Policy merging in ODRL

Gianluca Oldani

Work in progress

Contents

Chapter 1: ODRL

1.1 Il linguaggio

1.1.1 Definizione ed obiettivi

"L' Open Digital Rights Language (ODRL) è un linguaggio per l'espressione di policy che definisce: un modello dell'informazione flessibile ed interoperativo, un vocabolario e un meccanismo di codifica per la rappresentazione delle istruzioni sull'uso di contenuti o servizi"[4].
Il linguaggio si pone all'interno dello scenario applicativo nel quale vi è la necessità di definire:

- quali azioni siano permesse o proibite su una risorsa. Queste regole possono essere imposte da leggi o direttamente dal possessore dell'asset o servizio;
- indicare quali attori interagiscono con le policy definite; in particolare chi può definire le policy e a chi si applicano;
- indicare eventuali limiti riguardanti i permessi ed i divieti espressi;.

Avere un modello standard per definire questi bisogni dà 2 fondamentali vantaggi:

- chi possiede l'asset è in grado di definire in modo chiaro quali siano le azioni che un consumatore può fare, evitanto quindi usi indesiderati;
- chi usufruisce dell'asset conosce in modo preciso quali azioni può compiere, evitando così di infrangere regole o leggi.

ODRL definisce un modello semantico di permessi, divieti ed obblighi, che può essere usato per descrivere le modalità d'uso di un contenuto. In particolare si cerca di definire i concetti chiave per la creazione di policy machine-readable collegate direttamente all'asset al quale sono associate, permettendo all'utente finale di reperire facilmente informazioni sulla risorsa che utilizza. Quest'ultimo requisito è soddisfatto, poiché ODRL è costruito seguendo i *Linked Data principles*[1]:

- Utilizzo di URIs come nomi per le risorse;
- Gli URI sono indirizzi HTTP in modo che le persone possano cercare informazioni sulle risorse;
- L'URI deve fornire informazioni utili sulla risorsa;
- Tra le informazioni, fornire altri URI, in modo che l'utente possa raggiungere altre informazioni.

Nonostante questi principi siano più indicati per un'implementazione graph-based, è possibile anche definire utilizzi che non tengano conto dei link tra le varie informazioni.

1.1.2 Il modello

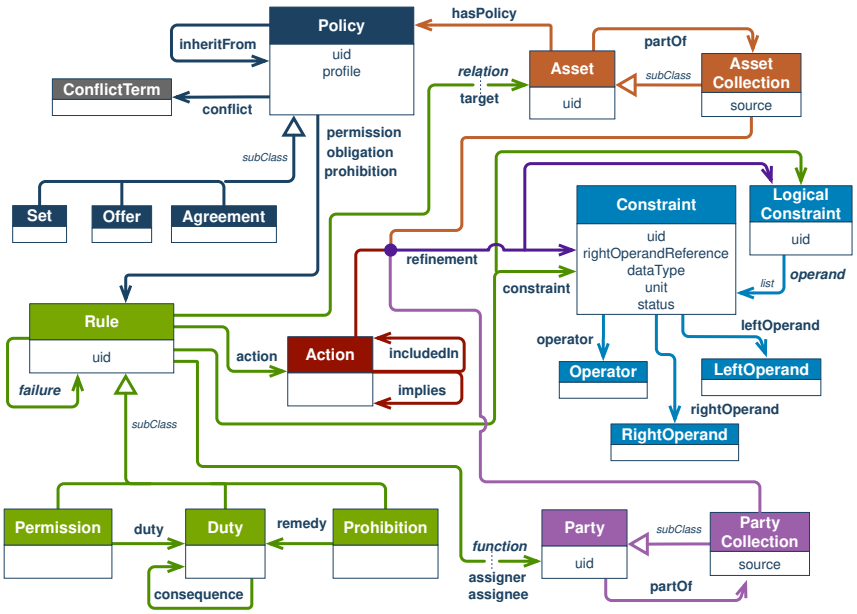

Figure 1.1: Schema del modello ODRL[4]

Come visibile all'interno dello schema in figura 1.1, il modello è basato sulle seguenti entità principali:

- **Policy**: un gruppo di una o più regole;
- **Regola**: concetto astratto che racchiude le caratteristiche comuni di **permesso**, **divieto**, **doveri**;
- **Asset**: risorsa o collezione di risorse soggette a regole;

- **Azione**: operazione su un asset;
- **Party**: entità o insieme di entità con un certo ruolo in una regola;
- **Limiti**: espressione logica o booleana imposta su azioni, party, asset o regole.

Vocabolari

"L' *ODRL Vocabulary and Expression* descrive i termini usati dalle policy ODRL e come codificarle"[3]. All'interno di ODRL, i vocabolari utilizzati per definire i termini all'interno delle policy vengono detti **profili**, i quali possono essere usati per definire termini che supportano specifiche applicazioni; all'interno di un profilo è possibile, ad esempio, fornire le specifiche riguardanti nuove sottoclassi di termini già presenti nei vocabolari standard di ODRL. I 2 vocabolari principali definiti sono:

- **ODRL Core Vocabulary**: rappresenta la minima espressione di policy supportata;
- **ODRL Common Vocabulary**: arricchisce il vocabolario precedente con un gruppo di azioni generiche, nuove sottoclassi per le policy, ruoli per i party e relazioni tra gli asset.

Una delle principali differenze tra i due vocabolari, la si ha all'interno delle **azioni** che possono essere indicate: nel primo caso si hanno a disposizione solamente 2 azioni **use** e **transfer**, nel secondo caso queste 2 azioni vengono estese da diverse azioni figlie, come mostrato in figura 1.2

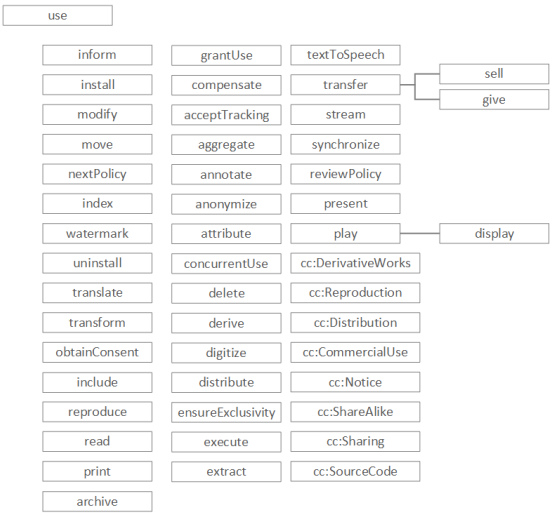

Figure 1.2: Tutte le azioni mostrate sono figlie di use, ad eccezioni di trasfer e le sue sottoazioni[5]

Policy

Come definito nel modello presente nella sezione 1.1.2, una policy è un gruppo non vuoto di **regole** e, quindi, di **permessi**, **divieti** o **obblighi**. Una policy deve soddisfare i seguenti requisiti:

- deve avere un identificativo univoco, detto **uid**;
- deve avere almeno una regola;
- può specificare un profilo, obbligatorio se non si usa il Core Vocabulary mostrato nella sezione 1.1.2;
- può specificare una policy da cui eredita le proprietà;
- può specificare una strategia per la risoluzione dei conflitti.

Come visibile dalla figura 1.1, una policy ha 3 prossibili sottoclassi:

- **Set**: un insieme di regole che hanno effetto;
- **Agreement**: regole concesse ad una entità assegnataria da una assegnatrice;
- **Offer**: proposta di una regola da parte di un assegnatore.

Di seguito un esempio di policy definita mediante ODRL:

```
{
 "@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Set",
 "uid": "http://example.com/policy:1010",
 "permission": [{
   "target": "http://example.com/asset:9898.movie",
   "action": "use"
 }]
}
```

Listing 1.1: Policy con sottoclasse **Set**

La policy mostrata nel listing 1.1 presenta i campi:

- @type: serve ad indicarne la sottoclasse;
- @context : serve ad indicare che il file deve essere conforme ad ODRL, rappresentato dall'URL da cui si può ottenere l'ODRL Common Vocabulary[2];

- nel contesto sono presenti altri link per le altre proprietà, ad esempio quello per la descrizione di *Set* e per *use*;
- non usando termini fuori dai 2 vocabolari principali, non necessita la definizione di un profilo;
- l'id univoco è rappresentato da un URL che porta alle informazioni relative la risorsa.

Asset

Come definito nel modello presente nella sezione 1.1.2, un asset è una risorsa o una collezione di risorse soggette a regole. Un asset può essere una qualunque risorsa identificabile. Ha **AssetCollection** come sottoclasse, la quale rappresenta una collezione di asset. La classe asset può avere:

- un identificativo univoco, il quale può essere omesso se l'asset è fornito direttamente con la policy; le specifiche di ODRL, pur supportando questo caso d'uso, sconsigliano questa pratica;
- una o più proprietà denominate **partOf**: identifica le collezioni di cui fa parte l'asset, il quale può essere a sua volta una collezione.

Esempio di utilizzo in una regola:

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Offer",
 "uid": "http://example.com/policy:3333",
 "profile": "http://example.com/odrl:profile:02",
 "permission": [{
        "target": "http://example.com/asset:3333",
        "action": "display",
        "assigner": "http://example.com/party:0001"
 }]
}
```

Listing 1.2: Utilizzo di asset nella proprietà **target** di una regola

La sottoclasse **AssetCollection** può avere i seguenti 2 campi aggiuntivi:

- **source**: sostituisce il campo **uid** nelle classe AssetCollection all'interno di un **refinement**;

- una o più **refinement**: limiti riguardanti la collezione che identificano solamente un sottogruppo di asset al suo interno.

Esempio di utilizzo della proprietà **partOf**:

```
{
        "@type": "dc:Document",
        "@id": "http://example.com/asset:111.doc",
        "dc:title": "Annual Report",
        ...
        "odrl:partOf": "http://example.com/archive1011",
        ...
}
```

Listing 1.3: L'asset definito è parte del target presente nel listing 1.2

In quest'ultimo esempio si ha che l'asset con id "`http://example.com/asset:111.doc`" è definito come parte della collezione "`http://example.com/archive1011`". Questa definizione ha come effetto l'applicarsi della policy presente nel listing 1.2 anche all'asset che fa parte della collezione.

Party

Come definito nel modello presente nella sezione 1.1.2, un party è una entità o una collezione di entità con una funzione determinata in una regola. Un party può essere un qualunque soggetto con un ruolo attivo nelle regole o che produce un effetto specifico, ad esempio controlla che le azione relative ad un dovere vengano effettuate. Ha **PartyCollection** come sottoclasse, la quale rappresenta una collezione di entità. La classe party può avere:

- un identificativo univoco, il quale può essere omesso se è possibile definire in altro modo l'entità; le specifiche di ODRL, pur supportando questo caso d'uso, sconsigliano questa pratica;
- una o più proprietà denominate **partOf**: identifica le collezioni di cui fa parte l'entità, la quale può essere a sua volta una collezione.

Esempio di utilizzo in una regola:

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Agreement",
"uid": "http://example.com/policy:8888",
```

```
"profile": "http://example.com/odrl:profile:04",
"permission": [{
"target": "http://example.com/music/1999.mp3",
"assigner": "http://example.com/org/sony-music",
"assignee": "http://example.com/people/billie",
"action": "play"
}]
}
```

Listing 1.4: Utilizzo di party nelle proprietà **assigner** ed **assignee** di una regola

La sottoclasse **PartyCollection** può avere i seguenti 2 campi aggiuntivi:

- **source**: sostituisce il campo **uid** nelle classe PartyCollection all'interno di un **refinement**;
- una o più **refinement**: limiti riguardanti la collezione che identificano solamente un sottogruppo di asset al suo interno.

Esempio di utilizzo della proprietà **partOf**:

```
{
"@type": "vcard:Individual",
"@id": "http://example.com/person/murphy",
"vcard:fn": "Murphy",
"vcard:hasEmail": "murphy@example.com",
...
"odrl:partOf": "http://example.com/team/A",
...
}
```

Listing 1.5: L'entità definita è parte di una PartyCollection

In quest'ultimo esempio si ha che l'entità con id "`http://example.com/person/murphy`" è definita come parte della collezione "`http://example.com/team/A`". Questa definizione ha come effetto l'affidare le funzioni di quest'ultima collezione anche alla singola entità che ne fa parte.

Action

Come definito nel modello presente nella sezione 1.1.2, **action** è una classe che rappresenta un'operazione che può essere esercitata su un asset, al quale viene associata mediante la propietà **action** di una regola. Nell' ODRL Core Vocabulary sono presenti 2 azioni principali:

- use: un qualsiasi utilizzo dell'asset;
- transfer: una qualsiasi azione che preveda il trasferimento di proprietà dell'asset;

Un'azione può avere le seguenti proprietà:

- refinement: raffinamenti semantici sull'azione, come ad esempio l'ammontare di un pagamento, il luogo nel quale l'azione può essere eseguita o il tempo massimo di esecuzione;
- includedIn: esprime l'azione padre; la conseguenza di questa dichiarazione risulta essere che tutte le regole applicate all'azione padre, devono valere anche per l'azione figlia;
- implies: esprime un'azione che non deve essere vietata per permettere l'azione con questa proprietà, ma le 2 azioni non hanno una relazione espressa tramite *includedIn*[1].

Come anticipato nel paragrafo 1.1.2 relativo ai profili, l'ORDL Common Vocabulary utilizza la proprietà *includedIn* per aggiungere azioni figlie sia ad *use* che *transfer*, come già mostrato nella figura 1.2.
Di seguito un esempio di azione in una regola:

```
1
2  {
3          "@context": "http://www.w3.org/ns/odrl.jsonld",
4          "@type": "Offer",
5          "uid": "http://example.com/policy:1012",
6          "profile": "http://example.com/odrl:profile:06",
7          "permission": [{
8                  "target": "http://example.com/music:1012"
                     ,
9                  "assigner": "http://example.com/org:abc",
10                 "action": "play"
11         }]
12 }
```

Listing 1.6: L'azione **play** è presente nella proprietà **action** della regola

[1] attualmente né l'ODRL Core Vocabulary né l'ODRL Common Vocabulary presentano azioni con questa proprietà

Constraint e Logical Constraint

Come definito nel modello presente nella sezione 1.1.2, **constraint** è una classe usata per comparare 2 espressioni che non sono constraint a loro volta, utilizzando un operatore relazionale. Rappresentano una limitazione tramite un confronto, la quale può essere soddisfatta o non soddisfatta. La classe presenta le seguenti proprietà:

- uno o nessun identificativo univoco, qualora si volesse riutilizzare l'espressione di confronto definita;
- un **leftOperand**: elemento a sinistra dell'operatore di confronto;
- un sottotitpo **operator**: operatore di confronto;
- uno tra:
  - **rightOperand**: elemento a destra dell'operatore di confronto, identificato direttamente;
  - **rightOperand**: elemento a destra dell'operatore di confronto, identificato con un riferimento;
- uno o nessun **dataType**: definisce il tipo dell'operando di destra;
- una o nessuna **unit**: unità di misura dell'operando di destra;
- uno o nessun **status**: per l'elemento di sinistra.

Oltre ai normali constraint, il modello definisce anche dei logical constraint, ovvero operazioni logiche su altri constraint già definiti. In questo caso le proprietà sono:

- uno o nessun identificativo univoco, qualora si volesse riutilizzare l'espressione di confronto definita;
- un sottotipo di **operand**: operatore logico tra i constraint espressi come lista al suo interno.

Esempi di utilizzi dei constraint:

```
{
 "@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Offer",
 "uid": "http://example.com/policy:6161",
 "profile": "http://example.com/odrl:profile:10",
 "permission": [{
  "target": "http://example.com/document:1234",
```

```
    "assigner": "http://example.com/org:616",
    "action": [{
          "rdf:value": { "@id": "odrl:print" },
          "refinement": [{
                  "leftOperand": "resolution",
                  "operator": "lteq",
                  "rightOperand": { "@value": "1200", "
                     @type": "xsd:integer" },
                  "unit": "http://dbpedia.org/resource/
                     Dots_per_inch"
          }]
          }]
   }]
}
```

Listing 1.7: Constraint su azione: l'azione **print** è permessa solo per risoluzioni minori di 1200 dpi

```
{
 "@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Offer",
 "uid": "http://example.com/policy:88",
 "profile": "http://example.com/odrl:profile:10",
 "permission": [{
  "target": "http://example.com/book/1999",
  "assigner": "http://example.com/org/paisley-park",
  "action": [{
    "rdf:value": { "@id": "odrl:reproduce" },
          "refinement": {
            "xone": {
              "@list": [
                    { "@id": "http://example.com/p:88/C1" },
                    { "@id": "http://example.com/p:88/C2" }
                  ]}
          }
   }]
 }]
}
```

Listing 1.8: Constraint logico su azione: l'azione **reproduce** è permessa solo nella forma di uno dei due constraint listati

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Offer",
"uid": "http://example.com/policy:4444",
"profile": "http://example.com/odrl:profile:11",
"permission": [{
 "assigner": "http://example.com/org88",
 "target": {
        "@type": "AssetCollection",
        "source":  "http://example.com/media-catalogue",
        "refinement": [{
                "leftOperand": "runningTime",
                "operator": "lt",
                "rightOperand": {
                 "@value": "60",
                 "@type": "xsd:integer"
                 },
                "unit": "http://qudt.org/vocab/unit/
                   MinuteTime"
        }]
 },
 "action": "play"
}]
}
```

Listing 1.9: Constraint su asset: l'azione **play** è permessa solo sui target di durata strettamente inferiore a 60 minuti

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Agreement",
"uid": "http://example.com/policy:4444",
"profile": "http://example.com/odrl:profile:12",
"permission": [{
  "target": "http://example.com/myPhotos:BdayParty",
```

```
  "assigner": "http://example.com/user44",
  "assignee": {
     "@type": "PartyCollection",
         "source":  "http://example.com/user44/friends",
         "refinement": [{
                "leftOperand": "foaf:age",
                "operator": "gt",
                "rightOperand": {
                 "@value": "17",
                 "@type": "xsd:integer"
                 }
         }]
        },
  "action": { "@id": "ex:view" }
}]
}
```

Listing 1.10: Constraint su party: l'azione **view** è permessa solo alle entità con età strettamente superiore a 17 anni

Rule

Come definito nel modello presente nella sezione 1.1.2, **rule** è una classe astratta che raccoglie gli aspetti comuni della classi **permission**, **prohibition**, and **duty**. Rappresenta una delle regole all'interno della policy. Presenta le seguenti proprietà:

- una **action**: azione regolamentata;
- una o nessuna **relation**: asset sul quale si applica la regola;
- una, nessuna o più **function**: funzioni che un party può avere all'interno di una regola;
- uno, nessuno o più **constraint** : limiti applicati alla validità della regola;
- uno o nessun identificativo univoco, necessario solo qualora si usare la regola per ereditarne le proprietà;

Le sottoclassi sono così definite:

- **permission**: permette un'azione sull'asset specificato, con tutti i **refinement** di quest'ultima soddisfatti; inoltre l'azione può essere eseguita solo se tutti i limiti

della regola sono soddisfatti e ogni dovere espresso come **duty** è stato rispettato. Un permesso rende obbligatoria la **relation** denominata **target**;

- **prohibition**: vieta un'azione sull'asset specificato, con tutti i **refinement** di quest'ultima soddisfatti; inoltre l'azione non può essere eseguita solo se tutti i limiti della regola sono soddisfatti; se si infrange il divieto, ogni dovere espresso come **remedy** deve essere eseguito. Un divieto rende obbligatoria la **relation** denominata **target**;

- **duty**: obbligo di eseguire un'azione, con tutti i **refinement** di quest'ultima soddisfatti; un dovere è compiuto se tutti i suoi limiti sono soddisfatti e la sua azione effettuata, con tutti i **refinement** definiti. Se un dovere non è stato compiuto, bisogna eseguirne le **consequences**, ovvero altri doveri da compiere.

Esempi di regole all'interno di policy:

```
{
 "@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Offer",
 "uid": "http://example.com/policy:9090",
 "profile": "http://example.com/odrl:profile:07",
 "permission": [{
    "target": "http://example.com/game:9090",
    "assigner": "http://example.com/org:xyz",
    "action": "play",
    "constraint": [{
        "leftOperand": "dateTime",
        "operator": "lteq",
        "rightOperand": { "@value": "2017-12-31", "@type"
           : "xsd:date" }
    }]
  }]
}
```

Listing 1.11: La regola esprime il permesso di eseguire l'azione **play** sul target fino al giorno 2017-12-31 compreso

```
{
 "@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Agreement",
 "uid": "http://example.com/policy:5555",
```

```
 "profile": "http://example.com/odrl:profile:08",
 "prohibition": [{
        "target": "http://example.com/photoAlbum:55",
        "action": "archive",
        "assigner": "http://example.com/MyPix:55",
        "assignee": "http://example.com/assignee:55"
 }]
}
```

Listing 1.12: La regola esprime il divieto di eseguire l'azione **archive** sul target

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Agreement",
"uid": "http://example.com/policy:42",
"profile": "http://example.com/odrl:profile:09",
"obligation": [{
 "assigner": "http://example.com/org:43",
 "assignee": "http://example.com/person:44",
 "action": [{
   "rdf:value": {
     "@id": "odrl:compensate"
   },
   "refinement": [
     {
     "leftOperand": "payAmount",
     "operator": "eq",
     "rightOperand": { "@value": "500.00", "@type": "xsd:
        decimal" },
     "unit": "http://dbpedia.org/resource/Euro"
   }]
  }]
 }]
}
```

Listing 1.13: La regola esprime l'obbligo di eseguire l'azione **compensate**, specificando come **refinement** l'ammontare del pagamento

1.1.3 Problematica trattata

Gestione dei conflitti
La trattazione fatta fino ad ora per ODRL prende in considerazione la definizione di una singola policy per volta. Questo caso non rispecchia però le necessità reali che il linguaggio punta a soddisfare ed, in particolare, non rispecchia i casi d'uso definiti in MOSAICO, dove è naturale che vi sia un numero molto alto di policy definite.
ODRL propone già alcuni elementi per permettere la definizione di più policy in modo agevole, ad esempio:

- è possibile utilizzare la proprietà **inheritFrom** per permettere ad una policy di ereditare tutte le regole definite in un'altra policy;
- è possibile utilizzare la proprietà **conflict** definire una strategia di risoluzione dei conflitti; le strategia attualmente definite nel modello sono le seguenti:
  - **perm**: le regole di tipo **permission** hanno la priorità in caso di conflitto;
  - **prohibit**: le regole di tipo **prohibition** hanno la priorità in caso di conflitto;
  - **invalid**: in caso di conflitto, la policy risulta non valida nella sua interezza; è il valore di default se non definito esplicitamente.

Questo sistema di gestione dei conflitti risulta avere però una problematica fondamentale all'interno dei casi d'uso di MOSAICO: si riferisce a conflitti all'interno di una singola policy o che avvengono in seguito all'utilizzo della proprietà **inheritFrom**; risulta possibile notare che in uno scenario multi-owner, come quello di MOSAICO, l'utilizzo della proprietà **conflict** risulta inadatto, ad esempio:

1. si supponga che un ospedale, indicato come A, voglia rendere disponibili i propri dati di ricerca in un mercato come quello di MOSAICO;
2. risulta plausibile che la collezione di dati di questo ospedale venga inserita in una raccolta di referti medici, i cui owner sono vari ospedali situati in stati diversi;
3. il dato fornito da A viene accompagnato da una policy ODRL, la quale può utilizzare la proprietà **inheritFrom** per inserire nella propria policy le regole richieste da normative europee;
4. oltre a questo, A inserisce nella policy anche regole relative a norme sulla privacy vigenti nel proprio paese, settando la proprietà **conflict** a **prohibit**, al fine di proteggere i dati sensibili degli utenti trattati dalla propria collezione dati.

Se si considera lo scenario mostrato in esempio non multi-owner, ODRL riesce perfettamente ad soddisfare le esigenze dell'owner della collezione di dati, permettendogli di aderire alle normative europee ed, allo stesso tempo, di tutelare la privacy dei propri pazienti in conformità con le normative vigenti nel suo paese.
Nel caso in cui, la collezione di dati medici appartenga a più ospedali, questa soluzione non risulta più adatta, in particolare nel seguente caso:

1. un secondo ospedale, indicato come B, segue il medesimo procedimento ma lascia la proprietà **conflict** uguale ad **invalid**;

2. se le norme dei paesi di A e B non sono le stesse, entrambe le policy vengono invalidate nella loro totalità, anche se in disaccordo solo su un sottoinsieme di regole;

3. questo comportamento risulta deleterio sia per quanto concerne il mercato, poiché scoraggia degli owner a partecipare ad una collezione;

4. il comportamento mostrato risulta dannoso anche per quanto concerne i soggetti dei dati, non più tutelati da regole poiché invalidate.

Una possibile soluzione a questo problema è attuare il merging delle varie policy che targettano la collezione di dati, rendendo l'azione sulle varie regole più granulare; prendendo ad esempio l'ultimo scenario mostrato:

1. avendo due strategie di conflitto differenti, nessuna delle due viene considerata;

2. per ogni regola in conflitto, si tengono solamente le regole di divieto, seguendo l'obiettivo di MOSAICO di tutela della privacy;

3. per ogni regola non in conflitto, la regola viene mantenuta, seguendo l'obiettivo d MOSAICO di portare un dato di qualità ed accessibile;

4. per ogni regola definita come permessa solo in una policy, la si invalida, poiché la seconda policy non si esprime in merito.

Questo procedimento è solo uno dei possibili, in base alle casistiche di conflitto possibili e risulta in linea con i requisiti che MOSAICO punta a soddisfare, poiché a differenza di quanto definito in ODRL:

- porta ad una diminuzione della qualità del dato graduale, anziché ad una invalidazione totale della policy al primo conflitto;

- durante la diminuzione della qualità del dato, preserva comunque il maggior grado di tutela della privacy che le policy definite possono offrire, poiché permette sempre e solo azioni che hanno una regola esplicita che le riguarda.

Controllo inefficiente

Uno dei requisiti fondamentali che MOSAICO punta a soddisfare è l'efficienza. Utilizzando il modello finora espresso, questo requisito viene meno, poiché si è sempre costretti a controllare tutte le regole all'interno di una policy: ciò accade a causa della proprietà *includedIn* di un'azione. Di seguito un esempio della problematica:

- all'interno di una policy sono presenti le seguenti 2 regole:
  - è permessa l'azione **transfer**;
  - non è permessa l'azione **sell**;
- oltre alle regole espresse, ne sono presenti altre;
- come visibile nella figura 1.2, l'azione **transfer** include le azioni **sell** e **give**;
- un utente desidera eseguire l'azione **give** sull'asset interessato dalla policy;
- un eventuale algoritmo di controllo, non può limitarsi a notare che l'azione **transfer** sia permessa, risulta costretto a controllare anche che **give** non sia vietato.

Una possibile soluzione a questa problematica può essere implementata mediante i seguenti passi:

1. invalidare i permessi che comprendono al loro interno un'azione proibita;
2. esprimere i divieti mediante i loro permessi complementari.

Nella casistica mostrata, questa soluzione si andrebbe a tradurre come:

1. il divieto sull'azione **sell**, poiché presente insieme al permesso su **transfer**, produce il permesse sull'effettuare **give**;
2. il permesso sull'azione **transfer** viene rimosso dalla policy.

Come è possibile notare, un eventuale controllo sulle regole prodotte può fermarsi al primo permesso che trova, anziché controllare anche tutti i divieti presenti nella policy.

Chapter 2: Valutazione casistiche

2.1 Casistiche di merging

Casi base

Asset diffrenti

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Set",
"uid": "http://example.com/policy:1010",
"permission": [{
"target": "http://example.com/asset:9898.movie",
"action": "play"
}]
}
```

Listing 2.1: La policy 1010 permette di riprodurre l'asset 9898.movie a chiunque

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Set",
"uid": "http://example.com/policy:1011",
"permission": [{
"target": "http://example.com/asset:1349.mp3",
"action": "play"
}]
}
```

Listing 2.2: La policy 1011 consente la riproduzione dell'asset 1349.mp3 a chiunque

Le due policy mostrate si riferiscono ad asset differenti, farne il merging porta semplicemente ad una policy avente entrambe le *permission* già enunciate.

Rule incompatibili

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Set",
"uid": "http://example.com/policy:1010",
"permission": [{
"target": "http://example.com/asset:9898.movie",
"action": "play"
}]
}
```

Listing 2.3: La policy 1010 consente la riproduzione dell'asset 9898.movie a chiunque

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Set",
"uid": "http://example.com/policy:1011",
"prohibition": [{
"target": "http://example.com/asset:9898.movie",
"action": "distribute"
}]
}
```

Listing 2.4: La policy 1011 proibisce la distribuzione dell'asset 9898.movie a chiunque

Nonostante le due policy si riferiscano allo stesso asset, anche in questo caso, non è possibile combinare le due regole, poiché si riferiscono a domini diversi. Un eventuale merging delle due policy avrebbe 2 regole distinte uguali a quelle di partenza.

Attori designati differenti

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"permission": [{
        "target": "http://example.com/asset:1212",
        "action": "use",
        "assigner": "http://example.com/owner:181",
        "assignee":
           "http://example.com/party:person:billie"
}]
}
```

Listing 2.5: La policy 0001 permette un qualsiasi utilizzo dell'asset 1212 da parte del soggetto Billie

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0002",
"prohibition": [{
"target": "http://example.com/asset:1212",
"action": "play",
"assigner": "http://example.com/owner:181",
"assignee":
"http://example.com/party:person:alice"
}]
}
```

Listing 2.6: La policy 0002 proibisce la riproduzione dell'asset 1212 da parte del soggetto Alice

In questo caso, le regole espresse dalle 2 policy sarebbero in conflitto, poiché "use", permesso dalla prima policy, comprende al suo interno anche "play"[1], proibito dalla seconda. Siccome però le due regole riguardano due soggetti diversi,un eventuale merging delle policy avrebbe 2 regole distinte uguali a quelle di partenza.

[1]Per le dipendenze tra le azioni, si fa riferimento all'ODRL Core Vocabulary

Conflitti senza strategia risolutiva indicata

Nel caso si presentino 2 policy in conflitto, se non viene indicato alcun processo di risoluzione dei conflitti, secondo lo standard ODRL entrambe le policy sono da considerarsi non valide. Attuando policy merging è possibile formulare una terza policy che racchiude entrambe le casistiche.

Conflitti permesso-permesso

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"permission": [{
"target": "http://example.com/asset:1212",
"action": "play"
}]
}
```

Listing 2.7: La policy 0001 permette un qualsiasi utilizzo dell'asset 1212 a chiunque

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0002",
"permission": [{
"target": "http://example.com/asset:1212",
"action": "display"
}]
}
```

Listing 2.8: La policy 0002 permette la riproduzione dell'asset 1212 a chiunque

La seconda policy risulta in conflitto con la prima, poiché quest'ultima permette un numero maggiore di utilizzi. Seguendo la procedura standard, entrambe le 2 policy dovrebbero essere considerate invalidate. Nella realtà dei fatti, facendo policy merging, si otterrebbe solamente la policy più restrittiva, ovvero la seconda. La procedura di policy merging in questione sarebbe quindi:

1. invalidare le policy 0001 e 0002;

2. creare una terza policy 0003, uguale alla 0002(la più restrittiva delle due), con indicato il procedimento di merging;

3. l'indicazione è opportuna per fare il processo inverso al merging qualora una delle 2 policy originarie venga ritirata.

Conflitti divieto-divieto

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"prohibition": [{
"target": "http://example.com/asset:1212",
"action": "play"
}]
}
```

Listing 2.9: La policy 0001 proibisce un qualsiasi utilizzo dell'asset 1212 a chiunque

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0002",
"prohibition": [{
"target": "http://example.com/asset:1212",
"action": "display"
}]
}
```

Listing 2.10: La policy 0002 proibisce la riproduzione dell'asset 1212 a chiunque

Caso duale del precedente; in questo caso una delle due policy proibisce un numero maggiore di utilizzi rispetto all'altra. La procedura da seguire risulta essere:

1. invalidare le policy 0001 e 0002;

2. creare una terza policy 0003, uguale alla 0001(la più ampia delle due), con indicato il procedimento di merging;

3. l'indicazione è opportuna per fare il processo inverso al merging qualora una delle 2 policy originarie venga ritirata.

Conflitti permesso-divieto

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"permission": [{
"target": "http://example.com/asset:1212",
"assigner": "http://example.com/owner:181",
"assignee":
"http://example.com/party:person:alice",
"action": "transfer"
}]
}
```

Listing 2.11: La policy 0001 permette il trasferimento dell'asset 1212 al soggetto Alice

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0002",
"prohibition": [{
"target": "http://example.com/asset:1212",
"assigner": "http://example.com/owner:181",
"assignee":
"http://example.com/party:person:alice",
"action": "sell"
}]
}
```

Listing 2.12: La policy 0002 proibisce la vendita dell'asset 1212 al soggetto Alice

In questo caso, l'azione "transfer", nell'ODRL Core Vocabulary, è inclusa in 2 sotto-azioni: "give" e "sell"; delle 2 sotto-azioni, solamente "sell" è esplicitamente proibita dalla policy 0002, di conseguenza la seguente policy permette solo azioni che soddisfano entrambe le policy precedenti:

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0002",
"permission": [{
"target": "http://example.com/asset:1212",
"assigner": "http://example.com/owner:181",
"assignee":
"http://example.com/party:person:alice",
"action": "give"
}]
}
```

Listing 2.13: La policy 0003 consente al soggetto Alice di cedere l'asset 1212 senza richiedere un compenso e cancellando l'asset dal proprio insieme di dati

Sono necessarie alcune valutazioni su questa casistica:

- il numero di regole all'interno della policy ottenuta per merging non è necessariamente minore rispetto al numero di regole di partenza; tale numero, dipende dal numero di sotto-azioni esistenti e da come è formulato il divieto di partenza;
- è necessario un meccanismo per notificare che il le sotto-azioni presentate in realtà indicano "tutte le sotto-azioni possibili, escluse quelle esplicitamente vietate";
- se "prohibition" e "permission" fossero invertite tra la policy 0001 e la 0002, la policy 0003 risulterebbe una policy non valida.

Conflitti con strategia risolutiva indicata

Le strategie risolutive indicabili all'interno di ODRL utilizzando il Core Vocabulary sono le seguenti:

1. "perm": tutte le "permission" hanno la precedenza sulle "prohibition";
2. "prohibit": tutte le "prohibition" hanno la precedenza sulle "permission";
3. "invalid": qualsiasi conflitto invalida la policy;

La casistica "invalid" è già stata trattata all'interno del paragrafo "Conflitti senza strategia risolutiva indicata"[2.1].

Strategia concorde

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"conflict":{"@type": "perm"},
"permission": [{
"target": "http://example.com/asset:1212",
"action": "use",
"assigner": "http://example.com/owner:181"
}]
}
```

Listing 2.14: La policy 0001 consente qualsiasi utilizzo dell'asset 1212 a chiunque, dando precedenza ai permessi in caso di conflitto

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"conflict":{"@type": "perm"},
"permission": [{
"target": "http://example.com/asset:1212",
"action": "play",
"assigner": "http://example.com/owner:182"
}],
"prohibition": [{
"target": "http://example.com/asset:1212",
"action": "display",
"assigner": "http://example.com/owner:181"
}]
}
```

Listing 2.15: La policy 0002 permette la riproduzione dell'asset 1212 a chiunque, mentre ne vieta la proiezione, dando precedenza ai permessi in caso di conflitto

In questo caso è possibile ricondursi alla procedura indicata nel sottoparagrafo "Conflitti permesso-permesso"[2.1] ignorando tutti i divieti espressi nelle policy. In seguito al merging delle policy si ottiene:

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Policy",
"uid": "http://example.com/policy:0001",
"conflict":{"@type": "perm"},
"permission": [{
"target": "http://example.com/asset:1212",
"action": "play",
"assigner": "http://example.com/owner:182"
}]
}
```

Listing 2.16: La policy 0003 risulta essere uguale ai permessi della policy 0002, più restrittiva

Dando la precedenza ai permessi, rimane comunque da tener in conto che nella policy 0002 il permesso è più ristretto rispetto a quello espresso nella policy 0001.
Dualmente, nel caso di strategia di conflitto concorde di tipo "prohibit", ci si riconduce al caso mostrato nel sottoparagrafo "Conflitti divieto-divieto"[2.1], tenendo conto solo dei divieti.

Strategia discorde

Utilizzando il normale sistema di ereditarietà delle policy di ODRL, una policy contenente 2 strategie di risoluzione dei conflitti dovrebbe essere considerata non valida. Questa casistica si può però ridurre al caso mostrato nel sottoparagrafo "Conflitti permesso-divieto" [2.1]. In questo modo, entrambe le policy risultano rispettate. Un'aggiunta che può essere fatta a questa casistica, rispetto a quella già mostrata, sarebbe considerare solamente i divieti o i permessi della singola policy, in base alla strategia indicata; quest'ultima opzione porterebbe ad un ibrido tra le due soluzioni precedenti.

Restrizioni su regole, asset, party

Possono essere poste delle restrizioni sulla validità di una regola o sulla composizione di una collezione di asset o gruppo di soggetti. Anche queste restrizioni possono essere composte similarmente alle regole a sé stanti.

```
{
"@context": "http://www.w3.org/ns/odrl.jsonld",
"@type": "Offer",
"uid": "http://example.com/policy:6163",
"profile": "http://example.com/odrl:profile:10",
"permission": [{
        "target": "http://example.com/document:1234",
        "assigner": "http://example.com/org:616",
        "action": "distribute",
        "constraint": [{
                "leftOperand": "dateTime",
                "operator": "lt",
                "rightOperand":
                  { "@value": "2018-01-01",
                    "@type": "xsd:date" }
                }]
        }]
}
```

Listing 2.17: Esempio di limite temporale per una regola, può essere composto similarmente ai limiti sull'azione regolata

Problematiche relative alla gestione di questo tipo di merging, non più necessariamente relativo a conflitti, possono essere rilevate nella grande varietà che può assumere:

- limiti di tipo temporale;
- limiti di tipo spaziale;
- limiti di tipo quantitativo(numero di volte che un permesso può essere sfruttato);
- limiti relativi al tipo di dato trattato(dimensione dell'immagine, durata del video).

A causa di questa varietà, la gestione di questo tipo di merging può facilmente esplodere. Vi è anche da contare che un "constraint" può essere ottenuto come operazione in logica booleana di altri "constraint". Questo tipo di merging non va necessariamente a risolvere conflitti nelle policy, ma può rilevare quando un numero eccessivo di constraint va a rendere la collezione di asset inutilizzabile(una policy permette l'uso di un asset solo la mattina, la seconda solo la sera, unendole l'asset risulta inutilizzabile).

Chapter 3: Implementazioni

3.1 Architettura generale

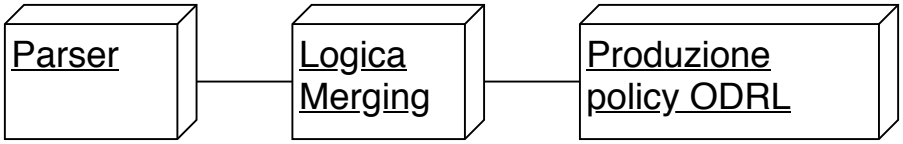

Figure 3.1: Pipeline dei macro componenti della soluzione

Come mostrato nella figura 3.1, l'architettura generale è una semplice pipeline di componenti, poiché l'obiettivo finale risulta attuare trasformazioni sui documenti ODRL in ingresso, per ottenere in uscita un unico documento ODRL che sia:

- il risultato dell'unione o dell'intersezione delle 2 policy in ingresso;
- esprima ogni divieto come il suo permesso complementare più ampio.

Queste 2 operazioni sono attuate all'interno del componente denominato **"Logica Merging"**, il quale si occupa di aggiungere la semantica necessaria alle policy ODRL, per poi produrre il documento finale.
Il primo componente della pipeline si occupa invece di estrarre le informazioni di rilievo dalle policy in ingresso, mentre il componente in coda si occupa di produrre un documento conforme ad ODRL per dare una forma al risultato ottenuto.

3.2 Parser

//TODO: l'implementazione è iniziata, devo ancora scrivere

3.3 Logica Merging

All'interno di questo componente, è presente la porzione di soluzione atta ad eseguire le effettive operazioni di merging delle policy in ingresso.
Le operazioni di merging che si punta a supportare sono di 2 tipi:

- **unione**: utilizzata all'iterno di uno scenario collaborativo, i permessi espressi da una sola delle due policy rimangono validi anche all'interno della policy in output;
- **intersezione**: utilizzata in uno scenario conservativo, un permesso appare nella policy in output solo se espresso da entrambe le policy in ingresso.

3.3.1 Rappresentazione delle Azioni

Tra i due focus dell'implementazione vi sono le **Action**, ovvero gli utilizzi dell'asset regolati dalle policy ODRL. Per rappresentarle, si è utilizzato un enumerativo con la struttura mostrata in figura 3.2:

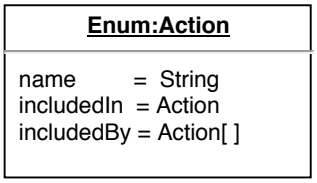

Figure 3.2: Struttura dell'enumerativo Action

All'interno dell'enumerativo è presente il nome che ha all'interno dell' ODRL Common Vocabulary, la lista delle azioni figlio e, se presente, l'azione padre. Tale struttura è assimilabile a quella di un nodo in una struttura dati al albero. L'enumerativo è provvisto dei vari *getter* per il recupero delle informazioni citate.

3.3.2 Rappresentazione delle Regole

Il secondo focus principale dell'implementazione sono le **Rule**, il cui stato all'interno della policy è gestito mediante un albero di regole, sfruttando le relazioni che si creano tra le azioni mediante l'enumerativo definito in figura 3.2. Per ottenere un maggior

incapsulamente delle funzioni, si è deciso di dividere la logica della struttura ad albero in due compenenti, mostrate in figura 3.3 ed esposte di seguito:

- gestore dell'albero delle regole: questa componente si occupa delle azioni più generali relative l'albero delle regole, come ad esempio il recupero di un nodo specifico, il settaggio di stato di un nodo, oppure il recupero dello stato di ogni regola;

- gestore del singolo nodo: questo nodo si occupa dell'effetiva gestione del nodo rappresentante una regola relativa ad un'azione, di conseguenza attua operazioni come il recupero del nodo padre o dei nodi figli; le funzioni principali riguardano però la propagazione di eventuali divieti lungo la gerarchia delle regole, sia verso i nodi figli che verso il nodo padre.

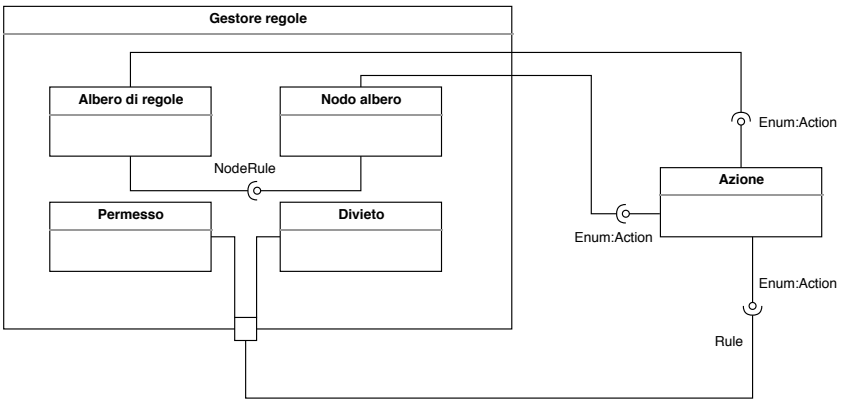

Figure 3.3: Diagramma dei componenti per la rappresentazione delle regole

Oltre ai componenti sopracitati, in figura è possibile vedere le implementazioni dell'interfaccia **Rule**, utilizzata per comunicare con il gestore della rappresentazione delle policy.

Creazione dell'albero

L'albero viene creato dal suo gestore specificandone l'azione radice, tramite l'enumerativo **Action**. Come esposto nella sezione 3.3.1, questo enumerativo permette di recuperare

la lista delle azioni incluse dalla radice e, quindi, permette la creazione di eventuali sottoalberi. La creazione dei nodi figli è demandata al gestore del singolo nodo dell'albero.
Nel caso in cui, ad esempio, si andasse a specificare l'azione **use**, l'albero creato ha la seguente struttura:

- il nodo relativo all'azione **use** è la radice dell'albero;
- tutte le azioni incluse da **use** compongono il livello successivo;
- le azioni che a loro volta ne includono altre, come ad esempio **play**, sono le radici del sottoalbero corrispondente alla loro gerarchia;
- risulta possibile notare come l'albero sia piatto, poiché un'azione generica tende ad avere molti figli, ma le azioni più specifiche tendono poi a non avere figli a loro volta.

Gestione dello stato di un'azione
Ogni nodo dell'albero creato si riferisce ad una azione che può essere regolata e, oltre a tener conto della struttura gerarchica tra le azioni, rappresenta se l'azione si possa svolgere o meno. Gli stati in cui si può trovare un nodo sono 3:

1. **Permitted**: l'azione del nodo è permessa esplicitamente;
2. **Prohibited**: l'azione del nodo è proibita esplicitamente oppure tutte le sue azioni figlie sono nello stato **Prohibited**;
3. **Undefined**: l'azione del nodo non ha regole che la interessano oppure era permessa ma almeno una azione figlia è **Prohibited** o **Undefined**.

I singoli nodi si occupano anche di trasmettere lungo l'albero le conseguenze di un determinato cambio di stato:

1. settaggio a **Permitted**: il cambiamento viene propagato a tutti i nodi figli, a meno che:
    - il nodo non fosse già nello stato **Prohibited**, in questo caso nessun settaggio viene svolto;
    - almeno uno dei nodi figli fosse nello stato **Prohibited**, in questo caso il nodo rimane **Undefined**, mentre i nodi figli sui nodi figli si attua il settaggio a **Permitted**;
    - il caso precedente si attua anche qualora uno dei nodi figli fosse **Undefined**.

2. settaggio a **Prohibited**: il cambiamento viene propagato a tutti i nodi figli, inoltre il nodo padre viene settato come **Undefined** se almeno un altro figlio non è ancora vietato, altrimenti anche il padre è settato come **Prohibited**; il passaggio del nodo padre ad **Undefined** è propagato fino alla radice della gerarchia;
3. settaggio a **Undefined**: questo settaggio avviene solamente come propagazione dai nodi figli nella casistica in cui non è necessario salire ulteriormente di livello.

Di seguito alcuni esempi delle varie casistiche:

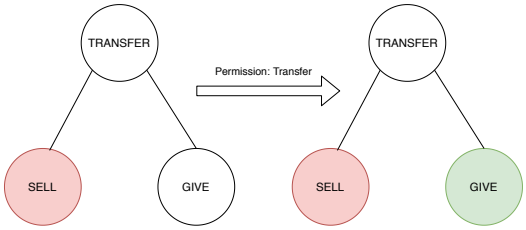

Figure 3.4: Esempio di propagazione di un permesso, con azione figlia proibita

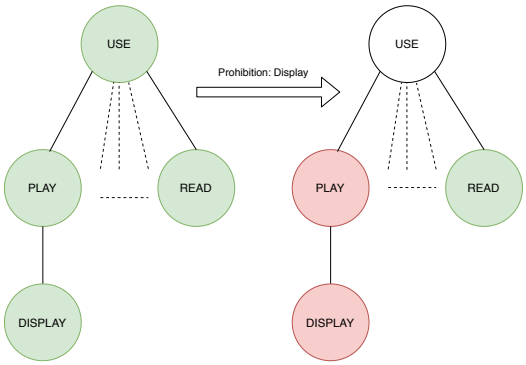

Figure 3.5: Esempio di propagazione verso il padre di un divieto

Recupero di un nodo e del suo stato

L'albero creato non è un albero binario, di conseguenza la struttura dati in sé non supporta una ricerca efficiente dei nodi. La ricerca può essere resa però efficiente grazie alla gerarchia delle **Action** ed, in particolare, sfruttando l'enumerativo in figura 3.2, è possibile ricostruire il percorso che si deve fare da un nodo foglia alla radice per risalire alla gerarchia. Di seguito lo pseudo codice utilizzato per recuperare il nodo relativo ad un'azione:

```
def getActionNode(Action a){
List steps = [];
Action visited = a;
Node exploredNode = Tree.getRoot();

while(visited != None){
  steps.add(visited);
  visited = visited.getFatherAction();
}

// Sempre vero a meno che non si chiami
// il metodo su un albero con radice errata
if(exploredNode.getAction() == steps.getLastStep()){
  steps.popLast();
  while(!(exploredNode.getAction() == a)){
        exploredNode=
           visitedNode.getChildsMap()
             .get(steps.getLastStep);
        steps.popLast();
  }
else{
  // Caso di albero con radice errata
  return null;
}

return exploredNode;

}
```

Listing 3.1: Il caso peggiore per questa ricerca è la profondità della gerarchia delle azioni. Il caso migliore si ha qundo l'azione non è nell'albero, con tempo di risposta costante.

Per il recupero dello stato relativo ad una azione, si usa il medesimo algoritmo mostrato nel listing 3.1 con un'ottimizzazione: qualora una delle azioni più in alto nella gerarchia fosse nello stato **Prohibited** o **Permitted**, si ritorna quello stato e si interrompe la ricerca del nodo, questo poiché:

- se l'azione più in alto nella gerarchia è **Permitted**, necessariamente tutte le

azioni nel suo sottoalbero sono **Permitted**, se anche solo avesse uno stato diverso, la radice del sottoalbero sarebbe **Undefined** o **Prohibited**;

- se l'azione più in alto nella gerarchia è **Prohibited**, anche il suo sottoalbero è **Prohibited**.

Le prestazioni nel caso peggiore rimangono comunque dipendenti dalla profondità della gerarchia delle azioni, mentre ai casi migliori si aggiunge lo scenario in cui l'azione radice dell'albero è **Prohibited** o **Permitted**: la casistica ha tempo d'esecuzione costante, come quella relativa all'azione non presente nell'albero.

3.3.3 Rappresentazione Policy

Questa componente della rappresentazione attua le seguenti logiche:

- assegna ai target asset un insieme di regole;
- le regole assegnate vengono rappresentate tramite 2 alberi delle regole: uno con radice **USE** ed uno con radice **TRANSFER**;
- si occupa di attuare il merging di 2 insiemi di regole;
- le modalità di merging sono **intersect** e **union**.

Unione di policy
L'unione di 2 policy è un'operazione creata per supportare uno scenario collaborativo, nel quale i permessi definiti da una policy vengono riconosciuti anche da quelli definiti da un'altra. Quest'operazione è attuata tramite i seguenti passi:

1. unione della lisa delle regole tra le regole della policy attuale e quella della policy con cui la si vuole unire;
2. creazione di una nuova policy avente il set di tutte le regole;
3. il metodo costruttore sfrutta la logica già esposta nel paragrafo 3.3.2 relativo alla gestione degli alberi delle regole.

Intersezione di policy
L'intersezione di 2 policy è un'operazione creata per supportare uno scenario conservativo, nel quale non si ha controllo sui permessi definiti dalla policy con cui si effettua l'intersezione, di conseguenza un permesso è mantenuto solo nel caso in cui sia definito da entrambe le policy. La logica è attuata mediante i seguenti passi:

1. si recuperano gli stati relativi ad ogni azione in entrambe le policy; gli stati sono mantenuti in mappe con chiave l'azione regolata;

2. il passo 1 è attuato per entrambi gli alberi delle policy;

3. si itera una delle 2 mappe e si fanno i seguenti controlli:

   - se l'azione è **Prohibited** in almeno una delle policy, si inserisce il divieto relativo tra le regole finali;
   - se l'azione è **Undefined** in almeno una delle policy, la si lascia non regolata;
   - se non si è nei due casi precedenti, l'azione allora è **Permitted** da entrambe e si inserisce il relativo permesso tra le regole finali;

4. si crea una nuova policy avente il set delle regole finali ottenute ai passi precedenti;

5. il metodo costruttore sfrutta la logica già esposta nel paragrafo 3.3.2 relativo alla gestione degli alberi delle regole.

3.3.4 Rappresentazione Asset

Questo componente si occupa di fornire funzioni relative alla gestione degli asset, le quali prevedono:

- supporto alla gerarchia degli asset, dando modo di esplicitare la proprietà **partOf**(attualmente la gerarchia è un albero, è possibile renderla un grafo a patto che non abbia cicli);
- collega un asset alla policy che definisce delle regole ad esso relative;
- propaga le regole di un asset alle collezioni figlie dell'asset, potendo scegliere come modalità di merging sia l'operazione **intersect** che quella di **union**, esposte in sezione 3.3.3.

Per supportare queste operazioni, si è usata un'architettura simile a quella delle regole mostrata nella sezione 3.3.2, nella quale si sono separate le funzionalità a livello di gerarchia da quelle relative al singolo nodo. Di seguito è riportato il diagramma delle componenti:

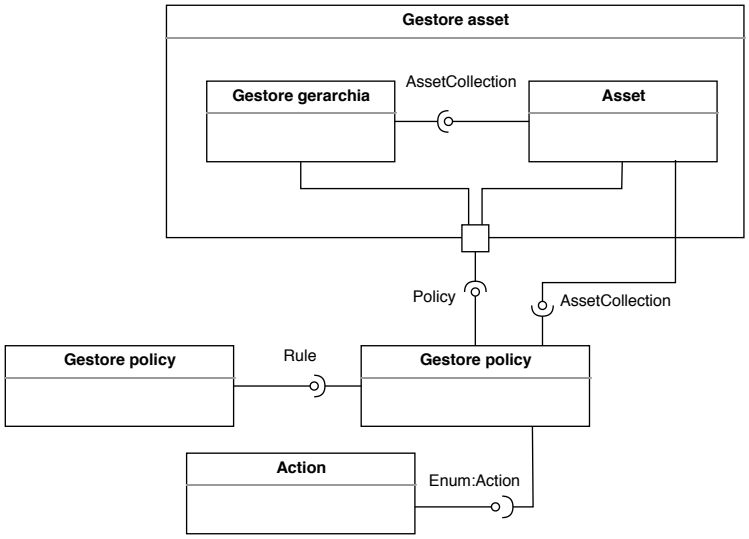

Figure 3.6: Diagramma dei componenti per la rappresentazione degli asset

Come è possibile notare dalla figura 3.6, questo componente sfrutta la logica presente in ogni altro componente esposto precedentemente.
Lo scenario di utilizzo di un **Asset** si compone delle seguenti fasi:

1. definizione dei legami di gerarchia degli asset, seguento quanto esplicitato nella proprietà **partOf**;

2. definizione delle regole relative all'asset;

3. creazione di una policy avente le regole definite al passo precedente;

4. assegnare la policy al gestore della gerarchia, il quale si occuperà di settare la policy per il nodo interessato e propagarla ai figli.

Propagazione delle policy
La propagazione di una policy avviene dopo che questa è stata settata mediante la gerarchia: risulta quindi necessario recuperare il nodo; per questa prima fase dell'operazione si è utilizzata una ricerca in profondità, data la natura ricorsiva dell'operazione di propagazione; sarebbe stato possibile utilizzare anche una ricerca in ampiezza, ma il caso peggiore sarebbe rimasto il medesimo, data l'assenza di cicli.
Dopo il recupero del nodo interessato dalla policy, si attua l'effettiva propagazione ad ogni nodo figlio, fino ad arrivare ai nodi foglia. Qualora il nodo non avesse ancora una policy, viene settata quella del nodo padre.
Di seguito alcuni esempi:

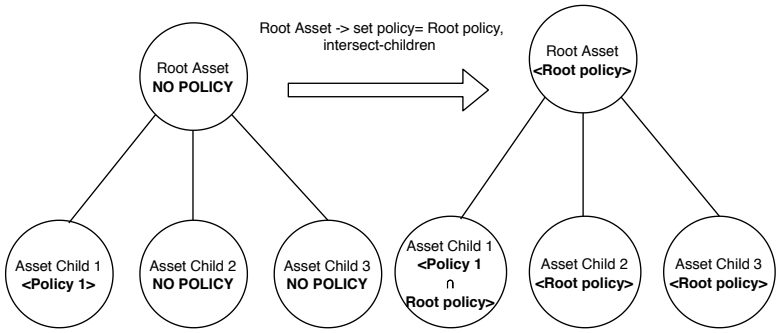

Figure 3.7: Esempio di propagazione di una policy quando l'asset padre non ha policy, modalità con intersezione

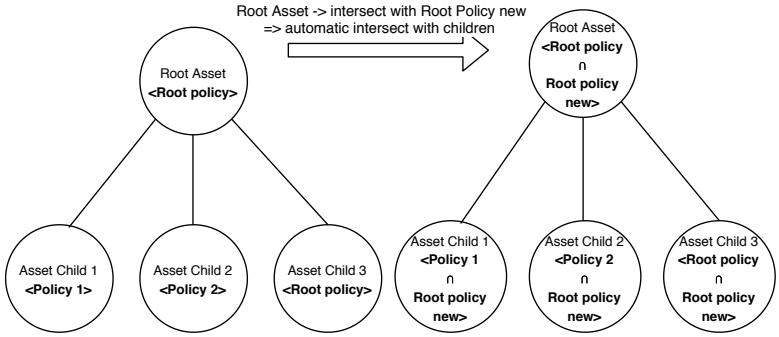

Figure 3.8: Esempio di propagazione di una intersezione di policy

Il duale delle operazioni mostrate nelle figure 3.7 e 3.8 risulta disponibile per le operazioni di unione. Sempre dalla figura 3.8 è possibile notare come se un asset figlio non abbia una propria policy, eredita in modo completo quella del nodo padre.

Recupero policy in seguito alla propagazione

La policy relativo ad un nodo ai livelli più bassi può cambiare drasticamente in seguito ad una operazione di propagazione, come visibili nelle figure 3.7 e 3.8. A supporto di altri componenti è stata implementata una procedura di recupero di policy del singolo

nodo all'interno della gerarchia: si esplora in ampiezza l'intera gerarchia, recuperando ogni policy di ogni risorsa. Il risultato finale è una mappa che collega l'URI di ogni risorsa presente nella gerarchia alla sua policy; nel set di chiavi sono presenti anche i nodi non aventi una policy.

3.3.5 Gestore procedura di merging

Tutte le componenti fino ad ora mostrate si concentrano in modo atomico su una singola porzione della procedura del merging di 2 policy. Come è possibile notare supportano le operazioni di intersezione ed unione, ma tutte le strutture dati lavorano su un singolo documento ODRL. Il componente mostrato in questa sezione, invece, si occupa di attuare l'effettivo flusso applicativo desiderato. Di seguito il diagramma delle componenti che interessano il processo di merging:

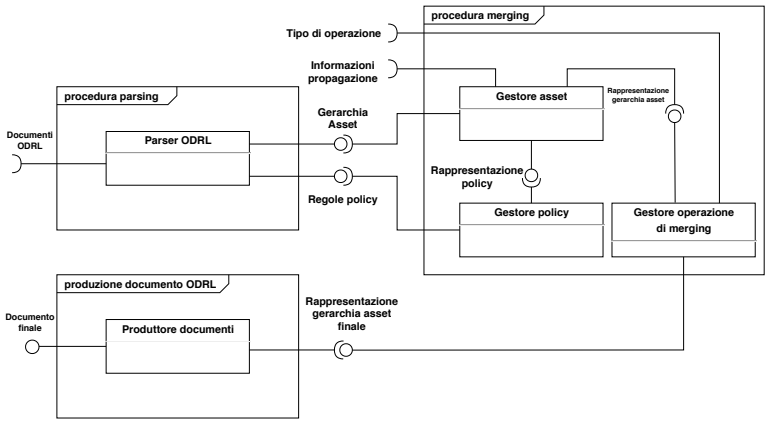

Figure 3.9: Componenti che interagiscono nel processo di merging di 2 policy ODRL

Dalle interfacce esposte è possibile notare quali siano le informazioni che un utente deve specificare per attuare il merging di 2 policy:

- i documenti contenenti le policy ODRL su cui si vuole operare;
- l'ambiente in cui le **singole** policy operano, identificato come:
    1. **collaborativo**: la policy espressa in un asset ai livelli più alti della gerarchia si propaga in modalità **union** ai nodi ai livelli più bassi della gerarchia;
    2. **non collaborativo**: la policy espressa in un asset ai livelli più alti della gerarchia si propaga in modalità **intersection** ai nodi ai livelli più bassi della gerarchia;

- l'operazione che si vuole attuare nel merging, che può essere **union** o **intersection**;

- l'operazione decisa, applicherà il corrispondente ambiente operativo all'intera gerarchia che risultante dal merging delle 2 policy.

Casistiche supportate
Tutti i casi di merging supportati, attualmente non prevedono un **Assignee** per le regole, si suppone quindi che tutte le regole si riferiscano allo stesso **Party** o **PartyCollection**. Un metodo per generalizzare queste casistiche per tener conto dell'**Assignee** delle regole sarebbe quello di creare una degli asset per ognuno degli attori interessati e ripetere i procedimenti di seguito per ognuno di loro.
Tenendo conto di questa prima semplificazione, il merging di 2 policy può essere attuato nei seguenti casi relativi alla gerarchia degli asset:

1. le 2 policy regolano la medesima gerarchia di asset;

2. le 2 policy regolano gerarchie totalmente disgiunte;

3. una delle 2 policy regola l'intera gerarchia, mentre l'altra ne gestisce solo una porzione;

4. una delle 2 policy inserisce nuovi nodi figli;

5. una delle 2 policy inserisce un nuovo nodo padre.

Tutte queste casistiche sono inoltre supportate per i seguenti scenari:

1. intersezione di policy entrambe in ambiente collaborativo;

2. intersezione di policy entrambe in ambiente non collaborativo;

3. intersezione di policy in ambienti diversi;

4. unione di policy entrambe in ambiente collaborativo;

5. unione di policy entrambe in ambiente non collaborativo;

6. unione di policy in ambienti diversi;

Implementazione della procedura

In questo paragrafo è riportata la procedura utilizzata per creare la rappresentazione della policy finale. All'interno dell'implementazione è stata usata la seguente semplificazione: le gerarchie degli asset sono alberi. Tale semplificazione è superabile implementando diversamente la rappresentazione degli asset al fine che sia possibile avere più di un unico parent node e verificando che il grafo ottenuto sia un **Directed acyclic graph** (**DAG**). L'algoritmo può essere diviso nelle seguenti fasi:

1. recupero della gerarchia complessiva regolata dalla policy finale, tale passo ha 2 sottopassi:
   (a) trattazione di asset presenti in entrambi i documenti di policy;
   (b) trattazione di asset presenti in uno solo dei due documenti di policy;
2. recupero delle 2 policy che descrivono ogni singolo asset all'interno della gerarchia;
3. esecuzione dell'operazione desiderata su ogni asset;
4. propagazione di tutte le policy ottenute all'interno della gerarchia degli asset finale, con ambiente coerente all'operazione effettuata.

Recupero gerarchia asset complessiva

Di seguito gli pseudocodici relativi al recupero della gerarchia complessiva. Il primo passo riguarda il recupero degli asset in comune alle policy:

```
def Map<String, AssetCollection> mergeHierarchyCommon(
        Map<String,Asset> assetsFirst,
        Map<String,Asset> assetsSecond,
        AssetTree treeFirst,
        AssetTree treeSecond){

  Set uriSetFirst = assets.keySet();
  Set uriSetSecond = assetsSecond.keySet();
  Map<String, AssetCollection> finalAssets;
  commonURI = uriSetFirst.intersect(uriSetSecond);
  finalAssets.put("EveryAsset", everyAssetNode);
  for(String uri : commonURI){

    if(uri.equals("EveryAsset"))
      skip;
```

```
16      if(not(finalAssets.containsKey(uri)))
17       finalAssets.put(uri,new Asset(uri));
18
19      parentFirst = assetsFirst.get(uri).getParent();
20      parentSecond = assetsSecond.get(uri).getParent();
21
22      // Uno dei due non aveva un padre definito
23      if(
24      !parentFirst.getURI().equals(parentSecond.getURI())
           &&
25      (parentFirst.getURI().equals("EveryAsset") ||
26       parentSecond.getURI().equals("EveryAsset"))){
27         actParent =
28          parentFirst.getURI().equals("EveryAsset") ?
29          new Asset(parentSecond.getURI()) :
30          new Asset(parentFirst.getURI());
31
32         if(finalAssets.containsKey(actParent.getURI())){
33          finalAssets.get(actParent.getURI()).addChild(
               finalAssets.get(uri));
34         }else{
35          actParent.addChild(finalAssets.get(uri));
36          finalAssets.put(actParent.getURI(),actParent);
37         }
38      }
39
40      //L'asset ha lo stesso padre in entrambe le policy
41      if(parentFirst.getURI().equals(parentSecond.getURI())
           ){
42       if(finalAssets.containsKey(parentFirst.getURI())){
43        finalAssets.get(parentFirst.getURI()).addChild(
             finalAssets.get(uri));
44       }else {
45        AssetCollection actParent = new Asset(parentFirst.
             getURI());
46        actParent.addChild(finalAssets.get(uri));
47        finalAssets.put(actParent.getURI(),actParent);
48       }
```

```
49      }
50      return finalAssets;
51    }
```

Listing 3.2: L'algoritmo ha prestazioni lineari in proporzione al più più grande degli insiemi di asset

All'interno del listing 3.2 è possibile notare:

- all'interno delle casistiche relative agli asset in comune, non viene considerata la possibilità di asset con padri diversi poiché, lavorando con gerarchie espresse tramite alberi, questo porterebbe ad una situazione d'errore. Qualora si lavorasse con DAG, questa casistica porterebbe all'aggiunta di un nuovo parent node all'interno della lista dei parent del nodo esaminato;
- all'interno della gerarchia è inserito un asset avente URI **EveryAsset**; questo è un asset fittizio al quale non viene mai assegnata una policy, il suo ruolo è fungere da punto d'inizio per la ricerca degli asset che non hanno un parent esplicitato, al fine di avere un'unica sorgente nella ricerca.

Il passo sucessivo è il recupero degli asset non in comune:

```
1  def Map<String, AssetCollection> mergeHierarchyUniq(
2         Map<String,Asset> assets,
3         AssetTree tree,
4         Map<String, AssetCollection> finalAssets){
5
6   //In questo momento finalAsset contiene asset comuni
7   Set commonURI = finalAssets.keySet();
8   for(String uri: assets.keySet()){
9     if(!commonURI.contains(uri)){
10      if(!finalAssets.containsKey(uri))
11       finalAssets.put(uri,new Asset(uri));
12
13          // Aggiunta parent similare ai comuni
14      if(finalAssets.containsKey(assets.get(uri).
           getParentURI())){
15        finalAssets.get(assets.get(uri).getParentURI()).
             addChild(finalAssets.get(uri));
16      }else {
17        AssetCollection actParent =
```

```
18          new Asset(assets.get(uri).getParenttURI());
19         actParent.addChild(finalAssets.get(uri));
20         finalAssets.put(actParent.getURI(),actParent);
21       }
22     }
23     return finalAssets;
24   }
```

Listing 3.3: L'algoritmo ha prestazioni lineari in proporzione al numero di asset

L'algoritmo esposto nel listing 3.3 viene richiamato per gli insiemi di asset di entrambe le policy, coprendo così entrambi i set ricavabili dalle gerarchie.
Applicando i 2 algoritmi descritti finora, l'output presente in **finalAssets** sarà una mappa che collega ogni URI di un asset ad un oggetto che lo rappresenta che contiene già le informazioni riguardanti nodi parent e nodi figli. Sfruttando le componenti descritte nella sezione 3.3.4 è quindi possibile definire un **Gestore gerarchia** che ha come nodo sorgente **EveryAsset**; il gestore si occuperà di propagare le policy lungo la gerarchia.

Recupero policy singole

Come visibile all'interno del prototipo definito nel listing 3.2, il componente di parsing recupera le informazioni necessarie poiché un **Gestore gerarchia** possa creare una gerarchia di asset e propagare le varie regole relative alla policy; tutto questo seguendo le indicazioni relative all'ambiente definite dall'utente, come presentato in figura 3.9.
Entrambe le policy originarie hanno un loro **Gestore gerarchia**, il quale si occupa di recuperare la policy di ogni nodo in seguito alla propagazione, la procedura è stata già presentata nella sezione 3.3.4.
L'output di questo passo sono quindi 2 mappe, una per documento di policy, che legano l'URI di una risorsa alla policy ricavabile dal documento in cui è presente.

Esecuzione del merging e propagazione

Gli ultimi due passi della procedura sono svolti iterando il set di chiavi della mappa **finalAssets** ottenuta in precedenza e recuperando entrambe le policy relative all'URI dell'asset dalle mappe ricavate al passo precedente. Risulta quindi possibile effettuare l'operazione di mergin desiderata, seguendo il corrispondente procedimento presentanto in sezione 3.3.3 per poi propagare ogni policy lungo la gerarchia finale, come presentato nella sezione 3.3.4.
Il risultato finale è quindi un Gestore gerarchia dal quale possono essere recuperate

tutte le policy finali relative agli asset. Queste policy vengono poi utilizzate dal produttore di policy per generare il documento ODRL relativo alle regole ottenute.

3.4 Produzione policy ODRL

//TODO

Bibliography

[1] Chris Bizer. Linkeddata.

[2] Renato Iannell, Michael Steidl, Stuart Myles, and Víctor Rodríguez-Doncel. Odrl version 2.2 ontology.

[3] Renato Iannella, Michael Steidl, Stuart Myles, and Víctor Rodríguez-Doncel. Odrl vocabulary & expression 2.2.

[4] Renato Iannella and Serena Villata. Odrl information model 2.2.

[5] Benedict Whittam Smith and Víctor Rodríguez-Doncel. Odrl implementation best practices.