OWL-POLAR: A Framework for Semantic Policy Representation and Reasoning

Murat Sensoy [a,*] Timothy J. Norman [a] Wamberto W. Vasconcelos [a]
Katia Sycara [b]

[a] *Department of Computing Science, University of Aberdeen, AB24 3UE, Aberdeen, UK*
[b] *Carnegie Mellon University, Robotics Institute, Pittsburgh, PA 15213, USA*

Abstract

In a distributed system, the actions of one component may lead to severe failures in the system as a whole. To govern such systems, constraints are placed on the behaviour of components to avoid such undesirable actions. Policies or norms are declarations of soft constraints regulating what is prohibited, permitted or obliged within a distributed system. These constraints provide systems-level means to mitigate against failures. A few machine-processable representations for policies have been proposed, but they tend to be either limited in the types of policies that can be expressed or are limited by the complexity of associated reasoning mechanisms. In this paper, we present a language that sufficiently expresses the types of policies essential in practical systems, and which enables both policy-governed decision-making and policy analysis within the bounds of decidability. We then propose an OWL-based representation of policies that meets these criteria and reasoning mechanisms that use a novel combination of ontology consistency checking and query answering. The proposed policy representation and reasoning mechanisms allow development of distributed agent-based systems that operate flexibly and effectively in policy-constrainted environments.

Key words: Semantic Web, Policies, Norms, Conflict Resolution, Multi-agent Systems

* Corresponding author. Tel: +44 7522474621
Email addresses: m.sensoy@abdn.ac.uk (Murat Sensoy),
t.j.norman@abdn.ac.uk (Timothy J. Norman),
w.w.vasconcelos@abdn.ac.uk (Wamberto W. Vasconcelos),
katia@cs.cmu.edu (Katia Sycara).

1 Introduction

Multi-agent systems are distributed systems whose components are intelligent software agents [37]. Each agent has a set of goals, capabilities and resources. Using their capabilities and resources, agents within a multi-agent system execute actions to achieve their goals. Individual actions, however, may result in undesirable consequences within the system. It is, therefore, important to regulate the actions of agents to reduce the risk of these undesirable consequences [9].

Authorities may enforce *policies* to regulate actions of agents within a specific context. Policies are soft constraints determining in which situations a certain action is obliged, permitted, or prohibited [35]. Authoring polices for a specific domain requires the ability to imagine all implications of policies within that domain. This challenge may lead to policies that are inconsistent or incomplete and, possibly, policies that do not fully capture the intentions of the authors. Furthermore, different authorities with different goals may enforce different policies in the same context. For instance, governments enforce policies to promote health and safety, while companies create their own policies to promote productivity and consumer satisfaction. Policies of a single authority may be assumed to be consistent, however policies of different authorities may conflict in a specific context. With mechanisms to reason with and about policies, one is able to anticipate contexts in which conflicts may arise; these mechanisms should enable the resolution of conflicts at design time, thus providing guarantees to systems before they are executed. Without such reasoning mechanisms, it would be difficult for policy authors to identify the contexts in which their policies conflict. The identification and resolution of policy conflicts are left to the agents operating in those contexts, with potentially undesirable run-time effects.

In this paper, we present a novel and powerful OWL 2.0 [15] knowledge representation and reasoning mechanism for policies: OWL-POLAR (an acronym for OWL-based POlicy Language for Agent Reasoning). Policies, which are also known as *norms*, are system-level principles of ideal activity that are binding upon the components of that system. Depending on the nature of the system itself, policies may serve to control, regulate or simply guide the activities of components. In systems security, for instance, the aim is typically to control behaviour such that the system complies with the policies [31]. In real socio-technical systems, however, there are important limits to this and the aim is to develop effective sets of policies along with incentives to regulate behaviour [2]. In systems of autonomous agents, the term norm is most prevalent, but the concept and issues remain the same [7]; for example, norms are used to regulate the behaviour of agents representing disparate interests in electronic institutions [13]. The objective of this research is to fulfil the essential requirements of policy representation, reasoning, and analysis. In meeting this objective three key requirements must be met:

(1) System/institutional policies must be machine understandable and underpinned by a clear interpretation.
(2) The representation must be sufficiently expressive to capture the notion of a policy across domains.
(3) Policies must be able to be effectively shared/interpreted at run-time.

The choice of OWL 2.0 as an underlying language addresses the first requirement, but in meeting the second two, we must clearly outline what is required of a policy language and what reasoning should be supported by it. The desiderata of a model of policies that motivates the language OWL-POLAR are as follows:

- **Representational adequacy.** Policies (or norms) must capture the distinction between activities that are required (obliged), restricted (prohibited) and, in some way, authorised but not necessarily expected (permitted) by some representational entity within the environment. It is essential to capture the authority from which the policy/norm comes, the subject (agent) to whom it applies, the object (activity) to which the policy/norm refers, and the circumstances within which it applies.
- **Supporting decisions.** Any reasoning mechanism that is driven/guided by policies must support both the determination of what policies/norms apply in a given situation, and what activities are warranted by the normative state of the agent if it were to comply with these policies.
- **Supporting analysis.** Any reasoning mechanism that is driven/guided by normative/policy constraints must support the assessment of policies in terms of: (i) whether a policy/norm is meaningful and (ii) whether norms conflict, and in what circumstances they do conflict.

We believe that this desideratum of a model of policies can be met within the confines of OWL-DL. If this claim can be shown to be valid (as we aim to do within this paper), we believe that OWL-POLAR provides, for the first time, a sufficiently expressive policy language for which the key reasoning mechanisms required of such a language are decidable. These mechanisms allow intelligent software agents and policy authors to reason within context. Inconsistent and unfounded policies are automatically identified. Heterogeneous knowledge across different authority domains could be better integrated by revealing the context in which the policies they enforce may be in conflict. Furthermore, the proposed mechanisms equip agents with the capability to resolve such conflicts.

The paper is organised as follows: in Section 2 we formally specify the OWL-POLAR language within OWL-DL; in Section 3 we describe how a set of active policies may be computed, and how decisions about what activities are warranted by some set of policies may be made; then in Section 4 we present in detail the reasoning mechanisms that support the analysis of policies and conflict detection between policies; in Section 5 we present conflict resolution mechanisms for policies. Section 6 discusses the computational complexity of the proposed reasoning

mechanisms. OWL-POLAR is then compared to existing languages for policies in Section 7, and we present our conclusions in Section 8.

2 Semantic Representation of Policies

The proposed language for semantic representation of policies is based on OWL-DL [15]. An OWL-DL ontology $o = (TBox_o, ABox_o)$ consists of a set of axioms defining the classes and relations ($TBox_o$) as well as a set of assertional axioms about the individuals in the domain ($ABox_o$). Concept axioms have the form $C \sqsubseteq D$ where C and D are concept descriptions, and relation axioms are expressions of the form $R \sqsubseteq S$, where R and S are relation descriptions. The ABox contains concept assertions of the form $C(a)$ where C is a concept and a is an individual name, and relation assertions of the form $R(a, b)$, where R is a relation and a and b are individual names.

Conjunctive semantic formulas are used to express policies. A conjunctive semantic formula $F^o_{\vec{v}} = \bigwedge_{i=0}^{n} \phi_i$ over an ontology o is a conjunction of atomic assertions ϕ_i, where $\vec{v} = \langle ?x_0, \ldots, ?x_n \rangle$ represents a vector of variables used in these assertions. For the sake of convenience, we assume $\bigwedge_{i=0}^{n} \phi_i \equiv \{\phi_1, \ldots \phi_n\}$ in order to consider a conjunctive formula as a set of atomic assertions. Based on this, $F^o_{\vec{v}}$ can be considered as $T^o_{\vec{v}} \cup R^o_{\vec{v}} \cup C^o_{\vec{v}}$, where $T^o_{\vec{v}}$ is a set of type assertions using the concepts from o, e.g., $\{student(?x_i), nurse(?x_j)\}$; $R^o_{\vec{v}}$ is set of relation assertions using the relations from o, e.g., $\{marriedTo(?x_i, ?x_j)\}$; $C^o_{\vec{v}}$ is a set of constraint assertions on variables. Each constraint assertion is of the form $?x_i \triangleleft \beta$, where β is a constant and $\triangleleft$ is any of the symbols $\{>, <, =, \neq, \geq, \leq\}$. A constant is either a data literal (e.g, a numerical value) or an individual defined in o.

Variables are divided into two categories; datatype and object variables. A datatype variable refers to data values (e.g., integers) and can be used only once in $R^o_{\vec{v}}$. On the other hand, an object variable refers to individuals (e.g., University_of_Aberdeen) and can be used freely many times in $R^o_{\vec{v}}$. Equivalence and distinction between the values of object variables can be defined using OWL properties *sameAs* and *differentFrom* respectively, e.g., *owl:sameAs(?x,?y)*. In the rest of the paper, we use the symbols α, ρ, φ, and e as a short hand for semantic formulas. Given an ontology o, a conditional policy is defined as $\alpha \longrightarrow N_{\chi:\rho}\,(a : \varphi)\,/e$, where

(1) α, a conjunctive semantic formula, is the activation condition of the policy.
(2) $N \in \{O, P, F\}$ indicates if the policy is an obligation, permission or prohibition.
(3) χ is the policy addressee and ρ describes χ using only the *role* concepts from the ontology (e.g., $?x : student(?x) \wedge female(?x)$, where *student* and *female* are defined as sub-concepts of the $role$ concept in the ontology). That is, ρ is of the form $\bigwedge_{i=0}^{n} r_i(\chi)$, where $r_i \sqsubseteq role$. Note that χ may directly refer to a

Table 1
A person has to leave a location when there is a fire risk.

α	$Place(?b) \wedge hasFireRisk(?b, true) \wedge in(?x, ?b)$
N	O
$\chi : \rho$	$?x : Person(?x)$
$a : \varphi$	$?a : LeavingAction(?a) \wedge about(?a, ?b) \wedge hasActor(?a, ?x)$
e	$hasFireRisk(?b, false)$

specific individual (e.g., *John*) in the ontology or a variable.

(4) $a : \varphi$ describes what is prohibited, permitted or obliged by the policy. Specifically, a is a variable referring to the action to be regulated by the policy and φ describes a as an action instance using the concepts and properties from the ontology (e.g., $?a : SendFileAction(?a) \wedge hasReceiver(?a, John) \wedge hasFile(?a, TechReport218.pdf)$, where *SendFileAction* is an *action* concept). Each action concept has only a number of functional relations (aka. functional properties) [15] and these relations are used while describing an instance of that action.

(5) e defines the expiration condition.

Table 1 illustrates how a conditional policy can be represented using the proposed approach. This policy states that a person is obliged to leave a location when there is a fire risk.

Given a semantic representation for the state of the world, policies are used to reason about actions that are permitted, obliged or prohibited. Let Δ_o be a semantic representation for a state of the world based on an ontology o. Each state of the world is partially observable; hence Δ_o is a partial representation of the world. Δ_o itself is represented as an ontology composed of $(TBox_o, ABox_\Delta)$ where $ABox_\Delta$ is an extension of $ABox_o$.

3 Reasoning with Policies

When its activation conditions are satisfied, a conditional policy leads to an activated policy. Definition 1 summarizes how a conditional policy is activated using ontological reasoning over a state of the world. Here we use query answering to determine activated policies and reason about actions. The query answering mechanism we use in this work is DL-safe; i.e. variables are bound only to the named individuals, to guarantee decidability [16]. In this section, we address some of the key issues in supporting decisions governed by policies: activation and expiration, and reasoning about interactions between policies and actions.

Definition 1 Let Δ_o be a state of the world represented based on a domain ontology o. If there is a substitution σ such that $\Delta_o \vdash (\alpha \wedge \rho) \cdot \sigma$, but there is no substitution

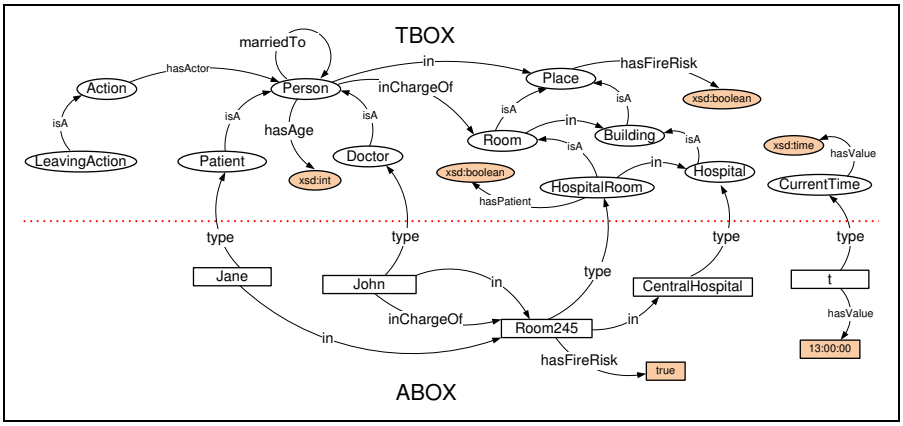

Fig. 1. A partial state of the world represented based on a domain ontology.

σ' such that $\Delta_o \vdash (e \cdot \sigma) \cdot \sigma'$, then the policy $(N_\chi\,(a : \varphi)) \cdot \sigma$ becomes active. This policy expires when there exists a substitution σ' such that $\Delta_o \vdash (e \cdot \sigma) \cdot \sigma'$. ∎

3.1 Policy Activation

A policy is activated for a specific agent when the world state is such that the activation condition holds for that agent and the expiration condition does not hold, and expires when this latter condition holds. The above definition is rather standard [19], but we now describe how this is implemented efficiently through query answering. A conjunctive semantic formula can be trivially converted to a SPARQL query [26] and can be evaluated by OWL-DL reasoners with SPARQL-DL [30] support such as Pellet [30] to find a substitution for its variables satisfying a specific state of the world. Therefore, we can test $\Delta_o \vdash (\alpha \wedge \rho) \cdot \sigma$ by writing a query for $(\alpha \wedge \rho)$ and testing whether it is entailed by Δ_o or not. Consider the conditional policy in Table 1 and assume that we have the partially represented state of the world in Figure 1. We can write the semantic query in Figure 2 to find σ for the conditional policy. When we query the state of the world using SPARQL, each result in the result set provides a substitution σ; in our case, we have two σ values: $\{$*?x/John, ?b/Room245*$\}$ and $\{$*?x/Jane, ?b/Room245*$\}$, representing that there is a fire risk in the room 245 of the Central Hospital and that John and Jane are in that room.

Now, using the computed σ values, we should try to find a σ' such that $\Delta_o \vdash (e \cdot \sigma) \cdot \sigma'$. In our case, for this purpose, we can use the semantic query "*q():- hasFireRisk(Room245, false)*". When the SPARQL representation of this query is executed over the state of the world shown in Figure 1, it returns *false*; that is the RDF graph pattern represented by the query could not be found in the ontology. This means that the policy in Table 1 should be activated using the variable bindings in σ. The result is activations of O_{John}*(?a : LeavingAction(?a)* $\wedge$ *about(?a,Room245))* and O_{Jane}*(?a : LeavingAction(?a)* $\wedge$ *about(?a,Room245))*. These policies mean that *John* and *Jane* are obliged to leave the *room 245*; the obligation expires when the

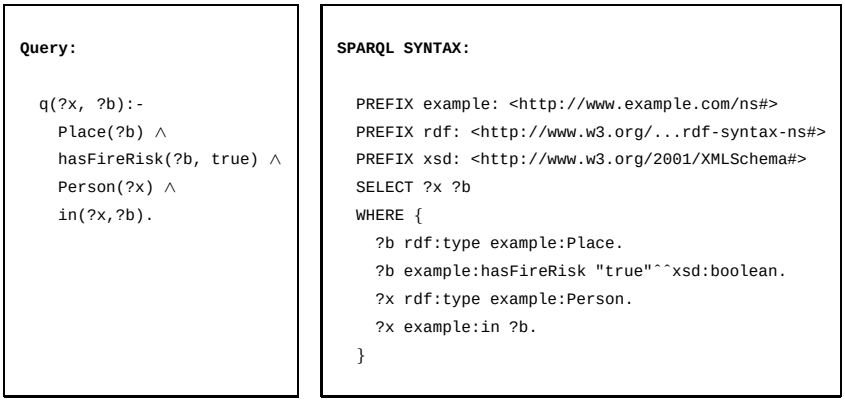

Fig. 2. Query for the activation of a policy.

fire risk is removed.

3.2 Reasoning about Actions

Let us assume that a specific action $a' : \varphi'$ will be performed by x, where a' is a URI referring to the action instance and φ' is a conjunctive semantic formula describing a' without using any variables. Let Δ_o be the current state of the world. We can test if the action a' is permitted, forbidden or prohibited in Δ_o. For this purpose, based on Δ_o, we create a "sandbox" (hypothetical) state of the world Δ'_o to make *what-if* reasoning [34], i.e., Δ'_o shows what happens if the action is performed. This is achieved by simply adding the described action instance to Δ_o, i.e., $\Delta'_o = \Delta_o \cup \varphi'$. For example, the state of the world in Figure 1 is extended using action instance *LeaveAct_1: LeavingAction(LeaveAct_1)* $\wedge$ *hasActor(LeaveAct_1,John)* $\wedge$ *about(LeaveAct_1,room245)*. The resulting state of the world is shown in Figure 3.

For each active policy $N_x(y : \varphi_y)$, we test the expiration conditions on Δ'_o as explained before. If the policy's expiration conditions are satisfied, we can conclude that the action $a' : \varphi'$ leads to the expiration of the policy. Otherwise, a semantic query Q of the form $q(\vec{v}_{\varphi_y})$:- φ_y is created, where $\vec{v}_{\varphi_y}$ is the vector of variables in φ_y. Then, Δ'_o is queried with Q. Let the query return a result set rs; each result $r \in rs$ is a substitution such that $\Delta'_o \vdash \varphi_y \cdot r$. If $y \cdot r = a'$ for any such r, then a' is regulated by the policy. In this case, we can interpret the policy based on its modality as follows:

(1) $N_x = O$**:** In this case, the policy represents an obligation; that is, x is obliged to perform a'. Performing a' will remove this obligation.
(2) $N_x = P$**:** Performing a' is explicitly permitted.
(3) $N_x = F$**:** Performing a' is prohibited.

After examining the active policies as described above, we can identify a number of possible normative positions with respect to the action instance a': (i) doing a'

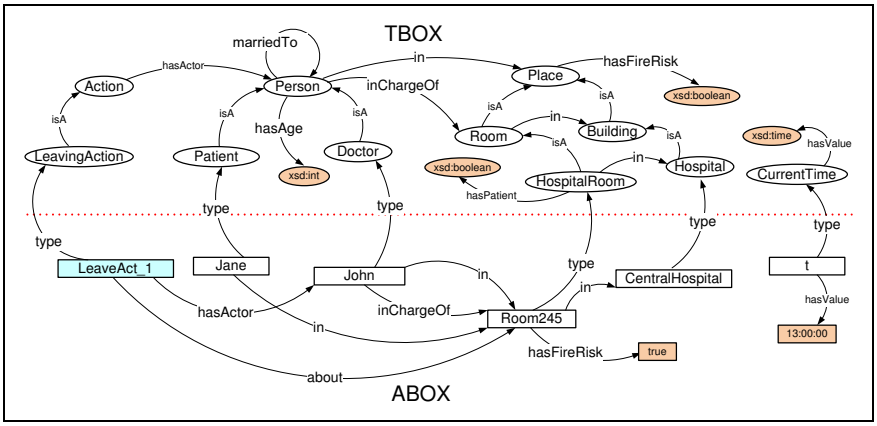

Fig. 3. The "sandbox" (hypothetical) state of the world.

may be explicitly permitted if there is a policy permitting it; (ii) doing a' may be obligatory if there exists a policy obliging it; (iii) doing a' may be prohibited if there is a policy prohibiting it; and (iv) there may be a conflict in the normative position with respect to a' if it is either both prohibited and explicitly permitted, or both prohibited and obliged.

4 Reasoning about Policies

In this section, we demonstrate reasoning techniques to support the analysis of policies in terms of their meaningfulness (Section 4.2) and possibility of conflict (Section 4.3), and hence address our third desideratum. Prior to this, however, we propose methods for reasoning about semantic formulas to underpin our mechanisms for policy analysis.

4.1 Reasoning about Semantic Formulas

Here, we introduce methods for reasoning about semantic conjunctive formulas using query freezing and constraint transformation.

4.1.1 Conjunctive Queries

There is an important relationship between conjunctive formulas and conjunctive queries that we exploit in this reasoning model. Conjunctive semantic formula can trivially be converted into a conjunctive semantic query. For example, $A^o_{\vec{v_1}}$ can be converted into the query $q_A()$:- $A^o_{\vec{v_1}}$. In this way, we can use query reasoning techniques to reason about semantic formulas. For instance, in order to reason about the subsumption between semantic formulas, we can use query subsumption (containment).

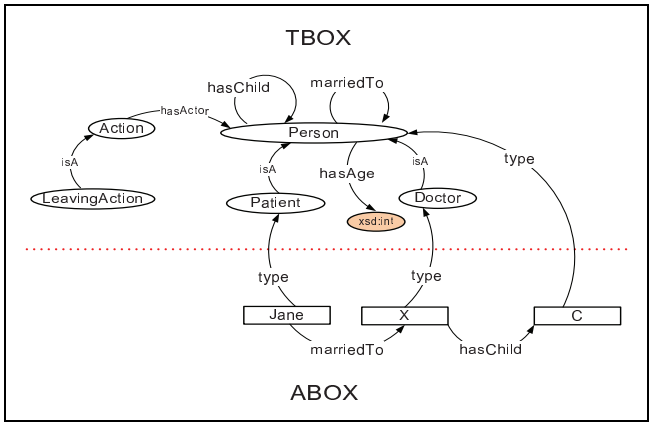

Fig. 4. The ontology created for q_B in Example 1.

In the conjunctive query literature, in order to test whether q_A subsumes q_B, the standard technique of *query freezing* is used to reduce the query containment problem to query answering in Description Logics [22,33]. For this purpose, we build a canonical ABox Φ_{q_B} from the query $q_B()$:- $B^o_{\vec{v_2}}$ in three steps. First, for each variable in $\vec{v_2}$, we put a fresh individual into Φ_{q_B} using the type assertions about the variable. Note that this individual should not exist in o. Second, we add each individual appearing in q_B into Φ_{q_B}. This is done using information about the individual from the $ABox_o$ (e.g., type assertions). Third, relationships between individuals and constants defined in q_B are inserted into Φ_{q_B}. As a result of this process, Φ_{q_B} contains a pattern that exists only in ontologies that satisfy q_B. We combine Φ_{q_B} and our $TBox_o$ to create a new canonical ontology, $o' = (TBox_o, \Phi_{q_B})$. Example 1 demonstrates a simple case. Based on [33,22], we conclude that $o \vdash q_B \sqsubseteq q_A$ if and only if o' entails q_A. In order to test whether o' entails q_A or not, we query o'. That is, o' entails q_A if there exists at least one match for q_A in o'. This can easily be achieved by converting q_A to SPARQL syntax and use Pellet's SPARQL-DL query engine to answer q_A on o' [30].

Example 1 Let query q_A be *q():- Person(?p)* $\wedge$ *marriedTo(?p,?x)* $\wedge$ *Patient(?x)* and query q_B be *q():- Doctor(?x)* $\wedge$ *marriedTo(?x,Jane)* $\wedge$ *hasChild(?x,?c)*. Then, Φ_{q_B} contains an individual x, which is created for the variable $?x$. The individual x is defined as of type $Doctor$. In Φ_{q_B}, we also have another individual $Jane$, which is defined in the original $ABox_o$ as an instance of the *Patient* class; we get all of its type assertions from the $ABox_o$. Then, we insert the object property *marriedTo* between the individuals x and $Jane$. Lastly, we create another individual c for the variable $?c$ in Φ_{q_B} and insert the $hasChild$ object property between x and c. The resulting ontology is shown in Figure 4.

The query freezing method described above enables us to create a canonical ABox for a semantic conjunctive formula; this ABox represents a pattern which only exists in ontologies satisfying the semantic formula. On the other hand, this method assumes that variables in queries can be assigned fresh individuals in a canonical

ABox. However, in OWL-DL, individuals can refer to objects, but not data values [32]. Therefore, the proposed query freezing method can be used to test for subsumption between q_A and q_B only if the variables in q_A and q_B refer to objects. A variable can refer to an object if it is used as the domain of an object or datatype property (e.g., *hasAge(?x,10)*) or if it is used as the range of an object property (e.g., *marriedTo(John,?x)*). Unfortunately, in many real-life settings, queries may have variables referring to data values with various constraints, which we refer to here as *datatype variables*. In these settings, the query freezing described above cannot be used to test subsumption. Example 2 illustrates a simple scenario.

Example 2 Let query q_A be *q():- Person(?p)* $\wedge$ *hasChild(?p,?c)* $\wedge$ *hasAge(?c,?y)* $\wedge$ *?y* $\geq$ *12* $\wedge$ *?y* $\leq$ *16* and query q_B be *q():- Doctor(?x)* $\wedge$ *marriedTo(?x,Jane)* $\wedge$ *hasChild(?x,?c)* $\wedge$ *hasAge(?c,?a)* $\wedge$ *?a* $\geq$ *10* $\wedge$ *?a* $\leq$ *20*. In this example, the query freezing method cannot be used directly to test subsumption between q_A and q_B, because the variables $?y$ and $?a$ refer to data values, which cannot be represented by individuals in an OWL-DL ontology.

4.1.2 Constraint Transformation

Here, we propose *constraint transformation*. It is a preprocessing step which enables us to create a canonical ABox for semantic formulas with datatype variables. Note that a datatype variable is used in a semantic formula to constrain one datatype property, e.g., $?y$ is used to constrain the $hasAge$ datatype property in q_A of Example 2. Constraint transformation in contrast uses *data-ranges* introduced in OWL 2.0 [15] to transform each constrained datatype property to a named OWL class. As a result, datatype variables and related datatype properties and constraints are replaced with type assertions. This procedure is detailed in Algorithm 1.

The algorithm takes a conjunctive semantic formula $F^o_{\vec{v}}$ and the ontology o as input (line 1). $F^o_{\vec{v}}$ is of the form $T^o_{\vec{v}} \cup R^o_{\vec{v}} \cup C^o_{\vec{v}}$, where $T^o_{\vec{v}}$, $R^o_{\vec{v}}$, and $C^o_{\vec{v}}$ are sets of type, relation and constraint assertions respectively. The output of the algorithm is the transformed semantic formula $F^\phi_{\vec{u}}$ (containing no datatype variables) and the updated ontology ϕ (line 2). Initially, $F^\phi_{\vec{u}}$ is set as equal to $T^o_{\vec{v}}$ and ϕ is the same as o (line 3). For each relation assertion $r(a, b)$ in $R^o_{\vec{v}}$, we do the following (line 4). First, we check if b is a datatype variable (line 5). If so, this means that r is a datatype property with a variable in its range. In this case, we extract the set of constraints related to b from $C^o_{\vec{v}}$, which is referred by γ_d (line 6). Based on r and γ_d, we create a concept c in $TBox_\phi$ using the *createConcept* function (line 7). This function works as follows:

(1) If $\gamma_d \neq \emptyset$, then b implies some restrictions on the range of r. In this case, c should refer to objects that have the property r with the restrictions defined in

Algorithm 1 Constraint transformation.

Input: Formula $F^o_{\vec{v}} \equiv T^o_{\vec{v}} \cup R^o_{\vec{v}} \cup C^o_{\vec{v}}$,
Ontology $o \equiv (ABox_o, TBox_o)$
Output: Formula $F^{\phi}_{\vec{u}}$,
Ontology $\phi \equiv (ABox_o, TBox_\phi)$
Initialization: $F^{\phi}_{\vec{u}} = T^o_{\vec{v}}$, $TBox_\phi = TBox_o$
for all $(r(a,b) \in R^o_{\vec{v}})$ **do**
  if $(isDatatypeVariable(b))$ **then**
    $\gamma_d = getConstraints(b, C^o_{\vec{v}})$
    $c = createConcept(r, \gamma_d, TBox_\phi)$
    $\tau = createTypeAssertion(a, c)$
    $F^{\phi}_{\vec{u}} = F^{\phi}_{\vec{u}} \cup \tau$
  else
    $F^{\phi}_{\vec{u}} = F^{\phi}_{\vec{u}} \cup r(a,b)$
    $\gamma_b = getConstraints(b, C^o_{\vec{v}})$
    if $(\gamma_b \neq \emptyset$ & $\neg(\gamma_b \subset F^{\phi}_{\vec{u}}))$ **then**
      $F^{\phi}_{\vec{u}} = F^{\phi}_{\vec{u}} \cup \gamma_b$
    end if
  end if
end for

```
<owl:Class rdf:about="#AgeConst1">
 <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasAge"/>
        <owl:allValuesFrom>
         <rdfs:Datatype>
            <owl:onDataRange rdf:resource="&xsd;nonNegativeInteger"/>
            <xsd:minInclusive rdf:datatype="&xsd;int">10</xsd:minInclusive>
            <xsd:maxExclusive rdf:datatype="&xsd;int">20</xsd:maxExclusive>
         </rdfs:Datatype>
        </owl:allValuesFrom>
    </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Fig. 5. A concept named **AgeConst1** is created for ***hasAge(?c, ?a)*** $\wedge$ ***?a*** $\geq$ ***10*** $\wedge$ ***?a*** $\leq$ ***20***.

γ_d on its range. While creating c in $TBox_\phi$, we use *data-ranges*[1] introduced in OWL 2.0 to restrict the range of r accordingly. For example, if $r(a, b)$ corresponds to $hasAge(?c, ?a)$ and γ_d = {*?a* $\geq$ *10, ?a* $\leq$ *20*}, then a concept named *AgeConst1* can be described as shown in Figure 5. For more sophisticated constraints, we create more complex class expressions using the OWL constructors *owl:unionOf*, *owl:intersectionOf*, and *owl:complementOf*.

(2) If $\gamma_d = \emptyset$, then b has no constraints, which means that the data-range of b is equivalent to the range of its datatype (i.e., for *xsd:int*, the range is min inclusive -2147483648 and max inclusive 2147483647).

After creating the concept c in $TBox_\phi$, we create a type assertion τ to declare a as an instance of c (e.g., *AgeConst1(?c)*) (line 8). This type assertion is added to $F^\phi_{\vec{u}}$ in order to substitute $r(a, b)$ and γ_d in $F^o_{\vec{v}}$ (line 9). On the other hand, if b is not a datatype variable (line 10), there are two possibilities: (1) r is a datatype property but b is not a variable, or (2) r is an object property. In both cases, we directly add $r(a, b)$ to $F^\phi_{\vec{u}}$ (line 11). If b has constraints defined in $C^o_{\vec{v}}$, we extract these constraints and add them to $F^\phi_{\vec{u}}$ if they are not already added (lines 12-15).

In order to test subsumption between q_A and q_B in Example 2, we should transform the bodies of these queries and update the ontology they are based on. For this purpose, we use constraint transformation twice. That is, we first update the ontology by adding the concept $AgeConst1$ to handle *hasAge(?c,?y)* $\wedge$ *?y* $\geq$ *10* $\wedge$ *?y* $\leq$ *20* and transform q_B to *q():- Doctor(?x)* $\wedge$ *marriedTo(?x,Jane)* $\wedge$ *hasChild(?x,?c)* $\wedge$ *AgeConst1(?c)*. Then, we add concept $AgeConst2$ to the ontology to handle *hasAge(?c,?y)* $\wedge$ *?y* $\geq$ *12* $\wedge$ *?y* $\leq$ *16* and transform q_A to *q():- Person(?p)* $\wedge$ *hasChild(?p,?c)* $\wedge$ *Age-*

[1] http://www.w3.org/TR/2008/WD-owl2-syntax-20081008/#Data_Ranges

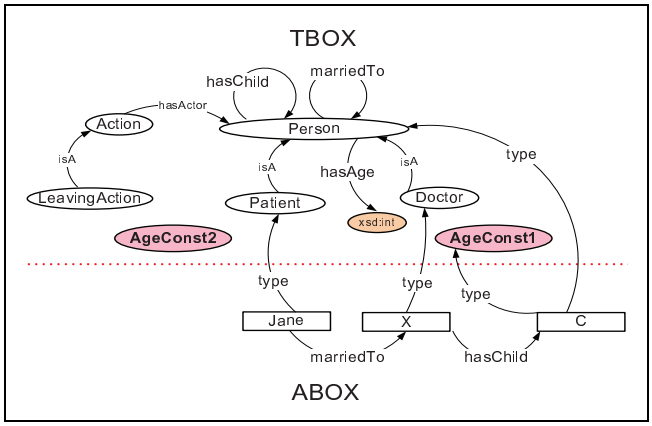

Fig. 6. The Ontology created for q_B in Example 2.

Const2(?c). After this preprocessing step, we use query freezing to test $q_B \sqsubseteq q_A$; the ontology with a canonical ABox created during query freezing is shown in Figure 6.

With these techniques in place, we are now in a position to address the issue of policy analysis supported by OWL-POLAR. It is descried in the following sections.

4.2 Idle Policies

A policy is *idle* if it is never activated or the policy's expiration condition is satisfied whenever the policy is activated. This condition is formally described in Definition 2. If a policy is idle, it cannot be used to regulate any action, because either it never activates or whenever it activates an obligation, permission, or prohibition about an action, the activated policy expires. While designing policies, we may take domain knowledge into account to avoid idle policies.

Definition 2 A policy $\alpha \longrightarrow N_{\chi:\rho}\,(a:\varphi)\,/e$ is an idle policy if it does not activate for any state of the world Δ_o or there is a substitution σ' such that $\Delta_o \vdash (e \cdot \sigma) \cdot \sigma'$, whenever there is a substitution σ such that $\Delta_o \vdash (\alpha \wedge \rho) \cdot \sigma$. ∎

Let us demonstrate idle policies with a simple example. Assume that object property *hasParent* is an inverse property of *hasChild*. Also, let us assume in the domain ontology, we have a SWRL rule such as $hasSponsor(?c, true) \leftarrow hasParent(?c, ?p) \wedge hasAge(?c, ?age) \wedge ?age < 18$, which means that children under 18 have a sponsor if they have a parent. Now, consider the policy in Table 2. This policy is activated when a person $?p$ has a child $?c$, which is a student under 18. The activated policy expires when $?c$ has a sponsor. Interestingly, whenever the policy is activated, the domain knowledge implies that $?c$ has a sponsor. That is, whenever the policy is activated, it expires.

Table 2
A simple idle policy example.

α	$hasChild(?p, ?c) \wedge Student(?c) \wedge hasAge(?c, ?age) \wedge ?age < 18$
N	O
$\chi : \rho$	$?p : Person(?p)$
$a : \varphi$	$?a : PayTuitionsOfStudent(?a) \wedge about(?a, ?c) \wedge hasActor(?a, ?p)$
e	$hasSponsor(?c, true)$

Table 3
A doctor cannot leave a room containing patients if he is in charge of the room.

α	$Room(?r) \wedge hasPatient(?r, true) \wedge inChargeOf(?d, ?r)$
N	F
$\chi : \rho$	$?d : Doctor(?d)$
$a : \varphi$	$?x : LeavingAction(?x) \wedge about(?x, ?r) \wedge hasActor(?x, ?d)$
e	$hasPatient(?r, false)$

In order to detect idle policies, we reason about the activation and expiration conditions of policies. Specifically, a policy $\alpha \longrightarrow N_{\chi:\rho}\,(a : \varphi)\,/e$ is an idle policy if $(\alpha \wedge \rho)$ is unrealistic or implies e using the knowledge in the domain ontology. More formally, we can show that the policy is idle if we show $(\alpha \wedge \rho)$ never holds or $(\alpha \wedge \rho) \rightarrow e$. This can be achieved as follows. First, we freeze $(\alpha \wedge \rho)$ and create a canonical ontology o'. If the resulting o' is not a consistent ontology, then we can conclude that the policy is an idle policy, because $(\alpha \wedge \rho)$ never holds. Let o' be consistent and σ be a substitution denoting the mapping of variables in $(\alpha \wedge \rho)$ to the fresh individuals in o'. If there exists a substitution σ' such that $o' \vdash (e \cdot \sigma) \cdot \sigma'$, we conclude that $(\alpha \wedge \rho) \rightarrow e$. We can test $o' \vdash (e \cdot \sigma) \cdot \sigma'$ by querying o' with $q() : - (e \cdot \sigma)$.

4.3 Anticipating Conflicts between Policies

In many settings, policies may conflict. In the simplest case, one policy may prohibit an action while another requires it. There are, however, many less obvious interactions between policies that may lead to logical conflicts [17,27,20,12]. Further developing our earlier example, consider the policy presented in Table 3 that states that a doctor cannot leave a room with patients if he is in charge of the room. This policy conflicts with the policy in Table 1 under some specific conditions. For example, in the scenario described Figure 1, *room 245* of Central Hospital has a fire risk and *Dr. John* is in charge of the room, in which there are some patients. In this setting, the policy in Table 1 obligates Dr. John to leave the room while the policy in Table 3 prohibits this action until the room has no patient.

If we can determine possible logical conflicts while designing policies, we can create better policies that are less likely to raise conflicts at run time. Furthermore, we can use various conflict resolution strategies such as setting a priority ordering be-

tween the policies to solve conflicts [19,34,35], once we determine that two policies may conflict.

In this section, we propose techniques to anticipate possible conflicts between policies at design time. Suppose we have two non-idle policies $P_i = \alpha^i \longrightarrow A_{\chi^i:\rho^i}\left(a^i : \varphi^i\right)/e^i$ and $P_j = \alpha^j \longrightarrow B_{\chi^j:\rho^j}\left(a^j : \varphi^j\right)/e^j$. These policies are active for the same policy addressee in the same state of the world Δ if the following requirements are satisfied:

(1) $\Delta \vdash (\alpha^i \wedge \rho^i) \cdot \sigma_i$, but no σ_i' such that $\Delta \vdash (e^i \cdot \sigma_i) \cdot \sigma_i'$
(2) $\Delta \vdash (\alpha^j \wedge \rho^j) \cdot \sigma_j$, but no σ_j' such that $\Delta \vdash (e^j \cdot \sigma_j) \cdot \sigma_j'$
(3) $\chi^i \cdot \sigma_i = \chi^j \cdot \sigma_j$

The policies P_i and P_j conflict if the following requirements are also satisfied:
(4) $(\varphi^i \cdot \sigma_i) \sqsubseteq (\varphi^j \cdot \sigma_j)$ or $(\varphi^j \cdot \sigma_j) \sqsubseteq (\varphi^i \cdot \sigma_i)$
(5) A conflicts with B. That is, $A \in \{P, O\}$ while $B \in \{F\}$ or vice versa.

Algorithm 2 Anticipate if P_i may conflict with P_j.

```
1:  Input:   Policy P_i = α^i ⟶ A_{χ^i:ρ^i}(a^i : φ^i)/e^i,
                Policy P_j = α^j ⟶ B_{χ^j:ρ^j}(a^j : φ^j)/e^j
2:  if ( (A ∈ {O,P} and B ∈ {F}) or (A ∈ {F} and B ∈ {O,P}) ) then
3:     ⟨Δ, σ_i⟩ = freeze(α^i ∧ ρ^i)
4:     ⟨o', _⟩  = freeze(φ^i · σ_i)
5:     rs       = query(o', φ^j)
6:     for all (σ_k ∈ rs) do
7:        ⟨Δ, σ_j⟩ = update(Δ, (α^j ∧ ρ^j) · σ_k)
8:        if (isConsistent(Δ)) then
9:           if (query(Δ, e^i · σ_i) = ∅ and query(Δ, (e^j · σ_k) · σ_j) = ∅) then
10:             return true
11:          end if
12:       end if
13:    end for
14: end if
15: return false
```

We can use Algorithm 2 to test if it is possible to have such a state of the world where P_i conflicts with P_j. The first step of the algorithm is to test if A conflicts with B (line 2). If they are conflicting, we continue with testing the other requirements. We create a canonical state of the world Δ in which P_i is active by freezing $(\alpha^i \wedge \rho^i)$ with a substitution σ_i mapping the variables in $(\alpha^i \wedge \rho^i)$ to the fresh individuals in Δ. Given that $(\varphi^j \cdot \sigma) \sqsubseteq \varphi^j$ for any substitution σ mapping variables into individuals, the requirement (iv) implies that $(\varphi^i \cdot \sigma_i) \sqsubseteq \varphi^j$. We test this as follows. First, we create a canonical ontology o' by freezing $(\varphi^i \cdot \sigma_i)$ (line 4) and then query o' with φ^j (line 5). Each answer to this query defines a substitution σ_k mapping variables in φ^j into the terms in $(\varphi^i \cdot \sigma_i)$, so that $(\varphi^i \cdot \sigma_i) \sqsubseteq (\varphi^j \cdot \sigma_k)$. If φ^j does not have any variable but it repeats in o' as a pattern, the result set contains only one empty substitution. If the query fails, the result set is an empty set ($\emptyset$),

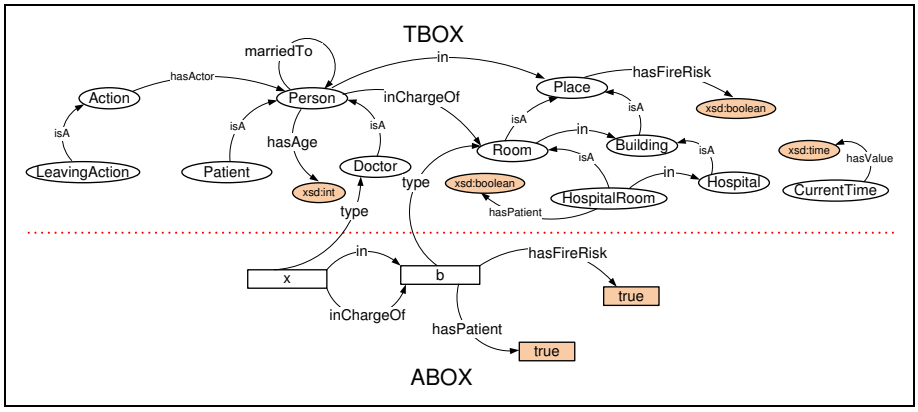

Fig. 7. The canonical state of the world where the policies of Table 1 and Table 3 conflict.

which means that it is not possible to have a σ_k such that $(\varphi^i \cdot \sigma_i) \sqsubseteq (\varphi^j \cdot \sigma_k)$.

For each σ_k satisfying $(\varphi^i \cdot \sigma_i) \sqsubseteq (\varphi^j \cdot \sigma_k)$, we test the other requirements as follows. First, we update Δ by freezing $(\alpha^j \wedge \rho^j) \cdot \sigma_k$ without removing any individual from its existing $ABox$ (line 7). Note that as a result of this process, σ_j is the substitution mapping the variables in $(\alpha^j \wedge \rho^j) \cdot \sigma_k$ to the new fresh individuals in the updated Δ, so that $\chi^i \cdot \sigma_i = (\chi^j \cdot \sigma_k) \cdot \sigma_j$. We test the consistency of the resulting state of the world Δ (line 8). If this is not consistent, we can conclude that it is not possible to have a state of the world satisfying the requirements. If the resulting Δ is consistent, we check the expiration conditions of the policies. If both are active in the resulting state of the world (line 9), the algorithm returns *true* (line 10). If any of these requirements do not hold, the algorithm returns *false* (line 15).

As described above, the algorithm transforms the problem of anticipating conflict between two policies into an ontology consistency checking problem. To check the consistency of the constructed canonical state of the world Δ, we have used the Pellet [30] reasoner. This reasoner adopts the *open world assumption* and does not have Unique Name Assumption (UNA). Hence, it searches for a model [2] of Δ, also considering the possible overlapping between the individuals (i.e., individuals referring the same object). If there is no model of Δ, it is not possible to have a state of the world satisfying the requirements stated above. We should also note that, while anticipating the conflict, Algorithm 2 tests only the case $(\varphi^i \cdot \sigma_i) \sqsubseteq (\varphi^j \cdot \sigma_j)$. However, we also need to test $(\varphi^j \cdot \sigma_j) \sqsubseteq (\varphi^i \cdot \sigma_i)$ to capture the possibility of conflict. Therefore, if the algorithm returns *false*, we should swap the policies and run the algorithm again. If it returns *true* with the swapped policies, we can conclude that there is a state of the world where these policies may conflict.

To demonstrate the algorithm, let us use the policies presented in Tables 1 and 3 and refer to them as P_i and P_j respectively. In this example, P_j is a prohibition while P_i is an obligation, so the algorithm proceeds as follows (line 2). We create a canonical state of the world Δ by freezing $Person(?x) \wedge Place(?b) \wedge$

[2] A model of an ontology o is an interpretation of o satisfying all of its axioms [1].

$hasFireRisk(?b, true) \wedge in(?x, ?b)$ with a substitution $\sigma_i = \{?x/x, ?b/b\}$ (line 3). Now we create a canonical ontology a' by freezing $\varphi^i \cdot \sigma_i$ with substitution $\{?a/a\}$ (line 4). This ontology has the following $ABox$ assertions: $LeavingAction(a)$, $about(a, b)$, $hasActor(a, x)$. We query o' with $LeavingAction(?x) \wedge about(?x, ?r) \wedge hasActor(?x, ?d)$ (line 5). The result set is composed of only one substitution: $\sigma_k = \{?x/a, ?r/b, ?d/x\}$. The next step is to update Δ by freezing $Doctor(x) \wedge Room(b) \wedge hasPatient(b, true) \wedge inChargeOf(x, b)$ without removing the current $ABox$ of Δ (line 7). The resulting canonical state of the world is shown in Figure 7. Lastly, we check whether both policies remain in effect by checking their expiration conditions (line 9). In this example, we query Δ with $hasFireRisk(b, false)$ and $hasPatient(b, false)$. Both of these queries return $\emptyset$, hence we conclude that there is a state of the world where these policies conflict (line 10).

5 Conflict Resolution

Legal theory and practice provide some doctrines to resolve conflict between norms [36,4]. These doctrines can also be used when a conflict arises between policies (aka norms). These doctrines are shortly described as follows:

- **Lex Superior**: The norm issued by a more important legal entity prevails when in conflict with another norm.
- **Lex Posterior**: The newer norm is preferred over the older one.
- **Lex Specialis**: The norm governing a specific subject matter overrides the norm which governs general matters.

Lex superior uses the hierarchy between the authorities issuing norms while resolving conflicts between these norms. Hence, it cannot be used to resolve conflicts between norms issued by one same authority or norms issued by authorities without hierarchical relationships between them. For instance, let us assume that the policy of Table 1 is issued by the government and the policy of Table 3 is issued by the hospital management. In this case, the policy of Table 1 would override the policy of Table 3, based on the *lex superior* principle.

Lex posterior assumes newer norms are preferred over the older ones. This assumption may hold only in certain conditions. In most of the cases, the lex posterior principle may not be useful. Especially if the conflicting norms are issues by different authorities, temporal relationships between these norms would be misleading. However, if these norms belong to the same authority, this principle may be applicable under some circumstances. For instance, let us assume that the policy of Table 3 is issued after the policy of Table 3 by the hospital management. Then, the policy of Table 3 would override the policy of Table 1.

Lex specialis considers a more specific norm as an exception of a more general one. This principle may work especially if these norms belong to the same organization. Unlike the other two doctrines, Lex specialis requires some non-trivial reasoning process that allow us to reason about the subsumption relationships between policies. In the next Section 5.1, we propose a subsumption reasoning mechanism for OWL-POLAR policies.

5.1 Policy Subsumption

Suppose we have two non-idle policies $P_i = \alpha^i \longrightarrow A_{\chi^i:\rho^i}\left(a^i : \varphi^i\right)/e^i$ and $P_j = \alpha^j \longrightarrow B_{\chi^j:\rho^j}\left(a^j : \varphi^j\right)/e^j$. The policy P_i subsumes P_j if whenever P_j is active for a policy addressee regarding an action, P_i is also active for the same policy addressee and action. This can be described formally as follows. For each state Δ such that $\Delta \vdash (\alpha^j \wedge \rho^j) \cdot \sigma_j$, there is a substitution σ_i such that

(1) $\Delta \vdash (\alpha^i \wedge \rho^i) \cdot \sigma_i$
(2) $\chi^i \cdot \sigma_i = \chi^j \cdot \sigma_j$
(3) $\varphi^j \cdot \sigma_j \sqsubseteq \varphi^i \cdot \sigma_i$
(4) there is no substitution σ_i' such that $\Delta \vdash (e^i \cdot \sigma_i) \cdot \sigma_i'$

Based on this definition, in this section, we propose Algorithm 3 to check if P_i subsumes P_j. The algorithm first creates a canonical state of the world characterizing all the states where P_j is active (line 2). Then, it checks if P_j is also active in this canonical state by finding all substitutions satisfying activation and role constraints of P_j (line 3). Lastly, it checks if any of these substitutions satisfies the requirements listed above (lines 4-8). The algorithm returns true if such a substitution exists (lines 5-7); otherwise it returns false (line 9).

Algorithm 3 Check if P_i subsumes P_j.

1: **Input:** Policy $P_i = \alpha^i \longrightarrow A_{\chi^i:\rho^i}\left(a^i : \varphi^i\right)/e^i$,
Policy $P_j = \alpha^j \longrightarrow B_{\chi^j:\rho^j}\left(a^j : \varphi^j\right)/e^j$
2: $\langle \Delta, \sigma_j \rangle = freeze(\alpha^j \wedge \rho^j)$
3: $rs = query(\Delta, \alpha^i \wedge \rho^i)$
4: **for all** $(\sigma_i \in rs)$ **do**
5: **if** $\chi^i \cdot \sigma_i = \chi^j \cdot \sigma_j$ **and** $\varphi^j \cdot \sigma_j \sqsubseteq \varphi^i \cdot \sigma_i$ **and** $query(\Delta, (e^i \cdot \sigma_i))$ fails **then**
6: return **true**
7: **end if**
8: **end for**
9: return **false**

If one of two conflicting policies subsumes the other, *Lex specialis* can be used to resolve the conflict. However, in many scenarios, there is no subsumption relationship between conflicting policies. These policies conflict only in certain cases, in which the activation conditions of both policies hold and their expiration conditions do not hold. To anticipate such policy conflicts, we have proposed the reasoning mechanisms presented in Section 4.3. These mechanisms try to construct a consistent state of the world satisfying the condition necessary for the conflict.

A policy addressee may wish to avoid a conflict between two specific policies by simply avoiding states of the world in which their activation conditions hold at the same time. This requires him to examine available courses of action, to avoid those actions that would trigger the activation conditions of conflicting norms. If the activation conditions of the policies concerned are in full control of the policy addressee, this may be feasible. In many cases, however, the agent may have limited control over the satisfaction of policy activation conditions. For instance, a doctor has limited ability to control the fire risk in a hospital. That is, he cannot control the activation of the policy in Table 1. On the other hand, he may control some of his responsibilities in the hospital. To avoid the possibility of conflict, he could avoid being in charge of any room in the hospital. This is a rather radical approach to conflict avoidance, and, of course, will lead to other conflicts with, for example, a duty of care. The doctor behaves as if there will indeed be a fire risk in the hospital, regardless of its actual probability. A more reasonable approach may be to consider the probability of specific world states occurring. That is, the doctor may avoid being in charge of any room in the hospital if the probability of fire risk in the hospital becomes greater than a threshold. However, this solution involves significant overhead of estimating likelihoods of policy activations.

Unless there is a subsumption relation between two conflicting policies, a conflict between them may occur only under certain circumstances. Hence, it may be more feasible to resolve a policy conflict once it occurs rather than trying to avoid it in the first place. This is especially the case when there are some predefined conflict resolution strategies available to resolve the policy conflict. For instance, if the policy of Table 1 is enforced by an authority with more power (or hierarchically superior) than that enforcing the policy of Table 3, then a conflict between these policies can be resolved using the *Lex superior* strategy. However, this would be disregarding the rationale behind the policy of Table 3, i.e., always having patients under the care of medical staff. A better approach would be to refine the expiration conditions of the policies concerned. If one of the two conflicting policies expires, the conflict would be automatically resolved without making one policy override the other. Hence, the problem of conflict resolution turns into the problem of satisfying policy expiration conditions once a conflict occurs between two policies.

A policy addressee may have certain actions to change the state of the world and achieve his goals (e.g., desired states of the world). Existing automated planning systems [23] can be used to find a plan (i.e., sequence of actions) to achieve a certain goal. These systems are given an initial state, a list of available actions (i.e., operators), and a goal state. Actions typically have preconditions and post-conditions. If the preconditions of an action hold in a specific state of the world, the action specification states that if it were performed in this state, the state is expected to change according to the postconditions. Given a state where two conflicting policies are activated at the same time for the same policy addressee, the policy addressee may use an automated planner to find a plan whose actions will cause the state of the world to change in a way that the expiration condition of one of the policies holds. The planner considers only actions available to the policy addressee while composing plans. Consider the previous example where a doctor is in charge of a room in a hospital H with fire risk, i.e., *hasFireRisk(H, true)*. In this situation, the policies of Tables 1 and 3, which are conflicting, will be active at the same time. The first policy expires if the goal state *hasFireRisk(H, false)* is achieved. However, a planner cannot produce a plan to achieve this state, since a doctor does not have actions available. On the other hand, the expiration conditions for the second policy is *hasPatient(R,false)*. This state of affairs can be achieved by the doctor if the doctor follows a plan for evacuating the patients in the room. Such a plan can easily be computed by an off-the-shelf planner [23]. Hence, in this example, it is possible to come up with a plan for the policy addressee to expire one of the conflicting active policies and resolve the conflict.

Algorithm 4 Obtain plans to expire active policies $N_x^1(\varphi_1)/e^1$ and $N_x^2(\varphi_2)/e^2$.

Input: $N_x^1(\varphi_1)/e^1$, $N_x^2(\varphi_2)/e^2$, $state$, $normstate$, A, $policies$
Output: R
$R = \emptyset$
$plans = \emptyset$
for all ($i = 1; i \leq 2; i$++) **do**
    $plans = plans \cup getPlansForGoal(e_i, state, A)$
end for
for all ($p \in plans$) **do**
    $s = clone(state)$
    $ns = clone(normstate)$
    for all ($a \in p$) **do**
        $applyActionToState(a, s)$
        $updateNormativeState(ns, s, policies)$
    end for
    $R = R \cup \{\langle p, ns \rangle\}$
end for
return R

Note that, there can be more than one plan to resolve a conflict. However, each plan may have a different cost for the policy addressee, so plans can be ranked based on their cost. Moreover, plans to be used to resolve a conflict between two poli-

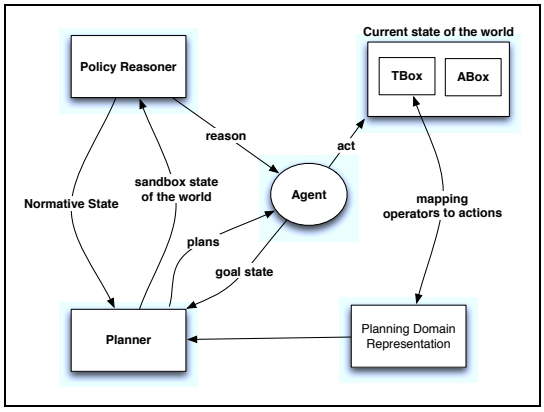

Fig. 8. Interaction between planner, policy reasoner, and the agent.

cies may contain actions whose execution violates some other policies. Therefore, before choosing and executing a plan among possibilities, it is important to estimate the outcomes of plans in terms of policy violations. Algorithm 4 formalises this procedure. The input of this algorithm is the conflicting active policies of the policy addressee x, as well as x's current *state* and *normative state*, actions available to x (A) and *policies* applicable to x. The normative state of x contains x's existing active policies (i.e., prohibitions, permissions, obligations) and policy violations (e.g., violated prohibitions). The algorithm firstly computes plans to satisfy expiration conditions of the active policies (lines 5-7). To estimate how the normative state of x would change if a specific plan p is followed, the algorithm creates copies of the current state of the world (i.e., sandbox state) and x's normative state (lines 9-10). Then, it updates the sandbox state by applying the actions in p and updates the normative state based on the sandbox state and policies of x (lines 11-14). As a result of the action, new active policies may be added to the normative state, some active policies in the normative state may become violated or expired, and some obligations may become satisfied. The algorithm associates each plan with the normative state it leads to (line 15) and lastly it returns all such associations as the output (line 17). Using Algorithm 4, a policy addressee not only finds plans to resolve conflicts but also gets informed about the outcomes of these plans in terms of policy activations, violations and fulfilments. Hence, he can review each plan based on this information and pick the one most suitable for himself if there is any.

Figure 8 illustrates the interactions between policy reasoner, planner, and an agent. Let us explain how these interactions take place through an example. Consider the state of the world in Figure 1 and policies of Tables 1 and 3. Then, *John* is obliged to leave the room 245 of Central hospital by the policy of Table 1 and he is prohibited to do so by the policy of Table 3 since the patient *Jane* is in the room. Let us assume that the planning domain has a simple operator *sendPatientTo* as described in Tables 4. Operators in AI planning correspond to atomic actions that can be performed to change state of the world. During planning, the planner tries to achieve the goal state by finding a sequence of actions with suitable parameters.

Operators are described using their precondition and post conditions (i.e., addition

Table 4
Description of *sendPatientTo* operator in a planning domain.

Operator	$sendPatientTo(?p, ?from, ?to)$
Precondition	$in(?p, ?from) \wedge patient(?p) \wedge differentFrom(?from, ?to)$
Deletions	$in(?p, ?from)$
Additions	$in(?p, ?to)$

Table 5
Additional TBox and ABox axioms

TBox	**ABox**
$sendPatientTo \sqsubseteq Action$	$in(room246, CentralHopital)$
$sendPatientTo \sqsubseteq \exists about.Patient$	$SafePlace(backyard)$
$sendPatientTo \sqsubseteq \exists from.Place$	
$sendPatientTo \sqsubseteq \exists to.Place$	
$SafePlace \sqsubseteq Place$	
$UnsafePlace \sqsubseteq \neg SafePlace$	
$\exists hasFireRisk.\{true\} \sqsubseteq UnsafePlace$	

and deletions). Hence, we need a language more complex than Description Logics to properly describe complex operators [23]. To combine planning with ontologies, researchers usually keep descriptions of operators in a planning domain file, but they map these operators to action classes within the ontology [29]. Table 5 describes *sendPatientTo* class added to $TBox$ to represent the operator described in Table 4. This class in $TBox$ is externally mapped to the *sendPatientTo* operator in the planning domain. Whenever the agent execute an instance of this operator, an instance of *sendPatientTo* class is added to the current state of the world, which is represented as an ontology.

To resolve the conflict between the policies of Tables 1 and 3, John interacts with the planner by requesting the plans to achieve the goal $hasPatient(room245, false)$. Given the additional ABox axioms in Table 5, the planners returns two plans: $\langle sendPatientTo(Jane, room245, room246)\rangle$ and $\langle sendPatientTo(Jane, room245, backyard)\rangle$. Both of these plans expire the policy of Table 3 for John. When John executes the first plan, an instance i of $sendPatientTo$ concept is created within the state of the world and described further using the triples $\langle i, hasActor, John\rangle$, $\langle i, about, Jane\rangle$, $\langle i, from, room245\rangle$, and $\langle i, to, room246\rangle$. In this example, the addition of this action instance does not lead to any policy violation for John. However, it would not be case if Central hospital has the policy of Table 6.

Let us describe how changes in normative state of John is determined, given Central hospital has the policy of Table 6. While selecting operations to add to the current plan, the planner creates a sandbox state of the world as described in [29]. That is, triples are removed from and added to the current state of the world based on operation descriptions. For instance, the planner starts with an empty plan and the state of the world in Figure 1. Then, $sendPatientTo(Jane, room245, room246)$ is added to the plan and the state of the world is changed by removing the triple $\langle Jane, in, room245\rangle$ and adding the triple $\langle Jane, in, room246\rangle$. Lastly, the planner

Table 6
A doctor is prohibited to send a patient to an unsafe place.

α	$Patient(?p) \wedge in(?p, ?from) \wedge UnsafePlace(?to)$
N	F
$\chi : \rho$	$?x : Doctor(?x)$
$a : \varphi$	$?a : SendPatientTo(?a) \wedge about(?a, ?p) \wedge hasActor(?a, ?x) \wedge from(?a, ?from) \wedge to(?a, ?to)$
e	$SafePlace(?to)$

sends the resulting sandbox state of the world to the policy reasoner for violation detection based on the methods described in Section 3.2. The policy reasoner reveals that the policy of Table 6 would be violated if this plan is executed. Therefore, the plan is annotated with this violation. However, the second plan does not lead to any violation, since backyard is defined as an instance of *SafePlace* in Table 5. Hence, for John, it would be more beneficial to follow the second plan to resolve the conflict between the policies of Tables 1 and 3. To quantitatively compare normative states, we may need to add penalties to policies [3]. The penalties determine the cost of policy violations and allows policy addresses to prefer violating one policy to another based on utility [35]. Penalties are not in the scope of this paper and this issue is set as a future work.

6 Complexity of reasoning mechanisms

The computational complexity of the methods and algorithms proposed in this paper can be summarised as follows. Policy activation described in Section 3.2 is based on testing activation and expiration conditions of the policy through conjunctive query answering in OWL-DL, which has been shown to be decidable under DL-safety restrictions [16]. Reasoning about actions is based on creating a sandbox state of the world from the description of an action, which is $O(n)$ in size of terms in action description, and testing activation and expiration conditions of policies. Hence, the complexity of this reasoning is equivalent to that of conjunctive query answering. Constraint transformation introduced in Section 4.1 has a complexity $O(n^2)$ in the size of terms in policy description.

Testing idle policies, policy subsumption, and anticipating conflicts require constraint transformation, query freezing, consistency checking, and query answering. Query freezing has complexity $O(n)$ in the size of terms in the policy's activation and role descriptions. Reasoning about idle policies, policy subsumption, and conflicts are decidable but not tractable, because of the complexities of consistency checking and query answering in OWL-DL [14].

We may note that although worst-case complexity of consistency checking and conjunctive query answering in OWL-DL is NEXPTIME-complete [1], there are sublanguages of OWL-DL such as DL-lite with a better complexity. For instance, rea-

soning services such as consistency and instance checking and conjunctive query answering in DL-lite have PTIME-complexity [14]. Furthermore, it has been shown that reasoning performance in most of the existing ontologies is much better than their worst-case complexities [30].

In this paper, planning is proposed as a tool to resolve conflicts under specific conditions. Complexity of planning significantly depends on the restrictions put on the planning domain representation [23]. Worst-case complexity of planning is PSPACE-complete in the set-theoretic representation [5]. Implementations such as GraphPlan [23] achieves significant speed-up and contributes the scalability of planning by backward constraint-directed search.

In summary, the reasoning mechanisms proposed in this paper are decidable, but if we use OWL-DL they are not tractable, in the worst case. If, however, we limit our language to DL-lite, we have decidability (although our expressiveness is curtailed).

7 Related Work and Discussion

We have proposed OWL-POLAR in [10] as an OWL-DL based policy language that supports decidable policy analysis. One key feature of OWL-POLAR is the reasoning mechanisms that allow anticipation of possible conflicts between policies. In this paper, we have extended [10] as follows. First, we have described how existing conflict resolution strategies can be used for OWL-POLAR policies. For this purpose, we have proposed a subsumption reasoning algorithm for policies, which allows us to determine if one policy is a specialization of another one. Then, we have discussed in which situations existing conflict resolution strategies may not be useful and proposed an automated planning-based approach to resolve policy conflicts under specific circumstances. In this work, we have also discussed the computational complexity of the proposed reasoning mechanisms and showed that these mechanisms are decidable.

There have been several policy languages proposed that are built upon Semantic Web technologies. Rei [18] is a policy language based on OWL-Lite and Prolog. It allows logic-like variables to be used while describing policies. This gives it the flexibility to specify relations like *role value maps* that are not directly possible in OWL. The use of these variables, however, makes DL reasoning services (e.g., static conflict detection between policies) unavailable for Rei policies. KAoS [34] is, probably, the most developed language for describing policies that are built upon OWL. KAoS was originally designed to use OWL-DL to define actions and policies. This, however, restricts the expressive power to DL and prevents KAoS from defining policies in which one element of an action's context depends on the value of another part of the current context. For example, KAoS cannot be

used to represent a policy like *two soldiers are allowed to communicate only if they are in the same team*. To handle such situations, KAoS has been enhanced with *role-value maps* using Stanford JTP, a general purpose theorem prover [34]. Unfortunately, subsumption reasoning is undecidable in the presence of arbitrary role-value-maps [1].

KAoS distinguishes between (positive and negative) obligation policies and (positive and negative) authorization policies. Authorization policies permit (positive) or forbid (negative) actions, whereas obligation policies require (positive) or do not require (negative) action. Thus the general types of policies that can be described are similar to those that we have discussed in this paper. Actions are also the object of a KAoS policy, and conditions on the application of policies can be described (context), although the subject (individual/role) of the policy is not explicit (it is, however, in Rei). In common with OWL-POLAR in its present form, KAoS does not capture the notion of the authority from which/whom a policy comes, but there is a notion of the priority of a policy which partially (although far from adequately) addresses this issue. Unlike OWL-POLAR, Rei and KAoS do not provide means to explicitly define expiration conditions of the policies.

Policy analysis within both KAoS and Rei is restricted to subsumption. A policy in KAoS is expressed as an OWL-DL class regulating an action, which is expressed as an OWL-DL class expression (e.g., using restrictions on properties such as *performedBy* and *hasDestination*). Two policies are regarded in conflict if their actions overlap (one subsumes another) while the modality of these policies conflict (e.g., negative vs. positive authorization). Similarly, if there exist two policies within Rei that overlap with respect to the agent and action concerned and they are obligued and prohibited, then a conflict is recognised. In such a situation, meta-policies are used to resolve the conflict. Policy conflicts can also be detected within the Ponder2 framework [31,38], where analysis is far more sophisticated than that developed for either KAoS or Rei, but analysis is restricted to design time. In general, different methods can be used to resolve conflicts between policies. This issue has been explored in detail elsewhere [19].

The expressiveness of OWL-POLAR is not restricted to DL. Using semantic conjunctive formulas, it allows variables to be used while defining policies. However, in semantic formulas, OWL-POLAR allows only object variables to be compared using *owl:sameAs* and *owl:differentFrom* properties. On the other hand, datatype variables can be used to define constraints on the datatype properties. In other words, semantic formulas are restricted to describe states of the world, each of which can be represented as an OWL-DL ontology. Therefore, when a semantic formula is frozen, the result is a canonical OWL-DL ontology. OWL-POLAR converts problems of reasoning with and about policies into query answering and ontology consistency checking problems. Then, it uses an off-the-shelf reasoner such as Pellet [30] to solve these problems. It is known that consistency checking in OWL-DL is decidable [30], and query answering in OWL-DL has also been shown

to be decidable under DL-safety restrictions [16].

Ontology languages like KAoS are built on OWL 1.0, which does not support data-ranges. Therefore, while defining policies, they either do not allow complex constraints to be defined on datatype properties or use non-standard representations for these constraints, which prevents them from using the off-the-shelf reasoning technologies. The clear distinctions between OWL-POLAR and KAoS, however, are manifest in the fact that data ranges are exploited in OWL-POLAR to enable the expression of more complex constraints on policies, and the sophistication of the reasoning mechansims described in this paper.

To the best of our knowledge, OWL-POLAR is the first policy framework that formally defines and detects idle policies. Existing approaches like KAoS and Rei analyse policies only to detect some type of conflict, considering only subsumption between policies. On the other hand, OWL-POLAR provides advanced policy analysis support that is not limited to subsumption checking. Consider the following policies: (i) *Dogs are prohibited from entering to a restaurant*, and (ii) *A member of CSI team is permitted to enter a crime scene*. There is no subsumption relationship between these policies, and so KAoS and Rei could not detect a conflict. However, OWL-POLAR anticipates a conflict by composing a state of the world where these policies are in conflict, e.g., the crime scene is a restaurant and there is a dog in the CSI team.

Deontic logics study representation and relationships among formal constructs asserting that certain actions or states of affairs are obligatory, permitted, or forbidden [21]. Standard Deontic Logic (SDL) builds upon propositional logic and is the most studied system of deontic logic. Especially, SDL is traditionally used to analyse and identify ambiguities in sets of legal rules. For instance, Sergot *at al.* represented aspects of the British nationality act using SDL [28]. Cholvy and Cuppens used SDL to represent security policies to detect conflicts in policy specification [8]. Their approach is based on translating SDL into first order predicate logic to perform the necessary conflict detection and analysis. SDL has been criticized as having some inherent paradoxes [21]. For instance, one axiom of SDL implies that an obligation can imply a permission while another axiom indicates that a permission implies no obligation. Extensions of deontic logic have been proposed to handle these paradoxes [25]. However, even with these extensions, deontic logics, as a formalism, pose challenges on humans who experience difficulties creating and understanding their (and others') policies [11]. On the other hand, in this paper, we have proposed an expressive policy specification language based on Semantic Web, which builds upon W3C standards and clear semantics both for humans and machines.

In order to achieve decidability and improve reasoning performance, we have imposed some restrictions in OWL-POLAR. First, we have limited representation of constraints in conjunctive formulas so that two datatype variables cannot be com-

pared. This allows us to convert these constraints into DL class expressions using data-ranges. Without this restriction, a policy language would be undecidable [1], since such constraints could be used to implement role-value maps. Therefore, policies requiring comparisons of datatype variables cannot be expressed using OWL-POLAR, for the sake of decidability. Second, we have only considered conjunctive semantic formulas while representing activation conditions of policies, because inclusion of disjunctions in these formulas may lead to more than one canonical states of the world during policy analysis. Disjunctions in activation conditions do not affect the decidability of OWL-POLAR, since a policy containing disjunctions in its activation conditions can be converted into a set of policies with only conjunctions in their activation conditions. However, we stick to conjunctive semantic formulas in this paper for clarity and simplicity. Another limitation of OWL-POLAR is its *monotonicity* during conflict analysis. To anticipate conflicts between two policies, OWL-POLAR creates a canonical state of the world, where these two policies are active at the same time. Then, standard OWL-DL consistency checking is used to test the possibility of such state. However, standard reasoning in OWL-DL is monotonic [1]. That is, if we know that x is an instance of A, then adding more information to the model cannot cause this to become *false*. Therefore, currently conflict detection between policies cannot be done using OWL-POLAR when it requires non-monotonic reasoning. However, this limitation would be relieved when non-monotonic reasoning mechanisms becomes available in standard OWL-DL reasoners [6].

8 Conclusions

Policies provide useful abstractions to constrain and control the behaviour of components in loosely coupled distributed systems. Policies, also called norms, help designers of large-scale, open, and heterogenous distributed systems (including multi-agent systems) to specify, in a concise fashion, acceptable (or policy-compliant) global and individual computational behaviours, thus providing guarantees for the system as a whole. In this paper, we have presented a semantically-rich representation for policies as well as efficient mechanisms to reason with/about them. OWL-POLAR meets all the essential requirements of policies, as well as achieving an effective balance between expressiveness (realistic policies can be adequately represented) and computational complexity of associated reasoning for decision-making and analysis (reasoning with and about policies operate in feasible time).

The mechanisms proposed in this paper allow policy authors to detect inconsistent or unfounded policies. These mechanisms provide means to determine the context in which different policies may conflict. This kind of reasoning with context allows policy authors and agents to resolve the conflict before they occur. We have describe how conflict resolution doctrines from legal theory and practice could be used to resolve policy conflicts. We have proposed a policy subsumption algorithm to allow

Lex Specialis to be used as a conflict resolution strategy for OWL-POLAR policies. Furthermore, first time in the literature, we have used AI planning for policy conflict resolution. Lastly, we have showed that all these mechanisms are decidable.

Building upon this research, we plan to explore various extensions to OWL-POLAR. We will explore extending the representation of policies to include deadlines and penalties associated with their violation, along the lines of [3]. Also, we would like to investigate how policing mechanisms [24] could make use of our representation and associated mechanisms to foster welfare in societies of self-interested components/agents. Two further extensions should address policies over many actions (as in, for instance, "ξ is obliged to perform φ_1 and φ_2") and disjunctions (as in, for instance, "ξ is obliged to perform φ_1 or φ_2"). Finally, we are exploring the use of OWL-POLAR in support of human decision-making, including joint planning activities in hybrid human-software agent teams.

References

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (eds.), The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, 2003.

[2] A. Beautement, D. Pym, Structured systems economics for security management, in: Proceedings of the Ninth Workshop on the Economics of Information Security, Harvard, USA, 2010.

[3] G. Boella, J. Broersen, L. Torre, Reasoning about constitutive norms, counts-as conditionals, institutions, deadlines and violations, in: PRIMA '08: Proceedings of the 11th Pacific Rim International Conference on Multi-Agents, Springer-Verlag, Berlin, Heidelberg, 2008.

[4] A. Boer, T. van Engers, R. Winkels, Mixing legal and non-legal norms, in: Proceeding of the 2005 conference on Legal Knowledge and Information Systems: JURIX 2005: The Eighteenth Annual Conference, 2005.

[5] T. Bylander, Complexity results for planning, in: Proceedings of the 12th international joint conference on Artificial intelligence (IJCAI'91), 1991.

[6] G. Casini, U. Straccia, Defeasible inheritance-based description logics, in: IJCAI, 2011.

[7] C. Castelfranchi, Modelling social action for AI agents, Artificial Intelligence 103 (1998) 157–182.

[8] L. Cholvy, F. Cuppens, Analyzing consistency of security policies, in: Proceedings of the 1997 IEEE Symposium on Security and Privacy, SP '97, IEEE Computer Society, Washington, DC, USA, 1997.

[9] C. Chopinaud, A. E. Fallah-seghrouchni, P. Taillibert, Prevention of harmful behaviors within cognitive and autonomous agents, in: Proceedings of the European Conference on Artificial Intelligence, 2006.

[10] M. Şensoy, T. J. Norman, W. W. Vasconcelos, K. Sycara, OWL-POLAR: Semantic policies for agent reasoning, in: Proceedings of the 9th International Semantic Web Conference (ISWC'10), 2010.

[11] N. Damianou, A. K. Bandara, M. Sloman, E. C. Lup, A survey of policy specification approaches, Tech. rep., Department of Computing, Imperial College (2002).
URL `http://www.doc.ic.ac.uk/ mss/Papers/PolicySurvey.pdf`

[12] A. Elhag, J. Breuker, P. Brouwer, On the formal analysis of normative conflicts, Information & Communications Technology Law 9 (3) (2000) 207–217.

[13] A. García-Camino, J. A. Rodríguez-Aguilar, C. Sierra, W. Vasconcelos, Constraint rule-based programming of norms for electronic institutions, Autonomous Agents and Multi-Agent Systems 18 (1) (2009) 186–217.

[14] B. C. Grau, OWL 2 web ontology language tractable fragments, http://www.w3.org/2007/OWL/wiki/Tractable_Fragments (December 2007).

[15] W. O. W. Group, OWL 2 web ontology language: Document overview, http://www.w3.org/TR/owl2-overview (October 2009).

[16] P. Haase, B. Motik, A mapping system for the integration of owl-dl ontologies, in: IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems, ACM, New York, NY, USA, 2005.

[17] H. Hill, A functional taxonomy of normative conflict, Law and Philosophy 6 (2) (1987) 227–247.

[18] L. Kagal, T. Finin, A. Joshi, A policy language for a pervasive computing environment, in: POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, 2003.

[19] M. J. Kollingbaum, T. J. Norman, Norm adoption and consistency in the NoA agent architecture, Lecture Notes in Artificial Intelligence 3067 (2004) 169–186.

[20] E. Lupu, M. Sloman, Conflicts in policy-based distributed systems management, IEEE Transactions on software engineering 25 (6) (1999) 852–869.

[21] J.-J. C. Meyer, R. J. Wieringa (eds.), Deontic logic in computer science: normative system specification, John Wiley and Sons Ltd., Chichester, UK, 1993.

[22] B. Motik, Reasoning in description logics using resolution and deductive databases, Ph.D. thesis, Universitt Karlsruhe (TH), Karlsruhe, Germany (January 2006).

[23] D. Nau, M. Ghallab, P. Traverso, Automated Planning: Theory & Practice, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[24] J. Patel *et. al*, Agent-based virtual organisations for the grid, Int. Journal of Multi-Agent and Grid Systems 1 (4) (2005) 237–249.

[25] H. Prakken, M. Sergot, Dyadic deontic logic and contrary-to-duty obligations., 1997, pp. 223–262.

[26] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, Tech. rep., W3C, http://www.w3.org/TR/rdf-sparql-query/ (2006).

[27] G. Sartor, Normative conflicts in legal reasoning, Artificial Intelligence and Law 1 (2) (1992) 209–235.

[28] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, H. T. Cory, The british nationality act as a logic program, Commun. ACM 29 (1986) 370–386.

[29] E. Sirin, Combining description logic reasoning with AI planning for composition of web services, Ph.D. thesis, College Park, MD, USA (2006).

[30] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Web Semant. 5 (2) (2007) 51–53.

[31] M. Sloman, E. Lupu, Policy specification for programmable networks, in: IWAN '99: Proceedings of the First International Working Conference on Active Networks, Springer-Verlag, London, UK, 1999.

[32] M. K. Smith, C. Welty, D. L. McGuinness, OWL: Web ontology language guide, http://www.w3.org/TR/owl-guide (February 2004).

[33] J. D. Ullman, Information integration using logical views, Theoretical Computer Science 239 (2) (2000) 189–210.

[34] A. Uszok, J. M. Bradshaw, J. Lott, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, H. Jung, New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS, in: POLICY '08: Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks, 2008.

[35] W. W. Vasconcelos, M. J. Kollingbaum, T. J. Norman, Normative conflict resolution in multi-agent systems, Autonomous Agents and Multi-Agent Systems 19 (2) (2009) 124–152.

[36] E. Vranes, The definition of norm conflict in international law and legal theory, European Journal of International Law 17 (2) (2006) 395–418.

[37] M. Woolridge, M. J. Wooldridge, Introduction to Multiagent Systems, John Wiley & Sons, Inc., New York, NY, USA, 2001.

[38] H. Zhao, J. Lobo, S. M. Bellovin, An algebra for integration and analysis of ponder2 policies, in: POLICY '08: Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks, IEEE Computer Society, Washington, DC, USA, 2008.