A Knowledge Driven Policy Framework for Internet of Things*

Emre Goynugur[1], Geeth de Mel[2], Murat Sensoy[1], Kartik Talamadupula[3] and Seraphin Calo[3]
[1]*Computer Science, Ozyegin University, Istanbul, Turkey*
[2]*Daresbury Laboratory, IBM Research , Warrington, U.K.*
[3]*T. J. Watson Research Center, IBM Research, Yorktown Heights, NY, U.S.A.*

Keywords: Internet of Things, IoT, Semantic Web, OWL-QL, Policy, Conflict Resolution.

Abstract: With the proliferation of technology, connected and interconnected devices (henceforth referred to as IoT) are fast becoming a viable option to automate the day-to-day interactions of users with their environment—be it manufacturing or home-care automation. However, with the explosion of IoT deployments we have observed in recent years, manually managing the interactions between humans-to-devices—and especially devices-to-devices—is an impractical task, if not an impossible task. This is because devices have their own obligations and prohibitions in context, and humans are not equip to maintain a bird's-eye-view of the interaction space. Motivated by this observation, in this paper, we propose an end-to-end framework that *(a)* automatically discovers devices, and their associated services and capabilities w.r.t. an ontology; *(b)* supports representation of high-level—and expressive—user policies to govern the devices and services in the environment; *(c)* provides efficient procedures to refine and reason about policies to automate the management of interactions; and *(d)* delegates similar capable devices to fulfill the interactions, when conflicts occur. We then present our initial work in instrumenting the framework and discuss its details.

1 INTRODUCTION

According to the vision of Smarter Planet initiative, efficient networks which are made up with low-cost devices are going to augment and enhance the day-to-day interactions of humans and organisations (Palmisano, 2008). With the proliferation of technology, and the internet connected and interconnected devices—collectively referred to as *Internet of Things* (IoT)—this is fast becoming a reality; according to Gartner[2], by 2020, there will be over 20 billion interconnected devices, and they will transform the lives of people and organisations alike by augmenting their experiences in environments.

In order for the above systems to function effectively, it is mandatory that an IoT-enabled system supports functionality for devices—especially the services support (or exposed) by those devices—to interact with each other in an efficient manner (Jara et al., 2014; Bak et al., 2015). For example, a weather app may use a location service coupled with a geo co-located temperature sensor service to provide a localised view of the environment to a user. In order to simplify the discussion, in this work, we will abstract devices to specific services provided by them—for example, *a television could be abstracted to a thing that provides display and audio services*. However, this yields to the problem of managing complex service interactions. Though human cognition is good at solving complex tasks, obtaining a bird's-eye-view of a network formed by these services is not feasible for humans. Furthermore, the complexity of the problem is exacerbated when the numbers of devices increases as the number of services provided by them increases too, thus even more services and interactions to govern. Therefore, any scalable solution to govern interactions in IoT should provide ways to seamlessly integrate new devices—hence new services—and expose them to the interactions space.

It is also important to note that when such services are used by humans—or other participating services—their interactions occur under varying constraints. This is due to a variety of reasons: *(a)* services have their own obligations and prohibitions in context; *(b)* services are owned and managed by different users and organisations, thus multiple con-

*Dr. Sensoy thanks to the Scientific and Technological Research Council of Turkey (TUBITAK) for its support under grant 113E238.

[2]http://www.gartner.com/newsroom/id/3165317

Goynugur E., De Mel G., Sensoy M., Talamadupula K. and Calo S.
A Knowledge Driven Policy Framework for Internet of Things.
DOI: 10.5220/0006200402070216
In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*, pages 207-216
ISBN: 978-989-758-220-2

straints could be placed on a single service; *(c)* dynamism in the environment, and the changes in preferences and goals could abruptly change constraints; and *(d)* constraints placed on a service could affect the functionality of another service.

In traditional systems, policies are typically used to govern such interactions, but in the IoT arena, we need expressive policy languages and efficient reasoning procedures that can *(a)* define high-level policies and refine them to device- and service- level policies in context; *(b)* detect policy violations and conflicts automatically; *(c)* automatically propose resolutions to conflicts when discovered; and *(d)* scale well in dynamic networks in terms of policy refinement, conflict detection and resolution with minimal human intervention. There is a multitude of policy frameworks—some with rich policy representations (Sensoy et al., 2012), and some targeting pervasive environments (Kagal et al., 2003). However, they are either not scalable or expressive enough with respect to the cost of execution—or reasoning about constraints—to be effective in IoT environments. The flexibility and the power of a policy management framework is to a large degree determined by the expressiveness and computational efficiency of its policy representation (Uszok et al., 2003). i.e. OWL-POLAR (Sensoy et al., 2012) uses OWL-DL, in which worst-case complexity of consistency checking and conjunctive query answering is NEXPTIME-complete (Baader et al., 2002).

Inspired by these observations, we present a framework that could be used to build IoT applications at scale which adhere to a set of governing rules set by the users and the environments in which they are deployed. To achieve this goal our framework first provides facilities to devices to advertise their capabilities on the network, and make them discoverable. Our framework then utilises an effective knowledge-based approach to represent high-level policies, efficient and scalable mechanisms to refine those policies to service level policies, automatic mechanisms to detect conflicts when enforcing service level policies, and state-of-the-art mechanisms to automatically resolve such conflicts. Specifically, the policy language is based on OWL-QL (Fikes et al., 2004), which supports efficient and scalable query re-writing mechanism for reasoning so that conflicts could be detected in polynomial time (i.e., `PTime`), and space-wise in many cases `LogSpace` or even $\mathtt{AC}_0$ for some specific classes of problems, and a planner-based set of techniques to resolve conflicts automatically using a polynomial amount of space (i.e., `PSpace`).

The rest of the paper is structured as follows: In Section 2 we provide an illustrative scenario, and use it to ground our discussions throughout the paper. In Section 3, we provide preliminaries to the proposed policy language, and in Section 4 we formalise our policy representation and present the policy management framework. In Section 5 we discuss the implementation of the framework with respect to the illustrative scenario. We discuss related work briefly and sketch the future directions for our research in Section 6, and conclude the document in Section 7 by providing final remarks.

2 ILLUSTRATIVE SCENARIO

Let us assume that a smart home is equipped with an intelligent doorbell amongst its many devices. A doorbell is typically tasked with notifying the household inhabitance when new events occur—e.g., when the doorbell is pressed, it could make a noise or send a message to a handheld device. Let us also assume that in association with the smart home hub is an interactive interface in which occupants of the house can enforce such conditions on the devices in context. Now, let us assume that the occupants have enforced a collection of such policies on the doorbell and a couple of such policy examples are *notify when the doorbell is pressed by an audio alarm*, and *if not responded in 15 minutes, send a message to the registered mobile phone*. We, now assume that the dynamics of the household have changed and there is a baby in the house. Now the occupants of the house place an extra policy on the smart hub to state that *no device should make noise when the baby is sleeping* —this is due to the current sleeping pattern of the baby which is monitored by another sensor. When this policy gets refined and applied to the doorbell, we have a conflict—i.e., the doorbell is obliged to make a noise, but what happens if the baby is sleeping?

Though simple—yet intuitive—the above scenario advocates for the need to have a policy framework that is agile enough to address the ever changing policy needs of the users, while providing efficient reasoning mechanisms quickly find conflicts and resolve them. In order to model such environments, we need effective domain modelling languages, and in the next section, we introduce one such language.

3 PRELIMINARIES

In this section, we introduce the language constructs to ground our policy representation.

3.1 Background: OWL-QL

Typically, languages grounded on expressive semantics focus on providing support for modelling complex relations and descriptions. As descriptions become more complex, the reasoning task becomes more complex too, thus requiring more computing power. Therefore, not all languages are suitable for IoT applications, which must handle very large volumes of instance data, ideally, with low power consumption. As stated in (Motik et al., 2008) OWL 2 QL (OWL-QL) specifically targets such applications, in which query answering is the most important reasoning task. In OWL-QL, sound and complete conjunctive query answering can be performed in LogSpace with respect to the size of the data (i.e., assertions), and polynomial time algorithms can be used to implement the ontology consistency and class expression subsumption reasoning problems (Motik et al., 2008). This made OWL-QL the perfect semantic language to base our policy language.

OWL-QL includes most of the main features of other ontology languages; however, it compromises some expressiveness to gain performance. Since the OWL 2 profiles are defined as syntactic restrictions without changing the basic semantic assumptions, in the OWL 2 QL profile, it was chosen not to include any construct that interferes with the Unique Name Assumption (UNA)—i.e., with the absence of the UNA, it would have had higher reasoning and query answering complexities. However, this also brings restrictions to OWL-QL such as no cardinality restrictions nor functionality constraints (Artale et al., 2009). For example, it is not possible to make a statement like *a room can only have one temperature*.

OWL-QL depends on the Description Logic $\mathsf{DL\text{-}Lite}_R$ (Artale et al., 2009). The complexity of logical entailment in most of the Description Logics is EXPTIME (Baader et al., 2002). Calvanese *et al.* (Calvanese et al., 2007b) proposed $\mathsf{DL\text{-}Lite}_R$, which can express most features in UML class diagrams with a low reasoning overhead—i.e., data complexity of AC_0 for ABox reasoning. It is for this reason that we base our policy framework on $\mathsf{DL\text{-}Lite}_R$ (to be referred to as DL-Lite in the rest of the paper); below we provide a brief formalisation of DL-Lite to ground the subsequent presentation of our model.

3.2 Representation and Semantics

A DL-Lite knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. Axioms of the following forms compose $\mathcal{K}$: (a) *class inclusion axioms*: $B \sqsubseteq C \in \mathcal{T}$ where B is a basic class $B := \mathsf{A} \mid \exists R \mid \exists R^-$, C is a general class $C := B \mid \neg B \mid C_1 \sqcap C_2$, A is a named class, R is a named property, and R^- is the inverse of R; (b) *role inclusion axioms*: $R_i \sqsubseteq P \in \mathcal{T}$ where $P := R_j \mid \neg R_j$; and (c) *individual axioms*: $B(\mathtt{a}), R(\mathtt{a},\mathtt{b}) \in \mathcal{A}$ where $\mathtt{a}$ and $\mathtt{b}$ are named individuals. Description Logics have a well-defined model-theoretic semantics, which are provided in terms of interpretations. An *interpretation* I is a pair $(\Delta^I, \cdot^I)$, where Δ^I is a non-empty set of objects and $\cdot^I$ is an *interpretation function*, which maps each class C to a subset $C^I \subseteq \Delta^I$ and each property R to a subset $R^I \subseteq \Delta^I \times \Delta^I$.

3.3 QL-based Policy Language

In this work we present our OWL-QL based policy language so that the policy reasoning framework can exploit OWL-QL's efficient and powerful query answering mechanisms (Fikes et al., 2004). We recall that an OWL-QL ontology consists of a TBox and an ABox. Concepts, properties, and axioms that describe relationships between concepts form the TBox of an ontology. We note that an ABox may be very large and volatile, while a TBox is small and static; OWL-QL allows ABox to be stored in a relational database and is organised based on the schema defined by the TBox. Each class in TBox is represented as a table, whose entries are the instances of the class. Similarly, each object or datatype property in TBox is represented as a table, whose entries are $\langle subject, object \rangle$ or $\langle subject, data \rangle$ pairs, respectively.

Table 1: An example TBox for an OWL-QL ontology.

An OWL-QL TBox
$Sleeping \sqsubseteq State$
$Awake \sqsubseteq State$
$Awake \sqsubseteq \neg Sleeping$
$Baby \sqsubseteq Person$
$SoundNotification \sqsubseteq Sound \sqcap Notification$
$TextNotification \sqsubseteq Notification$
$Speaker \sqsubseteq Device$
$Doorbell \sqsubseteq Device$
$\exists playSound \sqsubseteq Device$
$PortableDevice \sqsubseteq Device$
$MobilePhone \sqsubseteq PortableDevice$
$\exists hasSpeaker \sqsubseteq \exists playSound$
$MediaPlayer \sqsubseteq \exists playSound$
$TV \sqsubseteq \exists hasSpeaker \sqcap \exists hasDisplay$
$MakeSound \sqsubseteq Action \sqcap \exists playSound$
$Notify \sqsubseteq Action$
$NotifyWithSound \sqsubseteq MakeSound \sqcap Notify$
$Baby \sqsubseteq Person$
$SomeoneAtDoor \sqsubseteq Event$

We borrow syntax and semantics from the DL-Lite (Calvanese et al., 2007a) family to illustrate our TBox. For example, the statement: $Computer \sqsubseteq ElectronicDevice$ means that $Computer$ class is a subclass of $ElectronicDevice$; and the statement $ElectronicDevice \sqcap \exists playSound$ represents devices that can play sound. On the other hand, an ABox is a collection of extensional knowledge about individual objects, such as whether an object is an instance of a concept, or two objects are connected by a role (Calvanese et al., 2007a). In Description Logic, roles are binary relations between two individual objects—e.g. $livesIn(John, NewYork)$. In this statement, $livesIn$ is the role that connects $John$ and $NewYork$; example TBox and ABox of an OWL-QL ontology, which could be used to illustrate our scenario, are depicted in Table 1 and 2.

Table 2: Example ABox.

1	$Baby(John)$
2	$Person(Bob)$
3	$Doorbell(dbell)$
4	$Flat(flt)$
5	$inFlat(Bob, flt)$
6	$Sleeping(John)$
7	$SomeoneAtDoor(e1)$
8	$producedBy(e1, dbell)$

In the next section, we discuss the use of such an ontology to present policies in an IoT environment.

4 POLICY FRAMEWORK

In this section, we provide an overview of our policy framework. Specifically, we provide a formalism in which we ground our policy representation, and a framework that brings instances of such policies with OWL-QL reasoning to detect possible conflicts.

4.1 Policy Representation

Adhering to the policy formalism given in (Sensoy et al., 2012), we represent our policies by a tuple (α, N, $\chi : \rho$, $a : \varphi$, e, c) where

1. α is the activation condition of the policy;
2. N is either obligation (O) or prohibition(P);
3. χ is the policy addressee and ρ represents its roles;
4. $a : \varphi$ is the description of the regulated action; a is the action instance variable and φ describes a;
5. e is the expiration condition; and
6. c is the policy's violation cost.

ρ, α, φ, and e are expressed using a conjunction of concepts and properties from the underlying OWL-QL ontology—i.e., they are of the form $C(x)$ or $P(x,y)$, where C is a concept, P is either an object or datatype property, and x and y are either variables or individuals from the knowledge base. For example, using variables b and f, and the conjunction of atoms $Baby(?b) \wedge Sleeping(?b) \wedge inFlat(?b, ?f)$, describes a setting where there is a sleeping baby in a flat.

Table 3: An example prohibition policy.

$\chi : \rho$	$?d : Device(?d)$
N	P
α	$Baby(?b) \wedge Sleeping(?b) \wedge inFlat(?b, ?f) \wedge inFlat(?d, ?f)$
$a : \varphi$	$?a : MakeSound(?a)$
e	$Awake(?b)$
c	10.0

Table 3 illustrates a policy that prohibits devices from making sounds if there is a sleeping baby in the flat. It is important to note that, though the addressee of the policy is specified as a device (i.e., $Device(?d)$), concepts such as $Speaker(?d)$,$PortableDevice(?d)$, $MobilePhone(?d)$ are also included automatically while evaluating the policy by means of role inferencing through query re-writing.

4.2 Framework

Below, we propose our policy management framework that consists of three major components; a knowledge base, a QL reasoner, and a policy reasoner. In this section, we describe the components and discuss how they interact through the illustrative scenario highlighted in Section 2. Let us assume that there is a baby named *John* sleeps in a flat (i.e.,flt). John is defined as a baby boy in the KB and he has a wearable sensor that monitors his activities. The flat has an intelligent doorbell (i.e., $dbell$) that can create events and alert household when someone is at the door. At the moment, John is sleeping and Bob presses the button on the doorbell.

4.2.1 Knowledge Base (KB)

It contains instance data—both assertions and data coming from the sensors—and schema information related to the domain. For example, the ABox of our KB may contain the following set of assertions after the doorbell button is pressed: *(a)* type assertions such as $Baby(John)$, $Person(Bob)$, $Doorbell(dbell)$, $Flat(flt)$; *(b)* instance data such as $inFlat(Bob, flt)$, $Sleeping(John)$ added by the sleep monitor; and *(c)* $SomeoneAtDoor(e1)$, $producedBy(e1, dbell)$ added by the doorbell. We use the terms *OWL-QL ontology* and *knowledge base* synonymously in our context.

As the KB gets updated with new information, the policy reasoner queries the KB to check if a policy is activated or expired. Since policies are described using conjunctive ontology predicates, rewriting policy conditions to backend database queries is straightforward. We use the below discussed QL reasoner to perform consistency checks in a sandbox before actually updating KB with new information, since maintaining a consistent state of the world is paramount for reasoning with our OWL-QL ontology.

4.2.2 QL Reasoner

The QL reasoner is used to interpret role descriptions, activation conditions, action descriptions, and expiration conditions of a policy over the KB. However, directly querying the knowledge base may not reveal the inferred information that may be deduced through the TBox. For this purpose, query rewriting is used to expand the policy descriptions. Additionally, the KB must be in a consistent state with respect to the rules defined by the underlying ontology, since reasoning on an inconsistent KB might yield false results. Consistency checking is also performed by means of disjunctive queries that consist of conditions that may cause inconsistency based on the axioms in the TBox. The consistency and re-written queries can be cached to be re-used, as long as the TBox is not modified.

We have adopted OWL-QL package of Quetzal (Quetzal-RDF, 2016) for generating type inference and consistency check queries. Conjunctive formulae are converted into SPARQL queries, and are then fed into the reasoner; the re-written output queries are then converted into SQL. Quetzal uses Presto (Rosati and Almatelli, 2010) algorithm to do query reformulation.

4.2.3 Policy Reasoner

The policy reasoner utilises the above QL reasoner to keep a track of the normative state of the world—i.e., a list of active policies in that state of the world. Once a policy is rewritten through the QL reasoner, the expanded policy set is then used by the policy reasoner to create or delete active policy instances, or to detect conflicts between policies at design time. The policy reasoner uses activation and expiration conditions to determine if a policy is activated for a specific set of instances—e.g., in our scenario, the activation condition for the policy in Table 3 holds for the binding $\{?d = dbell, ?b = John, ?f = flt\}$, which is returned by the QL reasoner. Thus, an activated policy instance is created with the binding—i.e., an active policy *dbell is prohibited to perform MakeSound action until John is awake* is added to the normative state. The policy reasoner creates an active instance of a policy for each different binding. Whenever, expiration condition of an active policy instance holds, it is removed from the normative state—e.g., the active policy expires if John wakes up. Furthermore, some policies can be removed from the normative state when they are satisfied—e.g. active instance of the policy in Table 4 could expire after notifying a resident.

4.3 Query Re-writing

There are multiple algorithms to perform query rewriting. Any of these could be used in our implementation, however we use Presto (Rosati and Almatelli, 2010) algorithm, which is proven to compute the most efficient queries—i.e., it produces non-recursive datalog programs. The main ideas of the algorithm are eliminating existential join variables, and defining views corresponding to the expansion of basic concepts and roles. In other words, Presto tries to eliminate redundancies and to reduce the number of unions in conjunctive queries (UCQs) by reformulating them. Eventually, the remaining complexity of the generated query is handled by the database system. We refer the reader to (Rosati and Almatelli, 2010) for a detailed description of the algorithm to perform query rewriting.

Presto uses UCQs only in generated views, which would then replace the expanded concept or role in the original query. For example, if we had a query to get devices in a room $Q(?x) = Device(?x) \wedge inRoom(?x, livingRoom)$, it could generate the view $V = Device(?x) \vee Speaker(?x) \vee Doorbell(?x)$ for selecting devices. Then, the original query would be changed to $Q(?x) = V(?x) \wedge inRoom(?x, livingRoom)$. Furthermore, to illustrate the elimination of existential join variables, let us assume we have a query $Q(?x) = Television(?x) \wedge hasSpeaker(?x, ?y) \wedge Speaker(?y)$. In our example TBox, it is implied that every television has a display and a speaker and the query does not require variable $?y$. Thus, Presto algorithm re-writes query Q as $Q = Television(?x)$.

4.4 Policy Conflict Detection

When multiple policies get applied to a service, conflicts could occur. In our work, three conditions have to be met for two policies to conflict:

1. policies should be applied to the same policy addressee (e.g., same device, service, or individual);
2. one policy must oblige an action, while the other prohibits the same action; and
3. policies are active at the same time in a consistent world state.

Table 4: An example obligation policy.

$\chi : \rho$	$?d : Doorbell(?x)$
N	O
α	$SomeoneAtDoor(?e) \wedge producedBy(?e, ?x) \wedge$ $belongsToFlat(?x, ?f) \wedge hasResident(?f, ?p) \wedge Adult(?p)$
$a : \varphi$	$?a : NotifyWithSound(?a) \wedge hasTarget(?a, ?p)$
e	
c	4.0

Though it is trivial to figure-out policy conflicts within a specific state of the world, it is a non-trivial task to figure out if two policy may ever get into conflict at the design time.

In order to demonstrate the complexity in conflict detection—and to provide a solution—let us consider an obligation policy associated with our scenario. As shown by Table 4, the doorbell is obliged to notify the event with sound. In the remainder of the section, we denote the policies represented in Table 3 and 4 by $p1$ and $p2$, respectively. We can easily compute the fact that the modalities of $p1$ and $p2$ are in conflict, and the action description of $p1$ subsumes that of $p2$.

We use query freezing for both checking of an action description subsuming the other; and to detect if two policies can be active at the same time. Query freezing is a commonly used technique in database optimization(Motik, 2006; Ullman, 1997) to reduce the query containment problem in query answering.

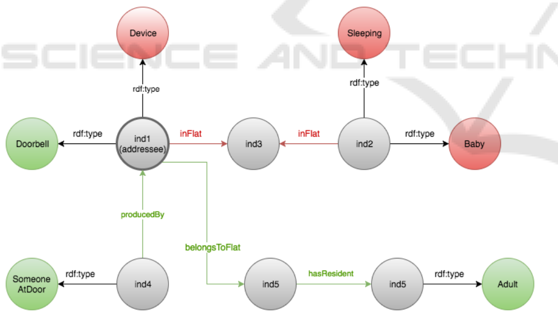

Figure 1: Sandbox of the world model created for the two policies $p1$ and $p2$ by freezing the activation conditions. (Green: Obligation, Red: Prohibition).

In order to prove that $p1$ and $p2$ are in conflict, all we need to do is to show a state of the world in which both $p1$ and $p2$ are active for the same addressee. For this purpose, we first create an empty ABox (a sandbox), and using query freezing techniques in (Motik, 2006), this sandbox is populated with the instances and relationships that appear in role and activation conditions of the policies. We first freeze role and activation conditions of $p1$, and populate the sandbox with the following set of assertions, which are the minimum requirements for $p1$ to be active: $Device(ind1)$, $Baby(ind2)$, $belongsToFlat(ind1, ind5)$, $inFlat(ind2, ind3)$.

We now freeze the role and activation conditions for $p2$. However, while doing so, we do not use a fresh individual for the policy addressee in $p2$ since for two policies to be in conflict, they should have the same policy addressee. The following assertions get inserted into the sandbox: $\{Doorbell(ind1)$, $SomeoneAtDoor(ind4)$, $producedBy(ind4, ind1)$, $hasResident(ind5, ind6)$, $Adult(ind6)\}$.

Since the resulting sandbox is consistent, it is apparent that $p1$ and $p2$ can be active at the same time. Thus, we can conclude that these policies are in conflict. We would have to make an additional check if $p1$ and $p2$ had expiration conditions since one of the policies might expire when the other one becomes active. Figure 1 illustrates the final state of the sandbox.

5 IMPLEMENTATION

In order to demonstrate the applicability of our framework, we designed and implemented the core of the proposal and tested it against the illustrated scenario; below we discuss the details of its architecture and the implementation.

5.1 Architecture of the Framework

The architecture of the proposed solution is depicted in Figure 2; it is composed of five main components: HyperCat Server, Device Coordinator, Knowledge Base, Policy Reasoner, and Planner. We implemented all these components along with the sensors and smart devices in our running example.

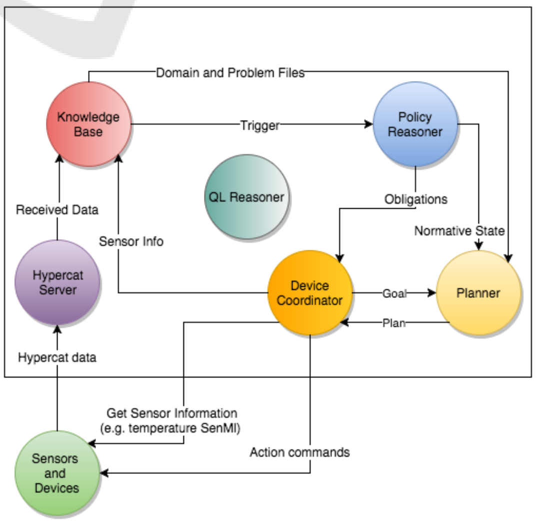

Figure 2: System Architecture: Policy-enabled IoT Framework.

5.1.1 HyperCat Server [3]

This is responsible for device registration and storing data that does not frequently change—e.g., capabilities (services) of devices. It is an open, lightweight JSON-based hypermedia catalogue for IoT devices, and stores information in triples. When used with an ontology, we can exploit this structure to exchange and store semantic information about devices and associated services. This method could act as a means for achieving semantic interoperability between heterogeneous IoT devices and services alike.

Devices that want to connect to the system have to register their capabilities through the server; furthermore, sensor devices may also stream collected data to the server. Our system considers all devices as a collection of services they provide. As mentioned in the introduction, a television could be modelled as a collection of a speaker, a video player, a photo viewer, a web browser, a notification tool and so forth. In addition, if a device needs to learn about the current state of the system, it can retrieve the necessary sensor data from the server. However, HyperCat does not specify an interface for devices to prioritise real time events like motion sensor or doorbell signals, thus we extended the protocol to provide an interface for the incoming events. Below code illustrates an example JSON request for a speaker, which only offers one service, to register itself and its capabilities.

```
item-metadata":[
    {"rel":"rdf-syntax-ns#type","val":"Speaker"},
    {"rel":"rdf-syntax-ns#about","val":"speaker1"},
    {"rel":"canPerformAction","val":"PlaySound"},
    {"rel":"inRoom","val":"room1"}],
items":[{"href":"http://speaker.ip/MakeSound",
    "i-object-metadata":[
     {"rel":"rdf-syntax-ns#type","val":"PlaySound" },
     {"rel":"rdf-syntax-ns#about",
      "val":"PlaySoundSpeaker1"}]}]}
```

5.1.2 Device Coordinator

This component acts as a mediator for devices that do not have enough computational resources to communicate with the Hypercat server and make decisions. It has three roles: (1) pull information from sensors; (2) compute action plans to achieve goals of devices; and (3) execute plans by sending action commands to the devices. Frequently updated data like sensor readings are stored in SenML (Jennings et al., 2016) files on the sensor according to HyperCat specification. Devices and sensors that are capable of communicating with HyperCat server push data to the server directly. However, data from other devices and sensors are polled by the Device Coordinator. It scans SenML files and finds the latest entry. The below JSON formatted text could be an output file of a sensor that measures temperature and humidity.

```
e":[{"n":"TemperatureOut","v":22.5,"u":"celsius","t":26},
    {"n":"TemperatureOut","v":295.6,"u":"kelvin","t":26},
    {"n":"HumidityOut","v":80,"u":"RH", "t":27}],
bn":"http://localhost/out.senml","bt":1320078429,"ver":1}
```

In our implementation, all active policy instances are stored in the Device Coordinator, which performs policy reasoning on behalf of the devices; individual devices do not know if they are prohibited or obliged to perform certain actions—this is a realistic assumption, especially for a swam of dumb devices. Whenever an obligation is activated, Device Coordinator runs the planner and executes the generated plan.

5.1.3 Knowledge Base

KB provides the domain descriptions—based on an ontology—and the initial state of the system to the planner, so that it can act, when policy conflicts are detected or an obligation policy gets activated.

5.1.4 QL Reasoner [4]

The QL Reasoner is implemented as described in Section 4.2.2. For example, when new information is received from the HyperCat Server—or the Device Coordinator—the QL reasoner simulates insertion of the new piece of information using a sandbox; consistency check query is then executed in the sandbox, and finally the new transaction is committed only if the world state is consistent. In addition to its described functions, the QL Reasoner is also used to rewrite conditions of planning domain actions which are used by the planner to verify plans to make sure they do not cause an inconsistent world states. In this way, we allow the planner to exploit semantic information about the domain. Additionally, the QL Reasoner checks for consistency at each step of the generated plans. It simulates each step as if they were executed by applying their effects to the current world state in a sandbox. As soon as one step causes inconsistency, that plan is discarded. Furthermore, the QL Reasoner modifies the static domain files by integrating the query rewrites to the planning process.

[3]https://bitbucket.org/egoynugur/iotserver

[4]https://bitbucket.org/egoynugur/iotql

5.1.5 Policy Reasoner

Currently, the policy reasoner reads policies from an XML file and stores each policy in the memory—code snippet below shows the XML representation of the policy in Table 4. Internally, the policy reasoner uses the QL reasoner to rewrite policies with respect to the roles, actions, and conditions. Active instances of policies are stored in the normative state and obligations are passed to the Device Coordinator. Furthermore, Policy Reasoner is used to compute accurate plan costs, as resulting plans may violate existing policies or new policies may become active during sub-steps of the plan.

```
<Policy Name="TestObligation" Addressee="?x"
       Modality="owlpolar:'Obligation'">
<AddresseeRole>sspn:'Doorbell'(?x)</AddresseeRole>
<Activation> sspn:'SomeoneAtDoor'(?e),
sspn:'producedBy'(?e, ?x), sspn:'Adult'(?p)
sspn:'belongsToFlat'(?x, ?f), sspn:'hasResident'(?f, ?p)
</Activation>
<Action var="?a"> sspn:'NotifyWithSound'(?a),
                  sspn:'hasTarget'(?a, ?p) </Action>
<Expiration></Expiration>
Cost>5.0</Cost> </Policy>
```

5.1.6 Planner [5]

Policy conflicts could be avoided by finding alternative ways of achieving obligations and delegating tasks to other devices. We used an artificial intelligence (AI) planner in our application to implement this approach. We used JSHOP2 as our planner as it was easy to integrate in to our application and its external calls allow us to simulate interleaved planning that requires the execution of non-deterministic actions such as *locate and search* actions.

After JSHOP2 receives an obligation from Device Coordinator, it starts planning and generates plan files. Each atomic action of the resulting plans is first checked by the QL Reasoner. If the action does not cause an inconsistency, the policy reasoner checks for possible violations and updates the normative state and the plan cost. It is important to note that new policies may become active or expire during the execution of a plan. If an action in a plan causes an inconsistent state of the world, that plan is discarded. Device Coordinator picks the plan with the lowest cost; let us note that the plan cost does include not only the cost of actions in the plan but also the cost of policy violations that will be made while executing the plan.

[5]https://bitbucket.org/egoynugur/iotplanner

5.2 Execution of the Demo

We simulated the environment with Java programs acting as sensors and devices along with an Android smartphone application behaving as the doorbell. In addition, the planning domain description has three actions; locate-people-in-flat, notify-with-visual, and notify-with-sound. Initial setup includes a locate service, two active policies, and two connected devices (a TV and a speaker) that can perform notify. Finally, two conflicting policies, which are represented in Table 3 and 4, exist in the initial setting.

The workflow starts with sensors sending requests to HyperCat Server to connect and to register their capabilities to the system. Then, we send a new event to the hub with the doorbell application. Once the hub receives a new doorbell event, an active instance of the obligation is created by the Policy Reasoner and forwarded to the Device Coordinator. Ideally, Device Coordinator should have a priority queue that sorts obligations based on how urgent they are. We use their violations costs to prioritize obligations. However, deadline fields and other metrics could be added and used to compute policies' priorities.

Next, we use the planner to find ways of notifying people in the house without violating no-sound policy. Within the current setting the planner generates three plans, which start by locating the people in the apartment. Two plans notify people with sound by using the speaker or the television. The last plan displays a visual message on the television. Before making the final decision, violation cost of the prohibition policy will be added to the plans making sound. Since the execution costs of other two plans are considerably increased, the plan which displays a message will be selected as the final choice due to its lower cost.

It can be seen from this example that interleaved planning is essential for the IoT domain. The planner cannot know if people are in the TV room without actually locating them. Hence, some actions have to be executed during planning to locate household first. To simulate interleaved planning, we created a service that returns locations of people in the house and it is called by using JSHOP2's external call feature.

6 DISCUSSION

The proposed framework is able to perform efficient reasoning and detect policy conflicts due to the properties of OWL-QL—i.e., expressiveness of OWL-QL, and database driven fast consistency checking and class expression reasoning—when compared to other languages from OWL family. However, the limitations of expressivity associated with the OWL-QL introduces limitations in expressing policies—e.g., number restrictions and functionality constraints are not supported by DL-Lite family of languages. For example, we cannot state that a room can only have one temperature in OWL-QL. However, we can use the numerical reasoning supported by planner software to overcome some of this limitation.

Additionally, when conflicts are detected, one could take several approaches to resolve conflicts automatically—e.g., one could use *(a)* policy violation costs—i.e., use violation costs to decide which policy has precedence over the other one. Also, instead of having predefined violation costs, it is possible to learn violation costs over time from user behaviour—e.g., if a user prefers hygiene over comfort; *(b)* user feedback through a reinforcement learning mechanism to resolve conflicts; or *(c)* AI planning techniques to automatically solve the conflicts.

Our current focus is on using AI planning techniques to automatically resolve policy conflicts. In an essence, AI planners could also be used to find alternative ways of accomplishing a goal—e.g., a conflict could be avoided instead of trying to resolve it. Additionally, in situations in which planner cannot avoid conflicts, we could apply other planning-based techniques to come-up with conflict resolution strategies.

We have presented a simple planner-based approach to resolve conflicts automatically. However, we can enhance the conflict resolution mechanism by considering user preferences as a heuristic, and reformulating policy conflict resolution as a planning problem. Furthermore, we can consider learning approaches wherein user preferences are captured as utility functions and be integrated with the planner domain. This is important in situation, especially in IoT enabled environments, where users have different preferences (e.g. the temperature setting in the room) and the system wants to come to a consensus among the users; this is the current focus of the work.

Though our current implementation has all the necessary backend components and services, intuitively authoring policies is a challenging tasks. This is mainly due to the steep learning curve users must go through to author policies with respect to ontologies. System assisted query writing (or generation) is an interesting research problem, but is out of the scope of this paper. However, we can get inspiration from techniques such as conversational aspects in query generation with respect to schema information (Hixon and Passonneau, 2013; Braines et al., 2014), and pragmatically aware query formulation (Viswanathan et al., 2015), to augment our system to address this issue. Furthermore, we plan to create an intuitive app which interfaces with the policy authoring framework supported by Google Now and Apple Siri APIs to provide a speech interface for policy authoring.

6.1 Policy Frameworks

There are a number of frameworks to represent and reason about polices: Ponder2 (Twidle et al., 2009), KAoS (Uszok et al., 2003), Rei (Kagal et al., 2003), and OWL-POLAR (Sensoy et al., 2012) to name a few. Ponder2 is a general-purpose object management system that can be used to enforce policies on entities (Twidle et al., 2009). However, it is not an ontology-based implementation and makes it difficult to infer policy conflicts. From the Autonomic Computing Systems domains, we find *goal-driven self assembly* framework such as Unity (Tesauro et al., 2004), and approaches that uses Self-Managed Cells in-conjunction with event buses (Keoh et al., 2007). These are based on Ponder2 policy language, thus policy reasoning and analysis are made difficult.

KAoS was the first policy framework to utilise an ontology-based approach to model and reason about policies (Uszok et al., 2003); the policies were defined using the concepts and object properties, thus it is not possible to use variables in policy descriptions. Therefore, KAoS is not expressive enough to represent policies we need for the IoT arena as we cannot represent policies such as *a speaker can notify a person if they are in the same room.* Rei (Kagal et al., 2003) is another effort towards an ontology based policy framework, especially for pervasive environments since the policy language is based on DL-Lite (Calvanese et al., 2007a). However, for the reasoning tasks, Rei uses Prolog—especially to specify relationships such as role-value-maps—which diminishes the open-world properties of OWL specification. Furthermore, Rei can only determine conflicts at run time, thus unable to prevent conflicts among high-level policies that we are interested in applying in the IoT setting. OWL-POLAR is a knowledge representation and reasoning framework for policies based on Description Logics (DL) (Sensoy et al., 2012). Though the effort is commendable, it is not tractable for IoT arena as DL-based reasoning is not efficient nor lightweight enough for IoT applications.

7 CONCLUSION

In this paper, we have proposed a lightweight framework to govern interactions in IoT environments. It allows users to *(a)* make their devices—and associated services—be discoverable in the network; *(b)* describe semantically rich policies; and *(c)* efficiently refine those policies to device- and service- level policies by means of efficient reasoning procedures and conflict detection mechanisms. In order to address the performance issues we have seen with other ontology-based frameworks, we have restricted our policy representation language to OWL-QL. This is because, OWL-QL supports efficient reasoning procedures whilst providing sufficient amount of expressiveness for IoT application. We then presented an implantation of this framework and showed its applicability through a smart home application. Additionally, we demonstrated how our system detects policy conflicts and uses an AI planner to find alternative means to achieve the goals so that policy violations are avoided. Lastly, we discussed the shortcomings of our work and described the lines of future research to address those shortcomings.

REFERENCES

Artale, A., Calvanese, D., Kontchakov, R., and Zakharyaschev, M. (2009). The dl-lite family and relations. *J. Artif. Int. Res.*, 36(1):1–69.

Baader, F., McGuiness, D. L., Nardi, D., and Patel-Schneider, P., editors (2002). *Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press.

Bak, P., Melamed, R., Moshkovich, D., Nardi, Y., Ship, H., and Yaeli, A. (2015). Location and context-based microservices for mobile and internet of things workloads. In *2015 IEEE International Conference on Mobile Services*, pages 1–8. IEEE.

Braines, D., Preece, A., de Mel, G., and Pham, T. (2014). Enabling coist users: D2d at the network edge. In *Information Fusion (FUSION), 2014 17th International Conference on*, pages 1–8. IEEE.

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2007a). Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated reasoning*, 39(3):385–429.

Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2007b). Tractable reasoning and efficient query answering in description logics: The dl-lite family. *J. Autom. Reason.*, 39(3):385–429.

Fikes, R., Hayes, P., and Horrocks, I. (2004). Owl-ql?a language for deductive query answering on the semantic web. *Web semantics: Science, services and agents on the World Wide Web*, 2(1):19–29.

Hixon, B. and Passonneau, R. J. (2013). Open dialogue management for relational databases. In *HLT-NAACL*, pages 1082–1091.

Jara, A. J., Lopez, P., Fernandez, D., Castillo, J. F., Zamora, M. A., and Skarmeta, A. F. (2014). Mobile digcovery: discovering and interacting with the world through the internet of things. *Personal and ubiquitous computing*, 18(2):323–338.

Jennings, C., Shelby, Z., and Arkko, J. (2016). Media types for sensor markup language (senml). https://tools.ietf.org/html/draft-jennings-senml-10. Accessed: 2016-10-02.

Kagal, L., Finin, T., and Joshi, A. (2003). A Policy Language for A Pervasive Computing Environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*.

Keoh, S. L., Dulay, N., Lupu, E., Twidle, K., Schaeffer-Filho, A. E., Sloman, M., Heeps, S., Strowes, S., and Sventek, J. (2007). Self-managed cell: A middleware for managing body-sensor networks. In *Proceedings of the 2007 Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking&Services (MobiQuitous)*, MOBIQUITOUS '07, pages 1–5, Washington, DC, USA. IEEE Computer Society.

Motik, B. (2006). *Reasoning in description logics using resolution and deductive databases.* PhD thesis, Karlsruhe Institute of Technology.

Motik, B., Grau, B. C., Horrocks, I., Wu, Z., Fokoue, A., and Lutz, C. (2008). Owl 2 web ontology language: Profiles. World Wide Web Consortium, Working Draft WD-owl2-profiles-20081202.

Palmisano, S. J. (2008). A smarter planet: the next leadership agenda. *IBM. November*, 6.

Quetzal-RDF (2016). Quetzal. https://github.com/Quetzal-RDF/quetzal. Accessed: 2016-10-02.

Rosati, R. and Almatelli, A. (2010). Improving query answering over dl-lite ontologies.

Sensoy, M., Norman, T., Vasconcelos, W., and Sycara, K. (2012). Owl-polar: A framework for semantic policy representation and reasoning. *Web Semantics: Science, Services and Agents on the World Wide Web*, 12(0).

Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., Kephart, J. O., and White, S. R. (2004). A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 464–471, Washington, DC, USA. IEEE Computer Society.

Twidle, K. P., Dulay, N., Lupu, E., and Sloman, M. (2009). Ponder2: A policy system for autonomous pervasive environments. In Calinescu, R., Liberal, F., Marín, M., Herrero, L. P., Turro, C., and Popescu, M., editors, *ICAS*, pages 330–335. IEEE Computer Society.

Ullman, J. D. (1997). Information integration using logical views. page pages. Springer-Verlag.

Uszok, A., Bradshaw, J. M., Jeffers, R., Suri, N., Hayes, P., Breedy, M. R., Bunch, L., Johnson, M., Kulkarni, S., and Lott, J. (2003). Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings of Policy*, Como, Italy. AAAI.

Viswanathan, A., de Mel, G., and Hendler, J. A. (2015). Pragmatics and discourse in knowledge graphs.