Tractable Policy Management Framework for Cognitive IoT

Emre Goynugur[a], Geeth de Mel[b], Murat Sensoy[a], and Seraphin Calo[c]

[a]Factuly of Engineering, Computer Science Department,Ozyegin University, Istanbul, Turkey
[b]IBM Research (UK), Warrington WA4 4AD, UK
[c]IBM Research, Yorktown Heights, NY, USA

ABSTRACT

Due to the advancement in the technology, hype of connected devices (hence forth referred to as IoT) in support of automating the functionality of many domains—be it intelligent manufacturing or smart homes—have become a reality. However, with the proliferation of such connected and interconnected devices, efficiently and effectively managing networks manually becomes an impractical, 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 state. Traditionally, policies are used to address the issue, but in the IoT arena, one requires a policy framework in which the language can provide sufficient amount of expressiveness along with efficient reasoning procedures to automate the management. In this work we present our initial work into creating a scalable knowledge-based policy framework for IoT and demonstrate its applicability through a smart home application.

1. INTRODUCTION

With the proliferation of connected and interconnected devices (hence forth referred to as IoT), automating the functionality of many domains—be it intelligent manufacturing or smart homes—have become a reality. According to *(a)* Gartner*, by 2020, there will be over 20 billion interconnected devices; and *(b)* IBM's executive report on *Device democracy, Saving the future of the Internet of Things*†, there will be over 100 billion devices interconnected in the next couple of decades in support of intelligent marketplaces, smart homes, intelligent manufacturing, and so forth.

However, with the explosion of such interconnected devices comes to problem of efficiently and effectively managing networks that they form; manually curating such networks is an impractical, if not an impossible task. This is due to a variety of reasons: *(a)* devices have their own obligations, and prohibitions in context; *(b)* devices are owned and managed by different organisations, thus constraints on them could change abruptly; and *(c)* though human cognition is good at solving complex tasks, obtaining a bird's-eye-view of such networks is not feasible. In traditional systems, policies are typically used to handle such, 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 specific devices in context; *(b)* detect policy violations and conflicts automatically; *(c)* propose resolutions to the decision maker; and *(d)* learn automatically about policy constraints in the operating environment by observing interactions and so on. This requires bringing multiple strands of artificial intelligence (AI) research in a cohesive manner—i.e., Cognitive Computing—to provide the capability.

There is a multitude of policy frameworks—some with rich policy representations.[1,2] However, they are not scalable nor expressive enough with respect to the cost of execution—or reasoning about constraints—over a given policy set, especially in the IoT setting. The problem is exacerbated when the numbers of devices increase as the number of services provided by them increase too—i.e., more constraints to govern. Therefore, a policy framework tailored for IoT applications should be able to *(a)* provide an expressive language for policy specification; *(b)* scale well in dynamic networks in terms of policy refinement and conflict detection; and *(c)* and propose sound resolutions such conflicts.

Further author information: (Send correspondence to E. Goynugur)
E. Goynugur: E-mail: emre.goynugur@ozu.edu.tr
G. de Mel: E-mail: geeth.demel@uk.ibm.com
*http://www.gartner.com/newsroom/id/3165317
†http://ibm.biz/device-democracy

In this paper we present our vision—and an initial implementation—for a policy framework that could be used to build cognitive IoT applications at scale. To achieve this goals, our framework utilises a knowledge base approach to represent high-level policies, efficient reasoning mechanisms to detect conflicts, AI-based automated mechanisms to resolve such conflicts. Specifically, the policy language is based on OWL-QL,[3] which supports efficient reasoning mechanisms and can be enhanced using the power of relational database query answering in the backend; policy conflicts are then detected using ontological reasoning, and planning and learning techniques are then proposed to solve such conflicts automatically.

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 a brief study of related work and preliminaries to the proposed policy language. In Section 4 we formalise our policy representation and present the policy management framework. In Section 5 we discuss the implementation of the framework w.r.t. the illustrative scenario. We sketch the future directions for the 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 equip 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 handle 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 the above scenario is simple, it shows 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, especially when the environments get complex.

3. PRELIMINARIES

In this section, we provide a brief review of policy frameworks, and introduces the language constructs we have used to ground our policy representation.

3.1 Policy Frameworks

There are a number of frameworks to represent and reason about polices: Ponder2,[4] KAoS,[5] Rei,[2] and OWL-POLAR[1] to name a few. Ponder2 is a general-purpose object management system that can be used to enforce policies on entities.[4] However, it is not an ontology-based implementation and makes it impossible to perform conflict detection. From the Autonomic Computing Systems domains, we find *goal-driven self assembly* framework such as Unity,[6] and approaches that uses Self-Managed Cells in-conjunction with event buses.[7] 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;[5] 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[2] is another effort towards an ontology based policy framework, especially for pervasive environments since the policy language is based OWL-*Lite*.[8] 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 knowledge representation and reasoning framework for policies based on Description Logics (DL).[1] 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.

3.2 Knowledge Representation and Reasoning

Cognitive systems are expected to *(a)* interact seamlessly with humans; *(b)* be transparent while making intelligent decisions; and *(c)* adopt to its environment. In order to do behave as such, these systems need knowledge about the domain and efficient AI algorithms to perform reasoning over the captured knowledge.

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.[3] We use the terms *OWL-QL ontology* and *knowledge base* synonymously, and in our context, a QL knowledge base (KB) consists of a *TBox* and an *ABox*. Concepts, properties, and axioms that describe relationships between concepts form the TBox of an ontology. We borrow syntax and semantics from the DL-Lite[8] 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.[8] 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 $NewYork$.

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 conflicts.

4.1 Policy Representation

Adhering to the policy formalism given in,[1] we represent our policies by a tuple (α, N, $\chi : \rho$, $a : \varphi$, e, c) where *(a)* α is the activation condition of the policy; *(b)* N is either obligation (O) or prohibition(P); *(c)* χ is the policy addressee and ρ represents its roles; *(d)* $a : \varphi$ is the description of the regulated action; a is the variable of the action instance and φ describes a; *(e)* e is the expiration condition; and *(f)* 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.

Table 1. 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 1 illustrates a simple 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.

4.2 Framework

Below, we propose a 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.

Table 2. 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

4.2.1 Knowledge Base (KB)

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: *(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. 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 a number of conjunctive ontology predicates, rewriting policy conditions to backend database queries is straightforward.

4.2.2 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 policies. Additionally, 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.

4.2.3 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 of a policy to determine if the policy is activated for specific set of instances—e.g., in our scenario, activation condition for the policy in Table 1 hold 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.

4.3 Policy Conflict Detection

When multiple policies get applied to a device, conflict could occur. In our work, three conditions have to be met for two policies to conflict: *(a)* policies should be applied to the same policy addressee (e.g., same device or individual); *(b)* one policy must oblige an action, while the other prohibits the same action; and *(c)* policies are active at the same time in a consistent world state. Though it is trivial to figure-out policy conflicts within a specific state of the world, it is non-trivial to reason 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 2, the doorbell is obliged to notify the event with sound. In the remainder of the section, we denote the policies represented in Table 1 and 2 by $p1$ and $p2$, respectively. We can easily compute the fact that the modalities of $p1$ and $p2$ are in conflict, and that the action description of $p1$ subsumes that of $p2$. 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,[9] 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: $\{Device(d0)$, $Baby(b0)$, $belongsToFlat(d0, f0)$, $inFlat(b0, f0)\}$ which gives the minimum requirements for $p1$ to be active. 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 gets inserted into the sandbox: $\{Doorbell(d0)$, $SomeoneAtDoor(e0)$, $producedBy(e0, d0)$, $hasResident(f0, p0)$, $Adult(p0)\}$. Since the resulting sandbox is consistent, it is apparent that both $p1$ and $p2$ can be active at the same time.

5. IMPLEMENTATION

In order to demonstrate the applicability of our proposed 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 Policy Framework

The architecture of the proposed solution is depicted in Figure 1; it constitutes of five main components: HyperCat Server, Device Coordinator, Knowledge Base, Policy Reasoner, and Planner.

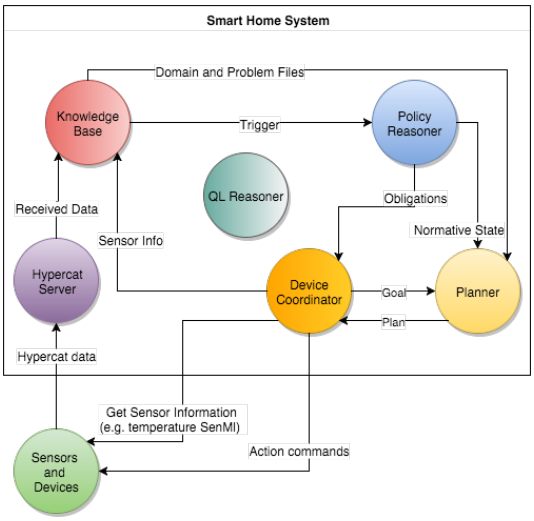

Figure 1. System Architecture: IoT Framework with Cognitive Policy Reasoning.

HyperCat Server is responsible for device registration and storing data that does not frequently change—e.g., capabilities of devices; however, it does not specify an interface for devices to prioritize real time events like motion sensor or doorbell signals, thus we extended the protocol to provide an interface for the incoming events. *Device Coordinator* 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. *Knowledge Base* 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 when the knowledge in the KB changes. Furthermore, it stores the most recent data from the sensors.

Policy Reasoner uses the *QL Reasoner* to rewrite policies w.r.t. the roles, actions, and conditions. Active instances of policies are stored in the normative state and obligations are passed to the Device Coordinator. *Planner* first try to avoid conflicts by finding alternative ways of achieving obligations and delegating tasks to other devices. We use JSHOP2[10] planner in this regard as it would especially allow us to simulate interleaved planning steps that requires the execution of non-deterministic actions such as *locate and search* actions. After the planner receives an obligation from the *Device Coordinator*, it starts generating plans, and the atomic action of those plans are 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. However, it is important to note that new policies may become activated or expired during the execution of a plan. If an action in a plan causes an inconsistent state of the world, that plan would be discarded. Device Coordinator then picks up the lowest cost plan, and carry-on with the execution.

5.2 Execution of the Demo

We simulated a smart home environment with Java programs acting as sensors and devices along with an Android smartphone application behaving as the doorbell. Initial setup also included a locate service, two active policies—i.e., the policies represented in Table 1 and 2, and two connected devices (a TV and a speaker) which can perform notification. In addition, the planning domain description has three actions; locate-people-in-the-flat, notify-with-visual, and notify-with-sound. The workflow starts with sensors sending requests to HyperCat Server to connect and to register their capabilities. Then using the app, we mimic the doorbell press and send a new event to the hub. Once the hub receives the new doorbell event, an active instance of the obligation is created by the Policy Reasoner and forwarded to the Device Coordinator. Next, planner is executed to find ways of notifying people in the house without violating no-sound policy. Within the existing settings, the planner generates three plans—i.e., two plans to notify people with sound by using the speaker and one plan to visually notification the occupants by means of the television. Before making the final decision, violation cost of the prohibition policy will be added to each plan. This would greatly increase the execution costs of audio plans, thus the visual plan would 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. 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 policy violation costs, it is possible to learn violation costs over time—e.g., if a user prefers hygiene over comfort, violation cost of a policy for maintaining hygiene overrides the one for avoiding discomfort; *(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. In this paper, we have presented a simple planner-based approach to resolve conflicts automatically. However, we have further enhanced the conflict resolution mechanism by considering user preferences as a heuristic in the planning problem formation; due to the page limitations of this submission we cannot provide all the details here, thus we refer the reader to our publication[11] for more details on preference based planning for conflict resolution.

7. CONCLUSION

In this paper, we have proposed a lightweight policy framework that specifically targets IoT application domain. It allows users to describe semantically rich policies, and systems to efficiently refine those policies to device-level 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.

ACKNOWLEDGEMENT

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

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

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

[3] Fikes, R., Hayes, P., and Horrocks, I., "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 (2004).

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

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

[6] Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., Kephart, J. O., and White, S. R., "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*, 464–471, IEEE Computer Society, Washington, DC, USA (2004).

[7] Keoh, S. L., Dulay, N., Lupu, E., Twidle, K., Schaeffer-Filho, A. E., Sloman, M., Heeps, S., Strowes, S., and Sventek, J., "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*, 1–5, IEEE Computer Society, Washington, DC, USA (2007).

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

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

[10] Nau, D., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F., "Shop2: An htn planning system," *Journal of Artificial Intelligence Research* **20**, 379–404 (2003).

[11] Goynugur, E., Talamadupula, K., de Mel, G., and Sensoy, M., "Automatic resolution of policy conflicts in iot environments through planning," *ICAPS 2016 Workshop on Scheduling and Planning Applications (SPARK)* (2016).