

$\mathcal{PI}$: Perceiver and Interpreter of Smart Home Datasets

Juan Ye, Graeme Stevenson, Simon Dobson
School of Computer Science, University of St Andrews, UK

Michael O'Grady, Gregory O'Hare
CLARITY, University College Dublin, IE

Abstract—Pervasive healthcare systems facilitate various aspects of research including sensor technology, software technology, artificial intelligence and human-computer interaction. Researchers can often benefit from access to real-world data sets against which to evaluate new approaches and algorithms. Whilst more than a dozen data sets are currently publicly available, their use of heterogeneous mark-up impedes easy and widespread use. We describe $\mathcal{PI}$ – the Perceiver and semantic Interpreter – which offers a workbench API for the querying, re-structuring and repurposing of a range of diverse data formats currently in use. The use of a single API reduces cognitive overload, improves access, and supports integration of generic and domain-specific information within a common framework.

Index Terms—Smart home, pervasive healthcare, context modeling, activity recognition

I. INTRODUCTION

With the rising ageing population, there is a pressing need to design and develop technologies that can assist independent living of elderly people. The introduction of sensing technology into everyday environments and appliances has given rise to so-called ‘smart spaces’, where sensor data is interpolated to infer user behaviours and activities, allowing services to provide pro-active ambient assistance based on user context.

Data sets are essential to various avenues of research within pervasive healthcare environments. They provide, for example, a basis for assessing activity recognition algorithms. However, collecting a high-quality data set is a difficult task. Although the monetary cost of sensors is decreasing, such technology is not yet available to all. A substantial amount of effort and expertise is required to plan the instrumentation of a space, carry out the installation, and set up the hardware and software infrastructure responsible for storing, processing, and providing access to the collected data.

Not every research group has access to the resources (either for time, money, space, or person constraints) to carry out these tasks. However, a number of projects have made their data sets publicly available. Many of these data sets exhibit commonalities in the types of sensor data collected, and in the nature of user activities they capture. Yet, as these data sets were developed in isolation, and stored using *ad-hoc* data structures, these similarities cannot be exploited without the researcher first adapting their tools and techniques to each. As users of these data sets, we have been frustrated by this problem when evaluating our activity recognition techniques. These issues motivate the work described in this paper, which develops a Perceiver and Interpreter, called $\mathcal{PI}$, that allows developers to work uniformly with different smart home data sets. Our contribution includes:

- a standard taxonomy and structural representation that captures sensor data along with metadata associated with sensors and sensor types;
- a uniform programming interface to access, interact with, and study currently available data sets;
- a middleware service to interpret and fuse sensor data into high-level domain specific data, which hides low-level details (such as sensors or network) from application developers.

The rest of this paper is organised as follows. Section II motivates our work by describing research in smart home environments and existing data sets. Section III describes the conceptual model of $\mathcal{PI}$, including the model of data produced from sensors, diary data (annotation of users’ activities during the collection), and metadata of sensors and user activities. Section IV describes the implementation of $\mathcal{PI}$ in terms of its core functions: interpreter, query, synchroniser, context converter, and output components. Section V demonstrates the use of $\mathcal{PI}$ and discusses our experience of using it. Section VI compares our work with related sensor specifications and middleware from the literature. Finally, we conclude our work and summarise our future directions in Section VII.

II. STUDYING SMART HOME

Smart homes facilitate various aspects of research for a large number of research communities [4]. By analysing sensor readings, sensor developers may improve their designs; robotics researchers may construct algorithms to take actions in the face of an imperfect world view; and HCI designers may improve user experiences by making interactions less intrusive. Researchers working in artificial intelligence and context-aware systems study sensor data to design algorithms for activity recognition, allowing services and applications to work at a higher level of abstraction that is impractical with raw sensor data. Psychologists, sociologists, ethnographers, anthropologists, and health-care professionals use annotated diary data to study temporal patterns, behavioural routines, and interaction models of human activities. Engineers too, need to accumulate knowledge on the technical specifications of different sensors and environments to which they are best suited. The study of existing data sets supports all these groups in their analysis.

Whilst it is desirable to make data sets available to researchers working in these and related domains, their construction is not a straightforward process. A suitable environment must first be found (which may require the use of a normal residence), sensors to instrument the environment must be carefully selected and purchased, and resources need to be

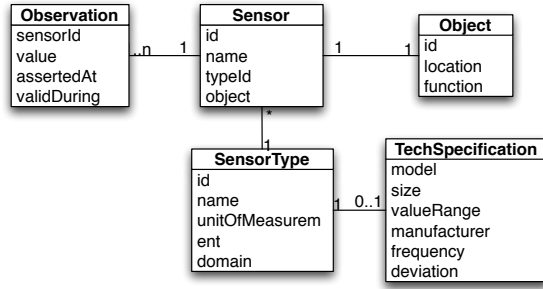


Fig. 1. The sensor and domain context models

allocated to recruiting external participants with varied age and background conditions for the collection. To make a data set more useful, a ground truth – the true state of participants and environments – needs to be recorded and added as annotation. All these processes require significant effort and investment, both in terms of time and money, in order to collect data for a meaningful set of activities or events [6]. The ability of researchers to share and reuse data sets is therefore of paramount importance.

The CHI’09-hosted workshop¹ on developing shared home data sets to advance HCI and ubiquitous computing research compiled a list of twenty data sets collected by research institutes across the world, half of which are publicly available to date. These data sets are heterogeneous in that they have been collected with different goals in mind. Each goal type plays a role in determining the collection methodology and focus of a data set. Heterogeneity also exists across the types of sensors used in the collection process; some of which are manufactured by the hosting research institute. This can result in encoding bias where data representation choices are made for their own convenience of notation or implementation.

To mask the heterogeneity of data sets and to make the data easily available to those performing different forms of research, we present an unifying model and an API to interpret various data sets and provide a uniform interface for working with them.

III. CONCEPTUAL MODEL OF $\mathcal{PI}$

Underlying $\mathcal{PI}$ is a conceptual model that defines a standard taxonomy to represent sensor data, diary data, and metadata covering the profile of sensors and sensor types, and activities. This taxonomy is rich enough to describe all the data sets we have encountered.

A. Sensor Model

A sensor model consists of representations of sensor types, sensors, and sensor readings (see Figure 1). Each sensor type has a set of properties including its *name*, *id*, *domain* – a parameter that describes what the sensor type measures (e.g., acceleration or usage of gas), and a *unitOfMeasurement* – the unit used to report observed values.

Each sensor type can be associated with a technical specification that includes its *manufacturer*, *model*, *size*, *deviation* of readings produced by its sensors (e.g., ‘a maximum precision of $\pm 1.5\%$ full scale’ for a gas flow sensor in the PlaceLab data set), its sampling *frequency*, and a number of *valueRange* parameters – boundary values that characterise different states; e.g., a gas flow is present when the sensor reports a value greater than 1. These values may come from the manufacturer or through the application of learning techniques.

Properties associated with a sensor type provide general information about all sensors that share this type, whilst properties associated with an individual sensor provide details specific to it [12]. The sensor specific properties consist of a *name*, *id*, *typeId*, installation *location*, and *object* to which it is attached (e.g., a cup, or a seat). Each sensor can also be associated with a quality matrix that describes its operating parameters like the deviation parameter mentioned above. These qualities can be derived from the technical specification of its sensor type, or can be refined in the presence of interference from an installation environment (such as temperature, humidity, or electromagnetic interference).

Sensor data is a collection of sensor readings and each reading can be represented as a combination of a literal *value*, a *sensorId* identifying the provider of the data, and a *startTime* and an *endTime*, during which the reported value is valid.

B. Domain Context Model

A domain context model describes the common concepts in a domain, such as environmental information (temperature, humidity, or noise level), location, acceleration of a body, biomedical information (heart rate or blood pressure), and so on. Figure 1 shows how we represent a domain context as a combination of six properties: a *startTime*, *endTime*, *location*, *object*, and *domain* where the *DomainValue* applies. Each domain value is represented as the combination of a literal value and a unit of measurement; for example, a flow of gas to a stove can be described as a numerical value 1.5 with a unit ‘gallons per minute’. The domain context is considered to be the value of an attribute of the specified object in the given location during the reported time period.

Using domain context, developers can access the data in a smart space by referring to an object and its attributes as opposed to readings from specific sensors. For example, a sensor ‘5B00000053C01E26’ monitors gas flow to the stove. Instead of referring to the output from that specific sensor, developers can refer to an abstract view of its readings by querying for the ‘GAS’ domain of the *Stove* object.

Another advantage of this approach is that rules defined on domain concepts can be uniformly applied to context data translated from different data sets. To serve application developers, we provide this higher-level abstraction by automatically translating sensor data into its corresponding domain context (see Section V-B).

C. Diary Model

The diary model captures the general structure of diary entries along with meta-information describing the method

¹<http://boxlab.wikispaces.com/List-of-Home-Datasets>

used for diary construction. This includes *recordMethod* – the capture process, such as a video recording or self-reporting; *translatedBy* – the researcher, third party, or software that translated the diary to this format; and *typeOfActivities* – the categories of activities captured, e.g., ranges of human motion, domestic activities, or complex interleaved activities involving multiple tasks or participants.

Each diary entry details the *participant* involved, a *startTime* and *endTime* describing the period covered, an *activityId*, and a *note* parameter, which is used to capture any additional information (e.g., interaction between the participant with another subject; or, during a ‘using computer’ activity, to indicate whether a laptop or desktop computer is being used).

Each activity is modelled as an *id*, *name* – a descriptive name for the activity, *superActivityId* – a higher-level activity category that the current activity belongs to, and *description* – an extended description on how the activity is characterised, for example a description of ‘watching TV’ can be ‘sitting in the couch and actively watching TV while not participating any other activities’. The parameter *superActivityId* is used when activities are modelled using a structured hierarchy of symbolic names. For example, the PlaceLab data set [7] classifies activities into several categories such as ‘meal preparation’, under which there are more specific activities like ‘retrieving ingredients’ or ‘mixing/stirring food’. The hierarchical approach can support analysis of activity recognition across multiple levels of granularity. Under this circumstance, the parameter *activityId* in a diary reading should be the identity of the activity that is the finest-grained to characterise the current scenario.

At the current stage, the diary model supports representing diary readings and activities, while it has not provided a generic taxonomy for activities. Activities to be captured and the way to annotate them can be determined by various factors including different goals of research (e.g., evaluating particular sensors or a generic sensor system), different types of environments where data sets are collected (e.g., home or office), and sensors to set up an environment (e.g., sophisticated sensors to detect fine-grained activities, or simple binary-state sensors). It is challenging to define such a uniform taxonomy that covers all the interesting activities.

IV. IMPLEMENTATION OF $\mathcal{PI}$

$\mathcal{PI}$ is implemented using a combination of Java and JDBC. This section introduces its main functions and demonstrates their use: (1) an *interpreter* that converts data sets to the uniform model and persists them to a database; (2) a *synchronisation component* that integrates data from multiple sources; (3) a *domain context converter* that translates sensor-specific data into a domain-specific data; (4) a *query component* that support traditional queries by time, sensor types, and sensor ids, in addition to advanced queries by location, object, and repeated temporal period; and (5) an *output component* that supports the export of data to standard formats such as RDF (Resource Description Format) and CSV (Comma Separated Values).

A. Interpreting Data Sets

To date, each data set we have encountered has had a unique encoding, and, in some cases, multiple formats are used within a single data set. A common problem is the lack of standardisation in selected data formats. Using time as an example, a portion of an entry from the PlaceLab data set contains ‘Time: 1156370400271 08/23/2006 18:00:0’, while an entry from the TK26M data set contains ‘25-Feb-2008 00:20:14’. Instead of writing a translator for each data set or for each sensor, we provide a generic translator class *Interpreter* (as shown in Figure 2) that is used to translate all types of data across different data sets into our representation format. The *read()* method of the *Interpreter* class accepts the following parameters: the property file containing information about a target database in Figure 2(b), the type of the target file to be interpreted, the target file, a set of properties describing the structure of the target files in Figure 2(c), and a name, which is used to set up a new table in the database in Figure 2(d). We will explain these parameters one by one.

```
read(databaseProperty, typeOfFile,
    fileToInterpret, propertyFile,
    newTableName);
```

A database property file consists of access information (i.e., username, password, database url, and driver), and a list of tables’ names (managed automatically by the software) where the sensor diary, activity data and metadata are stored. All the property files mentioned in this paper use JSON². As we shall see later, these lists of the tables play a role in querying and generating domain context.

The target file type indicates whether it stores sensor or diary data, which determines the interpretation method and the table schema to be applied.

A sensor or diary property file consists of *regex* – the regular expression used to parse a sensor reading into groups, *mapper* – the attribute names for each of the parsed groups in a reading, *dateFormatter* and *timeFormatter* – the formats used to represent date and time information in the readings, and *valueType* – the type of values generated. Most sensors produce numerical values, while some produce a numerical array (such as acceleration data or coordinate points) and others produce no value; e.g., once a RFID (Radio-frequency identification) tag is read by a reader then it means the tag-attached object has been accessed by the participant wearing the reader. To keep the representation uniform, we use a single attribute *value* to host all the types of values. If a sensor value is in a numerical array, we represent it as a string (e.g., ‘269, 261, 279’) and use post-processing to analyse the data. For sensors that do not produce values, we set their *value* to be 1 at the time point where the sensor id is read from the sensor data file. The reason that we do not provide more customised representations for different types of values is to simplify the interpretation interface.

The file to be interpreted must be readable. If it is a binary file (.b) or a matlab (.m) file, it needs to be decoded to a

²JSON - JavaScript Object Notation: <http://www.json.org/>

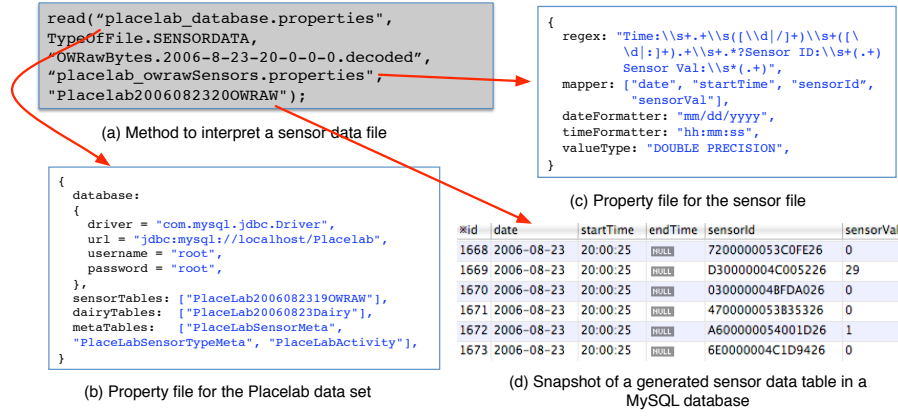


Fig. 2. An example of interpreting an OW RAW sensor file in PlaceLab data set

readable file first. We have collected the methods for some data sets and developers can use them to decode the binary or matlab files; e.g., the Java file ‘MITesDataAnal.java’ is provided to interpret the binary files in the PlaceLab data set.

Beyond sensor and diary data, the Interpreter class also supports interpretation of the metadata file describing sensor types, sensors, and activities. For example, readMeta() reads the property file that describes the metadata about individual sensors and generates a table named ‘SensorMeta’. This property file can be written by developers to describe the features of a sensor, as shown in Listing 1. Where data sets have provided this information for a large amount of sensors using a different format (e.g., the XML encoded files in the PlaceLab data set), we have provided a method to translate between the two.

Listing 1 A sample sensor metadata property file

```
type: sensorMeta,
mapper: ["sensorId", "location", "object"],
number: 14,
number_0: ["1", "kitchen", "microwave"],
number_1: ["5", "toilet", "Hall-Toilet door"],
...
```

B. Synchronisation

The process of synchronisation allows us to integrate data from multiple sources according to some criteria. *PI* supports synchronisation of sensor data across multiple time intervals (e.g., to obtain a snapshot of the environment every 10 seconds), and synchronisation with diary data records, which aids the evaluation of activity recognition algorithms.

The synchronise() method takes as parameters a database property file, a start and end time to select data, a temporal gap to slice the data sequences, a list of tables to synchronise, a function to aggregate sensor values collected during a given time slice, and a name for a new table to be created. Currently, we support simple aggregation functions including *AVG*, *MAX*, *MIN*, and *LATEST*.

```
synchronise(databaseProperty,
```

```
startTime, endTime, gap,
tablesToSynchronise,
aggregateFunction,
newTableName);
```

Figure 3 shows part of the synchronisation of the sensor and diary table in the TK26M data set on 25th February 2008 using a time slice of 60-seconds and a *LATEST* function that selects the latest reading for each set of sensor values during this slice. The ‘activityName’ column represents the activity occurring in the given period and all the following columns represent values produced by the sensors that are attached to the given objects. The synchronised diary and sensor data are useful in observing the correlation between the sensor data and the activities. For example in Figure 3, the sensor pattern for the activity ‘go to bed’ (in the blue box) is that there exists no sensor activities, while the sensor pattern for the activity ‘use toilet’ (in the orange box) is that the firings of the sensor on the toilet flush, the bathroom door, and the toilet door. The preliminary view presented in an activity-sensor synchronised table can help developers in observing such simple correlations and diagnosing sensors.

C. Converting to Domain Context

To hide the details of individual sensors from application developers, we map all sensor data to its associated domain context (see Figure 1), by taking a database property file, a target sensor data table to convert, and a name for a new table to create.

```
convertToContext(databaseProperty,
tableToConvert, newTableName);

convertToContext("tk26m_database.properties",
"TK26MSenseData", "TK26MContext");
```

The above example shows how to convert the sensor data into the ObjectAccess context, for the sensors in the TK26M data set are state-change sensors; that is, they are fired once their attached objects are accessed by the participant. Given the table ‘TK26MSenseData’, this method reads the sensor type and sensor metadata tables from the database, executes the join query on these tables, and stores the result

startTime	endTime	activityName	Toilet flush	Bathroom_door	Toilet_door
25/02/2008 00:22	25/02/2008 00:23	go to bed	NULL	NULL	NULL
25/02/2008 00:23	25/02/2008 00:24	go to bed	NULL	NULL	NULL
25/02/2008 09:37	25/02/2008 09:38	use toilet	1	1	1
25/02/2008 09:38	25/02/2008 09:39	use toilet	1	1	1
25/02/2008 09:39	25/02/2008 09:40	use toilet	1	1	1
25/02/2008 09:40	25/02/2008 09:41	use toilet	1	1	1
25/02/2008 09:41	25/02/2008 09:42	use toilet	1	1	1

Fig. 3. A snapshot of a table that synchronises diary and sensor data in the TK26M data set.

in a new table called ‘TK26MContext’. The underlying join query is generated as follows.

```
CREATE TABLE TK26MContext AS
SELECT TK26MSenseData.date,
       TK26MSenseData.startTime,
       TK26MSenseData.endTime,
       TK26MSenseMeta.location,
       TK26MSenseMeta.object,
       TK26MSenseTypeMeta.domain,
       TK26MSenseData.value,
       TK26MSenseTypeMeta.unitOfMeasurement
FROM TK26MSenseData, TK26MSensorTypeMeta,
     TK26mSensorMeta
WHERE TK26MSenseData.sensorId
      = TK26mSensorMeta.id
      AND TK26MSenseMeta.typeId
      = TK26MSensorTypeMeta.id;
```

D. Querying

The query component supports both basic and semantic queries. In the basic querying mode, developers can filter sensor data by specifying the database property file for the required data set, the corresponding sensor ids or sensor type ids, an absolute or repeated time period, and a name for a new table to store the query result if necessary.

```
querySensorBasic(databaseProperty, typeIds,
                 sensorIds, startTime, endTime, newTableName);
```

This type of query is useful for system engineers to observe the performance of certain sensors and debug sensors. The following example queries for sensor data, whose sensor id is either 24 or 10, and whose valid time is between 6 o'clock and midnight on 25th February 2008.

```
querySensorBasic("tk26m_database.properties",
                 "24, 10", "2008-02-25 18:00:00",
                 "2008-02-25 23:59:59");
```

For developers who do not have (and do not need) knowledge about the sensors in a data set, we support semantic queries that allow them to search sensor data within a certain domain or location, or associated with particular objects.

```
querySensorSemantic(databaseProperty, locations,
                   objects, domainTypes, startTime,
                   endTime, newTableName);
```

The following example queries for the gas usage in the kitchen from six to eight o'clock every day. The query result is materialised in a new table called ‘gasUsage’. Using the metadata about sensor types and sensors, semantic queries are

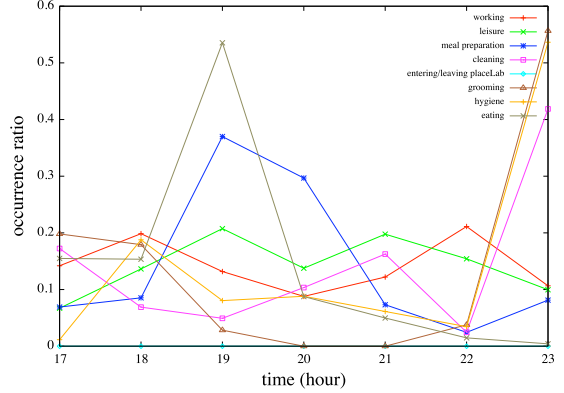


Fig. 4. An example using $\mathcal{PT}$ to produce a Gnuplot format of the occurrence ratio of different activities at different times of a day in the PlaceLab dataset.

translated into basic queries by mapping the domain type to sensor types and locations to particular sensors. The repeated temporal period allows developers to compare sensor data for a fixed duration over a number of days.

```
querySensorSemantic(
    "placelab_database.properties", "kitchen",
    "gas", "18:00:00", "20:00:00", "gasUsage");
```

Each query method compiles the user supplied requirements into an SQL statement that is used to examine the corresponding database. The queries provided by this API do not support complex logic; for example, a query for sensor readings whose sensor type id is ‘17’ and whose sensor id is not ‘1500000022CFF412’. As we have not yet included primitives for such expressions, we provide a method that allows developers to write their own custom SQL statements.

E. Output

The `output()` method supports the export of data from the database into several standard formats: RDF or XML, which can facilitate sharing and reuse of sensor data across data sets and systems as well as abstracting to higher-level domain context as seen in Section V-B; formats for standard systems or tools, such as CSV that is supported by spreadsheets and other database management systems, and WEKA machine learning software, commonly used for activity recognition [10]; and Gnuplot-format for the Gnuplot statistical tool to analyse and visualise sensor data (See in Figure 4 (a)).

The method takes as input a database property file, a table to

export, an output format, a property file to configure the output, and a destination file. A property file to configure an output vary with its output format, which can the head setup of these files (as for CSV and WEKA-format), a script configuring a plot (as for R- or Gnuplot-format), or a mapping between a schema of the current table to export and a schema of the target ontology or XML (as for XML or ontology).

```
output(databaseProperties, tableToExport,
       outputFormatType, outputProperty,
       destinationFile);

output("tk26m_database.properties",
       "TK26MSensorData", OutputFormats.RDF,
       "kasterenSensorData.n3");
```

The above example shows how sensor data from the table ‘TK26MSensorData’ is exported as an RDF file. This method automatically reads the sensor metadata from the database, and converts each result to the following format (The file is represented in Notation 3³):

```
:sensordata1 a pi:SensorData;
pi:startTime
  "2008-02-25T00:20:14"^^xsd:dateTime ;
pi:endTime
  "2008-02-25T00:22:57"^^xsd:dateTime ;
pi:sensorId "24"^^xsd:String ;
pi:value "1"^^xsd:int .
```

V. DEMONSTRATION AND EVALUATION

This section presents two case studies that utilise *PI* and discusses its strengths and limitations.

A. Activity Recognition Research

Activity recognition researchers need to synchronise sensor data with diary data to explore the relationship between sensor values and the occurrence of activities. For example, consider the process of learning the relationship between the usage of gas and the cooking activities in the PlaceLab data set, where we wish to synchronise the relevant data and output it to a standard format, e.g., CSV. The process is carried out in code as follows.

Listing 2 Showing how CSV-formatted synchronised sensor and diary data for activity recognition research is output.

```
1. querySensorSemantic(
2.   "placelab_database.properties", "kitchen",
3.   "gas", "18:00:00", "20:00:00", "gasUsage");
4. queryDiary("placelab_database.properties",
5.   "meal preparation", "cooking");
6. synchronise("placelab_database.properties",
7.   "cooking, gasUsage", 60000,
8.   Aggregator.AVG,
9.   "CookingGasSynchronised");
10. output("placelab_database.properties",
11.   "CookingGasSynchronised",
12.   OutputFormats.CSV,
13.   "CookingGasSynchronised.csv");
```

Lines 1-3 query the gas readings from the sensor tables, while lines 4-5 query the meal preparation activities from the

diary tables. Lines 6-9 synchronise these two results, where the aggregated gas reading per minute is the average of the collected readings in that period; and Lines 10-13 output the synchronised table in CSV format. In all, this process requires 4 method calls.

B. Working with Higher-level Contexts

Context-aware researchers may need to generate higher-level context from low-level sensor data. As we model domain context independently of sensors, high-level context may be generated by writing rules against domains rather than specific sensors or data sets. In this section, we demonstrate how context information from a gas flow sensor is added to an existing domain ontology – Ontonym [9]. We then show how an application rule base is used to derive high-level situations from these readings. The process is similar for other types of sensors.

The first step is to convert the gas readings into domain context and then add them to the ontology. In this example, *PI* generates the gas usage context from the relevant sensor data tables, following the target ontology schema.

Listing 3 Generating context to a target ontology

```
1. convertToContext(placelab_database.properties,
2.   "gasUsage", "gasContext");
3. output(placelab_database.properties,
4.   "gasContext", OutputFormat.RDF,
5.   "ontonymMapping.properties",
6.   "gasContext.trig");
```

Lines 1-2 convert the sensor data from the gas flow sensors (the result of the example in Listing 2), while lines 3-6 export the converted context using the Ontonym ontology, where the property file ‘ontonymMapping.properties’ records the mappings of attributes, their data types and the units of measurement between the context table and the ontology.

The output of this conversion is an Ontonym ontology individual as shown in Listing 4.

Listing 4 Context converted from a gas flow sensor reading

```
#http://example.com/gasEvent121
:gasEvent121 a sensor:Observation ,
  sensor:startTime
    "2006-09-14T18:48:39"^^xsd:dateTime ;
  sensor:value "2.16"^^xsd:double ;
  sensor:domain "GAS"^^xsd:string ;
  sensor:object :Stove ;
  sensor:location "kitchen"^^xsd:string ;
  muo:measuredIn ucum:gallons-per-minute .
```

The majority of sensor readings in existing datasets are simple numerical values like the gas readings above. The exception to this is the use of coordinates by positioning sensors or three-axis accelerometer readings. *PI* distinguishes different categories of value according using the property ‘sensor:domain’. As seen in Figure 1, the domain value in a context is determined by its sensor type. When the domain is set as ‘acceleration’, a particular method is called to analyse sensor

³Notation 3 is a compact RDF syntax: <http://www.w3.org/DesignIssues/Notation3>.

values for the given type. For example, the PlaceLab dataset contains acceleration values taken from an accelerometer worn on the right wrist of the participant; this is converted to the output in Listing 5.

Listing 5 Context converted from an accelerometer sensor reading

```
#http://example.com/accEvent587
:accEvent587 a sensor:Observation ;
sensor:startTime "2006-08-23T23:00:00"^^xsd:dateTime;
sensor:value [ a acceleration:Acceleration;
               acceleration:x "97"^^xsd:double;
               acceleration:y "99"^^xsd:double;
               acceleration:z "101"^^xsd:double ];
sensor:domain "Acceleration"^^xsd:string;
sensor:location "Dominant wrist"^^xsd:string.
```

After translating sensor data into the context ontology, we can use a rule base to reason over the ontology. The following rule in Listing 6 states that if a reading with the ‘GAS’ domain and the *Stove* object has a value that exceeds 1 gallon per minute, then create a ‘stoveOn’ classification.

Listing 6 A rule to abstrac gas sensor data into higher-level context

```
[stoveONRule:
  (?A sensor:domain "GAS"), (?A sensor:value ?B),
  (?A sensor:object ?C), (?C rdf:type :Stove),
  ge(?B "1"^^xsd:double),
  -> createClassification(?A, "stoveOn")]
```

This rule is triggered by a gas event described above and as a result a new classification is generated, as shown in Listing 7.

Listing 7 Higher-level context converted from the gas event in Listing 4

```
#http://example.com/gasClassification
:gasClassification a :Classification ;
:classifiedValue "stoveOn" ;
:createdAt "2006-09-14T18:48:39"^^xsd:dateTime ;
provenance:derivedFrom :gasEvent121 .
```

By following this process, rules can be codified for any domain, sensor type or specific sensor, and can be as simple or complex as required. Little effort is required to create a new classification rule and by providing rules for generating high level context in a domain, the complexity of application code can be greatly reduced. In addition, the property *provenance* can help to trace the derived context back to individual sensors. On the one hand, this property can indicate the quality (or reliability) of derived context when context from different sources need to be merged. On the other hand, this property can also be helpful in locating faulty or malfunctioning sensors when conflicting or incorrect contexts are detected. Although this is a simple example, a more expressive provenance model⁴ may be applied if desired.

⁴PML2 provenance ontology: <http://iw.stanford.edu/2006/06/pml-provenance.owl>.

C. Discussion

In developing *PI*, we aim to meet three objectives: to provide a suite of tools that support analysis of sensor data, to remove obstacles from the process of working with published data sets, and to encourage future data set developers to publish their data in a *PI* friendly format. We aim for *PI* to become a useful tool to researchers working in a wide range of areas related to smart homes.

1) *Expressiveness of the Model*: We believe the data model that underlies *PI* to be comprehensive in capturing the core aspects crucial to sensor data analysis. As well as capturing the values observed by sensors, we support the modelling of meta-information about environments, sensors and types of sensors, diary data, activities, and profile information describing the methodology used to capture the data. All of these may play a role in the analysis process. We also provide support for domain contexts, allowing us to abstract away from the raw sensor values and capture the semantics of the data sensed. Mappings to higher-level context allow us to represent information in both human friendly and application specific terms rather than using the raw observation values.

Our conceptual model forms a superset of entities and properties over all published data sets we have studied, and may be simply extended (e.g., to add new forms of meta-information about a sensor), without requiring any modifications to application code.

2) *Analysis Support*: *PI* provides powerful support for analysing sensor data in the form of its synchronisation and query APIs. The synchronisation API has two core features. Firstly, it allows data to be segmented into chunks of a specified duration, making it easy, for example, to sample the state of an environment every 5 seconds over a time period. Secondly, it allows data to be synchronised around the occurrences of activities if diary data is available. Both these operations make it easy for researchers to filter out or summarise data as part of their analysis process.

After synchronisation, the API supports querying the resultant data. In addition to low level queries that work with sensor and sensor type ids, the more advanced features of the query API allow the user to query data at a higher level of abstraction (i.e., by filtering on location, entity, or type of domain context). Beyond this, we allow developers to incorporate SQL into the querying process (where complex queries are required) and allow the API to be extended where developers wish to execute more significant jobs (such as the customised queries mentioned in Section IV-D).

After the querying process is complete, the results can be easily exported to a variety of standard data formats, making it easy to slot *PI* into a workflow that incorporates other analysis techniques or visualisations.

3) *Developer Effort*: Using *PI* to work with data sets is a straightforward process. The main functions to load and analyse data sets are provided in interfaces in each function package, and as shown in the above examples, few method calls are required to perform reasonably complex tasks, such as synchronising data against the occurrence of activities,

and translating the raw values in a data set into higher-level context, ready for further processing.

A certain amount of engineering effort is required of developers who wish to introduce new data sets. This includes setting up a database, and configuring property files that instruct $\mathcal{PI}$ how to process the data. With the possible exception of writing the regular expression to parse the sensor readings, this process is straightforward (if slightly time consuming). As we have written property files for most of the data sets currently available (such as the PlaceLab, TK26M and CASAS [3]), there are plenty of examples to work from. Developers who know of the tool and wish to construct new data sets can bear these issues in mind as they do so.

VI. RELATED WORK

We compare our work with the literature from two perspectives: sensor specifications and middleware. Researchers at the University of Florida propose a Sensory Data Set Description Language (SDDL), which is an XML-encoded description language for sensor data originating from pervasive spaces [6].

The scope of SDDL is to specify information about the pervasive space, including available sensors/actuators, data set parameters and sensor events. The profile of a data set in our conceptual model is built on SDDL, where we have selected the properties that are significant to characterise data rather than a comprehensive set of properties. The other difference between their work and our work is that we not only provide the specification to describe different parts of a data set, but we also provide a framework to operate over the data model.

Another sensor specification language is *SensorML*, the Sensor Model Language, which is an XML encoding schema that aims to enable remote discovery, access, and usage of real-time data directly from web-hosted sensors [2]. The strength of SensorML is in describing processes, including their inputs, outputs, parameters, methods, and relevant metadata. In contrast, our conceptual model focuses on describing features of data rather than processes and is more lightweight.

The examples of middleware in pervasive computing literature are CoBrA [1], SOCAM [5], and Gaia [8]. The common features of these middleware are acquiring, maintaining, and reasoning about domain context. CoBrA (a Context Broker Architecture) aims to share knowledge, detect and resolve inconsistent knowledge, and protect user privacy by supporting common policy language. SOCAM (a Service-Oriented Context-Aware Middleware) enables the building and rapid prototyping of context-aware services. It abstracts various physical spaces from which contexts are acquired into a semantic space, where contexts can be easily shared and accessed by applications [11]. Gaia brings the functionality of an operating system to physical spaces. It employs common operating system functions (including events, signals, file systems, security, and processes), and extends them with location context, mobile computing devices, and actuators. Using this functionality, Gaia integrates devices and physical spaces, and allows the physical and virtual entities to seamlessly interact.

As it is not concerned with sensor network and application design, $\mathcal{PI}$ is not a middleware. However, it is a tool to translate heterogeneous sensor data into a universal representation and provide functions (such as query, output, and synchronisation) to access and analyse sensor data. Therefore it shares some of its goals with the above projects. In practice, $\mathcal{PI}$ can be employed as a layer between the sensor network and context-aware middleware; feeding sensor data to the middleware's context models.

VII. CONCLUSION AND FUTURE WORK

This paper introduces $\mathcal{PI}$, an API to perceive and interpret data sets in smart home environments. This work not only facilitates the sharing and reuse of existing data sets but also provides standard representations for sensor and diary data for future data set publishers.

An OWL implementation of the conceptual models is available from <http://ontonym.org>. We are currently completing the documentation and tutorials for $\mathcal{PI}$, before making it publicly available. To use $\mathcal{PI}$, developers need to download the library and incorporate API calls in their own code. We try to simplify the process of adding new datasets, and investigating more expressive queries that will enhance our API. In the long term, we seek to pursue the integration of $\mathcal{PI}$ with live sensor data, providing a set of tools that can be used to support real-time analysis of the characteristics of a smart environment.

REFERENCES

- [1] H. L. Chen. *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. PhD thesis, University of Maryland, 2004.
- [2] X. Chu and R. Buyya. Service oriented sensor web. In *Sensor Networks and Configuration*, pages 51–74. Springer Berlin Heidelberg, 2007.
- [3] D. Cook, M. S. Edgecombe, A. Crandall, C. Sanders, and B. Thomas. Collecting and disseminating smart home sensor data in the CASAS project. In *Proceedings of the CHI 2009 workshops*, Apr. 2009.
- [4] D. J. Cook and S. K. Das. How smart are our environments? An updated look at the state of the art. *Pervasive Mob. Comput.*, 3(2):53–73, 2007.
- [5] T. Gu, H. K. Pung, and D. Q. Zhang. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl.*, 28(1):1–18, 2005.
- [6] S. Hossein, S. Hedai, and A. Mendez-Vasquez. Sensory data set description language specification. Technical Report SDDL_Specification_v1.0, University of Florida, 2009.
- [7] B. Logan, J. Healey, M. Philipose, E. M. Tapia, and S. S. Intille. A long-term evaluation of sensing modalities for activity recognition. In *UbiComp 2007*, pages 483–500, Innsbruck, Austria, Sept. 2007.
- [8] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67, 2002.
- [9] G. Stevenson, S. Knox, S. Dobson, and P. Nixon. Ontonym: a collection of upper ontologies for developing pervasive systems. In *Proceedings of the 1st Workshop on Context, Information and Ontologies*, page 18, Heraklion, Greece, 2009. ACM.
- [10] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.
- [11] J. Ye, L. Coyle, S. Dobson, and P. Nixon. Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review*, 22:315–347, Dec. 2007.
- [12] J. Ye, S. McKeever, L. Coyle, S. Neely, and S. Dobson. Resolving uncertainty in context integration and abstraction. In *ICPS' 2008*, pages 131–140. ACM, July 2008.