 12/12/2016)
8. Read, J., Žliobaitė, I., Hollmén, J.: Labeling sensing data for mobility modeling. Information Systems 57, 207 – 222 (2016)
9. Rehman, M.H.u., Liew, C.S., Wah, T.Y., Shuja, J., Daghighi, B.: Mining personal data using smartphones and wearable devices: A survey. Sensors 15(2), 4430 (2015)
10. Spaccapietra, S., Parent, C., Damiani, M.L., de Macedo, J.A., Porto, F., Vangenot, C.: A conceptual view on trajectories. Data Knowl. Eng. 65(1), 126–146 (Apr 2008)
11. Yan, Z., Spremic, L., Chakraborty, D., Parent, C., Spaccapietra, S., Aberer, K.: Automatic construction and multi-level visualization of semantic trajectories. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 524–525. GIS '10, ACM, New York, NY, USA (2010)