A Description Logic based Approach on Handling Inter-Domain Policy Conflicts using Meta-Policies

Julian Schütte, Tobias Wahl, *Fraunhofer SIT, Germany*

***Abstract*—In upcoming ambient intelligence applications, services provided by heterogeneous and often mobile platforms are combined to build "intelligent" environments. As these services are hosted within different administrative domains, equally respecting security policies of all services becomes a challenge and conflicts between the policies of different domains can occur. In this paper we propose an approach to resolve conflicts between policies of different administrative domains at runtime by means of meta-policies. The proposed meta-policy model allows policy administrators to define certain guarantees which must not be overwritten in any case, thereby acting as invariant security properties. While decisions characterized as *strict* according to the meta-policy are guaranteed to be enforced, decisions classified as *defeasible* may be overwritten by policy decisions created by other domains. We present a use case example illustrating how the approach provides a resolution of cross-domain conflicts, describe the underlying policy model based on description logics, explain how access requests are decided and inter-domain conflicts handled and discuss a proof-of-concept implementation of our approach.**

I. INTRODUCTION

With the advent ambient intelligence (AmI) environments and powerful mobile devices like smart-phones, more and more services are not only hosted by web servers that are placed in a fixed network infrastructure but also by mobile embedded platforms. As these services roam across different networks and are provided in different administrative domains, defining access control policies for them becomes a critical issue. Developers of traditional web services make use of access control policy languages like XACML [1] in order to specify who is allowed to access the service under certain conditions. However, the assumption of XACML as well as many other access control languages is that a service is always under control of a single administrative domain. While this assumption is feasible for fixed services, it does not fit the needs of mobile services which are potentially used in different environments, each comprising its own policy domain. If developers combine services from different administrative domains in order to create value-added services, the value-added service needs to abide by the security policies of each of the combined domains. Also, when mobile devices roam across different environments they become subject to multiple policy domains at the same time. In both cases, the security policies of multiple domains must be applied to a service. As the policies of different domains might contradict each other, mechanisms for recognizing and resolving such conflicts are critical. Simply preferring the policies of one of the domains is however not a satisfying solution. In that case, the policies of other domains would inadvertently be overridden and as a result the effective security properties of a system would be unclear to developers and users. So, while on the one hand detecting and solving inter-domain policy conflicts is a necessity, developers need to be sure of guaranteed security properties of their system which must not be overridden at a later time.

Another drawback of most current access control policy languages is that they rely on predefined identifiers for subjects and resources. Mobile services are however discovered and selected in an ad-hoc fashion and many AmI middleware systems make use of semantic service descriptions for that purpose. It becomes therefore difficult for policy administrators to specify policies at design time without knowing the actual entity to which the policies shall be applied later at run time. In order to overcome this limitation, several authors have proposed an integration of semantic knowledge and policies – an approach we deem as undoubtedly sensible and considered in our work.

Summarizing, in order to deal with the requirements of mobile services and overlapping policy domains, access control policies need to be extended by semantic descriptions and the ability to deal with inter-domain conflicts while preserving guaranteed security properties as wished by the developer. In this paper we therefore propose a model of access control policies on the basis of Description Logics (DL) which are the basis of knowledge representations in semantic web technologies (SWT), e.g. in ontologies. In order to cope with policy decisions from multiple domains and the resulting possible conflicts, we add the concept of meta-policies allowing administrators to specify policy invariants which are guaranteed to be enforced even in the case of conflicts. On the basis of this model we describe how a Policy Decision Point (PDP) can answer access requests using (decidable) extensions of DL. As a result of this policy decision process, a Policy Enforcement Point (PEP) is informed about the actual effect which has to be enforced as well as a classification of this effect, either as *strict* or *defeasible*, determining whether the decision must be enforced or may be overwritten in the case of conflicts. In case a conflict cannot be resolved due to several conflicting strict decisions, a compensating action can be specified – usually instructing the service to leave the conflicting policy domain or to simply log the conflict.

In the following section we review work related to ours and then present an Ambient Intelligence use case motivating our approach in section III. In section IV we describe how we modeled access control policies using description logics and introduce the concept of meta-policies for defining invariable policy decisions. Section V explains how access requests are decided on the basis of description logics and how conflicts between policy domains are

detected and handled. The practicability of our approach has been tested by means of a proof-of-concept prototype whose design is presented in section VI and section VII concludes the paper.

II. Related Work

Work related to ours is on the one hand concerned with the integration of semantic web technology (SWT) into policies and on the other hand with detecting and resolving conflicts by means of meta-policies:

The need for considering semantic information in policies has been recognized by various authors and a number of approaches have been proposed – ranging from optional semantic extensions of existing policy languages to completely semantic-based policy frameworks:

In [14] an approach of using semantic attributes in XACML is described. The authors propose using SWRL rules to infer implicit information like "is full age" from explicit facts provided in an access request like "age>18". The authors of [12] aim at the same goal but use RDF triplets to describe attributes in XACML. Both approaches work on RDF triplets and are thus not able to deal with advanced concepts like cardinalities which are supported by OWL. In [15], the combination of XACML and OWL is used to realize a role-based access control model. Although the approach of using class expressions for describing policy subjects and resources is promising, the suggestion of using XACML's obligation element to add inferred axioms to the knowledge base during the policy decision process appears a bit intricate.

Further, in [10] individuals from an ontology can be used in a WS-POLICY definition and are compared against the information contained in an access request by predefined custom *comparison operators* like "less than" or "is subclass of". From these approaches we adopt the idea of integrating semantic attributes into policies, yet our approach of using class expressions overcomes the complexity of SWRL rules and the limitations of RDF triplets, does not rely on custom operators and also does not require modifications of the knowledge base during the decision of an access request.

Other authors have worked towards realizing complete policy frameworks on the basis of description logics. In [2] Kolovski presents a formalization of XACML in so-called "defeasible description logic" (DDL^-) which he proposes to use for analysis purposes. Although the policy structure we use is similar to the one proposed in [2], our formalization considers ordering of rules and supports more rule-combining algorithms. Also, we our goal was to put the formal policy model into practice using standard semantic web technologies.

Rein [16] [11] is a distributed policy framework which defines policies using the Notation3[1] language and ontologies expressed in OWL. A related approach is followed by the policy framework of the KAoS [21] project where DAML, the predecessor of OWL, is used to define policy ontologies and builds the basis for policy analysis and –decision services. However, these frameworks do not deal with conflicts between the on pervasive systems policies of different administrative domains.

The policy framework Ponder [9] does not consider semantics but organizes policy domains hierarchically and resolves conflicts between their policies by either giving precedence to the rules of the most specific domain or by defining a default decision which must be applied to all child domains of the root domain [20]. This approach is however not suited for resolving conflicts between policies which have been stated by different authorities, as it implicitly allows one domain to override the policies of another one. Different types of policy conflicts and strategies on how to detect them either statically (at design time) or dynamically (at run time) have been discussed in [19], yet without providing methods for resolving such conflicts.

The idea of meta-policies has been discussed in depth in [7] and [8]. This idea has been put into practice to a limited extend by the abovementioned domain-wide default policy decision in the Ponder framework. In [18] meta-policies have been applied for resolving conflicts on a set of policies, in order to enforce constraints such as *separation of duty*. Yet, none of these approaches provides a solution for resolving conflicts between different administrative policy domains which is the focus of this paper.

III. Scenario

In this section we introduce a brief use case example to motivate the problem addressed in this paper and to illustrate our approach on solving it. The problem of conflicting policy domains can occur in many different scenarios, for example when creating value-added services which combine services from multiple administrative domains in an enterprise SOA. As in our work we put the focus on services provided by mobile devices in pervasive systems we have chosen a more future-oriented use case from this kind of application domain. Yet, the solution as discussed in the rest of this paper is not limited to this scenario.

We assume Alice owns a smart-phone which she uses in different environments, for example in an environment denoted as *@home* for private purposes and in environment @work for business tasks. These environments are not necessarily bound to a physical location but rather depend on the purpose of the current task and may thus overlap, i.e. be both active at the same time. Depending on the environment in which it is used, the device has to comply with different access control policies: while Alice uses the device in environment @home, she allows access to different services provided by the device: for everybody in the local network, access to an AdminService for controlling the settings of the phone should be granted. Further, a Camera service and LocationTracker service should be accessible for Alice's family members. So, a simplified policy for the @home environment could look as follows:

$$
\begin{aligned}
&@home: \\
&subject.isLocatedIn(localNetwork) \\
&\wedge\, resource.isInstanceOf(AdminService) \rightarrow allow \\
&subject.isFamilyMemberOf(Alice) \\
&\wedge\, resource.isInstanceOf(Camera) \rightarrow allow \\
&subject.isFamilyMemberOf(Alice) \\
&\wedge\, resource.isInstanceOf(LocationTracker) \rightarrow allow
\end{aligned}
$$

[1] http://www.w3.org/DesignIssues/Notation3

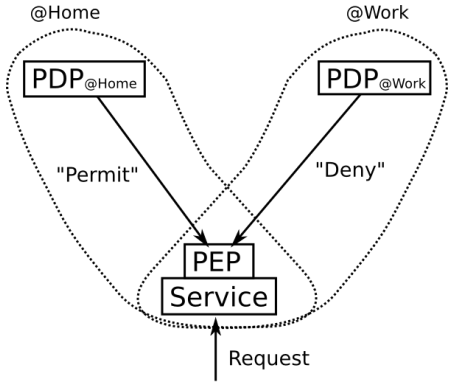

Figure 1 Access request to a service results in conflict between policy domains.

On the other hand, if the smart-phone is used in the @work environment it has to deal with potentially sensible information, so the administrator of Alice' company has set the following policies: access to all Camera- and LocationTracker services has to be denied and it is required that at least an administrator of the company has access to the phone's AdminService in order to be able to apply further configurations. The @work policy could thus look as follows:

$$
\begin{aligned}
&@work: \\
&resource.isInstanceOf(Camera) \rightarrow deny \\
&resource.isInstanceOf(LocationTracker) \rightarrow deny \\
&subject.hasRole(Administrator) \\
&\wedge resource.isInstanceOf(AdminService) \rightarrow allow \\
&!\, subject.hasRole(Administrator) \\
&\wedge resource.isInstanceOf(AdminService) \rightarrow deny
\end{aligned}
$$

In case both environments are active and thus policies of both domains shall become effective, conflicts between the policies of @home and @work can occur: for example, if a family member of Alice wants to locate her using the LocationTracker, the @home policy would allow this request while the @work policy would refuse it (c.f. Figure 1) - the same applies if Alice would be trying to use the camera service. Also, if an administrator from Alice' company would request access to her phone's AdminService – a legitimate request according to the @work policy, the @home policy would be indifferent and could refuse the request, depending on the default decision.

These types of cross-domain policy conflicts are likely to occur more often as mobile and multi-purpose devices increasingly provide services to their environment, as it is the vision of Ambient Intelligence. A simple approach to this problem would be to require environments to be disjunct so that only one policy domain is applicable at a time. However, this contradicts the idea of having multi-purpose devices roaming across different environments. Another approach which is also followed by policy frameworks like Ponder2[2] is to hierarchically structure policy domains and then either prefer the most specific or the most generic policy. Yet, a significant drawback of these approaches is that they allow policy domains to override rules set by other domains without any limitations. As a result, policy administrators would not be able to tell if and when their policies might become overwritten by those of a different domain and thus - in contrast to a single-domain scenario – policies could not be regarded as "guaranteed security properties" of a system anymore.

Instead of structuring policy domains and hardcoding the resolution of conflicts based on that structure, our approach focuses on using meta-policies in order to allow administrators to define certain invariants which must not be overwritten by different policy domains. Referring to our example, we could assume that Alice has defined the following meta-policy in order to avoid that she gets locked out from her own phone:

$$
\begin{aligned}
&@home: \\
&subject.is(Alice) \\
&\wedge resource.isInstanceOf(AdminService) \rightarrow allow
\end{aligned}
$$

This meta-policy is much more specific than the original policy and only requires that the AdminService stays accessible for Alice herself. As the meta-policy works as an invariant, it cannot be overwritten by any other policy domain. Also the company's administrator could specify a meta-policy to ensure access to the AdminService and to restrict the usage of Camera and LocationTracking service under all circumstances:

$$
\begin{aligned}
&@work: \\
&resource.isInstanceOf(Camera) \rightarrow deny \\
&resource.isInstanceOf(Camera) \rightarrow deny \\
&subject.hasRole(Administrator) \\
&\wedge resource.isInstanceOf(AdminService) \rightarrow allow
\end{aligned}
$$

By adding such meta policies, conflicts between the two policy domains can now be handled: access requests to the AdminService are granted to administrators of the company and to Alice herself (fulfilling the guarantees set by the meta policies), but not to family members of Alice (the @home policy's *grant* decision is overwritten by the more restrictive @work policy here). The Camera service and the LocationTracker are blocked for everyone, as required by the @work meta-policy.

In the following sections, we will describe the how the necessary components for realizing such a scenario can be designed: the structure of policies and meta-policies, the policy decision process and a possible implementation based on semantic web technology.

IV. Policy and Meta-Policy Model

We use Description Logics to describe a formal structure of policies and meta-policies. This way, we facilitate an integration of policies with external knowledge bases in the form of ontologies and thereby separate domain knowledge from the rules which reflect the security model of an application. Domain knowledge describes for instance properties of authorization methods, known vulnerabilities or the strength of cryptographic protocols. This domain knowledge may change over time even if the security requirements of an application are the same as new cryptographic mechanisms become available or weaknesses in existing ones become public. Integrating this domain knowledge into policy descriptions is desirable as it allows administrators specify policies at a higher, more

[2] http://www.ponder2.net

understandable level and to access all semantic service descriptions that are already available in the application. The policy structure we introduce in this paper is basically derived from XACML [1] and aligns in parts with the formalization provided by Kolovski [2]. Some details of the XACML specification have been abstracted away for clarity and meta-policies for defining policy invariants have been added. The notation used in the following subsections is based on the Description Logic terminology as described in [3].

A. Description Logics

At first, we provide a brief overview of the terminology and the main concepts of description logics (DL) which we use in the rest of this paper. DL comprises logic languages which are subsets of first order logic and which have mainly been designed for knowledge representations. The term "description logics" does not refer to a single dedicated logic language but rather to a family of logics which follow the same formalism but show different levels of expressivity. The two most important differences between first order logic and DL are decidability and the open-world-assumption (OWA): in contrast to first order logic, all description logics are decidable which makes them attractive to be used in the context of policy decisions. Indeed, although they are decidable, most reasoning problems show at least EXPTIME complexity but practice has shown that they can be efficiently solved in knowledge bases of reasonable size. While first order logic considers facts which are not contained in the model as non-existent – i.e. it assumes a closed world – DL is based on the open world assumption. It assumes the model to be incomplete and thus does not have any knowledge about facts which are not contained in the model. As a result of the OWA, it is not possible to infer negation of a fact from its absence in the model.

A knowledge base is modeled in DL by *concepts*, *roles* and *individuals* (these terms relate to *classes*, *properties* and *objects* in OWL and both terminologies are used interchangeable in this paper). The knowledge domain is modeled by a hierarchy of concepts which are connected by roles. Individuals are then assigned to these concepts and thereby build a specific description within the scope of the knowledge domain.

With the advent of semantic web technologies, description logics have gained importance as they build the underlying formalism of ontology languages like OWL – for example, OWL-DL v2 corresponds to the description logic $\mathcal{SHOIN}(\mathcal{D})$ which supports hierarchy of roles (H), object value restrictions (O), inverse roles (I), cardinality restrictions (N) and data types (D). This specific DL is also supported by the Pellet reasoner which we used in our prototype implementation, as explained below. Description logics and ontologies themselves do not allow expressing rules which makes some modeling tasks tedious and verbose (e.g. transitivity of properties) and can further be a limitation in cases where DL alone is not expressive enough. The rule language SWRL is based on DL and allows specifying rules over facts from OWL ontologies. Although SWRL itself is much more expressive than DL and can lead to non-decidable models it can be used in a "DL-safe" way that does not go beyond the expressivity of DL, as described in [5].

The concepts of description logics will be used in this paper for describing the policy model and the DL-based languages OWL, SWRL and its query extension SQWRL have been used for the prototype implementation of the policy model and a respective PDP.

B. Description logic based policy structure

The policy structure we use is similar to that of XACML in that we describe a *policy* as a *rule combining algorithm* and a set of *rules*, each with a *target* description and an *effect*. However, details of XACML which are not necessary within the scope of this paper have been abstracted away and further, in order to reflect the order of rules that is needed for some of the rule combining algorithms, we introduced an injective functional *hasNumber* relation that assigns a number to each rule. A policy is thus modeled as the following DL concept

$$Policy \equiv (\geq 1\ hasRule.Rule) \sqcap = 1\ hasCombAlgo.\{firstApplicble, denyOverrides, permitOverrides\}$$

where *rules*, *targets* and *effects* are denoted by the following class expressions:

$$Rule \equiv \forall hasTarget.Target \sqcap \forall hasEffect.Effect \sqcap \forall hasNumber.Integer$$

$$Effect \equiv \{permit, deny\}$$

$$Target \equiv Subject \sqcap Ressource \sqcap Action$$

C. Meta-policies

In addition to policies, we specify a model for meta-policies which can be thought of a "policies about policies" and are used to formulate guaranteed properties which must be fulfilled by the actual policies. On the one hand, these guarantees can be used by policy administrators as invariants to check their policies against, thereby for example verifying that a company's policy complies with certain usage restrictions as stated by the company's regulations. On the other hand, we propose using meta-policies in order to resolve conflicts between different policy domains, as described below in section V. A meta-policy comprises a target definition, an effect and an optional compensation which defines an action that must be executed in case the meta-policy's decision becomes overwritten. While target and effect are defined as above, *Compensation* is defined by the set of described nominals $\{comp_1,..,comp_n\}$ each referring to the (unique) name of a compensating action (e.g., a Java class name) whose purpose is described below. A meta-policy is thus denoted by the following class expression *Meta* and its two subclasses PermitMeta and DenyMeta, comprising those meta-policies with a *permit* and a *deny* effect, respectively:

$$Meta \equiv \forall hasTarget.Target \sqcap \forall hasEffect.Effect \sqcap \forall hasCompensation.Compensation$$

$$DenyMeta \sqsubseteq Meta \sqcap \text{not}\ PermitMeta$$

$$\sqcap \exists hasEffect\ \{deny\}$$

$$PermitMeta \sqsubseteq Meta \sqcap \text{not } DenyMeta \sqcap \exists hasEffect\ \{permit\}$$

A specific meta-policy is then defined as a subclass of either *DenyMeta* or *PermitMeta* and contains only the specification of a target, as the following exemplary meta-policy *DenyScientist* which denies access to all targets described as individuals of the concept *Scientist*.

$$DenyScientist \equiv DenyMeta \sqcap hasTarget\ (Scientist)$$

D. Conflict-freeness of meta-policies

The purpose of meta-policies as proposed in this paper is to state policy invariants which are guaranteed to be enforced, even in the case of conflicts. Obviously, these invariants must not contradict themselves as otherwise a policy decision would be classified as both strict and defeasible and the PEP would be indifferent about whether the decision is compatible with other domains or not. However, simply leaving it up to the developer to ensure conflict-freeness of meta-policies is not a sensible option as this would require the developer to manually identify every potential conflict in a (potentially large) set of meta-policies – a task which is tedious and becomes even more difficult as relevant information is implicitly "hidden" in ontologies. For instance, if the subjects of two meta-policies are described by the DL concepts *Scientist* and *ProjectManager* it is not immediately visible if these two meta-policies may apply to the same subject or not. Only if the ontology explicitly declares *Scientist* and *ProjectManager* as disjunct concepts one can be sure that both meta-policies are never applicable at the same time and that no conflict can occur. In order to support developers in ensuring the conflict-freeness of their meta-policies, the DL-based model can be used as follows to detect possibly conflicting meta-policies:

The idea of checking the model for possible conflicts is to construct a class that is only satisfiable if a possible conflict between meta-policies exists and that helps to detect conflicting meta-policies (the other way around, a class that is only satisfiable if all meta-policies are conflict-free would appear more natural but is cumbersome to realize because of the underlying open-world-assumption). We call this class *Impossible* and add it to the policy model, together with an individual *imp* of that class. Both are of course only added for the purpose of detecting conflicts and are removed after the validation process as otherwise the PDP had to work with a conflict-free, yet unsatisfiable (and thus unusable) model. The *Impossible* class is constructed as a subclass of a pair of *PermitMeta* and *DenyMeta* classes and the individual *imp* is an instance of it:

$$Impossible \equiv (D_1, \sqcup \ldots \sqcup D_m) \sqcap (P_1 \sqcup, \ldots, \sqcup P_n)$$
$$Impossible(imp)$$

In case the meta-policies do not contain any possible conflicts the *Impossible* class becomes unsatisfiable and as a consequence the model will be inconsistent because the *imp* individual cannot be assigned to an unsatisfiable class. If however possible conflicts exist between meta-policies, the model will be satisfiable and the reasoner will infer properties for the *imp* individual which allow a developer to identify the source of the conflict so it can be removed.

To illustrate this, we give a brief example, assuming that two meta-policies *PermitProjectManager* and *DenyScientist* exist – the former allowing a certain request for all subjects of type *ProjectManager* and the latter denying the same request for all subjects of type *Scientist*. As mentioned before, the source of a conflict lies here in the fact that a subject might exist that is both, *Scientist* and *ProjectManager* at the same time (which might not be immediately obvious in more complex models). In this case, the reasoner will identify the conflicting meta-policies by assigning the *imp* individual to them and the inferred properties of *imp* reflect the values of the access request which would lead to the conflict. Figure 2 shows which values of an access request would lead to a possible conflict between *PermitProjectManager* and *DenyScientist*. The cause of the conflict itself is identified by the two different values for the *hasEffect* property.

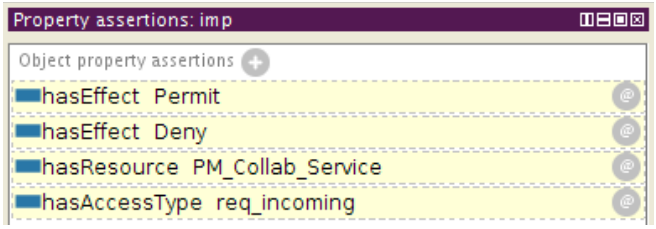

Figure 2 Properties of *imp* individual, explaining a possible conflict.

From this information, explanations could be generated to inform the developer about the possible conflict, its cause and possible resolutions of it.

V. Policy Decision

In the previous subsections we described the structure of rules, policies and meta-policies by means of DL. Based on this structure, we will now describe how an access request is decided using semantic web technology (subsection A) and how conflicts between different policy domains are detected and solved (subsection B).

A. Decisions of a single domain

When a subject wants to access a resource, an access request is intercepted by the PEP and forwarded to the PDP. The PDP evaluates the policy and returns an access decision, determining whether the access is permitted or denied and whether the decision may be overwritten by other policy domains. The PEP is then responsible for enforcing the PDP's decision. We denote an access request as the triplet representing a target *target=<(subject),(resource),(action)>* and a policy decision as *decision=<{permit,deny},{strict,defeasible}, compensation>*.

As every policy can comprises several rules with different effects, the decision process must support overriding the decision of one rule by that of another one (in XACML, this procedure is determined by the *rule-combining-algorithm* element). As overriding existing facts in a knowledge base would require non-monotonic reasoning which is not supported by plain DL, it is not possible to decide a policy

request only on the basis of DL. However, query languages like SPARQL and SQWRL allow retrieving the relevant facts from the knowledge base without requiring non-monotonic reasoning. We therefore propose using SQWRL [6] queries for evaluating access requests – an extension to the SWRL [4] rule language which applies non-monotonic operations only to the result set of a SWRL query and does not write them back into the knowledge base. For each of the supported rule combining algorithms there is a separate SQWRL query and each of them is executed during the policy decision process. As an example, the SQWRL query for the *denyOverrides* algorithm looks as follows; the other queries for *permitOverrides* and *firstApplicable* are constructed alike and omitted here for the sake of brevity:

$$Rule(?r) \wedge Policy(?p) \wedge hasRule(?p, ?r) \\ \wedge hasTarget(?r, target) \\ \wedge hasCombAlg(?p, denyOverwrites) \\ \wedge hasEffect(?r, deny) \rightarrow sqwrl{:}select(deny)$$

For each access request these queries return a single effect value (*deny* or *permit*) which is communicated back to the PEP as the final decision. In a second step, the PDP needs to evaluate whether the decision should be classified as strict (i.e. it has to be enforced in every case) or defeasible (i.e. it can be combined with other domains and possibly be overridden). For this purpose, the same access request is tested against the set of meta-policies, using a similar SQWRL query:

$$Meta(?m) \wedge hasTarget(?m, target) \\ \wedge hasEffect(?m, ?e) \rightarrow sqwrl{:}select(?e)$$

If this query returns an empty result set, the access request is not covered by any meta-policy and is classified as *defeasible* by the PDP. If the query results in the same effect as the previous evaluation of the policy, the decision is classified as *strict* and the PDP returns the effect and the classification to the PEP. In case this query results in a different effect than the result of the policy evaluation an *internal conflict* occurs, i.e. the policy contradicts its meta-policy. If that should happen, the decision of the meta-policy is preferred and classified as strict, thereby ensuring that it always the meta-policy which determines the final decision. In general, developers might want to avoid such internal conflicts as the effective results of a policy are not easily recognizable from the model anymore if they become overridden by the meta-policy, thereby making the policy model harder to understand and maintain. Detecting and removing internal conflicts statically (i.e. at design time) using DL-based techniques would be possible (e.g. by constructing a subclass of $\neg Meta \sqcap Policy$ and checking for its unsatisfiability) but has not been implemented in the prototype for two reasons: at first, internal conflicts might be wanted, for example in scenarios where a set of pre-defined meta-policies specifies the overall access rights for a whole company and each department of the company may define its own set of policies within the scope of these overall meta-policies. Second, being able to statically check for internal conflicts implies that the whole policy has to be specified invariably on the basis of DL and facts which are known at design time. This would be a serious limitation as it would not be possible to include conditions referring to runtime information like time into policies – an option that we certainly did not want to exclude. Rather, our approach envisions detecting internal conflicts upon their occurrence at runtime, preferring the meta-policy's decision and logging the issue so administrators are informed about possible inconsistencies in their policy model. This way, it is possible to include data that is only available at runtime into policy decisions while ensuring that meta-policies act as invariants which are guaranteed to be enforced.

B. Decisions of multiple domains

Up to now, we have only considered the traditional case of a policy decision within a single domain. Now, we will look at PEPs which reside in multiple policy domains at the same time and how they can make use of meta-policies to handle conflicting policy decisions by these domains. We assume that a PEP is originally associated to an initial PDP and then can connect to further PDPs at runtime, as the illustrated by domains @home and @work in the example above. If the PEP is controlled by multiple policy domains, it will forward an access request to each domain's PDP and will subsequently receive a number of decisions, each consisting of an effect, a classification as strict or defeasible and a compensation action, as explained in the previous section. Conflicts between these decisions arise if one PDP decides to allow the access request while another PDP denies the request. The PEP then uses the strict/defeasible classification to resolve this conflict as follows:

If all conflicting decisions are classified as defeasible, all domains accept overriding their policies in support of combined policy domains and the PEP can simply select and enforce one of the decisions, for example that of the PDP which was first connected to the PEP. In case only one of the conflicting decisions is marked as strict while all others are defeasible, it is of course the strict decision that is enforced by the PEP. If however multiple conflicting decisions are classified as strict, the consequence is that these policy domains cannot be combined with each other – otherwise invariants set by the meta-policies would be violated. The only option in this case is to disconnect the PEP from the conflicting PDPs, thereby releasing the linkage of the conflicting policy domains. So, the PEP selects the first *strict* decision and executes the *compensating action* of all further conflicting decisions being marked as *strict*. The compensating action refers to a function which immediately removes the PEP from the conflicting PDPs and whose implementation depends of course on the underlying protocols used for combining and leaving domains which are not in the scope of this paper and will therefore not be discussed in more detail. As a result from leaving a policy domain, all services provided in that domain will not be accessible anymore. Applied to the example above, if @work and @home were incompatible, Alice had to leave either of them and consequently the services she provides could not be used from either her home- or her company domain.

This way, it can be guaranteed that the invariants defined by meta-policies are not violated by the PEP – at the price of different non-combinable domains in the case of conflicting *strict* decisions.

VI. PROTOTYPE

The above described policy model, decision process, classification as *strict/defeasible* and the validation of conflict-free meta-policies have been implemented in form of a proof-of-concept prototype in order to test the practical applicability of the approach. In this section, we present the design of the prototype and discuss the insights which were gained during the implementation.

In a first step, the DL-based policy model from section IV has been realized in form of an OWL ontology and SQWRL queries were applied for deciding access requests as described in subsection V.A. Although this straight-forward way of putting the DL-based policy decision process into practice worked as expected, it had several drawbacks:

For creating SQWRL queries, the Jess rule engine, a reasoner (Pellet[3], in our case) and the Protegé-OWL API[4] were required. This results in a heavyweight implementation of the PDP with about 40 MB of libraries which counteracts the envisioned application scenario of AmI environments with potentially resource-restricted devices. Further, defining policies directly on the basis of OWL is possible but may appear cumbersome to policy administrators which are not used to semantic web technologies.

In order to overcome these limitations, we split up the policy decision process into one part which is purely based on a simple XML structure and one part resolving the semantic information where necessary. So it is possible to run a lightweight PDP on resource-restricted devices while providing reasoning capabilities for resolving class expressions and individuals by a full-blown semantic PDP which can be hosted on a more powerful platform. For each policy domain, at least one such fully-equipped PDP is required while multiple lightweight PDPs can be spread across different platforms in the policy domain and connected to the fully-equipped PDP, as shown in Figure 3. Another benefit is that developers can write their policies in simple XML files and use semantic class expressions only for describing the target of a rule. The process of deciding an access request using our prototype thus works as follows:

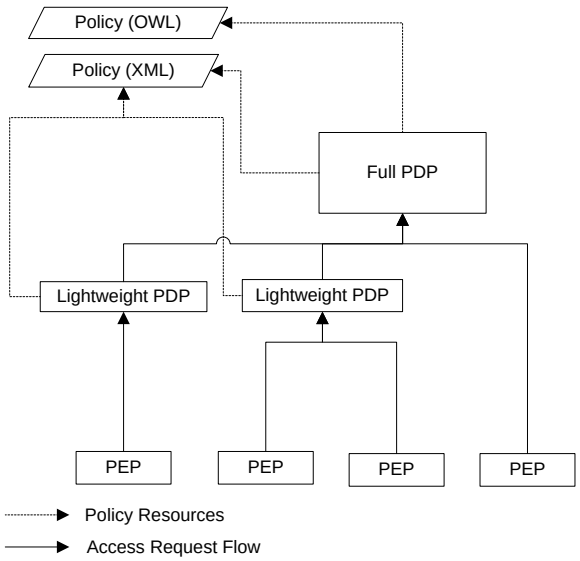

Figure 3 A policy domain with a PDP split up into one full and two lightweight versions.

Developers do not have to formulate policies and meta-policies in form of an ontology but rather can define a simple XML structure that includes class expressions in Manchester syntax in order to reference semantic information stored in external ontologies. When loading such a policy into the PDP, it is processed by a lightweight parser and then converted into an OWL ontology, using a pre-defined OWL template which reflects the policy model introduced in section IV. The result of this process is a complete ontology representing policies and meta-policies which is then loaded into the Pellet reasoning engine.

The next step is then to ensure the conflict-freeness of meta-policies. This is done by adding the *Impossible* class introduced in subsection IV.D and checking it for satisfiability. In case this class is satisfiable, i.e. if a conflict between meta-policies has been detected, the policy developer should be informed about the inferred values of the *imp.hasEffect* property. As this step has not yet been integrated into the prototype it must currently be manually executed using the Protegé 4 ontology editor but a later integration into the PDP is planned, of course. After confirming the conflict-freeness of the model, the PDP keeps the policy specification in memory, both as a XML structure and as an ontology and is ready to evaluate access requests. This process of loading a policy into the PDP is shown in Figure 4. Compared to the illustrated process, a "lightweight" PDP which is not capable of any reasoning functionality would only execute steps 1 and 2a and as a consequence would only able to decide access requests whose evaluation does not involve any class expressions.

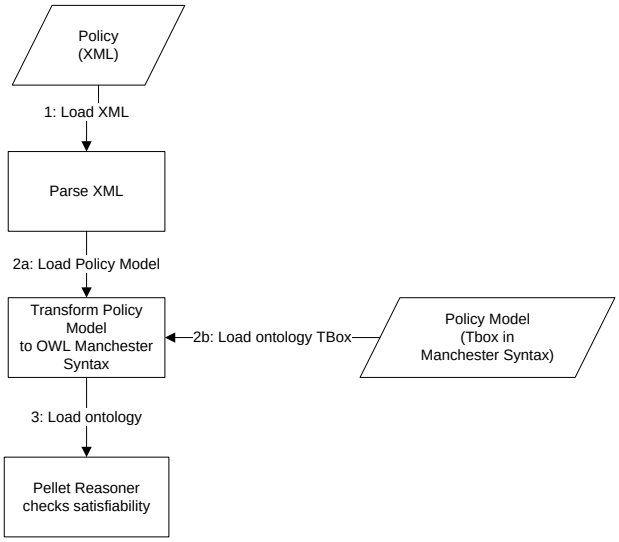

Figure 4 Process of loading a XML-formatted policy, converting it into an ontology and checking meta-policies for conflict-freeness.

When an access request reaches the PDP, it first extracts all available information about the policy target from the request and evaluates the policy, using XPath queries to retrieve the relevant rules and OWL-API[5] together with the Pellet reasoner to test class expressions from the policy target against the information contained in the access request. After the policy has been evaluated to either *permit* or *deny*, the same evaluation is executed against the meta-policies and the decision is classified as *strict* or *defeasible*,

[3] http://clarkparsia.com/pellet/
[4] http://protege.stanford.edu/overview/protege-owl.html
[5] http://owlapi.sourceforge.net/

depending on the evaluation of the meta-policies.

Summarizing, after replacing the SQWRL-based policy decision by the more lightweight combination of XPath and the Pellet reasoner, our prototype fulfills all functionality we described above by means of a generic DL-based model and shows characteristics which suggest its suitability for AmI use cases: the implementation is based on Java 1.6 and OSGi, the footprint of the lightweight version of the PDP is about 8MB which is reasonable for most embedded devices hosting a Java runtime. The full-featured PDP (of which at least one must be present in a policy domain) has a significant larger footprint of about 50 MB, mainly caused by the Pellet and OWL-API libraries.

VII. Conclusion and Outlook

In this paper we presented an approach on resolving conflicts between multiple policy domains by means of meta-policies. Policies and meta-policies have been specified on the basis of description logics – the logical foundation of the semantic web. This way, we foster the integration of access control policies and semantic information as it is used in many ambient intelligence systems, for example. By defining meta-policies, developers can formulate invariants which are guaranteed to be enforced even in the case of conflicts. Taking advantage of the underlying description logics, the set of meta-policies can be checked automatically for conflict-freeness using a standard semantic web reasoner so developers can be informed about conflicts in their meta-policies and possible resolution strategies. As a result of our approach, developers can build systems where services can reside in multiple policy domains without unintentionally overwriting security-relevant decisions of one domain by those of another. This is essential in all architectures where services from different authorities shall be combined, for example in traditional business SOAs as well as in more dynamic AmI environments.

The prototypical implementation of our approach showed that the DL-based model can be realized in a straight-forward way by means of OWL ontologies, SQWRL queries and the Pellet reasoner in combination with the Jess rule engine and the OWL-API. It became however also obvious that the overhead of semantic web libraries is not tolerable for most resource-restricted devices and as a consequence, an implementation strategy featuring a reduced lightweight PDP has been realized.

For further testing, we expect to be able to integrate our approach into existing close-to-market Ambient Intelligence systems such as the Hydra middleware. As part of our future work, we will extend our prototype by protocols for joining and leaving policy domains as well as we aim to support arbitrary conditions in rules. A further possible extension is to not only return deny/permit decisions but rather add blurring modifiers for different data types so a compromise between full denial and allowance becomes possible.

References

[1] Tim Moses (editor), “eXtensible Access Control Markup Language (XACML), Version 2.0”, OASIS Standard, February 2005

[2] V. Kolovski, J. Hendler, B. Parsia, „Formalizing XACML Using Defeasible Description Logics“, in Proceedings of the 16th international conference on World Wide Web, 2007, pp. 677–686

[3] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (editors), “The Description Logic Handbook: Theory, Implementation, and Applications”, Cambridge University Press, 2003, ISBN 0521781760

[4] I. Horrocks et. al. (editors), “SWRL: A Semantic Web Rule Language Combining OWL and RuleML”, W3C Member Submission, May 2004

[5] B. Parsia and E. Sirin and B. C. Grau and E. Ruckhaus, and D. Hewlett, “Cautiously approaching SWRL”, Preprint submitted to Elsevier Science, 2005.

[6] M. O’Connor, .A. Das, “SQWRL: a Query Language for OWL”, in Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), October 2009

[7] H. H. Hosmer, “Metapolicies I”, in ACM SIGSAC Review, 10(2-3), pp. 18–43, 1992, Special issue on Issues '91: data management security and privacy standards.

[8] H. H. Hosmer, “Metapolicies II”, in Proceedings of the 15th National Computer Security Conference, pp. 369–378. October 1992.

[9] K. Twidle, N, Dulay, E. Lupu, M. Sloman, “Ponder2: A Policy System for Autonomous Pervasive Environments”, in Proceedings of the Fifth International Conference on Autonomic and Autonomous Systems (ICAS), 2009

[10] K. Verma, R. Akkiraju, R. Goodwin, „Semantic Matching of Web Service Policies“, in Proceedings of the Second Workshop on SDWP, 2005

[11] L. Kagal, T. Finin, A. Joshi “A Policy Based Approach to Security for the Semantic Web”, in The Semantic Web – ISWC, 2003

[12] E. Damiani, S. De Capitani di Vimercati, C. Fugazza, P. Samarati, “Extending Policy Languages to the Semantic Web”, in Proceedings of the International Conference on Web Engineering, pp. 330-343, 2004

[13] V. Kolovski, J. Hendler, B. Parsia, “Analyzing web access control policies”, in Proceedings of the 16th International Conference on World Wide Web (WWW), 2007

[14] T. Priebe, W. Dobmeier, C. Schläger, N. Kamprath, „Supporting Attribute-based Access Control in Authorization and Authentication Infrastructures with Ontologies”, in Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES), 2006

[15] R. Ferrini, E. Bertino, “Supporting RBAC with XACML+OWL”, in Proceedings of the 14th ACM symposium on Access control models and technologies (SACMAT), pp. 145-154, 2009

[16] L. Kagal, “The Rein Policy Framework for the Semantic Web”, http://dig.csail.mit.edu/2006/06/rein/, 2006

[17] W3C Recommendation, “OWL 2 Web Ontology Language”, http://www.w3.org/TR/owl2-overview/, 2009

[18] T. Dursu, “A Generic Policy-Conflict Handling Model”, in Proceedings of Computer and Information Sciences (ISCIS), 2005

[19] N. Dunlop, J. Indulska, K. Raymond, “Dynamic conflict detection in policy-based management systems”, in Enterprise Distributed Object Computing Conference, IEEE International, 2002

[20] G. Russello, C. Dong, N. Dulay, “Authorisation and Conflict Resolution for Hierarchical Domains”, in Proceedings of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), 2007

[21] A. Uszok et.al., “KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement”, in Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY), 2003