Partial Decision Overrides in a Declarative Policy Framework

Karsten Martiny, Grit Denker
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025
Email: {firstname.lastname}@sri.com

***Abstract*—This article describes how a privacy policy framework can be extended with a mechanism to partially override decisions based on specified constraints. The ability to specify various policies with different overriding criteria allows for complex sets of sharing policies. However, if overriding policy decisions constrain the affected data, decisions from overridden policies should not be suppressed completely, because they can still be applicable to subsets of the affected data. In this work we present a mechanism that automatically generates sets of sharing decisions for complementary sets of affected data, which provides means to specify a wide set of policies tailored to specific properties of the protected data.**

I. INTRODUCTION

Privacy and private data sharing have emerged as critical issues in the development and use of enterprise information systems and personal information management. While sharing data is essential for enterprises and individuals, mistakenly sharing data with the wrong partner can result in exploitation, and unnecessarily sharing sensitive, private data—even with a trusted partner—can result in serious harm.

In [1], we introduced a declarative policy framework based on semantic technologies for enterprise privacy systems. This framework makes the specification of privacy policies available to users without experience in formal knowledge representations. The policy author can express concise privacy policies using the vocabulary of the application domain (modeled as ontology) and intuitive interfaces for policy creation [2]. Our framework does not require the user to have knowledge about the technical details of the underlying formalism. A general shareability theory in our framework allows policy authors to specify with one policy a large variety of property combinations of the domain-specific ontology rather than needing to specify separate policies for every relevant combination of properties.

We have implemented this framework to provide automated policy decisions for data requests and have applied it in a novel enterprise privacy system that is jointly developed by several research teams under Defense Advanced Research Projects Agency's (DARPA's) Brandeis program[1]. Similar to the architecture of XACML [3], our high-level architecture is separated into Policy Administration, Decision and Enforcement Points. The concepts described in this paper are part of the *Policy Decision Point* (PDP), which decides whether or not to approve data requests. The PDP operates without any knowledge of specific data instances and forwards decisions to the *Policy Enforcement Point* (PEP) which then can serve the requester with policy-compliant results. Our framework uses SRI International's (SRI's) Sunflower system[2], which extends the Flora[3] language and reasoner with features such as a HTTP REST Web server, Java APIs, editing UIs, and generating natural language explanations of reasoning results.

A key feature of our policy framework is the ability to specify policies with different precedence levels, enabling overrides of policy decisions depending on their precedence. In [1], a simple overriding mechanism was introduced: A decision D_2 from a policy P_2 *completely* replaces a decision D_1 from another policy P_1, iff P_1 and P_2 issue opposing decisions and decision D_2 overrides decision D_1 according to some criteria. While this approach provides a useful first step to make sure that the final result reflects the decision from the policy with the highest precedence, it is rather coarse and can result in undefined decisions for parts of the data set. To illustrate this, consider the following informal example:

Assume that there is a policy defined as follows: P_1*: "Share medical data with care providers", priority=0*. This policy would grant all requests from care provides access to medical data. Since there are no constraints specified for this policy, the decision applies to *all* medical data stored in the data base. Now, assume that medical data of minors is considered more sensitive and should not be shared with anyone. Thus, another policy is added to deny access to medical data of persons if the persons are less than 18 years old: P_2*: "Deny medical data to care providers", constraints: age*$\leq$*18, priority=1*. This policy has higher priority than the baseline policy P_1 and thus overrides decisions from that

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

[1]This work was supported by DARPA and SPAWAR under contract N66001-15-C-4069. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

[2]http://https://sunflower.csl.sri.com/

[3]http://flora.sourceforge.net/

policy, ensuring that medical data of minors is not shared. Consequently, a care provider requesting medical data now receives the decision *"deny if age*$\leq$ 18*"*. While this result correctly captures the intended protection of minors, the simple overriding mechanism overshoots the mark and removes the decision from P_1 completely. As a result, decisions are now undefined for parts of the dataset, i.e., there is no longer any valid decision for medical data of persons over 18, even though the combination of P_1 and P_2 attempted to enable sharing the medical data of adults with care providers.

To resolve this issue, we extend our framework with *partial overrides* to ensure that overriding decisions with constraints do not suppress overridden decisions completely, but only for the parts of the decision that are actually governed by the overriding constraints. For the remainder of the overridden decision, we want to identify the subset of data that is not affected by the override and make sure that this will be returned as part of the final decision. For the above example, next to the constrained deny decision, the request should return an additional allow decision with complementary constraints, i.e., *"allow if not age*$\leq$ 18*"*.

The remainder of this paper is structured as follows. The following section gives a brief overview of related work on policy languages. A more comprehensive overview of the mentioned privacy policy frameworks comparing several different aspects can be found in [1]. Next, Section III summarizes the key concepts of our existing privacy policy framework and introduces some improvements since its original introduction. Then, in Section IV we discuss required extensions to the constraint specification in order to support partially overridden decisions. Section V shows how the extended constraint specifications can be used to handle partial overrides such that final decisions contain complementary decisions for both the overridden and non-overridden parts of a policy's decision. Finally, the paper concludes with Section VI.

II. Related Work

A number of machine-readable privacy policy languages exist to protect access of sensitive information. Most notable in the context of our work are Ponder [4], EPAL [5], Rei [6], KAoS [7], AIR [8], SecPAL [9], and XACML [3]. A common feature of our framework and all of these languages is that they provide some means of privacy protection through role-based access control policies. However, a key difference between our privacy framework and existing languages is the representation of policy objects. Previous work can be broadly categorized into two approaches: XACML, SecPAL and Ponder require a unique identification of targeted resources and a requested object needs to exactly match a policy object in order to trigger a policy decision. If several related resources (e.g., class hierarchies) are to be addressed by data sharing policies in these systems, a large number of overlapping policy specifications is required to address all relevant cases. EPAL alleviates this bottleneck to some extend by tagging resources with category labels and allows to express policies over categories, but cannot handle more sophisticated relationships of resources.

KAoS, Rei, and AIR on the other hand are expressive enough to represent richer relationships between targeted resources. They do so by essentially exposing a complete logic language to the policy author, who is left to define the precise semantics of each policy from scratch, including the meaning and effect of any background theory in the context of each policy. This makes the task of specifying intended policies much more challenging and much less accessible to non-experts. Moreover, defining relevant resource relationships on an individual policy level will likely lead to significant overhead in the policy specifications, and thus, makes it hard to ensure that specified policies actually reflect the specifier's intent in every case.

Instead, in our approach, policy authors use the domain-specific ontology together with an axiomatic characterization of general background knowledge, and a general, domain-independent shareability theory. This allows the policy engine to generalize to a high degree, with one policy covering many types of requests. In turn, it allows the policy author to write expressive policies in a concise way, capturing their intent without requiring extensive knowledge of the underlying specification formalism.

Another significant difference between our framework and other privacy languages is the treatment of conditions under which data is shared. Virtually all of the above languages evaluate conditions (such as "share only data for persons over the age of 18") before a policy decision is obtained. A disadvantage of this approach is that data already needs to be accessed by the policy decision engine before an access control decision is made, introducing a tight coupling between the policy engine and the database during the decision process. To overcome these problems, our approach does not evaluate conditions prior to deciding on a request, but instead attaches constraints to a potential decision and forwards this to the Policy Enforcement Point. The PEP in turn will enforce these constraints before actually returning results to the requester. This approach allows the PDP to operate completely independent of actual data.

III. Overview of the Privacy Policy Framework

In this section, we summarize the main aspects of the privacy policy framework together with some improvements since its original introduction in [1].

A. Ontology as a Common Data Model

To define privacy policies for some domain, an ontology is used as a common data model for that domain. The ontology defines a domain-specific vocabulary and background theory that describes the types of information relevant to the domain. The policy framework uses the ontology to formally describe requested data (i.e. database queries) as well as constraints. Furthermore, as we outline in Section III-E, axioms in the ontology can be used to infer sharing decisions. The ontology can be defined directly in Flora, or it can be defined in OWL and translated into Flora.

To illustrate this concept, we introduce a small CDM that represents relations between nations, communities, persons, and their medical information. This simplified CDM excerpt is part of a use case developed to specify data sharing policies for medical data to support emergency response coordination

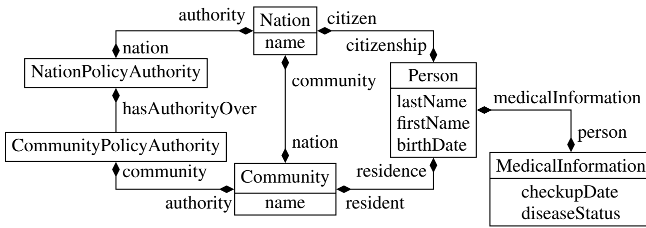

Fig. 1. Pandemic ontology excerpt to illustrate possible links between nations, persons, and their medical information

in the case of a disease outbreak. The ontology represents different classes of the domain (shown as rectangles in Figure 1), together with properties associated to each class. Properties can be both used to represent connections to other classes (depicted by labeled edges), or to represent actual data properties (shown in the lower part of the class representations). As shown in Figure 1, the sample use-case involves nations, communities, policy authorities, and persons. Policy authorities (PAs) can be on a national or on a community level, and there are hierarchical structures between nations and communities, and their corresponding PAs, i.e., communities belong to a nation, and nation PAs have authority over the corresponding community PAs. Each person is associated to a nation and to a community via its `citizenship` and `residence` property, respectively. Moreover, there is medical information associated to each person.

B. The Privacy Policy Framework

In Sunflower, we define a generic privacy policy framework and a shareability theory that support the definition of both permissive and restrictive policies. We explain Flora syntax as it is used, and only to the extent needed to understand the policies. The real policies use fully qualified names with namespace prefixes for all identifiers. In this paper, we omit namespaces for brevity. Sunflower/Flora is used as a *meta-framework* to implement our policy framework. This allows for easy modifications, extensions and analysis of the policy framework's features and semantics. The targeted end users do not need to be familiar with the syntax or technical details of the policy framework. Instead, we provide a user-friendly interface (described in [2]) that allows non-technical users to specify policies, which then are automatically translated into Flora representations.

C. General Sharing Decisions

In general, subject-property-object triples in Flora are represented as `?subject [ ?property -> ?object ]` (variables are preceded by a question mark, `?`). This is used to specify sharing decisions on the highest level through statements `?pa [decide(?req,?dd)]` stating that a policy authority `?pa` decided on a request `?req` with details of the decision specified in `?dd`. In this case, `decide` is a *boolean* property of `?pa`, thus omitting the usual arrow `->` and object specification. Moreover, this property is parametrized, taking `?req` and `dd` as parameters.

The object `Request` contains all information representing an individual request and has properties to specify

- the data requester,

```
@!{PolicyRequestRule}
?pr : Request [ requester -> ?r,
                requestFormula -> ?rf,
                requestTime -> ?rt ] :-
   isnonvar{?pr}, ?pr = policyRequest(?r,?rf,?rt).
```

Fig. 2. Rule to specify a policy request.

- the requested data (as a request formula to identify elements in the ontology), and
- the time of the request.

The `DecisionDetails` object associated with a decision for a specific requests contains all details of this decision, namely

- the actual decision (*allow* or *deny*),
- information (id, description, and priority) of the policy responsible for the decision,
- potential constraints, and
- start and end time of the sharing decision.

Both the `Request` and the `DecisionDetails` objects are defined through functional terms. To illustrate this, the request definition is shown in Figures 2; an analogous definition is used to define the decision details. A flora rule has an (optional) label specified in `@!{..}`, and a head and a body separated by `:-`. Rules work similarly to Prolog rules; they are Horn clauses where the body logically implies the head, and all variables are implicitly universally quantified on the outside of the rule as a whole. The comma `,` denotes conjunction, i.e. logical *and*, the semicolon `;` denotes disjunctions, i.e., logical *or*. The infix operator = denotes unification, `isnonvar{?v}` is a built-in meta-property test to check whether `?var` is not an unbound variable. This rule allows us to use `policyRequest(?r, ?rf, ?rt)` in other rules to automatically create an object of the class `Request` (and analogously for decision details) and then conveniently access its properties subsequently. The use of functional terms to represent requests and decision details are an extension to the policy framework as introduced in [1]. This allows for an easy extension with additional parameters by simply adapting the respective functional term definitions, while existing rules using specific properties of request or decision details can remain unchanged.

D. Request Formulas

A subset of the Flora language itself is used to specify data requests. A request is described using a Flora formula, which may contain a conjunction of subject-property-object triples, such as `?nation [ citizen -> ?citizen ]`, and instance-of formulae, such as `?citizen : Person`. Such formulae are called *Request Formulas (RFs)*. Variables in the formula are free (not quantified), and thus the formula can be interpreted as describing the set of all matching data values.

The classes and properties in the RF must be part of the ontology. Thus, as the ontology provides a common representation of the vocabulary of the domain, the RFs provide a common representation of requests for data in the same domain. As an example, consider the ontology depicted in Figure 1 and assume we want to request the disease status of nations' citizens. In Flora, this RF would be represented as

```
RF_diseasestatus = ${
 ?nation : Nation [
   citizen -> ?citizen, name -> ?nationName],
```

```
?citizen : Person [medicalInformation -> ?medInfo],
?medInfo : MedicalInformation [diseaseStatus -> ?status]}
```

The `${..}` syntax is Flora's *reified formula* construct, which allows us to treat a formula as a term (or object). Note that only the variables `?nationName` and `?status` actually represent data properties of the ontology (and thus are elements that could potentially be shared with a data requester). The other variables are required to unambiguously specify how these two data properties are to be connected.

E. Reasoning about Data Sharing

The concept of Request Formula is used in the privacy policy framework to specify (i) what data is requested in a specific request, and (ii) what data is affected (allowed or disallowed) by a policy rule. However, it does not suffice to check whether the requested RF *exactly* matches the RF in the policy rules. In most cases, a policy author intents to specify policies that capture a large variety of property combinations. Expecting separate specifications for every relevant combination of properties is infeasible, as it puts a heavy burden on the policy author. Instead, a more flexible relation between requested and allowed RFs needs to be defined.

Policies allow to specify affected data using the above concept of request formulae. A formal shareability theory has been developed to characterize the sharing implications of specified policy data wrt. individual requests. A predicate `implies_sharing(?polData, ?reqData, ?filterCstr, ?actionCstr)` is used to test whether a policy's data specification `?polData` implies that a request for data `?reqData` is permitted, potentially under certain constraints `?filterCstr` and `?actionCstr`, as explained in detail below. This is a Prolog-style predicate, not using the Flora frame syntax (Flora allows you to mix and match these styles). Roughly speaking, a sharing decision is entailed if the requested data `?reqData` is a subset of the policy's data `?polData`. For negative policies, the sharing implications are reversed, i.e., `implies_sharing(?reqData, ?polData, ?filterCstr, ?actionCstr)`, effectively disallowing requests for all supersets of the data specified in a disallow policy. The intuition behind this reversed implication check is that deny policies always specify a minimal combination of "critical data" that is not to be shared. Consequently, if only subsets of the denied data are requested, the deny policy is not triggered, while a request for any superset contains the critical data and thus should not be shared. As a simplified example, consider the combination of persons' first and last names: this combination has quite a high chance of identifying a person, while usually neither first nor last name can reliably identify a single person. Thus, if a policy forbids sharing of first and last names, requests for either only first or last names would not trigger this deny policy, while requests for any supersets such as, say, first name, last name, and birth date should obviously be denied.

A set of background knowledge axioms allows to infer additional additional sharing relations from what has been explicitly specified, such as implying *inverse* properties and *subclasses*. Shareability reasoning with a background theory thus allows a policy to generalize over many different requests.

```
@!{P1_CebuNationSharesDiseaseStatus}
CBUNationPA [ decide_sa(?req, ?dd) ] :-
  ?decision = Allow,
  ?polData = ${
    ?nation : Nation [
      name -> ?nationName, citizen -> ?citizen ],
    ?citizen : Person [ medicalInformation -> ?medInfo ],
    ?medInfo : MedicalInfo [ diseaseStatus -> ?status ] },
  ?req.requester : CareProvider,
  implies_sharing(?polData,?req.requestFormula,
    ?filterCstr, ?actionCstr),
  ?id = "P1_CebuNationSharesDiseaseStatus"\^^string,
  ?descr = "Cebu Nation shares disease status"\^^string,
  ?priority = 1,
  ?dd = decisionDetails(?decision, ?pa, ?req,
    ?filterCstr, ?actionCstr, ?startTime, ?endTime,
    ?id, ?description, ?priority).
```

Fig. 3. Policy P_1 CebuNationSharesDiseaseStatus allows to share disease status information of Cebu citizens.

Automated reasoning technology allows us to realize the entailment checks in the policy decision engine. A detailed characterization of the shareability theory can be found in [1].

F. Policy Rules

The actual policy rules in Flora are specified as *stand-alone* predicates, represented with the suffix `_sa`, as shown in Figure 3. These stand-alone decision predicates give information how a single policy rule would decide on a given request *in isolation*, i.e., without considering the interplay with other rules, which could potentially retract decisions of a given policy due to overrides. The rule head specifies the policy rule with the parameters introduced in Section III-C.

Using `P1_CebuNationSharesDiseaseStatus` shown in Figure 3 as an example, it specifies that the policy authority Cebu Nation PA (`CBUNationPA`) allows sharing for a given request `?req` with additional information about the decision specified in the decision details `?dd`. The rule body specifies under which circumstances the rule's head is true, i.e., when the policy rule is triggered. The first rule of the body specifies that the sharing decision is `Allow`. Next, the data allowed by the policy is specified in `?polData`, as a request formula. This policy uses the RF introduced in Section III-D and specifies a connection between nation names and the disease status of the nations' citizens. Following the data specification, the targeted requesters are specified as all instances of the class `CareProvider` (an object-oriented dot notation can be used to conveniently access the properties of the request object created by the rule shown in Figure 2). Then, it is checked whether the requested data is shareable according to this policy, using the `implies_sharing` predicate described in Section III-E and returns potential constraints for the sharing decision (discussed in more detail in the next section). Next, some meta information about the policy (namely id, description, and priority) is specified and finally, all information about the policy's decision is used to create a `DecisionDetails` object. It should be noted that some variables for the decision details (namely `?filterCstr` and `?actionCstr` pertaining to constraint specifications and `?startTime` and `?endTime` pertaining to timing information) are not instantiated in this policy and thus could be replaced by anonymous "don't care" variables, denoted by `?` in Flora. We only present this rule with

```
@!{FinalDecision}
?pa1 [ decide(?req, ?dd1) ] :-
  ?pa1 [ decide_sa(?req1, ?dd1) ],
  \+ ( ?pa2 [ decide_sa(?req, ?dd2) ],
    ?dd1.decision != ?dd2.decision,
     overrides(?pa2, ?dd2, ?pa1, ?dd1, ?req) ).

@!{PriorityOverride}
overrides(?pa1, ?dd1, ?pa1, ?dd2, ?req) :-
  ?dd1.priority > ?dd2.priority.
```

Fig. 4. Override rules

named variables here to help an easier understanding of the meaning of different parameters. We will discuss the meaning of the constraint variables in detail below. Policy timing is not relevant for the scope of this paper, and thus we always use uninstantiated timing variables here—a thorough treatment of policy timing considerations can be found in [10].

G. Policy Overrides

As stated before, the policy rules introduced in the previous section only define *stand-alone* decisions that do not take the interplay of different policies into account yet. To determine final sharing decisions based on a set of stand-alone policies, the privacy policy framework provides an overriding mechanism, which determines the final decision based on various overriding criteria. The general overriding rule is `FinalDecision`, shown in Figure 4. It derives a final decision (note that the allow predicate in the rule's head now does not have the `_sa` suffix), if (i) there is a stand-alone policy rule that provides a corresponding decision, and (ii) there is no stand-alone rule that makes an opposing decision *and* overrides the allow decision (\+ is Flora's operator for Prolog negation). Overriding criteria can be defined through the auxiliary predicate `overrides`. Figure 4 shows the definition of a simple overriding criterion based on priorities of different decisions from the same policy authority. Additional definitions are possible to tailor the policy overrides to specific use cases (e.g., based on a hierarchy of policy authorities).

The overriding mechanism implemented purely through the rule `FinalDecision` exhibits the problem outlined in the introduction: it makes sure that the stand-alone decision with the highest precedence is turned into the final decision, but all overridden decisions are completely suppressed, thus yielding potentially under-specified decisions. In the following sections we show how to extend the policy framework with partial overrides to overcome this problem.

IV. Extending the Constraint Framework

Due to the architectural separation between the Policy Decision Point (PDP) and the Policy Enforcement Point (PEP), the PDP does not have access to any actual data and is therefore unable to enforce any constraints on its own. Instead, any constraints attached to policy decisions are forwarded to the PEP for enforcement. Thus, the PDP does not even need to know the semantics of constraints; from the PDP's perspective, the only restriction on constraint specifications is that they need to be part of a shared vocabulary between PDP and PEP.

Examples of defined constraints in our existing use-cases include among others equality and inequality constraints (e.g., "share if name equals", or "share if age greater then"), geographical constraints (e.g., "share for entities in a specific region"), aggregation constraints (e.g., "only share aggregated counts of disease states"), noise constraints (e.g., "add two nautical miles of noise to shared locations"), and differential privacy constraints (i.e., "only share DP-protected histograms").

Constraint specifications in our framework consist of several parameters:

- The constraint subject: this determines whether a constraint is "triggered". If the constraint subject is part of a request, the corresponding decision will include the specified constraint.
- The constraint object: this can be an arbitrary entity in the CDM.
- A Formula representing a path through the CDM to connect the constraint subject to the constraint object.
- The actual constraint predicate specifying how the constraint object is restricted.

To illustrate this, assume that we want to restrict our previous request formula `RF_diseasestatus` to address only the medical status of persons that are less than 18 years old. In the previous implementation, this would have been specified through the following addition to `RF_diseasestatus`:

```
?medInfo [ constraints -> ?constr ]
```

and then define the actual constraint as

```
?eighteenYrs \is 18*365*24*60*60,
?req.requestTime [ subtractTime(?eighteenYrs) -> ?thres ],
?constr = [ ${ ?medInfo [ person -> ?citizen ],
    ?citizen [ birthDate -> ?birthDate ],
    GreaterThan(?birthDate, ?thres) }  ]
```

i.e., `?medInfo` is the constraint subject, and the keyword `constraints` signals that the specification defined in `?constr` (specifying a path from `?medInfo` to the constraint's object `?birthDate`, and specifying that the value of the object has to be greater than eighteen years before the current request time) has to be returned whenever medical information is requested.

As a result, if this constraint was added to the policy specification in 3, a request including `?medInfo` would trigger this constraint and return the age constraint, while a request for, say, only nation names would not touch the constraint subject and thus still return an unconstrained decision.

A. Additional Requirements for Constraint Specifications

The overall goal of this work is to provide additional mechanisms to not suppress overridden decisions completely, but instead—if an overriding decision is constrained—automatically derive a complementary set of constraints to attach to the overridden decision. As a result, we get *partial overrides* where the higher-precedence decision applies to the subset of the data that matches the constraints, while the lower-precedence decision still applies to the complementary set of data. I.e., if for a given request we have a pair of applicable policies

- P_1: decision D_1 with constraint C_1,
- P_2: decision D_2 with constraint C_2,
- P_2 overrides P_1,

```
@!{P2_CebuNationForbidsMinorDiseaseStatus}
CBUNationPA [ decide_sa(?req, ?dd) ] :-
  ?decision = Deny,
  ?polData = ${
    ?nation : Nation [
      name -> ?nationName, citizen -> ?citizen ],
    ?citizen : Person [ medicalInformation -> ?medInfo ],
    ?medInfo : MedicalInfo [ diseaseStatus -> ?status ],
    ?medInfo [ filterConstraints = ?fConstr ] },
  ?eighteenYrs \is 18*365*24*60*60,
  ?req.requestTime [subtractTime(?eighteenYrs) -> ?thres],
  ?fConstr = ( ${ ?medInfo [ person -> ?citizen ],
       ?citizen [ birthDate -> ?birthDate ] },
     ${ GreaterThan(?Birthdate, ?thres) } )
  ?req.requester : CareProvider,
  implies_sharing(?req.requestFormula, ?polData,
    ?filterCstr, ?actionCstr),
  ?id = "P2_CebuNationForbidsMinorDiseaseStatus"\^^string,
  ?descr = "Cebu Nation denies data of minors"\^^string,
  ?priority = 1,
  ?dd = decisionDetails(?decision, ?pa, ?req,
    ?filterCstr, ?actionCstr, ?startTime, ?endTime,
    ?id, ?description, ?priority).
```

Fig. 5. Policy P_2 CebuNationForbidsMinorDiseaseStatus denies sharing disease status information of minor Cebu citizens.

the goal is to create a pair of decisions

- D_2 with constraint C_2,
- D_1 with constraint $C_1 \wedge \neg C_2$.

In order to be able to negate constraints, we can no longer treat all arbitrary constraint specifications as atomic objects. First of all, a closer inspections of possible constraints shows that they can be categorized into two different types of constraints:

a) Filter Constraints: restrict the decisions to subsets of the available data. Examples of filter constraints are equality and inequality constraints or regional constraints. If filter constraints are involved in a decision override, they should be negated to identify the complementary part of the data set that is not captured by the overriding decision. For example, in our introductory example we have a policy *"share medical data"* and another, overriding policy with a filter constraint *"don't share medical data if age is less than 18"*. In this case, the filter constraint should be negated to create the complementary decision *"share medical data if age is* not *less than 18"*.

b) Action Constraints: constrain the decisions to apply some post-processing actions to the affected data. Examples of action constraints are "add differential privacy", "aggregate values", or "add noise to values". Constraints of this type do not partition the data, and thus they should not be taken into consideration when determining partial overrides.

To distinguish these two types of constraints in the policy specification, we replace the keyword `constraints` with two new keywords `filterConstraints` and `actionConstraints`. This allows for a clear separation and different treatments of these two different types of constraints.

Moreover, the previous approach of specifying constraints treated both the CDM path to identify the constraint object and the actual constraint term together as a single object. Of course, the path specification cannot be negated when negating a filter constraint, and thus the constraint specification needs to be separated into an explicit representation of pairs *(PathToConstraintObject, ActualConstraintSpecification)*. This allows us to apply negation only to the actual constraint specification, while leaving the path information unmodified.

Finally, in our previous approach multiple constraints for a policy were specified simply as a list of constraint specifications, and it was implicitly assumed that this list represents a conjunction of constraints. For our purpose of enabling partial overrides, only allowing for conjunctions and atomic negation of constraint predicates is not expressive enough: as soon as a conjunction of more than one constraint needs to be negated, additional logical connectives are required. To overcome this problem, the actual constraint specification can now take constraint predicates combined into arbitrary propositional formula.

To illustrate how this extended policy specification can be used, consider the policy specified in Figure 5. This policy represents the implementation of policy P_2, as informally introduced in Section I to forbid sharing medical data of minors. The significant differences (marked in bold) compared to the allow policy shown in Figure 3 are: (i) the decision changed from `Allow` to `Deny`, (ii) a filter constraint is added to ensure that only disease status data of citizens over the age of eighteen is shared, now specified as a pair of reified formulae to specify the path to the constraint object, and the actual constraint formula, respectively, and (iii) the direction of the sharing implication check is flipped, as explained in Section III-E.

V. Handling Partially Overridden Decisions

With the extended constraint specifications outlined in the previous section, we are now able to define additional rules to handle partially overridden decisions.

A. Adding a Rule for Partially Overridden Decisions

The main rule for these additional decisions is shown in Figure 6: To determine whether a partially overridden rule is turned into another final decision (note that the head again doesn't have the `_sa` suffix), the rule collects all stand-alone decisions (line 3), and then eliminates from these the ones that are *completely overridden* by another decision (lines 4 to 6). A complete override is identified by a decision that does not specify filter constraints (i.e., empty entries for path specification and constraint formula, as checked in line 5). In the case of complete overrides, there is no complementary data set left, and thus those situations are excluded from this rule. Next, the rule collects the filter constraints from other decisions that override the current decision `?this_dd` (lines 7-13), using Flora's `setof{}` construct. The overriding decisions are additionally restricted to *direct overrides* (lines 11-13) by ensuring that no other decisions exist in the overriding hierarchy between the collected overriding decisions `?o_dd1` and the overridden decision `?this_dd`. This additional restriction ensures that in the presence of multiple override levels, partially overridden decisions are handled level by level, thus creating a hierarchy of partially overridden decisions. We will illustrate this with an example below. Flora's `setof{}` construct woule return empty sets if no decisions are found that match the overriding criteria, and thus line 14 ensures that we are only handling cases where constrained overrides

```
1 @!{PartiallyOverriddenFinalDecision}
2 ?pa [ decide(?req, ?dd) ] :-
3   ?pa [ decide_sa(?req, ?this_dd) ],
4   \+ (?co_pa [ decide_sa(?req, ?co_dd) ],
5     ?co_dd [ filterConstraints -> ([],[]) ],
6     overrides(?co_pa2, ?co_dd, ?pa, ?this_dd, ?req) ),
7   ?cstr_set = setof{ ?o_f_cstr |
8     ?o_pa1 [ decide(?req, ?o_dd1) ],
9     ?o_dd1 [ filterConstraints -> ?o_f_cstr ],
10    overrides(?o_pa1, ?o_dd1, ?pa, ?this_dd, ?req),
11    \+ ( ?o_pa2 [ decide_sa(?req, ?o_dd2) ],
12      overrides(?o_pa1, ?o_dd1, ?o_pa2, ?o_dd2, ?req),
13      overrides(?o_pa2, ?o_dd2, ?pa, ?this_dd, ?req) )},
14  ?cstr_set != [],
15  combineConstraints(?cstr_set,
16    (?path1, ?overriding_cstr)),
17  negate(?overriding_cstr, ?negated_cstr),
18  ?dd1.filterConstraints = (?path2, ?this_f_cstr),
19  conj(?path1, ?path2, ?path),
20  conj(?negated_cstr, ?this_f_cstr, ?f_cstr),
21  ?dd = setFilterConstraints(?dd1, (?path, ??f_cstr)).
```

Fig. 6. Rule to handle partially overridden decisions

```
@!{CombineConstraintsBaseRule}
combineConstraints([],([],[])).

@!{CombineConstraintsRule}
combineConstraints([?h|?t], (?outPath, ?outFormula)) :-
  combineConstraints(?t,?tOut),
  ?tOut = (?tPath, ?tFormula),
  ?h = (?hPath, ?hFormula),
  conj(?tPath, ?hPath, ?outPath),
  disj(?tFormula, ?hFormula, ?outFormula).
```

Fig. 7. Rules to combine collected constraints into reified formulae

actually exists by checking that the constraint set is nonempty. Then, line 15 turns the collected sets of constraints into reified propositional formulae as described below. Line 17 negates the collected filter constraints and finally, lines 19-21 combine the negated filter constraints and potentially existing additional filter constraints from this decision and the result is then turned into a new `DecisionDetails` object that sets the filter constraints to the new filter constraint object and copies all other parameters from the previous standalone decision.

B. Additional Helper Rules

The rules `combineConstraints` to combine all collected overriding constraints are shown in Figure 7: they take the list of collected pairs of formulae for CDM paths and constraints, and recursively join them into two reified formulae; one for all path specifications and one for all constraint specifications. All path formulae are collected in a single conjunctive formula to provide a unified set of path information. The constraint formulae from different overriding decisions are collected into a single disjunctive constraint formula—in order to override the current decision, the constraints need to satisfy the constraints of (at least) one of the overriding decisions, i.e., one of the disjuncts of this formula. The helper rules `conj` and `disj` (not shown here for brevity) combine the first two arguments into a reified conjunctive resp. disjunctive formula.

The rules to negate reified formulae are shown in Figure 8. The *meta decomposition* operator `=..` decomposes terms into individual elements. This allows us to restrict the negation rules to specific terms: the first element of a decomposed atomic term is `hilog(?,?)`, for negated terms

```
@!{AtomicNegationRule}
negate(?in, ?out) :-
  ?in =..[hilog(?,?)|?],
  ?out =..
    [negation(neg)|[?in]].

@!{DoubleNegationRule}
negate(?in, ?out) :-
  ?in =..
    [negation(neg)|[?out]].

@!{DeMorganConjunctionRule}
negate(?in, ?out) :-
  ?in = (?i1, ?i2),
  negate(?i1,?o1),
  negate(?i2,?o2),
  ?out = (?o1; ?o2).

@!{DeMorganDisjunctionRule}
negate(?in; ?out)
  ?in = (?i1, ?i2),
  negate(?i1,?o1),
  negate(?i2,?o2),
  ?out = (?o1, ?o2)
```

Fig. 8. Negation rules for reified formulae

```
@!{SetFilterConstraints}
?dd_new : DecisionDetails [
  ?propName -> ?propVal, filterConstraints -> ?fCstr ] :-
    isnonvar{?dd_new},
    ?dd_new = setFilterConstraints(?dd, ?fCstr),
    ?dd [ ?propName -> ?propVal ],
    ?propName != filterConstraints.
```

Fig. 9. Setting filter constraint of a decision

it is `negation(neg)` and for conjunctions and disjunctions it is `logic(and)` resp. `logic(or)`[4]. Using meta decomposition operators, the first rule `AtomicNegationRule` negates atomic terms by prepending the negation operator to the output term. If negative literals are to be negated again, the next rule `DoubleNegationRule` removes the existing negation operator from the output instead of prepending another one, so that negation rules will never have multiple levels of negation. Finally, the last two rules implement DeMorgan's laws by turning a negated conjunction (disjunction) into a disjunction (conjunction) of its negated elements. Together, these rules can negate arbitrary propositional formulae and by using DeMorgan's laws to push negation operators inside, the resulting formulae will have only atomic negations, i.e., they will be in *Negation Normal Form (NNF)*. Restricting constraint formulae to NNF allows for a simpler representation of constraint formulae to the Policy Enforcement Point.

Finally, Figure 9 shows how a functional term is used to create a new `DecisionDetails` object that sets the `filterConstraints` property to the specified parameter and copies all other properties from the provided existing `DecisionDetails` object.

C. Examples of Partially Overridden Decisions

Now, with all of these additional rules in place, we can repeat the request from a care provider to the Cebu Nation PA for disease status data of its citizens. The result will now be a set of complementary decisions covering the decisions for *all* of Cebu Nation's citizens: As before, policy `P2_CebuNationForbidsMinorDiseaseStatus` will trigger the rule `FinalDecision` and return the decision D_1*: "deny if age is under 18"*. The overridden rule `P1_CebuNationSharesDiseaseStatus` on the other hand is now handled by the new rule `PartiallyOverriddenFinalDecision`: (i) the stand-alone

[4]Meta decomposition is not required to identify conjunctions and disjunctions (as shown in the implementation of DeMorgan's laws), but showing how they are represented in decomposed terms helps to illustrate that the first negation rule only applies to atomic terms.

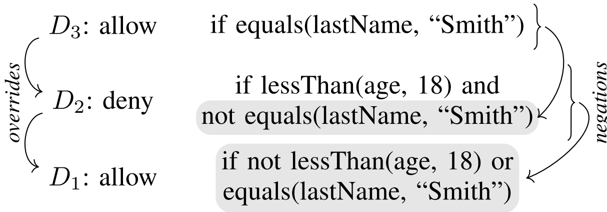

Fig. 10. A hierarchy of partial override decisions

decision from `P1_CebuNationSharesDiseaseStatus` is obtained, (ii) the check for unconstrained overrides passes because there is only one override from `CebuNationForbidsMinorDiseaseStatus` which specifies a filter constraint, (iii) that constraint is collected, negated, and combined with the (empty) filter constraints from `P1_CebuNationSharesDiseaseStatus` and (iv) consequently, another decision D_2*: "share if age is not under 18"* is returned to complement the existing deny decision D_1. Together, these decisions ensure that correct decisions are issued for citizens of all ages.

To illustrate how these partial overrides handle a hierarchy of overrides with multiple levels, assume that—just for the sake of example—a third policy is added: P_3*: "share disease status data with care provides if last name equals 'Smith' (priority = 3)"*. The results of again repeating the request for disease status data is schematically depicted in Figure 10: The result of the newly added policy P_3 is the only one that is not overridden and thus the policy that triggers `FinalDecision`. Consequently, its decision D_3 is topmost in the hierarchy and only contains the constraints specified in P_3. Next in the hierarchy follows policy P_2, which is overridden by P_3. Thus the resulting decision D_2 contains both the constraints specified in P_2 as well as the negation of the constraints specified in P_3. Finally, the decision from P_1 (overridden by P_2) is obtained by negating the constraints inferred for D_2.

This example illustrates several key properties of the partial overriding mechanism. The restriction to collect only constraints from direct overrides in `PartiallyOverriddenFinalDecision` ensures that constraints move down the hierarchy in an alternating manner, e.g., the constraint from P_3 first appears as a positive constraint term in D_3, then as a negative term in $D2$, and then finally as a positive term again in D_1. Without this restriction to direct overrides, the partially overridden decision would directly collect the constraint from D_3 and negate it; thus leading to the contradictory requirement that the last name must be at the same time equal and not equal to "Smith".

An inspection of the constraints from D_1 and D_3 shows that decision D_3 is redundant now because all of its applicable situations are already covered by D_1. This is a special case because policy P_1 is unconstrained—if P_1 was also constrained, the resulting decision would contain additional conjuncts and thus not be a superset of D_3's constraints any longer. Having redundant decisions for a policy request is not a problem—after all, all returned decisions are valid, not contradictory, and can be enforced correctly by the PDP—but this shows some potential for future optimizations of the decision process by filtering out such redundant results.

VI. Conclusion

In this work, we have shown how we can extend overriding mechanisms for policy decisions to correctly handle partially overridden decisions. This provides a new ability to identify for both overriding and overridden decisions the respective subsets of the affected data. This ability provides means for a much finer-grained specification of policies and overriding hierarchies that allow for a large variety of customized policy decisions tailored to specific properties of the protected data set, which in turn enables our policy framework to be applied to a broader spectrum of use cases.

While the mechanisms proposed in this work generate valid sets of policy decisions to be enforced by the Policy Enforcement Point, we plan to optimize the handling of constraints within the Policy Decision Point: As outlined in the previous section, a hierarchy of partially overridden decisions can yield some redundant decisions—identifying such redundancies while reasoning about sharing decisions can reduce both the number of constraints that need to be taken into account for further reasoning, and the number of final decisions to be returned to the PEP. Moreover, decision constraints can potentially be further simplified by considering the semantics of constraint predicates and the relations between them. For instance, if a constraint formula contains two different "greater than" constraints for the same value, they can be reduced to a single value by dropping the one with the lower threshold (for conjunctions) resp. the higher one (for disjunctions). This can also lead to the identification of tautologies (e.g., "greater than x OR less or equal than x) and contradictions (e.g., "greater than x AND less than x), which in turn could be simplified to unconstrained decisions or be removed from the result completely.

References

[1] K. Martiny, D. Elenius, and G. Denker, "Protecting privacy with a declarative policy framework," in *Proceedings of the 12th IEEE International Conference on Semantic Computing*, IEEE ICSC 2018, 2018.

[2] L. Briesemeister, W. Gustafson, G. Denker, A. Martin, K. Martiny, R. Moore, D. Pavlovic, and M. St. John, "Policy creation for enterprise-level data sharing," in *HCI for Cybersecurity, Privacy and Trust*, pp. 249–265, Springer International Publishing, July 2019.

[3] OASIS Standard, "eXtensible Access Control Markup Language (XACML) Version 3.0," January 2013. [Online].

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder specification language," in *Policy 01: Workshop on Policies for Distributed Systems and Networks,*, January 2001.

[5] P. Ashley, S. Hafa, G. Karjoth, C. Powers, and M. Schunter, "Enterprise policy authorization language," 2003. [Online].

[6] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Policy 03: 4th International Workshop on Policies for Distributed Systems and Networks,*, June 2003.

[7] A. Uszok, J. M. Bradshaw, and R. Jeffers, "KAoS: A policy and domain services framework for grid computing and semantic web services," in *iTrust 2004, Second International Conference on Trust Management*, March 2004.

[8] L. Kagal, C. Hanson, and D. Weitzner, "Using dependency tracking to provide explanations for policy management," in *2008 IEEE Workshop on Policies for Distributed Systems and Networks*, pp. 54–61, June 2008.

[9] M. Becker, C. Fournet, and A. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18/4, pp. 619–665, January 2010.

[10] K. Martiny and G. Denker, "Expiring decisions for stream-based data access in a declarative privacy policy framework," in *Proceedings of the 2nd International Workshop on Multimedia Privacy and Security*, MPS '18, pp. 71–80, ACM, 2018.