A SEMANTIC POLICY FRAMEWORK FOR INTERNET OF THINGS

A Thesis

by

Emre Göynügür

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in the
Computer Science Department

Özyeğin University
October 2018

A SEMANTIC POLICY FRAMEWORK FOR INTERNET OF THINGS

Approved by:

Associate Professor Murat Şensoy,
Advisor
Computer Science Department
Özyeğin University

Associate Professor Ali Fuat Alkaya
Computer Science and Engineering
Department
Marmara University

Assistant Professor Ahmet Tekin
Electrical and Electronics Engineering
Özyeğin University

Associate Professor Arzucan Özgür
Department of Computer Engineering
Boğaziçi University

Date Approved: 31 October 2018

Associate Professor Hasan Sözer
Computer Science Department
Özyeğin University

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 environments. 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 equipped to maintain a bird's-eye-view of the interaction space.

Motivated by this observation, in this thesis, we propose a semantic policy framework that *(a)* supports representation of high-level and expressive user policies to govern the devices and services in the environment; *(b)* provides efficient procedures to refine and reason about policies to automate the management of interactions; and *(c)* delegates similar capable devices to fulfill the interactions, when conflicts occur.

We then describe how to combine ontology-based policy reasoning mechanisms with in-use IoT applications to customize and automate device behaviors and discuss how the policy framework can be extended with data federation to handle diverse and distributed data sources. We demonstrate that smart devices and sensors can be orchestrated through policies in diverse settings, from smart home environments to hazardous workplaces, such as coal mines. Lastly, we evaluate our approach using real applications with real data and demonstrate that our approach is scalable under high load of data and devices.

ÖZET

Teknolojinin ilerlemesiyle, birbirine bağlı cihazlar (nesnelerin interneti, kısaca IoT), insanların çevreleriyle günlük etkileşimlerini otomatikleştirmek için hızlı ve etkili bir seçenek haline gelmektedir. Bununla birlikte, son yıllarda gözlemlediğimiz IoT uygulamalarının artmasıyla, insan-cihaz ve özellikle cihaz-cihaz arasındaki etkileşimlerin manuel olarak yönetilmesi imkansız olmasa bile pratik olmayan bir iş haline gelmiştir. Bunun nedeni, cihazların kendi yükümlülükleri ve kısıtlamaları olması ve insanların bu etkileşim alanının tümüne hakim olabilmek için donanımlı olmamalarıdır.

Bu gözlemden yola çıkarak, çevrede bulunan cihazları ve hizmetleri yönetmek için genel ve semantik regulatif kuralların temsil edilmesini destekleyici bir sistem önermekteyiz. Bu sistem düzenleyici kurallarla ilgili etkileşimlerin yönetimini otomatikleştirmek ve kurallarla ilgili akıl yürütmek için etkili prosedürler sağlamakla birlikte, bu kurallar arasında zıtlık meydana geldiğinde, yükümlülükleri yerine getirmek için benzer yetenekli aygıt veya servisleri kullanmayı hedefler.

Daha sonra cihaz davranışlarını özelleştirmek ve otomatikleştirmek için ontolojiye dayalı akıl yürütme mekanizmalarının kullanılmakta olan IoT uygulamaları ile nasıl birleştirileceğini ve farklı ve dağıtılmış veri kaynaklarını aynı anda kullanabilmek için veri federasyonu ile düzenleyici kurallar sisteminin nasıl genişletilebileceğini anlatıyoruz. Akıllı cihazların ve sensörlerin, akıllı ev ortamlarından kömür madenleri gibi tehlikeli işyerlerine kadar çeşitli ortamlarda düzenleyici kurallar yoluyla nasıl yönetilebileceğini gösterdikten sonra, son olarak, yaklaşımımızı gerçek uygulamaları kullanarak değerlendiriyor ve yaklaşımımızın yüksek veri ve cihaz yükü altında ölçeklenebilir olduğunu tartışıyoruz.

ACKNOWLEDGEMENTS

This research is supported by the Newton Fund and the Scientific and Technological Research Council of Turkey (TUBITAK) under grant 116E918.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

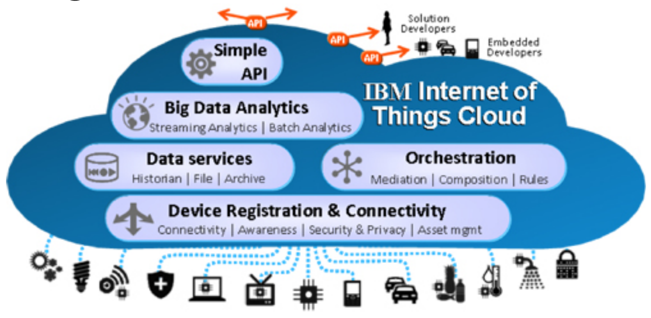
INTRODUCTION

The rapid growth of the Internet of Things (IoT) paradigm has fostered new tracking and sensor technologies, which have made the deployment of large numbers of sensors cheaper and more efficient. This has, in turn, made it more convenient to measure, infer and understand environmental indicators, from delicate ecologies and natural resources to urban and hazardous environments [3]. In recent past, we have seen an emerging trend wherein IoT systems are deployed to solve a wide range of complex problems in many industries: from providing independent living for aging [4, 5] (i.e., health care) to improving worker safety in dangerous environments such as mining [6]. IoT is fast becoming a reliable and cost effective means to automate daily activities for people and organizations.

In order for the above systems to function effectively, it is mandatory that an IoT-enabled system supports functionality for devices, and especially the services supported (or exposed) by those devices, to interact with each other in an efficient manner [7, 8]. For example, a smoke detector in a smart home environment may invoke a sprinkler's water spraying capability to thwart a perceived fire unless the cooker indicates that the smoke detected is in a acceptable range for grilling food. This interconnection among devices not only yields to the need for effectively representing such interactions, but also to the problem of efficiently managing them. 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.*

A common way of achieving some form of autonomy in distributed systems is

Figure 1: IBM's IoT Cloud Architecture[1]


API
Solution Developers
API
API
Embedded Developers
Simple API
IBM Internet of Things Cloud
Big Data Analytics
Streaming Analytics | Batch Analytics
Data services
Historian | File | Archive
Orchestration
Mediation | Composition | Rules
Device Registration & Connectivity
Connectivity | Awareness | Security & Privacy | Asset mgmt

using policies, which define and manage the behaviors of system components without human interaction and without modifying their source code [9, 10, 11]. Policies might resemble event-condition-action rules at first, yet they are more expressive. Policies can be seen as an external list of instructions that define a software application's business logic. Therefore, policies stand as an intuitive and principled way for regulating actions and interactions among smart objects in the IoT paradigm. Principles of policies and normative reasoning are widely studied in Law literature [12, 13] and they have attracted significant attention from the AI community to regulate the actions of autonomous intelligent agents [14].

Figure 1 illustrates IBM's cloud architecture [1] for IoT and its components. In this thesis, we are primarily interested in the *Orchestration* component. We do not particularly target IBM's cloud architecture, however we use this figure to highlight the focus of this thesis.

1.1 Motivation

Policies (or norms) are soft constraints that regulate the actions of autonomous entities such as humans through permissions, prohibitions, and obligations [15]. They are classified as *soft constraints*, since they can be violated without making the system

[1]`http://ibm.biz/iot-cloud`

unpredictable. Policies can be applied in every IoT setting where actions of intelligent entities should be regulated by one or more authorities. We predominantly observe two types of policies—*obligations* which mandates actions, and *prohibitions* which restrict actions [16] within the IoT domain. A policy could be a simple statement of intent—e.g., *do not make sound, if the baby is sleeping* for a smart home environment or policies could be used to modify the behaviors of other entities in the environment—i.e. policies could be used to program an intelligent farming application with respect to the changing environmental conditions and the status of the crops (e.g. *start watering, if the humidity level is low*).

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

It is essential that multiple policies apply to a device in order to cover the diversity of management functions and of management domains [17]. However, there is typically a potential for a conflict, whenever multiple policies act upon a device. For example, let us assume that in a smart home environment there are two policies: if someone rings the doorbell, then there should be a notification; and if the baby is sleeping, devices should not make any sound. In this scenario, a conflict occurs if someone rings the doorbell while the baby is sleeping. We use this simple scenario as a running example through the rest of the thesis and introduce more complex examples in later chapters.

Though human cognition is good at solving complex tasks, obtaining a bird's-eye-view of a network formed by heterogeneous services and managing them effectively are 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. Consequently, managing these devices—and their interactions—manually has become an impractical task, if not impossible. Thus, the policy system has to automatically handle exceptions when obligations and prohibitions overlap due to unforeseen situations.

In case of a conflict, a resolution strategy, which is commonly a pre-defined rule, is applied to determine the policy with the higher priority. i.e. is it more important to notify the residents or to keep the baby asleep. As IoT systems gain more complex capabilities – especially capabilities to learn, reason, and understand their environments and user needs – one can consider applying policy techniques to handle conflicts found in the IoT environment. However, the dynamism of IoT environments when compared with traditional systems must be taken into account; this necessitates more intelligent automation techniques for policy conflict management. For instance, there might be different actions that achieve a similar effect (e.g., *a doorbell event could be executed through a ringtone, by a message on the television, or through the smart home assistant*). Ideally, a policy system should choose to execute an alternative action or a composition of actions to avoid or minimize policy violations.

The flexibility and the power of a policy management framework depends to a large degree on the expressiveness and computational efficiency of its policy representation [18]. The challenge is to find a balance between the expresiveness of the policy language, the efficiency of policy reasoning mechanisms, and ease of use [11]. It is difficult to quantify these notions that highly depend on the target domain. Experiments with people can be conducted to evaluate the expressiveness and ease of use. Below we discuss these terms as desired features of a policy framework with respect to the requirements of IoT.

1.1.0.1 Expressiveness

A policy framework needs to be expressive enough to describe the concepts, relationships and conditions used in the IoT applications. There is, however, a trade-off between the expressiveness of a representational language and its computational tractability [19], which generally determines the complexity of the reasoning process. Without being expressive a policy language may not be able to regulate complex system behaviour [20], yet expressiveness can be compromised to meet the performance requirements of IoT applications as long as the language is able to represent the policies of the application.

1.1.0.2 Computational Efficiency

The increase in the number of connected devices and sensors do not only increase the generated data, but also increase the available services and policies in the system. Therefore, methods used to reason about policies and to resolve conflicts must scale well with the data size and the number of devices. Otherwise, the policy system will become impractical, when new devices and sensors are introduced to the system. In this thesis, we only focus on the scalability of the conflict resolution task and the policy reasoning task that includes maintaining a list of active policies and detecting policy conflicts. Building a scalable IoT system (i.e. network, communication platform etc.) is not in the scope of this work.

1.1.0.3 Ease of Use

As smart objects become an increasingly important part of our lives, we intend policies to be authored by non-technical people, preferably by domain experts or end users. For example, if the framework is being used at a mining facility, a health and safety expert should write the policies to ensure workers' safety. However, different users might formalize policies differently and validating the formalizations of policies may require additional analysis or tools (i.e. graphical or speech interface etc.). Even though there is on going work on this subject, we do not address these issues in this thesis.

1.2 Hypothesis

Inspired by these observations, we present a semantic 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. We utilize techniques based on knowledge representation (KR) to represent high-level policies, efficient and scalable mechanisms to refine those policies to service level policies, automatic mechanisms to detect conflicts at design time when enforcing service level policies, and an AI planner to automatically resolve such conflicts. Below we state our hypothesis for the thesis.

> Existing policy frameworks some with rich policy representations [21, 22], others targeting pervasive environments [23, 24, 25] are either computationally intensive or are not expressive enough (i.e. policies do not have expiration conditions or frameworks are not capable of semantic reasoning) to be effective within the IoT domain. An approach based on OWL-QL can effectively find the balance between the epxressivity and computational efficiency that is required by IoT applications.

Specifically, the policy language is based on OWL-QL [26] (QL), 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`). We believe the OWL-QL based policy language will also help us with the following aspects:

1.2.0.1 *Ease of Adoption*

Many of today's IoT applications operate by collecting data from sensors and devices into a central hub that makes the decisions. Moreover, it is a common practice among those applications to store data in multiple different databases (i.e. streams, relational and non-relational databases etc.) that rely on varying schemata. Augmenting these systems with semantics require re-designing of databases to conform certain rules, which would not be feasible for many applications. Thus, a policy system should provide the means for its integration into in-use applications without affecting their existing components and their interactions.

An alternative to implementing a semantic knowledge base with a strict schemata is using Ontology-based data access (OBDA) methods. OBDA turns a relational database into a knowledge base by mapping the concepts and properties of an ontology over the target database. Thus, it is possible to define a high level vocabulary to author policies over multiple data sources, when OBDA is used together with data federation methods.

1.2.0.2 *Conflict Resolution via Planning*

Conflicts among policies occur when prohibitions and obligations get applied to the same action of a device or a service at the same time. Most existing conflict avoidance strategies offered by existing policy frameworks are *static*, i.e., they use pre-defined rules that don't make adaptive decisions. Users cannot be expected to manually foresee and find solutions to resolve all such conflicts. Furthermore, by providing information in real-time, all the sensors and tags available make it possible for us to model the application environment in a detailed and up-to-date manner. In order to provide an automated solution to this problem, we propose and implement a mechanism which minimizes the policy violations by automatically reformulating the conflict resolution as an AI planning problem. Since QL is less expressive than

Planning Domain Definition Language (PDDL) [27], it is possible to model policies along with QL inference rules using off-the-shelf planners.

Even though we are concerned with all the aforementioned notions, realistically addressing or evaluating all is too ambitious for a single PhD work. For example, developing an intuitive user interface for authoring policies, validating the formalization of policies, or analyzing how much of the policies in the IoT domain can be represented with our policy language are beyond the scope of this thesis. However, we believe these ideas could be easily incorporated into our framework. In the concluding chapter of this document, we discuss the limitations of our framework and highlight our plans to address them.

1.2.1 Main Contributions

The work described in this thesis makes the following contributions:

1) We introduce a policy language that provides efficient mechanisms with the right amount of expressivity to describe and reason about policies. The approach allows to detect conflicts at design time and to use high-level concepts to refine policies to individual devices or services.

2) We provide a conflict resolution strategy which utilizes a general purpose AI planner. The planner determines the best course of action (minimizing or avoiding policy violations) by making use of available services at the run time. The planner can also be used as a conflict detection mechanism.

3) We describe how ontology based data access and data federation methods can be adopted to implement the policy framework and to consolidate in-use IoT applications that use multiple diverse data sources.

We, further, show the applicability of the framework through two different use cases; a smart home application and an intelligent mine solution.

1.3 Thesis Outline

The outcome of this research is a policy framework that utilizes 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. The framework can also be integrated into in-use IoT applications without affecting their existing workings. The rest of this document is structured as follows.

Chapter 2 The chapter presents an overview of the preliminaries to the proposed policy framework. It also surveys and discusses the previous research done in policy frameworks and policy conflict resolution.

Chapter 3 The chapter introduces the formalization of our policy representation and the conflict detection algorithm using an illustrative scenario. We, then, discuss the implementation of the policy management framework with respect to the same scenario.

Chapter 4 The chapter motivates the need for planning and introduces the use of automated planning techniques in our system to automatically resolve conflicts, and other extensions that could be explored. Next, we present means to automatically translate semantic policy representations to the planning domain, and we evaluate the proposed approach.

Chapter 5 In this chapter, we discuss how to do policy reasoning over distributed data through a case-study. We specifically focus on health and safety policies in underground mines and show how planning could be useful in this new domain. Finally, we evaluate the performance of our implementation using official mining safety regulations and a database that we obtained from a mining company.

Chapter 6 This chapter discusses the limitations of our work and possible solutions to those limitations. Then, we sketch some ideas for future extensions and conclude the document by highlighting the contributions of our work.

CHAPTER II

BACKGROUND AND RELATED WORK

This chapter outlines the technical background, semantics, and the related work that the contributions of this thesis rely on. Several other topics and methods, such as planning, ontology based data access, and data federation will be discussed in the following chapters.

2.1 Background

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 [28]. In this section, we provide the theoretical foundations of our policy framework and introduce the language constructs to ground our policy representation.

2.1.1 Knowledge Representation

Formal representation of knowledge enables computers to solve complex tasks (e.g. predicting drug side effects) and interact with their environment in a more intelligent fashion (i.e. context aware computing). Knowledge representation methods provide the means for automated reasoning, which is the ability of deriving new knowledge from the data and making inferences. The most common knowledge representation methods involve logic, rules, and semantic nets [29].

Typically, knowledge representation languages grounded on expressive semantics focus on providing support for modeling complex relations and descriptions. As descriptions become more complex, the reasoning task becomes more complex too, thus requiring more compute power. Therefore, not all languages are suitable for IoT applications, which must handle large volumes of instance data, ideally, with low power consumption.

2.1.1.1 OWL-QL

OWL 2 QL (OWL-QL) [30] is a sub-language of Ontology Web Language (OWL) [31], which specifically targets applications that produce large volumes of instance data and 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 [31].

OWL-QL includes the 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. UNA is a simplifying assumption that enforces entities to have different names [32], however QL's logical foundation does not make this assumption. Thus, this brings restrictions to OWL-QL such as no cardinality restrictions nor functionality constraints [33]. For example, it is not possible to make a statement like *a room can only have one temperature.* It is also not possible to use individual equality assertions (**SameIndividual**).

OWL-QL depends on the Description Logic $\mathsf{DL\text{-}Lite}_R$ [33]. The complexity of logical entailment in most of the Description Logics is EXPTIME [34]. Calvanese *et al.* [35] 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 $\mathsf{DL\text{-}Lite}$); below we provide a brief formalisation of $\mathsf{DL\text{-}Lite}$ to ground the subsequent presentation of our model. We refer the reader to the comprehensive introduction [1] of OWL 2 Profiles for further information.

2.1.1.2 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*: $B(\mathsf{a}), R(\mathsf{a}, \mathsf{b}) \in \mathcal{A}$ where a and b are named individuals. Description Logics have a well-defined model-theoretic semantics, which are provided in terms of interpretations. An *interpretation* $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set of objects and $\cdot^{\mathcal{I}}$ is an *interpretation function*, which maps each class C to a subset $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and each property R to a subset $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.
Using a trivial normalization, it is possible to convert class inclusion axioms of the form $B_1 \sqsubseteq C_1 \sqcap C_2$ into a set of simpler class inclusions of the form $B_1 \sqsubseteq B_i$ or $B_1 \sqsubseteq \neg B_j$, where B_1, B_i, and B_j are basic classes ([36]). For instance, during normalisation, $B_1 \sqsubseteq B_2 \sqcap \neg B_3$ is replaced with $B_1 \sqsubseteq B_2$ and $B_1 \sqsubseteq \neg B_3$. With a normalized TBox, tractable semantic reasoning and query answering can be implemented using query rewriting techniques ([37]).

We borrow syntax and semantics from the DL-Lite [35] 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 [35]. In Description Logic, roles are binary relations between two individual objects—e.g.

$livesIn(John, NewYork)$.

2.1.1.3 Language Definition

Table 1 presents the constructs that can be used in *QLtiny* (OWL-QL) language. IName, CName, and PName are used for individual names, class names, and property names. The complete list of supported axioms can be found in the official documentation [2].

Table 1: Definition of *QLtiny* language [1, 2]

Axiom	::=	**CL** $\sqsubseteq$ — **CR(IName)** — **P (IName, IName)**
CL	::=	**CName** — $\top$ — $\bot$ — $\exists$ **P.**$\top$
CR	::=	**CName** — $\top$ — $\bot$ — **CR** $\sqcap$ **CR** — $\neg$ **CL** — $\exists$ **P.CR**
P	::=	**PName** — $\mathbf{PName}^{-}$

2.1.2 Reasoning

Reasoning in ontologies and knowledge bases is one of the motivations why a specification needs to be a formal one [38]. It is required for deriving facts that are not explicitly defined in a knowledge base (KB). It makes it possible to keep a consistent KB, to infer types of individuals (i.e. a dog is an animal), and, essentially, to answer queries over ontologies. In the context of policy frameworks reasoning can be used to develop a policy language with high level concepts and properties, to find active and expired policies, and to detect conflicting policies.

We base our policy language on OWL-QL so that the policy reasoning framework can exploit OWL-QL's efficient and powerful query answering mechanisms [26]. 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 large and volatile, while a TBox is small and static.

OWL-QL is designed to store the data ($\mathcal{A}$) in a relational database and to reduce the reasoning process to *query answering* by *query rewriting*. Query answering is performed by first computing a rewriting of the initial query with respect to the intensional part of the ontology ($\mathcal{T}$), and then evaluating this re-written query over the database. This property of OWL-QL is called first-order rewritability of conjunctive queries, which allows perfect reformulations to be represented with SQL. The worst-case complexity of the size of re-written queries is exponential [35]. Hence, the resulting query might consists of hundreds or thousands of queries given a large ontology [39]. There are various different rewriting algorithms [40, 41, 42, 43, 44, 45] to produce an efficient reformulation in a reasonable amount of time.

These algorithms produce non-recursive datalog programs. The main idea is to eliminate existential join variables, while defining views corresponding to the expansion of basic concepts and roles. In other words, efficient reformulation algorithms try 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 can illustrate the query answering process with the following scenario. Let us assume that an ontology defines students at a university as; undergrad students $UndergradStudent \sqsubseteq Student$ and anyone who has an advisor $Student \sqsubseteq \exists hasAdvisor$. Query to retrieve all students $Student(?s)$ will be rewritten to include both axioms. $Student(?s) \sqcup UndergradStudent(?s) \sqcup hasAdvisor(?s, ?someone)$. The subclass axiom with the *hasAdvisor* property can be used to discover students in the KB, even if they are not explicitly defined as students.

2.1.3 Knowledge Base Implementation

It is not trivial to store and reason over large volumes of ontology data, due to the dynamic nature of ontologies and the data heterogeneity [46]. Depending on the reformulation algorithm and the ontology, the schema of the target relational database must conform to certain specifications. A naive approach is to represent each class in TBox 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. However, this approach suffers heavily from joins (unions of subclass axioms) which are generated by the rewriting algorithm. There are efficient frameworks like Quetzal [1] that provides efficient mechanisms [46] to store Resource Description Framework (RDF) [47] in relational databases. A comparison of various methods on managing large volumes of RDF data can be found in this survey [48].

It may not always be possible to mold the target database schema into the desired structure or the ontology data might reside in different data sources. For example, it is a common practice in IoT applications to store less volatile data (i.e. machine configurations, employee information etc.) in relational databases, while maintaining more frequent data like sensors readings in a non-relational database. In such cases ontology based data access methods can be utilized to convert the target database into a QL knowledge base. We discuss this method further in Chapter 5

2.2 Related Work

2.2.1 Policy Frameworks

There is a multitude of policy frameworks; some with rich policy representations and some targeting pervasive environments. We begin this section by reviewing the policy frameworks, which we found to be the most relevant to our work, in chronological order. Finally, we discuss policy-based control mechanisms developed for Wireless Sensors Networks (WSNs).

[1] `https://github.com/Quetzal-RDF/quetzal`

2.2.1.1 Rei

Rei [21] is a policy language designed to control the security functionality and behaviors of pervasive computing applications. The language is based on deontic concepts [49] and implements four types of policies; rights, prohibitions, obligations, and dispensations (obligations that no longer apply). Rei allows policies to be defined for roles, groups, and individuals. Furthermore, it is possible to delegate policies to other users. e.g. a lab owner can grant access to her students for the lab machines without requiring any action from another authority.

The policy engine of Rei is implemented with Prolog [50]. It takes an RDF-S [51] ontology as an input that must provide domain specific information based on the concepts in Rei's ontology. Prolog provides Rei the reasoning power and enables Rei to specify role value maps that are not directly possible in OWL. i.e. people who are the same age.

The ability of using role value maps prevents Rei from detecting policy conflicts in design time. In Rei's context, conflicts can occur between right-prohibition and obligation-prohibition policies and the policy engine requires meta-policies to resolve these conflicts. Meta-policies are rules about other policies that are used for specifying if positive policies hold precedence over negative policies or not.

Rei is an expressive policy language, however RDF-S policies must be based on the concepts of Rei ontology. Furthermore, Prolog uses depth-first search to match predicates. Therefore, the policy engine is prone to infinite loops, if this specific issue is not addressed in Rei's implementation.

2.2.1.2 KAoS

KAoS [18] is one of the most advanced ontology-based policy frameworks. KAoS services are described using an OWL-DL [33] domain ontology by linking concepts to the Kaos's generic framework ontology. The underlying ontologies make it possible to represent groups, actors, actions, and other necessary concepts such as computing resources of actions. KAoS supports obligation and authorization policies, which can either be positive or negative.

As the policy language is limited to description logics, KAoS is able to detect policy conflicts at design time by performing subsumption reasoning. e.g. if one description subsumes to other. The reasoning algorithm is implemented using Java Theorem Prover (JTP)[2]. However, this makes it impossible to use variables (i.e. to bind values to object properties) in policy descriptions. This can be considered as a limitation of a policy language, since it is not possible to represent statements like *two speakers in the same room*.

Authors addressed the above issue in a later work [52] by extending their implementation with role-value maps [53] by adopting syntax from Semantic Web Rules (SWRL) [54]. This enables KAoS to represent role-value maps along with parametric values like thresholds. Authors integrated a separate reasoner into their JTP reasoner to detect conflicts in the presence of role-value maps. However, subsumption reasoning in DL with arbitrary role-value maps is undecidable [34].

Finally, KAoS also offers policy harmonization to resolve detected policy conflicts. The algorithm makes use of metrics like update time of policies and priority values determined by policy authors to decide which of the conflicting policies is more important. If the reasoner cannot come to a conclusion with the given rules, an error is prompted and an action from a human administrator is required.

2.2.1.3 Protune

Protecting user privacy from web services is particularly important in the IoT domain, as information sources greatly increase in number, provide sensitive information (e.g.

[2]`http://www.ksl.stanford.edu/software/JTP/`

health data), and become even more integrated into our lives. PRovisional TrUst NEgotiation [55] (Protune) is a rule-based policy framework that is designed to regulate access control in web service applications. It allows users to define policies about how much information they want to reveal under certain conditions. Protune's rule language extends PAPL [56] and PEERTRUST [57] and developed specifically to handle trust negotiations.

Protune still depends on lightweight ontologies, which models concepts used in policies, the relationship between those concepts, and the evidence required to prove their truth. Protune can detect conflicts and inconsistencies in metapolicies, however, this is a computationally exhaustive task [55]. Protune uses meta-policies for making trust negotiation decisions. To our knowledge, Protune does not offer a mechanism to detect and resolve conflicts.

2.2.1.4 Ponder2

Ponder2 [25] -the successor of the famous policy framework Ponder [10]- comprises a general-purpose object management system, which is not built on top of an ontology. Ponder2 targets pervasive environments and addresses the limitations of using its predecessor in such applications. Ponder2 aims to build a simple, easy-to-use, and scalable policy framework that does not rely on any other infrastructure services and supports dynamically adding new functionality.

Unlike Ponder, Ponder2 does not rely on a Policy Decision Point. It is implemented as a self-managed cell [58], which is a set of hardware and software components forming an administrative domain that is able to function autonomously and is capable of self-management [25]. SMCs can be thought of as virtual machines that are able to make all kinds of policy decisions. This ability of Ponder2 makes it suitable for a wide range of environments like WSNs, robots, mobile phones etc.

Ponder2 policies are written in a policy language called PonderTalk, which is inspired by Smalltalk [59]. PonderTalk supports authorization and obligation policies, as well as the use of variables in policy descriptions. There is a simplified and scaled-down version of Ponder2 called Finger2 [23], which is an embedded policy-framework that specializes in WSNs. As Ponder2 does not offer any semantic descriptions of policy actions, addressees or activation conditions, it is not possible to use inferred knowledge or to reason over Ponder2 policies.

Since it is not possible to perform conflict detection on PonderTalk policies, authors have chosen to put constraints on the expressions and actions of the language by defining a formal Alloy specification. Alloy is a first-order logic based declarative specification language for expressing complex structural constraints and behavior in a software system. This enables Alloy Analyzer [60, 61] to perform such analysis on policies. Ponder2 resolves conflicts at design time by selecting the more specific policy with respect to the type hierarchy. It is also possible to specify global rules (i.e. meta policies) for resolving conflicts instead of selecting the most specific policy. However, Ponder2 does not offer any adaptive conflict resolution strategies at run time.

Ponder2 is designed for developing self-contained applications in pervasive environments. Thus, the policy language and its framework is tightly coupled with the software on these devices, which makes Ponder2 difficult to integrate into in-use applications. Furthermore, a centralized decision point might be more beneficial to some IoT systems (especially for resolving conflicts), if the policy engine makes decisions by maintaining a bird-eye-view of the system.

2.2.1.5 Proteus

Proteus [62] can be considered as a hybrid policy framework that combines rule-based and ontology-based methods to add context awareness to access control policies. Proteus prioritizes the contextual information such as location, time, current activity etc over the identity and the role of subject. While frameworks like KAoS and Rei uses contextual information as a means to filter policies and use identity and role to refine policies to subjects, Proteus considers context information as the primary basis to refine policies.

The policy language of Proteus is based on DL and uses DL reasoning to determine active contexts. Further, the language is extended with horn clauses and Logic Programming (LP) reasoning to support role-value maps. This approach is adopted from authors' previous work [63]. Proteus can also detect policy conflicts at design time and allows definitions of constraints to avoid conflicting situations. Although the approach is promising, Proteus do not focus on resolving conflicts and it would be difficult to scale rules and DL reasoning due to their computational complexity.

2.2.1.6 OWL-POLAR

OWL-POLAR [22] (OWL-based POlicy Language for Agent Reasoning) is an OWL 2.0 knowledge representation and reasoning framework for policies. It is developed to offer an expressive policy language with decidable reasoning mechanisms. Later in this chapter, we discuss the policy representation and reasoning methods of OWL-POLAR in more detail. Policies in OWL-POLAR can be used to oblige, prohibit, or to permit an action. Even though OWL-POLAR does not provide an authorization policy type, these modalities can represent access control policies.

The policy language of OWL-POLAR is based on OWL-DL and its reasoning capabilities depends on the Pellet [64] ontology reasoner. Policies are described using concepts and properties that exist in the input ontology. Moreover, it is possible to use both specific individuals and variables in the policy descriptions, which are not only event-condition-action rules with priorities, but they also have an expiration condition to determine if the policy is no longer active.

In order to support role-value maps in DL, policies are represented with conjunctive semantic formulas, which can be translated into SPARQL [65] queries. These queries are executed using the SPARQL-DL [66] engine of Pellet. This allows OWL-POLAR to detect policy conflicts and idle policies (policies that can never be active) using query containment methods. *Query freezing* [67, 68] is used to reduce the query containment problem -if a conjunctive formula subsumes the other- to query answering in DL [67, 68]. The policy language also restricts comparing two different data types in policy descriptions to preserve the decidability of the reasoning process.

Furthermore, OWL-POLAR is the first policy framework that uses AI planning [69] for automatic conflict resolution. This approach tries to minimize (avoid if possible) policy conflicts by finding an alternative route rather than resolving it for good. Planning would be especially useful in the IoT domain, as connected devices and sensors increase the number and variety of available actions and make it possible to model their environment. The complexity and performance of planning significantly relies on the constraints put on the planning domain [69]. The worst-case complexity of planning is PSPACE-complete in the set-theoretic representation [70].

Although OWL-POLAR is powerful and expressive, reasoning with Pellet and OWL-DL cannot be considered as efficient or lightweight for IoT applications. This performance loss is mainly caused by the expressiveness of OWL-DL. Finally, OWL-POLAR offers a conflict resolution method that uses an HTN planner, yet it cannot automatically formulate conflicts as planning problems and external analysis of generated plans are required to select the right plan.

2.2.1.7 Summary

In this section we reviewed the most notable policy frameworks and discussed the trade-off between expressivity and reasoning power. Ontologies-based methods are widely adopted by policy frameworks to model context with an expressive language. These frameworks are extended with logic programming languages to support the use of variables in policy descriptions and to be able to reason about policies. i.e. to

Table 2: Comparison of the policy frameworks

Policy Framework	Language	Semantic Reasoning	Conflict Detection	Conflict Resolution	Reasoning Complexity	Central Decision Point
Rei	OWL-Lite + Prolog Variables	Yes	Not in design time	Meta-Policies	NP-Complete	Yes
KAoS	OWL-DL + Role-Value Maps	Yes	Java Theorem Prover	Meta-Policies	NP-Complete	Yes
Ponder2	PonderTalk	No	Alloy Analyzer	Meta-Policies	NP-Complete	No
OWL-POLAR	OWL-DL	Yes	Query Freezing	HTN-Planning	NEXPTIME	Yes
Proposed Framework	OWL-QL	Yes	Query Freezing	PDDL Planners	LOGSPACE	Yes

detect policy conflicts. However, this extension makes reasoning process undecidable.

Ponder2 does not use ontology-based methods and it is probably the most developed platform among the introduced frameworks. It is designed as a self-contained system and targets pervasive environments. Each self-managed cell in the system is able to make all kinds of policy decisions. PonderTalk, the underlying language, cannot detect policy conflicts, thus Alloy Analyzer is used to perform such analysis.

OWL-POLAR differs from other ontology-based frameworks, as it uses conjunctive formulas, which are converted to SPARQL queries to do DL reasoning with Pellet. It aims to offer an expressive policy language, while preserving the decidability of the reasoning process. OWL-POLAR can detect both policy conflicts and idle policies. Table 2 depicts a comparison of the policy frameworks.

There are various conflict resolution algorithms adopted by these policy frameworks. We can roughly group them as meta-policies, which are policies about policies used to determine the more important policy. Finally, OWL-POLAR's conflict resolution method differs from others as it uses an AI planner to automatically resolve those conflicts. The planner dynamically utilize the available resources to avoid or minimize conflicts. We present a more comprehensive review of conflict resolution algorithms in the next section.

2.2.2 Policy Conflict Resolution

The majority of the policy applications adopt a rule-based or an ontology-based reasoning to detect policy conflicts. In addition to the frameworks discussed in the previous section, some of the more recent work [71, 72, 73, 74] on policies–including smart homes, web services, and multi-agent environments–employ similar methodologies (ontologies, rule-based approaches, csp solvers) for detecting and resolving conflicts. Thus, in this section, we focus on conflict resolution methods.

In our context, conflicts among policies occur when negative and positive policies get applied to the same action of a device or a service at the same time. For example, a conflict may arise if a policy might require an air conditioner to keep the temperature at 25, while another prohibits it from working when there is an open window. Furthermore, there are other types of conflicts such as two people who prefer different temperatures or two policies trying to open and close the same window at the same time. However, conflicts between positive policies (i.e. management of conflicting obligations [75]) and the policies of different agents are not in the scope of this work.

Vasconcelos et al. [15] represents policies as atomic formulae whose variables may have arbitrary associated constraints. They provide a conflict detection mechanism that uses unification (first order logic and theorem proving) and a conflict resolution strategy called policy ”curtailment” based on constraint satisfaction. Policy curtailment is the process of manipulating the constraints of policies to avoid overlapping values of variables. Policies are simply re-written in such a way that these policies do not get activated at the same time.

Lupu et al. [17] comprehensively discusses policy conflicts and resolution strategies in their survey. These strategies include; *a)* prioritizing negative policies over positives and vice versa *b)* assigning explicit priorities to each policy *c)* computing the distance between a policy and the managed objects: Priority is given to the policy applying to the closer class in the inheritance hierarchy. e.g. Speaker is closer to the ”Bluetooth Speaker” class than ”Device”. *d)* choosing the specific policy over the more general one (sometimes policies with same precedence might conflict) *e)* using meta policies, which are the management policies.

We can even add more conflict resolution strategies like choosing the more recent policy or the policy of a higher authority [22]. However, these examples should be sufficient to show that the most of the resolution strategies are pre-defined rules to select the more significant policy. We believe that such static strategies fail to exploit the true potential and the dynamic nature of IoT, since they do not consider options like delegating obligations to another actor or doing a composition of available actions.

CHAPTER III

POLICY FRAMEWORK

In this chapter, we formalize the representation of policies, and discuss the conditions for policy activation, expiration, and conflicts. We first provide an illustrative scenario and use it to ground our discussions throughout the thesis. Then, we introduce the terminology that will be used in the rest of this paper to represent policies.

3.1 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*. 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 modeling languages, and in the next section, we introduce one such language.

3.2 Policy Representation

We use OWL-QL, which is based on DL-lite family to represent and reason about policies. Even though DL-lite seems like a simple language, it allows expressive queries and efficient query answering over large instance data sets. By using OWL-QL as an ontology language, we exploit its efficient and powerful query answering mechanisms [26] in our policy language and reasoning framework. We provide an example TBox and ABox of an OWL-QL ontology, which could be used to illustrate our scenario, are depicted in Table 3 and 4.

Table 3: An example TBox for an OWL-QL ontology.

An OWL-QL TBox
$Awake \sqsubseteq \neg Asleep$
$Baby \sqsubseteq Person$
$Adult \sqsubseteq Person$
$TV \sqsubseteq \exists hasSpeaker \sqcap \exists hasDisplay$
$TV \sqsubseteq Device$
$DoorbellEvent \sqsubseteq Event$
$SoundAction \sqsubseteq Action \sqcap \exists playSound$
$Doorbell \sqsubseteq Device$
$SoundNotification \sqsubseteq SoundAction \sqcap \exists hasTarget$
$PortableDevice \sqsubseteq Device$
$MobilePhone \sqsubseteq PortableDevice$

Motivated by the work of [22], we formalize a policy as a six-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;

Table 4: Example ABox.

$Baby(Jane)$
$Baby(John)$
$Adult(Bob)$
$Doorbell(dbell)$
$Flat(flt)$
$hasResident(flt, Bob)$
$inFlat(John, flt)$
$inFlat(Jane, flt)$
$inFlat(dbell, flt)$
$Asleep(John)$
$Awake(Jane)$
$DoorbellEvent(e1)$
$producedBy(e1, dbell)$
$SoundNotification(playAudio)$

d) $a : \varphi$ is the description of the regulated action; a is the action instance variable 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. For example, using variables b and f, and the conjunction of atoms $Baby(?b) \wedge Asleep(?b) \wedge inFlat(?b, ?f)$, describes a setting where there is a sleeping baby in a flat. c is a numerical value determined by policy authors to reflect how important a policy is. We assume that all actions are permitted unless they are explicitly prohibited.

Table 5 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 $TV(?d)$ and $Doorbell(?d)$ are also included automatically while evaluating the policy by means

Table 5: Prohibition policy example: Sleeping baby.

$\chi : \rho$	$?d : \mathit{Device}(?d)$
N	P
α	$\mathit{Baby}(?b) \wedge \mathit{Asleep}(?b) \wedge \mathit{inFlat}(?b, ?f) \wedge \mathit{inFlat}(?d, ?f)$
$a : \varphi$	$?a : \mathit{SoundAction}(?a) \wedge \mathit{actor}(?a, ?d)$
e	$\mathit{Awake}(?b)$
c	10.0

Figure 2: Rewritten activation query.

```
p(d,f,b) :- Device(d), Baby(b), Asleep(b), inFlat(d,f), inFlat(b,f)
p(d,f,b) :- Doorbell(d), Baby(b), Asleep(b), inFlat(d,f), inFlat(b,f)
p(d,f,b) :- Television(d), Baby(b), Asleep(b), inFlat(d,f), inFlat(b,f)
p(d,f,b) :- hasSpeaker(d,_), hasDisplay(d,_), Baby(b), Asleep(b), inFlat(d,f), inFlat(b,f)
```

of role inferencing through query re-writing. The conjunctive semantic formulas can simply be translated to conjunctive semantic queries. Figure 2 depicts the expanded query required for this policy's activation.

Table 6, on the other hand, illustrates an obligation policy, which enforces doorbells to notify the residents using a sound action.

Table 6: Obligation policy example: Doorbell notification.

$\chi : \rho$	$?d : \mathit{Doorbell}(?d)$
N	O
α	$\mathit{DoorbellEvent}(?e) \wedge \mathit{inFlat}(?d, ?f) \wedge \mathit{producedBy}(?e, ?d) \wedge \mathit{hasResident}(?f, ?p) \wedge \mathit{Adult}(?p)$
$a : \varphi$	$?a : \mathit{SoundNotification}(?a) \wedge \mathit{actor}(?a, ?d) \wedge \mathit{hasTarget}(?a, ?p)$
e	$\mathit{gotNotifiedFor}(?p, ?e)$
c	5.0

3.3 Policy Activations and Expirations

A policy is activated for a specific set of instances that fulfill its activation condition. Likewise, an active policy instance expires if its expiration condition holds true or the goal of that policy is fulfilled. The conjunctive semantic queries that describe the activation and expiration conditions of policies are converted to SPARQL [65] queries, which can be re-written and evaluated using a QL reasoner over triple stores [76] or

Figure 3: Rewritten activation query. (SPARQL)

```
PREFIX  :   <www.ozyegin.edu.tr/sensoy/smart-home.owl#>
SELECT DISTINCT ?d ?b ?f
WHERE {
     { ?d  a  :Device }
   UNION
     { ?d  a  :Doorbell }
   UNION
     { ?d  a  :Television }
   UNION
     { ?d  :hasSpeaker  ?unbound_0 ;
           :hasDisplay  ?unbound_1 }
   ?d  :inFlat   ?f .
   ?b  :inFlat   ?f ;
       a  :Baby ;
       a  :Asleep
}
```

Figure 4: Rewritten expiration query. (SPARQL)

```
PREFIX  :   <www.ozyegin.edu.tr/sensoy/smart-home.owl#>
ASK
WHERE { :John  a  :Awake . }
```

relational databases [77]. This is a trivial conversion, since SPARQL queries are more expressive than conjunctive queries [77].

The re-written SPARQL query required for the activation of the policy in Table 5 can be seen in Figure 2. In our scenario, the activation condition for this policy holds for the binding $\{?d = dbell, ?b = John, ?f = flt\}$. As a result, an activated policy instance is created with this binding;"*dbell* is prohibited to perform *playAudio* action until *John* is *awake*".

Whenever the expiration condition of an active policy instance holds, that instance should be removed; e.g., the activated policy expires if the baby John wakes up. The devices would be allowed to use sound actions again, if there is no active instance of the prohibition policy. i.e. if there is no baby in the flat or both $John$ and $Jane$ are awake. Unlike creating active instances of a policy the existence of a single solution for expiration queries is sufficient for active instances to expire. Therefore, expiration conditions are translated into ASK queries as seen in Figure 4.

Some active policies expire when they are satisfied. For instance, obligation policies can expire after obligations are fulfilled. Let us consider the policy example in Table 6, which defines an obligation policy stating that a doorbell has to notify adult residents of a flat if someone rings the bell. In this case, since the *dbell* is ringed, the active policy "*dbell* is obliged to *notify* an *adult resident* of *flt* with *sound*" should be created. After notifying the targeted person *Bob* in the flat, the obliged action would be performed and the activated policy would be satisfied. Alternatively, there could be an expiration condition to keep that policy instance active until someone explicitly acknowledges the notification or the door is opened.

A deadline field can be added to the policy definition as a complimentary mechanism to expiration conditions, since certain actions have to be performed within a reasonable time frame. For example, there is no point in notifying the residents 5 hours later than the doorbell was ringed. However, introducing deadlines increases the complexity of the conflict detection problem and render the proposed conflict detection algorithm impractical. Even though deadlines are beyond the scope of this thesis, we believe ideas from temporal logic and planning [78, 79, 80] could be utilized.

3.4 Policy Conflicts

Policies may conflict under various settings [17, 81]. Detecting conflicts at design time helps with writing policies that less likely to conflict and recognizing a potential conflict scenario ahead of time. The most obvious example is when an obligation policy requires a prohibited action. The other examples of conflicts with different types include; two policies that require opening and closing the same window at the same time, two people with different temperature preferences sharing the same room, two actions that require the same resource for different purposes etc. However, in this work we only focus on the first conflict type, thus three conditions have to hold true for two policies to conflict:

a) Both 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.

c) These two policies should be active at the same time in a consistent world state according to the underlying ontology.

It is important to state here that unless an addressee has to violate one of its own policies to fulfill another one, there is no conflict. For instance, in our scenario, the doorbell is obliged to notify the household with sound due to one policy while the very same doorbell is prohibited to make any sound due to another policy. As it has to violate its own prohibition policy to fulfill its goal policy, these policies are considered to be in conflict.

Early detection of conflicts is particularly important in hazardous environments like underground mines, since unnoticed conflicts may cost money and time for companies and at worst lives. The success of the detection algorithm relies on the action descriptions of policies and the structure of the TBox. We note that the subsumption relation between the sound and notify with sound actions are explicitly defined in the TBox. For example, the conflict detection algorithm described below fails to capture the relation between two actions, if one of them *implicitly* subsumes the other. This situation generally occurs when a defined action is a composition of actions. Let us consider a scenario in which a policy requires all occupants of a building to evacuate in case of a fire. There might be another policy, which got activated long before the fire, prohibiting some occupants to use a door that might be crucial for the evacuation. In this scenario, *evacuate* action in the policy definition is a chain of actions that includes using a prohibited door. In the next chapter, we discuss how automated planners can be utilized both to minimize policy violations and to capture *implicit* conflicts.

3.4.1 Conflict Detection Algorithm

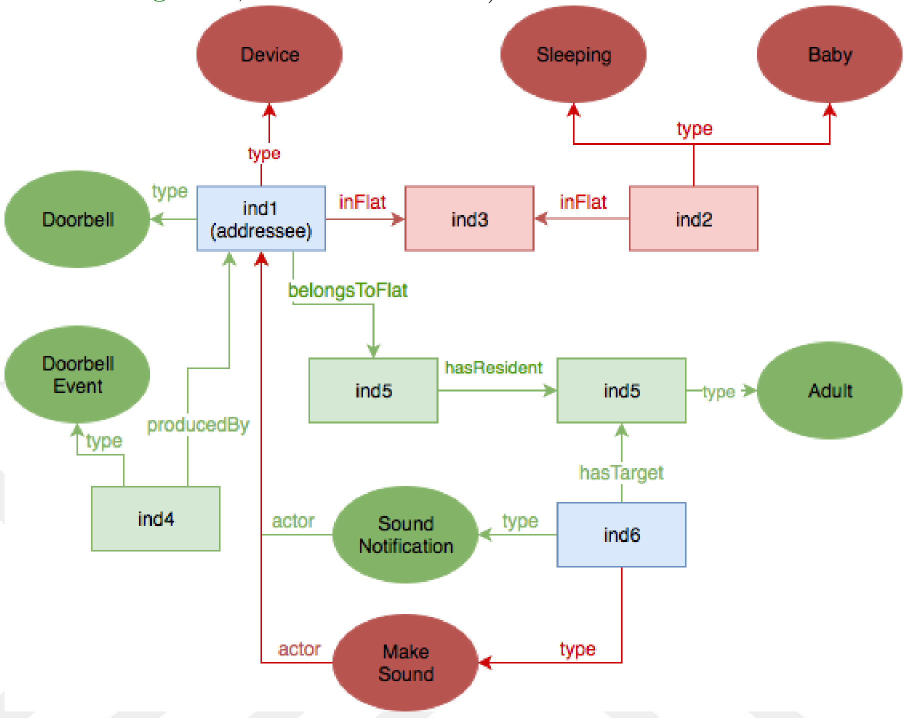
We can easily compute if modalities of two policies are in conflict. The tricky part of the conflict analysis is determining whether two policies can be active at the same time and get refined to the same action. Since conjunctive queries are used to describe actions in policy definitions, we need to check if one of the queries subsumes the other one. If there is a subsumption relation between action descriptions, that means these queries will have common tuples (action instances) in their result set.

The above mentioned query containment problem can be reduced to query answering in Description Logics using the standard technique of query freezing [67, 68]. However, subsumption reasoning becomes undecidable in the presence of arbitrary role-value-map [34, 22], thus datatype variables cannot be compared in our policy language. For example, $q(p)$:- $Person(p) \wedge hasAge(p, 25)$ is a valid query, yet $q(a, p)$:- $Person(p) \wedge hasAge(p, a) \wedge a \leq 25$ is not allowed in our policy language. In order to describe real IoT policies, unfortunately, it may be necessary to use variables in data properties and to place constraints on them, since a large portion of data is generated by sensors. We partly address this issue by shifting some of these constraints to OBDA mappings. This is further discussed in Chapter 5.

We describe the conflict detection algorithm using our running example. In the remainder of the section, we denote the policies represented in Table 6 and 5 by $p1$ (obligation) and $p2$ (prohibition), respectively. To test whether q_A subsumes q_B, a canonical ABox (sandbox) is created and populated with the instances and relationships that appear in role and activation conditions of the policies.

We first freeze the role, the activation condition, and the action description of $p1$, and populate the sandbox with the following set of assertions, which are the minimum requirements for $p1$ to be active: $\{Doorbell(ind1), DoorbellEvent(ind4), Adult(ind5),$ $producedBy(ind4, ind1), inFlat(ind1, ind3), hasResident(ind3, ind5), actor(ind6, ind1),$ $SoundNotification(ind6), hasTarget(ind6, ind5)\}$. We then query this sandbox with

Figure 5: The canonical state of the world generated by the conflict detection algorithm. (Green: Obligation, Red: Prohibition)



$p2$'s action description: $q(a,d)$:- $SoundAction(a) \wedge actor(a,d)$. Since the action description of $p2$ subsumes $p1$'s', this query holds true for the binding: $\{?a = ind6, ?d = ind1\}$.

Next, we 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: $\{Device(ind1)$, $Baby(ind2)$, $inFlat(ind1, ind3)$, $inFlat(ind2, ind3)$, $SoundAction(ind6)$, $actor(ind6, ind1)\}$. Now, the canonical ABox represents a world state, in which both policies are active and one of the action descriptions subsumes the other one. 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 5 illustrates the final state of the sandbox.

If $p1$'s action description does not subsume $p2$'s action, we swap the policies and restart the algorithm with a fresh sandbox. If there is no subsumption relation between actions, the algorithm terminates concluding no conflict. Figure 6 depicts the pseudocode of our conflict detection algorithm.

3.5 Implementation

In order to show the mechanics of our approach, we implemented a system that would act like a hub (central point) to govern the behaviors of all devices in a smart home environment. These devices include, and not limited to air conditioning systems, television, coffee maker, doorbell, laptop, smart phones, windows, cleaning robots etc. However, in this section, we only discuss the implementation of the policy reasoning mechanisms and leave the discussions of the conflict resolution process to the next chapter.

3.5.1 Architecture of the Framework

The architecture of our solution is depicted in Figure 7; 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.

3.5.1.1 HyperCat Server

HyperCat [82] Server 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.

Figure 6: The pseudocode of the conflict detection algorithm.

```
input: $\pi_A, \pi_B, \mathcal{T}_{in}$
output: bool

begin

  if !$checkModalityConflict(\pi_A, \pi_B)$
    return false

  $\mathcal{A}_C = createSandbox(\mathcal{T}_{in})$
  for i = [0 1]
    $p_1 = (i == 0)?\pi_A : \pi_B$
    $p_2 = (i == 0)?\pi_B : \pi_A$

    if i == 1
      $\mathcal{A}_{can} = createSandbox(\mathcal{T}_{in})$

    $freezeQuery(p_1.getActivation(), \mathcal{A}_C)$
    $freezeQuery(p_1.getAction(), \mathcal{A}_C)$

    $subsumed = executeQuery(p_2.getAction(), \mathcal{A}_C)$
    if $size(subsumed) > 0$
      break

  if $size(subsumed) == 0$
      return false

  $freezeQuery(p_2.getActivation(), \mathcal{A}_C)$
  $freezeQuery(p_2.getAction(), \mathcal{A}_C)$

  $instances_1 = executeQucery(p_1.getActivation(), \mathcal{A}_C)$
  $instances_2 = executeQuery(p_2.getActivation(), \mathcal{A}_C)$

  $instances_1 = removeExpiredInstances(instances_1, \mathcal{A}_C)$
  $instances_2 = removeExpiredInstances(instances_2, \mathcal{A}_C)$

  if $size(instances_1) == 0$ || $size(instances_2) == 0$
      return false

  return $isConsistent(\mathcal{A}_C)$
end
```

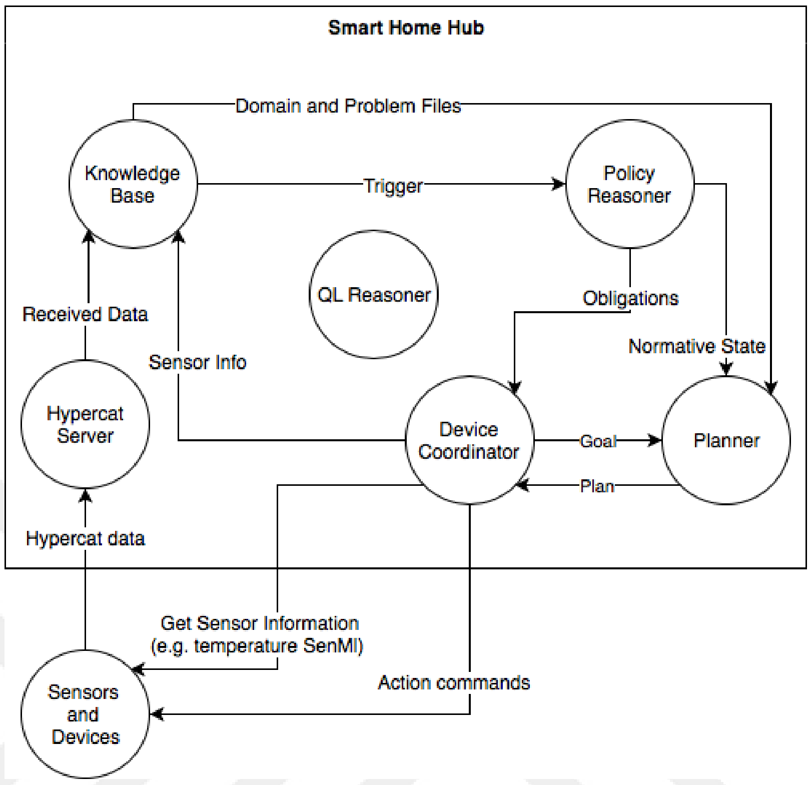


Figure 7: System Architecture: Policy-enabled IoT Framework.

Devices that want to connect to the system have to register their capabilities through the server; furthermore, sensors 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. Figure 8 illustrates an example JSON request for a speaker, which only offers one service, to register itself and its capabilities.

Figure 8: Example HyperCat request of a speaker.

```
{
  "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"
}]}]}
```

3.5.1.2 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 [83] 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.

In our implementation, all active policy instances are stored in the Device Coordinator, which performs policy reasoning on behalf of the devices; individual devices

Figure 9: SenML output example.

```
{
   "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
}
```

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.

3.5.1.3 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.

3.5.1.4 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. Updating knowledge bases is an error-prone process [84, 85] and the most common problem is *integrity constraints checking.* Even though efficient integrity checking methods have been developed [86, 87], we adopt a trivial solution to make the policy system work, since maintaining a knowledge base is not in the scope of this work. Whenever 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. 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.

The QL Reasoner is also used for integrating reformulation rules into the static domain files to allow the planner to exploit semantic information about the domain. We have adopted OWL-QL package of Quetzal [88] 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.

3.5.1.5 Policy Reasoner

The policy reasoner utilizes 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 reads policies from an XML file and stores each policy in

Figure 10: XML representation of a policy.

```
<Policy Name="NotifyDoorbell" Addressee="?d" Modality="Obligation">
    <AddresseeRole> Doorbell(?d) </AddresseeRole>
    <Activation> DoorbellEvent(?e), producedBy(?e, ?x),
                 Adult(?p), inFlat(?d, ?f), hasResident(?f, ?p)
    </Activation>
    <Action var="?a">SoundNotification(?a), hasTarget(?a, ?p), actor(?a, ?d)</Action>
    <Expiration> gotNotifiedFor(?p,?e) </Expiration>
    <Cost>5.0</Cost>
</Policy>
```

the memory—code snippet in Figure 10 shows the XML representation of the policy in Table 6. 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 substeps of the plan.

In the next chapter, we discuss the conflict resolution problem, present our planning approach and complete the implementation of this system.

3.6 Discussion

The policy 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.

Policies are represented using conjunctive semantic formulas. Although disjunctions in policy definitions do not increase the complexity of the policy reasoning, we decided not to include them for simplicity. For instance, the conflict detection algorithm would have to create multiple canonical states when comparing policies with disjunctions. Such policies can easily be converted to a set of policies described with only conjuntive formulas.

The numerical values associated with the violation costs of policies are set by policy authors. These costs are compared to make an automated decision in case of conflicts. However, it could be difficult for people to quantify the importance of a policy and set correct individual costs without some assistance. Moreover, the significance of these values are relative to each other, as we sometimes need to make a decision between more than two policies. Thus, we can consider learning approaches wherein user preferences are captured as utility functions, however it is not within the scope of this thesis. We discuss how conflicts are resolved in the next chapter.

We note that we are only concerned with the modality conflicts. Logical conflicts between two obligation policies are beyond the scope of this thesis. The conflict detection algorithm can detect policy conflicts at design time, if the subsumption relation between action descriptions of policies are explicitly defined in the TBox. Thus, the algorithm cannot capture the relation between two actions, if one of them is a composition of actions that includes the other. We discuss how this issue can be addressed using AI planners in the next chapter.

Furthermore, data properties can be used in the policy descriptions, yet data values cannot be compared. The reasoning process in the conflict detection algorithm becomes undecidable, when such constructs are allowed in the policy language. However, in Chapter 5, we exploit OBDA methods to reduce the need for data value comparisons in policy descriptions.

Though our current implementation has all the necessary backend components and services, intuitively authoring policies is a challenging task. 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 thesis. However, we can get inspiration from techniques such as conversational aspects in query generation with respect to schema information [89, 90], and pragmatically aware query formulation [91], to augment our system to address this issue.

CHAPTER IV

AUTOMATED CONFLICT RESOLUTION

Policy conflicts may arise between two given policies when the conditions outlined in the previous chapter are met. In such cases, it is essential for the system to devise a way to resolve the conflict and move forward. In this chapter, we first outline a way of posing this conflict resolution problem as a planning problem, and using automated planning technology to solve that problem instance. Then, we briefly discuss a preliminary proof-of-concept evaluation of the planning approach and complete the implementation of the policy system from the previous chapter.

4.1 Utilizing Planning to Resolve Conflicts

When compared with traditional IT systems, one of the major issues in managing IoT-based systems is the impracticality of using humans to configure, maintain, and manage all these connected devices, and the services associated with them. This is because services related to IoT are dynamic (especially in terms of availability), agile, and context sensitive. Thus, resolving policy conflicts is a critical part of policy based systems.

IoT systems are data-intensive and highly dynamic environments, in which new devices join and leave the system on the fly. Each connected device introduces new capabilities from a system's perspective. Systems' capabilities do not have to consist of only device actions, but they could also include web services. We believe that utilizing all available capabilities to completely or partially avoid policy conflicts could prove to be an effective conflict resolution mechanism. In order to resolve a conflict: *(a)* a composition of actions *(b)* a different action, which serves the same purpose *(c)* a different set of parameters, which is not prohibited, for the same action could be found.

Some of the policies are more important than others and, if critical circumstances arise that makes it impossible to satisfy all the constraints, it might become necessary to violate the less important ones. For instance, notifying a doorbell event may not be more important than waking up the baby or an alternative notification action can be just as useful. Let us assume that in addition the baby, the baby's parents are sleeping as well, and the doorbell is rung. Now, the system has to make a decision; to either violate the sound policy, or to not notify and ignore the visitor at the door. In this extended scenario, the planner needs to make a decision according to the violation costs set by the policy's authors. The real-world domains are more complex, thus there may be much more complicated scenarios in which multiple policies are active and the solution is much more complicated; this demonstrates the need for soft constraints of some nature, such as preferences.

The actors involved in IoT systems cannot act unilaterally because they do not possess complete context-awareness about the scenario; and they may be violating policy compliance by taking unilateral action. AI planners are commonly used for solving above described problems [92, 93]. Moreover, planners can answer to the performance requirements of IoT and cope with the dynamically changing environment. Thus, we think that using planners as a conflict resolution mechanism would enhance IoT systems by allowing connected devices to collaborate with each other and possibly available web services.

We propose using planners to solve conflicts at run time, which is important for two reasons; a previously found solution may not work the next time due to the changes in the environment or a better solution might become available. For example, let us assume that there is an available service which displays a message on the TV in our smart home scenario. The planner can use this action to notify the residents without making a sound if somebody is watching the TV. However, this plan becomes ineffective, if nobody is watching the TV or the TV is just turned off. Then, the policy system needs to devise a new plan to notify the residents. Therefore, we treat each conflict resolution problem (even the encountered conflicts) as a different planning problem.

4.1.1 Representing Planning Problems

A planning problem, which is essentially a search problem, consists of: an initial state description, definitions of available actions and their effects, and a goal state. A planner looks for a solution by considering applicable actions until it either finds a chain of actions that make it possible to reach the goal state from the initial state or it concludes that there is no solution. In order to represent planning problems and utilize off-the-shelf planners, we use Planning Domain Definition Language (PDDL) [27] that is an expressive language yet carefully constrained to scale up to large problem sizes [32].

The descriptive model–created using PDDL–is called a *planning domain*, which is necessarily an imperfect approximation that must incorporate trade-offs among several competing criteria: accuracy, computational performance, and understandability to users [79]. There are several versions of PDDL with various features (disjunctions, preferences, temporal actions etc.) that enable modeling complex problems. However, the most of the planners do not support all these constructs due to the aforementioned trade-offs. We now introduce the basic concepts of PDDL and discuss the additional features necessary for the reformulation process later in this chapter.

A state in PDDL in its simplest form is described using conjunctions of positive unary and binary predicates. The use of variables are not allowed in state definitions, as they are either used to represent the information in the knowledge base (initial state) or a desired state (goal state) to be achieved using available objects. For instance, the predicates $Baby(John) \wedge hasFather(John, Bob)$ are valid, however the

Figure 11: Pick-up action from Blocksworld PDDL domain.

```
(:action pickup
 :parameters (?ob)
 :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
 :effect (and(holding ?ob) (not (clear ?ob)) (not (on-table ?ob)) (not(arm-empty))))
```

following *hasResident*$(?f, Bob)$ is invalid. Empty predicates that do not contain any variables or objects are also valid.

Action definitions consist of three parts; parameters, precondition, and effect. Each variable that is used to describe the preconditions and the effects of an action has to be declared as a parameter. The precondition field of an action describes the state that needs to be true with respect to its parameters prior to the execution of the action. Finally, the effect field describes how an action changes its environment when it is performed.

The precondition and the effect of an action can be described using conjunctions of both positive and negative predicates. If a predicate that exists in the preconditions of an action but it is not mentioned in its effects, that predicate remains unchanged. However, if a predicate is negated in the effect field, then that predicate is removed from the world state. Figure 11 illustrates the pick up action from the famous blocks-world domain in PDDL syntax. In this example, a robot arm can pick up an object if the robot is not already holding anything and the object on the table does not have another block on top of it. After performing the action, negated predicates are removed and the $Holding(?ob)$ predicate is added to the world as the robot just picked up the object.

4.2 Policy Reasoning in Planning

In this section, we outline a way of translating conflict resolution problems into a PDDL planning problem. We particularly focus on PDDL 3.1 [94].

4.2.1 OWL-QL Reasoning in PDDL

Our policy framework exploits OWL-QL to cope with very large volumes of instance data. OWL-QL is less expressive compared to PDDL; hence, it is possible to represent its reasoning mechanisms in PDDL. PDDL makes the unique name assumption (UNA), which means that objects with different names are considered to be distinct individuals. QL neither strictly requires this assumption nor violates it. Thus, we assume that used QL ontology respects UNA. However, PDDL uses the closed-world assumption that states any predicate, which is not present in the current state of a planner is assumed to be false. i.e. the planner does not assume that Bob is a father, unless $Father(Bob)$ predicate is explicitly provided. On the other hand, QL adopts the open-world assumption and utilizes query re-writing to infer information.

Given the exploratory nature of this work, as well as the use of multiple planners, there is a lot of related work that must be cataloged and explored. Web-PDDL [95] adopts and extends PDDL with namespaces and sameAsClass to make ontologies more suitable for web applications. From the same author, another software tool called PDDOWL [96] converts OWL-QL queries to Web-PDDL, which are then converted to SQL. Our work does not share the same goal as PDDOWL and Web-PDDL; however [95] explains more in detail what PDDL lacks to fully represent ontologies.

4.2.1.1 Query Rewriting

PDDL's typing feature is suitable to encode simple class hierarchies into the domain file. However, typing alone is not sufficient to express multiple inheritance and subclass expressions with object or data properties, e.g. $TV \sqsubseteq \exists hasSpeaker \sqcap \exists hasDisplay$. For this reason, we represent types of an object with PDDL predicates. All the reasoning formulas concerning types can be then integrated into the PDDL domain by either encoding them in action preconditions with disjunctions or using derived predicates. We use the latter approach as it is more concise.

Figure 12: PDDL representation of the rewritten query TV(?d).

```
(:derived (TV ?d) (and (hasSpeaker ?d ?unbound_1) (hasDisplay ?d ?unbound_2)))
(:derived (Parent ?p) (or (hasChild ?p ?unbound_1) (Mother ?p) (Father ?p)))
```

The first derived predicate in Figure 12 indicates that anything with a display and a speaker can be used as a television even if it is not defined explicitly as a TV. We note that the above approach is insufficient to encode inference queries that include data properties, since numeric values in PDDL are represented using cost functions which are not allowed in derived axiom definitions. This can be addressed using disjunctions, which are also required to fully support type inferences. For instance, to infer that someone is a parent we can use the second rule in the same Figure 12

4.2.1.2 Consistency Check

As explained in the previous chapters, an ontology consists of a TBox and an ABox. Each world state created after applying an action during planning represents an ABox, and an ABox of an ontology is valid as long as it is consistent according to the rules defined in the TBox. Hence, we need to be sure that none of the steps in a generated plan make the ontology inconsistent; otherwise the generated plan is inapplicable.

In other words, as each state during planning represents an actual, real-world state, none of the actions of a valid plan should put the world in an inconsistent state; e.g. a door cannot be both open and closed at the same time. Action preconditions could be designed to handle such inconsistencies; however, here it is important to focus on the fact that this state cannot be achieved in real life. For this reason, we have to check for consistency of the current state every time the planner applies an action. The rules that may cause inconsistency in an ontology are derived from its TBox. Hence, either an external program must check if the generated plans cause inconsistencies, or the planner must handle this. Most planners do not provide a mechanism to run a program after each step; hence we propose the following solution.

Since we can express the consistency query in PDDL using predicates, we create a special action called `check-consistency` and use a special empty predicate called *isConsistent*. *isConsistent* is true in the initial state and it must also be true in the goal state and in all states that lead to the goal state. Furthermore, all the action descriptions are modified to include *isConsistent* in their preconditions along with $\neg$*isConsistent* in their effects. This simply means that we need the *isConsistent* predicate to apply an action, and that the predicate is deleted after an action is applied. Furthermore, the special `check-consistency` action has the negation of the consistency check in its preconditions and *isConsistent* in its effects. As `check-consistency` is the only action that can add the *isConsistent* predicate, it has to be applied after each action. However, the precondition of the `check-consistency` action has to be reinforced with the $forall$ construct since none of the existing objects should violate the consistency rules.

If the world state is inconsistent, the `check-consistency` action will not add the *isConsistent* predicate and all actions will become inapplicable; the goal state will then be unreachable. This will prevent the planner from going even deeper in the current branch of its search space, as that branch will not produce a valid plan. A notify action's PDDL definition implementing the above mentioned approach is presented in Figure 13. However, this solution adds some extra complexity to the planning problem. Considering the performance requirements of IoT applications and to reduce the load on planners we skip the consistency check during planning. Depending on the context we either assume that PDDL actions (i.e. in a controlled environment) do not cause an inconsistency, or we do the consistency analysis via an external program only to validate found plans.

Figure 13: PDDL actions using consistency check.

```
(:action check-consistency
    (:parameters ...)
    (:precondition (and (not (isConsistent))
                        (not (forall ...)...)))
    (:effect (isConsistent)))

(:action notify-with-sound
    :parameters (?device ?action ?person ?flat ?event)
    :precondition (and (canPerform ?device ?action)
                      (SoundAction ?action) (Event ?event)
                      (Flat ?flat) (Person ?person) (isConsistent)
                      (inFlat ?person ?flat) (inFlat ?device ?flat))
    :effect (and (gotNotifiedFor ?person ?event) (notifiedWith ?person ?action)
                (not (isConsistent))))

(:action notify-with-visual
    :parameters (?device ?action ?person ?room ?event)
    :precondition (and (canPerform ?device ?action) (VisualAction ?action)
                      (Event ?event) (Person ?person) (isConsistent)
                      (inRoom ?person ?room) (inRoom ?device ?room))
    :effect (and (gotNotifiedFor ?person ?event) (notifiedWith ?person ?action)
                (not (isConsistent))))
```

4.2.2 Policies in PDDL

The central contribution of this chapter is automating the policy conflict resolution process using automated planning techniques. The first step towards this goal is the modeling of the conflicting situation and its attendant information into a planning problem instance. Specifically, obligations and prohibitions relating to a specific entity need to be handled, since they are the primary reason that a conflict might arise.

The key concept here is the framing of obligations and prohibitions as *soft constraints* on a given policy that must be handled by the underlying planner. In this notion, obligations and prohibitions can be seen first and foremost as goals that an entity must achieve. These goals may be soft in nature, i.e., there may be degrees of fulfillment rather than just binary *true* or *false*. Additionally, since we are considering conflicts in the policy space, obligations and prohibitions may be competing with each other. In such instances, it may be the case that not every conflict can be fully resolved; instead, a plan needs to be formulated that least violates some goodness metric defined for the domain.

We illustrate this idea with the use of our running example, outlined in Chapter 3 previously. Envision a scenario where if the doorbell is pressed, an obligation to notify someone within the house (that there is someone at the door) is immediately created due to an existing policy. However, there is also a current prohibition on making sound, since someone is sleeping – this is due to a second policy that exists on the doorbell. This forces the addressee (in this case, the doorbell) to make a determination and pick between one of the two policies to violate.

However, if there were another way to fulfill both of these policies (one of them with an obligation, the other with a prohibition) at the same time without violating the other, the conflict and ensuing violation could be avoided. Given that we are dealing with complex domains with multiple entities and services, it is entirely possible that notification is possible in a number of ways. We investigated two different methods to model policies in PDDL; using preferences, using cost functions.

4.2.2.1 Policies as Preferences

Policies are soft-constraints (preferences) by definition, thus it is more intuitive to use a preference based planner for this work. However, policies are more expressive and dynamic in nature than PDDL preferences. Now, we discuss the advantages and limitations of preference based planning.

In our smart home domain, the preferences (one on the obligation, and one on the prohibition) would be represented as follows:

a) `preference pref-0 (gotNotifiedFor Bob dbell)`

b) `preference pref-1 (notifiedWith Bob visualaction)`

In the above, `pref-0` stands for the obligation that when the doorbell rings, Bob (a person in the house) must be notified. `pref-1` is the prohibition that whenever the doorbell rings, the notification must happen visually[1]. These two preferences are

[1]An alternate way of encoding this preference would be to define a `(DoNotUseSound ?entity)` predicate; this is a domain modeling question

in conflict, and will be resolved by the planner based on the violation cost that is prescribed for each.

One question that crops up is whether this can just be achieved with regular PDDL actions, without the use of preferences. For example, consider the actions in Figure 13. The two notification actions both give the same main effect, namely `(gotNotifiedFor ?person ?event)`. Therefore, it bears asking why the conflict resolution problem cannot just be handled in a straightforward manner by the planner without the need to invoke preferences; clearly, there are two choices, and the planner can take the visual notify action if the sound notify will violate some other constraint.

However, this line of thought precludes the possibility that sometimes there *may be no other way* to uphold a specific obligation, or avoid a certain prohibition. In certain cases, the problem may be overconstrained to the point where some constraint *has* to be violated. In such cases, it is useful to think of these constraints as no longer hard goals but instead soft constraints that carry violation costs – preferences. The planner now has more room in a complex problem setting to decide which constraints can be violated, and to arrive at the best possible solution.

Unfortunately, preferences cannot capture the whole notion of prohibition policies, as they do not allow placing constraints on domain actions. This solution requires domain engineering and enforces policy authors to define a fallback strategy to model a prohibition. Furthermore, multiple restrictions might apply to a certain action, in which case the cost of the action should reflect all the violation costs associated with that particular action. Even though preferences alone cannot represent policies, they can be used as a complementary mechanism in the conflict resolution process.

4.2.2.2 Policies as Cost Functions

PDDL models support a global function that keeps track of the accumulated cost associated with executing all the actions in the domain. Planners can be specified to minimize the total cost, while trying to achieve the goal. In addition to the global one, we introduce a new cost function for each different active prohibition policy. These prohibition cost functions are associated with the effects of the actions that they regulate to increase the global cost if the prohibited action is used.

Using cost functions makes it straightforward to model obligations in the problem file. We simply set the goal state required by the obligation policy. Unlike obligations, there are more details that need to be considered, while modeling prohibitions. These are, *(a)* multiple prohibition policies might prohibit an action *(b)* multiple prohibition policies might apply to a device *(c)* a device might be restricted to use an action, while another is not *(d)* each prohibition policy has its own cost Keeping all these points in mind, we create a specific predicate to represent the violation cost of a prohibition policy and couple it with the addressee individual in the problem file. If a prohibition policy does not apply to an individual, its violation cost could be considered and represented as 0.

Whenever the effects of an action take place, the action cost and any cost function associated with it are added to the total cost. To encode the sound prohibition (sleeping-baby) a new cost function `(sleeping-baby ?device)` is added into the effects of `notify-with-sound` action. A complete example is provided in the next section in Figure 14.

By adding that violation cost to the total cost, we add a penalty, if a prohibited action/device pair is used in the plan. Furthermore, this formulation allows us to map each different policy to each different device and action they apply to. If we had more active prohibition policies, we could create more cost predicates like `p2Cost, p3Cost,` .... Encoding prohibition policies as cost functions in PDDL is more intuitive than using preferences. Furthermore, the planner utilizing this method can detect implicit conflicts, which occur when an obligation policy's target action is a composition of actions that includes a prohibited action. The planner can detect such conflicts as it computes the costs of each individual action in a plan.

Modeling policies this way, however, only works if each action description contains its actor as a parameter, since policies are refined to actor-action pairs. Otherwise, it is not possible to associate a penalty with an action. For example, performing the `notify-with-sound` action may be prohibited for electronic devices, but the same policy may not apply to fire alarms. Thus, the device that performs the action has to be explicitly defined.

Although using above discussed methods make the formulation of conflict resolution problem more natural, they do not completely solve it, since policy activation and expiration conditions are queries. Hence, during planning, we need to query the current state with the activation and expiration conditions of policies to check if a new policy is activated or an active policy is expired. Keeping performance issues in mind this can be addressed by finding a limited number of plans with the assumption of an immutable normative state. An external program can then simulate the found plans while updating the normative state and compute the correct plan costs.

4.3 Reformulation of Policy Conflicts to PDDL

A PDDL definition consists of two parts: The domain and the problem definition. Below we describe how the policy conflicts were automatically translated into these PDDL definitions while preserving their semantics.

4.3.1 PDDL Domain

A domain definition primarily provides the predicates, functions, and action definitions available to generate a planning problem. The predicates are used as a vocabulary to describe the action definitions and the problem instances. Additionally, any PDDL features required by the planning problem should be declared in the domain definition. i.e. `ADL` for disjunctions, `derived-predicates`, `action-costs` etc.

The translation process starts with encoding all concepts and object properties from the ontology as predicates and their inference rules as derived predicates in the PDDL domain. Concepts are defined as unary predicates (i.e. $Flat(?f)$), while object properties are defined as binary predicates (i.e. $inFlat(?o, ?f)$). Thus, both the service descriptions of devices and the policy definitions must share the same vocabulary. For instance, if devices expose their services using a different vocabulary, then it must be possible to translate it to the ontology used by the policy system.

The next step is to declare all the numerical data properties and active prohibition policies as functions. e.g.`battery-level`. Comparing and modifying **`numeric-valued fluents`** require planners to support the *fluents* feature of PDDL. Unfortunately, some planners operate with integers, hence encoding floating numbers as integers might be necessary.

The final step of the translation process is to define actions and associate them with the active prohibition functions. However, discovering and translating service descriptions into PDDL is beyond the scope of this thesis. An example domain definition generated by extending the TBox 3 is shown in Figure 14

4.3.1.1 Action Descriptions

Given the importance of the constituent actions in our domain description, we briefly describe the genesis of this knowledge. In the context of our application, an action can be an API offered by a device or a web service. For example, an action could be moving a robot, downloading information from the internet, or turning a TV on. For an illustration, see the simplified version of the **`notify-with-sound`** action shown in the previous section.

In order to keep things simple, we assume that service descriptions are available to us in quasi-PDDL form using our ontology and that we do not need to do complicated conversions from a description language. However, it is possible to expose

Figure 14: Smart Home PDDL domain definition.

```
(define (domain iot)
  (:requirements :adl :derived-predicates :action-costs)
  (:predicates
    (Person ?p) (Adult ?p) (Baby ?p)
    (SoundAction ?a) (SoundNotification ?a) (VisualAction ?a)
    (Flat ?f) (inFlat ?o ?f) (hasResident ?f ?r) (inRoom ?r ?o)
    (Event ?e) (DoorbellEvent ?e) (notifiedWith ?p ?a)
    (producedBy ?e ?d) (gotNotifiedFor ?p ?e) (canPerform ?d ?a))

  (:functions (total-cost) (sleeping-baby ?d))

  (:action notify-with-sound
    :parameters (?device ?action ?person ?flat ?event)
    :precondition (and (canPerform ?device ?action)
                       (SoundAction ?action) (Event ?event)
                       (Flat ?flat) (Person ?person)
                       (inFlat ?person ?flat) (inFlat ?device ?flat))
    :effect (and (gotNotifiedFor ?person ?event) (notifiedWith ?person ?action)
            (increase (total-cost) (sleeping-baby ?device))))

  (:action notify-with-visual
    :parameters (?device ?action ?person ?room ?event)
    :precondition (and (canPerform ?device ?action) (VisualAction ?action)
                       (Event ?event) (Person ?person)
                       (inRoom ?person ?room) (inRoom ?device ?room))
    :effect (and (gotNotifiedFor ?person ?event) (notifiedWith ?person ?action)))

 (:derived
    (Event ?e)
    (DoorbellEvent ?e))

 (:derived
    (SoundAction ?e)
    (SoundNotification ?e))

 (:derived
    (Person ?p)
    (or (Adult ?p) (Baby ?p))))

(define
  (problem iot)
  (:domain iot)
  (:objects dbell flt room speaker television
                        Bob Jane John e1 playAudio displayMessage)
  (:init
    (Baby Jane) (Baby John) (Asleep John) (Awake Jane)
    (Doorbell dbell) (DoorbellEvent e1) (producedBy e1 dbell)
    (Flat flt) (inFlat John flt) (inFlat Jane flt)
    (Adult Bob) (hasResident flt Bob) (inFlat Bob flt)
    (SoundNotification playAudio) (VisualAction displayMessage)
    (canPerform speaker playAudio) (canPerform television playAudio)
    (canPerform television displayMessage) (inRoom Bob room)
    (inFlat speaker flt) (inFlat television flt) (inRoom television room)
    (= (sleeping-baby speaker) 10) (= (sleeping-baby television) 10)
    (= (sleeping-baby dbell) 10)
    (= (total-cost) 0))
  (:goal (gotNotifiedFor Bob e1))
  (:metric minimize (total-cost)))
```

device capabilities as services with the appropriate infrastructure. A numeric value is associated with each action to represent how much they are preferred by the user or how much resource their execution requires in terms of computation power or other resources. These numerical values could be difficult to come up with for policy authors, yet various methods could be used to compute them.

We tackled the service definition issue in our implementation by using concepts and properties from the policy ontology. For example, a sound action could be defined using *SoundAction* class and *actor* and *hasParameter* and *hasEffect* properties. Devices expose their capabilities through the HyperCat [82] server. This is a trivial solution and is probably not sufficient for sophisticated real applications.

4.3.2 PDDL Problem

The instances in ABox, which contains knowledge about individual objects, are mapped to the initial state and the goal of the planning instance. For example, the atom `(canPerform speaker playAudio)` indicates that the doorbell can produce a sound to notify when it has been pressed. In addition, cost functions are also initialized for each capable device in the initial state. Whenever there is a change in the external world, we assume that this is reflected in the initial state. For example, when the baby wakes up, the value of function `(= (sleeping-baby speaker) 10)` is updated to 0 in the initial state.

The goal of the planning problem is to fulfill the obligation policy, while minimizing the total cost. However, the final plan cost should not exceed the violation cost of the obligation. In that case, the framework would choose to violate the obligation policy instead of executing the found plan. The plan metric requires the minimization of the total cost: `(:metric minimize (cost))`. The PDDL problem definition of our running example is illustrated in Figure 14. We generated the initial state by extending the ABox in Table 4.

All individuals in the KB are defined as objects in the PDDL problem. Similarly, class and property axioms of these individuals are selected from the KB and written into the initial state using PDDL predicates. We note that each entry in the KB is either a concept or a property assertion. This approach is trivial to implement and probably would work for a smart home example, however it could prove to be an exhaustive task for large KBs. Then, methods can be developed to use only a part of the KB for planning.

The total cost is initially set to zero and prohibition cost functions for each policy-addressee pair are set to the corresponding violation costs. Cost functions for unaffected objects are initialized to zero. Finally, the goal state is produced by using the expiration conditions of the active obligation's instances with disjunctions. Let's assume there is another resident, *Alice*, at home. Now, it could be sufficient to notify either *Bob* or *Alice*. The goal state is defined as `(:goal (or (gotNotifiedFor bob someoneAtFrontDoor) (gotNotifiedFor alice someoneAtFrontDoor)))`

All active prohibition policies along with their bindings are encoded in the planning problem to prevent unintentional violation of other active policies while avoiding the actual conflict. In our implementation, we used a central server to processes the policies of all the connected devices for convenience. We note that for each prohibition instance, a cost function predicate is created using its name, and added to the effects of actions that the policy prohibits. For example, if `sleeping-baby` is the name of the policy, then (increase (total-cost) (sleeping-baby ?x)) statement is added into the effects of `notify-with-sound` action. Finally, each function is initialized in the problem definition.

Encoding an obligation policy becomes relatively simple when the desired goal state is already defined in the expiry conditions—i.e., the variables in the condition get bounded when the activation query is executed over the knowledge base. For example, in the case of Bob and Alice, activation condition would return two rows with different bindings; $\{?d = dbell, ?p = Bob, ?f = flt, ?e = e1\}$ and $\{?d = dbell, ?p = Alice, ?f = flt, ?e = e1\}$.

4.4 Evaluation

At the outset, we clarify that the current evaluation is presented in the spirit of a proof-of-concept rather than as a full-scale evaluation of our design choice to implement policy conflict resolution as a preference-based planning problem. The approach explained in the previous section requires a PDDL planner that supports action costs, derived predicates, and preferences at the same time. Other PDDL features can easily be added to the list to make the planning domain more realistic. Unfortunately, we could not obtain a planner that implements all of those requirements, and were thus unable to evaluate our approach using a single planner.

4.4.1 Implementation

In order to complete the implementation of our system introduced in Chapter 3, we implemented the following: a cost-based planning strategy based on LAMA [97], which is a planner that builds on Fast Downward [98], supports ADL descriptions, actions costs, and derived predicates. We thus used it to check if we could accommodate OWL-QL reasoning into PDDL. However, Lama does not support numeric-valued fluents, which are required to deal with data properties.

We then tested our planner implementation against our running scenario of smart doorbell and home automation by simulating sensors with Java and Android applications. The policy system introduced uses the planner to find all solutions, and then analyzes the found plans with external scripts to update the plan costs. The plan, as expected, which displays a message is selected as the final choice due to its lower cost.

4.4.2 Experimental Setup

We compared our approach with the most generalized conflict resolution strategies: always prohibition, always obligation, and higher violation cost. There are other conflict resolution strategies that use meta-policies, class hierarchies etc., however these approaches are highly customizable and their success depends on the application context (i.e. how domain ontology is created). Thus, we do not consider those approaches in this experiment.

In order to make a comparison, using the ontology developed for our smart home scenario we generated random problem files (settings) that have at least one policy conflict. Then, we tried solving each of these conflicts with our planner and compared the results of the plans with the rewards obtained by the other methods. This experiment also allows us to show that planning is not only used to do a composition of actions, but also used to select the right parameters for a single action.

We generated 75 problem files in total; 15 problem files for 5 different number of devices (2, 3, 5, 10, 20). Our intuition here was that each newly added device favors the planning method even if they did not add a new capability. We randomly generated problems as follows:

- Each problem has only 1 goal: Bob has to be notified.
- Add X number of devices to each problem instance.
- Add capabilities (sound, visual, sms) with probability 1/3 to a newly created device. e.g. a device has all three capabilities with a probability of 1/9.
- For each device and for each capability of that device, add a prohibition with the probability 1/5 along a random violation cost from 1 to 10.
- Max violation cost is set to 10.

Table 7: Obtained results for problems with 2, 3, 5, 10, 20 devices.

				Number of Devices: 2					
Method	Total	S_{Cnt}	F_{Cnt}	S_{Avg}	S_{Max}	S_{Min}	F_{Avg}	F_{Max}	F_{Min}
Always Prohi.	15	0	15	0	0	0	4	8	1
Higher Cost	15	4	11	3	6	1	4	8	1
Planner	15	15	0	0	3	0	0	0	0
Always Obli.	15	15	0	6	10	1	0	0	0
				Number of Devices: 3					
Always Prohi.	15	0	15	0	0	0	6	10	1
Higher Cost	15	7	8	4	8	3	4	7	1
Planner	15	15	0	0	3	0	0	0	0
Always Obli.	15	15	0	5	8	2	0	0	0
				Number of Devices: 5					
Always Prohi.	15	0	15	0	0	0	4	10	1
Higher Cost	15	4	11	4	7	1	3	7	1
Planner	15	15	0	0	0	0	0	0	0
Always Obli.	15	15	0	5	10	1	0	0	0
				Number of Devices: 10					
Always Prohi.	15	0	15	0	0	0	6	10	2
Higher Cost	15	6	9	4	7	1	4	8	2
Planner	15	15	0	0	0	0	0	0	0
Always Obli.	15	15	0	5	10	1	0	0	0
				Number of Devices: 20					
Always Prohi.	15	0	15	0	0	0	7	10	3
Higher Cost	15	9	6	4	8	2	6	10	3
Planner	15	15	0	0	0	0	0	0	0
Always Obli.	15	15	0	6	10	2	0	0	0

- Randomly choose the device, which is obliged to notify the person, until a conflict happens.

- Randomly choose the violation cost of the obligation.

The result is X number of devices with random capabilities and random prohibitions. Through the first part of the evaluation, one problem file corresponds to one specific conflict of a device, since devices cannot collaborate using other strategies. e.g. our running example becomes a question of should doorbell make sound or not. If prohibitions always method is applied, it should not make sound. If obligation always method is used, it should make sound.

4.4.3 Results

Now, we show and discuss the outcomes of our experiments. The abbreviations in the result tables are as follows: Total = total number of problems, S_{CNT} = number of

Table 8: Total number of conflicts in the settings.

Device No	Problem Size	Total Prohibitions	Max # Prohibitions	Min # Prohibitions	Avg
2	15	22	3	1	1
3	15	19	3	1	1
5	15	26	4	1	1
10	15	45	6	1	3
20	15	93	10	2	6

times the obligation is fulfilled, S_{AVG} = average violation cost to fulfill the obligation, S_{MAX} = max violation cost, and F stands for Failed.

Although in most cases planning seems like it never violates a policy, this is not what we claim and these results are obtained due to the simplicity of our domain. However, results in Table 7 with 2 devices show that planning approach choose to violate a policy to fulfill the obligations at least once. So, as we add more capabilities to devices or more devices with different capabilities into the system, we add more solutions to our planning problem. The IoT system gets more powerful by allowing devices to collaborate. However, adding more devices does not affect the results of other strategies, as they are resolving conflicts from a single device's perspective.

Since planning approach resolves conflicts at a global scale, we also computed the number of potential conflicts (prohibition policies) that exist in the system to show how much of these could be avoided by the planner. Table 8 illustrates how many prohibitions existed in the generated problem files. The fields "Max Prohibition" and "Min Prohibitions" show the maximum and the minimum number of prohibitions that a setting contains. For example, average violation cost for the planning approach for planning problems with sizes of 3,5, 10, and 20 is 0. This means that none of the policies are violated and could be interpreted as planning avoided all possible conflicts (19, 26, 45, 93). However, the planner still had to violate a policy in some of the problem files, which consist of only 2 devices.

4.5 Discussion

As part of our IoT policy framework, the biggest advantage of planning compared to other systems (i.e. *Smart Environments Configurator* [99]) is that planning can avoid policy violations in scenarios where there exists a path to do so. The use of planning allows conflicts to be resolved at a global scale by enabling all available and connected devices to collaborate and avoid violations. Additionally, it also confers a scalability advantage—as problem instances get larger, one of the following three situations arise, once again highlighting the advantages of a planning-based approach: *(a)* handcrafted policies can be customized to specific problem devices, capabilities, and problem instances in order to minimize violations, but such policies cannot scale as the instance size increases; and *(b)* planning approaches—which can be general enough to apply to diverse problem instances and efficient enough to scale to increasing instance size—always look to optimize the metric. *(c)* larger problem instances may actually work in favor of planning approaches, since larger instances may contain more devices and resources, which allow for more solution paths through the problem space.

There is still room for improvement in our implementation. As the number of objects in IoT applications can get very high, similar policy conflicts will be raised. Our current implementation treats each conflict (even if it is encountered before) as a new planning problem. We can improve the implementation by developing a re-use mechanism such as creating and storing plan templates or caching previous plans in a database. Furthermore, the framework can preprocess a conflict to only use a part of the ABox to improve the planning performance by decreasing the problem size.

The main idea behind adopting PDDL in our work is to utilize off-the-shelf planners. However, we were not able to find a single planner that can handle all the features we required. Unfortunately, to create a more realistic planning domain we can complicate the planning problem even more by introducing concepts such as interleaved planning that requires the execution of non-deterministic actions such as *locate* *and search* actions, probabilistic planning, and temporal planning [100] to handle durative actions like ventilating. As with many other real-world applications which do service compositions, these concepts are essential for the IoT Domain.

Violation costs of policies and action costs play a significant role in resolving conflicts. These numeric values are difficult to come up with and highly depend on the context. i.e. if an IoT application is trying to minimize the energy consumption, then the violation and action costs could represent the consumed power. However, in most cases setting the violation costs is not so trivial. For instance, policy costs in the context of health and safety applications are not only more sensitive, but also they do not represent tangible resources.

CHAPTER V

REAL LIFE APPLICATION

In this chapter, we describe how to combine ontology-based policy reasoning mechanisms with in-use IoT applications to customize and automate device behaviors. First, we present our case study. We then discuss how the policy framework can be extended with data federation to handle diverse and distributed data sources. We demonstrate that smart devices and sensors can be orchestrated through policies in hazardous workplaces, such as coal mines. Lastly, we evaluate our approach using real applications with real data and demonstrate that our approach is scalable under high load of data and devices.

5.1 Case Study: Health and Safety in Mines

Monitoring and following regulations are especially important in health and safety applications in hazardous working environments. The great majority of occupational accidents are preventable through adherence with existing International Occupational Health and Safety standards[1], which are not always implemented by workplaces. Incorporating these regulations in IoT solutions can be an effective way to improve worker's health and safety in dangerous environments, as mental and physical fatigue under tough working conditions may harm workers' decision making process. In this section, we show how our policy framework can be used to enhance health and safety of mine workers.

We particularly focus on mines due to their complexity. IoT solutions are already adopted within this domain to monitor the environment and to take safety measures.

[1]`https://www.iso.org/iso-45001-occupational-health-and-safety.html`

However, the code of these applications has to be changed to modify or customize policies, since they are hard-coded and generic. Furthermore, detecting conflicts within this domain is particularly important, as undetected conflicts between policies in these applications can cost people's lives. Deployed sensors and readers enable these solutions to maintain a near-to complete view of the environment that makes the planning approach more suitable for resolving conflicts, while responding to events in real time becomes even more challenging.

We believe that the safety of the workers in an underground mine can be greatly enhanced by assessing their health status, measuring gas levels in the mine, tracking assets and preventing vehicle collisions. In this section, we present our use case domain, discuss the significance of regulations, and explain how our framework could be applied to in-use solution.

5.1.1 Smart Mine Solutions

IoT solutions[2,3] for underground mines, in general, have two main components; one for tracking RFID tags (assets, vehicles, personnel), and another one for measuring gas levels. In addition to tags and gas sensors, there are other sensors like panic buttons, vibration sensors, and motion sensors to detect fall accidents. In this work, we only focus on the location sensors, panic button, and gas sensors for the sake of simplicity.

The gas levels are measured approximately every 5 seconds and tags are sensed arbitrarily as assets or personnel come close to indoor readers. These systems are able to perform geofencing, locate assets in the dark, detect dangerous gas levels and so forth. However, they do not offer adaptive solutions, and they are not capable of making intelligent decisions in case of multiple conflicts. Although IoT devices carry a huge potential for improving mining operations, existing *smart* mine solutions are

[2] `http://litumiot.com/mine-rtls-worker-tracking/`
[3] `www.cat.com/en_US/by-industry/mining/underground-mining`

Table 9: Examples from mine regulations of Turkey.

1	Employees are not allowed to use tools without the proper license.
2	Assets should not function outside their assigned regions.
3	Empoyees should not enter unauthorized regions.
4	All personnel must go to a safe zone before blasting.
5	Personnel are obliged to help, if a person is in need.
6	Personnel must exit a zone if oxygen level is $\leq$ 19.0%
7	Firemen are obliged to respond to fire dangers in the mine.
8	Employer must take her personnel to safety.

limited since they mostly operate on raw data with dangerous events and outcomes being hard-coded along with *predefined one-size* fits all remediations and the collected data is monitored by human in control rooms without any intelligent assistance.

5.1.2 Domain Policies

Coal mines host various dangers and people die or suffer for various different reasons such as operating a vehicle for which they are not licensed, unnoticed fires in coals, staying in an explosion area, inhaling toxic gas etc. Since underground mines are extremely dangerous environments, worker safety is highly regulated. We use policies taken from the health and safety regulations of underground mines in Turkey as our motivating examples. These policies are depicted in Table 9.

The language used in these regulations is too vague for people without expert knowledge to understand. In addition, multiple policies might be necessary to capture a rule's true meaning. For example, the 4th policy obliges each personnel to leave a blasting zone. This statement can be represented with a single obligation policy. However, the obligation alone is not sufficient to fulfill the rule's safety goal, unless it is accompanied by a prohibition policy that prohibiting all personnel from entering into the blasting zone. Furthermore, if critical circumstances arise that makes it impossible to satisfy all the constraints and it might become necessary to violate some regulations; for instance, a personnel can take initiative and cut the electricity in a dangerous location, if there is not enough time for an engineer to arrive.

5.2 Policy Reasoning Over Distributed Data

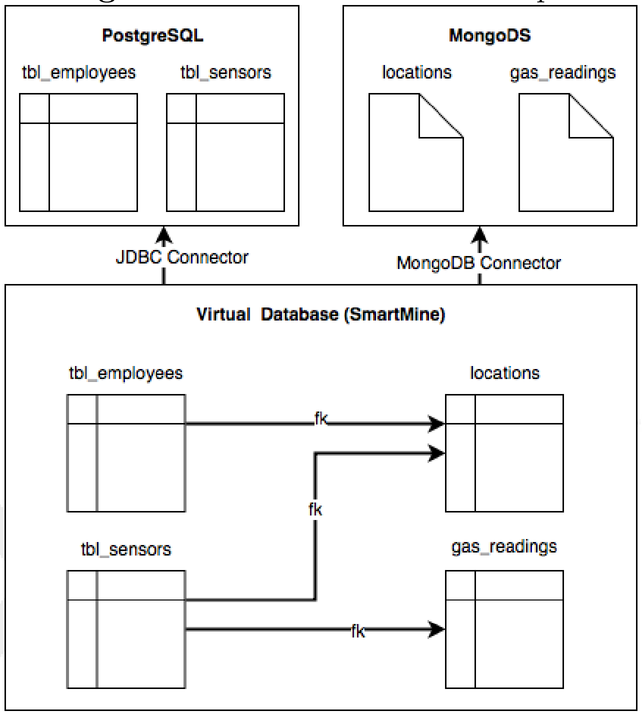
Typical in-use IoT solutions come with multiple heterogeneous databases (and/or software components) that rely on varying schemata, and they implement their own communication protocols. Thus, augmenting such systems with semantics require re-designing of databases to conform certain rules, which would not be feasible for many applications.

We propose using a data federation solution to provide a unified access for target application's data sources. Then, our policy system can be integrated into the application by adopting ontology-based data access (OBDA). OBDA allows to define an ontology over a database (i.e. virtual database), thus it eliminates the QL knowledge base restrictions for implementing the policy system. Let us note that data federation is not necessary to adopt OBDA if the target application has a single data source and OBDA is not necessary for tasks that do not depend on the ABox. i.e. TBox is sufficient to create a sandbox for detecting policy conflicts.

5.2.1 Data Federation

It is a common practice among IoT applications to store sensor readings in non-relational databases or process them as streams, while storing less volatile data in relational databases. *Data federation* (also known as *data virtualization*) is a method to query data that resides on disparate data sources through one virtual (federated) database [101, 102]. The virtual database does not maintain copies of data, however it provides a way of querying separate data sources, as if there is one single database system. The physical data sources can be relational or non-relational databases, streams, or even documents. These data sources can be geographically decentralized and operate autonomously and independently of each other. Thus, federation does not only provide a unified interface, but it also makes it possible to scale data sources individually.

Figure 15: Data federation example



For example, let us assume a mining company stores employee and less volatile sensor information (i.e. placed locations, types etc.) in a relational database (i.e. PostgreSQL[4]) and writes frequently updated data such as locations of vehicles and gas readings into a document (non-relational) database (i.e. MongoDB [5]). Then, a state-of-the-art federation tool like JBoss Data Virtualization [6] can be used to unify these two data sources as illustrated in Figure 15.

The created virtual database does not have to represent the whole data in these data sources. Sensitive data (i.e. personal information of employees) and data that is not required by the policy ontology can be left out during the mapping process. Representing only the relevant data creates a less complex database view and it could also eliminate some privacy concerns. The schema of the virtual database might have certain limitations depending on the adopted federation solution, as the data

[4]`https://www.postgresql.org`
[5]`https://www.postgresql.org`
[6]`https://www.redhat.com/en/technologies/jboss-middleware/data-virtualization`

Table 10: OBDA mapping of the Blaster class

mappingId	Blasters
target	:employee{employee_id} a **:Blaster** ; **:authorizationLevel** {auth_level} ; **:hasTag** :tag{tag_id} .
source	**SELECT** "employee_id", "tag_id", "auth_level" **FROM** tbl_employees **WHERE** role = 5

is physically scattered across heterogeneous data sources. Accessing data requires the translation of user requests into database queries and the accumulation of their results.

The virtualization layer provides a unified layout to access data through virtual tables. Then, we can create mappings between the ontology and these virtual tables to turn the virtual database into an OWL-QL knowledge base. Consequently, this approach enables policy framework to do semantic reasoning over the data that is split across different data sources.

5.2.2 Ontology Based Data Access

Ontology-based data access (OBDA) is a paradigm that allows users to query existing data sources by using concepts and properties from an ontology. It makes it possible to define an ontology over a relational database by mapping [103] the concepts and properties of an ontology to the columns of tables in the target database using SQL queries. Table 10 illustrates the mapping of the *Blaster* class and some associated properties in a mine ontology, which we developed for our use case, over an intelligent mine solution's database. The mapping defines what a *Blaster* individual corresponds to in the target database, and how her/his tag and authorization level can be found.

Using axioms in the ontology and the mapping definitions, the OBDA system rewrites queries (e.g. activation conditions of policies) into the vocabulary of the data sources and then delegates the actual query evaluation to a suitable query answering

Table 11: OBDA mapping of the Low Oxygen Danger class

mappingId	Low Oxygen Danger
target	:danger{region_id} a **:LowOxygen** . :region{region_id} **:hasDanger** :danger{region_id} .

source	**SELECT** "region_id" **FROM** tbl_gas_readings
	WHERE oxygen ¡= 19

system such as a relational database [104]. On the negative side, this evaluation adds an overhead to QL query answering process [105]. The re-written query is a SQL query with the `UNION` operator that includes inference rules and it is typically optimized for execution. For instance, *Blaster* is a sub-class of *Employee*, thus its individuals constitute a subset of all employees. However, including *Blaster* mapping in a query that retrieves all employees would be redundant, since all employees are already stored in the same table.

Unfortunately, OWL-QL lacks expressivity to represent the most of the real life scenarios. For example, it is not possible to represent a statement like "oxygen $\leq$ 19%" that is necessary to describe the 6th policy in Table 9. However, if we make the same assumption that we make in Chapter 4 and assume that UNA is respected in the underlying QL ontology, unqualified number restrictions become harmless to the reasoning process [104]. Placing this constraint on the ontology makes even more sense in the context of OBDA as all databases and PDDL work with the UNA. Then, native operators in SQL can be utilized to reason over the numerical constraints found in QL knowledge bases, which is not possible w.r.t. the QL language. Utilizing OBDA we can create a *LowOxygen* concept that can be used to describe the low oxygen policy. The mapping of the concept and the policy description are shown in Table 11 and Table 12, respectively.

The adopted OBDA framework's reformulation algorithm and query execution performance directly affects how fast the policy system can reason about policies.

Table 12: Prohibition policy example: Low Oxygen Danger.

$\chi : \rho$	$?e : Employee(?e)$
N	O
α	$inRegion(?e, ?region) \wedge hasDanger(?region, ?danger) \wedge LowOxygen(?danger)$
$a : \varphi$	$?a : EvacuateRegion(?a) \wedge actor(?a, ?e) \wedge target(?a, ?region)$
e	$inRegion(?e, ?safe) \wedge SafeZone(?safe)$
d	`10:Minutes`
c	20.0

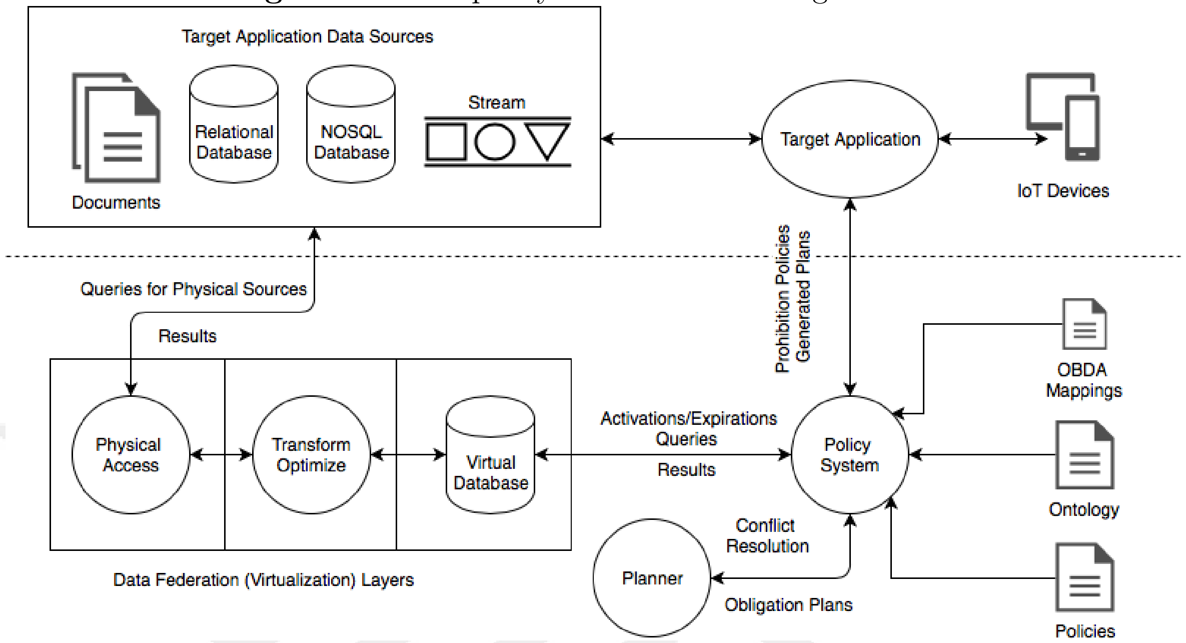
Moreover, the complexity of the mapped ontology is critical for achieving good performance with the OBDA system and re-written queries. The common practices include avoiding imports and deep hierarchies in the ontology [106]. Furthermore, queries used in the mappings can significantly degrade the performance if they include unnecessary joins or unions.

5.2.3 Revised Policy Framework

We recognize the costs and difficulties in modifying existing systems, thus, any semantic solution to the problem should have the least overhead to encourage the adoption. To overcome the shortcoming of the proposed framework, we reconsider its architecture without affecting existing functionality, components—and their interactions. Figure 16 depicts an overview of the framework.

The solution not only minimizes (or avoids) changes in target data sources, but also enables applications to scale each data source independently. Moreover, we propose exposing only the necessary information to the policy framework to increase the performance and to prevent privacy related issues. For instance, the policy framework does not need to know about the model of a drill machine to decide whether the quality of air requirement is met for the people working in the mine. The policy system in this new architecture can be considered as an external component or a software library that provides policy reasoning and conflict resolution mechanisms to in-use IoT applications.

Figure 16: The policy framework utilizing OBDA



5.3 Adding Temporal Constraints on Policies

It is easier to make compromises from the expressivity of a policy language to increase the efficiency of the reasoning process, when working with more tolerant domains such as managing smart home devices. However, certain factors like timeliness of actions become vital when reasoning with health and safety policies. For example, if the residents cannot be notified of a doorbell event, in the worst case scenario the guest will leave. On the other hand, if a worker does not evacuate a dangerous zone in a short amount of time, then his/her life will be at stake. Furthermore, some obligations might be more urgent than the others. Thus, in order to help IoT applications with prioritizing their obligations, we extend the policy definition described in Chapter 3 with a deadline field. i.e. Table 12 depicts the description of the 6th policy with an added time constraint.

When modeling prohibition policies, in our experience, the deadline field does not seem to be as crucial as it is for prioritizing obligations. It might be necessary to place time constraints on prohibitions to restrict actions for a certain amount of

Figure 17: A sample temporal drive action in PDDL

```
(:durative-action drive
    :parameters (?v - vehicle ?l1 ?l2 - location)
    :duration (= ?duration (road-length ?l1 ?l2))
    :condition (and (at start (at ?v ?l1)) (at start (road ?l1 ?l2)))
    :effect (and (at start (not (at ?v ?l1))) (at end (at ?v ?l2))))
```

time. However, we were able to avoid this need by utilizing the expiration conditions of prohibitions. Let us consider the following policy; *all machines should stop working for T minutes, if the carbon monoxide level is above a pre-defined threshold.* If we assume that the IoT solution will ventilate the region in `T` minutes, then we can represent the same policy as *all machines should stop working, unless the carbon monoxide level is below a pre-defined threshold.*

Introducing deadlines in the policy descriptions do not affect how we detect conflicts. We still consider two policies to be in conflict, even if the prohibition policy has a shorter deadline than the obligation. The conflict detection algorithm can be extended to handle this situation. However, the deadline field, enforces the conflict resolution strategy to adopt temporal planning methods, as now goals have to be achieved in an absolute deadline. Subsequently, action definitions in the planning domain has to provide at least an estimation of the execution of time with respect to its parameters. An example definition of a drive action is presented in Figure 17.

We can model safety regulations as a temporal planning problem instead of hard-coded rules and, by exploiting the technology, offer adaptive solutions. For instance, having access to real-time location and sensor information it is possible to create a comprehensive model of a mine. Planning can then be used in two different ways: *(1)* in under normal conditions, to organize the activities of the workers in the mine with the goal of maximizing worker safety; and *(2)* in abnormal conditions, to deal with emergencies or anomalies. However, in both cases, the main goal of planner is to minimize the violation cost of policies. In some cases, a planner might decide not to fulfill an obligation if it violates too many prohibitions.

5.4 Evaluation

In Section 5.2, we described how our framework can be extended to perform policy reasoning over federated data sources. However, the performance of the policy reasoning over federated data sources depends on many additional factors, such as, the structure of the target data, the number of different data sources and their geographical locations, the OBDA software, and the data federation solution.

That is why, it is not possible to evaluate the performance of our framework with a federated database in isolation from all these factors. Nevertheless, we employed the state-of-the-art solutions for data federation and OBDA to avoid any bottlenecks as much as possible. For instance, we used Ontop [106] and JBoss Data Virtualization software, which are among the most efficient implementations available and are being actively developed. Their benchmarks[7] [8] can provide some insight into how efficient policy queries can be executed over federated data sources.

In this section, we describe how we evaluated the performance of our policy framework by integrating it with a real mine solution's database. Let us note that we do not use a data federation software in this evaluation due to aforementioned issues.

5.4.1 Experimental Setup

We ran our experiments on a server with 32 GB RAM and two 8-core Intel Xeon CPU (E5-2650 0 - 2.00GHz). However, we obtained similar results using a Late 2011 MacBook Pro with 8GB RAM and one 4-core Intel i7 (2.4GHz), since we did not parallelize any of the computations. We mainly conducted our experiments on the server due to its larger main memory capacity.

[7]`https://www.redhat.com/en/resources/jboss-data-virtualization-query-performance-benchmark-study`

[8]`https://github.com/ontop/ontop/wiki/ObdalibQuestBenchmarks`

5.4.1.1 Ontology

We developed an underground mine ontology with the help of the faculty members of Istanbul Technical University's (ITU) mining engineering department and OHS professionals from Turkish Ministry of Labor. The ontology in total has 45 classes, 7 object properties, and 20 data properties. The ontology can be considered as small, however it can easily capture the concepts and relationships in the target database. It is also important to keep the ontology as simple as possible to obtain better performance with OBDA methods.

5.4.1.2 Policies

We derived 8 policies from the official health and safety regulations of mines in Turkey[9] to evaluate the framework. We selected those policies among many others, as they are easy to understand even without domain knowledge. The number of active instances typically grow, as the data in the database increases. Thus, the number of active instances can be in thousands even with eight policies. The regulation document only contains high level descriptions of policies. Thus, the complexity of the policy conditions depends on how policies are formalized and how concepts and properties used in this formalization are mapped over the target database. The implemented policies are depicted in Table 9.

5.4.1.3 Data

We were told that there are 1600 personnel (200 in a given shift), 185 gas sensors, and approximately more than 10 vehicles working (at the same time) at one of the largest coal mine facilities of Turkey. This specific company does not track its assets, however we assume that they are being tracked and there are 6400 assets (4 per personnel) in the underground facility to make the problem more challenging. Measurements of

[9]`http://www.lawsturkey.com/law/occupational-health-and-safety-law`

sensors are made every 5 seconds and tag readings are done arbitrarily. Based on this information, we filled in the database with randomly generated data. The above numbers belong to a fairly large mining facility.

The initial experimental setting represents smaller mines, which composed of 200 personnel, 30 gas sensors, 30 vehicles and 800 assets. Then, at each iteration of the experiment, we increased the size of the data by 100 personnel, 15 gas sensors, and 400 assets. Hence, we started with 1060 rows in total and increased it up to 83460 rows (160 iterations). We used the values in IJCRS 15 Data Challenge: Mining Data from Coal Mines [107] to generate values for gas levels realistically. However, this dataset does not contain any events risking OHS. Hence, we modified some sensor readings to generate unusual events (e.g., dangers or hazardous situations) to increase the number of active policies during experiments.

5.4.1.4 Implementation

We implemented the policy system using an OBDA framework Ontop [106] to map the mine ontology over the target application. We were provided with the empty schema of a smart mining solution and sample values for the columns in tables. The target application can track employees, assets, vehicles, and gas levels and all the data is stored in a single relational database. The schema did not allow efficient mappings for reasoning, since the database was normalized. Moreover, the dynamic and large part of the database mostly consists of location and gas readings, however we need only the most recent readings, and they can easily be fit into an in-memory database (i.e. H2[10]) along with the rest of the necessary data. For this reason, we developed a basic data federation solution (a software agent) that pulls the relevant information and the most recent readings from the target database into an OBDA friendly in-memory database.

[10]`http://www.h2database.com/html/main.html`

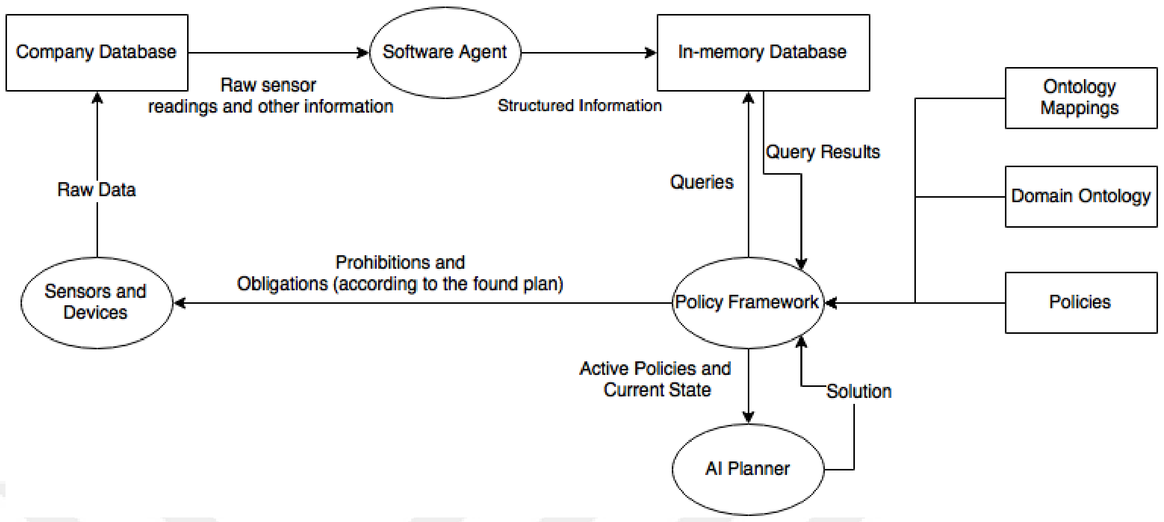


Figure 18: Overview of the experimental setup

Finally, we developed a software agent in Java, which uses Ontop API and connects to the database of the company to update our in-memory database. We do not try to evaluate the performance of Ontop, however we show that an OBDI approach could be used to implement a policy framework that can meet the performance demands of a large scale existing IoT system without performing any modifications to the target databases. It is important to note that only frequently changing data is the locations of personnel and assets, and sensor readings. Small changes in sensor readings might be useful for predicting the future value of gas levels or locations, however we do not need to update the sensor data so often to check policy activations. For efficiency reasons and to put less workload on the tables in the disk, we created two materialized views that capture the last locations and sensor readings. The agent simply queries these tables, when it needs to refresh the data in the memory. Figure 18 depicts an overview of our implementation for the experiments.

5.4.2 Experiments

The main goals of our experiments are as follows: 1. to see how fast we can run activation/expiration queries and how many policy instances we can create before a new update in the database; and 2. how these numbers change with respect to the increase in the number of working personnel, assets, and sensors.

Ontop translates the activation conditions of policies (SPARQL queries) into rewritten SQL queries and executes them over the H2 in-memory database. We measured two different times: the query execution time and the policy instance creation time. The latter measures how fast we can create Java objects which represent active instances (with bindings) of a policy. In other words, we execute an activation condition and create an active instance for each tuple in the result. It would not be useful just to show the query performance of Ontop as we still need to refine these activated policies to their addressees.

We evaluated the performance of our framework with the 8 policies presented in Table 9; each of the policies are given 3 warm-up runs and then 10 consecutive runs for evaluation. We discard the slowest run and take the average of the remaining 9 runs. For each policy, we timed how fast we created policy instances and computed the average. We repeated this experiment 10 times and calculated their mean and standard deviation.

5.4.2.1 Policy Activation Performance

Figure 19 illustrates the execution times of the activation conditions of our 8 different policies. The slowest run is about 1.3 milliseconds and the time does not increase as the data increases; hence the returned number of rows increases. Aside from the fact that Ontop being a very efficient framework, the main reason behind this is the fact that even the largest data set is manageable. The data stored on the memory is greatly reduced compared to the data stored on target system's disk, however the in-memory database still contains more information than the real system, since we increased the number of entities to test the system. The history of readings could be useful for doing prediction or anomaly detection, yet we only require the most recent readings and only a part of the database to see which policies are activated or

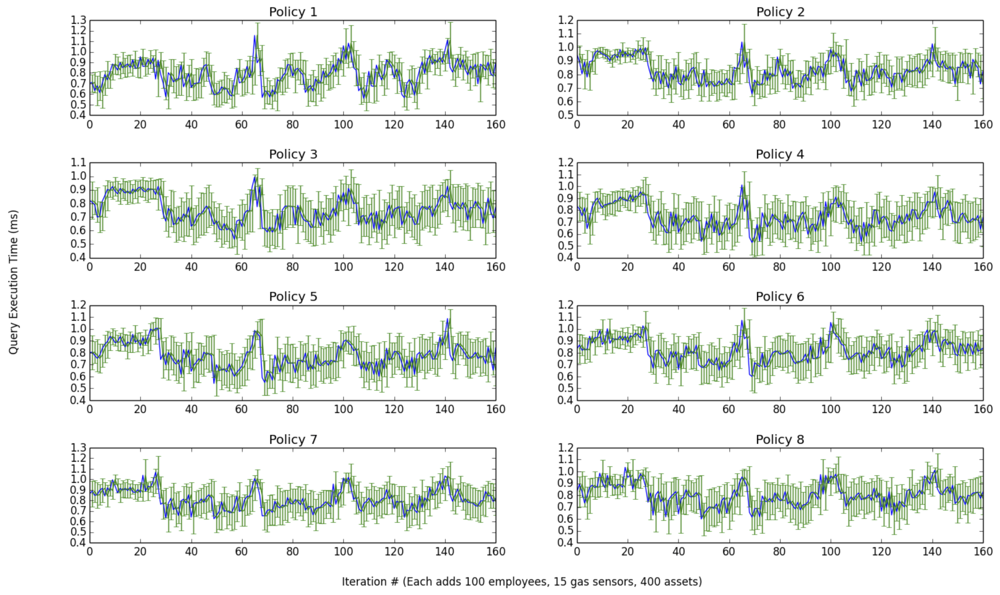


Figure 19: Query execution times of policies w.r.t. increasing dataset

expired. Isolating the arbitrary information from the policy system reduces the size of the data, hence improves the query times.

5.4.2.2 Policy Instance Creation Performance

The policy instance creation times for each policy are shown in Figure 20. It can be seen that deviation is higher and instance creation time varies for each policy. The deviations are caused by garbage collection and system related factors, however instance creation time mainly depends on the expiration conditions of policies and how fast we can bind them. The amount of time spent executing the expiration query and the number of tuples returned by it (the more tuples it returns the longer it takes) directly affects the object creation time. These run times could be improved by doing lazy binding. The slowest creation time for an active instance belongs to the first policy and the required time is ∼3 milliseconds. The fastest time is less than a millisecond, since that policy does not have an expiration condition—i.e., always active.

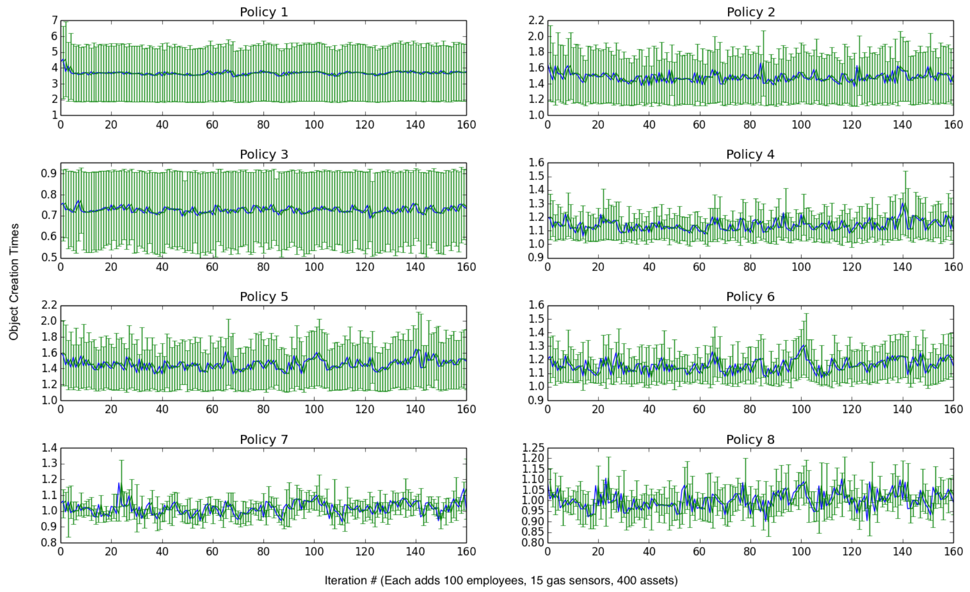


Figure 20: Active instance creation times w.r.t. increasing dataset

5.4.3 Planning Performance

It is crucial to model this high-risk domain in PDDL by making as few assumptions as possible and evaluate the performance of planners on real-world problem instances. This would allow us to test if planners can meet the performance demand posed by mining problems.

We modeled a simple domain and a problem in which a person is in panic at a fire zone as depicted by Figure 21. The initial state consists of firemen, transporter vehicles, drivers, gas levels, the map of the mine, and so on. We also put explosion risks to some regions to create policy conflicts such as some people are obliged to help the person in panic, but they are also prohibited to enter the regions with explosion risk. We used temporal Fast Downward planner [108] and OPTIC [100] to perform temporal planning. Although both planners were able to provide a solution for this scenario, they fail to solve slightly more complex problems.

We provide a sample initial state and the plan generated by OPTIC in Table 13. In this scenario, we assume that it is an early stage of a fire and it is possible for a

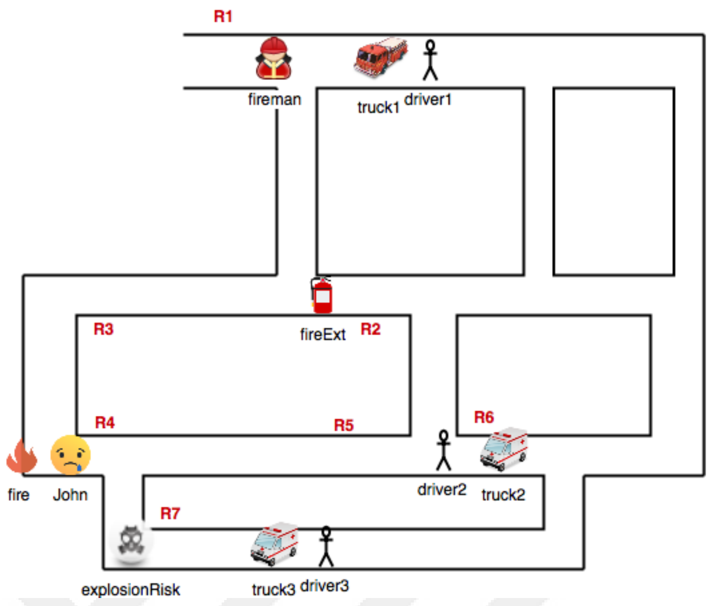


Figure 21: Initial state of the PDDL problem

Step	Action
1)	*move-asset* truck1 driver1 fireman r1 r2
2)	*go-to-region* truck2 driver2 r6 r5
3)	*move-asset* truck driver1 fireman r2 r3
4)	*go-to-region* truck2 driver2 r5 r4
5)	*move-asset* truck2 driver2 john r4 r3
6)	*go-to-region* truck1 driver1 r3 r2
7)	*move-asset* truck1 driver1 fireExt r2 r3
8)	*move-asset* truck2 driver2 john r3 r2
9)	*extinguish-fire* fireman fireExt r3 r4 fire
10)	*move-asset* truck2 driver2 john r2 r1

Table 13: Rescue plan.

driver to pick up the person in panic without the help of firemen. However, without much effort we could also model the scenario, in which fire has to be extinguished before saving the person in panic.

The planner is intelligent enough to perform a division of tasks between two available drivers. The planner assigns *driver1* to locate *fireman* along with the fire extinguisher, and take them both close to the fire. Meanwhile, *driver2*'s task is to bring *john* to safety, while the ambulance, which is closer to *john* delegates its task to avoid the explosion zone.

One issue we highlighted is that rarely a single planner can handle all the expressivity requirements of complex real-world domains such as mining. OPTIC seems to be the most promising available planner as it implements features like *ADL*, *durative-actions*, *numeric-fluents*, and*preferences* to represent this real-world application. However, we could not test it extensively as the planner started to fail due to a bug in its software.

5.4.4 Results

In order to increase the efficiencies, we can separate the querying and object creation processes; with two different threads running simultaneously, the system could execute activation and expiration queries while the activation that returns rows could create a new thread to create policy instances. We note that it takes between 0.2ms to 0.3ms to refresh the information in memory when there are more than 470k entries in sensor tables, so the process is efficient and applicable in highly dynamic situations. Since the mining company generates approximately 200k entries in a day from 185 gas sensors, this is applicable in real-time for the mining problem we have highlighted in this document.

Based on our experimentation, we observed that we can execute ∼770 queries in a second. Considering the gas levels are measured every 5 seconds and the database is only updated when their value change, at the worst case, we could run 3850 queries which may be expiration conditions of active instances or activation conditions of policies before the next update in the database. Furthermore, we can create ∼300 instances of a policy in a second, which allows us to refine a policy to 1500 assets and personnel in 5 seconds. This number approximately corresponds to all personnel in the mining facility. It is also important to note that the most of the policies will only be refined to a small part of the assets and personnel.

The results demonstrate that our framework can efficiently react to changes in sensor readings in real time and maintain its performance even if the number of employees and sensors go way above the initial numbers. The data stored in the memory is greatly reduced compared to the data stored on target system's disk. The history of readings could be useful for doing prediction or anomaly detection, yet we only require the most recent readings and only a part of the database to see which policies are activated or expired. Further performance evaluations could be conducted to investigate how efficiently the system can determine the affected actions and addressees be notified.

5.5 Discussion

In this chapter, we first discussed how ontology based data access (OBDA) and data federation can be adopted to integrate the policy system into in-use applications. Data federation methods provide a unified view of separate data sources that can be used by OBDA methods to map concepts and properties of an ontology. These methods introduce some overhead, however they make it possible to create a high level vocabulary to author policies without modifying the target application. Furthermore, OBDA mappings compensate OWL-QL's lack of expressivity to represent real life policies.

Through a case-study and publicly available data, we showed how the policy framework can be used to enhance health and safety of mine workers. Specifically, we showed that our system can respond to real-time events in a hasty manner and handle the increase in the volume of data. However, querying the database every time it is updated to maintain the normative state (activated/expired policies) is not a feasible solution for real applications. Thus, a smarter strategy has to be deployed for managing the normative state and reducing the load on the target database. For instance, when a tuple is updated, the application should determine and run only the queries of policies that might potentially get activated or expired. There is no need to execute queries of those unaffected.

Some limitations discussed in Chapter 3 such as finding the right violation costs for policies and validating policy descriptions still remains. In fact these topics become more important as the domain gets more complex and mistakes in policy descriptions may endanger worker's safety. In addition to the violation costs, we extended the policy descriptions with deadlines to better represent obligations and help with prioritizing the active policies. However, introducing deadlines increases the complexity of our conflict resolution strategy and requires domain actions to provide a formulation to estimate their execution times.

CHAPTER VI

DISCUSSIONS AND CONCLUSIONS

In this chapter, we highlight our contributions, discuss our framework's limitations, and outline possible solutions to those limitations. In Section 6.2 we highlight potential future research directions for this work, and we conclude the chapter in Section 6.3 by reiterating the main contributions of the thesis.

6.1 Discussions

The Internet of Things (IoT) is a highly agile (sensitive to availability, connectivity, and so forth) and complex (i.e. cross-connected devices) environment managed via the Internet. IoT promises a paradigm shift in which internet-enabled devices – and the services provided by them – are seamlessly meshed together such that end-users can experience improved situational awareness (e.g., "*Your usual route home has a 30 min. delay*"), added context (e.g., "*An accident at Broadway and 8th*"), and so forth to effectively and efficiently function in the environment. However, with growing adoption and deployment, the complexity of such IoT systems is fast growing. Such complexity is exacerbated in urban environments where large human populations will deploy ubiquitous devices (and in turn services), consume them, and interact with such systems to fulfill daily goals by meshing IoT services with external services such as location, weather and so forth. Hence, we believe that policies can be an effective means of regulating device actions.

We recall that there is a multitude of policy frameworks; some with rich policy representations[21, 22, 18], and some targeting pervasive environments [23, 24, 25]. However, they are either not scalable (to create and refine policy instances to large number of devices) or expressive enough to use high level concepts to describe devices and situations. For example, OWL-POLAR [22] uses OWL-DL and places various constraints (i.e. does not allow comparing data properties) on its policy language to preserve the decidability of query answering. However, data properties play a key role in IoT and the worst-case complexity of consistency checking and conjunctive query answering in OWL-DL is NEXPTIME-complete. Other policy frameworks such as [23, 24, 25] target pervasive environments and do policy reasoning on the edge devices. These policy systems can run on resource constrained devices by compromising either expressivity or other features like detecting conflicts. Even if policy conflicts can be detected, these systems do not offer a global resolution strategy that considers other available services.

The above mentioned policy frameworks are built for different purposes, thus they cannot cope with our requirements. In order to develop an expressive policy framework that provides scalable (w.r.t. increase in the data) reasoning mechanisms (i.e. policy conflicts, active/expired policies), in Chapter 3, we introduced a policy language that is based on OWL-QL, a formal model to represent policies, and an algorithm to detect conflicts between *obligation* and *prohibition* policies. In Chapter 4, we introduced our conflict resolution mechanism that utilizes a PDDL planner to minimize violation costs with respect to available resources (devices and services). We assume that the underlying knowledge base respects to the Unique Name Assumption (UNA) like PDDL to reformulate policy conflicts as a planning problem. Then, we discussed the complete policy framework through a smart home application. Unfortunately, OWL-QL lacks expressivity and requires the target application's database to conform to particular rules. Thus, we adopt ontology based data access (OBDA) to compensate the expressivity and to eliminate the constraints on the target database's schemata. Furthermore, many real life applications use multiple data sources. Thus, in Chapter 5, we discuss OBDA and data federation methods and show how our policy framework can be integrated with an intelligent mining solution that uses heterogeneous data sources.

6.1.1 Need for Policy Authoring Tools

Developing ontologies and creating their OBDA mappings considering design and performance requirements of an application require expertise and should be handled by the solution providers. However, authoring policies remains to be a challenging task, due to the steep learning curve users must go through to describe policies with the vocabulary provided by these ontologies. Furthermore, a user still has to validate the described policy to make sure that the system will behave as expected. i.e. does a policy get activated/expired exactly when user wanted it to be or does it only get refined to the intended devices. Thus, we believe that developing a tool for writing and validating policies is necessary.

Assisting users with writing queries is a well studied problem, but writing authoring policies is slightly different as policies consist of four different queries, a violation cost, and a modality. However, we believe that methods developed for question answering systems over linked data [109] or tools that provide a natural language interface for SPARQL queries [110] can be very helpful in developing a policy authoring tool with voice support, which has become a very popular user interaction method due to the advancements in voice assistants. As discussed in Chapter 3 we can also get inspiration from techniques such as conversational aspects in query generation with respect to schema information [89, 90], and pragmatically aware query formulation [91].

Validating a policy and its associated violation cost might require a simulation environment to present the user what is expected to happen in the system. If some scenarios can be simulated, then the system can try to learn violation costs with reinforcement learning methods [111]. Users can interact with the simulator and decide which policies should be violated in given scenarios. Otherwise, violation costs can be modeled as a utility function [112] to compute the right value according to some pre-defined features.

6.1.2 Need for Interoperability

Another issue that needs to be addressed is how action descriptions of policies can be matched with the services provided by the devices. This is also essential for creating the domain of the planning problems. It is possible to create semantically rich descriptions of device services [99, 113] using service description formats like RESTdesc[1], but it is not crucial for devices to share the same description format as the implementation of policy systems can simply be extended to support different standards. However, the exposed service descriptions must either use the same vocabulary or a vocabulary that can be translated to the policy ontology to maintain interoperability, which is one of the most fundamental issues of IoT systems. Only then the policy system can refine policies to individual device actions and resolve conflicts.

6.1.3 Need for Minimizing the Load on Target Databases

We note that in this work, we implemented the entire policy framework utilizing two different approaches; using a QL knowledge base [46] and using an OBDA [106] framework. Both of the methods have proven their efficiency, but in our context the OBDA approach seems to be superior as it improves the expressivity of the policy language and more practical in real life. Throughout the thesis we also discussed how fast the number of policies and devices can increase within an IoT system. Thus, the number of queries that the policy system needs to execute over its target database can increase even faster. The extra load that our policy system puts on the target database should not degrade the application's performance.

We implemented a naive normative state manager, which polls the database every five seconds (specific to the mining application) to determine activated and expired

[1]`http://restdesc.org`

policies. Even though we use efficient state-of-the-art solutions, executing more than a thousand policies every five seconds might degrade the regular performance of the application. However, not all policies have to be checked with the same frequency. For example, the location of a vehicle might change very frequently if it is moving but the model of a machine does not change. Furthermore, an update frequency analysis on data may help us with caching the less volatile data and removing atoms from conjunctive queries to increase the performance (i.e. reducing joins in a query).

An alternative can be developing a *pushing* mechanism for the query answering problem. For instance, if the application knows which columns are updated at a given transaction, it can either determine the affected policies and run their queries or try to directly update those policies' result sets. We can get inspired by the work on continuous queries [114, 115].

6.1.4 Need for Realistic Planning

We reformulated policy conflict resolution as a planning problem by encoding policies in PDDL by means of cost functions and goals. We then utilized a planner to avoid or mitigate conflicts found in plethora of policies. It is crucial to model domains by making as few assumptions as possible and evaluate the performance of planners on real-world problem instances. Unfortunately, one known issue is that there is rarely a single planner that can handle all the expressivity requirements of complex real-world domains. OPTIC [100], which is an open-source temporal planner that supports preferences and numeric fluents, seems to be the most promising planner we tested.

Our current approach allows us to embed OWL-QL reasoning rules and initial policies (preferences) into PDDL. However, there is much room for improvement. As discussed in previous chapters, we could not find a singular planner that could check if new preferences were activated, expired, or violated during planning. We had to develop a small program solely for this purpose. Furthermore, we still need to explore a variety of different solutions, among them translating web and device service descriptions into PDDL, interleaved (reactive) planning and scheduling, and contingent planning. These topics are not trivial, but they are studied extensively by the robotics community and necessary for applicability in real-world scenarios.

6.2 Future Work

Given the exploratory nature of this work there is a lot of topics from different fields (systems, robotics, multi-agent systems etc.) that must be cataloged and explored. Our primary focus is to keep the implementation of this framework lightweight and make it as efficient as possible. Thus, reducing the load that the policy framework puts on target databases would be the next step. Then, developing a policy authoring tool would have the highest priority.

The history of the data can be used for automatically extracting obligation policies from users behaviors by applying methods inspired by reverse correlation. For instance, the system can learn that it needs to lower the room temperature whenever the user leaves the house. Users decision might be affected by many things such as time, room and outside temperature, health condition of user, guests in the house, recent activities of user (such as running) and so on. Hence, if a frequent pattern is recognized by the system, this could be used to add a new policy. e.g., user is always turning on the TV, when she comes home. Automatically learned policies might include some correlations that humans might miss or may not think of.

The conflict detection algorithm can be extended to consider logical conflicts between obligation policies . Then, we hope to focus on a conflict resolution strategy, which could try to find a middle ground between conflicting policies. For instance, instead of keeping all devices silent when the baby is asleep, devices could be allowed to use sound as long as they don't wake the baby up. Another example (mentioned in

Section 3.4) could be a conflict between two people who prefer different room temperatures. A solution for this case could be setting to a temperature somewhere in the middle of both preferences. If modeled properly, constrained satisfaction problems can be solved by using preference-based planning or with general AI planners that use mixed integer linear programming. However, this may not be possible to resolve conflicts between policies opening and closing the same door at the same time. User interaction or a meta-policy approach might be necessary for such actions.

We assume that all device actions are permitted unless they are explicitly prohibited. A new modality can be introduced to represent actions that entities are authorized for performing but do not have to perform. However, this eventually could increase the number of policies that need to be maintained and the load put on the target database. If the prohibition policies are extensively used in an application, then introducing permission policies might be necessary to increase the performance. In that case, the conflict detection algorithm has to be revised to consider conflicts between permissions and prohibitions.

6.3 Conclusions

In this thesis, we have proposed a new policy framework that specifically targets IoT applications. Our focus has been to provide a lightweight semantic framework with efficient reasoning support that scales with data size. Due to the heterogeneity of resources and the dynamism associated with such environments, we developed an adaptive policy conflict resolution mechanism. We have kept the discussion in general terms, however we have also provided specific technology solutions and prototypical implementation of the research. Three of the main contributions of this thesis are:

A Scalable Semantic Policy Framework: A formal model to represent obligation and prohibition policies was introduced in Chapter 3. Then an efficient algorithm to detect policy conflicts in design time was introduced. We provided a use case to illustrate the way policies are used to manage the behaviors of devices in a smart home environment and to discuss the need for an automated conflict resolution mechanism. Finally, we provided a prototype implementation of the introduced framework and the use case.

Automated Conflict Resolution Strategy: Utilizing PDDL planners, in Chapter 4, we addressed the need for an automated conflict resolution mechanism introduced in Chapter 3. The expressivity of PDDL allowed us to reformulate conflicts as planning problems and to offer a global solution to the problem. We discussed the necessary features to represent policies in a PDDL and its shortcomings. Through controlled experiments, we have shown that utilizing a resolution mechanism that considers all available resources within a system is more flexible and powerful than rule based solutions.

Implementing the Framework in Real Applications: Methods to integrate the policy framework introduced in Chapter 3 with in-use applications were introduced in Chapter 5. We adopted ontology based data access (OBDA) to minimize the restrictions on the schemata of target databases and to improve the expressivity of our policy language. When the target application had heterogeneous data sources, we utilized data federation methods to create a unified single view that we can use for OBDA. We further discussed temporal planning and other topics to do conflict resolution in complex environments. These methods enabled us to model and reason with health and safety policies in a real coal mine environment. We finally evaluated the scalability of our approach through a database provided by an intelligent mining solution.

Through this thesis work, we have discussed and demonstrated the applicability of a knowledge-based policy framework for managing devices in IoT systems. We believe that this can be a useful foundation for a policy system that operates real applications.

Bibliography

[1] M. Krötzsch, "Owl 2 profiles: An introduction to lightweight ontology languages," in *Reasoning Web. Semantic Technologies for Advanced Query Answering*, pp. 112–183, Springer, 2012.

[2] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, "Owl 2 web ontology language: Profiles," w3c working draft, W3C, October 2008.

[3] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, pp. 1645–1660, Sept. 2013.

[4] A. Dohr, R. Modre-Opsrian, M. Drobics, D. Hayn, and G. Schreier, "The internet of things for ambient assisted living," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pp. 804–809, Ieee, 2010.

[5] C. Doukas and I. Maglogiannis, "Bringing iot and cloud computing towards pervasive healthcare," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 922–926, IEEE, 2012.

[6] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang, "A vision of iot: Applications, challenges, and opportunities with china perspective," *IEEE Internet of Things journal*, vol. 1, no. 4, pp. 349–359, 2014.

[7] P. Bak, R. Melamed, D. Moshkovich, Y. Nardi, H. Ship, and A. Yaeli, "Location and context-based microservices for mobile and internet of things workloads," in *2015 IEEE International Conference on Mobile Services*, pp. 1–8, IEEE, 2015.

[8] A. J. Jara, P. Lopez, D. Fernandez, J. F. Castillo, M. A. Zamora, and A. F. Skarmeta, "Mobile digcovery: discovering and interacting with the world through the internet of things," *Personal and ubiquitous computing*, vol. 18, no. 2, pp. 323–338, 2014.

[9] J. M. Bradshaw, "Making agents acceptable to people," in *Multi-Agent Systems and Applications III* (V. Mařík, M. Pěchouček, and J. Müller, eds.), (Berlin, Heidelberg), pp. 1–3, Springer Berlin Heidelberg, 2003.

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," *Policy*, vol. 1, pp. 18–38, 2001.

[11] G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok, "Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder," in *International Semantic Web Conference*, pp. 419–437, Springer, 2003.

[12] T. J. M. Bench-Capon, “Deep models, normative reasoning and legal expert systems,” in *Proceedings of the 2Nd International Conference on Artificial Intelligence and Law*, ICAIL ’89, (New York, NY, USA), pp. 37–45, ACM, 1989.

[13] A. Soeteman, *Logic in Law: Remarks on logic and rationality in normative reasoning, especially in law*, vol. 6. Springer Science & Business Media, 2013.

[14] G. Andrighetto, G. Governatori, P. Noriega, and L. W. van der Torre, *Normative multi-agent systems*, vol. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[15] W. W. Vasconcelos, M. J. Kollingbaum, and T. J. Norman, “Normative conflict resolution in multi-agent systems,” *Autonomous agents and multi-agent systems*, vol. 19, no. 2, pp. 124–152, 2009.

[16] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, “Smart objects as building blocks for the internet of things,” *IEEE Internet Computing*, vol. 14, no. 1, pp. 44–51, 2010.

[17] E. C. Lupu and M. Sloman, “Conflicts in policy-based distributed systems management,” *IEEE Transactions on software engineering*, vol. 25, no. 6, pp. 852–869, 1999.

[18] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, “Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement,” in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pp. 93–96, IEEE, 2003.

[19] H. J. Levesque and R. J. Brachman, “Expressiveness and tractability in knowledge representation and reasoning,” *Computational intelligence*, vol. 3, no. 1, pp. 78–93, 1987.

[20] R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, and A. Bandara, “Expressive policy analysis with enhanced system dynamicity,” in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pp. 239–250, ACM, 2009.

[21] L. Kagal, T. Finin, and A. Joshi, “A policy language for a pervasive computing environment,” in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pp. 63–74, IEEE, 2003.

[22] M. Sensoy, T. J. Norman, W. W. Vasconcelos, and K. Sycara, “Owl-polar: A framework for semantic policy representation and reasoning,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 12, pp. 148–160, 2012.

[23] T. Bourdenas, M. Sloman, and E. C. Lupu, “Self-healing for pervasive computing systems,” in *Architecting dependable systems VII*, pp. 1–25, Springer, 2010.

[24] N. Qwasmi, *Distributed Policy-Based Management Framework for Wireless Sensor Networks.* PhD thesis, University of Ontario Institute of Technology (Canada), 2014.

[25] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, “Ponder2: A policy system for autonomous pervasive environments,” in *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on*, pp. 330–335, IEEE, 2009.

[26] R. Fikes, P. Hayes, and I. Horrocks, “Owl-ql?a language for deductive query answering on the semantic web,” *Web semantics: Science, services and agents on the World Wide Web*, vol. 2, no. 1, pp. 19–29, 2004.

[27] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “PDDL—The Planning Domain Definition Language,” 1998.

[28] A. Uszok, J. M. Bradshaw, J. Lott, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, and H. Jung, “New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of kaos,” in *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, pp. 145–152, IEEE, 2008.

[29] R. Davis, H. Shrobe, and P. Szolovits, “What is a knowledge representation?,” *AI magazine*, vol. 14, no. 1, p. 17, 1993.

[30] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, “Owl 2 web ontology language primer,” *W3C recommendation*, vol. 27, no. 1, p. 123, 2009.

[31] D. L. McGuinness, F. Van Harmelen, *et al.*, “Owl web ontology language overview,” *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[32] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[33] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev, “The dl-lite family and relations,” *J. Artif. Int. Res.*, vol. 36, pp. 1–69, Sept. 2009.

[34] F. Baader, *The description logic handbook: Theory, implementation and applications.* Cambridge university press, 2003.

[35] D. Calvanese, G. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, “Tractable reasoning and efficient query answering in description logics: The dl-lite family,” *J. Autom. Reason.*, vol. 39, no. 3, pp. 385–429, 2007.

[36] D. Calvanese, G. D. Giacomo, M. Lenzerini, R. Rosati, and G. Vetere, "DL-Lite: Practical Reasoning for Rich DLs," in *Proc. of the DL2004 Workshop*, p. 92, 2004.

[37] R. Rosati, "Prexto: Query rewriting under extensional constraints in dl- lite," in *Extended Semantic Web Conference*, pp. 360–374, Springer, 2012.

[38] J. O'Shea, N. T. Nguyen, K. Crockett, R. J. Howlett, and L. C. Jain, *Agent and Multi-Agent Systems: Technologies and Applications: 5th KES International Conference, KES-AMSTA 2011, Manchester, UK, June 29–July 1, 2011, Proceedings*, vol. 6682. Springer Science & Business Media, 2011.

[39] H. Pérez-Urbina, E. Rodrıguez-Dıaz, M. Grove, G. Konstantinidis, and E. Sirin, "Evaluation of query rewriting approaches for owl 2," in *Proc. of the Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+ HPCSW 2012)*, vol. 943, 2012.

[40] A. Chortaras, D. Trivela, and G. Stamou, "Optimized query rewriting for owl 2 ql," in *International Conference on Automated Deduction*, pp. 192–206, Springer, 2011.

[41] S. Kikot, R. Kontchakov, and M. Zakharyaschev, "On (in) tractability of obda with owl 2 ql," CEUR Workshop Proceedings, 2011.

[42] R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev, "The combined approach to query answering in dl-lite.," in *KR*, 2010.

[43] H. Pérez-Urbina, B. Motik, and I. Horrocks, "Tractable query answering and rewriting under description logic constraints," *Journal of Applied Logic*, vol. 8, no. 2, pp. 186–209, 2010.

[44] M. Rodrıguez-Muro and D. Calvanese, "High performance query answering over dl-lite ontologies," in *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, 2012.

[45] R. Rosati and A. Almatelli, "Improving query answering over dl-lite ontologies," 2010.

[46] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 121–132, ACM, 2013.

[47] G. Klyne and J. J. Carroll, "Resource description framework (rdf): Concepts and abstract syntax." W3C Recommendation, 2004.

[48] Z. Kaoudi and I. Manolescu, "Rdf in the clouds: a survey," *The VLDB Journal*, vol. 24, no. 1, pp. 67–91, 2015.

[49] J.-J. C. Meyer, "Deontic logic: A concise overview," in *Deontic Logic in Computer Science: Normative System Specification*, John Wiley & Sons, 1993.

[50] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[51] D. Brickley and R. V. Guha, "Rdf schema 1.1," Jan. 2018.

[52] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken, "Kaos policy management for semantic web services," *IEEE Intelligent Systems*, vol. 19, no. 4, pp. 32–41, 2004.

[53] M. Schmidt-Schau, "Subsumption in kl-one is undecidable," in *Principles of Knowledge Representation and Reasoning: Proceedings of the 1st International Conference*, pp. 421–431, 1989.

[54] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosofand, and M. Dean, "SWRL: A semantic web rule language combining OWL and RuleML." W3C Member Submission, May 2004. Last access on Dez 2008 at: http://www.w3.org/Submission/SWRL/.

[55] P. Bonatti and D. Olmedilla, "Driving and monitoring provisional trust negotiation with metapolicies," in *Policies for Distributed Systems and Networks, 2005. Sixth IEEE International Workshop on*, pp. 14–23, IEEE, 2005.

[56] P. Bonatti and P. Samarati, "Regulating service access and information release on the web," in *Proceedings of the 7th ACM conference on Computer and communications security*, pp. 134–143, ACM, 2000.

[57] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett, "No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web," in *ESWS*, pp. 342–356, Springer, 2004.

[58] E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, and A. Schaeffer-Filho, "Amuse: autonomic management of ubiquitous e-health systems," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 3, pp. 277–295, 2008.

[59] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[60] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[61] D. Jackson, *Software Abstractions: logic, language, and analysis.* MIT press, 2012.

[62] A. Toninelli, A. Corradi, and R. Montanari, "A quality of context-aware approach to access control in pervasive environments," *MobileWireless Middleware, Operating Systems, and Applications*, pp. 236–251, 2009.

[63] A. Toninelli, R. Montanari, L. Kagal, and O. Lassila, "A semantic context-aware access control framework for secure collaborations in pervasive computing environments," in *International semantic web conference*, pp. 473–486, Springer, 2006.

[64] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Web Semant.*, vol. 5, pp. 51–53, June 2007.

[65] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF." W3C Recommendation, 2008.

[66] E. Sirin and B. Parsia, "Sparql-dl: Sparql query for owl-dl.," in *OWLED*, vol. 258, 2007.

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

[68] J. D. Ullman, "Information integration using logical views," in *International Conference on Database Theory*, pp. 19–40, Springer, 1997.

[69] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice.* Elsevier, 2004.

[70] T. Bylander, "Complexity results for planning.," in *IJCAI*, vol. 10, pp. 274–279, 1991.

[71] M. Belchior and V. T. da Silva, "Detection of normative conflict that depends on execution order of runtime events in multi-agent systems," in *Proceedings of the International Conference on Web Intelligence*, pp. 372–380, ACM, 2017.

[72] R. Camacho, P. Carreira, I. Lynce, and S. Resendes, "An ontology-based approach to conflict resolution in home and building automation systems," *Expert Systems with Applications*, vol. 41, no. 14, pp. 6161–6173, 2014.

[73] H. Kamoda, M. Yamaoka, S. Matsuda, K. Broda, and M. Sloman, "Policy conflict analysis using free variable tableaux for access control in web services environments," in *Proceedings of the Policy Management for the Web Workshop at the 14th International World Wide Web Conference (WWW)*, 2005.

[74] A. A. Nacci, B. Balaji, P. Spoletini, R. Gupta, D. Sciuto, and Y. Agarwal, "Buildingrules: a trigger-action based system to manage complex commercial buildings," in *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, pp. 381–384, ACM, 2015.

[75] R. Ananthanarayanan, M. Mohania, and A. Gupta, "Management of conflicting obligations in self-protecting policy-based systems," in *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pp. 274–285, IEEE, 2005.

[76] A. Fokoue, M. Bornea, J. Dolby, A. Kementsietsidis, and K. Srinivas, "An offline optimal sparql query planning approach to evaluate online heuristic planners," in *International Conference on Web Information Systems Engineering*, pp. 480–495, Springer, 2014.

[77] R. Kontchakov, M. Rezk, M. Rodriguez-Muro, G. Xiao, and M. Zakharyaschev, "Answering sparql queries over databases under owl 2 ql entailment regime," in *International Semantic Web Conference*, pp. 552–567, Springer, 2014.

[78] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artificial Intelligence*, vol. 116, no. 1-2, pp. 123–191, 2000.

[79] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting.* Cambridge University Press, 2016.

[80] J. Kvarnström, P. Doherty, and P. Haslum, "Extending talplanner with concurrency and resources," in *ECAI*, pp. 501–505, 2000.

[81] G. Sartor, "Normative conflicts in legal reasoning," *Artificial intelligence and law*, vol. 1, no. 2-3, pp. 209–235, 1992.

[82] H. Consortium, "Hypercat." `http://www.hypercat.io/standard.html`, 2016. Accessed: 2016-10-02.

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

[84] R. Kowalski, "Database updates in the event calculus," *The Journal of Logic Programming*, vol. 12, no. 1-2, pp. 121–146, 1992.

[85] S. Abiteboul, "Updates, a new frontier," in *International Conference on Database Theory*, pp. 1–18, Springer, 1988.

[86] E. Teniente and A. Olivé, "Updating knowledge bases while maintaining their consistency," *The VLDB Journal*, vol. 4, no. 2, pp. 193–241, 1995.

[87] G. De Giacomo, X. Oriol, R. Rosati, and D. F. Savo, "Updating dl-lite ontologies through first-order queries," in *International Semantic Web Conference*, pp. 167–183, Springer, 2016.

[88] Quetzal-RDF, "Quetzal." `https://github.com/Quetzal-RDF/quetzal`, 2016. Accessed: 2016-10-02.

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

[90] B. Hixon and R. J. Passonneau, "Open dialogue management for relational databases.," in *HLT-NAACL*, pp. 1082–1091, 2013.

[91] A. Viswanathan, G. de Mel, and J. A. Hendler, "Pragmatics and discourse in knowledge graphs," 2015.

[92] M. Klusch, A. Gerber, and M. Schmidt, "Semantic web service composition planning with owls-xplan," in *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pp. 55–62, sn, 2005.

[93] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "Htn planning for web service composition using shop2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377–396, 2004.

[94] A. Gerevini and D. Long, "Plan constraints and preferences in pddl3," tech. rep., Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy, 2005.

[95] D. Dou, "The formal syntax and semantics of web-pddl," tech. rep., University of Oregon, 2008.

[96] D. Dou, P. LePendu, S. Kim, and P. Qi, "Integrating databases into the semantic web through an ontology-based framework.," in *ICDE Workshops* (R. S. Barga and X. Zhou, eds.), p. 54, IEEE Computer Society, 2006.

[97] S. Richter and M. Westphal, "The lama planner: Guiding cost-based anytime planning with landmarks," *J. Artif. Int. Res.*, vol. 39, pp. 127–177, Sept. 2010.

[98] M. Helmert, "The fast downward planning system.," *J. Artif. Intell. Res.(JAIR)*, vol. 26, pp. 191–246, 2006.

[99] S. Mayer, R. Verborgh, M. Kovatsch, and F. Mattern, "Smart configuration of smart environments," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 3, pp. 1247–1255, 2016.

[100] J. Benton, A. Coles, and A. Coles, "Temporal Planning with Preferences and Time-Dependent Continuous Costs," in *Proceedings of the Twenty Second International Conference on Automated Planning and Scheduling (ICAPS-12)*, 2012.

[101] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki, "Just-in-time data virtualization: Lightweight data management with vida," in *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, no. EPFL-CONF-203677, 2015.

[102] L. Weng, G. Agrawal, U. Catalyurek, T. Kur, S. Narayanan, and J. Saltz, "An approach for automatic data virtualization," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pp. 24–33, IEEE, 2004.

[103] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati, "Journal on data semantics x," ch. Linking Data to Ontologies, pp. 133–173, Berlin, Heidelberg: Springer-Verlag, 2008.

[104] R. Kontchakov, M. Rodriguez-Muro, and M. Zakharyaschev, "Ontology-based data access with databases: A short course," in *Reasoning web. semantic technologies for intelligent data access*, pp. 194–229, Springer, 2013.

[105] M. Bienvenu, S. Kikot, R. Kontchakov, V. V. Podolskii, V. Ryzhikov, and M. Zakharyaschev, "The complexity of ontology-based data access with owl 2 ql and bounded treewidth queries," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pp. 201–216, ACM, 2017.

[106] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao, "Ontop: Answering SPARQL queries over relational databases," *Semantic Web*, vol. 8, no. 3, pp. 471–487, 2017.

[107] A. Janusz, M. Sikora, L. Wrobel, S. Stawicki, M. Grzegorowski, P. Wojtas, and D. Slkezak, "Mining Data from Coal Mines: IJCRS15 Data Challenge," in *Proceedings of RSFDGrC 2015*, vol. 9437 of *LNAI*, pp. 429–438, Springer, 2015.

[108] D. Furelos-Blanco, A. Jonsson, H. Palacios, and S. Jiménez, "Forward-search temporal planning with simultaneous events," *COPLAS 2018*, p. 11, 2018.

[109] A. Bouziane, D. Bouchiha, N. Doumi, and M. Malki, "Question answering systems: survey and trends," *Procedia Computer Science*, vol. 73, pp. 366–375, 2015.

[110] C. Pradel, O. Haemmerlé, and N. Hernandez, "Swip: a natural language to sparql interface implemented with sparql," in *International Conference on Conceptual Structures*, pp. 260–274, Springer, 2014.

[111] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart, "Utility-function-driven resource allocation in autonomic systems," in *null*, pp. 342–343, IEEE, 2005.

[112] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Autonomic Computing, 2004. Proceedings. International Conference on*, pp. 70–77, IEEE, 2004.

[113] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, E. Mannens, R. Van de Walle, and J. G. Vallés, "Integrating data and services through functional semantic service descriptions," in *Proceedings of the W3C Workshop on Data and Services Integration*, 2011.

[114] S. Babu and J. Widom, “Continuous queries over data streams,” *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001.

[115] Y. Watanabe and H. Kitagawa, “Query result caching for multiple event-driven continuous queries,” *Inf. Syst.*, vol. 35, pp. 94–110, Jan. 2010.

[116] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, and R. Rosati, “Ontologies and databases: The *DL-Lite* approach,” in *Semantic Technologies for Informations Systems – 5th Int. Reasoning Web Summer School (RW 2009)* (S. Tessaris and E. Franconi, eds.), vol. 5689 of *Lecture Notes in Computer Science*, pp. 255–356, Springer, 2009.

[117] G. De Giacomo, D. Lembo, X. Oriol, D. F. Savo, and E. Teniente, “Practical update management in ontology-based data access,” in *International Semantic Web Conference*, pp. 225–242, Springer, 2017.

[118] M. Rodrıguez-Muro, R. Kontchakov, and M. Zakharyaschev, “Obda with ontop,” in *Proc. of the OWL Reasoner Evaluation Workshop*, 2013.

[119] W3C, “Owl 2 web ontology language document overview (second edition),” Jan. 2018.