Extended Privacy Definition Tool

Martin Kähmer, Maike Gilliot

Institute of Computer Science and Social Studies, Department of Telematics
Albert-Ludwigs University Freiburg
Friedrichstrasse 50, 79098 Freiburg
{kaehmer,gilliot}@iig.uni-freiburg.de

Abstract: Eliciting non-functional security requirements within a company was one of the major aspects of the SIKOSA[1] project [WKKG07]. Scenarios, such as the METRO one presented in this paper, show how besides the company's internal requirements, customers' preferences play an important role as well. However, conflicts between specific customers' privacy policies and the company's one need to be detected and dealt with. We present a policy language able to tackle the comparison problem and outline a monitor for the enforcement of such policies.

1 Introduction

Personalized services are often viewed as the panacea of e-commerce [SaSA06]. User profiles, such as click streams logging every site the user accesses, are used to generate a profile of interests of the users. The decisive advantage of such services lies in the opportunity of entering a one-to-one relationship in order to thereby achieve more effective customer loyalty. Thereby, service tailoring is not limited to e-commerce anymore. In Germany, the METRO-Group is developing the "Future Store", where shopping trolleys are fitted with personal shopping assistants, i.e. little computers connected to the store's information system. Services, such as recommender systems, are personalized by means of the customer cards [KaAc06]. However, personalization involves intensive collection and usage of personal-related data. If customers want to benefit from such services, enforcing privacy by minimizing data disclosure is no longer possible. Customers need means to control the usage of their data [PHBa06].

In this paper, we present the Extended Privacy Definition Tool (ExPDT) as a mean for companies to comply not only with regulatory and business requirements, but also with customers' privacy preferences. In chapter 2, we classify our ExPDT solution consisting of a policy language described in chapter 3 and a corresponding monitor described in chapter 4. In the conclusion, an outlook on further work is given.

---

[1] SIKOSA: Sichere Kollaborative Softwareentwicklung und Anwendung. A project in collaboration with the Uni of Heidelberg (Software Engineering Group), Uni of Hohenheim (Information Systems II), the ETH Zürich (Databases and Information Systems) and Uni of Freiburg (Telematics) funded by MWK of BW.

2 Tackling the privacy problem

Privacy and security for enterprise information systems is about ensuring that business processes are executed as expected and operations such as data accesses are in accordance with a prescribed or agreed on set of norms, such as laws, regulations, and decisions. Solutions for achieving that can be broken down to two main approaches according to the time of application. One approach is called the retrospective reporting where traditional audits usually done through manual checks based on comprehensive logs and reports of the last period of time are used to show policy conformance [Acco07]. The other, more recent approach is often called security by-design exhibiting a more preventive focus. Non-functional privacy and security requirements are captured and subsequently propagated into the enterprise applications. We propose a model of different layers w.r.t abstraction and possibility for automation by IT (c.f. Figure 1). Since laws only describe, what has to be done in general, these regulations have to be interpreted into control objectives for the particular business domain of a company. Although formulated by experts, those control objectives are still in natural language. For IT systems, they need to be interpreted once more and mapped to the particular services, components, and employees of the company. Policies are a set of formalized rules precisely describing for each unit what is allowed or mandatory and what is prohibited. Such policies serve as input for security monitors, enforcing them on the lower system level. A high degree of automation is only possible within the policy and the monitor layer, where ExPDT is situated, as this is the first level providing laws in a machine-understandable format.

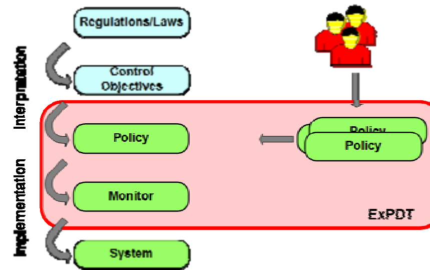

Figure 1: ExPDT within the layer model

2.1 Policy and monitor requirements

For a policy language, it is not only essential to feature sufficient expressiveness based on a wide range of compassing modalities like permissions, prohibitions and orders and on context inclusion based on a fine grained vocabulary [HPSW06, BAMK05]. It is also necessary for a policy language to allow for modular specification and, in particular, for comparison of policies. Not only regulatory objectives that could be realized in one central policy have to be enforced by a company. Its customers need to be able to control the collection and usage of their own personal data by formulating their own privacy and security preferences [Bund83, PHBa06]. Their policies need to be adhered to at runtime, too.

For enforcing such expressive languages, monitors have not only to cope with conditions, i.e. he "traditional" access control, but also to enforce orders and obligations. Particularly, evaluation of policy rules depends on the evaluation of their conditions. Therefore, a monitor has to provide and track the current values of condition variables by requesting the relevant information from the system.

2.2 Related Work

Tackling privacy by means of policies is not new. The W3C developed P3P to express website privacy policies in a machine-readable format and its counterpart APPEL to express customers' preferences [P3P, CrLM05]. Both designed for comparing policies, they lack conditions, prohibitions, obligations, and enforceability. For internal policies, XACML was designed by the OASIS consortium [Mose04]. While XACML can express privacy related policies, IBM's EPAL is dedicated to this task with its fine grained vocabulary, high expressiveness and monitor integration [EPAL03]. NAPS enhances EPAL to an algebra allowing for modular specification of policies [RaSt06].

3 ExPDT - Expressing Privacy Policies

The ExPDT language is a policy specification language in OWL-DL that allows users to develop declarative privacy and security policies over specific domain knowledge. The ExPDT language is used to describe positive permissions, negative prohibitions, and orders that have to be adhered to if certain contextual conditions are met or some obligations have to be fulfilled. Based on the algebraic framework NAPS, it inherits semantic and combination operators allowing for a modular specification of policies. A distinguishing feature of the language is the difference operator for policy for analysis and comparison. As ExPDT is geared towards dynamic environments, it can deal with incomplete context information and also includes sanctions that can be imposed.

For the presentation of the ExPDT language in the following chapters, we introduce three logical layers: the language layer, the domain layer and the instance layer. At the bottom, the language layer establishes the basis by providing fixed vocabulary for the specification of language itself, just like the grammar of a natural language. Based upon that, the domain layer fills up vocabulary by defining the instances of assets, actuators and environment. In contrast to the language layer, the specifications on the domain layer have to be consistent with the current scenario. Therefore, they are subject to occasional adaptations. A common understanding of privacy preferences is not possible, until language and domain are commonly defined. On the third layer, concrete policy instances both of the companies as well as of the customers can be defined, exchanged and agreed upon.

3.1 Language specification layer

A language is made from its syntax and semantic. Hereby, syntax is the definition of all words allowed to be used in the language as well as their set up, in particular the definition of a rule and its parts. The semantic describes the meaning behind the syntax. For a policy, the semantic is given by its evaluation function that provides the results for a particular policy query. The subsequent specification of the ExPDT language layer follows this sectioning and starts with the description of the syntax, before the semantic is given with its evaluation function and the policy operators for combining and comparing policies.

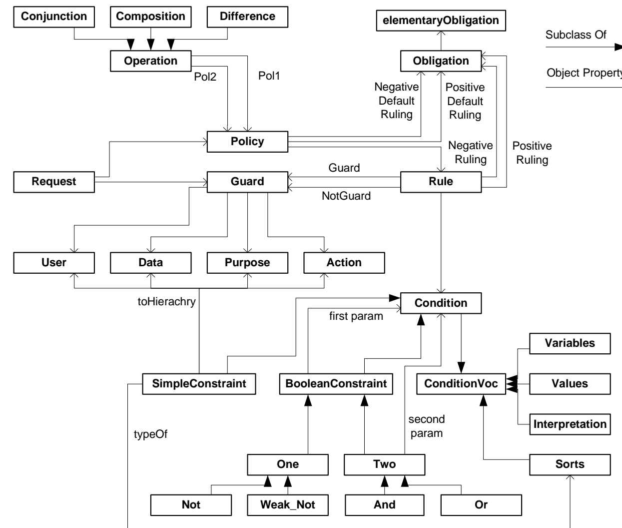

Figure 2: OWL class diagram of ExPDT policy language

3.1.1 Syntax

Although the ExPDT language features a representation in OWL, its syntax is presented in a more space-saving way on the basis of the simplified owl class diagram with the inheritance and selected properties of the OWL classes shown in Figure 2. Short examples of the actual OWL syntax[2] are given in chapters 3.2 and 3.3 for the domain and instance layer.

A ***policy*** consists of a prioritized list of rules and a ***default ruling*** in the case where no rule applies. A ***rule*** itself is comprised of one or more possibly negated guards constraining the scope of this rule from users, actions, data and purpose, a number of conditions and the ruling that subsequently delivers the decision of this rule. Hence, a generic rule has the following form: [(user, action, data, purpose), conditions, ruling]

For intuitive specification of the scenario on the domain layer later on, the element instances of a ***guard*** are partially ordered in hierarchical structures allowing for grouping of instances and the formulation of policies rules applying to entire sub-hierarchies, e.g. to all users of a particular department or all the data belonging to contact information. Thereby, each of them has his own structure: customers, employees and services are combined in the *user* structure, sensitive data items are described in ***data***, possible actions on these data items are given in ***action*** and the possible intentions of actions in question are structured in ***purpose***. It is not required that hierarchies have unique predecessors, as long as they form a directed acyclic graph.

Regulations often depend on context information, e.g. permitting data access only if the customer is not under age or the legal guardian has given his consent. For the inclusion

---

[2] The full OWL ontology can be found at <http://www.telematik.uni-freiburg.de/mitarbeiter/kaehmer/ExPDT/>

of such constraints, ExPDT reverts to a 3-valued, many-sorted condition logic. A condition is a formula of this condition logic defined over the condition ***vocabulary*** and its ***interpretation*** functions. A vocabulary consists of the final set of ***sorts*** (i.e. variable types) each with a final set of ***variables***. The set of non-logical symbols of ***simple constrains*** includes relations, the set of logical symbols the operators ***and***, ***or***, ***not***, ***weak not***, ***0***, ***1*** and u as undefined. The single-valued operators not and weak not have only a ***first*** parameter, the others additionally a ***second*** one. Formulas and terms of the condition logic are recursively defined as usual as in the predicate logic free of quantors. The semantic of a formula is given as in the 3-values Łukasiewicz L_3 logic [Gall88]. The undefined value is advantageous to an environment of dynamic character, such as stores with continuously changing customers and modified or switched services, in that it supports the rule evaluation even with incomplete context information. If the evaluation of a condition does not return a clear decision 0 or 1 due to lack of information, the evaluation is continued and the decision conjunctively linked with the final decision, as will be shown in chapter 3.1.2.2.

A policy rule not only regulates the actions on data items, but can impose ***obligations***, such as "notify customer" or "delete data within one day". ExPDT does not consider obligations as pure black box instructions, but has an underlying obligation model (O, ≤, ∧, T, ⊥) of a half lattice with maximum element top T as the empty obligation and the minimal element bottom ⊥ as the unfulfillable obligation. Imposing the obligation T means that the action of the guard can be carried out without further undertaking, imposing ⊥ that an action may not be carried out. For policy specification, we use a lattice above the power set of the ***elementary obligations*** Õ, of the conjunction as aggregation, T := ∅, ⊥ := Õ and for a, b in *Pot*(Õ): A ≤ B :⇔ A ⊆ B. The usage of this model allows us for the exclusion of impossible obligation combinations, such as "delete data within 1 week" and "keep data for a year" at the same time.

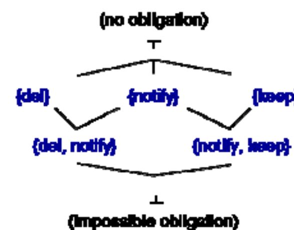

Figure 3: Half lattice of obligations

The ***ruling*** of an ExPDT rule and the default ruling of the overall policy are specified by a tuple of obligation (***postiveObligation***, ***negativeObligation***), each an element of *Pot*(Õ). While the tuple (⊥, ⊥) indicates an internal error state, the actual semantic of a ruling is discussed in chapter 3.1.2.1. Since ExPDT policies make statements about users performing an action on some data for a particular purpose, the policy ***query*** is accordingly also a tuple of user, action, data, and purpose, too.

3.1.2 Semantic

While the specification of policies, queries and rulings were part of the syntax, the evaluation function of a query resulting in a ruling for a given policy defines the semantic of the ExPDT policy language. Firstly, the semantic of a single ruling is explained, followed by the description of the evaluation function. Afterwards, the combination and comparison operators on policies are introduced and defined.

3.1.2.1 Ruling of a rule

The required authorization and order rule modalities presented can both be expressed in the ExPDT language, as shown in Table 1. Actions can therefore not only be permitted but also forbidden. It is also possible to compulsorily regulate the execution of actions. In addition to the conditions (sometimes called provisions), obligations can also be imposed on the user. These are actions that have to be performed in future. The ExPDT language also allows the users a certain degree of freedom in rule compliance. While this always applies in the case of a permit – if an action is allowed, one does not necessarily have to use this right – users can decide for themselves whether they adhere to a prohibition or a command. If they do not, sanctions in the form of additional obligations can be specified in an ExPDT rule. If these sanctions correspond however to the impossible obligation ⊥, adherence becomes necessary for the users. The various rule modalities as well as the obligations and sanctions are mapped in ExPDT via the tuple of obligations of the ruling, whereby the first obligation specifies the future additional actions and the second possible sanctions. Here are some examples:

- Permission: A retailer can access the customer number. Ruling: (⊤, ⊤)
- Permission with obligation: A retailer can access the customer's shopping list but not secretly. Ruling: (notify, ⊤)
- Prohibition with sanction: A retailer may not access the shopping list, which is achieved by imposing the impossible obligation. If he disregards this prohibition and reads it out, he must inform the customer and pay him a fine as sanction. Ruling: (⊥, payFine)
- Essential command: The security administrator must in any event classify the data requested according to its sensitivity. For the ruling, this means that there are no additional obligations but there is the sanction according to the impossible obligation. Ruling: (⊤, ⊥)

Modality	Obligations	Sanctions	Ruling
Permission			(⊤, ⊤)
	O+		(O+, ⊤)
Prohibition			(⊥, ⊤)
		O-	(⊥, O-)
Order			(⊤, ⊥)
	O+		(O+, ⊥)
		O-	(⊤, O-)
	O+	O-	(O+, O-)
Error			(⊥, ⊥)

Table 1: ExPDT codes the modalities into the ruling

3.1.2.2 Evaluation of a policy

The semantic of the policy language is determined through the evaluation function $eval_{\alpha}(P,q)$ for a query q regarding a particular policy P and current assignment α of the contextual condition variables. Roughly, the function searches through the list of policy

rules until a rule is matched by the query. Matching means that all elements of the rule guard are either equal to the user, action, data, and purpose of the query or stand higher up in their corresponding hierarchy. Additionally, the condition of the rule must not evaluate to false using the current variable assignment. Thus, queries are not restricted to minimal elements of the guard hierarchies and allow for scenarios, where, for example, a basic policy company policy referring to departments is only composed with a department policy making concrete statements about individuals. Although the particular user Bob may not be mentioned in the company policy, its rules still apply to him. The complete evaluation algorithm works as follows:

1) Initialize the ruling r_p with $(\top, \top)$ and preset evaluation status v to default.
2) Evaluate rules one by one according to their priority.
    a. If the rule's guard is matched by the query and its condition evaluates to 1, return the conjunction of the rule's ruling and hitherto accumulated r_p as policy ruling and an evaluation status v of final.
    b. If the query is matched by a rule's guard and its condition evaluates to u, add rule's ruling to r_p, set the status v to applicable and proceed with the next rule.
3) If the status v is applicable, than return r_p as ruling and that status.
4) If the status is still default, no rule has matched and the default ruling is returned together with the status v default.

The case of incomplete context information resulting in an undefined condition value for a rule is taken into account by accumulating the ruling of such a rule with a possibly previous found ruling, i.e. conjunct both the positive obligations and the negative obligations, and proceeding with the evaluation. Hence, it is ensured that the evaluated ruling is possibly too restrictive due to the additional obligations, but never too weak.

3.1.2.3 Combination of policies

The extensive dragging along of the evaluation status v with its distinction of final, applicable or default ruling allows ExPDT the definition combination operators despite the stub-behavior of the policies. The stub-behavior corresponds to the intention of the default ruling is to ensure a safe ruling until another rule matches, therefore the refinement of a default ruling with an applicable or final one should be possible in case of a policy combination. In ExPDT, two combination operators are defined: the conjunction $P_1 \wedge P_2$ thereby evaluates P_1 and P_2 with equal priority, while the composition $P_1 \parallel P_2$ gives P_1 higher priority for the evaluation, therefore evaluates it first. For more detailed combination tables of the rulings and generating algorithms, see [Raub04]. On account of the evaluation status, an adequate policy in turn emerges from connected policies, which leads to numerous algebraic laws, such as the query evaluation of $(P_1 \wedge P_2)$ is equal to the conjunction of the rulings separately evaluated for P_1 and P_2. If a policy P consists of the conjunction of two partial policies P_1 and P_2, then each query at the entire policy P leads to the same result as when the query at P_1 and P_2 is each placed separately and only their results are subsequently conjunctively connected. The same thing similarly applies for the composition. Hence, local evaluation of two partial

policies is possible instead of generating the combination of two policies beforehand.

3.1.2.4 Comparison of policies

In related literature, for the comparison of two guidelines one often finds the equivalence where both guidelines always supply the same results and the refinement which examines a guideline as to whether it is more restrictive or specific than another. In practice, these tests are however only suitable to a certain extent for users, if they can only determine with them whether their preferences are fulfilled by a service. How should the users behave, however, if the policy of a service does not correspond with that of their own, therefore being not equivalent or more restrictive? They will not reject a utilization of the service in each case. It can even be their wish, depending on the current situation, to lower their data protection demands in favor of the utilization.

In such situations, the users must be able to quickly survey and estimate how far the service guideline deviates from their own preferences or the service's previous ruling. In dynamic environments, in particular, where the users are faced with many different services and their respective individual policies, this task can no longer be manually accomplished. In order to alleviate the user’s personal decision for or against service utilization, the difference operator for two guidelines is defined in the following. This operator reduces the regulation of the policy to become effective to precisely those rules describing situations that are of interest to the users for their assessments, namely to those that allow additional actions or at least actions on weaker conditions or obligations and therefore supply more generous results.

Difference: Given two policies P_1 and P_2 over compatible vocabulary, the difference $P_2 - P_1$ is a mapping from P x P to a list of rules R that covers exactly those queries q and assignments α of conditional variables that result in a less restrictive ruling for P2, so $(r_i, v_i) = eval_\alpha(P_i,q)$ for $i = 1, 2$ $r_1 \not\leq r_2$. For these, the difference rule list results in the same decisions as P_2.

This rule list describes the functional difference of both policies, so they are compared independent of their possible evaluation status; the stub behavior of the policies is not taken into consideration. This is particularly significant if policy P_1 is to be replaced by a different policy P_2, for example if a customer discards his own preferences P_1 and releases his personal data under the service's policy P_2. Then it is irrelevant whether an action is forbidden owing to the standard ruling of P_1, but this standard ruling is refined with a permit. It is only important here that this action is subsequently permitted. However, if policies P_1 and P_2 are intended to be connected afterwards, the difference should consider the stub-behavior of P_1. For instance, the P_1 default ruling can be replaced by an arbitrary non-default ruling of P_2, which would provide a more specific result without the need to get the users' attention– as long as they are aware of this stub-behavior. However, we abstain from defining difference versions regarding this special behavior by considering the evaluation status v in this paper because they only add complexity without giving additional insight.

In fact, the former mentioned equivalence and refinement of two policies can be computed by means of the difference: if $P_2 - P_1$ results in an empty list, P_2 describe less

restrictive situations and P_2 is a functional refinement of P_1. If the difference of switched policies results in an empty rule list as well, P_1 and P_2 are functional equivalent.

An initial implementation can take place here by way of a brute force approach. The decisions for all possible enquiries and all possible allocations of the environment parameter must simply be calculated for both P_1 and P_2 policies. A rule with the scope of the query (i.e. corresponding guard) and the conditions and ruling of the rule appropriate in each case of P_2 is included in the rule list P_2-P_1, if the ruling r_1 has other or lesser obligations than r_2. This brute force approach tests all possible combinations of enquiries and parameter allocations so that its complexity grows exponentially with the vocabulary used.

Algorithm 1 difference(R_1, R_2): walking through the rule sets R_1 and R_2

```
1:  PolicyDIFF := Ø
2:  for i:= max(R2) downto min(R2) do
3:    for j:= max(R1) downto min(R1) do
4:      RuleDIFF := rulecomp(R1[j], R2[i])
5:      if RuleDIFF ≠ Ø then
6:        for all rd in RuleDIFF do
7:          if guardOf(rd) ∧ ¬guardOf(R1[j]) ≠ Ø then
8:            PolicyDIFF += differenz( (R1[j − 1] downto R1[max]), rd )
9:          else
10:           PolicyDIFF += rd
11:         end if
12:       end for
13:       next i
14:     end if
15:   end for
16: end for
17: return PolicyDIFF
```

Therefore, a more efficient approach is presented in this paper. As outlined by algorithm 1, the rules of both policies are looped through according to their priority, so that each rule of P_2 is compared with all rules of P_1. According to the guard logic, guards can contain disjunctions, conjunctions and negations of guards. Therefore, guards cannot be compared using their top-elements alone, but by the set of hierarchy elements described by them, defined by the closure. For an element h, the closure $\overline{h}$ consists of h itself and all elements lower in the hierarchy. The guard operators $\vee$, $\wedge$, and $\neg$ can be mapped accordingly to $\cup$, $\cap$, and $\not\leq$. If such a comparison detects only equal or more restrictive situations with bigger scope or weaker conditions and obligations, the looping is continued with the remaining rules of P_1. If there are, however, such situations and if they are not captured by a following P_1-rule with lower priority (cf. recursive call), they are formally captured by a new rule that is appended to the difference result. Then, the looping is discontinued for the current P_2 rule and starts with the next rule anew. If all P_2 rules are examined, the construction of the difference terminates. However, before algorithm 1 can start, the

Algorithm 2 DiffNorm(R): Normalization of rule list

```
1:  R^N := Ø
2:  for i:= max(R) downto min(R) {rules in order of priority} do
3:    (g, c, r) := split i {splitting of a rule in its parts}
4:    M := Ø
5:    for all f ∈ {0, 1}^|l_g| do
6:      g_f:=g
7:      c_f:=c
8:      for n = 1 to |l_g| do
9:        if f(n) = 0 then
10:         c_f := c_f ∧ l_c
11:       end if
12:       if f(n) = 1 then
13:         g_f := g_f ∧ l_g
14:       end if
15:       M := M ∪ (g_f, c_f, r)
16:     end for
17:   end for
18:   R^N := R^N ∪ M
19:   l_c := l_c ∪ ¬c
20:   l_g := l_g ∪ ¬g
21: end forR^N
```

policies have to be preprocessed by upgrading the default rulings and normalizing P_2.

For the construction of the functional difference, a distinction between normal rules and default ruling is not necessary. Hence, the standard rulings of both policies are upgraded to normal rules by appending rules to the list with the root elements of the guard hierarchies, without conditions, and with default ruling as rule ruling:

$$(user_{root}, data_{root}, action_{root}, purpose_{root}), 1, (default\ ruling)$$

If there is more than one root for a hierarchy, append a rule for each of them. These rules match all possible queries by design, so that the default ruling is not triggered anymore and can be disregarded for the difference construction.

Since the evaluation function of ExPDT considers the policy rules as prioritized a list with dependencies, the rules of P_2 cannot be individually compared; P_2 has to be normalized first. Following algorithm 2, for each rule all the situations matched by rules with higher priority are explicitly excluded from its guard and conditions. For P_1, this normalization is not necessary because the recursive call already copes with the dependencies.

Algorithm 3 rulecomp($rule_1, rule_2$): comparison of two rules

1: $(g_1, c_1, r_1) :=$ split $rule_1$
2: $(g_2, c_2, r_2) :=$ split $rule_2$
3: RuleDIFF $:= \emptyset$
4: **if** $\overline{g_2 \wedge g_1} \neq \emptyset$ **then**
5:     **if** $r_1 \ngeq r_2$ **then**
6:         RuleDIFF += (g_2, c_2, r_2)
7:     **else**
8:         **if** $\overline{g_2 \wedge \neg g_1} \neq \emptyset$ **then**
9:             RuleDIFF += $((g_2 \wedge \neg g_1), c_2, r_2)$
10:         **end if**
11:         **if** $c_2 \nrightarrow c_1$ **then**
12:             RuleDIFF += $((g_2 \wedge g_1), (c_2 \wedge \neg c_1), r_2)$
13:         **end if**
14:     **end if**
15: **end if**
16: **return** RuleDIFF

For the comparison of two rules in algorithm 3, it has to be determined whether there is an overlap between the two scopes. If they do not overlap, the comparison stops. Otherwise, the difference between these two rules is examined by the following case differentiation:

- $r_1 \nleq r_2$: The ruling of $rule_2$ is less restrictive or different. Independent of the conditions, rule2 is appended to the rule difference and returned.
- Otherwise: The r_2 is stricter or equal. Hence, up to two rules have to be appended to the rule difference:
    1. For queries matching $guard_2$ but not $guard_1$, this stricter or other ruling is in any case new, so that a rule with these queries as guard, conditions of c_2, and ruling r_2 is appended.
    2. For queries also matching $guard_2$ as $guard_1$ (i.e. the disjunction of their closures), the r_1 is less restrictive, but a less restrictive condition c_2 can necessitate another difference rule. This new rule should only describe the new situations, so that its condition is $c_2 \wedge (\neg c_1)$.
       Example: $c_1 = \leq 18$; $c_2 = (\leq 18 \vee guardianOK)$,
       $c_2 \wedge (\neg c_1) = (\nleq 18 \wedge guardianOK)$

For the last case, it is essential to not only evaluate conditions but to also compare the ability to satisfy two given conditions. It must be determined whether one rule restricts its applicability with more strict or equivalent conditions than another rule or features a

greater application space with additional context situations. This yields in the examination whether a condition c_1 satisfies another c_2, i.e. if c_1 is satisfied, then c_2 is also satisfied, or if c_1 results in the undefined state u, c_2 is also undefined or even satisfied. For independency of the current situation, this has to hold for all possible variable assignments. This examination is, however, NP-complete over the number of variables of the vocabulary considered, so that no efficient algorithm is to be expected for the general case. Nevertheless, in order to be able to compare the conditions, the examination is reduced to a satisfy relation similar to [BKBS04], which is at least correct, i.e. if two conditions c_1 and c_2 are contained in the relation, then the above-mentioned satisfaction holds true.

Satisfy relation: Given a conditional vocabulary $\mathcal{Voc}$, the satisfy relation for $\mathcal{Voc}$ is the relation $\rightarrow C_{\mathcal{Voc}} \times C_{\mathcal{Voc}}$. The relation is correct if for all conditions $c_1, c_2 \in C_{\mathcal{Voc}}$ und for all possible assignments α holds. In infix notation:

$$c_1 \rightarrow_{\mathcal{Voc}} c_2 \quad :\Leftrightarrow \quad [\, eval_\alpha(c_1) = 1 \;\Rightarrow\; eval_\alpha(c_2) = 1 \,] \;\vee$$
$$[\, eval_\alpha(c_1) = u \;\Rightarrow\; eval_\alpha(c_2) = u \vee 1 \,]$$

If the opposite direction also holds, the satisfy relation is complete. A correct satisfy relation can often be constructed via the symbolic evaluation by all pairs of atomic formulas with known semantic satisfy dependency, but also via the comparison on a pure syntactic level of their interpretation functions of the same sort. Such a correct satisfy relation is mostly adequate for practical application, even if it is not complete. For if two conditions are mutually dependent and this dependency is unknown to the users and therefore not included in the relation, then such conditions should be independently treated in order to meet the users' expectation. At worst, the differential result hereby increases by an additional rule, however without changing its evaluation.

3.2 Domain layer

Based upon the language layer, the vocabulary is filled up by defining concrete instances of assets, actuators, conditions, and obligations and by and categorizing them accordingly on the domain layer. These specifications should always be consistent with the current scenario and, therefore, needs to be adapted in case of environmental changes. ExPDT uses an ontology specified in OWL-DL, as it supports not only for the representing and displaying of the domain specific knowledge, but also for the automated interpretation and reasoning.

Figure 6 gives an example for the data hierarchy specification. Here, a shopping list is a subclass of the $data_{root}$ DATA. In turn, a medical prescription is a subclass of shopping list, and its instances are dynamically assigned by a property restriction: all shopping lists that contain at least one drug to buy are considered as prescription. Hence, customers can formulate rules for this particular kind of shopping list and taking care not to provide its contents to normal salespersons, but only the druggist of the shop.

Examples of obligation are given in Figure 5. The obligation "delete data after usage" is coded as elementary obligation that, of course, is incompatible with obligation to keep the data afterwards. Obligations used in rulings are either instances of single or multiple

```
<owl:Class rdf:about="DOM#OBL:Delete">
  <rdfs:subClassOf rdf:resource="LANG#OBL:ELEOBLIG"/>
  <owl:disjointWith>
    <owl:Class rdf:about="DOM#OBL:keep"/>
  </owl:disjointWith>
</owl:Class>

<Obligation rdf:about="DOM#OBL:deleteAndNotify">
  <rdfs:subClassOf rdf:resource="DOM#OBL:keep"/>
  <rdfs:subClassOf rdf:resource=" DOM#OBL:notify"/>
</Obligation>
```

Figure 5: Definition of obligations

```
<owl:Class rdf:about="DOM#DATA:shoppingList">
   <rdfs:subClassOf rdf:about="DATA"/>
</owl:Class>

<owl:Class rdf:about="DOM#DATA:Prescription">
  <rdfs:subClassOf rdf:resource="DOM#DATA:shoppingList"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="DOM#COND:contains"/>
      <owl:someValuesFrom rdf:resource="DOM#DATA:drugs"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<Prescription rdf:ID="DOM#DATA:shoppingAtDrugstore"/>
```

Figure 6: Definition of guard elements

elementary obligations. A condition instance is the simple constraint gaveConsent in Figure 4 that is basically spanning a relation between the sort Consent and an element of data hierarchy. The Consent-variable has two possible strings "no" and "yes" that are mapped to the Boolean values 0 and 1 for evaluation purpose by the interpretation function.

```
<CON:SimpleConstraint rdf:ID="DOM#COND:gaveConsented">
  <CON:ToHierarchy rdf:resource="LANG#DATA:DATA"/>
  <CON:TypeOf rdf:resource="DOM#COND:Consent"/>
</CON:SimpleConstraint>

<owl:Class rdf:about="DOM#Consent:Consent">
  <rdfs:subClassOf rdf:resource="LANG#COND:Interpretation"/>
</owl:Class>
<Consent:Consent rdf:about="DOM#Consent:no">
  <LANG:HasValue rdf:datatype=
       "http://www.w3.org/2001/XMLSchema#int">0</LANG:HasValue>
</Consent:Consent>
<Consent:Consent rdf:about="DOM#Consent:yes">
  <LANG:HasValue rdf:datatype=
       "http://www.w3.org/2001/XMLSchema#int">1</LANG:HasValue>
</Consent:Consent>

<owl:ObjectProperty rdf:about="LANGPOL#ASS:Consent">
  <rdfs:range rdf:resource="DOM#COND:Var_Consent"/>
  <rdfs:domain rdf:resource="#ASS:Data"/>
</owl:ObjectProperty>
```

Figure 4: Example condition

3.3 Policy layer

On the policy layer, instances of policy can be formulized by combining the building blocks of the domain layer glued together with the elements of the language layer. Figure 7 shows a privacy policy containing only two rules. The first rule allows a druggist to access a particular prescription for the purpose of giving further information, e.g. the compatibleness of some ingredients. The second rule allows all persons working for sales to read shopping lists, if the customer has consented, the data is deleted afterwards and the customer is notified about this event. Queries for all other situations are not matched by the rules, but by the restrictive default ruling of the policy that prohibits everything not already covered by the

```
<POL:Policy rdf:ID="MyPolicy">
  <POL:PolicyHasRules>
    <rdf:Seq rdf:ID="MyPolicyRules">
      <rdf:li>
        <POL:Rule rdf:ID="Rule1">
          <POL:RuleHasUser rdf:resource="DOM#USER:druggist"/>
          <POL:RuleHasAction rdf:resource="DOM#ACTION:read"/>
          <POL:RuleHasData rdf:resource="DOM#DATA:shoppingAtDrugstore"/>
          <POL:RuleHasPurpose rdf:resource="DOM#PURP:information "/>
          <POL:RuleHasPosObligation rdf:resource="LANG#OBL:top"/>
          <POL:RuleHasNegObligation rdf:resource="LANG#OBL:top"/>
        </POL:Rule>
      </rdf:li>
      <rdf:li>
        <POL:Rule rdf:ID="Rule2">
          <POL:RuleHasUser rdf:resource="DOM#USER:sales"/>
          <POL:RuleHasAction rdf:resource="DOM#ACTION:read"/>
          <POL:RuleHasData rdf:resource="DOM#DATA:advertisment"/>
          <POL:RuleHasPurpose rdf:resource="DOM#PURP:shoppingList"/>
          <POL:RuleHasCondition rdf:resource="DOM#COND:hasConsented"/>
          <POL:RuleHasPosObligation rdf:resource="DOM#OBL:deleteAndNotify"/>
          <POL:RuleHasNegObligation rdf:resource="LANG#OBL:top"/>
        </POL:Rule>
      </rdf:li>
    </rdf:Seq>
  </POL:PolicyHasRules>
  <POL:PolicyHasPosDefaultRuling rdf:resource="LANG#OBL:bottom"/>
  <POL:PolicyHasNegDefaultRuling rdf:resource="LANG#OBL:top"/>
</POL:Policy>

<ASS:Data rdf:about="DOM#DATA:Customer">
  <ASS:gaveConsent rdf:resource="DOM#Consent:no"/>
</ASS:Data>
```

Figure 7: Expamle of a policy

rules. Alike the policy instance, the assignment of the conditional variables to the guard hierarchies – customer has not given consent for actions on his data – is part of the policy layer, but the actual variable values have to be provided by the monitor.

4 ExPDT - Query evaluation and policy enforcement at runtime

The previous section described in detail the semantic and syntactic aspects of the policy language itself. This section focuses on how – at runtime – an incoming query is evaluated and enforced. For example, the store's marketing process "create personalized advertisement" analyzes the purchases of a customer in order select special offers according to his shopping habits. Thus, the system requests access the data of a customer *C*. The access control monitor evaluates this request according to the policy rules of customer C and of the store and, as a result, grants or denies the access to the data of customer *C*.

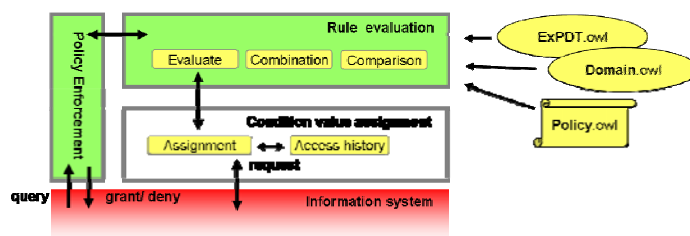

Figure 8: ExPDT monitor

The components of the access control architecture are depicted in Figure 8 . For evaluating the guard, the subject, the action and the object are mapped to their corresponding hierarchy in order to determine the relevant policy rules. Once this is done, based on system events the monitor evaluates the current values of the conditions. This requires an interpretation of earlier access decisions with respect to the current access request. Finally, the monitor must enforce its access decision.

4.1 Rule evaluation

In order to identify the relevant policy rules for a given query, the evaluation functions checks for each rule if the rules elements user, data, action, purpose match the corresponding element of the request. This might be an exact match or the policy's elements are higher within the element hierarchy than the query's elements. Once a relevant rule is detected, it is evaluated according to the evaluation function given in section 3.1.2.2.

4.2 Condition value assignment

In order to evaluate the conditions, the concrete values of the conditions have to be known. Contextual conditions can refer either to the user, the data, the action or the purpose. Some condition values are static. Once the value of the condition "over 18" is true, it stays true. In contrast, the value of the condition "has agreed" has to be evaluated at each request, as the user may in between have revoked his consent. The monitor gets information, like the current time, on demand form the system and keeps itself track of earlier access decisions. This allows the monitor to evaluate conditions like "read at most twice" autonomously.

4.3 Enforcing a policy

Policies of the type [(guard)(conditions)] can be enforced by an access control monitor [HaMS06]. Some obligations are liveness properties, as for example "customer must be notified". This can neither be enforced, nor can violation of these obligations be reported, as the notification may take place some time later. By adding time constraints on these obligations, they are turned into safety-properties: "customer must be notified within 2 days". Such obligations are in general still not enforceable by a monitor, but at least can violations be detected if after two days no notification took place.

5 Summary and Conclusion and further steps

Motivated by an example of the retailer METRO, we show that enterprises have – beside their internal security requirements – also to consider their customers' preferences. This requires an expressive policy language permitting the comparison of two different policies. We therefore present the extended privacy definition tool ExPDT for expressing privacy preferences for access to and usage of personal data allowing also the comparison of two policies. Further more, we show how ExPDT rules are evaluated and how they can be enforced at runtime by a monitor.

For now, ExPDT allows to find differences between policies. How these differences can be resolved has not been considered yet. The next step, therefore, is a negotiation protocol. The goal is not a fully automated procedure, but a tool to assist the negotiation process step by step. Enforcement of obligations and orders is a current open research issue. We currently investigate how obligations can be enforced by rewriting business process. A second approach to the enforcement of obligations uses heuristics in order to determine at runtime process executions that will probably lead to obligation violation.

References

[Acco07] Accorsi, R.: "Automated Privacy Audits to Complement the Notion of Control for Identity Management". In: To appear: Policies and Research in Identity Management, IFIP, 2007.

[AHKP03] Ashley, P.; Hada, S.; Karjoth, G.; Powers, C.; et al.: "Enterprise Privacy Authorization Language (EPAL 1.2)". Submission to W3C, 2003.

[BAKK05] Breaux, T. D.; Anton, A. I.; Karat, C.-M.; Karat, J.: "Enforceability vs. Accountability in Electronic Policies". TR-2005-47, North Carolina State University Computer Science, 2005.

[BKBS04] Backes, M.; Karjoth, G.; Bagga, W.; Schunter, M.: "Efficient comparison of enterprise privacy policies". In: Proc. of 2004 ACM Symposium on Applied Computing, pp. 375 – 382, 2004.

[Bund83] Bundesverfassungsgericht: "Volkszählungsurteil". In: Entscheidungen des Bundesverfassungsgerichts, vol. 65, Urteil vom 15.12.1983; Az.: 1 BvR 209/83; NJW 84, 419, 1983.

[CrLM05] Cranor, L. F.; Langheinrich, M.; Marchiori, M.: "A P3P Preference Exchange Language 1.0 (APPEL)". Tech. rep., W3C, 2005.

[Gall88] Gallier, J. H.: "Logic for Computer Science". John Wiley and Sons, 1988.

[HaMS06] Halmen, K. W.; Morrisett, G.; Schneider, F. B.: "Computability Classes for Enforcement Mechanisms". In: ACM Transactions on Programming Languages and Systems, vol. 28, 2006.

[HPSW06] Hilty, M.; Pretschner, A.; Schaefer, C.; Walter, T.: "Enforcement for Usage Control - A System Model and an Obligation Language for Distributed Usage Control". I-ST-18, Do-CoMo Euro-Labs Internal, 2006.

[KaAc06] Kähmer, M.; Accorsi, R.: "Kundenkarten in hochdynamischen Systemen". In: Proc. of KiVS NETSEC´06, 2006.

[Mose04] Moses, T.: "eXtensible Access Control Markup Language (XACML) Version 2.0". OASIS, 2004.

[PrHB06] Pretschner, A.; Hilty, M.; Basin, D.: "Distributed Usage Control". In: Communications of the ACM, vol. 49(9), pp. 39–44, 2006.

[RaSt06] Raub, D.; Steinwandt, R.: "An Algebra for Enterprise Privacy Policies Closed Under Composition and Conjunction". In: Proc. of Int. Conf. on Emerging Trends in Information and Communication Security (ETRICS), pp. 132 – 146, 2006.

[Raub04] Raub, D.: "Algebraische Spezifikation von Privacy Policies". Master's thesis, Uni. Karlsruhe, 2004.

[SaSA06] Sackmann, S.; Strücker, J.; Accorsi, R.: "Personalization in Privacy-Aware Highly Dynamic Systems". In: Communications of the ACM, vol. 49(9), 2006.

[W3C06] W3C: "Platform for Privacy Preferences (P3P) Project". http://www.w3.org/P3P/, 2006.

[WKKG07] Weiß, D.; Kaack, J.; Kirn, S.; Gilliot, M.; et al.: "Die SIKOSA-Methodik: Unterstützung der industriellen Softwareproduktion durch methodische integrierte Softwareentwicklungsprozesse". In: Wirtschaftsinformatik, vol. 49(3), 2007.