Automatic Resolution of Policy Conflicts in IoT Environments Through Planning

Emre Goynugur§ and Kartik Talamadupula† and Geeth de Mel‡ and Murat Sensoy§

§Ozyegin University
Department of Computer Science
Istanbul, Turkey
emre.goynugur,murat.sensoy @ ozyegin.edu.tr

†IBM Research
T.J. Watson Research Center
Yorktown Heights, NY, USA
krtalamad @ us.ibm.com

‡IBM Research
Daresbury Laboratory
Warrington, UK
geeth.demel @ uk.ibm.com

Abstract
The Internet of Things (IoT) is a highly agile and complex environment managed via the Internet. The management of such an environment requires robust automated mechanisms, since manual curation becomes a prohibitively expensive and near-impossible task. Motivated by this observation, we present a mechanism to resolve conflicts among the services provided by IoT devices in such environments, by reformulating conflict resolution as a planning problem. Specifically, we propose to use planning with soft constraints (preferences) to resolve conflicts that arise when fulfilling multiple goals using varied services in an IoT environment. We build on previous work on creating a semantic policy framework for detecting conflicts within an IoT environment, and present a proof-of-concept implementation of our approach to demonstrate its promise in the IoT space.

1 Introduction
The Internet of Things (IoT) is a highly agile (sensitive to availability, connectivity, and so forth) and complex (i.e., multitude of cross-connected devices) environment managed via the Internet. IoT promises a paradigm shift in which a multitude of 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. Due to advances in technology, the promise of IoT is fast becoming a reality, and in recent years there has been strong evidence in both the research and commercial sectors showing the applicability of this technology in multiple domains (Vermesan and Friess 2011; Iera et al. 2010; Gubbi et al. 2013). Furthermore, software systems and digital assistants such as IBM’s Watson, Apple’s Siri, Google’s Now, and Microsoft’s Cortana – all utilizing IoT enabled services – are fast becoming the go-to assistants for a variety of users. There is evidence to suggest that more and more users are tasking services provided by IoT ecosystems with automatically informing them about their environments (Holler et al. 2014; Li et al. 2011).

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. According to the United Nations, 54% of the world’s population lives in cities – a number that will increase to 66% by 2050 (Jayarajah et al. 2015). This will introduce many new services into IoT environments, and put a huge strain on the management of such systems. Furthermore, due to this explosion in services, conflicts are bound to occur more frequently, especially unexpected ones. For example, a mobile application may instantiate a meshed service that uses a location service which is prohibited in some locations. Manually curating such exceptions is impractical, if not impossible; thus, an emerging requirement in IoT environments is intelligent management of services, handling of exceptions, and the provision of automatic resolution techniques, so that the cognitive overload on the user is reduced.

Policies are typically used in system design to specify obligations and prohibitions, and automatically handle exceptions when obligations and prohibitions overlap due to unforeseen situations. 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.

In our recent work, we have created a semantic policy framework to efficiently detect conflicts within an IoT environment by restricting the expressivity of OWL-POLAR (Sensoy et al. 2012) to OWL-QL (Fikes, Hayes, and Horrocks 2004). In this paper, we present our initial work and thoughts on automatically handling these conflicts by reformulating the conflict resolution problem as an automated planning problem. Specifically, we formulate policy conflicts as preferences to enable the use of preference-based...
planning techniques to automatically resolve such conflicts to the best extent possible.

The rest of the paper is organized as follows: Section 2 provides the motivation for our work by means of use-cases. In Section 3 we formalize the policy representation and conflict detection, and define a number of terms that will be used in the paper. Section 4 presents the use of automated planning techniques in our system to automatically resolve conflicts, and other extensions that could be explored. In Section 5 we briefly discuss a preliminary proof-of-concept evaluation of our current system, and in Section 6 we discuss related literature. We conclude the document in Section 7 by discussing future work and by providing final remarks.

2 Motivation

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. Gartner, Inc. forecasts that 6.4 billion connected things will be in use worldwide in 2016, up 30 percent from 2015; this number will reach 20.8 billion by 2020 (van der Meulen 2015). This rapid increase in device numbers precipitates the need for an expressive and efficient policy framework, one which would allow its users to define high level rules that can be refined to individual devices. As there are a lot of diverse products from different manufacturers, it is difficult to tailor rules to cover all these individual devices. In addition to creating rules for device X or product type Y, users should be able to create rules by describing device properties or capabilities (e.g. devices with displays or devices that can play sound, and so forth).

Furthermore, these connected devices will be used by different people who have different preferences, habits, or expectations. It is safe to assume that these users together will generate a large number of policies and there will be more than one policy applying to a specific device; indeed, it is essential that multiple policies apply to a device in order to cover the diversity of management functions and of management domains (Lupu and Sloman 1999). However, whenever multiple policies apply to an object, there is usually a potential for some form of conflict.

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. The default action for a doorbell is to make sound to notify the household; however, if it fulfills this goal, it will violate the “no-sound” policy. In a truly connected environment, the notify action could be delegated to another device which uses different notification methods such as displaying a visual message, sending an SMS, and so on. Through the rest of the paper, we use this simple scenario as a running example.

Our policy framework is based on OWL-QL, which offers efficient reasoning mechanisms, can handle large numbers of policies, and detect conflicts before they actually happen; however, it cannot resolve or offer solutions for these conflicts. Users cannot be expected to manually resolve all such conflicts. In order to automate the process of conflict resolution and to address the missing features of our framework, we believe that automated planning techniques can be used.

3 Problem Definition

In this section, we formalize the representation of policies, and discuss the conditions for policy activation, expiration, and conflicts. We first introduce the terminology that will be used in the rest of this paper to represent policies.

3.1 Policy Terminology

As our method is based on an OWL-QL ontology, we use the terms OWL-QL ontology and knowledge base synonymously. In our context, a QL knowledge base (KB) consists of a TBox and an ABox. Concepts, properties, and axioms that describe relationships between concepts form the TBox of an ontology. We borrow syntax and semantics from the DL-Lite family to illustrate our TBox (Calvanese et al. 2007). For example, the statement: Computer ⊑ ElectronicDevice means that Computer class is a subclass of ElectronicDevice; and the statement ElectronicDevice ⊓ ∃playSound represents devices that can play sound. In PDDL terms, a TBox could be considered as the collection of types, type hierarchy, predicates, and derived predicate rules.

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 (Artale et al. 2009). In Description Logic, roles are binary relations between two individual objects; these can be considered binary PDDL predicates in our context: e.g. livesIn(John, NewYork). In this statement, livesIn is the role that connects John and NewYork. In PDDL, objects and predicates that exist in a planning state could be considered as an ABox—e.g. initial and goal states represent two different ABoxes.

3.2 Policy Representation

In this work, we formalize a policy as a six-tuple ($\alpha, N, \chi : \rho, a : \varphi, e, c$) where:

- $\alpha$ is the activation condition of the policy;
- $N$ is either obligation ($O$) or prohibition ($P$);
- $\chi$ is the policy addressee and $\rho$ represents its roles;
- $a : \varphi$ is the description of the regulated action; $a$ is the variable of the action instance and $\varphi$ describes $a$;
- $e$ is the expiration condition; and
• $c$ is the policy’s violation cost.

In a policy, $\rho$, $\alpha$, $\varphi$, and $e$ are expressed using a conjunction of query atoms. A query atom is in the form of either $C(x)$ or $P(x, y)$, where $C$ is a concept, $P$ is either a object or datatype property, $x$ is either a variable or individual, and $y$ may be a variable, an individual, or a data value.

For instance, using variables $b$ and $f$, the conjunction of atoms $\text{Baby}(b) \land \text{Sleeping}(b) \land \text{inFlat}(b, f)$ describes a setting where there is a sleeping baby in a flat. Concepts, properties, and individuals used in the definition of a policy should come from the underlying OWL-QL knowledge base. An example TBox of an OWL-QL ontology is shown in Table 1.

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

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

#### 3.3 Policy Activation and Expiration

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. For instance, in our scenario, the activation condition for the policy in Table 2 holds for the binding $\{ ?d = \text{dbell}, ?b = \text{John}, ?f = \text{flt} \}$. As a result, an activated policy instance can be created with this binding: "$\text{dbell is prohibited to perform MakeSound action until John is awake}$". Whenever the expiration condition of an active policy instance holds, that policy should be removed; e.g., the activated policy expires if the baby John wakes up.

Some active policies also expire when they are satisfied. For instance, obligation policies can expire after obligations are fulfilled. Let us consider the policy example in Table 3, 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 Bob rings the bell $\text{dbell}$, the active policy "$\text{dbell is obliged to notify an adult resident of the flat with sound}$" should be created. After notifying the targeted person 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.

#### 3.4 Policy Conflicts

In our work, three conditions have to hold true for two policies to conflict. First of all, these policies should be applied to the same policy addressee, e.g., same device or individual. Second, one policy must oblige an action, while the other prohibits the same action. Third, these two policies should be active at the same time in a consistent world state according to the underlying ontology. This situation forces the addressee to make a decision and violate one of the policies. 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. Let us note that the subsumption relation between the make sound and notify with sound actions are explicitly defined in the TBox. Table 4 illustrates a state in which the policies in Table 3 and Table 2 are in conflict.

It is almost trivial to figure out policy conflicts within a specific state of the world, as in our example. However, it is non-trivial at design time to reason if two policies will ever get into conflict. Our policy framework can detect such

---

### Table 2: An example prohibition policy.

<table>
<thead>
<tr>
<th>$\chi : \rho$</th>
<th>$?d : \text{Doorbell}(?x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$O$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>SomeoneAtDoor(?e) $\land$ producedBy(?e, ?x) $\land$ belongsToFlat(?x, ?f) $\land$ hasResident(?f, ?p) $\land$ Adult(?p)</td>
</tr>
<tr>
<td>$a : \varphi$</td>
<td>$?a : \text{NotifyWithSound}(?a) \land$ hasTarget(?a, ?p)</td>
</tr>
<tr>
<td>$e$</td>
<td>4.0</td>
</tr>
</tbody>
</table>

### Table 3: An example obligation policy.

<table>
<thead>
<tr>
<th>$\chi : \rho$</th>
<th>$?d : \text{Device}(?d)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$P$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Baby(?b) $\land$ Sleeping(?b) $\land$ inFlat(?b, ?f) $\land$ inFlat(?d, ?f)</td>
</tr>
<tr>
<td>$a : \varphi$</td>
<td>$?a : \text{MakeSound}(?a)$</td>
</tr>
<tr>
<td>$e$</td>
<td>10.0</td>
</tr>
</tbody>
</table>
Policy conflicts may arise between two given policies when the conditions outlined in the previous section are met. In such cases, it is essential for the system to devise a way to resolve the conflict and move forward. In this section, we outline a way of posing this conflict resolution problem as a planning problem, and using automated planning technology to solve that problem instance.

4 Representing OWL-QL in PDDL

Our policy framework exploits OWL-QL to cope with very large volumes of instance data. OWL-QL is less expressive compared to other OWL languages; however, it makes the implementation of conjunctive query answering using relational databases possible. Furthermore, we can map the concepts and relations of our knowledge base to PDDL using derived predicates.

Table names and their fields in our database can be mapped as predicates in a PDDL domain file. The instance data that exist in the database would represent the initial state of the system. In this context, a predicate is in the form of either \( C(x) \) or \( P(x, y) \), where \( C \) is a concept, \( P \) is either a object or datatype property, \( x \) is either a variable or individual, and \( y \) may be a variable, an individual, or a data value.

However, directly querying the knowledge base does not reveal the inferred information that may be deduced through the TBox. For this purpose, OWL-QL uses query rewriting to expand queries. Also, consistency checking is done by a disjunctive query that consists of conditions that might cause inconsistency based on the axioms in the TBox (Artale et al. 2009).

PDDL’s typing feature allow us 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.,

<table>
<thead>
<tr>
<th>1</th>
<th>Device(d0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Doorbell(d0)</td>
</tr>
<tr>
<td>3</td>
<td>Baby(b0)</td>
</tr>
<tr>
<td>4</td>
<td>Sleeping(b0)</td>
</tr>
<tr>
<td>5</td>
<td>inFlat(d0, f0)</td>
</tr>
<tr>
<td>6</td>
<td>inFlat(d0, f0)</td>
</tr>
<tr>
<td>7</td>
<td>SomeoneAtDoor(e0)</td>
</tr>
<tr>
<td>8</td>
<td>producedBy(e0, d0)</td>
</tr>
<tr>
<td>9</td>
<td>belongsToFlat(d0, f0)</td>
</tr>
<tr>
<td>10</td>
<td>hasResident(f0, p0)</td>
</tr>
<tr>
<td>11</td>
<td>Adult(p0)</td>
</tr>
</tbody>
</table>

 conflicts during design time, but we refrain from discussing that capability in this paper; instead, we focus on conflict resolution.

4.1 Representing OWL-QL in PDDL

Below, we give an example of a quasi-PDDL domain and its contents as it relates to our application\(^1\).

\( TV \sqsubseteq \exists hasSpeaker \sqcap \exists hasDisplay \). For this reason, instead of using the typing support of PDDL, we represent type(s) of an object with predicate(s). Hence, all these reasoning formulas can be integrated into the PDDL domain file by either rewriting action preconditions or using derived predicates. We believe that the latter approach is a cleaner solution rather than filling up action predicates with disjunctions.

Finally, OWL-QL doesn’t offer support for numerical constraints to provide efficient reasoning: e.g. it is not possible to express that one object can only be in one place or that a room can only have one temperature at a given time. PDDL can also help compensate for this limitation by embedding numerical constraints into the PDDL domain file. Below, we give an example of a quasi-PDDL domain and its

\(^1\)Note that this is not a complete and correct PDDL specification that a planner can parse.
4.2 Consistency Check

As explained in the previous section, 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 (generated using the TBox) 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 of the action descriptions are modified to include isConsistent in their preconditions along with (not (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. 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.

4.3 Policies as Preferences

The central contribution of this paper 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 preferences 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 Section 3.4 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. For example, instead of making a sound to notify (and fulfill the obligation) that someone is at the door, the doorbell could instead hand off the notification task to another currently active entity. An example of such
an entity could be a television; the television has a service that can visually notify that someone is at the door. This takes care of the obligation on the doorbell’s policy, while at the same time not violating the prohibition on the other policy of not making a sound while someone is sleeping.

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

1. preference pref-0 (gotNotifiedFor Adam Doorbell1)
2. preference pref-1 (NotifyWithVisual Doorbell1)

In the above, pref-0 stands for the obligation that when the doorbell rings, Adam (a person in the house) must be notified. pref-1 is the prohibition that whenever the doorbell rings, the notification must happen visually\(^2\). These two preferences are 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 following actions:

\[
\begin{align*}
1. & \text{:(action notify-with-sound} \\
2. & \text{:(parameters ?person ?device ?event ?notifyWithSound)} \\
3. & \text{:(precondition (and (playSound ?device ?notifyWithSound) } \\
4. & \text{(NotifyWithSound ?notifyWithSound) (Event ?event) } \\
5. & \text{(Person ?person) (isConsistent) (Device ?device))} \\
6. & \text{:(effect (and (gotNotifiedFor ?person ?event) } \\
7. & \text{not (isConsistent))) (increase (total-cost) 4)))}
\end{align*}
\]

The two actions above 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.

For instance, in our running example, let us assume that in addition the baby, the baby’s parents are sleeping as well, and the doorbell is rung. In this case, 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. In a real-world, deployed IoT scenario, 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.

Future work in casting policy constraint violations as planning preferences includes modeling the domain such that the planner can pick not only which constraints are violated, but also the degree to which those constraints are violated. For example, in a scenario where there is a temperature controller, and two different people have competing preferences (one wants the room to be cold, the other would like it warm), the domain could be modeled in a way that the eventual solution is a compromise between too cold and too warm.

**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. Furthermore, as with many other real-world applications which do service compositions, interleaving planning is essential for IoT applications. However, in this paper we do not focus on these issues.

5 Preliminary 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. 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. Although our approach is based on PDDL, we initially tested it with the JSHOP2 (Ilghami and

\(^2\)An alternate way of encoding this preference would be to define a (DoNotUseSound ?entity) predicate; this is a domain modeling question.
Nau 2003) HTN planner. We then used the LAMA (Richter and Westphal 2010) planner to determine if we could integrate OWL-QL rules into PDDL. Finally, we converted policies into preferences and used SGPlan5 (wei Hsu, Wah, and et al. 2006). As future work, we intend to build a planning system that supports all three requirements: action costs, derived predicates, and preferences.

5.1 JSHOP2

We initially used the JSHOP2 planner since our policy framework was also developed in Java. Additionally, JSHOP2’s external calls allowed us to simulate interleaved planning that requires the execution of non-deterministic actions such as locate and search. In an IoT environment, where there are a lot of sensors, interleaving planning with sensing and execution seems to be essential.

However, JSHOP2 does not support domain predicates or preferences. We thus had to develop a component to validate the plans that were generated by JSHOP to check if caused an inconsistent world state or violated/activated new policies. Since JSHOP2 does not support derived predicates, we had to rewrite action preconditions with disjunctions. Unfortunately, disjunctions in the domain file cause JSHOP2 to product a combinatorial number of duplicate plan files — a large burden on computational resources.

5.2 Evaluation of Derived Predicates

LAMA, which is a planner that builds on Fast Downward (Helmer 2006), supports ADL descriptions, actions costs, and derived predicates. We thus used LAMA to check if we could accommodate OWL-QL reasoning into PDDL.

Throughout our experiments, we used the same ontology that we used with JSHOP2. However, instead of filling up action descriptions with disjunctions in the domain file, we defined re-write rules as derived predicates. Without any additional effort, we successfully managed to find the same plans that we found using JSHOP2 and external programs. Additionally, we tested our approach by slightly modifying the ontology (by making some classes disjoint) to make the planner reach an inconsistent state. As expected, the planner did not find any plans with the modified ontologies. Finally, utilizing derived predicates did not cause LAMA to produce duplicate plan files, which was the case with JSHOP2. Below we reproduce one of the plans produced by LAMA.

5.3 Policies as Preferences

The main focus of this paper was to automate policy conflict resolution using AI planners. In order to realize this goal, we first needed to reformulate the conflicting situation as a planning problem. In the previous sections, we discussed how we did this reformulation. In this section, we only discuss how we conducted our preliminary experiments.

As mentioned previously, we could not find a planner that supported all the features we needed in this work. Thus, class type inferences and inconsistent states (according to the underlying ontology) are excluded from these experiments. However, we still use the same initial and goal state files without disjunctions and derived predicates. We chose the winner of the IPC 2006 satisficing track, SGPlan5 (wei Hsu, Wah, and et al. 2006) for our evaluation. SGPlan5 does not support action costs, and actions have to be defined in a STRIPS-like representation. After slightly modifying our domain and problem files to make them SGPlan5 compatible and to include preferences (policies), we were able to produce the same results as we did with the other planners. Below, we reproduce a plan similar to the one shown previously; however, this plan is produced by SGPlan5 taking preferences into account. Thus the notify-people-with-sound action is now replaced by the notify-people-with-visual action.

Although using preferences makes the formulation of our original problem more natural, it does not completely solve it. 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 preference is activated or an active policy is expired.

Generally speaking, the semantic representation of our policies is very suitable for generating a goal state with preferences. Moreover, specifying a goal state instead of specifying an action name is more intuitive in an heterogeneous IoT domain. Although HTN planners can solve the same problems, PDDL makes it easier to automatically generate domain files, as we just need to add action descriptions without defining an action hierarchy. Furthermore, PDDL’s support for derived predicates allows us to embed rewrite rules of a QL ontology into the domain file without filling action preconditions with disjunctions. This allows us to find plans by using class inference rules.
In our implementation we were able to integrate QL reasoning into PDDL; however, the planners we found do not offer a way of dealing with interleaved planning, or with the issue of updating preferences (policies) during planning. In our context, these are the shortcomings of existing PDDL-based planners, and we currently still need to analyze generated plans with an external script and re-plan if needed.

6 Related Work

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 (Dou 2008) 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 (Dou et al. 2006) 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 (Dou 2008) explains what PDDL lacks to fully represent ontologies.

In addition to PDDL planners, there is an HTN planner called HTNPLAN-P (Sohrabi, Báier, and McIlraith 2009) that supports preferences. HTNPLAN-P extends PDDL3 with HTN-specific constructs; in contrast to the state-centric preferences of PDDL, HTPLAN-P supports preferences that apply to tasks; e.g., “when booking inter-city transportation, I prefer to book a flight”.

KAoS (Uszok et al. 2003) was the first effort to offer an ontology based approach for creating a policy framework. Policies are defined as concepts in the ontology using description logic class expressions based on constructs such as subclassOf or object properties like hasCapability, inRoom etc. Thus, it is not possible to use variables in policy descriptions. KAoS can detect conflicts and if a conflict is detected, KAoS checks policies’ update times and priority values to resolve conflicts.

OWL-POLAR (Sensoy et al. 2012), which is an OWL-DL based policy framework, can also detect conflicts during design time. Although a conflict resolution strategy is not implemented in that work, authors suggest that an AI planner could be used to avoid conflicts. In addition, it is stated in their paper that doctrines of legal theory and practice could be adopted to resolve policy conflicts. However, OWL-POLAR uses the Pellet reasoner, which is slow and inefficient considering IoT requirements. Furthermore, it may not be possible to completely add the inference and consistency rules of OWL-DL into the PDDL domain.

Rei (Kagal, Finin, and Joshi 2003) is another effort towards an ontology based policy framework. It is based on OWL-Lite and allows policies to be specified as constraints over allowable and obligated actions on resources in the environment (Kagal, Finin, and Joshi 2003). Rei is implemented with Prolog and allows using a logic-like language to describe policies. Prolog provides Rei with the flexibility of specifying relations like role-value maps that are not directly possible in OWL. However, Prolog cannot be used for expressing all DL expressions. Furthermore, Rei cannot detect conflicts between policies at design time; it can detect conflicts when they happen and uses meta-policies to resolve them.

A different method – which is not based on an ontology – proposed in (Vasconcelos, Kollingbaum, and Norman 2009) resolves conflicts by manipulating the constraints associated with the policy’s variables, removing any overlap in their values. This approach is similar to adding the conflicting obligation policy as an exception to the prohibition policy. For example, if devices are not allowed to make sound when the baby is asleep, this approach modifies the prohibition policy to exclude the doorbell.

7 Future Work & Conclusion

In this paper, we discussed how an AI planner could be used in a lightweight policy framework to automate policy conflict resolution. The policy framework is based on OWL-QL mainly because it targets IoT applications that generate large volumes of instance data, and query answering is an essential task in these systems. We managed to encode type inference and consistency check rules of an OWL-QL ontology into a PDDL domain file automatically using a Java program. Furthermore, we converted policies into preferences on PDDL, and used an AI planner to search for a solution which could alleviate the possible conflict.

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 sections, 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, interleaving actions and translating web and device service descriptions into PDDL are not trivial tasks to achieve, but they are necessary for applicability in real-world scenarios.

Another aspect of future work that we hope to focus on is 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 (outlined in Section 4.3) 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. We think that planners would also be helpful in this approach.

Acknowledgements

We wish to thank Biplav Srivastava and Shirin Sohrabi for their help and discussions on planning and preferences. We would also like to thank the anonymous reviewers for their helpful feedback.
References


wei Hsu, C.; Wah, B. W.; and et al. 2006. Handling soft constraints and goals preferences in sgplan.