Journal of Ambient Intelligence and Smart Environments 1 (2013) 1
IOS Press

Reasoning for Sensor Data Interpretation: an Application to Air Quality Monitoring

Marjan Alirezaie [a,*], Amy Loutfi [a,**]
[a] *Center for Applied Autonomous Sensor Systems (AASS), Dept. of Science and Technology*
Örebro University, SE-701 82, Örebro
Sweden

Abstract. In this paper we introduce a representation and reasoning model for the interpretation of time-series signals of a gas sensor situated in a sensor network. The interpretation process includes inferring high level explanations for changes detected over the gas signals. Inspired from the Semantic Sensor Network (SSN), the ontology used in this work provides an adaptive way of modelling the domain-related knowledge. Furthermore, exploiting (Incremental) Answer Set Programming (ASP) enables a declarative and automatic way of rule definition. Converting the ontology concepts and relations into ASP logic programs, the interpretation process defines a logic program whose answer sets are considered as eventual explanations for the detected changes in the gas sensor signals. The proposed approach is tested in a kitchen environment which contains several objects monitored by different sensors. The contextual information provided by the sensor network together with high level domain knowledge are used to infer explanations for changes in the ambient air detected by the gas sensors.

Keywords: semantic sensor ontology, non-monotonic reasoning, answer set programming, knowledge representation

1. Introduction

The ability to detect, quantify and describe odours is of ever increasing interest in a society with greater awareness of environmental sustainability and greater interest in promoting better quality of products and services to its citizens. Technologies which enable detection and identification of complex gases, so called electronic noses, have been on the market for some time and development of sensor techniques are increasing, thereby providing affordable and commercial solutions. With this suite of technologies available, a large segment of research in the field has often been focused on integration of these sensor technologies in other intelligent systems. Such systems may consist of a mobile system, such as a robot [19] or a distributed system of heterogeneous networked sensors, such as a smart home.

An advantage of using electronic noses in larger context of a sensor network is that the aggregation of information from the sensor network can be used to better assess and interpret the results from the gas sensors. This is useful as the continuous and multivariate time series (and unintuitive) readings provided from a gas sensing array involve a non-trivial process of feature extraction and learning methods to provide a classification/identification of a target gas or odour. By modelling high level knowledge about odours, their causes, and relations to other phenomenon, it is possible to assist the interpretation of the gas sensor signals. However, to automate this process, the high level (symbolic) knowledge needs to be seamlessly connected to the lower level sensor (quantitative) data. Furthermore suitable reasoning techniques are also required in order to infer information beyond that which is measured by the gas sensors or the sensor network.

This paper presents a knowledge driven approach to reasoning about changes detected over gas sensor signals in a sensor network. Automated reasoning is achieved via answer set programming (ASP). The mo-

*E-mail: marjan.alirezaie@oru.se
**E-mail: amy.loutfi@oru.se

tivation to using ASP is twofold. First, the proposed application should deal with incomplete data caused by uncertain behaviour of sensors or the lack of observations. Second, the dynamic nature of the sensor network should be taken into account where new observations can influence the reasoning process and if necessary invalidate previously inferred results. For these reasons, a non-monotonic reasoner is advocated and ASP has the further advantage of using an incremental solver which is suitable for stream reasoning [16]. To model the necessary high level knowledge for the reasoner, ontologies inspired by the Semantic Sensor Network (SSN) [4] are used. Such ontologies provide the basis for declarative definition of entities, features and events in the environment. Also, these ontologies enable a reuse of information, compatible with the emerging trend of linking contextual information to sensor data seen in the Internet of Things (IoT) paradigm [32]. However, the integration of ontology languages such as OWL-DL with non-monotonic reasoning is non-trivial. Therefore a contribution in this work is to implement a conversion between OWL-DL ontologies and the incremental ASP reasoner.

The proposed approach for reasoning about odours is validated on a smart kitchen sensor network. The network contains a heterogeneous set of sensors which also include gas sensors located in various parts of the kitchen. Monitoring of the kitchen occurs over several days and when a change point is detected, the reasoner provides explanations for the possible causes of the change. In this way, the gas sensor signal is annotated with high level descriptions. While the approach proposed in this paper focuses on the application of air quality monitoring, the method has been designed with the overarching goal to enable high level reasoning upon low level sensor data using non-monotonic reasoning methods and reusable sources of domain knowledge. This goal is important given the trend IoT towards a global connectivity between objects from the physical world. As more sensors of varying modality become connected, it will be of importance to provide automated interpretation of the sensor data and to exploit existing resources to this aim if possible.

This paper is organized as follows. In Section 2 we study the related work. A short introduction to Answer Set Programming and OWL-DL are given in Section 3 and 4, respectively. The essential components required for implementation of a smart environment are explained in Section 5. Section 6 describes the details of the reasoning process. The evaluation results of the proposed approach are discussed in Section 7 which is continued by the conclusion.

2. Related Work

Regardless of the type of chemical sensors in electronic noses, a large segment of the research in machine olfaction lies in sensor data analysis and studies have been conducted in a number of application areas including food quality analysis [21,22,24], medical applications [23,25] and air quality monitoring [26,27]. Since the main approach in these applications is based on data driven techniques, the focus of these research works lies in finding algorithms to model and learn sensor patterns based on previously seen samples of known odours. For a restricted subset of odours, good classification have been achieved, still yet a remaining challenge for electronic olfaction is to deploy the sensor technology in semi-structured and dynamic environments where little pre-training is done, and a large set of odours are possible to detect. Use of high level (symbolic) information to improve the classification performance has been previously considered in only a handful of works. Notable examples include [18] in which an ontology alignment technique is used together with a decision tree in order to resolve misclassification by using symbolic information from the ontology.

Considering the more general problem of reasoning over sensor signals for time series data, several adjacent works are relevant for the methodology presented in this paper [14]. There are a few works such as symbol grounding [30], and anchoring [11] which have the common goal of abstracting sensor data into symbolic knowledge. Much of the research in this work is related to the robotics community and has concentrated on the issue of the bridging the gap between data and knowledge [6] and in specific cases using reasoning methods to reason about perceptions.

Another relevant subset of works deal with semantic web technologies, and the areas of the semantic sensor web, where works such as [12,13] propose to semantically annotate sensor data with semantic annotations published on the semantic Web (Semantic Sensor Web). In this way, interoperability between sensor networks and consequently the situation awareness increases [6]. In this domain, ontologies are the most popular method to achieve such structures [15]. For example, the Semantic Sensor Network (SSN) ontology is a suggestion from the W3C Semantic Sensor Net-

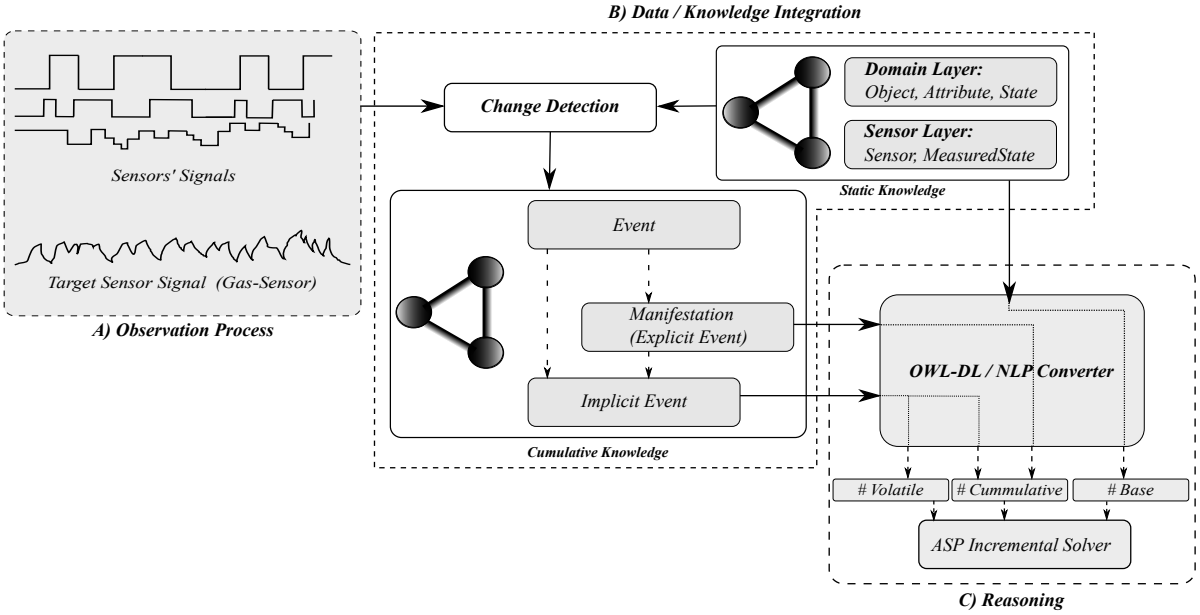

Fig. 1. Knowledge Representation and Reasoning Framework. (A) Sensors Observation Process in the Environment. (B) Domain Knowledge & Ontology Population. (C) Conversion/Reasoning process.

works Incubator Group (SSN-XG) [4] for representing the sensor data independent of the domain knowledge. The concentration of the SSN ontology is on sensors (and their capabilities), observations, and systems rather than the domain-dependent concepts.

Due to the expressivity of the ontology languages such as OWL-DL (explained in Section 4), definition of concepts, relations and axioms in the domain knowledge are declarative. In order to reason upon this knowledge, therefore, an expressive reasoning technique is required. All the deductive (monotonic) reasoners based on description logic (DL) implemented for ontologies, depending on their formalisations, provide different degrees of complexity and expressiveness [34]. Nevertheless, applications need to deal with the issue of incomplete data caused by inherently uncertain behaviour of sensors or the lack of observation. Furthermore, the dynamic behaviour of sensory systems where the newly added observation can influence the reasoning process such that the new reasoning outputs may contradict the previously inferred results [35], indicates the need for non-monotonic reasoning techniques. Recently, hybrid approaches to combining monotonic and non-monotonic reasoning have been proposed. For example, [31,36] combine an handmade ontology representing high level concepts and the ASP solver as a non-monotonic reasoner to reason over sensor data. These works illustrate the potential of ASP to cope with stream reasoning. In this paper, in addition to using an incremental ASP solver [16], we utilize an ontology synonymous to the SSN ontology and provide the means to automatically generate the logic used by the solver.

3. Answer Set Programming

Answer Set Programming is a declarative problem solving paradigm which in this work is used as the solver (reasoner). The basic idea of ASP is to describe a problem statement within a logic program composed of a set of facts, rules and constraints. The logical entailments of the program or answer sets [10], are considered as the solution of the problem statement. Answer set semantic which is based on the *stable model* semantic [2] is a declarative semantic that allows the logic programs to use the *default negation*. Due to the possibility of using the *default negation*, non-monotonic situations can be modelled in a logic program which facilitates reasoning upon incomplete information. Furthermore, making use of negation as a natural linguistic concept can increase the declarativity of the logic program [10].

A non-monotonic logic (normal logic) program *P* consists of a set of rules r_i of the form[1]:

$$r_i : a_0 \leftarrow a_1, ..., a_m, not \;\; a_{m+1}, ..., not \;\; a_n \quad (1)$$

where all a_i are atoms in a first order language and $m, n \geq 0$. A literal is an atom a or its negation $not\ a$. Each rule has its own head ($h(r)$) and body ($b(r)$) which are equivalent to the left (a_0) and the right side of the rule, respectively. The body of a rule is divided into two parts w.r.t the *negation as failure* (*not*) operator. The set of positive literals of r is $b(r)^+ = \{a_i \mid 1 \leq i \leq m\}$ and the rest of literals in the body belong to the negative body part of the rule, $b(r)^- = \{a_i \mid m < i \leq n\}$. A *fact* is a rule with an empty body ($n = 0$). On the contrary, *integrity constraints* are those rules with empty heads ($h(r) = \emptyset$). The rule r_i is intuitively saying that a_0 is derived if all the literals $a_1, ..., a_m$ are true while the truth of non of $a_{m+1}, ..., a_n$ can be inferred. Therefore, the answer set of the program *P* is the minimal set of atoms X if for each rule r_i, $a_0 \in X$ whenever $\{a_1, ..., a_m\} \subseteq X$ and $\{a_{m+1}, ..., a_n\} \cap X = \emptyset$. Calculation of an answer set *S* for the program *P* which is formally explained in below is computed by a program called ASP solver.

$$S = \{h(r_i) \mid r_i \in P \wedge b^+(r_i) \subseteq X \wedge b^-(r_i) \cap X = \emptyset\} \quad (2)$$

In ASP, rules can be extended so as to express further meanings. For instance, the *cardinality rule* [1] represented in (3), has its own specific syntax. The two numeric values l and u are respectively the lower and upper bound of the number of satisfied atoms in the body part. By making use of this type of rules, we indicate that the atom a_0 will be inferred ($a_0 \in S$), if and only if at least l and at most u atoms in $\{a_1, ...a_m\}$ are members of the answer set *S*.

$$r_i : a_0 \leftarrow l \; \{a_1, ..., a_m\} \; u \quad (3)$$

The non-monotone behaviour of ASP which is enabled by using the *negation as failure* (*"not"*) operator, specifies the possibility of having a set of conclusions being temporary true. More precisely, according to the non-monotonicity feature, we may have a set of conclusions which will not be concluded any more if new information is added. Using the *negation as failure*, the logic program representing the default knowledge, can also specify tentative exceptions. The defaults hold unless at least the truth of an exception is inferred. The ability of reasoning over the default knowledge along with the possible exceptions, as mentioned above, makes ASP a suitable paradigm dealing with incomplete knowledge [1].

3.1. Incremental ASP Solver

Although ASP has already been used for solving dynamic applications, the approaches taken in many attempts such as [28,29] are based on restarting the solving process from scratch whenever a change happens. For large scales programs, redoing the solving process after each change in the logic program is an insufficient approach [5]. The substitute approach can be an incremental solver. The incremental solver, instead of redoing the solving process, first solves the initial program and then incrementally updates the solving results based on accumulated effects of changes in the program [8]. These changes can be in form of extra information logically implemented.

According to the idea of incremental solving, a logic program $\mathcal{R}$ is defined as the combination of three different logic programs, namely the base ($\mathcal{B}$), cumulative ($\mathcal{P}[t]$) and volatile ($\mathcal{Q}[t]$) logic programs. The base $\mathcal{B}$ refers to the time independent logic program containing the static knowledge. Both the time dependent logic programs $\mathcal{P}$ and $\mathcal{Q}$ dynamically change w.r.t the parameter t showing the time step. The size of the program $\mathcal{P}$ which represents the cumulative knowledge increases by the growth of t. However, the content of $\mathcal{Q}$ always indicates the situation at the current time step. In other words, by changing the time step, the contents of $\mathcal{Q}$ are overwritten by the new content. Assuming that the time step has reached to the jth step, the content of the incrementally extended program $\mathcal{R}$ at step j is formally shown in (4).

$$\mathcal{R}[t_j] = \mathcal{B} \cup (\bigcup_{i=1}^{j} \mathcal{P}[t_i]) \cup \mathcal{Q}[t_j] \quad (4)$$

The incremental solver used in this paper is *oClingo* which is built upon the *iClingo* [9] ASP solver. This solver provides an incremental interface for updating the program ($\mathcal{P}[t]$) with extra knowledge added at subsequent time steps. The eventual answer sets of $\mathcal{R}[t]$

[1] In our implementation, the "implies sign" ($\leftarrow$) in rules is replaced by ":-".

will be considered as the solution for the problem statement logically modelled.

Given the structured knowledge (ontology) described in Section 5, the *logic program conversion* process converts the different parts of the ontology into a logic program solvable by the incremental ASP solver (See Fig. 1, Block *C*). The explicit events which are formed based on the dynamic and the static knowledge are also translated into the cumulative $\mathcal{P}$ and the static $\mathcal{B}$ logic programs, respectively. Once a change (event) detected in a gas signal, the volatile $\mathcal{Q}$ logic program triggers the solver to find an explanation for the event.

4. Knowledge Representation in OWL-DL

Description Logic (DL) is known as a family of logic based formal knowledge representation languages for modelling concepts, roles (relations) and individuals of a domain. Description Logic formalisms, in addition, allow us to define logical statements called axioms by relating concepts and/or roles together [3]. The DL semantics are founded on predicate logic (restricted first order logic), however, with good computational properties (e.g., decidability) for practical modelling.

The syntax of each member of the DL family is characterized based on their constructors. DL constructors which are related to logical connectives in first order logic are: intersection or conjunction ($\sqcap$), union or disjunction ($\sqcup$) , negation or complement ($\neg$), universal ($\forall$) and existential ($\exists$) restrictions [7]. The DL syntax of a class defined based on restrictions are as follows: For example, the axiom $\exists R.C$ is equivalent to all *things* that at least one relation (property) *R* connects them to the instances of class *C* and axiom $\forall R.C$ is equivalent to all *things* whose all *R* properties relate them to class *C*'s instances. DL semantics also allow us to mention cardinality restriction in terms of the number of properties between any two concepts. For instance, axiom $\geq nR$ where *n* is a positive integer, states all concepts that are at least related to *n* other things via the property *R*.

Furthermore, specialization of concepts in DL is achieved via creating subsumption relation ($\sqsubseteq$) between two concepts. For example, the axiom $C \sqsubseteq D$ means that concept *C* which is subsumed by concept *D*, is a specialized version of D.

Web Ontology Language (OWL) is an ontology language recommended by World Wide Web Consortium (W3C) for representing ontologies on the semantic web. OWL which is a logical knowledge model represents the knowledge within RDF/XML syntax.

In order to provide declarativity and reasoning power of DL to the semantic web, three different versions of OWL has been defined: OWL Lite, OWL DL and OWL Full [7,20]. In this work, data / knowledge are represented within an OWL-DL ontology which is based on description logic semantics and provides a high degree of expressivity.

5. System Architecture

Before detailing the integration between the reasoner and the OWL-DL ontology, this section provides a high level view of the system architecture used in the smart home environment. The architecture contains both the high level models and the reasoning components. The input is the sensor data from the sensor network and the output is the annotations that consist of the reasoner's explanations to changes detected in the gas sensor (target sensor). Figure. 1 illustrates the interactions among different parts of the implemented system. The system architecture consists of three blocks, namely *A) Observation Process*, *B) Data/Knowledge Integration* and *C)Reasoning*. In this section we describe the details of block *A* and *B*. Block *C* is separately described in Section 6.

5.1. Block A : Observation Process

The observation process of a smart home is performed through a set of heterogeneous sensors which are synchronously and continuously observing the environment. Each object in the smart home, depending on its attributes of interest can be observed by one or several sensors. Sensors include motion detectors, luminosity sensors, temperature sensors, magnetic contacts etc. Regardless of both the sensor and the observed object, there are two ways of handling sensor outputs either via continuous data sampling or via event capturing. In continuous data sampling, if s_i is a data sample (or sensor reading) at time step i, the sensor output is:

$output_{cds} = \{s_1, ..., s_i, s_{i+1}, ...\}$ *where* $1 \leq i$.

In event capturing, the sensor output is $output_{ec} \subseteq output_{cds}$. As shown below, the event capturing output is made of those data samples in $output_{cds}$ that are different from their previous (in terms of time-step) data sample:

$output_{ec} = \{s_i \in output_{cds} \mid s_i \neq s_{i-1}\}$

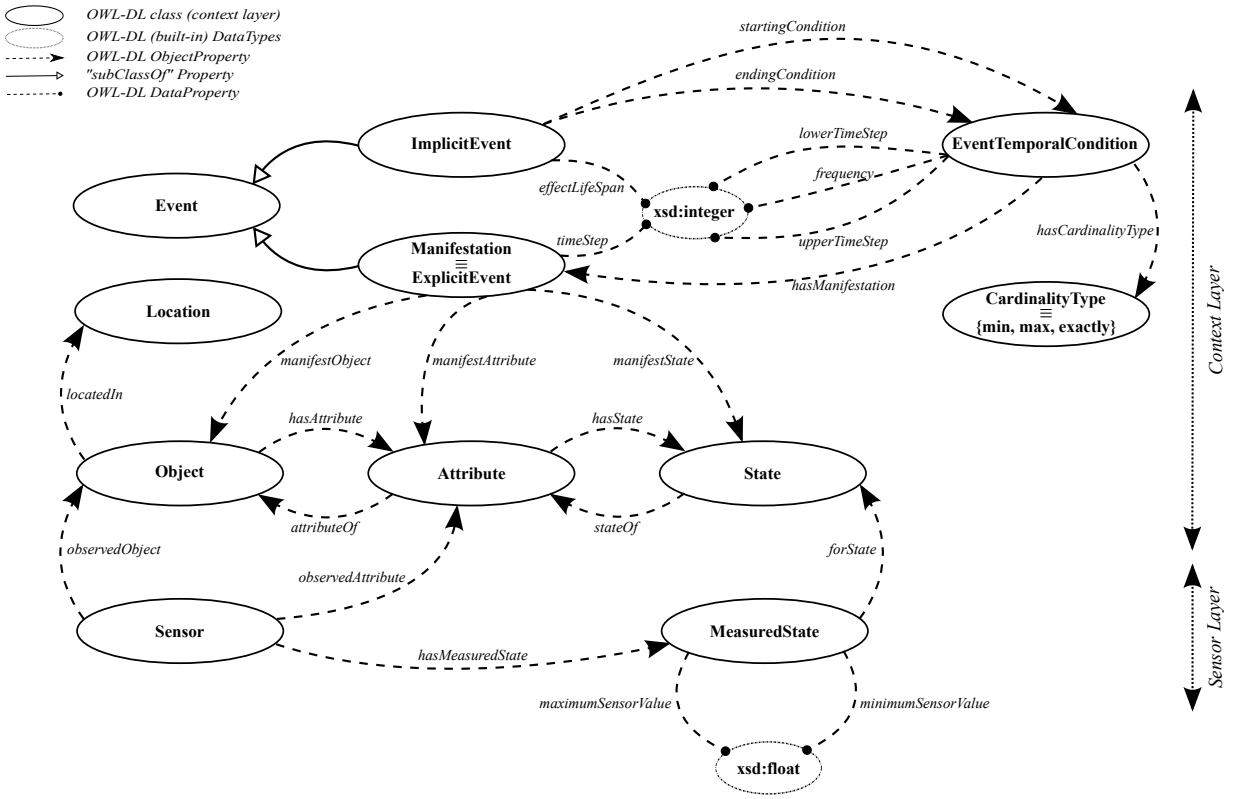

Fig. 2. The hierarchical structure of the OWL-DL Ontology

In order to lighten the data load, in this work, we chose the event capturing approach. The event capturing is accomplished by passing the data stream through the change detection process, depicted in Fig. 1. According to its definition, the event capturing process extracts those data points that indicate an event (a significant change) in the signal.

In Section 5.2.2, we show how each data point related to a change in the signal is represented in form of a *manifestation* considered as an event. Although the data output of an event capturing approach is not as complete as the data in the continuous data sampling approach, it is possible to cope with gaps in the sensor output via ASP in handling incompleteness of data.

5.2. Block B: Data/Knowledge Integration

Block *B* integrates the sensor data and maps it into a symbolic representation. Both time dependent and independent concepts are modelled within the processes in this block. The static knowledge which represents the time independent concepts are related to the observed phenomena and their properties. The cumulative knowledge represents the time dependent concepts related to observations in the sensor data and events. The static knowledge is separated from the cumulative knowledge in order to facilitate the transformation from the ontology to the logic program solvable by the incremental ASP solver.

5.2.1. Static Knowledge

We start by describing the representational details of the static knowledge modelled in the ontology whose hierarchical structure is illustrated in Fig. 2. The ontology is made up of two layers, namely a context layer and a sensor layer. The context layer specifies general facts about the context along with their constraints, whereas the sensor layer contains concepts with quantitative values related to sensors.

Context Layer: The class *Object*, in the ontology refers to all entities that can be a source of an event (detected by the target sensor) in the environment. For instance: a freezer is considered as an object (*Freezer* $\sqsubseteq$*Object*), since its stuff inside can be rotten and smell. Similarly, an oven is regarded as an object (*Oven* $\sqsubseteq$*Object*), since it initiates the process of cooking which can emit an odour.

Table 1
DL definition of fundamental concepts in the ontology)

Layer	Class	DL Definition
Context Layer	**Object**	$\forall\ hasAttribute.$**Attribute** $\sqcap\ (\geq 1\ hasAttribute)\ \sqcap$ $\exists\ locatedIn.$**Location**
	Attribute	$\forall\ hasState.$**State** $\sqcap\ (\geq 2\ hasState)\ \sqcap$ $\exists\ attributeOf.$**Object**
	State	$\exists\ stateOf.$**Attribute**
	Manifestation $\equiv$ **ExplicitEvent**	$\sqsubseteq$ **Event** $\sqcap$ $\forall\ manifestObject.$**Object** $\sqcap\ (=1\ manifestObject)\ \sqcap$ $\forall\ manifestAttribute.$**Attribute** $\sqcap\ (=1\ manifestAttribute)\ \sqcap$ $\forall\ manifestState.$**State** $\sqcap\ (=1\ manifestState)\ \sqcap$ $\forall\ timeStep.$**xsd:integer** $\sqcap\ (=1\ timeStep)$
	EventTemporalCondition	$\forall\ hasManifestation.$**Manifestation** $\sqcap\ (=1\ hasManifestation)\ \sqcap$ $\forall\ lowerTimeStep.$**xsd:integer** $\sqcap\ (=1\ lowerTimeStep)\ \sqcap$ $\forall\ upperTimeStep.$**xsd:integer** $\sqcap\ (=1\ upperTimeStep)\ \sqcap$ $\forall\ frequency.$**xsd:integer** $\sqcap\ (=1\ frequency)\ \sqcap$ $\forall\ hasCarddinalityType.$**CardinalityType** $\sqcap\ (=1\ hasCarddinalityType)$
	CardinalityType	$\{min\} \sqcup \{max\} \sqcup \{exactly\}$
	ImplicitEvent	$\sqsubseteq$ **Event** $\sqcap$ $\forall\ startingCondition.$**EventTemporalCondition** $\sqcap\ (\geq 1\ startingCondition)\ \sqcap$ $\forall\ endingCondition.$**EventTemporalCondition** $\sqcap\ (\geq 1\ endingCondition)\ \sqcap$ $\exists\ effectLifeSpan.$**xsd:integer** $\sqcap\ (=1\ effectLifeSpan)$
Sensor Layer	**MeasuredState**	$\forall\ forState.$**State** $\sqcap\ (=1\ forState)\ \sqcap$ $\forall\ maximumSensorValue$ **xsd:float** $\sqcap\ (=1\ maximumSensorValue)\ \sqcap$ $\forall\ minimumSensorValue$ **xsd:float** $\sqcap\ (=1\ minimumSensorValue)$
	Sensor	$\forall\ observedObject.$**Object** $\sqcap\ (=1\ observedObject)\ \sqcap$ $\forall\ observedAttribute.$**Attribute** $\sqcap\ (=1\ observedAttribute)\ \sqcap$ $\forall\ hasMeasuredState.$**MeasuredState** $\sqcap\ (=1\ hasMeasuredState)$

Each object is defined based on a set of attributes such as temperature (*Temperature* $\sqsubseteq$ *Attribute*), illumination (*Illumination* $\sqsubseteq$ *Attribute*) or electric-current (*ElectricCurrent* $\sqsubseteq$ *Attribute*), whose situations are significant in event definition. The situation of attributes, furthermore, are defined as individuals of the class *State* which independent of the sensor values, represent all possible situations (e.g., *cold* $\in$ *State*, *warm* $\in$ *State*, *dark* $\in$ *State*) of an attribute. The list of complete objects, attributes and states used in this work are shown in Table 2.

As mentioned before, there is at least one sensor in the environment whose data is targeted for the interpretation process. In our scenario, the target sensor is a gas sensor which is continuously sniffing the ambient air. The difference between modelling the target sensor data and the others is in definition of their states. For the target sensor, the possible states are limited into two *normal* and *abnormal* states. Once the sensor data is out of the value range of the *normal* state, the state is switched into *abnormal* which needs to be interpreted.

Therefore, according to the ontology structure, the following concepts related to the target sensor are also defined in the ontology:

KitchenAir $\sqsubseteq$ *Air* $\sqsubseteq$ *Object*

Smell $\sqsubseteq$ *Attribute*

normal $\in$ *State*, *abnormal* $\in$ *State*

Sensor Layer: The sensor layer shown in both Fig. 2 and Table 1, contains concepts representing quantitative sensor-related data. The class *Sensor* is responsible for holding sensors' information used for the observation process. As shown in Fig. 1, the change detection component in charge of event capturing process (Section 5.1), has access to the ontology for retrieving the range of values indicating an event in sensor data. Individuals of the class *State* (such as

Table 2
DL definition along with individuals of the Object, Attribute and State classes

Class	DL Definition	Individuals
Air	$\sqsubseteq$ **Object** $\sqcap$ $hasAttribute$.**Smell**	kitchenAir1
Freezer	$\sqsubseteq$ **Object** $\sqcap$ $hasAttribute$.**ElectricCurrent** $\sqcap$ $hasAttribute$.**Door** $\sqcap$ $hasAttribute$.**Temperature** $\sqcap$ $hasAttribute$.**Illumination**	freezer1
Fridge	(same as Freezer)	fridge1
Microwave	$\sqsubseteq$ **Object** $\sqcap$ $hasAttribute$.**ElectricCurrent** $\sqcap$ $hasAttribute$.**Motion**	microwave1
Oven	$\sqsubseteq$ **Object** $\sqcap$ $hasAttribute$.**ElectricCurrent** $\sqcap$ $hasAttribute$.**Motion**	oven1
TrashBin	$\sqsubseteq$ **Object** $\sqcap$ $hasAttribute$.**Door** $\sqcap$ $hasAttribute$.**Illumination**	trashbin1
Door	$\sqsubseteq$ **Attribute** $\sqcap$ $hasState$.{**open, close**}	doorFreezer1, doorFridge1, doorTrashBin1
Illumination	$\sqsubseteq$ **Attribute** $\sqcap$ $hasState$.{**dark, bright**}	illuminationFreezer1, illuminationFridge1, illuminationTrashBin1
ElectricCurrent	$\sqsubseteq$ **Attribute** $\sqcap$ $hasState$.{**on, off**}	ecFreezer1, ecFridge1, ecOven1, ecMicrowave1
Temperature	$\sqsubseteq$ **Attribute** $\sqcap$ $hasState$.{**cold, warm**}	temFreezer1, temFridge1, temTrashBin1
Motion	$\sqsubseteq$ **Attribute** $\sqcap$ $hasState$.{**moving, steady**}	motionOven1
Smell	$\sqsubseteq$ **Attribute** $\sqcap$ $hasState$.{**normal, abnormal**}	smell1
State		cold, warm, on, off, dark, bright, open, close, moving, steady, normal, abnormal

cold, *warm*, *on*, *off*) symbolically explain objects in terms of their attributes' situations. However, since it is the sensor output which shows the state of the observed object, we need to create a link between abstract (symbolic) states and the real observation values. For this, we define the *MeasuredState* class whose individuals relate sensor values to specific states. As represented in Table 1, each individual of the *MeasuredState* class has two properties, *minimumSensorValue* and *maximumSensorValue*, providing a range of values for a state. For example, the triple *(min=1, max=3, forState=cold)* creates a *MeasuredState* instance which is related to a particular sensor via the *hasMeasuredState* property. Instances of this class specialized for our scenario are listed in Table 3. Ranges of values set by the user for a state depends on both the sensor type and the object. For example, although the two last rows of Table 3 are about the *cold* state, there are two different ranges of values for them indicating the *cold* state for a fridge and a freezer, respectively.

Table 3
Samples of MeasuredState Individuals

State	Minimum	Maximum	Sensor	Object
close	0	0	DoorSensor	TrashBin
open	1	1	DoorSensor	TrashBin
dark	0	200	IlluminationSensor	TrashBin
bright	200	1600	IlluminationSensor	TrashBin
...	...	...	...	...
cold	-2	3	Thermometer	Fridge
cold	-30	-1	Thermometer	Freezer

5.2.2. Cumulative Knowledge

The cumulative knowledge takes events captured by the change detection and translates them into symbolic concepts that either denote *ExplicitEvents* called *Manifestations* or *ImplicitEvents*. An explicit event is one which is measured by the sensors. For instance, whenever the temperature of the freezer changes from a cold state to a warm state, the change detection component generates a manifestation such as *m:(freezer, temperature, cold, t)*, regardless of the range of sensor data for a cold freezer. The last property of the manifestation class is the *timeStep* (*t*) which is related to the incremental step considered by the reasoner.

Once the user populates the ontology with proper objects, attributes and states, tentative classes referring to events that are directly observable by sensors (explicit events) are generated as subclasses of the *Manifestation* class. The name of each subclass is literally generated by combining the name of object, attribute and state concepts involved in the event, respectively. For instance, a manifestation subclass that can be probably detected is *FreezerTemperatureWarm* which indicates an explicit event (a manifestation) related to the freezer (object) whose temperature (attribute) gets warm (state).

The *ImplicitEvent* class addresses those events that are not directly observable by sensors but triggered based on a rule set that consists of Explicit Events. For example, in case of a kitchen, events such as *Cooking* and *Rotting* are defined as subclasses of the *ImplicitEvent* class, and are the main events that cause a change in the quality of the kitchen's ambient air.

Each *ImplicitEvent* so as to be inferred by the reasoner, needs a set of specific a priori conditions to be detected in the environment. These pre-conditions are characterized as instances of the *EventTemporalCondition* class. Two properties, *startingCondition* and *endingCondition* (Fig. 2), relate an implicit event to its conditions which indicate the situations required for inferring the event started or ended, respectively. The class *EventTemporalCondition* as such is related to manifestations. In other words, an event's conditions are defined based on manifestations. Therefore, the class *EventTemporalCondition*, has one relation with the *Manifestation* class via the *hasManifestation* property.

Moreover, the detection of an event's preconditions is expected to be during a specific time around the time step of the event. Therefore, each precondition, apart from its manifestation, is also assigned with two integer values indicating the lower and upper bound of a range for the event time step. The *lowerTimeStep* and the *upperTimeStep* properties are used for this purpose. The preconditions of each implicit event inferred at time step *t*, can happen either at the same time step or before it. In case of the coincidence, the two integer values are set to zero, otherwise they are set to positive numbers. For example, given *Rotting* as an implicit event, we define a subclass for *EventTemporalCondition*, e.g., ETC1. One of the manifestations that can be related to the rotting process is *FreezerTemperatureWarm*. In order to infer *Rotting* at time step *t*, its manifestation, for example, has to be detected between time step *t-B* and *t-A* where both *A* and *B* ($0 \leq A \leq B$) are integer values assigned to the class ECT1 via the *upperTimeStep* and the *lowerTimeStep* properties by the user, respectively. If the manifestation needs to be detected exactly as the same time of the event, the user set both values of *A* and *B* to zero ($A = B = 0$).

For some implicit events, furthermore, we need to indicate the rate of frequency for each of their manifestations which is set by the *frequency* property. *CardinalityType* property also indicates the type of cardinality (including min, max and exactly) set for the frequency value. For example, the *frequency* of ECT1 is set to 1 and its *cardinality* is set to *exactly*. It means that, in order to be able to infer the *Rotting* event at *t*, we need to detect exactly 1 *FreezerTemperatureWarm* manifestation occurred at time step *t* (because *A* $=B=$ 0).

As shown in Fig. 2, each implicit event is assigned to one or several *EventTemporalCondition* concepts via its *startingCondition* and *endingCondition* properties. These relations between implicit events and their conditions are set by the user based on the features of the scenario. Further details about implicit events are given in Section 6.

6. Reasoning about Odours

In this section, the details about the conversion process and the reasoning process, depicted in Block *C* (Fig. 1), are discussed based on the notations explained in previous sections.

6.1. Generating the Base Logic Program ($\mathcal{B}$)

In order to generate the time independent logic program $\mathcal{B}$, for each individual object, attribute and state defined as the static knowledge in the ontology (Ta-

ble 2), two logical literals including a fact and a unary predicate are added to the program $\mathcal{B}$ as follows:

$$\forall o \in O \sqsubseteq \textit{Object} \mapsto B = B \cup \{o, \textit{O(o)}\}$$
$$\forall a \in A \sqsubseteq \textit{Attribute} \mapsto B = B \cup \{a, \textit{A(a)}\}$$
$$\forall s \in \textit{State} \mapsto B = B \cup \{s, \textit{State(s)}\}$$

For example, given an individual *freezer1* as an instance of the class *Freezer* (*freezer1* $\in$ *Freezer* $\sqsubseteq$ *Object*), the conversion process ends up with $\mathcal{B}$ = $\mathcal{B} \cup$ {*freezer1*, *freezer(freezer1)*} where *freezer1* is characterized as a logical fact and *freezer* is a unary and time independent predicate. Similarly, for *temp1* as an instance of the class *Temperature* (*temp1* $\in$ *Temperature* $\sqsubseteq$ *Attribute*), $\mathcal{B}$ is extended with two new literals {*temp1*, *temperature(temp1)*}.

The final $\mathcal{B}$ logic program is partially shown in the following:

$\#base.$
$fridge(fridge1).$
$freezer(freezer1).$
$oven(oven1).$
$microwave(microwave1).$
$trashbin(trashbin1).$
...
$door(doorFreezer1).$
$door(doorFridge1).$
...
$illumination(illuminationTrashBin1).$
...
$state(on).$
$state(off).$
...
$state(normal).$
$state(abnormal).$

6.2. Generating the Cumulative Logic Program ($\mathcal{P}[t]$)

The cumulative logic program $\mathcal{P}$ is generated by time dependent concepts in the ontology. Time dependent concepts containing the time step parameter t in their definitions include both the explicit (manifestation) and the implicit events.

As mentioned before, $\mathcal{P}$ is incrementally extended meaning that whenever an explicit event is captured, a manifestation indicating an object ($o \in O \sqsubseteq$ *Object*), its attribute ($a \in A \sqsubseteq$) and its state ($s \in$ *State*) at time step *t*, is converted into an appropriate predicate and extends $\mathcal{P}$ with a new manifestation predicate as follows:

$$\mathcal{P} = \mathcal{P} \cup \{manifestation(o, a, s, t)\}.$$

Moreover, whenever a manifestation is generated, a rule based on the pattern r_1 shown in the following is added to the program $\mathcal{P}$. This rule allows the solver to infer appropriate explicit events related to the manifestation. The head of the logic rule, *O_A_s(t)*, indicates a predicate whose name is generated by concatenating the name of the subclass of the class *Object* (O), the name of its attribute (A) and the state name (s). The integer value t of $\mathcal{P}[t]$ refers to the last parameters in a manifestation which correspond to the time step at which the change is captured (see Section 5.1). Table 4 shows samples of generated rules related to explicit events.

$$r_1 : O\_A\_s(t) :- manifestation(X, Y, s, t), O(X), A(Y).$$

Apart from explicit events, implicit events (such as *Cooking*, *Rotting*, etc,.) also have a considerable impact on extension of the program $\mathcal{P}$. Since implicit events are not directly observable via sensors, and are defined based on temporal relations between explicit events, the time difference plays an essential role in their definition. For instance, the event *Rotting* starts after passing a specific amount of time (e.g., 1 day) when the explicit event *FreezerTemperatureWarm* has been detected. Therefore, for the sake of inferring implicit events and measuring the time difference between events, we need to continuously sample sensor data.

Nevertheless, as mentioned in Section 5.1, to hinder the reasoner to be overwhelmed with data, instead of continuous data sampling, we chose the event capturing approach. Compensating the lack of continuous data sampling is achieved by defining each implicit event within three logical rules. The two first rules indicate the conditions required for inferring the starting and the ending of an implicit event, respectively. These conditions are defined in the ontology as the relations between an implicit event and manifestations. As we can see in Fig. 2, the *ImplicitEvent* class is related to

Table 4
Samples of Explicit Event Rules

Object	Property	State	ExplicitEvent Predicates / Rules
freezer	door	open	freezerDoorOpen(t) :- manifestation(X, Y, open, t), freezer(X), door(Y).
oven	electricCurrent	on	ovenElectricCurrentOn(t) :- manifestation(X, Y, on, t), oven(X), electricCurrent(Y).
trashbin	illumination	dark	trashbinIlluminationDark(t) :- manifestation(X, Y, dark, t), trashbin(X), illumination(Y).
...	...	...	...
air	smell	normal	airSmellNormal(t) :- manifestation(X, Y, normal, t), air(X), smell(Y).
air	smell	abnormal	airSmellAbnormal(t) :- manifestation(X, Y, abnormal, t), air(X), smell(Y).

the *Manifestation* class via the *EventTemporalCondition*. The body part of the first rule which indicates the conditions required to infer an implicit event's inception, is generated based on the conjunction of all manifestations that are connected to the instances of the *EventTemporalCondition* class via the *startingCondition* property. Likewise, the body part of the second rule regarding the ending conditions of an implicit event, is the conjunction of all manifestations, that are related to the *EventTemporalCondition* class, however, via the *endingCondition* property. For instance, assuming the following axioms are defined in the ontology, we generate the two first rules for the implicit event *Garbage*:

$g \in Garbage \sqsubseteq$ **ImplicitEvent**
$m1 \in TrahBinDoorOpen \sqsubseteq$ **Manifestation**
$m2 \in TrahBinIlluminationBright \sqsubseteq$ **Manifestation**
$m3 \in TrahBinIlluminationDark \sqsubseteq$ **Manifestation**
$c1, c2, c3 \in$ **EventTemporalCondition**

$(c1, m1) \in$ **hasManifestation**
$(c1, 0) \in$ **lowerTimeStep**
$(c1, 0) \in$ **upperTimeStep**
$(c1, 1) \in$ **frequency**
$(c1, exactly) \in$ **hasCardinality**

$(c2, m2) \in$ **hasManifestation**
$(c2, 0) \in$ **lowerTimeStep**
$(c2, 0) \in$ **upperTimeStep**
$(c2, 1) \in$ **frequency**
$(c3, exactly) \in$ **hasCardinality**

$(c3, m3) \in$ **hasManifestation**
$(c3, 0) \in$ **lowerTimeStep**
$(c3, 0) \in$ **upperTimeStep**
$(c3, 1) \in$ **frequency**
$(c3, exactly) \in$ **hasCardinality**

$(g, c1) \in$ **startingCondition**
$(g, c3) \in$ **startingCondition**
$(g, c1) \in$ **endingCondition**
$(g, c2) \in$ **endingCondition**

Given the above axioms in the ontology, the first rule (r_2) related to the inception of the *Garbage* event will be as follows:

$$r_2 : garbage(t) :- \\ trashBinDoorOpen(t), \\ trashBinIlluminationDark(t).$$

According to r_2, the garbage event is inferred at time step *t* if there are *exactly* 1 manifestation, *trashBinDoorOpen* and 1 manifestation, *trashBinIlluminationDark* detected at time step *t*. Likewise, the second rule (r_3) related to the ending of the *Garbage* event is given in the following:

$$r_3 : garbageEnd(t) :- \\ garbage(t-1), \\ trashBinDoorOpen(t), \\ trashBinIlluminationBright(t).$$

The third rule (r_4), in addition, indicates the required conditions for inference of the event progres-

Table 5
Examples of Implicit Event Rules Generating the Program *P*

Implicit Event	Rules
Rotting	**rotting(t)** :- freezerTemperatureWarm(t). **rottingEnd(t)** :- rotting(t-1), freezerTemperatureCold(t). **rotting(t)** :- rotting(t-1), not rottingEnd(t).
Garbage	**garbage(t)** :- trashBinDoorOpen(t), trashBinIlluminationDark(t). **garbageEnd(t)** :- garbage(t-1), trashBinDoorOpen(t), trashBinIlluminationBright(t). **garbage(t)** :- garbage(t-1) not garbageEnd(t).
Cooking	**cooking(t)** :- ovenElectricCurrentOn(t), 1{ovenMotionMoving(t-15..t)}. **cookingEnd(t)** :- cooking(t-1), ovenElectricCurrentOff(t), 1{ovenMotionMoving(t-15..t)}. **cooking(t)** :- cooking(t-1), not cookingEnd(t).

sion to the next time step. The term "ImplicitEvent" in r_4 refers to a predicate equivalent to an *ImplicitEvent* concept (e.g., *Rotting*)):

$$r_4 : ImplicitEvent(t) : -$$
$$ImplicitEvent(t-1),$$
$$not \;\; ImplicitEventEnd(t).$$

$$e.g.,$$
$$garbage(t) : -$$
$$garbage(t-1),$$
$$not \;\; garbageEnd(t).$$

In our scenario, the numeric values set for the *upperTimeStep* and the *lowerTimeStep* properties are in seconds. These values specify the time interval during which the detection of the manifestation is expected. Table 5 shows the set of generated rules as part of the program $\mathcal{P}$, based on the content of our ontology.

Implicit events which can declaratively express the ambient smell in the kitchen are used as meaningful explanations for changes detected over the gas sensor. In order to infer that the smell of an implicit event is sensed, two conditions have to hold: First, the truth of a manifestation denoting an abnormal state for the target object (*airSmellAbnormal*) at the current time t, and second, the truth of an implicit event:

$$r_5 : smellImplicitEvent(t) : -$$
$$airSmellAbnormal(t),$$
$$ImplicitEvent(t).$$

However, there are situations in which we cannot infer the truth of an implicit event, its smells still stays in the environment. Due to gradually fading of smells, apart from r_5 which indicates the preliminary conditions for the *smellImplicitEvent* predicate, the rule r_6 is also added to the program $\mathcal{P}$:

$$r_6 : smellImplicitEvent(t) : -$$
$$airSmellAbnormal(t),$$
$$smellImplicitEvent(t-1),$$
$$1\{ImplicitEventEnd(t - VALUE..t)\}.$$

The term "VALUE" refers to an integer value showing an approximate time interval during which the smell, even after ending the event, normally stays in the ambient air. This value is set in the ontology for an implicit event instance, via the *effectLifeSpan* property (Fig. 2). According to r_6, the smell of an implicit event is inferred at time step *t* if at the same time step an abnormal state for the ambient air is detected, the smell of the implicit event has been inferred at the previous time step, and the implicit event has been ended within recent "VALUE" time steps. Examples of these rules related to our scenario are given in Table 6. For instance, the "VALUE" in the second rule related to the *smellCooking(t)* has been set to 7200, meaning that the reasoner will interpret the abnormal smell of the ambient air as the cooking smell if at the previous time step, the cooking smell was inferred and there is at least a *cookingEnd* event during recent 2 hours (7200 seconds).

In order to interpret the smell of an inferred implicit event and explain the current state of the ambient air, the following rule (r_7) which is defined for each *smellImplicitEvent* predicate, is also added to the program $\mathcal{P}$:

Table 6
Examples of Smell Implicit Event Rules Generating the Program *P*

Implicit Event	Rules
smellRotting	**smellRotting(t)** :- airSmellAbnormal(t), rotting(t). **smellRotting(t)** :- airSmellAbnormal(t), smellRotting(t-1), 1{rottingEnd(t-21600..t)}.
smellGarbage	**smellGarbage(t)** :- airSmellAbnormal(t), garbage(t). **smellRotting(t)** :- airSmellAbnormal(t), smellGarbage(t-1), 1{garbageEnd(t-5400..t)}.
smellCooking	**smellCooking(t)** :- airSmellAbnormal(t), cooking(t). **smellCooking(t)** :- airSmellAbnormal(t), smellCooking(t-1), 1{cookingEnd(t-7200..t)}.

$$r_7 : explained(t) : - \\ smellImplicitEvent(t).$$

However, due to many reasons such as the lack of observations, or misreading of sensors, along with the aforementioned rules, we also add the two last rules (r_8 and r_9) to the cumulative part of the logic program. In this way, the answer set will always contain either the *airSmellNormal* or *airSmellAbnormal* as a description of the ambient air. The later, depending on the inference results, can be accompanied by the other explanations.

$$r_8 : explained(t) : - \\ airSmellAbnormal(t).$$

$$r_9 : explained(t) : - \\ airSmellNormal(t).$$

6.3. Generating the Volatile Logic Program ($\mathcal{Q}[t]$)

In order to guarantee having an explanation for the ambient air at each time step, the conversion process generates the volatile program $\mathcal{Q}[t]$. Given the predicate *explained(t)*, the volatile part of the logic program will be modelled as:

$$\#volatile\ \ t. \\ :-not\ \ explained(t).$$

According to the integrity constraint rule given in the volatile part, the ASP solver needs to always provide an explanation, otherwise it ends up with dissatisfaction. For this, the solver successively accept new manifestations until the *explained* predicate is inferred which consequently implies the inference of a smell.

7. Results

For experimental validation of the proposed method, a smart kitchen equipped with a set of sensors using ZigBee wireless communication standard [17] is deployed. Table 7 represents the details about sensors and which object the sensor monitors. Figure. 3 also depicts the map of sensors in the smart kitchen. As we can see, each object may be observed by several sensors. For instance, a fridge is monitored with 4 sensors which measure the status of its door, inside illumination, inside temperature and the electric current. Using the two sensors, door and illumination sensors that monitor the trash bin, the system can approximately guess whether the bin is full or not (i.e., *if the cabinet door is open and the light sensor detects darkness, the state of the bin is full*). With plug sensors, we can check the state of appliances such as oven, fridge, freezer and microwave to see if they are working or not. A gas sensor (tin dioxide semiconductors) is used to monitor the ambient air in the smart home. Two batch measurements have been performed. The first batch is a three day run and the second batch is a five day run. The purpose of the system is to annotate each change detected by the gas sensor with an explanation that outlines the possible reasons for the change.

The reasoner is ready to infer in real time, meaning that once a manifestation is reported to the reasoner, at least one explanation for the current ambient air in the environment is inferred.

Considering the three day experimental run, Fig. 4 shows the set of signals collected. The time labels along the x-axis indicate the time at which a change is detected. Due to the lack of space, we selected and showed only a subset of these detected changes in each signal. Fig. 5 visualizes the events along with their

Table 7
List of sensors, their details and monitored objects

Name	Output Data Type	Observed Object(s)	Quantity
Door Sensor	Binary [Open/Close]	Fridge, Freezer, TrashBin	3
Light Sensor	Positive Integer	Fridge, Freezer, TrashBin	3
Plug Sensor	Binary [On/Off]	Fridge, Freezer, Microwave, Oven	4
Thermometer	Integer	Fridge, Freezer	2
Motion Detector	Binary [Motion/No Motion]	Oven	1
ENose	Float	Ambient Air	1

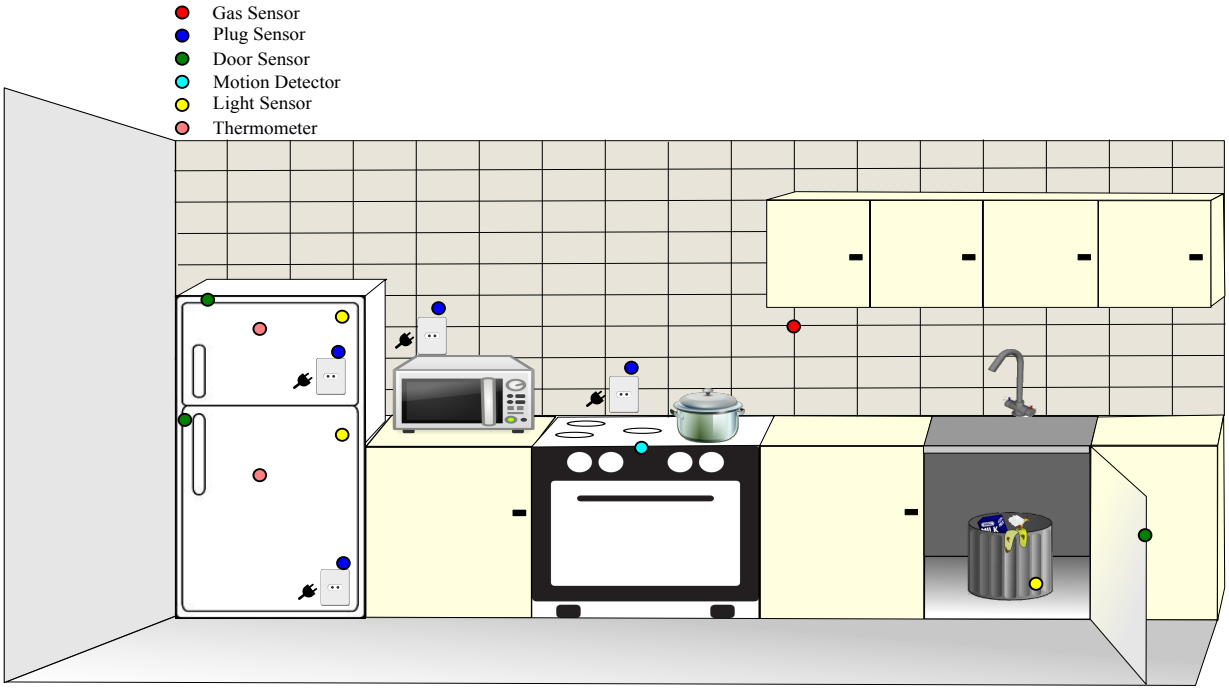

Fig. 3. Perspective of the Smart Kitchen - Sensors' Map (Color figure online).

causes for the same data package. Illustrated in the legend, continuous lines are divided into two main types representing the normal (in green) and abnormal (in red) smells. As we explained in Section 6.2, in case of an abnormal smell, the answer set may be augmented with inferred implicit events as the cause of the detected smell which are represented in different colors in Fig. 5. The relation between a cause and its relevant inference results is also shown in dotted lines.

The incremental reasoner provides the answer set for the set of manifestations given at the specific time step. In Table 8, we partially show the given manifestations as well as the inferred answer set at each time step. The answer set expresses the smell in the environment if there is an *abnormal* smell, otherwise it expresses the situation as *normal* smell. For instance, at the first step ($t = 1$) the initial states of each objects' attributes in form of descriptive manifestations are given. As we can see, the reasoner results in $airSmellNormal(1)$, meaning that the state of the ambient air at $t = 1$ (or first day at 7:00), is *normal*. On the same day, at $t = 16560$ (11:36), two manifestation indicating the oven is on and some one is moving around, are reported. According to the implicit events definition in Table 5, the event *Cooking* is hence inferred at 11:36 (Fig. 5). However, since no abnormal smell in the air is reported, the current ambient air is explained as *normal* (Table 8).

In less than 1 hour, at 12:14 ($t = 18840$), the manifestation *airSmellAbnormal(t)* triggers the reasoner

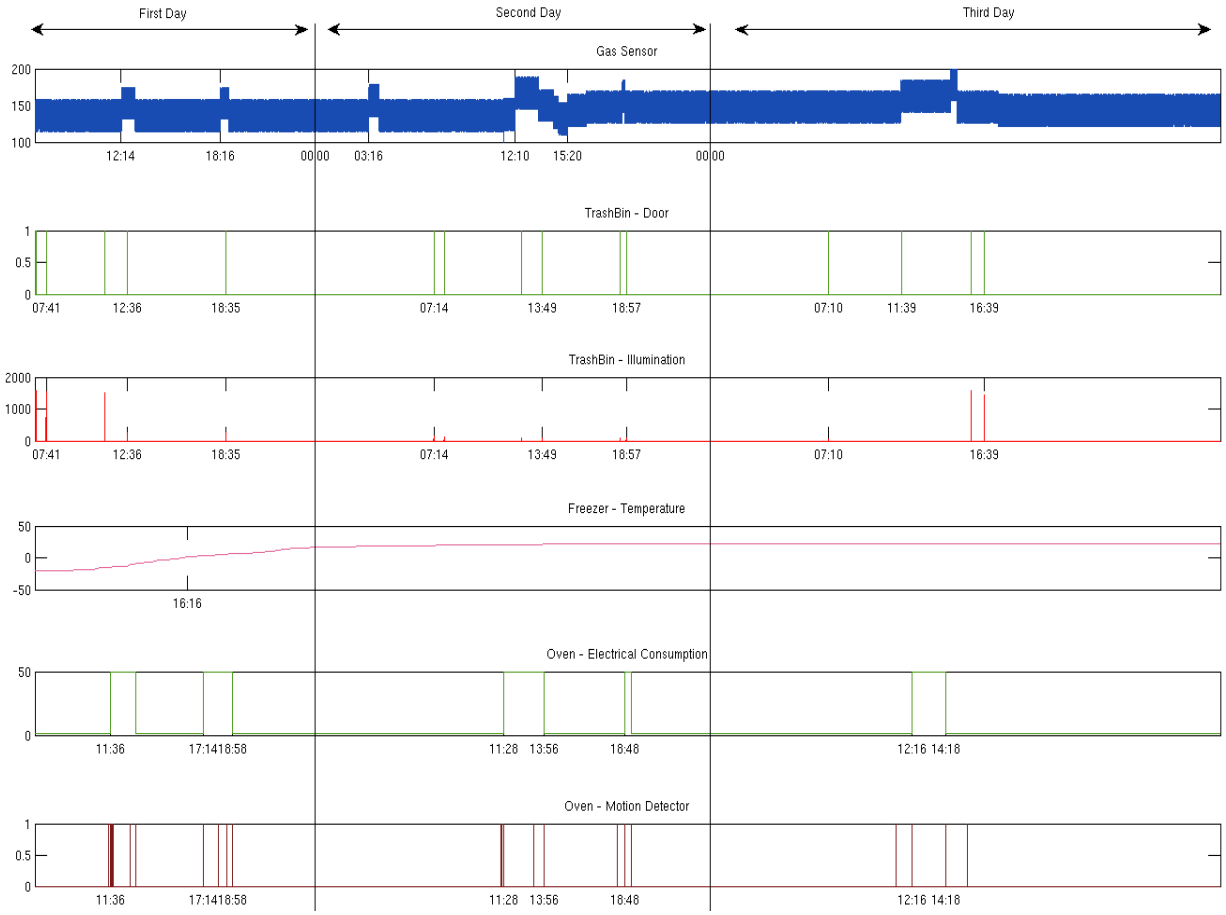

Fig. 4. Set of sensor signals during 3-days of Observation

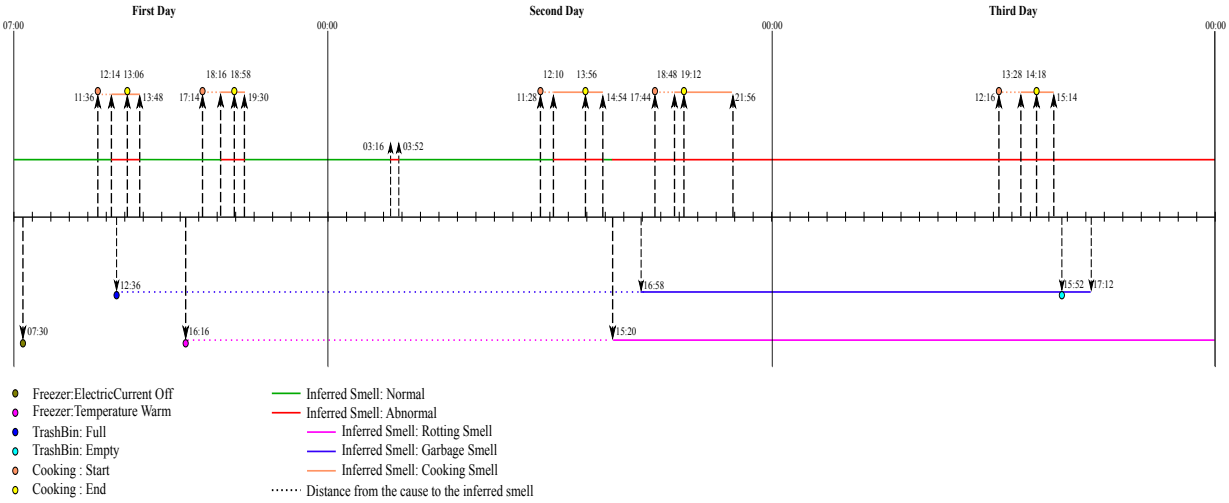

Fig. 5. Set of detected events during 3-days of Observation (Color figure online).

Table 8
A subset of stepwise detected manifestations along with their inferred explanations (answer sets)

Day	Time	TimeStep	Observation	Answer Set (Smell Description)
First Day	7:00	1	manifestation(oven1, electriccurrent, off, 1). manifestation(oven1, motion, steady, 1). manifestation(freezer1, temperature, cold, 1). manifestation(trashbin1, illumination, dark, 1). manifestation(trashbin1, door, close, 1). manifestation(microwave1, electriccurrent, off, 1). manifestation(kitchenAir1, smell, normal, 1).	airSmellNormal(1).
	...	...	...	...
	11:36	16560	manifestation(oven1, electriccurrent, on, 16560). manifestation(oven1, motion, on, 16560).	airSmellNormal(16560).
	12:14	18840	manifestation(kitchenAir1, smell, abnormal, 18840).	airSmellAbnormal(18840). smellCooking(18840).
	12:36	20160	manifestation(trashbin1, illumination, dark, 20160).manifestation(trashbin1, door, open, 20160).	airSmellAbnormal(20160). **smellCooking**(20160).
	...	...	...	...
	16:16	33360	manifestation(freezer1, temperature, warm, 33360).	airSmellNormal(18840). smellCooking(18840).
	...	...	...	...
Second Day	03:16	72960	manifestation(kitchenAir1, smell, abnormal, 72960).	airSmellAbnormal(72960).
	...	...	...	...
	15:20	116400	manifestation(kitchenAir1, smell, abnormal, 116400).	airSmellAbnormal(116400). **smellRotting**(116400).
	16:58	122280	manifestation(kitchenAir1, smell, abnormal, 122280)	airSmellAbnormal(122280). **smellRotting**(122280). **smellGarbage**(122280).
	...	...	...	...
Third Day	...	...	...	...
	15:52	204720	manifestation(trashbin1, illumination, bright, 204720).manifestation(trashbin1, door, open, 204720).	airSmellAbnormal(204720). **smellRotting**(204720). **smellGarbage**(204720).
	17:12	209520	manifestation(kitchenAir1, smell, abnormal, 209520).	airSmellAbnormal(209520). **smellRotting**(209520).

to result in the answer set containing *airSmellAbnormal(18840)* which is accompanied by a further explanation, *smellCooking(18840)*. Although the *cooking* event was inferred at $t = 16560$, due to the progression rule (r_4) of implicit events explained in Section 6.2, the inference of the *Cooking* event is continued during next steps. Since there was no *cookingEnd* predicate inferred between $t = 16560$ and $t = 18840$, the *smellCooking* is inferred as the explanation of the abnormal smell at $t = 18840$.

There are situations in which the change detector component finds a change in the gas signal, for which there is no cause inferred beforehand. The example of this is shown in Table 8, on the beginning of the second day at 03:16 ($t = 72960$). As we see, the answer set only contains *airSmellAbnormal(72960)* as the explanation.

Moreover, there are other situations in which the reasoner results in more than one explanation. For instance in Fig. 4 and 5, the rotting smell explanation is accumulated with the smell of garbage at $t = 122280$ (second day at 16:58) whose causes are two manifestations at $t = 20160$ (first day at 12:36) stating the state of the trash bin as full[2] (see Table 8).

Although an implicit event eventually ends, its effect may remain in the environment. The second rule of each implicit event's smell in Table 6 enables the reasoner to continue the inference of the event's smell even after its ending. The effect of this rule is seen in Table 8 on the third day. As we can see, after detecting that the trash bin is empty at $t = 204720$, the ambient air explanation contains the smell of garbage not more than 5400 time steps (one and half hour set in Table 6), and the only explanation for the abnormal smell at $t = 209520$ (third day at 17:12) is the rotting smell. Likewise, shown in Fig.5, on the first day, the cooking event ends at 13:06, however, due to the rules in Table 6, the *smellCooking* is still inferred while the abnormal smell exist in the ambient air till 13:48.

Figure 6 illustrates the percentage of each explanation inferred by the reasoner during 3 days of ob-

[2]The light sensor installed at the bottom of the bin detects darkness while the cabinet door of the bin is open

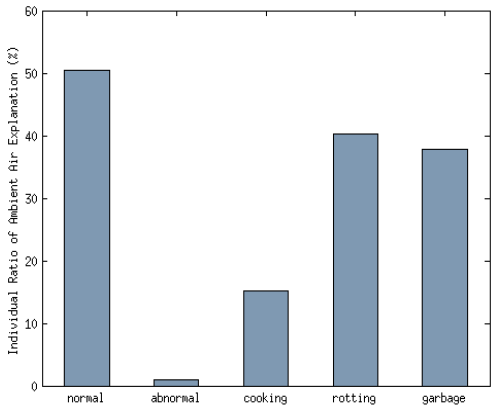

Fig. 6. Individual Ratio of Ambient Air Explanations to the Total Observation Time

servation. The normal state with the ratio of 50.45% is the most frequent explanation, and the abnormal state which is not accompanied by any further explanation, with the ratio of less than 1% (0.93%) is the least frequent inferred smell. It is worth mentioning that the percentage of each smell is individually calculated only with respect to the total observation time. Furthermore, as explained before, there are situations in which the abnormal (not normal) smells such as rotting, cooking and garbage can be inferred at the same time, and that is why the percentage of each smell in Fig. 6 is calculated independent of that of the other smells.

7.1. Evaluation

In order to further study the performance of the reasoner, we extend the experiments with the second package of data containing 5 days of observations. This observation process has been done with the same set of sensors. Depending on the data package, the ontology contains different amount of individuals related to both the time independent and time dependent concepts. Since the event based classes are time dependent, the number of their individuals depends on several parameters including the length of the observation process, the number of events happen in the environment, and the measuring unit of time in the observation process. For the first round of the observation process (3 days), for instance, the number of events individuals is $i \approx 762584$, and for a longer observation (5 days), this number increases to $i \approx 1512907$.

In the following we compare the reasoning time of our system based on the incremental ASP solver with that of the monotonic ontology reasoner, Pellet [33]. Taking the two advantages of the incremental ASP solver into account, we evaluate the scalability of our approach.

The first advantage of using ASP solver is the ability of interpreting the negation as failure (NAF) operator whose lack, as we see later, implies a rise in number of individuals for a deductive reasoner (for 5 days of observation, $i \approx 2174903$). The NAF operator in definition of implicit events (the progression rule) mentioned in Section 6.2, allows the reasoner to infer based on closed world assumption (CWA). According to CWA, the reasoner considers a predicate as false if the truth of the predicate cannot be proven. However, in deductive reasoning which is based on the open world assumption (OWA), the predicate is considered as false only if its falsity is proved, otherwise it is unknown. Therefore, in order to be able to model the progression rules for a deductive reasoner, a number of extra individuals of negated predicates preceded by the strong negation operator ($\neg$ *ImplicitEventEnd*), needs to be added. The amount of required individuals is the ratio of the number of time steps t between the two time steps at which the implicit event starts and ends ($t : t_{start}..t_{end}$). Due to its dependency to the number of time steps, the number of required extra individuals for a deductive reasoner, therefore, is influenced by the measuring unit of time. For instance, the number of added individuals would be less if the time unit was in minute rather than in second. The number of individuals (in grounding phase of the reasoner) can impact the reasoning time. Since the ASP reasoner does not need to have the extra individuals, the unit of time measuring hence does not impact the reasoning time. In Fig. 7 the number of individuals increasing during the observation process is shown for both types of reasoners.

The incremental solver, furthermore, extends the grounded logic program (which contains no variable but the individuals) incrementally and enables the reasoning process to only consider the recently added manifestations (events) during the solving process, rather than the entire grounded individuals. Figure. 8 shows the difference of the reasoning time between the deductive Pellet ontology reasoner and the ASP incremental solver. Therefore, in addition of the re-usability feature, our ontology-based knowledge representation and reasoning approach provides a considerable efficiency in reasoning time which consequently enhances the scalability of the system.

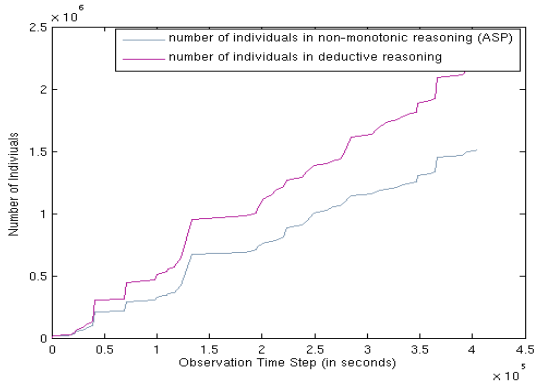

Fig. 7. Number of Individuals (Incremental ASP vs. Pellet)

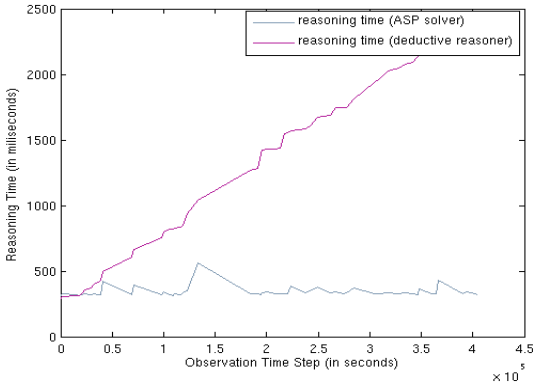

Fig. 8. Comparing Reasoning Time (Incremental ASP vs. Pellet)

8. Conclusion

In this work, using contextual information, we implemented a general time series signals explanation process. The key point is enabling non-monotonic reasoning over an ontology which is inspired from the SSN ontology. Separate layers (context and sensor layers) considered in this ontology increases the adaptivity of the system. Moreover, integrating the ontology with the ASP solver provides the opportunity of intuitively modelling the environment at different phases of the process. The incremental ASP solver also enables the encoding of the environment's history incrementally which leads into an efficient reasoning time, specially when the size of individuals grows. Because of its expressivity, the ASP semantics, in addition, simplifies the automated creation process of the logic program. The whole logic program except the implicit events creation is done automatically. The manually added rules related to implicit events are also defined based on the combination of the existing expressive predicates that state explicit events.

However, the temporal relation between manifestations can be further developed. If the number of states of an object increases, the number of rules considering temporal relation between events and their causes also grows. For example, with 2 states defined for the trash bin, *bright* and *dark*, the reasoner infers the garbage smell after passing 2 days of being in the *dark* state. Given the third state *semi dark*, the reasoner needs a new rule which leads to the same result (garbage smell) however after passing more than 2 days of being in the *semi dark* state. Extending the temporal relations between events, as our future work, can leads to a more enriched temporal reasoning.

References

[1] C. Baral, *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press New York, NY, USA, 2003.

[2] M. Gelfond and V. Lifschitz, *The Stable Model Semantics for Logic Programming*. ICLP/SLP, P.1070-1080, MIT Press, Massachusetts, MA, 1988.

[3] F. Baader and D. Calvanese and D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider (Eds.), *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[4] M. Compton and P. Barnaghi and L. Bermudez and R. Garcia-Castro and O. Corcho and S. Cox and J. Graybeal and M. Hauswirth and C. Henson and A. Herzog and V. Huang and K. Janowicz and W. Kelsey and D. Phuoc and L. Lefort and M. Leggieri and H. Neuhaus and A. Nikolov and K. Page and A. Passant and A. Sheth and K. Taylor, *The SSN ontology of the W3C semantic sensor network incubator group*. J.Web Semantics: Science, Services and Agents on the World Wide Web (17), P.25-32, Elsevier, 2012.

[5] T. Grote, *A reactive System for Declarative Programming of Dynamic Applications*. University of Potsdam, Knowledge Processing and Information Systems, Germany, P.25-32, 2010.

[6] L. Lefort and C. Henson and K. Taylor and P. Barnaghi and M.Compton and O. Corcho and R. Garcia-Castro and J. Graybeal and A. Herzog and K. Janowicz and H. Neuhaus and A. Nikolov and K. Page , *Semantic Sensor Network XG Final Report*. World Wide Web Consortium, Technical Report, 2011.

[7] M. Obitko, *Web Ontology Language OWL*. http://www.obitko.com/tutorials/ontologies-semantic-web/web-ontology-language-owl.html, 2007.

[8] M. Gebser and G. Torsten and K. Roland and S. Torsten, *Reactive answer set programming*. Proc. Int. Conf. on Logic programming and nonmonotonic reasoning (LPNMR), P.54-66, Springer Verlag, 2011.

[9] M. Gebser and R. Kaminski and B. Kaufmann and M. Ostrowski and S. Torsten and S. Thiele, *Engineering an Incremental ASP Solver*. Proc. Int. Conf. on Logic Programming (ICLP), P.190-205, Springer, 2008.

[10] T. Eiter and G. Ianni and T. Krennwallner, *Answer Set Programming: A Primer*. Lecture Notes in Computer Science, Reasoning Web, P.40-110, Springer, 2009.

[11] A. Loutfi and S. Coradeschi and A. Saffiotti, *Maintaining Coherent Perceptual Information Using Anchoring*. Proc. 19th international joint conference on Artificial intelligence (IJCAI'05), P.1477-1482, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[12] W. Wang and P. Barnaghi, *Semantic Annotation and Reasoning for Sensor Data*. Lecture Notes in Computer Science, P.66-76, Springer, 2009.

[13] K. Thirunarayan and C. Henson and A. Sheth, *Situation awareness via abductive reasoning from Semantic Sensor data: A preliminary report*. International Symposium on Collaborative Technologies and Systems(CTS), P.111-118, IEEE, 2009.

[14] P. Barnaghi and F. Ganz and C. Henson and A. Sheth, *Computing Perception from Sensor Data*. IEEESensors, IEEE Sensors Council, 2012.

[15] R. Anand and R. Campbell, *An infrastructure for context-awareness based on first order logic*. Personal and Ubiquitous Computing, P.353-364, Springer-Verlag, 2003.

[16] M. Gebser and T. Grote and R. Kaminski and Ph. Obermeier and O. Sabuncu and T. Schaub, *Stream Reasoning with Answer Set Programming: Preliminary Report*. Proc. 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR), AAAI Press, 2012.

[17] J. Torresen and E. Renton and A.R. Jensenius, *Wireless Sensor Data Collection based on ZigBee Communication*. Proceedings of New Interfaces for Musical Expression++, P.368-371, NIME, 2010.

[18] M. Alirezaie and A. Loutfi, *Ontology Alignment for Classification of Low Level Sensor Data*, Proc. Int. Conf. on Knowledge Engineering and Ontology Development (KEOD), P.89-97, SciTePress, 2012.

[19] A. Loutfi and S. Coradeschi and L. Karlsson and M. Broxvall, *Putting olfaction into action: using an electronic nose on a multi-sensing mobile robot*, Proc. Int. Conf. on Intelligent Robots and Systems (IROS), P.337-342, IEEE, 2004.

[20] F. Baader and I. Horrocks and U. Sattler, *Description Logics as Ontology Languages for the Semantic Web*, Festschrift in honor of Jörg Siekmann, Lecture Notes in Artificial Intelligence, P.228-248, Springer, 2003.

[21] M. O'Connell and G. Valdora and G. Peltzer and R. Negri, *A practical approach for fish freshness determinations using a portable electronic nose*. Sensors and Actuators B: Chemical, P.149-154, Elsevier, 2001.

[22] M. Penza and G. Cassano and F. Tortorella and G. Zaccaria, *Classification of food, beverages and perfumes by {WO3} thin-film sensors array and pattern recognition techniques*. Sensors and Actuators B: Chemical, P.76-87, Elsevier, 2001.

[23] M. Trincavelli and S. Coradeschi and A. Loutfi and B. Söderquist and P. Thunberg, *Direct Identification of Bacteria in Blood Culture Samples Using an Electronic Nose*. IEEE Trans. Biomed. Engineering, P.2884-2890, IEEE, 2010.

[24] GC. Green amd ADC. Chan and RA. Goubran, *Monitoring of food spoilage with electronic nose: Potential applications for Smart Homes*. Proc. 3rd Int. Conf. on Pervasive Computing Technologies for Healthcare, P.1-7, IEEE, 2009.

[25] M. Holmberg and F. Gustafsson and G. Hörnsten andF. Winquist and L.E Nilsson and L. Ljung and I. Lundström, *Bacteria classification based on feature extraction from sensor data*. Biotechnology techniques, P.319-324, Springer, 1998.

[26] C. Delpha and M. Siadat and M. Lumbreras, *An electronic nose for the identification of Forane {R134a} in an air conditioned atmosphere*. Sensors and Actuators B: Chemical, P.243-247, Elsevier, 2000.

[27] R.A.a.b Fenner and R.M.a. Stuetz, *The application of electronic nose technology to environmental monitoring of water and waste water treatment activities*. Water Environment Research, P.282-289, Water Environment Federation, 1999.

[28] A. Mileo and D. Merico and S. Pinardi and R. Bisiani, *A Logical Approach to Home Healthcare with Intelligent Sensor-Network Support*. The Computer Journal, P.1257-1276, Oxford University Press, 2010.

[29] L. Padovani and A. Provetti, *Qsmodels: ASP Planning in Interactive Gaming Environment*. Logics in Artificial Intelligence, P.689-692, Springer, 2004.

[30] S. Coradeschi and A. Loutfi and B. Wrede, *A Short Review of Symbol Grounding in Robotic and Intelligent Systems*. KI - Künstliche Intelligenz, P.129-136, Springer-Verlag, 2013.

[31] A. Mileo and D. Merico and R. Bisiani, *Support for context-aware monitoring in home healthcare*. Journal of Ambient Intelligence and Smart Environments, P.49-66, IOS Press, 2010.

[32] C. Perera and A. Zaslavsky and P. Christen and D. Georgakopoulos *Context Aware Computing for The Internet of Things: A Survey*. Communications Surveys & Tutorials, IEEE, P.414-454, IEEE, 2014.

[33] E. Sirin and B. Parsia and B.C. Grau and A. Kalyanpur and Y. Katz, *Pellet: A Practical OWL-DL Reasoner*, Web Semant. (5), P.51-53, Elsevier Science Publishers B. V., 2007.

[34] K. Dentler and R. Cornet and A. Teije and N. Keizer, *Comparison of Reasoners for Large Ontologies in the OWL 2 EL Profile*, Semant. web. P.71–87, IOS Press, 2011.

[35] H. Tompits, *A survey of non-monotonic reasoning*, Open Systems & Information Dynamics. P.369-395, Springer, 1995.

[36] T. Lukasiewicz, *A Novel Combination of Answer Set Programming with Description Logics for the Semantic Web*, IEEE Transactions on Knowledge and Data Engineering. P.1577-1592, IEEE Computer Society, 2010.