Policy merging in ODRL

Gianluca Oldani

Work in progress

Indice

Capitolo 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"[10].
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*[2]:

- Utilizzo di URIs come nomi per le risorse;
- Gli URI sono indirizzi HTTP in modo che si 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

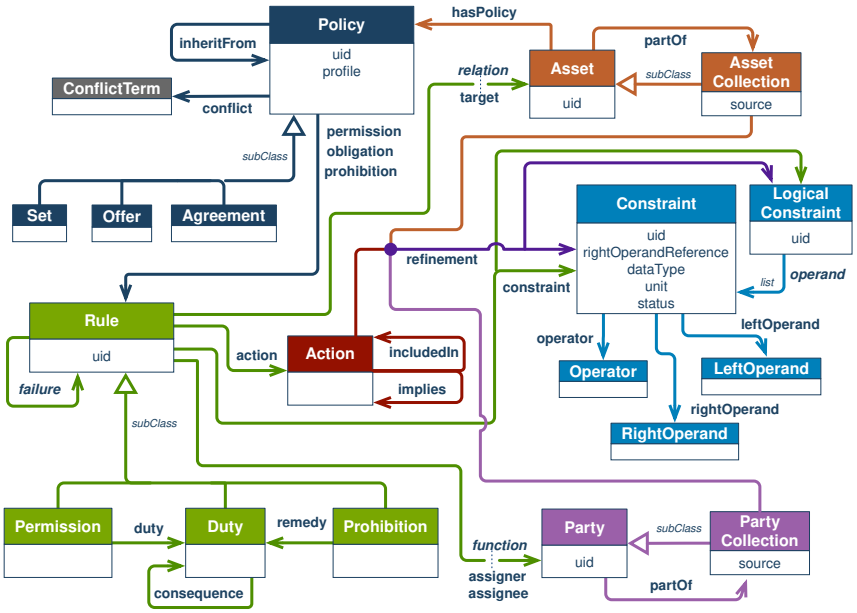


Figura 1.1: Schema del modello ODRL[10]

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"[9]. 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

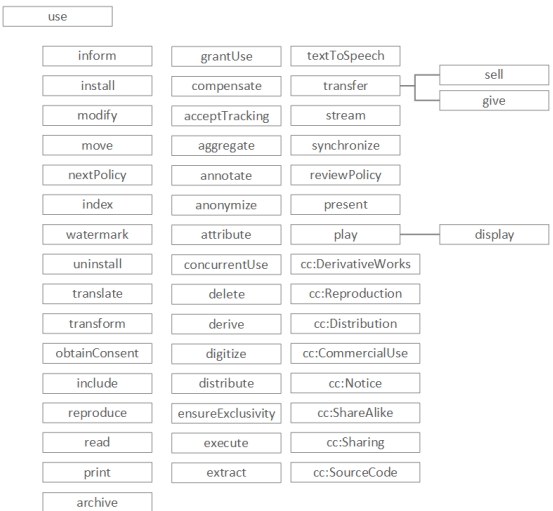


Figura 1.2: Tutte le azioni mostrate sono figlie di use, ad eccezioni di trasfer e le sue sottoazioni[17]

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[8];

- 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.

Capitolo 2: Tecnologie utilizzate

All'interno di questo capitolo sono trattate le varie tecnologie e paradigmi utilizzati per implementare una soluzione alle problematiche mostrate nella sezione 1.1.3. Tale problematica necessita lo sviluppo di una componente che attui la semantica necessaria al merging di 2 policy e che mostri il risultato in un formato non ambiguo; al fine di realizzare questo componente, è stato necessario effettuare una ricerca per quanto concerne:

- tool già sviluppati per il trattamento di policy ODRL;
- tool per interrogare un documento ODRL;
- tool per realizzare un documento ODRL.

2.1 Tool già esistenti

I risultati relativi della ricerca di tool già realizzati hanno portato le seguenti conclusioni:

- gran parte dei tool già sviluppati, non lavorano sull'attuale versione di ODRL[1], come è possibile notare anche dalla pagina del progetto[5];
- in particolare, gli unici tool documentati ufficialmente sono:
  - alcuni valutatori di stato di una regola all'interno di una policy[13], attualmente in sviluppo;
  - un validatore di policy ODRL[12], attualmente in sviluppo;
  - un editor di policy ODRL[15];
  - API per la rappresentazione tramite Java di policy ODRL[14], supporta ODRL 2.0.

[1] La versione utilizzata durante la stesura del lavoro di tesi è ODRL v2.2

Risulta possibile notare che le problematiche di merging e rappresentazione non efficiente delle regole non sono affrontate direttamente dai tool sviluppati, i quali tendono invece ad avere un carattere più generale. Sempre grazie allo scouting è possibile quindi dedurre che l'implementazione di un nuovo tool può non reimplementare i seguenti procedimenti:

- eventuali documenti di policy presentati in ingresso possono essere validati in precedenza;
- non necessita di informazioni sullo stato attuale di asset ed attori coinvolti, in quanto lo stato di questi può essere valutato;
- non deve supportare nella creazione della policy l'utente.

Le uniche componenti relative ai tool già esistenti che nel lavoro esposto sono state reimplementate risultano essere le API per la rappresentazione di policy ODRL in Java: quelle già esistenti sono relative ad una versione di ODRL non attuale e non forniscono interfacce utili al procedimento di merging di 2 policy.

2.2 Interrogazione e produzione documenti ODRL

Le ricerche relative al parsing ed alla produzione di un documento ODRL, hanno portato al framework Apache Jena[4], utilizzato per creare applicazioni secondo i modelli "Semantic web"[20] e "Linked Data". Il framework ed i modelli di cui supporta la creazione, sono descritti nelle sezioni seguenti.

2.2.1 Semantic Web

Il Semantic Web è definita come un'estensione del World Wide Web, nel quale i dati all'interno della rete *machine-readable*. In particolare "nel contesto del Semantic Web, il termine semantico assume la valenza di "elaborabile dalla macchina" e non intende fare riferimento alla semantica del linguaggio naturale e alle tecniche di intelligenza artificiale. Il Semantic Web è, come l' XML, un ambiente dichiarativo, in cui si specifica il significato dei dati, e non il modo in cui si intende utilizzarli"[16]. Alla base dell'idea del Semantic Web vi sono:

- le risorse, non necessariamente interpretabili da una macchina;
- i metadati connessi alle risorse, i quali sono machine-readable.

Per realizzare quest'idea, risulta necessario poter standardizzare i metadati collegati alle risorse, specificandone semantica e sintassi.
Il formato sintattico scelto come standard è il modello Resource Description Framework (RDF)[11], mentre invece si lascia libertà sulla semantica, la quale può venir specificata mediante vocabolari relativi al dominio, utilizzando RDF Schema[3]. Sia RDF che RDFS sono descritti in dettaglio nelle prossime sezioni.
Prendendo come esempio il listing 1.2, risulta possibile notare:

- il documento ODRL è la rappresentazione machine-readable di un documento di specifica dei requisiti d'accesso ad una risorsa;
- la sintassi è standard ed è in formato RDF;
- la semantica è stata definita per gli scopi necessari ad ODRL, esplicitandone gli elementi nel **contesto**;
- la risorsa alla quale è collegato il metadato non è specificata, ma il metadato dà informazioni utili su tutte le risorse presenti, fornendo un'informazione non centralizzata poiché recuperabile con query relative a varie risorse o semplicemente allegando il metadato a più risorse.

2.2.2 Linked Data

Fino ad ora, si è mostrato i principi scelti come base per rendere ogni risorsa machine-readable, permettendo ad una macchina di ottenere maggiori informazioni sulla risorsa. Il Semantic Web non cerca solamente di migliorare la qualità dell'informazione sulla singola risorsa, bensì di fornire ad un applicativo la possibilità di consultare informazioni relative anche a risorse collegate a quella iniziale.
Questo secondo obiettivo viene attuato mediante i 4 principi dei Linked Data, già enunciati nella sezione 1.1.1.
Un importate risultato raggiunto in questo ambito risulta essere DBPedia[18], progetto che permette:

- di consultare Wikipedia in formato RDF;
- collega gli articoli Wikipedia a risorse presenti al di fuori del sito, permettendo così di effettuare query che recuperano informazioni da più fonti.

Come già citato, questo modello è ben visibile all'interno di ODRL, per esempio nel listing 1.1 è possibile notare:

- che la policy ha un proprio URI, risulta quindi sia un metadato, che una risorsa consultabile a sé;

- in quanto risorsa consultabile, ha al suo interno diversi URI relativi ad altre risorse, come ad esempio:
  - l'URI relativo all'asset target della policy;
  - l'URI relativo al contesto del documento, il quale contiene a suo volta informazioni utili per leggere correttamente una policy;

2.2.3 RDF

"RDF, o *Resource Description Framework*, si pone alle fondamenta del processing di metadati; fornisce interoperabilità tra applicazioni che si scambiano infromazioni machine-readable nel Web. RDF enfatizza procedure per abilitare l'elaborazione automatica per risorse web"[11]. Tra i casi d'uso supportati da RDF vi sono:

- recupero di risorse web per ottenere migliori motori di ricerca;
- catalogazione di contenuto ed espressione di relazioni tra varie fonti nel Web;
- rappresentazione di proprietà intellettuale e diritti relativi ad una risorsa Web.

RDF inoltre, condivide la visione per la quale un metadato può essere a sua volta un dato, poiché per all'interno del framework è si definisce come risorsa qualsiasi entità identificabile mediante un URI.
Tra le caratteristiche di RDF vi è il proporre una sintassi che per descrivere risorse che sia indipendente dal dominio di applicazione:

- non vi sono assunzioni specifiche di dominio applicativo;
- non vi sono restrizioni di semantica, definibile in base al dominio.

La sintassi mostrata, fa riferimento a ciò che è definito come *serialization syntax*: formato XML per la rappresentazione del modello RDF. Tale sintassi non è l'unica possibile né l'unica valida:

- la *serialization syntax* si riferisce al lavoro che la W3C sta svolgendo in questo ambito per affinare il modello RDF;
- altre rappresentazioni possono essere più utili in base allo scenario applicativo, per esempio nel caso in cui il documento debba essere anche leggibile da persone.

Il modello RDF

Il modello RDF è definito come segue[11]:

1. dato un insieme di nodi, definito come N;
2. sia un sottoinsieme di N, detto *PropertyTypes*, definito come P;
3. sia un insieme di triple, definito come T, i cui elementi vengono informarmalmente chiamati proprietà. Il primo elemento di ogni tripla è un elemento di P, il secondo un elemento di N ed il terzo può essere un elemento di N o un valore atomico, ad esempio una stringa Unicode.

La rappresentazione del modello equivale ad un grafo orientato, i cui nodi sono le risorse descritte o i valori che le descrivono; gli archi che collegano i nodi presentano come label i nomi dei *PropertyTypes* che li collegano. Per esempio:

1. la frase
   “Ora Lassila" è l’ "autore" della pagina web “http://www.w3.org/People/Lassila””
2. è serializzata come
   {autore, [http://www.w3.org/People/Lassila], "Ora Lassila"}
3. dalla definizio del modello, è possibile dedurre che la label *autore* è il nome di un nodo appartenente a P;
4. tale nodo, identificato con X,avrà a sua volta le seguenti serializzazioni:
   (a) {PropName, X, author}
   (b) {PropObj, X, [http://www.w3.org/People/Lassila]}
   (c) {PropValue, X, "Ora Lassila"}

Nell’esempio precedente, si è mostrato il tipo di asserzione più semplice che RDF è in grado di supportare; quanto segue mostra come si possano creare relazioni tra le varie risorse:

1. la frase
   “Ralph crede che il documento ’L’origine delle specie’ abbia come autore Charles Darwin”
2. è serializzata come
   {crede, "Ralph", Statement1}
3. le serializzazioni relative al nodo di *crede*, identificato con X, sono:

(a) {PropName, X, crede}

(b) {PropObj, X, Ralph}

(c) {PropValue, X, Statement1}

4. *Statement1* è un elemento di T, ovvero un tripla;

5. le serializzazioni relative a Statement1 sono:

   (a) {PropName, Statement1, autore}

   (b) {PropObj, Statement1, [http://loc.gov/Books/OrigineSpecie]}

   (c) {PropValue, Statement1, Charles Darwin}

Dagli esempi mostrati, risulta possibile denotare che il cuore del modello RDF siano quindi le triple che si definiscono, dove in ognuna è possibile individuare:

- **predicato**: nodo appartente a P;
- **oggetto**: nodo sorgente dello statement; appartiene a P ed è il nodo a cui si applica il predicato;
- **valore**: nodo destinazione dello statement; appartiene a P, T, o è un valore atomico ed è il valore del predicato.

Grammatica RDF e rappresentazione grafica

Il seguente codice mostra un frammento di descrizione di dati conforme al modello RDF tramite XML:

```
<?namespace
        href="http://docs.r.us.com/bibliography-info"
        as="bib"?>
<?namespace
        href="http://www.w3.org/schemas/rdf-schema"
        as="RDF"?>
<RDF:serialization>
  <RDF:assertions href="http://www.bar.com/some.doc">
    <bib:author>John Smith</bib:author>
  </RDF:assertions>
</RDF:serialization>
```

Listing 2.1: Esempio di metadato conforme ad RDF in XML

Risulta possibile notare i seguenti tag:

- **serialization**: indica un insieme di triple all'interno del modello;
- **assertions**: indica un'effettiva tripla all'interno del modello, i cui elementi sono individuati nei tag:
  - **href** nel tag assertions: indica l'oggetto della tripla;
  - **prefix:PropertyName**: tag definito dall'applicazione che indica il predicato della tripla; il valore di quest elemento XML indica invece il valore del predicato.

Di seguito la rappresentazione grafica del grafo codificato:

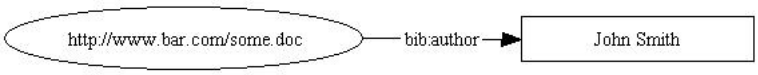


Figura 2.1: Rappresentazione grafica del grafo codificato nel listing 2.1

Nell'esempio che segue è invece possibile notare come sfruttare uno statement come valore di una tipla:

```
<?namespace
  href="http://docs.r.us.com/bibliography-info"
  as="bib"?>
<?namespace
  href="http://www.w3.org/schemas/rdf-schema"
  as="RDF"?>
<RDF:serialization>
  <RDF:assertions href="http://www.bar.com/some.doc">
    <bib:author>
      <RDF:resource>
        <bib:name>John Smith</bib:name>
        <bib:email>john@smith.com</bib:email>
        <bib:phone>+1 (555) 123-4567</bib:phone>
      </RDF:resource>
    </bib:author>
  </RDF:assertions>
</RDF:serialization>
```

Listing 2.2: Esempio di metadato conforme ad RDF in XML

Risulta possibile anche ottenere una scrittura più modulare, sfruttando la proprietà *id* del tag *resource*:

```
<?namespace
        href="http://docs.r.us.com/bibliography-info"
        as="bib"?>
<?namespace
        href="http://www.w3.org/schemas/rdf-schema"
        as="RDF"?>
<RDF:serialization>
  <RDF:assertions href="http://www.bar.com/some.doc">
    <bib:author href="#John_Smith"/>
  </RDF:assertions>
</RDF:serialization>

<RDF:resource id="John_Smith">
  <bib:name>John Smith</bib:name>
  <bib:email>john@smith.com</bib:email>
  <bib:phone>+1 (555) 123-4567</bib:phone>
</RDF:resource>
```

Listing 2.3: Esempio di metadato conforme ad RDF in XML

Entrambi gli esempi mostrati nei listing 2.2 e 2.3, si riconducono alla seguente rappresentazione grafica:

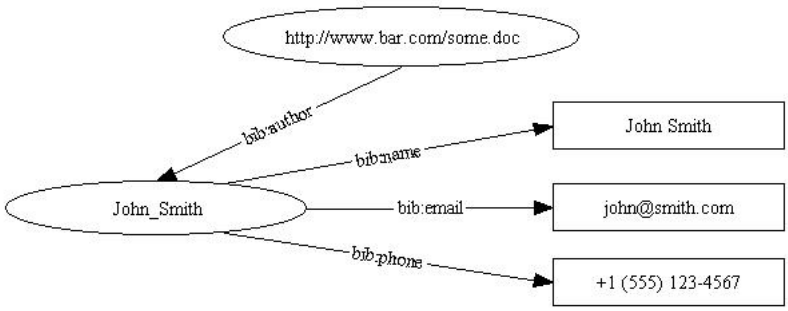


Figura 2.2: Rappresentazione grafica del grafo codificato nei listing 2.2 e 2.3

Tag namespace

Il tag *namespace* è un'estensione alla specifica RDF attualmente non mandatorio. Tale tag è stato introdotto per rendere la sintassi relativa alla semantica di un tag specifico di dominio meno prolissa:

- **as**: proprietà del tag che indica il nome del namespace;

- **href**: porzione di identificativo da sostituire alla proprietà *as* quando la si incontra.

Prendendo ad esempio il listing 2.2: il tag *bib:name* è da interpretare come:
`http://docs.r.us.com/bibliography-info/name`
quest'ultimo risulta infatti essere l'URI che identifica la proprietà, inoltre è un riferimento a dove è possibile recuperare informazioni su di essa.
Il suo utilizzo è denotabile all'interno di ODRL, come ad esempio nel listing 1.4, dove il tag *@context* svolge un ruolo simile: fa riferimento ad un file RDF all'interno del quale sono definite tutte le abbreviazioni utili ad ODRL tramite le seguenti linee di codice:

```
1 "odrl":       "http://www.w3.org/ns/odrl/2/",
2 "Permission": "odrl:Permission",
3 "permission": {"@type": "@id", "@id": "odrl:permission"}
```

Listing 2.4: Estratto context ODRL[19]

Come è possibile notare, il formato di serializzazione utilizzato nel listing 2.4 non è XML, bensì JSON-LD, così come anche nei vari listing presentati nel capitolo 1 relativo ad ODRL.
All'interno del lavoro di tesi, le serializzazioni utilizzate sono:

- JSON-LD: utilizzato negli esempi ed in input al processo di merging;
- Turtle[1]: utilizzato come output del processo di merging, tale decisione è stata presa poiché risulta essere la serializzazione più matura nel framework Jena.

2.2.4 RDFS

Come esposto in precedenza, il modello RDF fornisce la sintassi necessaria per creare metadati in grado di rappresentare relazioni tra le varie risorse, lasciando libertà assoluta sul significato che queste relazioni possono avere. RDFS è "un'estensione semantica"[3] di RDF che permette di descrivere gruppi di risorse correlate e le relazioni tra le varie risorse. RDFS basa il suo funzionamento sulla definizione di 2 termini:

- **rdfs:Class**: raggruppamento di risorse con le stesse proprietà, come ad esempio la classe *odrl:Policy*;
- **rdf:Property**: relazione tra risorse soggetto e risorse soggetto; istanze di questa particolare classe possiedono, a loro volta, le seguenti proprietà:

- **rdfs:range**: i *valori* in una tripla della proprietà devono essere istanze delle classi specificate in questa proprietà;
- **rdfs:domain**: gli *oggetti* in una tripla della proprietà devono essere istanze delle classi specificate in questa proprietà;

Oltre a quanto appena mostrato, esistono anche 2 proprietà denominate: **rdfs:subClassOf** e **rdfs:subPropertyOf**, le quali permettono di definire un sistema di ereditarietà tra classi e tra proprietà, come ad esempio nel vocabolario ODRL si ha:

```
<rdfs:Class
  rdf:about="http://www.w3.org/ns/odrl/2/Prohibition">

  <rdfs:isDefinedBy
    rdf:resource="http://www.w3.org/ns/odrl/2/"/>

  <rdfs:label xml:lang="en">Prohibition</rdfs:label>
  <rdfs:subClassOf
   rdf:resource="http://www.w3.org/ns/odrl/2/Rule"/>

</rdfs:Class>
```

Listing 2.5: Estratto vocabolario ODRL[9]

L'estratto mostrato è la definizione della sottoclasse *Prohibition*, esposta nella sezione 1.1.2. Un secondo esempio, sempre visibile in ODRL, è la definizione della proprietà "odrl:partOf", mostrata nei listing 1.3 e 1.5, tale proprietà:

- 2 possibili *rdfs:domain* e 2 possibili *rdfs:range*;
- solo 2 coppie *rdfs:domain-rdfs:range* sono valide, tale restrizione è espressa mediante OWL[6], ulteriore estensione semantica utilizzata per espriremere concetti logici complessi;
- all'interno delle proprietà sono specificate le 2 classi *odrl:Party* ed *odrl:Asset*.

2.2.5 Apache Jena

"Apache Jena (o Jena) è un framework Java free ed open source per creare applicazioni del semantic web o che sfruttano Linked Data. Il framework è composta da diverse API che interagiscono al fine di processare dati RDF."[4]
Per questo motivo si è scelto Jena come insieme di utility sia a supporto della porzione

di parsing, descritta nella sezione 4.2, sia a supporto della porzione di produzione del documento ODRL finale, descritta nella sezione 4.4. Nel dettaglio, le API utilizzate comprendono:

- **Core RDF API**;
- **Jena schemagen**;
- **ARQ - A SPARQL Processor for Jena**;
- **Modulo I/O**.

Core RDF API
Questa porzione del framework offre interfacce per l'interazione con il modello RDF. La rappresentazione offerta da questa libreria si basa su triple, i cui elementi sono denominati:

- **resource**: l'oggetto dello statement codificato dalla tripla;
- **property**: il predicato dello statement;
- **value**: il valore dello statement, il quale può essere a sua volta una risorsa.

L'interfaccia core fornita da questo componente è denominata *Model*, il quale denota un'intero grafo RDF; questa interfaccia è intesa per supportare operazioni molto astratte sull'intero modello, e può essere usata per:

- specificare su quale file o collezione di file RDF si sta lavorando;
- aggiungere, recuperare e rimuovere nodi dal grafo;
- settare i *namespace* utilizzati all'interno del modello;
- settare la serializzazione con la quale il modello viene mostrato;
- interagire con l'API ARQ per l'interrogazione del modello tramite query SPARQL.

Jena schemagen
Questo tool fornito dal framework ha utilità puramente tecnica, in quanto è uno strumento per "convertire vocabolari OWL o RDFS in una classe Java class che contiene costanti statiche per i termini nel vocabolario".
L'effettivo utilizzo del tool si traduce nel suo processare un file RDF, in questo caso il file context di ODRL[19], automatizzando il processo di estrazione dei termini all'interno del vocabolario. I vantaggi di questo approccio si traducono in:

- l'utilizzo di variabili facilmente leggibili all'interno del linguaggio ARQ, anziché dell'URI che definisce la classe o la proprietà;

- interazione più trasparente con le *Core RDF API* in fase di aggiunta risorse o di definizione delle proprietà di un nodo.

Il contributo di questo tool non è direttamente visibile all'interno del codice del progetto di tesi, poiché è stata sfruttata la versione standalone del tool.

ARQ e SPARQL

"ARQ è il query engine fornito da Jena che supporta il linguaggio SPARQL RDF Query."[4]

Il linguaggio "SPARQL può essere usato per esprimere query su diverse sorgenti di dati, sia su dati nativi RDF o esposti come RDF via middleware."[7]

L'obiettivo di queste tecnologie è fornire interfacce che permettano l'interrogazione di modelli RDF. ARQ è l'API esposta da Jena per formulare tramite Java le interrogazioni; SPARQL è l'effettivo linguaggio nel quale è espressa la query.

Prendendo ad esempio il seguente file in formato RDF:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
@prefix ns:   <http://example.org/ns#> .

:book1  dc:title  "SPARQL Tutorial" .
:book1  ns:price  42 .
:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
```

Listing 2.6: File sul quale si vuole eseguire una query

```

PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { ?x dc:title ?title
          FILTER regex(?title, "^SPARQL")
        }
```

Listing 2.7: Query SPARQL per il recupero dei titoli che iniziano per SPARQL

Il risultato della query nel listing 2.7 è la tupla:

(**title** = “SPARQL Tutorial”)

Questo secondo esempio mostra come sia possibile filtrare uno statement sulla base di più espressioni:

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Johnny Lee Outlaw" .
_:a  foaf:mbox   <mailto:jlow@example.com> .
_:b  foaf:name   "Peter Goodguy" .
_:b  foaf:mbox   <mailto:peter@example.org> .
_:c  foaf:mbox   <mailto:carol@example.org> .
```

Listing 2.8: File sul quale si vuole eseguire una query

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  { ?x foaf:name ?name .
    ?x foaf:mbox ?mbox }
```

Listing 2.9: Query SPARQL per il recupero dei nomi e delle mail solo per soggetti che presentano entrambe le proprietà

Il risultato della query nel listing 2.7 sono le tuple:

(**name** = "Johnny Lee Outlaw", **mbox** = <mailto:jlow@example.com>)
(**name** = "Peter Goodguy", **mbox** = <mailto:peter@example.org>)

Risulta possibile notare che l’ultima tripla presente nel listing 2.8 non sia stata utilizzata per costruire il risultato della query, poiché l’oggetto *_:c* non ha la proprietà *foaf:mbox*.
Il ruolo che hanno le API ARQ all’interno di del progetto è di collegamente tra le query e le *CORE RDF API*, nonché di semplificazione della sintassi, evitando il costrutto *PREFIX* laddove già presente all’interno del modello sul quale si eseguono le query.

Modulo I/O
Il modulo Input/Output fornisce le interfacce necessarie:

- alla lettura di un file in formato RDF, supportando le serializzazioni:
  - Turtle;
  - N-Triples;
  - N-Quads;
  - TriG;
  - RDF/XML;
  - JSON-LD;
  - RDF/JSON;
  - TriX;
  - RDF Binary;
- alla scrittura di un file in formato RDF, supportando le medesime serializzazione mostrate in lettura.

Il modello espresso mediante *CORE RDF API* otttenuto da questi moduli è il medesimo a discapito della serializzazione utilizzata, di conseguenza non è necessario, almeno in lettura, attuare implementazioni specifiche basate sul formato. Quest'ultima constatazione vale anche per il procedimento di parsing e, quindi, per l'intera procedura di merging a valle. L'unica porzione del progetto che necessita di adattamenti in base al formato risulta essere il produttore di documenti ODRL, in quanto alcuni formati possono necessitare di alcuni adattamenti per essere resi più leggibili dagli utenti. Non sono invece necessari adattamenti per l'utilizzo da parte di un programma.

Capitolo 3: Valutazione casistiche

3.1 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 3.1: 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 3.2: 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 3.3: 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 3.4: 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 3.5: 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 3.6: 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 3.7: 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 3.8: 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 3.9: 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 3.10: 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 3.11: 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 3.12: 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 3.13: 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"[3.1].

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 3.14: 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 3.15: 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"[3.1] 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 3.16: 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"[3.1], 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" [3.1]. 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 3.17: 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).

Capitolo 4: Implementazioni

4.1 Architettura generale

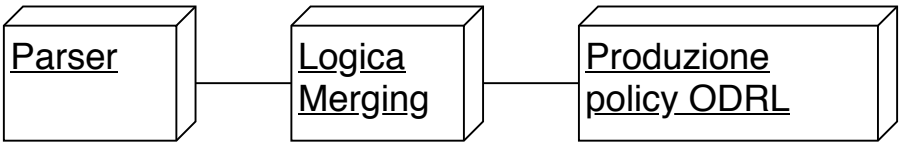


Figura 4.1: Pipeline dei macro componenti della soluzione

Come mostrato nella figura 4.1, 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.

4.2 Parser

Il primo componente della pipeline risulta il parser di documenti ODRL. Il suo ruolo è estrarre le informazioni utili alla procedura di merging. Tale componente sfrutta il framework Apache Jena, esposto nella sezione 2.2.5. Questa porzione del progetto, espone 2 interfacce:

- un'interfaccia per il recupero delle *Rule* relative ad ogni *Asset* all'interno della policy;
- un'interfaccia per il recupero delle relazioni *partOf* tra i vari *Asset* all'interno della policy.

Questa decisione è stata presa al fine di incapsulare al meglio queste 2 funzionalità e permettere di usarle atomicamente dove fosse necessario. L'architettura del componente è mostrata nella seguente figura:

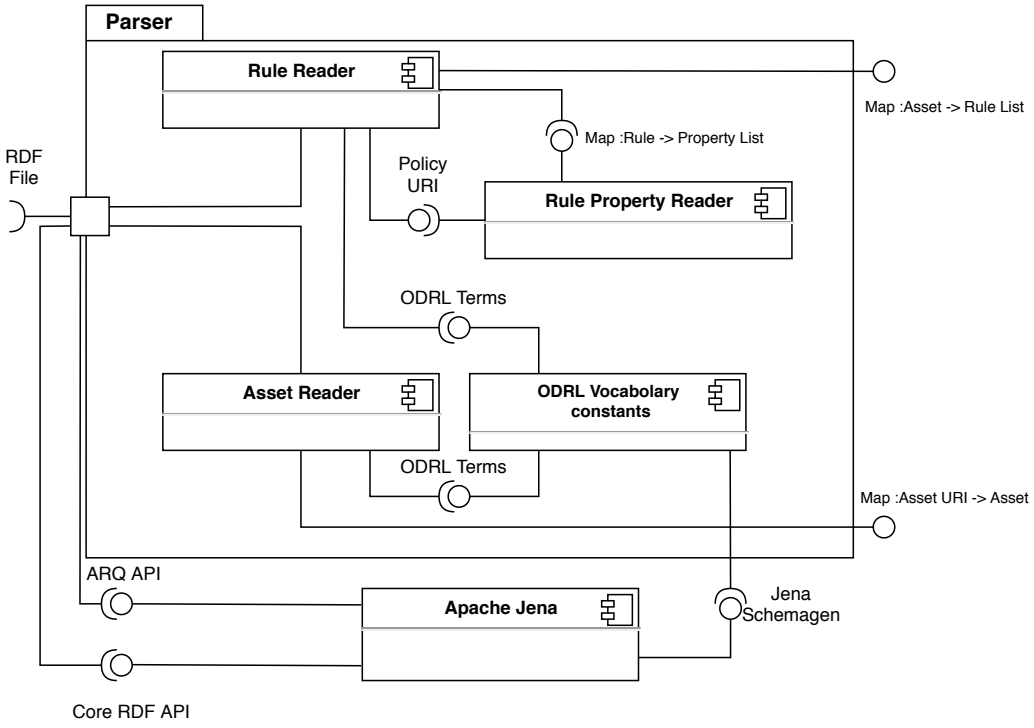


Figura 4.2: Diagramma delle componenti per il parsing di documenti ODRL

4.2.1 Rule Reader

Questo componente ha il ruolo di estrarre le regole presenti in una policy, organizzandole in una mappa avente come chiavi gli *Asset* e come valori le liste delle rispettive regole. Il procedimento attuato segue i seguenti passi:

1. recupero degli URI delle singole policy contenute nel documento ODRL;
2. recupero della lista delle proprietà all'interno di una regola ODRL;
3. estrazione della proprietà *target* dalle regole per popolare le chiavi della mappa;
4. estrazione delle proprietà *type* e *action* per creare il wrapper di una regola ODRL;
5. inserimento del wrapper all'interno della lista del rispettivo target.

Il primo passo della procedura è attuato grazie alla query SPARQL:

```
SELECT *  WHERE {?policy <rdf:type> <odrl:Policy> .}
```

Listing 4.1: Query per il recupero dei dati relativi a tutte le policy contenute nel documento ODRL

In questo modo, nonostante sia una pratica poco diffusa, è possibile identificare più policy all'interno del documento ODRL. L'URI delle varie policy è contenuto nella chiave **policy** delle tuple recuperate dalla query.
Il secondo passo della procedura è attuato mediante questa seconda query SPARQL:

```
SELECT ?rule ?pred ?ogg ?type
WHERE {
        ?policy ?type ?rule .
        ?rule ?pred ?ogg
        FILTER (
                ?type IN (<odrl:permission>,
                          <odrl:prohibition> )
                )
}
```

Listing 4.2: Query per il recupero delle proprietà di tutte le regole

Le tuple recuperate da questa query hanno la seguente struttura:

- **rule**: uri autogenerato da ARQ per le regole, in quanto prive di URI nelle policy;
- **pred**: proprietà della all'interno della regola, per esempio *rdf:type* o *odrl:target*;
- **ogg**: valore della proprietà identificata dalla chiave *pred*;
- **type**: valore delle proprietà *rdf:type* della regola; come è possibile notare dalla clausola *FILTER* può assumere solo i due valori *odrl:permission* o *odrl:prohibition*.

Il risultato viene riorganizzato in una mappa che lega ogni regola ad una coppia *proprietà-valore*.
Gli ultimi 3 passi della procedura sono eseguiti iterando la mappa ottenuta tramite l'algoritmo espresso dal seguente pseudo codice:

```
Map<Asset,List<Rule> mapAssetRuleList;
Iterator itRule = mappaParams.entrySet().iterator();

//Iterazione sulla mappa precedente
while(itRule.hasNext() ){

    MapEntry singleRule = itRule.next();
    List<Pair<String,String>> propertyList = singleRule.
       getValue();

    //Possibili property
    Asset collection = new Asset("EveryAsset");
    Action action = null;
    String type = "";

    //Itero per recuperare il valore delle varie property
        for(Pair<String,String> actProperty :
           propertyList){

         if(actProperty == odrl:permission)
          type = "Permission";
         if(actProperty == odrl:prohibition)
          type = "Prohibition";
         if(actProperty == odrl:target)
          collection = new Asset(actProperty.Value);
         if(actProperty == odrl:action)
              action = actProperty.Value;
        }

    //Creazione del wrapper
    if(type.equals("Permission")
          rule = new Permission(action)
    else
```

```
32            rule = new Prohibition(action)
33      //Assegnazione wrapper a chiave corretta
34      if(mapAssetRuleList.hasKey(collection){
35            mapAssetRuleList.get(collection).add(rule);
36      }else{
37            List<Rule> assetRuleList = new List();
38            assetRuleList.add(rule);
39            mapAssetRuleList.put(collection,assetRuleList);
40      }
41      resetTmp(collection,action,type);
42
43  }
44  return mapAssetRuleList;
```

Listing 4.3: Pseudo codice per il popolamento della mappa finale

Il codice mostrato nel listing 4.5 risulta avere complessità lineare proporzionale al numero di proprietà recuperate dalle query SPARQL.

4.2.2 Asset Reader

Questo componente ha il ruolo di estrarre ogni asset presente nella policy che soddisfi almeno una delle seguenti condizioni:

- è il target di almeno una regola;
- è il valore della proprietà *partOf* di almeno un altro asset.

Tutti gli asset che non soddisfano almeno una di queste condizioni non sono utili alla procedura di merging, poiché non sono effettivamente regolamentati dalla policy ODRL e non portano informazioni di ereditarietà tra gli asset. Gli asset recuperati vengono poi organizzati in una mappa che collega il loro URI al wrapper utilizzato dalla procedura di merging. I wrapper sono creati poiché possano fornire informazioni relative a legami di ereditarietà tra asset.
I passi per fornire questa funzionalità sono:

1. recupero di tutti gli asset presenti nella policy tramite query SPARQL che filtra le 2 proprietà;
2. creazione dei wrapper relativi ed inserimento all'interno della mappa che viene restituita come risultato finale;

3. per ogni asset, si esegue una query SPARQL che recupera tutti gli asset che lo hanno come parent;

4. si aggiorna ogni wrapper con l'informazione ottenuta dal passo precedente.

Di seguito la prima delle due query SPARQL effettuate:

```
SELECT DISTINCT ?obj  WHERE {
  ?sub ?pred ?obj
     FILTER (?pred IN
               (<odrl:target>, <odrl:partOf> )
             )
}
```

Listing 4.4: Query SPARQL per il recupero degli asset utili alla procedura di merging

Le tuple restituite da questa query presentano come valore con chiave *obj* l'URI degli asset che almeno una volta sono il valore di una proprietà *odrl:target* o *odrl:partOf*.
La seconda query necessaria al procedimento invece è la seguente:

```
SELECT DISTINCT ?sub  WHERE {
   ?sub <odrl:partOf> <parentURI>}
}
```

Listing 4.5: Query SPARQL per il recupero degli asset utili alla procedura di merging

In questo caso, le tuple restituite presentano come valore con chiave *sub* l'URI degli asset che hanno la proprietà *odrl:partOf* valorizzata con l'URI presente al posto del placeholder *parentURI*; l'identificatore effettivamente utilizzato è quello dell'asset per cui si sta iterando nel passo 3 della procedura.
Per ogni URI recuperato da questa query, si aggiorna il relativo wrapper aggiungendo alla lista dei parent, il nodo dell'iterazione corrente.

4.3 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.

4.3.1 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 4.3:

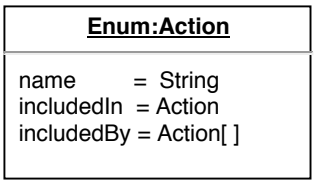


Figura 4.3: Struttura dell'enumerativo Action

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.

4.3.2 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 4.3. Per ottenere un maggior incapsulamente delle funzioni, si è deciso di dividere la logica della struttura ad albero in due compenenti, mostrate in figura 4.4 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.

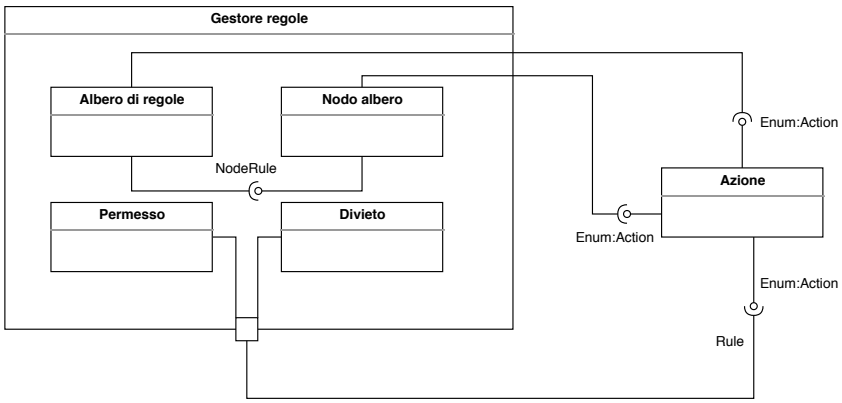


Figura 4.4: Diagramma dei componenti per la rappresentazione delle regole

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 4.3.1, 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:

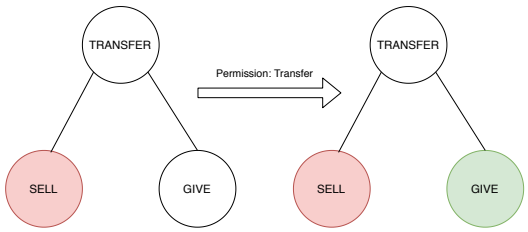


Figura 4.5: Esempio di propagazione di un permesso, con azione figlia proibita

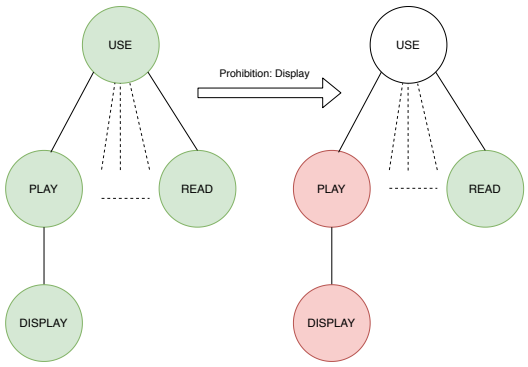


Figura 4.6: Esempio di propagazione verso il padre di un divieto

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 4.3, 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:

```
1 def getActionNode(Action a){
2 List steps = [];
3 Action visited = a;
4 Node exploredNode = Tree.getRoot();
5
6 while(visited != None){
7   steps.add(visited);
8   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 4.6: 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 4.6 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.

4.3.3 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 4.3.2 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 4.3.2 relativo alla gestione degli alberi delle regole.

4.3.4 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 4.3.3.

Per supportare queste operazioni, si è usata un'architettura simile a quella delle regole mostrata nella sezione 4.3.2, 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:

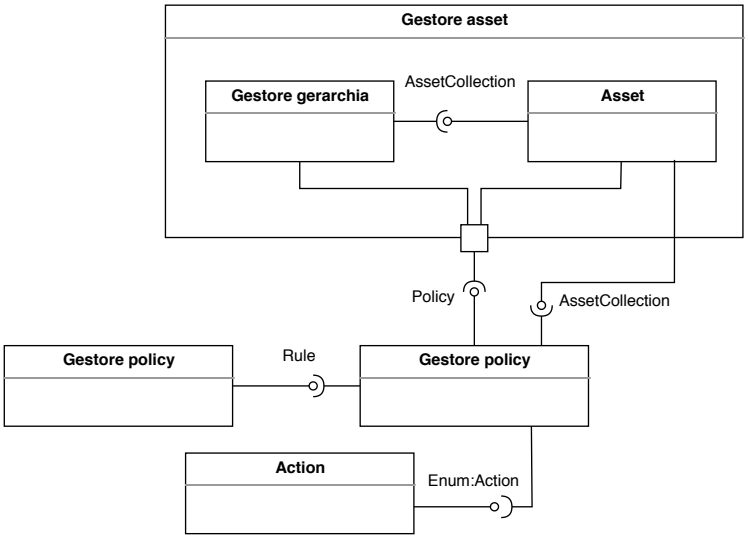


Figura 4.7: Diagramma dei componenti per la rappresentazione degli asset

Come è possibile notare dalla figura 4.7, 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:

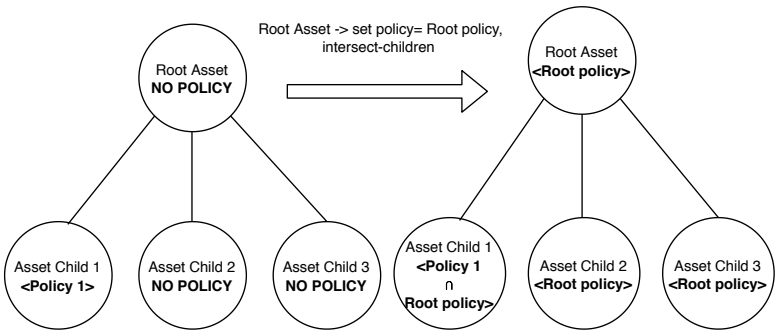


Figura 4.8: Esempio di propagazione di una policy quando l'asset padre non ha policy, modalità con intersezione

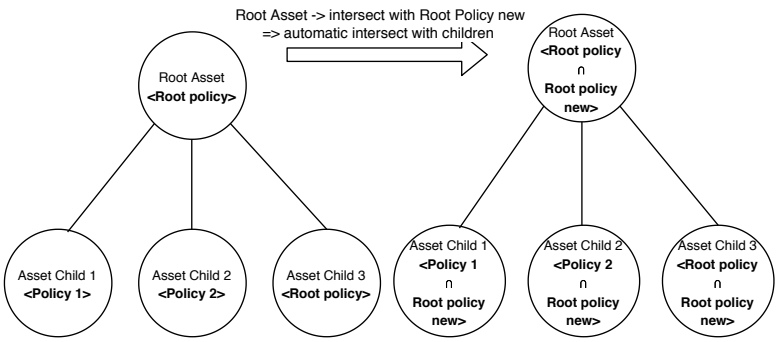


Figura 4.9: Esempio di propagazione di una intersezione di policy

Il duale delle operazioni mostrate nelle figure 4.8 e 4.9 risulta disponibile per le operazioni di unione. Sempre dalla figura 4.9 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 4.8 e 4.9. 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.

4.3.5 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:

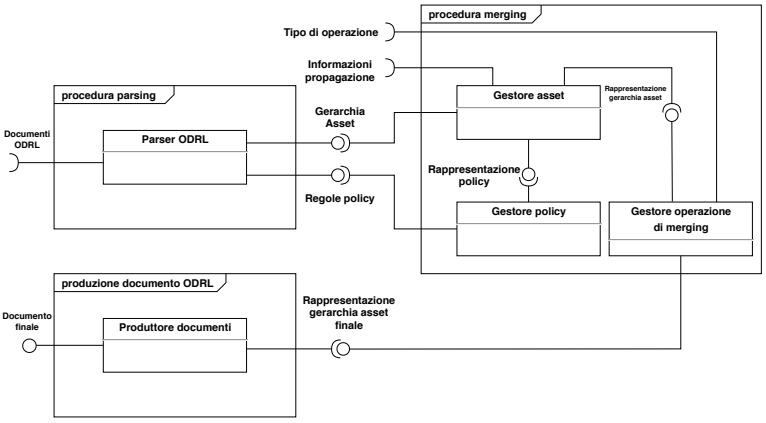


Figura 4.10: Componenti che interagiscono nel processo di merging di 2 policy ODRL

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**;
- risulta possibile decidere se propragare l'operazioni di merging sull'intera gerarchia degli asset finale, attuando un'override delle singole policy, oppure attuare la propagazione dell'operazione solo sugli asset interessati da entrambe le 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 basata sulla seguente assunzione: le gerarchie degli asset è rappresentabile attraverso 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;
    if(not(finalAssets.containsKey(uri)))
     finalAssets.put(uri,new Asset(uri));

    parentFirst = assetsFirst.get(uri).getParents();
    parentSecond = assetsSecond.get(uri).getParents();

    // Uno dei due non aveva un padre definito
    if(
    !parentFirst.equals(parentSecond) &&
    (parentFirst.contains("EveryAsset") ||
     parentSecond.contains("EveryAsset"))){
       actParents =
        parentFirst.contains("EveryAsset") ?
        assetsFirst.get(uri).getParents() :
        assetsSecond.get(uri).getParents();
     for(actParent in actParents){
       if(finalAssets.containsKey(actParent.getURI())){
        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
41      //L'asset ha gli stessi parents in entrambe le policy
42      if(parentFirst.equals(parentSecond)){
43       for(parent in parentFirst){
44        if(finalAssets.containsKey(parent.getURI())){
45         finalAssets.get(parent.getURI()).addChild(
             finalAssets.get(uri));
46        }else {
47         AssetCollection actParent = new Asset(parent.
             getURI());
48         actParent.addChild(finalAssets.get(uri));
49         finalAssets.put(actParent.getURI(),actParent);
50        }
51       }
52      }
53
54      // Almeno un parent diverso
55      if(!parentFirst.equals(parentSecond) &&
56         !parentFirst.contains("EveryAsset") &&
57         !parentSecond.contains("EveryAsset")){
58        Set totalParents = new Set(parentFirst);
59        totalParents.addAll(parentSecond);
60        for (String parentURI : totalParents){
61         if (finalAssets.containsKey(parentURI)) {
62          finalAssets.get(parentURI)
63           .addChild(finalAssets.get(uri));
64         } else {
65          AssetCollection actParent = new Asset(parentURI);
66          actParent.addChild(finalAssets.get(uri));
67          finalAssets.put(actParent.getURI(), actParent);
68         }
69      }
```

```
70    }
71
72
73    return finalAssets;
74  }
```

Listing 4.7: L'algoritmo ha prestazioni lineari in proporzione al più più grande degli insiemi di asset

All'interno del listing 4.7 è possibile notare:

- 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:

```
def Map<String, AssetCollection> mergeHierarchyUniq(
       Map<String,Asset> assets,
       AssetTree tree,
       Map<String, AssetCollection> finalAssets){

 //In questo momento finalAsset contiene asset comuni
 Set commonURI = finalAssets.keySet();
 for(String uri: assets.keySet()){
   if(!commonURI.contains(uri)){
     if(!finalAssets.containsKey(uri))
      finalAssets.put(uri,new Asset(uri));

         // Aggiunta parent similare ai comuni
   for(parent in assets.get(uri).getParents())
     if(finalAssets.containsKey(parent.getURI())){
       finalAssets.get(parent.getURI()).addChild(
          finalAssets.get(uri));
     }else {
       AssetCollection actParent =
        new Asset(parent.getURI());
       actParent.addChild(finalAssets.get(uri));
       finalAssets.put(actParent.getURI(),actParent);
     }
```

```
23      }
24      return finalAssets;
```

Listing 4.8: L'algoritmo ha prestazioni lineari in proporzione al numero di asset

L'algoritmo esposto nel listing 4.8 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 4.3.4 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 4.7, 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 4.10.
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 4.3.4.
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 4.3.3 per poi propagare ogni policy lungo la gerarchia finale, come presentato nella sezione 4.3.4.
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.

Estensione per propagazione solo su asset comuni

I passi descritti fino ad ora sono relativi alla procedura di merging che effettua un override dell'ambiente delle singole policy, di conseguenza:

- per un qualsiasi ambiente definito dalle 2 policy iniziali, la policy risultante attua una propagazione coerente con l'operazione;

- la gerarchia finale potrebbe quindi portare cambiamenti anche in porzioni del DAG non interessate da entrambe le policy effetto che, in alcuni casi, può risultare indesiderato.

Un esempio di effetto non voluto è visibile nella seguente immagine:

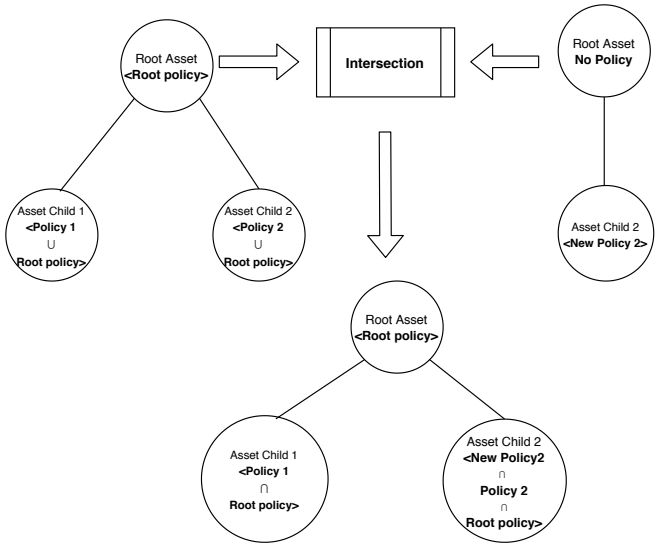


Figura 4.11: Intersezione di policy precedentemente in ambiente collaborativo

In questo caso l'effetto ottenuto sull'*Asset Child 2* è quello desiderato: l'asset non è più considerato in ambiente collaborativo ed interseca le varie policy che lo riguardano. Non si può dire lo stesso della policy finale relativa ad *Asset Child 1*: nonostante l'unica policy introdotta sia *New Policy 2*, la propagazione dell'ambiente non collaborativo produce cambiamenti anche in *Policy 1*.
La problematica appena presentata è stata trattata aggiungendo i seguenti passi all'algoritmo finora mostrato:

1. recupero degli asset che hanno subito una modifica di policy; questi asset sono quelli che hanno in entrambe le policy almeno una regola definita; questi nodi sono un sottoinsieme dei nodi comuni alle 2 policy.

2. recupero dell'intero sottografo che ha come origine un nodo modificato;

3. reset della policy originaria per i nodi recuperati nei 2 passi precedenti, senza attuare una propagazione di policy.

Il sottoalbero così recuperato è l'insieme degli asset effettivamente interessati da un'operazione di intersezione o unione. Questa affermazione può essere dimostrata come segue:

1. sia C l'insieme degli asset regolati da entrambe le policy;

2. sia M un sottoinsieme di C formato dai nodi per cui entrambe le policy presentano almeno una regola;

3. sia F l'insieme dato dall'unione dei nodi appartenti ai sottografi di tutti i nodi in M;

4. sia R l'unione di M ed F;

5. gli elementi di R sono i nodi nel sottografo interessato dall'operazione di intersezione o unione delle 2 policy;

6. supponendo per assurdo che altri nodi siano parte del sottografo:

   (a) un nodo non appartenente a C è per definizione, interessato in modo diretto solamente da una policy; si noti che F può contenere nodi non appartenenti a C;

   (b) un nodo non appartenente ad M con una sola policy definita non subisce modifiche alla sua policy, di conseguenza la propagazione della sua policy finale ai nodi figli è già stata attuata;

   (c) un nodo non appartenente ad M con entrambe le policy non definite non dà alcun contributo alla propagazione; inoltre non avendo alcuna policy, anche tutti i suoi nodi parent non hanno policy definite;

   (d) se un nodo non appartiene ad F, nessuno dei nodi parent appartiene ad F o M, di conseguenza ha già subito eventuali propagazioni di policy ed ha già propagato le proprie regole.

   Queste modifiche producono i seguenti effetti:

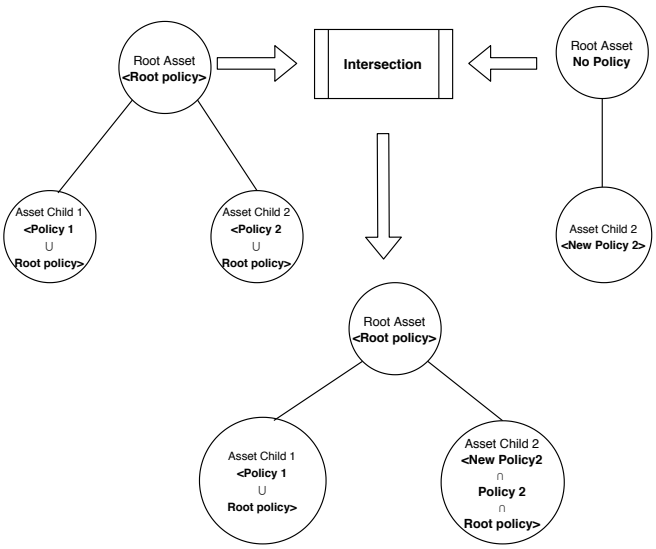


Figura 4.12: Intersezione di policy precedentemente in ambiente collaborativo senza override di ambiente

4.4 Produzione policy ODRL

Il produttore di documenti ODRL è l'ultimo componente della pipeline per il policy merging. Il ruolo svolto risulta essere di comunicazione tra i wrapper prodotti dalla logica di merging ed il modulo I/O del framework Jena. Le funzionalità sono attuate mediante la seguente procedura:

1. tramite le *Core RDF API* fornite da Jena, si crea un nuovo modello RDF;
2. dal gestore gerarchia asset relativo alla policy finale, si recupera una mappa che collega l'URI di un asset alla propria policy;
3. per ogni URI:
   (a) si aggiunge al modello una risorsa relativa all'asset;
   (b) si recuperano le risorse relative ai parent degli asset;
   (c) a tutte le risorse recuperate al passo precedente si aggiunge alla proprietà *odrl:partOf* il nuovo nodo;
   (d) per ogni regola nella policy si crea un nodo;
   (e) i nodi relativi alle regole vengono inseriti nella proprietà *permission* o *prohibition* del nuovo nodo relativo all'asset;
4. viene settato al modelo il namespace relativo al contesto ODRL;

5. il modello viene utilizzato dal modulo I/O che ne crea la corrispondente serializzazione Turtle.

Il documento RDF finale prodotto ha le seguenti proprietà:

- ogni regola espressa appare una sola volta per target;
- ogni asset ha una propria definizione;
- la lista mostrata nelle proprietà *partOf* non contiene ripetizioni.

Capitolo 5: Valutazione dei risultati

I principali obiettivi che questo lavoro di tesi si è posto sono stati esposti nella sezione 1.1.3 e riguardano principalmente 2 aree:

1. l'assenza da parte di ODRL di un metodo per trattare policy conflittuali all'interno dei casi d'uso individuati da MOSAICO;

2. controllo ineficciente di una regola, poiché con query puntuali sul documento ODRL è possibile ottenere informazioni non corrette sulla regolamentazione di un'azione; attualmente risulta necessario controllare l'intero file per avere certezze su quanto è permesso su di un asset.

Il tool sviluppato affronta entrambe queste problematiche, risolvendole nel caso di azioni prive di **constraint**. La soluzione proposta non è applicabile solamente ai casi d'uso di MOSAICO, ovvero quello di mercati digitali dove gli asset possono avere più possessori, ma anche a scenari più tradizionali, come ad esempio la definizione di policy d'accesso a risorse in un sistema operativo.
Per risolvere la prima problematica, il focus principale è stato lo sviluppo di procedure che supportassero 2 operazioni di merging: **unione** ed **intersezione**. Tali operazioni identificano i 2 principali ambienti dove è possibile definire policy d'accesso: **collaborativo** e **non collaborativo**.
La seconda problematica è stata affrontata su più livelli:

- a livello di **singola** policy: durante la lettura di un documento ODRL che definisce una policy d'accesso, il framework si occupa di inferire quelli che sono le **implicazioni** tra le regole, seguendo le definizioni date dall' *ODRL Common Vocabulary*. In questo modo si impediscono ambiguità su regole interne alla policy, dove una sottoazione è permessa ma non l'azione padre è invece vietata;

- a livello di **asset**: durante la lettura del documento di ODRL, il framework si occupa di inferire, in base all'ambiente specificato per la policy, le regole che un asset eredita dai propri parent. Ottenute queste regole, si attua il livello su singola policy.

Entrambi i procedimenti appena esposti vengono attuati dall'architettura osservabile in figura 4.10, i cui componenti sono stati esposti in dettaglio all'interno del capitolo 4 relativo all'implementazione della soluzione. L'output della pipeline è un nuovo documento di policy ODRL che rappresenta il risultato dell'operazione desiderata.
Si può notare come la soluzione sia modulare: non si è costretti ad utilizzare il tool unicamente per procedure di merging, poiché risulta possibile sfruttare ogni componente, e relativi sottocomponenti, nella pipeline in figura 4.1 senza l'ausilio degli altri. Questo permette sia lo sfruttamento delle singole funzionalità di ogni componente, sia la possibilità di estensione o sostituzione di un componente.

5.1 Esempi di casi d'uso

All'interno di questa sezione vengono mostrati casi d'uso di complessità crescente supportati dal lavoro di tesi, confrontando i risultati ottenuti effettuando query sull'output del policy merging rispetto a quello che è possibile effettuare solamente sui documenti ODRL originali.

5.1.1 Merging di policy relative ad un singolo asset

Casistica più semplice in assoluto, utile per comprendere il comportamento del tool e cosa si intende con procedura di merging. Di seguito i file ODRL utilizzati come input:

```
{
   "uid":"http://singleExample1",
   "type":"Policy",
   "@context":"http://www.w3.org/ns/odrl.jsonld",
   "permission":[

      {
         "target":"http://Child2",
         "action":"display"
      },
      {
         "target":"http://Child2",
         "action":"stream"
      }
   ],

```

```
    "prohibition":[
        {
            "target":"http://Child2",
            "action":"uninstall"
        }
    ]
}
```

Listing 5.1: Policy ODRL su singolo asset

```
{
    "uid":"http://singleExample2",
    "type":"Policy",
    "@context":"http://www.w3.org/ns/odrl.jsonld",
    "permission":[

        {
            "target":"http://Child2",
            "action":"play"
        },
        {
            "target":"http://Child2",
            "action":"install"
        }
        ,
                {
                    "target":"http://Child2",
                    "action":"uninstall"
                }
    ],
    "prohibition":[
        {
            "target":"http://Child2",
            "action":"delete"
        }
    ]
}
```

Listing 5.2: Policy ODRL su singolo asset

Di seguito, invece, vi sono i file prodotti, 1 per possibile operazione:

```
@prefix odrl:  <http://www.w3.org/ns/odrl/2/> .

<http://single>  a odrl:Set ;
 odrl:permission   ( [ a odrl:Permission ;
                       odrl:action  odrl:display ;
                       odrl:target  <http://Child2>
                     ]
                    ) ;
odrl:prohibition  ( [ a odrl:Prohibition ;
                      odrl:action  odrl:uninstall ;
                      odrl:target  <http://Child2>
                     ]
                     [ a odrl:Prohibition ;
                       odrl:action  odrl:delete ;
                       odrl:target  <http://Child2>
                       ]
                    ) .

<http://Child2>  odrl:partOf  <EveryAsset> .
```

Listing 5.3: Risultato dell'intersezione delle policy nei listing 5.1 e 5.2

```
@prefix odrl:  <http://www.w3.org/ns/odrl/2/> .
<http://singleUnion>  a odrl:Set ;
odrl:permission   ( [ a odrl:Permission ;
                      odrl:action  odrl:play ;
                      odrl:target  <http://Child2>
                     ]
                     [ a odrl:Permission ;
                     odrl:action  odrl:display ;
                     odrl:target  <http://Child2>
                     ]
                     [ a            odrl:Permission ;
                     odrl:action  odrl:install ;
                     odrl:target  <http://Child2>
                     ]
                     [ a odrl:Permission ;
```

```
                        odrl:action  odrl:stream ;
                        odrl:target  <http://Child2>
                      ]
                  ) ;
odrl:prohibition  ( [ a odrl:Prohibition ;
                      odrl:action  odrl:uninstall ;
                      odrl:target  <http://Child2>
                      ]
                      [ a odrl:Prohibition ;
                      odrl:action  odrl:delete ;
                      odrl:target  <http://Child2>
                      ]
                      ) .

<http://Child2>  odrl:partOf  <EveryAsset> .
```

Listing 5.4: Risultato dell'unione delle policy nei listing 5.1 e 5.2

Il risultato è riassunto dalla seguente immagine:

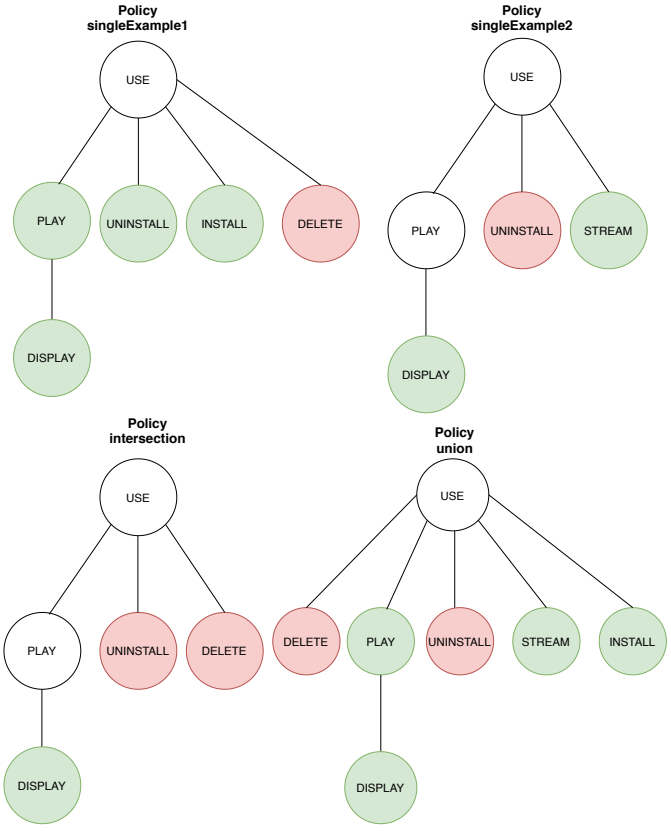


Figura 5.1: Albero delle regole delle varie policy all'interno dell'esempio

Vantaggi osservabili utilizzando i file mostrati nei listing *unionSingle* e *intersectionSingle*:

- risulta possibile ottenere la regolamentazione di un'azione mediante una query puntuale accedendo ad un singolo file. L'implementazione baseline è invece costretta ad accedere ad almeno 2 file;

- l'implementazione baseline è anche costretta a controllare l'intera gerarchia dell'azione per cui si esegue il check, effettuando un'ulteriore accesso al vocabolario ODRL;

- la procedura di merging può essere considerata computazionalmente più onerosa rispetto a quella baseline, poiché aggiunge overhead relativi alla gestione della gerarchia degli asset, inutilizzati in questo caso;

- l'overhead aggiunto al punto precedente è però considerarsi un costo pagato una singola volta, a discapito invece dell'approccio baseline che deve ogni volta ricercare le informazioni relative agli asset.

Il vantaggio principale dell'approccio adottato all'interno della tesi risiede nell'aver prodotto un documento non ambiguo e concentrato delle informazioni relative alle policy d'accesso. Tale costo è pagato solo all'inserimento di un nuovo documento ODRL, in cambio di una procedura più efficiente nel lungo termine.

5.1.2 Merging di policy relative ad asset con gerarchia ad albero

Questa casistica risulta la seconda meno complessa: le 2 policy lavorano sugli stessi asset, i quali hanno una struttura gerarchica ad albero, ovvero ogni nodo ha un solo padre possibile. Nella trattazione di questo caso d'uso viene dato per scontato il corretto funzionamento di intersezione tra singole policy, focalizzandosi invece su quello che è il comportamento del tool nel propagare le policy.
La casistica più interessante da studiare risulta essere l'*intersezione* di 2 ambienti *non collaborativi*.

	Root Policy 1	**Root Policy 2**	**Child 1 Policy 1**	**Child 1 Policy 2**	**Child 2 Policy 1**	**Child 2 Policy 2**
play						
diplay						
install						
annotate						
delete						
stream						

Tabella 5.1: Regolamentazione delle policy prima dell'applicazione di ambiente non collaborativo e intersezione

La struttura della gerarchia dello scenario è visibile in figura 5.2. In caso di ambiente **non collaborativo** si attua un'intersezione tra la policy del nodo padre ed i suoi figli. L'operazione d'intersezione finale interseca le varie policy ad ogni livello, per poi andare ad attuare la propagazione di ambiente **non collaborativo** sull'intero albero. Il risultato finale in questo caso è quanto segue:

	Root	**Child 1**	**Child 2**
play			
diplay			
install			
annotate			
delete			
stream			

Tabella 5.2: Regole all'interno della policy prodotta in output

Dalla tabella 5.2 si nota che:

- una policy sulla root mantiene i suoi permessi solo se presenti in entrambe le policy intersecate, per effetto dell'operazione di **intersezione**;
- una policy su un nodo figlio mantiene i suoi permessi se presenti in entrambe le policy intersecate e se anche la policy finale del nodo padre presenta tali permessi, quest'ultimo effetto è dato dall'ambiente **non collaborativo**.

Questa casistica risulta la maggiormente interessante poiché:

- l'intersezione di 2 ambienti collaborativi in questa situazione produce come risultato la propagazione della policy della root a tutti i nodi figli, ciò è causa-

to dall'utilizzo dell'operazione di unione preliminare poi seguita da quella di intersezione;

- i 2 casi relativi all'unione di policy in 2 ambienti sono poco interessanti, poiché non avviene alcun filtraggio particolare delle regole.

Un'ulteriore motivazione d'interesse verso questa casistica è il fatto che può essere considerata una prima modellizzazione degli scenari d'interesse di MOSAICO: un eventuale utente che possiede l'*asset child 1*, qualora volesse modificare la propria policy d'accesso, scatenerebbe questa casistica(salvo la propagazione dell'intersezione verso gli altri asset).
In questo caso il tool sviluppato porta i medesimi vantaggi già mostrati nel caso precedente, aggiungendo i seguenti:

- in questo caso risulta necessaria un'ulteriore query da parte della baseline per recuperare la gerarchia relativa all'asset interessato dall'accesso;

- a seconda della tipologia di nodo, risulterà quindi necessario recuperare anche lo stato dell'azione all'interno della gerarchia dell'asset, tenendo conto del fatto che non è stata attuato alcun tipo di propagazione nei documenti originali;

- il costo dell'esecuzione dell'approccio baseline in questo caso si avvicina molto a quello che si ha per attuare il merging di policy, rendendo evidente il vantaggio portato dalla produzione di una policy output;

- infine, l'approccio implementato permetterebbe ad un owner degli asset di valutare la perdita di valore introdotta da una modifica alla propria policy d'accesso.

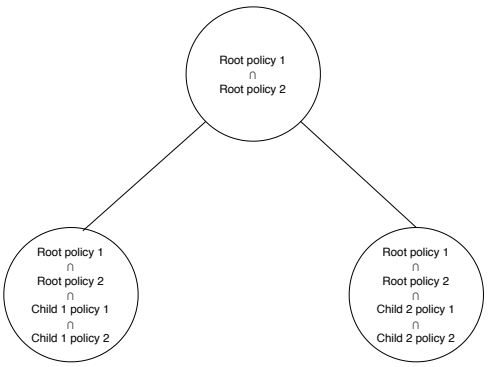


Figura 5.2: Policy finali del caso d'uso

5.1.3 Merging di policy relative a gerarchie differenti

In questo scenario, le 2 policy operano su gerarchie parzialmente sovrapposte. La casistica trattata sarà in questo caso l'*intersezione* di ambienti *collaborativi*. La procedura seguita dall'algoritmo è riassunta nella seguente figura: Questa casistica assume

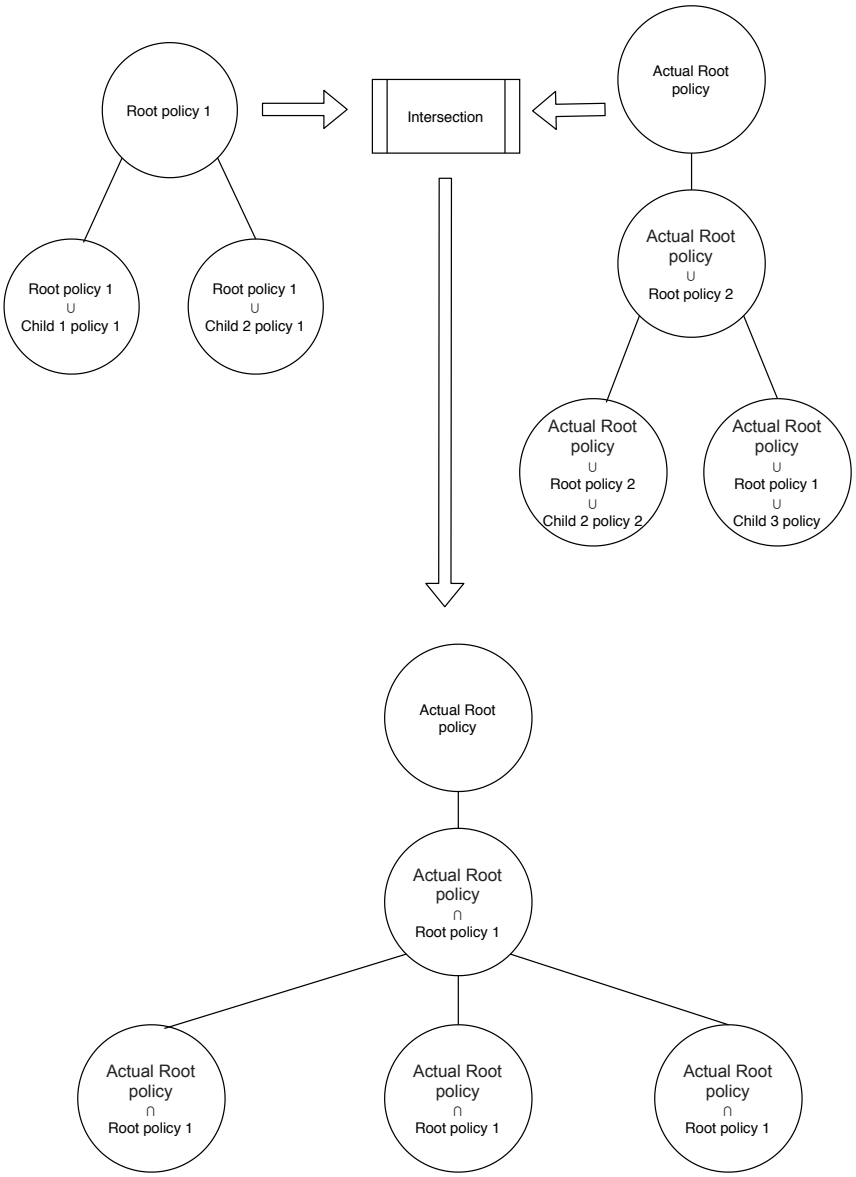


Figura 5.3: Procedura seguita per l'intersezione di 2 policy in ambiente collaborativo con l'aggiunta di nuovi nodi padre e foglia

maggiore interesse rispetto allo scenario precedente, poiché l'aggiunta di nuovi parent all'interno della gerarchia, modifica quello che è il comportamento dell'unione seguita da intersezione: se nel caso precedente questo portava alla replica dell'intersezione dei nodi radice all'interno delle foglie, in questo caso le differenze nella gerarchia fanno in modo che cambi il punto dal quale inizia la copia delle policy. Si noti che le in-

tersezioni a cui fa riferimento la figura 5.3 sono relative ai permessi delle policy, che vengono "mascherati" dalla sequenza unione-intersezione; ciò vuol dire che i divieti delle policy non visibili sono comunque propagati lungo l'albero.
I vantaggi introdotti in questo caso sono analoghi a quelli dello scenario precedente, poiché si tratta semplicemente di un'estensione a livello logico delle stesse casistiche.

5.1.4 Merging di policy relative ad asset con gerarchia complessa

Quest'ultimo scenario mostrato risulta essere il più complesso dal punto di vista della gerarchia degli asset, non più rappresentata da un albero, ma da un DAG. L'esempio è caratterizzato dalle seguenti proprietà:

- ambienti iniziali collaborativi;
- intersezione di policy;
- aggiunta di 2 parent da parte di una policy;
- aggiunta di una nuova foglia da parte di una policy;
- valutazione dei risultati utilizzando sia l'algoritmo senza e con l'estensione introdotta nella sezione 4.3.5.

La struttura precedente al merging è visibile nella figura sottostante:

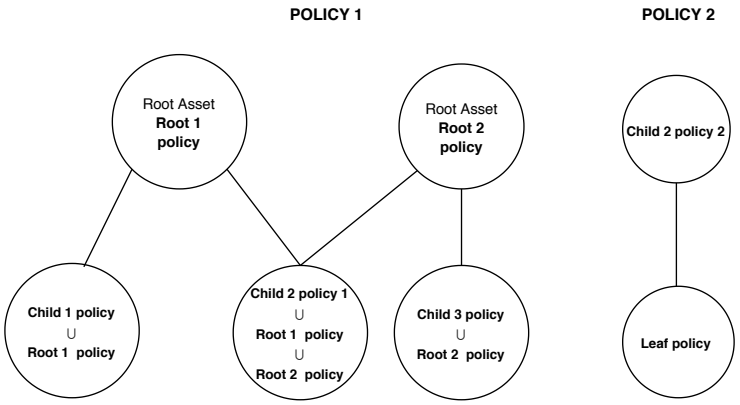


Figura 5.4: Struttura dei DAG regolati dalle policy

All'interno della tabella 5.3 sono presenti i permessi utilizzati come esempio. Le regole sono da intendersi come input allo strumento e, quindi, precedenti all'applica-

zione dell'ambiente collaborativo. Le tabelle 5.4 e 5.5 contengono invece i permessi nella policy prodotta a seconda del metodo di propagazione scelto.

	Root 1	**Root 2**	**Child 1**	**Child 2 Policy 1**	**Child 2 Policy 2**	**Child 3**	**Leaf**
play							
diplay							
install							
annotate							
delete							
stream							
anonymize							
translate							

Tabella 5.3: Permessi espresi esplicitamente dalle 2 policy di cui si fa l'intersezione

	Root 1	**Root 2**	**Child 1**	**Child 2**	**Child 3**	**Leaf**
play						
diplay						
install						
annotate						
delete						
stream						
anonymize						
translate						

Tabella 5.4: Permessi in seguito all'intersezione con propagazione dell'ambiente sull'intero DAG

	Root 1	**Root 2**	**Child 1**	**Child 2**	**Child 3**	**Leaf**
play						
diplay						
install						
annotate						
delete						
stream						
anonymize						
translate						

Tabella 5.5: Permessi in seguito all'intersezione con propagazione solo su asset interessati dal merging

Come risulta possibile notare dalla tabella 5.4, forzare l'ambiente in base all'operazione di merging effettuata può portare ad effetti indesiderati su porzioni del DAG non interessate dalla procedura:

- l'asset *Child 1* perde il permesso relativo all'azione *annotate*;
- l'asset *Child 3* perde il permesso relativo all'azione *translate*.

All'interno dello strumento viene comunque lasciata questa possibilità poiché questo caso d'uso può avere comunque delle applicazioni.
La tabella 5.5 mostra invece quello che è l'effetto desiderato la maggior parte delle volte:

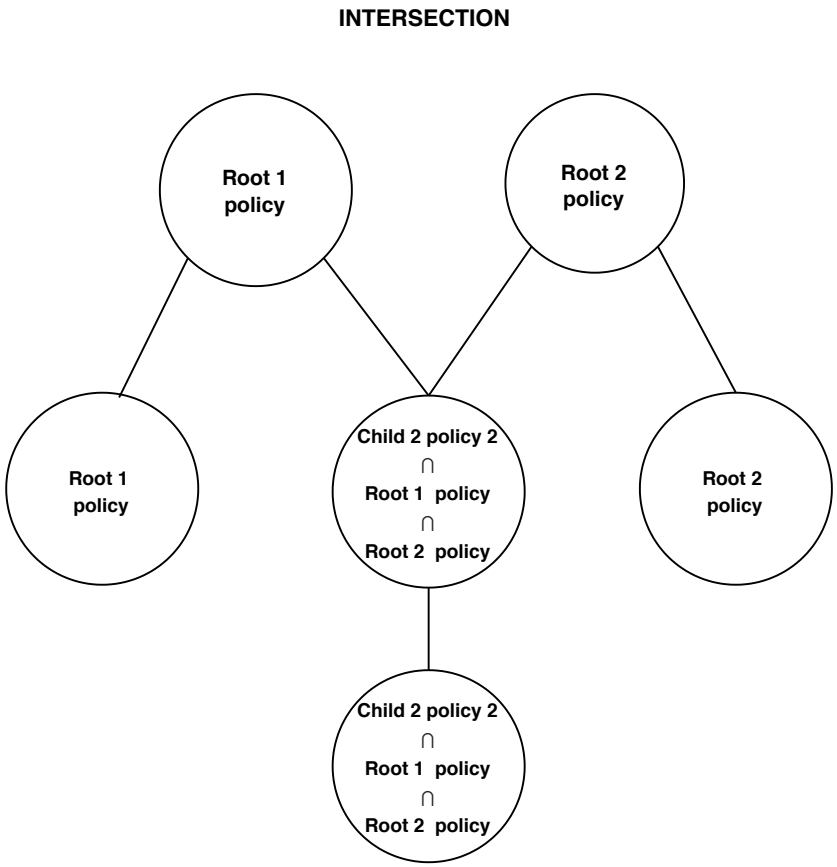
- solamente l'asset *Child 2* perde le i permessi che potrebbe mantenere in ambiente collaborativo, ovvero *anonymize*;
- le risorse restanti mantengono le policy in ambiente collaborativo;
- i nodo figlio di *Child 2*, pur non essendo interessato direttamente dall'operazione di intersezione, risulta in ambiente non collaborativo.

La procedura per ottenere gli effetti esposti nelle tabelle 5.4 e 5.5 sono riassunti nella figura 5.5.

Lo scenario mostrato risulta interessante poiché è possibile notare quali siano gli effetti dati dalla più ampia variaretà di casistiche possibile. Lo strumento risulta in grado di trattare queste casistiche, dando una base per l'interrogazione efficiente anche con relazioni complesse tra gli asset.
Quest'ultimo caso d'uso non è però l'effettiva casistica presa in esame da MOSAICO. La caratterizzazione di quel caso sarebbe la seguente:

- ambienti iniziali non collaborativi, dato da collezioni multi owner;
- intersezione di policy, sempre poiché si parla di collezioni multi owner;
- aggiunta di 1 parent da parte di una policy, rappresentante la collezione di cui si fa parte nel mercato digitale;
- aggiunta o modifica di una foglia da parte di una policy, rappresentante la collezione dati dell'owner che vuole aggiungersi al mercato;
- necessario l'utilizzo della procedura di merging estesa.

Questa casistica risulta però più limitata da un punto di vista della presentazione dei risultati raggiunti: non vi è un cambiamento di ambiente e la gerarchia può essere espressa mediante un albero. Infine, per MOSAICO, risulta interessante solamente l'utilizzo dell'algoritmo esteso: cambiamenti portati da un owner non devo avere effetti su dati da lui non posseduti.


INTERSECTION
Root 1 policy
Root 2 policy
Root 1 policy
Child 2 policy 2 ∩ Root 1 policy ∩ Root 2 policy
Root 2 policy
Child 2 policy 2 ∩ Root 1 policy ∩ Root 2 policy

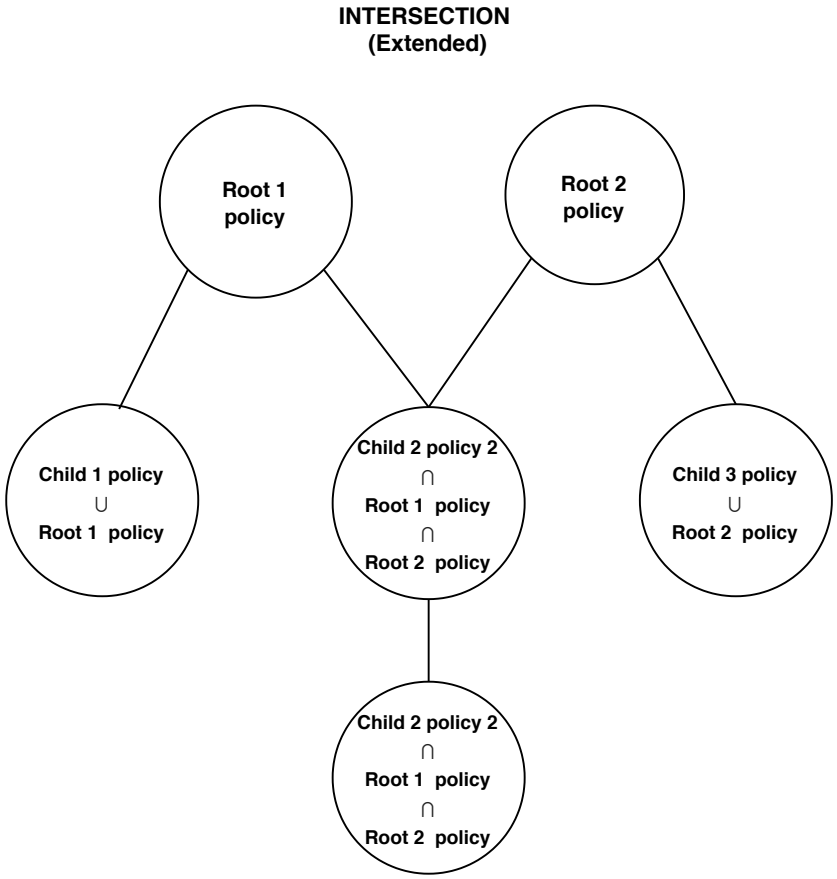

INTERSECTION (Extended)
Root 1 policy
Root 2 policy
Child 1 policy ∪ Root 1 policy
Child 2 policy 2 ∩ Root 1 policy ∩ Root 2 policy
Child 3 policy ∪ Root 2 policy
Child 2 policy 2 ∩ Root 1 policy ∩ Root 2 policy

Figura 5.5

5.2 Possibili sviluppi futuri

TODO: trattazione delle regole con constraint TODO: Aggiungere supporto alla cronologia delle modifiche TODO: supporto al deployment

Bibliografia

[1] David Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language.

[2] Chris Bizer. LinkedData.

[3] Dan Brickley and R.V. Guha. RDF Schema.

[4] The Apache Software Foundation. Apache Jena.

[5] ODRL Community Group. ODRL Community Group web page.

[6] OWL Working Group. Web Ontology Language.

[7] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language.

[8] Renato Iannella, Michael Steidl, Stuart Myles, and Víctor Rodríguez-Doncel. ODRL Version 2.2 Ontology.

[9] Renato Iannella, Michael Steidl, Stuart Myles, and Víctor Rodríguez-Doncel. ODRL Vocabulary & Expression 2.2.

[10] Renato Iannella and Serena Villata. ODRL Information Model 2.2.

[11] Eric Miller and Bob Schloss. Resource Description Framework (RDF).

[12] W3C Permissions and Obligations Expression Working Group. ODRL Implementation.

[13] W3C Permissions and Obligations Expression Working Group. ODRL Candidate Recommendation - Implementation Report.

[14] Víctor Rodríguez-Doncel. ODRLAPI: A Java API to manipulate ODRL2.0 RDF expressions.

[15] Víctor Rodríguez-Doncel and Guillermo Gutierrez Lorenzo. ODRL Editor.

[16] Oreste Signore. Introduzione al Semantic Web.

[17] Benedict Whittam Smith and Víctor Rodríguez-Doncel. ODRL Implementation Best Practices.

[18] DBPedia team. DBPedia.

[19] W3C. ODRL context file, formato JSON-LD.

[20] W3C. Semantic Web.