Aggiunta della semantica per merging di policy in ODRL

Gianluca Oldani

28 giugno 2020

Indice

Capitolo 1: Introduzione

Questo lavoro di tesi si inserisce all'interno del contesto del progetto europeo *Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNer control* (MOSAICrOWN). Come da titolo, il progetto nasce come risposta all'esigenza di ricercare metodi per la realizzazione di mercati digitali relativi allo scambio di dati, garantendo il rispetto delle norme vigenti in materia di privacy ed eventuali requisiti di riservatezza espressi dagli utenti finali in un ambiente multi-owner. Questa tesi si inserisce all'interno del pacchetto di lavoro *Policy specifications - Data governance framework*, il quale si occupa della definizione di un linguaggio, e relativo modello, per la definizione di requisiti di privacy; questo componente risulta critico poiché interessa l'intero ciclo di vita del dato, oltre ad influenzare l'operato degli altri due *work package*. Tra i candidati esaminati per la definizione di questi metadati vi è l' *Open Digital Rights Language* (ODRL). Inizialmente, il lavoro di tesi si occupa di esaminare la struttura del linguaggio ed evidenziarne due problematiche all'interno del contesto di MOSAICrOWN. La prima problematica risulta essere l'assenza di una gestione dei conflitti efficace in scenari multi-owner; la seconda problematica risulta essere l'inefficiente rappresentazione delle regole in fase di interrogazione del modello.
Individuate queste problematiche, la tesi mostra quali sono le tecnologie ed i principi che hanno guidato l'implementazione di una soluzione. La proposta consiste in un tool che attua il *merging* di due policy qualora queste operino sulla stessa risorsa; prima di procedere all'effettiva implementazione, sono state individuate le varie casistiche possibili ed esempi degli scenari in cui si presentano, al fine di caratterizzare al meglio la problematica.
Dopo questa caratterizzazione, il lavoro di tesi espone l'architettura della soluzione implementata, mostrando schemi UML e pseudocodice Java-like per dimostrare l'efficacia e la modularità della proposta nelle parti più importanti.
L'ultimo capitolo della tesi si concentra sulla valutazione dei risultati ottenuti, riassumendo quali fossero le problematiche affrontate e mostrando i vantaggi del tool creato tramite un confronto con una soluzione baseline in una serie di scenari dalla complessità crescente.

Capitolo 2: Il progetto MOSAICrOWN

Il progetto MOSAICrOWN nasce come risposta all'esigenza di ricercare metodi per la realizzazione di mercati digitali relativi allo scambio di dati[13]. In particolare, si inserisce nel contesto relativo alle sfide introdotte dai requisiti di privacy che è necessario garantire in queste applicazioni. Questi requisiti rappresentano ostacoli per task quali: aggregazione di dati posseduti da più attori, analisi di dati.
Gli obiettivi che il progetto si pone sono:

- peremttere lo scambio di dati in sicurezza, sia per quanto concerne la protezione di informazioni personali che informazioni sensibili in ambito aziendale;
- dare supporto a realtà in cui vi è carenza di skills in ambito gestione dati, fornendo la possibilità ai possessori delle risorse di usufruire di strumenti per la protezione, scambio ed analisi dei dati;
- standardizzare le procedure di scambio di dati tra diverse realtà in ambito europeo.

Il metodo con cui si intende realizzare questi obiettivi risulta essere la creazione di un framework che permetta la definizione di requisiti relativi alla protezione dei dati in modo dichiarativo; tali requisiti sono poi utilizzati durante l'intero ciclo di vita del dato per mettere in atto tecniche di protezione. Le procedure utilizzate devono, inoltre, risultare efficienti, scalabili ed essere realizzate con in mente uno scenario che pone l'attenzione sulla privacy in ambienti multi-owner.
Altri aspetti che il progetto MOSAICrOWN tiene in considerazione sono:

- implementazione adatta al deployment in sistemi reali, cercando di adottare le migliore tecniche che la ricerca ha da offrire nella tutela della privacy;
- indipendenza del framework dalla specifica piattaforma in cui viene impiegato, cercado di supportare come casi d'uso principali:
  - una sola entità che gestisce il mercato digitale e che offre strumenti di analisi dei dati;

  - una sola entità che gestisce il mercato digitale ma che demanda l'analisi dei dati ad entità esterne.

- varietà dei gradi di sicurezza garantiti in relazione al degrado di performance introdotto dalle tecniche di protezione impiegate, permettendo così agli utilizzatori di gestire un trade-off tra certificazioni sulla privacy e valore dei dati.

2.1 Architettura del progetto

Il progetto modella il ciclo di vita del dato individuandone tre fasi:

1. **ingestion**: il dato passa dal suo possessore al mercato digitale; in questa fase si punta a fornire tecnologie che permettano al data owner di specificare le restrizioni che devono essere rispettate sui dati inseriti nel mercato e, inoltre, si dà la possibilità di attuare tecniche che diano garanzie sul rispetto di questi requisiti; queste ultime tecnologie supportano scenari dove l'entità che possiede il mercato digitale non è un soggetto trusted.

2. **storage**: il dato è conservato e gestito dall'entità che comanda il mercato digitale; le tecnologie fornite in questa fase puntano a proteggere il dato sia durante la conservazione, sia durante l'interrogazione del dato; in questa fase risulta importante garantire l'interoperabilità con altri ambienti che non fanno uso di questi strumenti. Particolare attenzione viene data all'evitare il degrado delle performance quando si opera su dati protetti con granularità fine.

3. **analytics**: il dato è processatto all'interno del mercato digitale; in questa fase le tecnologie fornite permettono ai vari possessori di effettuare analisi sui dati in maniera efficiente, basandosi sull'entità richiedente e sui requisiti di protezione specificati per i dati.

A livello implementativo il progetto è diviso in tre *work package*:

1. **Policy specifications - Data governance framework**: questo pacchetto di lavoro si pone come obiettivo la definizione di un framework per la gestione e la specifica di policy in scenari di aggregazione dati multi-owner. Il fine è rendere possibile ai possessori dei dati la regolazione dell'uso che è possibile fare sui propri asset sviluppando un linguaggio, ed un relativo modello, per la dichiarazione di policy. Sia il modello che il linguaggio devono risultare flessibili, poiché i requisiti espressi devono riguardare ogni fase della vita del dato.

2. **Data wrapping**: questo pacchetto di lavoro si preoccupa della definizione di tecniche che permettano l'accesso ai dati in modo efficiente ma che non compromettano i requisiti di protezione specificati mediante il modello fornito nel package precedente. Particolare enfasi è data sull'evitare la re-identificazione dei dati;

3. **Data sanitisation**: quest'ultimo pacchetto di lavoro mira alla definizione di tecniche efficienti e scalabili che operino su intere collezioni di dati, al fine di offrirne una versione offuscata e/o aggregata che sia robusta ad attacchi che mirano a compremetterne la privacy e/o la confidenzialità. Questo package mira anche a fornire tecniche che permettano di valutare il grado di protezione di un dato in relazione al valore che questo mantiene.

Tutti i tre workpackage esposti devono essere impiegabili in ogni fase del ciclo di vita del dato, come mostrato nella seguente figura:

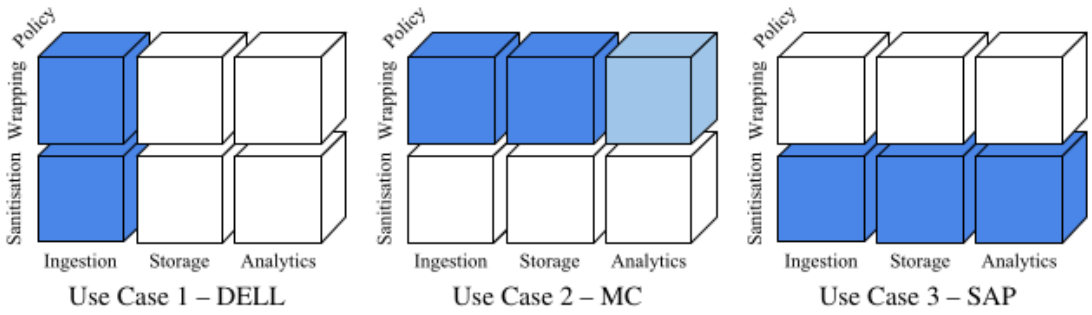

Figura 2.1: Caratterizzazione dei casi d'uso presi in considerazione da MOSAICrOWN[13]

2.2 Scenari d'uso

Lo sviluppo del progetto si basa sul soddisfare i requisiti di tre scenari individuati; ogni caso studio prende in considerazione uno specifico tipo di attore che può prendere parte ad un mercato digitale. Come visibile anche all'interno della figura 2.1, gli scenari affrontati sono:

- **DELL**: in questo caso d'uso si affronta la problematica dal punto di vista dei possessori di dati; per questi attori risulta cruciale la possibilità di definire requisiti di privacy e legislativi sui propri dati e poter sfruttare tecniche che assicurino questi requisiti anche in ambienti non trusted. Questi utenti sono interessati solamente dalla fase di *ingestion* ma sfruttano ogni tecnologia all'interno del progetto;

- **MasterCard**: in questo caso d'uso si affronta la problematica dal punto di vista di un gestore di una piattaforma per l'analisi dei dati; per questi attori è cruciale il rispetto delle norme vigenti in termini di privacy ed, allo stesso tempo, poter fornire dati di qualità anche con granularità fine; questo tipo di utenti ha il controllo sull'intero ciclo di vita di un dato, con la possibilità di far effettuare l'analisi dei dati ad altre entità; data la granularità del dato, non è necessario l'uso del layer di *data sanitisation*;

- **SAP**: in questo caso d'uso si affronta la problematica dal punto di vista di un gestore di una piattaforma per lo storage dei dati; per questi attori è cruciale il rispetto delle norme vigenti in termini di privacy e poterne misurare l'efficacia; questo tipo di utenti ha il controllo sull'intero ciclo di vita di un dato; data la granularità del dato, non è necessario l'uso del layer di *data wrapping*.

2.3 Collocamento del lavoro di tesi

Questa tesi si colloca all'interno degli studi effettuati per l'implementazione del pacchetto di lavoro denominato *Policy specifications - Data governance framework*. In particolare viene analizzato uno dei linguaggi candidati per la modellizzazione dei requisiti di privacy: l' Open Digital Rights Language (ODRL), le cui proprietà sono trattate nel capitolo 3.
Le motivazioni per cui questo linguaggio risulta un candidato interessante sono:

- l'essere uno standard definito dal W3C, patner del progetto ed importante risorsa per la standardizzazione delle soluzioni prodotte da MOSAICrOWN;

- la possibilità di implementazione mediante l'uso di linguaggi di definizione di metadati diffusi, quali XML ed JSON;

- l'essere nato come linguaggio per l'espressioni di policy, senza assunzioni di contesti specifici;

- l'essere utilizzato all'interno dell'industria del settore *Digital rights management* (DRM).

Questo *work package* è fondamentale in ogni fase di vita del dato ed, inoltre, è necessario in tutti i casi d'uso individuati; alla luce di ciò, la comprensione dei punti di forza e delle limitazioni del modello risultano aspetti cruciali per il progetto MOSAICrOWN. ODRL soddisfa sia il requisito di flessibilità che di vicinanza alle tecnologie attualmente in essere.

Capitolo 3: ODRL

3.1 Il linguaggio

3.1.1 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"[9].
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 le regole al suo interno;
- indicare eventuali vincoli riguardanti i permessi ed i divieti espressi.

Avere un modello standard per definire questi bisogni dà due vantaggi:

- chi possiede l'asset è in grado di definire in modo chiaro quali siano le azioni che un consumatore può fare, evitandone utilizzi 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 cerca di definire i concetti chiave per la creazione di policy machine-readable collegate direttamente all'asset al quale sono associate: ciò permette all'utente finale di reperire facilmente informazioni sulla risorsa alla quale si accede. Quest'ultima proprietà è fondamentale per comprendere il linguaggio, poiché ODRL è costruito seguendo i *Linked Data principles*[2]:

- utilizzo di URIs come identificativi per le risorse;
- gli URI sono indirizzi HTTP, così che si possano cercare informazioni sulle risorse;
- l'URI deve fornire informazioni utili sulla risorsa;
- tra le informazioni si cerca di fornire altri URI, in modo che l'utente possa raggiungere altre risorse.

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 inserite nel documento ODRL.

3.1.2 Il modello

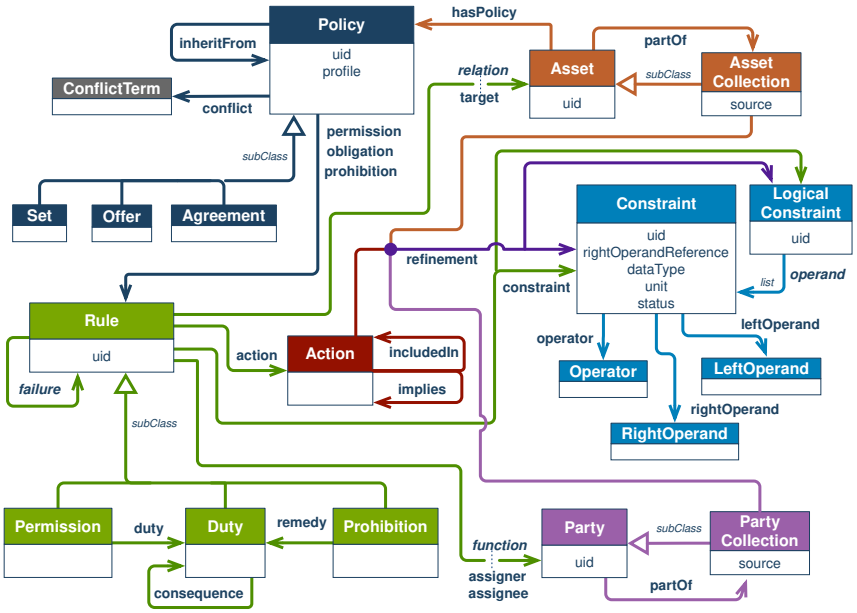

Figura 3.1: Schema del modello ODRL[9]

Come visibile all'interno dello schema in figura 3.1, 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** e **dovere**;
- **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;
- **Vincoli**: 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"[8]. 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 due vocabolari principali definiti sono:

- **ODRL Core Vocabulary**: rappresenta l'insieme minimo di termini per l'espressione di policy supportato;
- **ODRL Common Vocabulary**: arricchisce il vocabolario precedente con gruppi di *azioni*, *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 due azioni **use** e **transfer**, nel secondo caso queste azioni vengono estese da altre azioni figlie, come mostrato in figura 3.2.

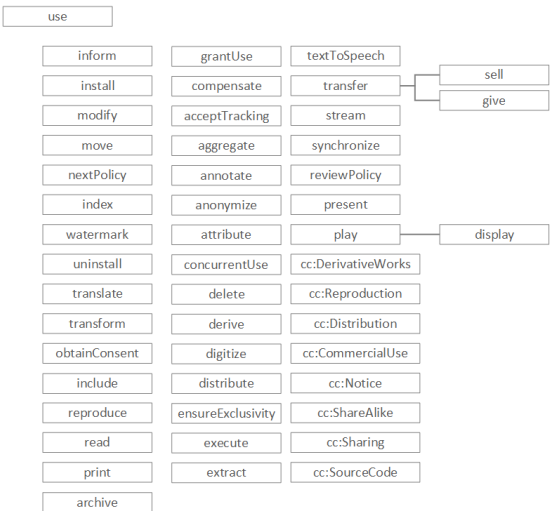

Figura 3.2: Tutte le azioni mostrate sono figlie di **use**, ad eccezione di **trasfer** e le sue sottoazioni[17]

Policy

Come definito nel modello presente nella sezione 3.1.2, 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, inserito nel campo **uid**;
- deve avere almeno una regola;
- può specificare un profilo; campo obbligatorio se non si usa il Core Vocabulary descritto nella sezione 3.1.2;
- può specificare una policy da cui eredita le proprietà;
- può specificare una strategia per la risoluzione dei conflitti.

Come visibile dalla figura 3.1, una policy ha tre 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 3.1: Policy con sottoclasse **Set**

Dalla policy mostrata nel listing 3.1 si può notare:

- @type: serve ad indicarne la sottoclasse;
- non usando termini fuori dai due vocabolari principali, non necessita la definizione di un profilo;
- l'id univoco è rappresentato da un URL che conduce ad informazioni relative alla risorsa.

Asset

Come definito nel modello presente nella sezione 3.1.2, un asset è una risorsa o una collezione di risorse soggette a regole. Un asset può essere una qualsiasi entità 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/archive:1011",
        "action": "display",
        "assigner": "http://example.com/party:0001"
 }]
}
```

Listing 3.2: Utilizzo di asset nella proprietà **target** di una regola

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

- **source**: sostituisce il campo **uid** nelle classe *AssetCollection* all'interno di un **refinement**;
- uno o più **refinement**: vincoli 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/archive:1011",
        ...
}
```

Listing 3.3: L'asset definito è parte del target presente nel listing 3.2

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 3.2 anche all'asset figlio.

Party

Come definito nel modello presente nella sezione 3.1.2, un party è una entità o una collezione di entità con funzione di assegnatario o assegnatore in una regola. Un party può essere un qualunque soggetto con un ruolo attivo nelle regole o che produce un effetto specifico; ad esempio può essere chi controlla che le azione relative ad un dovere vengano rispettate. Presenta **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 3.4: Utilizzo di party nelle proprietà **assigner** ed **assignee** di una regola

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

- **source**: sostituisce il campo **uid** nelle classe PartyCollection all'interno di un **refinement**;
- uno o più **refinement**: vincoli riguardanti la collezione che identificano solamente un sottogruppo di entità 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 3.5: 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 anche alla singola entità.

Action

Come definito nel modello presente nella sezione 3.1.2, **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 due 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; le due azioni non hanno una relazione espressa tramite *includedIn*[1].

Come anticipato nel paragrafo 3.1.2 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 3.2.
Di seguito un esempio di azione in una regola:

```
{
 "@context": "http://www.w3.org/ns/odrl.jsonld",
 "@type": "Offer",
 "uid": "http://example.com/policy:1012",
 "profile": "http://example.com/odrl:profile:06",
 "permission": [{
  "target": "http://example.com/music:1012",
  "assigner": "http://example.com/org:abc",
  "action": "play"
 }]}
```

Listing 3.6: 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 3.1.2, **constraint** è una classe usata per comparare due 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;
  - **rightOperandReference**: 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**: valore iniziale per l'elemento di sinistra.

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

- uno o nessun identificativo univoco, qualora si volesse riutilizzare l'espressione logica 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 3.7: 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 3.8: 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 3.9: 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 3.10: 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 3.1.2, **rule** è una classe astratta che raccoglie gli aspetti comuni della classi **permission**, **prohibition**, e **duty**. Rappresenta una delle regole all'interno della policy ed è caratterizzata dalle 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** : vincoli applicati alla validità della regola;
- uno o nessun identificativo univoco, necessario solo qualora si usasse 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 vincoli 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 vincoli 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 vincoli 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 3.11: 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 3.12: 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 3.13: La regola esprime l'obbligo di eseguire l'azione **compensate**, specificando come **refinement** l'ammontare del pagamento

3.1.3 Problematica trattata

Gestione dei conflitti

La trattazione di ORDL fatta fino ad ora considera la definizione di una singola policy per volta. Questo caso non rispecchia però le necessità reali che MOSAICrOWN che emergono dai casi: in questi scenari è naturale che ogni *data owner* sia interessato a definire la propria policy sulla propria porzione asset.
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** per definire una strategia di risoluzione dei conflitti; le possibili strategie di risoluzione attualmente definite sono:
  - **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; questa è la procedura di default se non si valorizza la proprietà.

Questo sistema di gestione dei conflitti risulta avere una problematica fondamentale all'interno degli scenari di MOSAICrOWN: si riferisce a conflitti all'interno di una singola policy e che, solitamente, avvengono in seguito all'utilizzo della proprietà **inheritFrom**; risulta possibile notare che in uno scenario multi-owner, l'utilizzo della proprietà **conflict** risulta inadatto. Si prenda come esempio il seguente scenario:

1. si supponga che un ospedale, indicato come A, voglia rendere disponibili i propri dati di ricerca in un mercato come quello di MOSAICrOWN;
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 un contesto single-owner, ODRL riesce perfettamente a soddisfare le esigenze dell'owner della collezione di dati: l'utente è in grado di aderire alle normative europee ed allo stesso tempo di tutelare la privacy dei propri pazienti in conformità con le leggi vigenti nel suo paese.
Nel caso in cui la collezione di dati medici appartenga a più ospedali, la soluzione proposta da ODRL 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** settata come **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 per quanto concerne il mercato, poiché scoraggia i possessori dei dati a partecipare ad una collezione;

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

Una possibile soluzione a questo problema è attuare il merging delle varie policy che hanno come target 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 MOSAICrOWN di tutela della privacy;

3. per ogni regola non in conflitto, la regola viene mantenuta, seguendo l'obiettivo di MOSAICrOWN di mantenimento della qualità ed accessibilità dei dati;

4. per ogni azione permessa solo in una policy, si invalida la relativa, poiché solo uno dei due attori coinvolti si esprime in merito.

Questo procedimento è solo uno dei tanti possibili in base alle casistiche di conflitto che possono presentarsi; questo metodo risulta in linea con i requisiti che MOSAICrOWN punta a soddisfare, poiché a differenza di quanto definito in ODRL:

- porta ad una diminuzione della qualità del dato graduale, anziché ad una invalidazione completa della policy per conflitti anche solamente parziali;

- nonostante la diminuzione della qualità del dato, preserva comunque il maggior grado di tutela della privacy indicato dalle policy interessate dal processo, poiché permette sempre e solo azioni che hanno una regola esplicita che le riguarda e nessun divieto.

Controllo inefficiente

Uno dei requisiti fondamentali che MOSAICrOWN punta a soddisfare è l'efficienza. Utilizzando il modello ODRL come mostrato fino ad ora non soddisfa questo requisito, poiché si è sempre costretti a controllare tutte le regole all'interno di una policy: ciò è causato della proprietà *includedIn*. 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 3.2, l'azione **transfer** include le azioni **sell** e **give**;
- un utente desidera eseguire l'azione **give** sull'asset interessato dalla policy;
- un algoritmo di controllo, non può limitarsi al recupero del permesso relativo all'azione **give** ma risulta costretto a controllare anche che **transfer** non sia vietata.

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 in:

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

Attuando questa procedura è possibile notare come un algoritmo di controllo possa fermarsi al primo permesso che trova, anziché controllare anche tutti i divieti presenti nella policy; inoltre non è più necessario recuperare la gerarchia completa dell'azione e può limitarsi a controlli puntuali.

[2]Si ipotizza che le azioni più generiche non siano atomiche, in caso contrario non è necessario invalidare

Capitolo 4: Tecnologie utilizzate

All'interno di questo capitolo sono trattate le varie tecnologie e i paradigmi utilizzati per implementare una soluzione alle problematiche mostrate nella sezione 3.1.3. Tali problematica necessitano lo sviluppo di una componente che attui la semantica necessaria al merging di due policy e che produca un 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.

4.1 Tool già esistenti

I risultati relativi alla ricerca di tool già realizzati hanno portato alle 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 dello stato di una regola all'interno di una policy[12], attualmente in sviluppo;
  - un validatore di policy ODRL[11], attualmente in sviluppo;
  - un editor di policy ODRL[15];
  - API per la rappresentazione tramite Java di policy ODRL[14] ma supportano ODRL 2.0.

[1] La versione utilizzata durante la stesura del lavoro di tesi è ODRL 2.2

Le problematiche di merging e rappresentazione non efficiente delle regole non sono affrontate direttamente dai tool già sviluppati, i quali tendono invece ad avere un carattere più formale anziché applicativo. Sempre grazie allo scouting è stato possibile dedurre che l'implementazione di un nuovo tool può non reimplementare i seguenti procedimenti:

- i documenti utilizzati come input possono essere considerati già validati;
- lo stato attuale degli asset può essere tralasciato, in quanto valutatori di stato sono già in sviluppo;
- non deve supportare l'utente nella creazione della policy.

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

4.2 Interrogazione e produzione documenti ODRL

Le ricerche relative a strumenti per il parsing ed la produzione di documenti ODRL, hanno portato al framework Apache Jena[4], utilizzato per creare applicazioni secondo i principi del "Semantic web"[20] e dei "Linked Data". Il framework ed i modelli di cui supporta la creazione sono descritti nelle sezioni che seguono.

4.2.1 Semantic Web

Il Semantic Web è definito come un'estensione del World Wide Web, nel quale i dati all'interno della rete sono *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 o 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)[10] mentre si lascia libertà sulla semantica, la quale può venir definita mediante vocabolari specifici rispetto al dominio di utilizzo realizzati mediante RDF Schema (RDFS)[3]. Sia RDF che RDFS sono descritti in dettaglio nelle prossime sezioni.
Utilizzando come esempio il listing 3.2, 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 modo di recuperare informazioni utili su tutte le risorse trattate; l'informazione fornita risulta quindi non centralizzata poiché recuperabile con query su varie risorse o semplicemente allegando il metadato a più risorse.

4.2.2 Linked Data

I principi mostrati fino ad ora forniscono linee guida su come rendere ogni risorsa *machine-readable*, permettendo ad una macchina di ottenere maggiori informazioni sul dato.
Il Semantic Web non cerca solamente di migliorare la qualità dell'informazione di una singola risorsa; un ulteriore obiettivo è fornire ad un applicativo la possibilità di consultare informazioni relative anche a risorse collegate a quella iniziale.
Questo secondo fine viene attuato mediante i quatrro principi dei Linked Data, già enunciati nella sezione 3.1.1.
Un importate risultato raggiunto in questo ambito risulta essere DBPedia[18], progetto che permette:

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

Come già citato, questo modello è ben visibile all'interno di ODRL, per esempio nel listing 3.1 è possibile notare:

- un URI proprio della policy, la quale risulta, quindi, sia un metadato che una risorsa consultabile;

- essendo un documento ODRL una 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 interpretare correttamente una policy;

4.2.3 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"[10]. Tra i casi d'uso supportati da RDF vi sono:

- recupero di risorse web per ottenere migliori motori di ricerca;
- catalogazione di contenuti 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 all'interno del framework è definita come risorsa qualsiasi entità identificabile mediante un URI.
Tra le caratteristiche di RDF vi è il proporre una sintassi che descriva le risorse indipendentemente dal dominio applicativo:

- non vi sono assunzioni specifiche sull'area di applicazione;
- non vi sono restrizioni di semantica, definibile in base al dominio.

La sintassi mostrata di seguito 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 raffinare 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 dagli utenti finali.

Il modello RDF

Il modello RDF è definito come segue[10]:

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* all'interno della tripla. Per esempio:

1. la frase
   Ora Lassila** è l'**autore** della pagina web **http://www.w3.org/People/Lassila
2. è serializzata tramite la tripla
   {autore, [http://www.w3.org/People/Lassila], "Ora Lassila"}
3. dalla definizione 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 supportato da RDF; quanto segue mostra come si possano creare relazioni tra le varie risorse:

1. la frase
   Ralph crede che il libro "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 le triple definite all'interno del documento. In ogni tripla è possibile individuare:

- **predicato**: nodo appartente a P;
- **oggetto**: nodo sorgente dello statement; appartiene a N ed è il nodo a cui si applica il predicato;
- **valore**: nodo destinazione dello statement; appartiene a N, 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 utilizzando XML come linguaggio:

```
<?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 4.1: 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 questo elemento XML indica invece il valore della tripla.

Di seguito la rappresentazione grafica del grafo codificato:

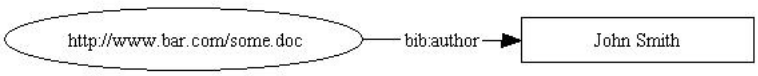

Figura 4.1: Rappresentazione grafica del grafo codificato nel listing 4.1

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

```
<?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 4.2: 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 4.3: Esempio di metadato conforme ad RDF in XML

Entrambi gli esempi mostrati nei listing 4.2 e 4.3 hanno la seguente rappresentazione grafica:

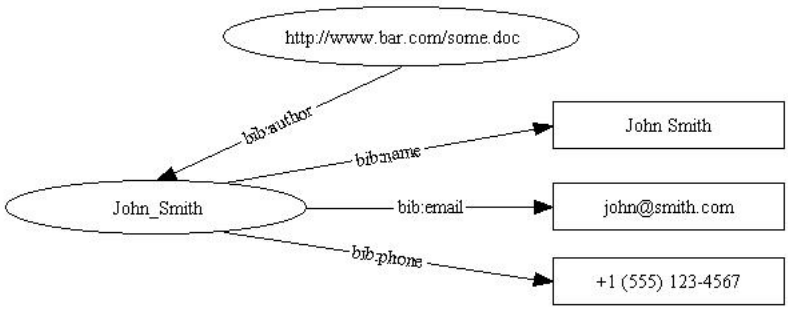

Figura 4.2: Rappresentazione grafica del grafo codificato nei listing 4.2 e 4.3

Tag namespace

Il tag *namespace* è un'estensione alla specifica RDF attualmente non obbligatorio. Questo tag è stato introdotto per rendere la sintassi di un tag specifico al dominio applicativo meno prolissa:

- **as**: proprietà del tag che indica il nome del namespace;
- **href**: porzione di identificativo da sostituire al valore della proprietà *as* quando lo si incontra.

Prendendo ad esempio il listing 4.2: il tag *bib:name* è da interpretare come:
`http://docs.r.us.com/bibliography-info/name`
il valore così ottenuto risulta essere l'URI che identifica la proprietà; è anche un riferimento a dove è possibile recuperare informazioni su di essa.
L'utilizzo di questa pratica è diffuso anche in ODRL: nel listing 3.4 il tag *@context* svolge un ruolo simile: fa riferimento ad un file RDF nel 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 4.4: Estratto context ODRL[19]

Il formato di serializzazione utilizzato nel listing 4.4 non è XML, bensì JSON-LD, lo stesso utilizzato nei vari esempi presentati nel capitolo 3 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 supportata dal framework Jena.

4.2.4 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 tra loro e le relazioni tra i vari gruppi. RDFS basa il suo funzionamento sulla definizione di due termini:

- **rdfs:Class**: raggruppamento di risorse con le stesse proprietà, come ad esempio la classe *odrl:Policy*;
- **rdf:Property**: relazione tra risorse oggetto e risorse valore di una tripla; istanze di questa particolare classe possiedono a loro volta le seguenti proprietà:
  - **rdfs:range**: i *valori* in una tripla di questa *Property* devono essere istanze delle classi specificate in questa proprietà;
  - **rdfs:domain**: gli *oggetti* in una tripla di questa *Property* devono essere istanze delle classi specificate in questa proprietà.

Oltre a quanto appena mostrato, esistono anche due proprietà denominate: **rdfs:subClassOf** e **rdfs:subPropertyOf**. Queste *Property* permettono di definire un sistema di ereditarietà tra classi e tra proprietà; 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 4.5: Estratto vocabolario ODRL[8]

L'esempio appena mostrato è la definizione della sottoclasse *Prohibition*, esposta nella sezione 3.1.2. Un secondo esempio, sempre visibile in ODRL, è la definizione della proprietà "odrl:partOf", mostrata nei listing 3.3 e 3.5, che presenta:

- due possibili *rdfs:domain* e due possibili *rdfs:range*;
- solo due coppie *rdfs:domain-rdfs:range* sono valide; questa restrizione è espressa mediante OWL[6], ulteriore estensione semantica utilizzata per espriremere concetti logici complessi;
- i valori possibili per queste proprietà sono le due classi *odrl:Party* ed *odrl:Asset*.

4.2.5 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 composto 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 6.2, sia a supporto della porzione di produzione del documento ODRL finale, descritta nella sezione 6.4. 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 principale fornita da questo componente è denominata *Model* e denota un intero grafo RDF. Questa interfaccia è utilizzata per operazioni con alto livello di astrazione sull'intero modello, come per esempio:

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

Jena schemagen
Questo tool 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 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;
- un'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 ne è stata sfruttata la versione standalone.

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 le interrogazioni tramite Java ; SPARQL è il 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 4.6: 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 4.7: Query SPARQL per il recupero dei titoli che iniziano per SPARQL

Il risultato della query nel listing 4.7 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 4.8: 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 4.9: Query SPARQL per il recupero dei nomi e delle mail solo per soggetti che presentano entrambe le proprietà

Il risultato della query nel listing 4.7 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 4.8 non sia stata utilizzata per costruire il risultato della query, poiché l'oggetto *_:c* non ha la proprietà *foaf:name*.
Il ruolo che hanno le API ARQ all'interno di del progetto è di collegamente tra le query e le *CORE RDF API*; introducono anche una semplificazione della sintassi evitando il costrutto *PREFIX* dove è noto il 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 serializzazioni mostrate in lettura.

Il modello espresso mediante *CORE RDF API* ottenuto 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. 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 alcune modifiche per essere resi più leggibili dagli utenti finali. Non sono invece necessari adattamenti per l'utilizzo da parte di un programma.

Capitolo 5: Valutazione casistiche

5.1 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 5.1: 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 5.2: 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 indipendenti

```
{
 "@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 5.3: 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 5.4: 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 ad azioni tra loro scollegate. Un eventuale merging delle due policy avrebbe due 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 5.5: 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 5.6: La policy 0002 proibisce la riproduzione dell'asset 1212 da parte del soggetto Alice

In questo caso le regole espresse dalle due policy sarebbero in conflitto, poiché *use*, permesso dalla prima policy, comprende al suo interno anche *play*, proibito dalla seconda. Siccome però le due regole sono applicate a due soggetti diversi, un eventuale merging delle policy avrebbe due regole distinte uguali a quelle di partenza. Questa casistica non è stata ulteriormente esplorata all'interno del lavoro di tesi poiché:

- la procedura di merging implementata utilizza come semplificazione regole riferite ad **EveryOne**; questo perché il progetto MOSAICrOWN ha il suo focus sui dati;
- un'eventuale procedura che tenga conto anche degli utenti aggiungerebbe solo un filtraggio preliminare sulle entità cui si riferiscono le regole.

Conflitti senza strategia risolutiva indicata

Nel caso due policy siano in conflitto quando non viene indicato alcun processo di risoluzione dei conflitti, ODRL invalida entrambe le policy. Attuando policy merging è possibile formulare una terza policy che racchiude entrambi i requisiti espressi dai documenti originali.

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 5.7: 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 5.8: 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 due policy dovrebbero essere considerate non valide. 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), al fine di segnalare il processo di merging avvenuto.

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 5.9: 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 5.10: 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), al fine di segnalare il processo di merging avvenuto.

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 5.11: 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 5.12: La policy 0002 proibisce la vendita dell'asset 1212 al soggetto Alice

In questo caso, l'azione *transfer* è inclusa in due sotto-azioni: *give* e *sell*; delle due 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 5.13: 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

Risulta possibile fare le seguenti valutazioni in merito a questa casistica:

- il numero di regole all'interno della policy ottenuta per merging non è necessariamente minore rispetto al numero di regole di partenza;
- risulta necessario identificare se le sotto-azioni sono le uniche azioni atomiche possibili o se anche le azioni padre possano essere considerate atomiche; all'interno dell'implementazione è stata scelta la prima convenzione;
- se *prohibition* e *permission* fossero invertite tra la policy 0001 e la 0002, la policy 0003 risulterebbe una policy con solo divieti.

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 in sezione 5.1. Per quanto concerne le altre due casistiche, come già esposto nella sezione 3.1.2, lo standard ODRL risulta inadatto al trattamento di conflitti; questa proprietà viene quindi ignorata dal procedimento di merging.

Capitolo 6: Implementazione

6.1 Architettura generale

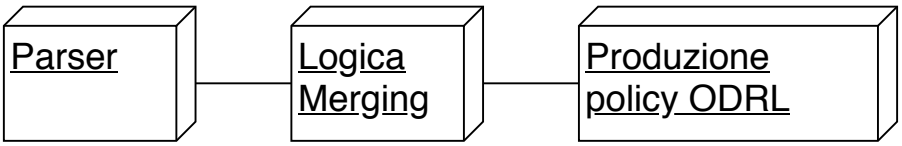

Figura 6.1: Pipeline dei macro componenti della soluzione

L'architettura generale è una semplice pipeline di componenti: il processo implementato risulta essere una serie di estrazioni di dati dai documenti ODRL in ingresso. L'output ottenuto ha le seguenti proprietà:

- è il risultato dell'unione o dell'intersezione delle policy in ingresso;
- esprime le regole in un formato non ambiguo per l'interrogazione puntuale.

Queste processo viene attuato all'interno del componente denominato *Logica Merging*, il quale si occupa di aggiungere la semantica necessaria al modello ODRL.
Il primo componente della pipeline si occupa di estrarre le informazioni di rilievo dalle policy in ingresso mentre il componente in coda ha il ruolo di produttore di documenti conformi ad ODRL.

6.2 Parser

Il primo componente della pipeline è il parser di documenti ODRL. Il suo ruolo è estrarre le informazioni utili alla procedura di merging. Questo componente sfrutta il

framework Apache Jena esposto nella sezione 4.2.5. Le interfacce esposte da questa porzione della soluzione sono:

- 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.

La componentizzazione è stata fatta al fine di incapsulare al meglio queste due funzionalità, permettendone l'uso atomico qualora necessario. L'architettura del componente è mostrata nella seguente figura:

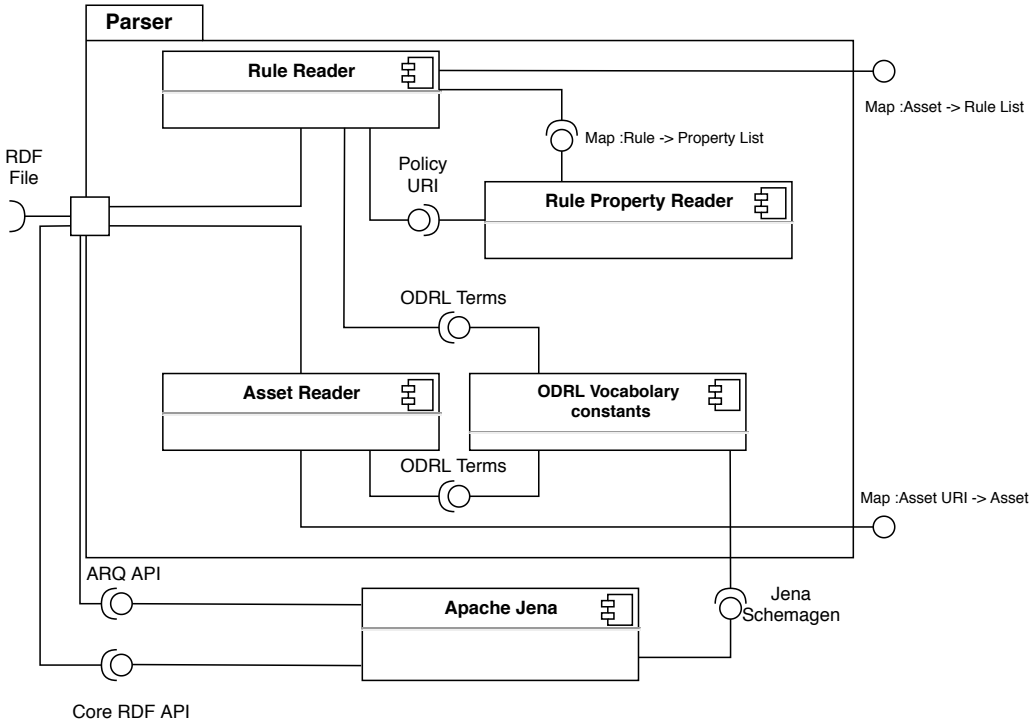

Figura 6.2: Diagramma delle componenti per il parsing di documenti ODRL

6.2.1 Rule Reader

Questo componente ha il ruolo di estrarre le regole presenti in una policy ed organizzarle in una mappa avente come chiavi gli *Asset* e come valori le liste delle regole che hanno la chiave come *target*. Il procedimento per la creazione della mappa 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 seguente query SPARQL:

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

Listing 6.1: Query per il recupero dei dati relativi a tutte le policy contenute nel documento ODRL

In questo modo è possibile identificare più policy all'interno del documento ODRL. L'URI delle varie policy è contenuto nel valore con chiave **policy** nelle tuple recuperate dalla query.
Il secondo passo della procedura è attuato mediante quest'ulteriore query SPARQL:

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

Listing 6.2: Query per il recupero delle proprietà di tutte le regole

Le tuple recuperate hanno la seguente struttura:

- **rule**: uri autogenerato da ARQ per le regole, in quanto prive di URI nelle policy;
- **pred**: proprietà all'interno della regola, per esempio *odrl:action* 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 valori *odrl:permission* o *odrl:prohibition*.

Il risultato viene riorganizzato in una mappa che lega ogni regola ad una coppia *proprietà-valore*.
Gli ultimi passi della procedura sono eseguiti analizzando la mappa ottenuta tramite il seguente algoritmo:

```
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
          rule = new Prohibition(action)
    //Assegnazione wrapper a chiave corretta
    if(mapAssetRuleList.hasKey(collection){
          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 6.3: Pseudo codice per il popolamento della mappa finale

Il codice mostrato nel listing 6.5 risulta avere complessità lineare proporzionale al numero di proprietà recuperate dalle query SPARQL.

6.2.2 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 con le 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 due proprietà citate precedentemente;
2. creazione dei wrapper 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 6.4: 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 6.5: 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 *parentURI*: l'identificatore dell'asset per cui si sta iterando nel terzo passo della procedura.
Per ogni URI recuperato da questa query, si aggiorna il relativo wrapper aggiungendo alla lista dei parent il nodo dell'iterazione corrente.

6.3 Logica Merging

All'interno di questo componente sono implementate le funzionalità necessarie per l'effettivo merging delle policy in ingresso.
Le operazioni di merging che si punta a supportare sono di due tipi:

- **unione**: utilizzata all'iterno di uno scenario collaborativo; i permessi espressi da almeno una delle due policy rimangono validi nella policy output;

- **intersezione**: utilizzata in uno scenario non collaborativo; un permesso appare nella policy in output solo se espresso all'interno di entrambi gli ingressi.

6.3.1 Rappresentazione delle Azioni

Uno dei fondamenti dell'implementazione sono le **Action**, ovvero gli utilizzi dell'asset regolabili dalle policy ODRL. Per rappresentarle si è utilizzato un enumerativo con la struttura mostrata in figura 6.3:

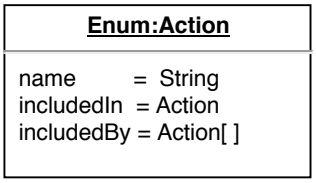

Figura 6.3: Struttura dell'enumerativo Action

All'interno dell'enumerativo è presente il nome che ha all'interno dell' ODRL Common Vocabulary, la lista delle azioni figlie e l'azione padre se esiste. Questa struttura è assimilabile a quella di un nodo in una struttura ad albero. L'enumerativo è provvisto dei metodi *get* per il recupero delle informazioni presentate.

6.3.2 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. Questo è reso possibile sfruttando le relazioni tra le azioni espresse mediante l'enumerativo definito in figura 6.3. Per ottenere un maggior incapsulamento delle funzioni si è deciso di dividere la logica della struttura ad albero in due compenenti:

- il gestore dell'albero delle regole: questa componente si occupa delle azioni ad alta astrazione sull'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 una regola;

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

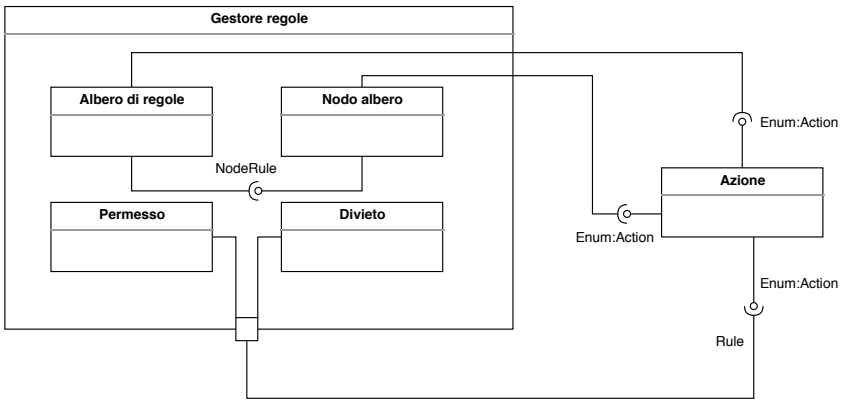

Figura 6.4: Diagramma dei componenti per la rappresentazione delle regole

Oltre ai componenti sopracitati in figura 6.4 è possibile vedere le implementazioni dell'interfaccia **Rule**, utilizzata anche per comunicare con il wrapper di policy.

Creazione dell'albero

L'albero viene creato mediante un gestore apposito specificandone l'azione radice. Come esposto nella sezione 6.3.1, l'enumerativo **Action** permette il recupero della lista delle azioni incluse dalla radice e quindi la creazione di eventuali sottoalberi. La creazione dei nodi figli è demandata al gestore del singolo nodo dell'albero.
Prendendo ad esempio l'azione **use** l'albero creato avrebbe la struttura seguente:

- 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;
- l'albero ha una struttura piatta, questo perché un'azione generica tende ad avere molti figli ma le azioni più specifiche tendono 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 tre:

1. **Permitted**: l'azione nel nodo è permessa esplicitamente;
2. **Prohibited**: l'azione nel nodo è proibita esplicitamente oppure tutte le sue azioni figlie sono nello stato **Prohibited**[1];
3. **Undefined**: l'azione nel nodo non ha regole che la interessano oppure 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** o **Undefined**, in questo caso il nodo rimane **Undefined**, mentre 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 stato vietato, altrimenti anche il padre è settato come **Prohibited**. La procedura attuata sul nodo padre è propagata 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:

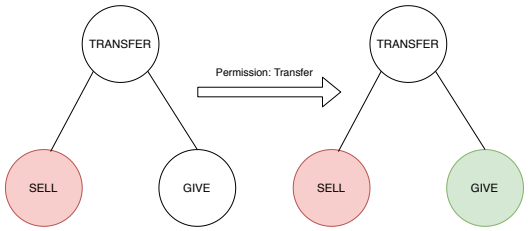

Figura 6.5: Esempio di propagazione di un permesso, con azione figlia proibita

[1]Questa procedura è attuata tenendo conto dell'assunzione che azioni generiche non siano attuabili. Risulta possibile adattare l'approccio ad altre ipotesi.

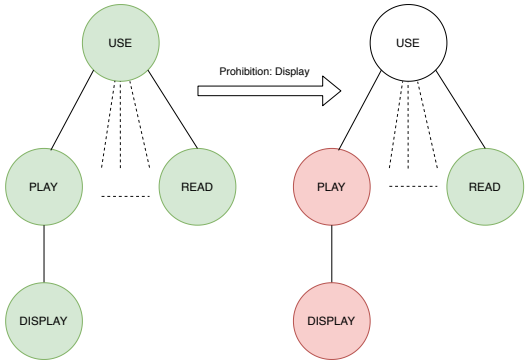

Figura 6.6: Esempio di propagazione verso il padre di un divieto

Recupero di un nodo e del suo stato

L'albero creato non è binario e, di conseguenza, non supporta una ricerca efficiente dei nodi. Grazie alla gerarchia delle **Action** si ha un'alternativa efficiente: sfruttando l'enumerativo in figura 6.3 è possibile ricostruire il percorso da un nodo foglia alla radice. Di seguito lo pseudo codice utilizzato per recuperare il nodo relativo ad un'azione:

```
1  public Node 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();
9  }
10
11 // Sempre vero a meno che non si chiami
12 // il metodo su un albero con radice errata
13 if(exploredNode.getAction() == steps.getLastStep()){
14   steps.popLast();
15   while(!(exploredNode.getAction() == a)){
16         exploredNode=
17            visitedNode.getChildsMap()
18              .get(steps.getLastStep);
19         steps.popLast();
20   }
21 else{
```

```
22    // Caso di albero con radice errata
23    return null;
24  }
25
26  return exploredNode;
27
28  }
```

Listing 6.6: 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 6.6 con un'ottimizzazione: se una delle azioni più in alto nella gerarchia si trova nello stato **Prohibited** o **Permitted** si ritorna quello stato e si interrompe la ricerca del nodo. Questa ottimizzazione è possibile poiché:

- se l'azione più in alto nella gerarchia è **Permitted**, necessariamente tutte le azioni nel suo sottoalbero sono **Permitted**; in caso contrario la radice del sottoalbero sarebbe **Undefined** o **Prohibited**;
- se l'azione più in alto nella gerarchia è **Prohibited**, ogni azione nel suo sottoalbero è **Prohibited**.

Le prestazioni nel caso peggiore rimangono comunque dipendenti dalla profondità della gerarchia delle azioni, mentre tra i casi migliori si aggiunge lo scenario in cui l'azione radice dell'albero è **Prohibited** o **Permitted**: questa casistica ha tempo d'esecuzione costante.

6.3.3 Rappresentazione Policy

Questa componente implementa le seguenti logiche:

- assegna agli asset target 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 policy è un'operazione creata per supportare uno scenario collaborativo. All'interno di questo ambiente i permessi definiti da un attore vengono condivisi anche dall'altro. L'operazione è attuata eseguendo i seguenti passi:

1. unione della lisa delle regole tra delle policy in input;
2. creazione di una nuova policy avente il set di regole ottenuto al passo precedente;
3. la policy finale creata sfrutta la logica già esposta nel paragrafo 6.3.2 per la gestione degli alberi delle regole.

Intersezione di policy

L'intersezione di policy è un'operazione creata per supportare uno scenario non collaborativo. In questo ambiente un attore non ha controllo sui permessi definiti da altre entità. In questo scenario 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 aventi come chiave l'azione regolata; questo passo è attuato su entrambi gli input;
2. si itera sulle chiavi delle mappe e si effettuano 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;
3. si crea una nuova policy avente il set delle regole ottenuto grazie ai passi precedenti;
4. la policy finale creata sfrutta la logica già esposta nel paragrafo 6.3.2 per la gestione degli alberi delle regole.

6.3.4 Rappresentazione Asset

Questo componente implementa funzionalità relative alla gestione degli asset, le quali prevedono:

- supporto alla gerarchia degli asset, dando modo di esplicitare la proprietà **partOf**;
- collega un asset alla policy con regole che lo interessano;
- propaga le regole di un asset alle collezioni figlie. Le possibili modalità di merging sono **intersect** ed **union**, esposte in sezione 6.3.3. Viene anche data la possibilità di non propragare la policy alle risorse figlie.

Per effetturare queste operazioni è stata usata un'architettura simile a quella mostrata nella sezione 6.3.2; nel caso precedente sono state separate le funzionalità a livello di gerarchia da quelle relative al singolo nodo. Di seguito il diagramma delle componenti di questa sezione:

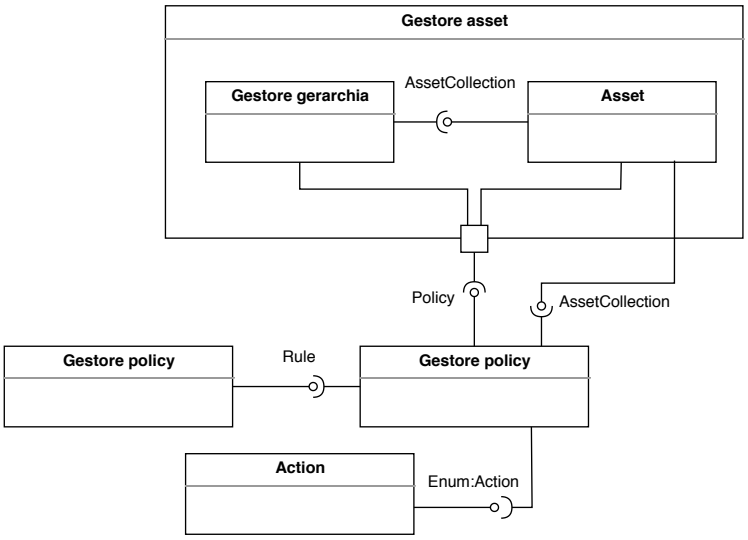

Figura 6.7: Diagramma dei componenti per la rappresentazione degli asset

Come è possibile notare dalla figura 6.7 questo componente sfrutta tutte le parti esposte in precedenza.
Lo scenario di utilizzo di un **Asset** si compone delle seguenti fasi:

1. definizione dei legami di gerarchia degli asset, tramite lettura della proprietà **partOf**;
2. definizione delle regole relative all'asset e creazione della relativa policy;
3. assegnazione della policy al gestore della gerarchia degli asset;
4. il gestore si occupa di settare la policy per il nodo interessato e propagarla ai figli secondo la modalità indicata.

Propagazione delle policy

La propagazione di una policy avviene dopo che questa è stata settata dal gestore della gerarchia degli asset. Il gestore si occupa quindi il nodo interessato dalla policy; per questa prima operazione si utilizza una ricerca in profondità data la natura ricorsiva dell'operazione di propagazione. Un'alternativa possibile è l'utilizzo di una ricerca in ampiezza, mantenendo il medesimo caso peggiore 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 alle foglie. Per nodi senza policy, viene settata quella del nodo padre.

Di seguito alcuni esempi:

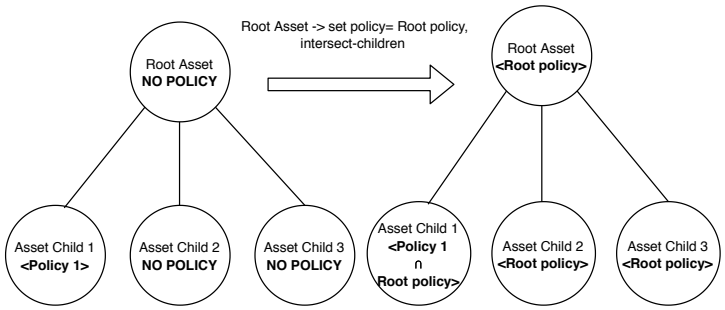

Figura 6.8: Esempio di propagazione di una policy quando l'asset padre non ha policy, modalità con intersezione

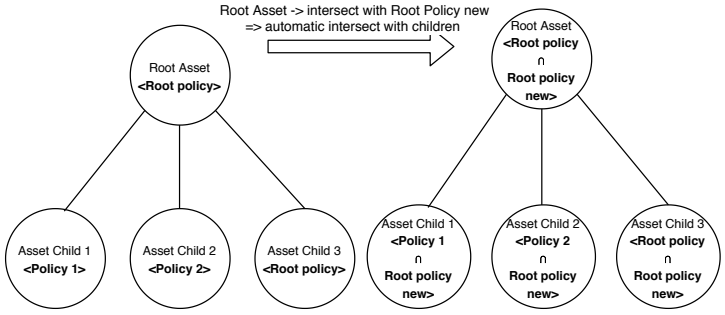

Figura 6.9: Esempio di propagazione di una intersezione di policy

Un processo analogo a quanto mostrato nelle figure 6.8 e 6.9 può sfruttare anche operazioni di unione. nella figura 6.9 è possibile notare che se un asset figlio non ha una policy eredita in modo completo quella del nodo padre.

Recupero policy in seguito alla propagazione
La policy relativa ad un nodo ai livelli più bassi può cambiare drasticamente in seguito ad una operazione di propagazione, come mostrato nelle figure 6.8 e 6.9. Il gestore della gerarchia degli asset implementa una procedura di recupero policy di uno specifico nodo: si percorre in ampiezza l'intera gerarchia, recuperando ogni policy. Questa procedura restituisce una mappa che collega l'URI di ogni risorsa nella gerarchia alla sua policy. nel set di chiavi sono presenti anche i nodi non aventi una policy.

6.3.5 Gestore procedura di merging

Tutte le componenti fino ad ora mostrate si concentrano in modo atomico su una singola porzione della procedura di merging, supportando operazioni di intersezione ed unione. Tutte le componenti lavorano però su un singolo documento ODRL. Il componente mostrato in questa sezione si occupa di supervisionare l'intera procedura. Di seguito il diagramma delle componenti che interessano il processo:

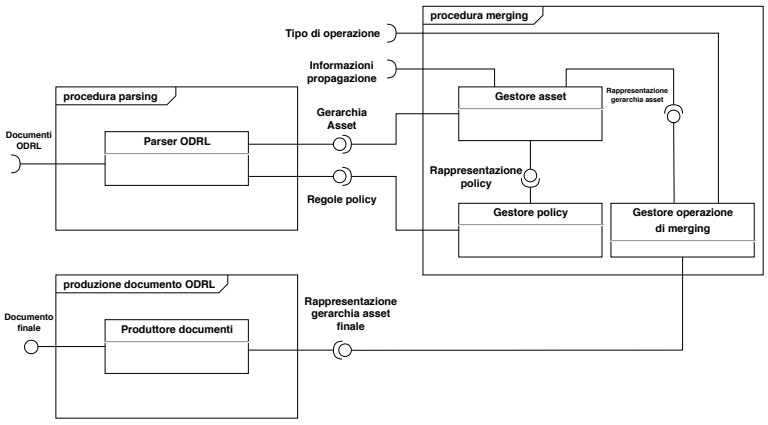

Figura 6.10: Componenti che interagiscono nel processo di merging di policy ODRL

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

- 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 tramite **union** ai livelli più bassi della gerarchia;
2. **non collaborativo**: la policy espressa in un asset ai livelli più alti della gerarchia si propaga tramite **intersection** ai livelli più bassi della gerarchia;
3. **nessun ambiente**: non avviene alcuna propagazione di policy.

- l'operazione che si vuole effettuare, ovvero **union** o **intersection**;
- estensione dell'ambiente dell'operazione alla gerarchia finale. L'effetto provocato da questa decisione è l'override dell'ambiente dato dall'operazione sull'intera gerarchia o solo sul sottografo interessato dal merging.

Casistiche supportate

Tutti i casi di merging supportati, attualmente non prevedono un **Assignee** per le regole: si suppone che tutte le regole si riferiscano allo stesso **Party** o **PartyCollection**. Questa semplificazione è stata attuata al fine di concentrarsi sul trattamento dei dati; la gestione utenti può essere attuata ripetendo il processo di merging per ogni attore.
Tenendo conto di questa prima semplificazione, il merging di policy può essere attuato nei seguenti casi di gerarchia degli asset:

1. le policy regolano la medesima gerarchia di asset;
2. le policy regolano gerarchie totalmente disgiunte;
3. una delle policy regola l'intera gerarchia, mentre l'altra ne gestisce solo una porzione;
4. una delle policy inserisce nuovi nodi figli;
5. una delle policy inserisce nuovi nodi 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 viene esposta la procedura utilizzata per creare la policy finale. L'implementazione è basata sulla seguente assunzione: la gerarchia 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 che a sua volta si divide in:
   (a) elaborazione di asset presenti in entrambi i documenti di policy;
   (b) elaborazione di asset presenti in uno solo dei due documenti di policy;
2. recupero delle regole relative ad ogni 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:

```
1  public Map<String, AssetCollection> mergeHierarchyCommon(
2           Map<String,Asset> assetsFirst,
3           Map<String,Asset> assetsSecond,
4           AssetTree treeFirst,
5           AssetTree treeSecond){
6
7    Set uriSetFirst = assets.keySet();
8    Set uriSetSecond = assetsSecond.keySet();
9    Map<String, AssetCollection> finalAssets;
10   commonURI = uriSetFirst.intersect(uriSetSecond);
11   finalAssets.put("EveryAsset", everyAssetNode);
12   for(String uri : commonURI){
13
14     if(uri.equals("EveryAsset"))
15       skip;
16     if(not(finalAssets.containsKey(uri)))
17      finalAssets.put(uri,new Asset(uri));
18
```

```
19      parentFirst = assetsFirst.get(uri).getParents();
20      parentSecond = assetsSecond.get(uri).getParents();
21
22      // Uno dei due non aveva un padre definito
23      if(
24      !parentFirst.equals(parentSecond) &&
25      (parentFirst.contains("EveryAsset") ||
26       parentSecond.contains("EveryAsset"))){
27         actParents =
28          parentFirst.contains("EveryAsset") ?
29          assetsFirst.get(uri).getParents() :
30          assetsSecond.get(uri).getParents();
31       for(actParent in actParents){
32         if(finalAssets.containsKey(actParent.getURI())){
33          finalAssets.get(actParent.getURI())
34           .addChild(finalAssets.get(uri));
35         }else{
36          actParent.addChild(finalAssets.get(uri));
37          finalAssets.put(actParent.getURI(),actParent);
38         }
39       }
40      }
41
42      //L'asset ha gli stessi parents in entrambe le policy
43      if(parentFirst.equals(parentSecond)){
44       for(parent in parentFirst){
45        if(finalAssets.containsKey(parent.getURI())){
46         finalAssets.get(parent.getURI())
47          .addChild(finalAssets.get(uri));
48        }else {
49         AssetCollection actParent = new
              Asset(parent.getURI());
50         actParent.addChild(finalAssets.get(uri));
51         finalAssets.put(actParent.getURI(),actParent);
52        }
53       }
54      }
55
```

```
56      // Almeno un parent diverso
57      if(!parentFirst.equals(parentSecond) &&
58         !parentFirst.contains("EveryAsset") &&
59         !parentSecond.contains("EveryAsset")){
60        Set totalParents = new Set(parentFirst);
61        totalParents.addAll(parentSecond);
62        for (String parentURI : totalParents){
63         if (finalAssets.containsKey(parentURI)) {
64          finalAssets.get(parentURI)
65            .addChild(finalAssets.get(uri));
66         } else {
67          AssetCollection actParent = new Asset(parentURI);
68          actParent.addChild(finalAssets.get(uri));
69          finalAssets.put(actParent.getURI(), actParent);
70         }
71      }
72    }
73
74
75    return finalAssets;
76  }
```

Listing 6.7: L'algoritmo ha prestazioni lineari in proporzione al più più grande degli insiemi di asset

Nel listing 6.7 è possibile notare come all'interno della gerarchia è inserito l'asset definito come **EveryAsset**. Questa è risorsa fittizia alla quale non viene mai assegnata una policy. Il suo ruolo è fungere da nodo sorgente per la ricerca degli asset poiché viene settata come nodo padre di ogni risorsa senza padre.

Il passo sucessivo è l'elaborazione degli asset non in comune alle policy:

```
1  public Map<String, AssetCollection> mergeHierarchyUniq(
2         Map<String,Asset> assets,
3         AssetTree tree,
4         Map<String, AssetCollection> finalAssets){
5
6   //In questo momento finalAsset contiene asset comuni
7   Set commonURI = finalAssets.keySet();
8   for(String uri: assets.keySet()){
9     if(!commonURI.contains(uri)){
10      if(!finalAssets.containsKey(uri))
11       finalAssets.put(uri,new Asset(uri));
12
13          // Aggiunta parent similare ai comuni
14    for(parent in assets.get(uri).getParents())
15      if(finalAssets.containsKey(parent.getURI())){
16        finalAssets.get(parent.getURI())
17        .addChild(finalAssets.get(uri));
18      }else {
19        AssetCollection actParent =
20         new Asset(parent.getURI());
21        actParent.addChild(finalAssets.get(uri));
22        finalAssets.put(actParent.getURI(),actParent);
23      }
24    }
25    return finalAssets;
```

Listing 6.8: L'algoritmo ha prestazioni lineari in proporzione al numero di asset

L'algoritmo esposto nel listing 6.8 viene utilizzato sugli insiemi di asset di entrambe le policy.
Applicando gli algoritmi descritti finora si ottiene l'output **finalAssets**: una mappa che collega ogni URI ad un wrapper dell'asset. Questo oggetto contiene le informazioni riguardanti nodi parent e nodi figli. Sfruttando le componenti descritte nella sezione 6.3.4 è quindi possibile definire un **gestore della gerarchia** degli asset con nodo sorgente **EveryAsset**. Il gestore si occupa di propagare le policy lungo la gerarchia seguendo la procedura relativa all'ambiente indicato.

Recupero policy singole

Come visibile all'interno del listing 6.7 il componente di parsing recupera le informazioni necessarie ad un **gestore della gerarchia** degli asset. Grazie a questi dati il componente crea una gerarchia di asset e propaga le varie regole lungo il grafo delle dipendenze. Il processo è eseguito seguendo le indicazioni relative all'ambiente indicate dall'utente, come visibile in figura 6.10.
Entrambe le policy originarie hanno un loro **gestore della gerarchia** che si occupa di recuperare la policy di ogni nodo. Questa procedura è stata già presentata nella sezione 6.3.4.
L'output di questo passo sono due mappe che legano l'URI di una risorsa alla regole presenti nel documento in cui l'asset con quell'identificativo si presenta.

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 merging desiderata, seguendo il corrispondente procedimento presentanto in sezione 6.3.3 per poi propagare ogni policy lungo la gerarchia finale, come presentato nella sezione 6.3.4.
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 policy iniziali, la policy risultante attua una propagazione coerente con l'operazione di merging;
- la gerarchia finale potrebbe subire cambiamenti anche in porzioni del DAG non interessate da entrambe le policy. Questo effetto può risultare indesiderato.

Nella seguente immagine vi è un esempio di effetto indesiderato:

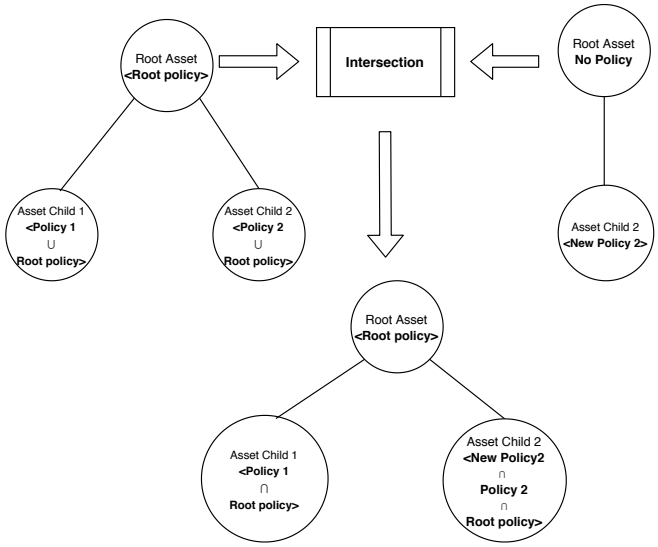

Figura 6.11: Intersezione di policy precedentemente in ambiente collaborativo

Sull'*Asset Child 2* si ha l'effetto desiderato: la risorsa non è più considerata in ambiente collaborativo ed ha come regole quelle ottenute dall'intersezione delle policy che la riguardano. Per quanto concerne la policy finale di *Asset Child 1* si ha che nonostante l'unica policy introdotta sia *New Policy 2* la propagazione dell'ambiente non collaborativo produce effetti anche su *Policy 1*.
La problematica appena presentata è stata trattata aggiungendo i seguenti passi all'algoritmo mostrato fino ad ora:

1. recupero degli asset che hanno subito una modifica di policy, ovvero le risorse che hanno in entrambe le policy almeno una regola che le riguarda. Questi nodi sono un sottoinsieme dei nodi comuni alle policy.

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

3. reset della policy originaria per i nodi recuperati nei passi precedenti, senza attuare propagazioni.

Il sottoagrafo 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 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:

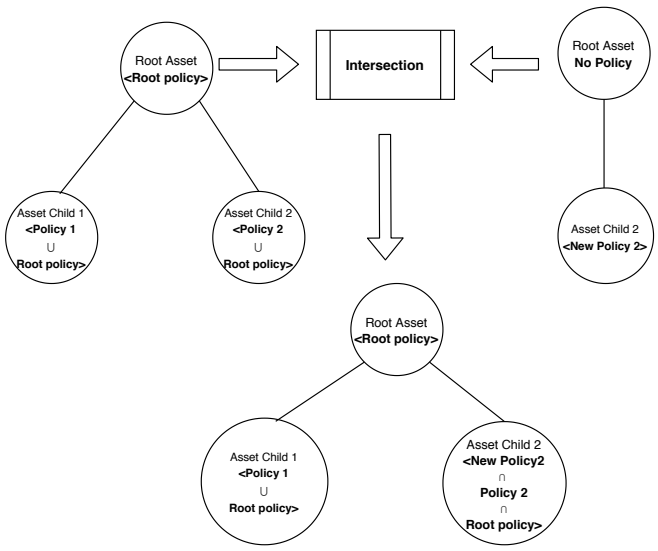

Figura 6.12: Intersezione di policy precedentemente in ambiente collaborativo senza override di ambiente

6.4 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. tramite il gestore gerarchia asset della 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 il namespace relativo al contesto ODRL;
5. il modello viene utilizzato dal modulo I/O che ne produce 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 7: Valutazione dei risultati

I principali obiettivi che questo lavoro di tesi si è posto sono stati esposti nella sezione 3.1.3 e riguardano due aree principali:

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

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 e risolve entrambe queste problematiche nel caso di azioni prive di **constraint**. La soluzione proposta non è applicabile solamente ai casi d'uso di MOSAICrOWN, come quelli relativi a mercati digitali dove gli asset possono avere più possessori, ma anche a scenari più tradizionali, come 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 le operazioni di merging **unione** ed **intersezione**. Tali operazioni identificano i 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 la procedura vista per policy singola.

Entrambi i procedimenti appena esposti vengono attuati dall'architettura osservabile in figura 6.10 i cui componenti sono stati esposti in dettaglio all'interno del capitolo 6 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, senza l'ausilio degli altri. Questo permette sia lo sfruttamento delle singole funzionalità di ogni porzione del lavoro sia la possibilità di estensione o sostituzione di un componente.

7.1 Esempi di casi d'uso

All'interno di questa sezione vengono mostrati casi d'uso dalla complessità crescente supportati dal lavoro di tesi. I risultati ottenuti effettuando query sull'output del policy merging vengono confrontati con ciò che è possibile effettuare solamente sui documenti ODRL originali.

7.1.1 Merging di policy relative ad un singolo asset

Casistica più semplice in assoluto. Risulta 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 7.1: 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 7.2: 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 7.3: Risultato dell'intersezione delle policy nei listing 7.1 e 7.2

```
@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 7.4: Risultato dell'unione delle policy nei listing 7.1 e 7.2

Il risultato è riassunto dalla seguente immagine:

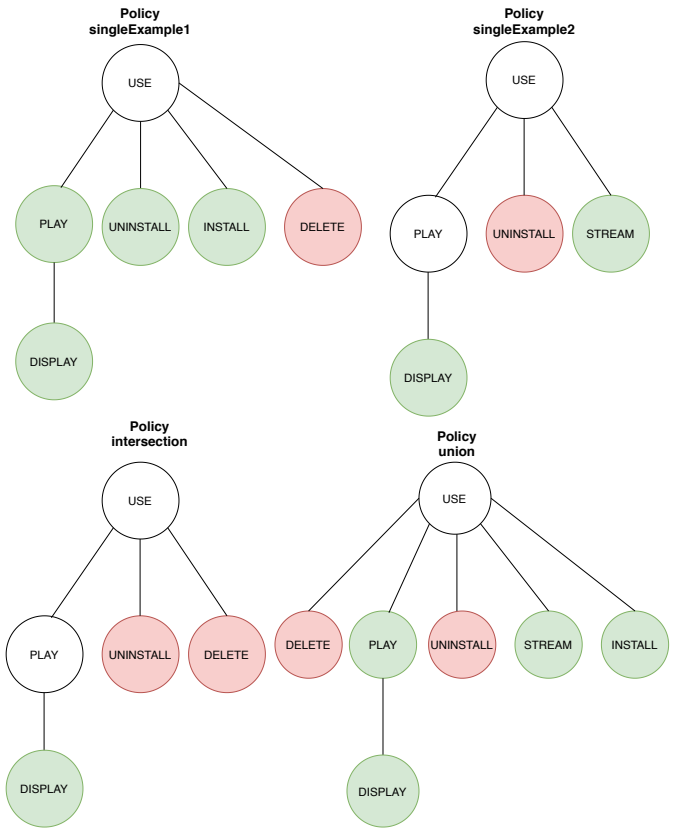

Figura 7.1: Albero delle regole delle varie policy all'interno dell'esempio

Vantaggi osservabili utilizzando i file mostrati nei listing 7.4 e 7.3:

- 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 da considerarsi un costo pagato una singola volta, a discapito invece dell'approccio baseline che deve ogni volta ricercare le informazioni relative alle *action*.

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, ottenendo una procedura più efficiente nel lungo termine.

7.1.2 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 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 7.1: Regolamentazione delle policy prima dell'applicazione di ambiente non collaborativo e intersezione

La struttura della gerarchia dello scenario è visibile in figura 7.2. 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 7.2: Regole all'interno della policy prodotta in output

Dalla tabella 7.2 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 ambienti collaborativi in questa situazione produce come risultato la propagazione della policy della radice a tutti i nodi. Questo è causato dall'operazione di unione preliminare poi seguita da quella di intersezione;

- i casi relativi all'unione di policy 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 MOSAICrOWN: 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;

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

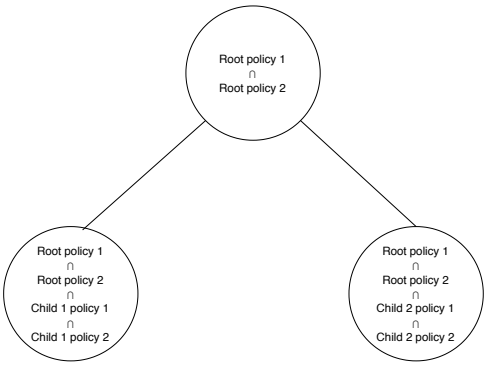

Figura 7.2: Policy finali del caso d'uso

7.1.3 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:

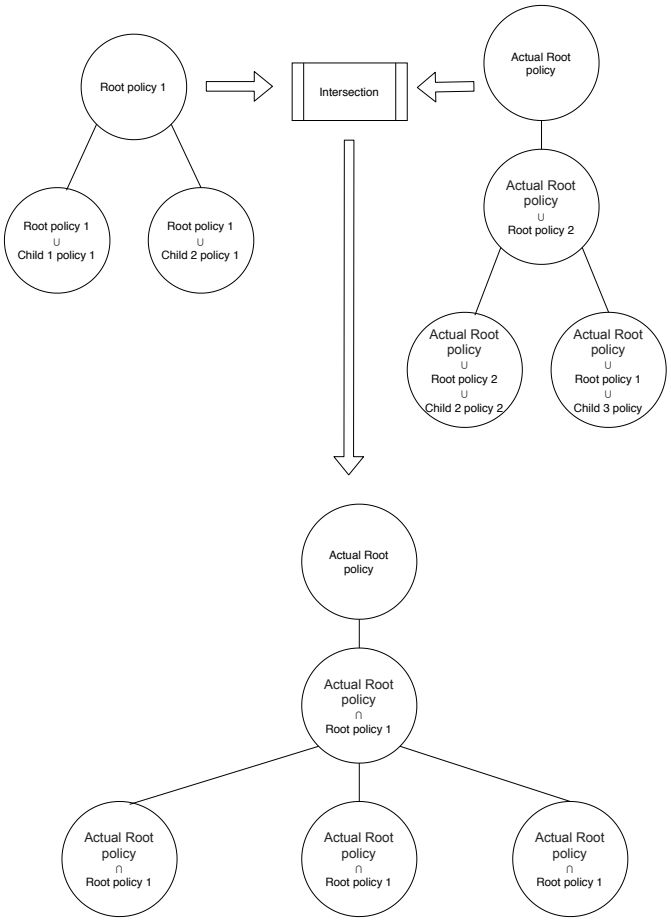

Figura 7.3: Procedura seguita per l'intersezione di 2 policy in ambiente collaborativo con l'aggiunta di nuovi nodi padre e foglia

Questa casistica assume 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: nel caso precedente questo portava alla replica dell'intersezione dei nodi negli altri nodi; in questo caso le differenze nella gerarchia fanno in modo che cambi il punto dal quale inizia la replica. Si noti che le operazioni a cui fa riferimento la figura 7.3 sono relative ai permessi delle policy, che vengono "mascherati" dalla sequenza unione-intersezione. Eventuali i divieti delle policy non visibili in figura 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.

7.1.4 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 rappresentata da un DAG anziché da un albero. L'esempio è caratterizzato dalle seguenti proprietà:

- ambienti iniziali collaborativi;
- intersezione di policy;
- aggiunta di 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 6.3.5.

La struttura precedente al merging è visibile nella figura sottostante:

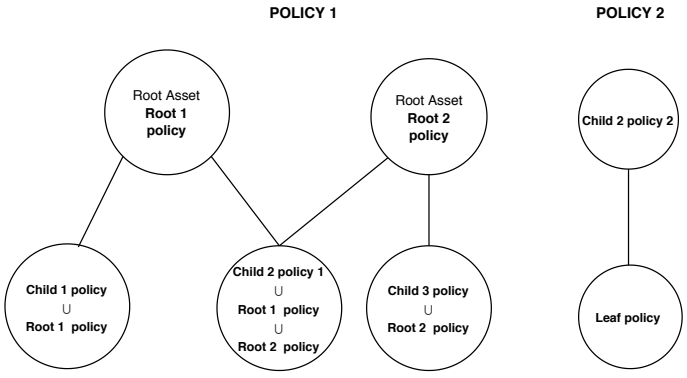

Figura 7.4: Struttura dei DAG regolati dalle policy

All'interno della tabella 7.3 sono presenti i permessi utilizzati come esempio. Le regole sono da intendersi come input allo strumento e quindi precedenti all'applicazione dell'ambiente collaborativo. Le tabelle 7.4 e 7.5 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 7.3: 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 7.4: Permessi in seguito all'intersezione con propagazione dell'ambiente sull'intero DAG

Come risulta possibile notare dalla tabella 7.4 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 7.5 mostra invece quello che è l'effetto desiderato nella maggior parte dei casi applicativi:

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

Tabella 7.5: Permessi in seguito all'intersezione con propagazione solo su asset interessati dal merging

- 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 7.4 e 7.5 sono riassunti nella figura 7.5.

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 MOSAICrO-WN, la cui caratterizzazione sarebbe la seguente:

- ambienti iniziali non collaborativi, avendo collezioni multi owner;
- intersezione di policy, avendo collezioni multi owner;
- aggiunta di un parent rappresentante la collezione di cui si fa parte nel mercato digitale;
- aggiunta o modifica di una foglia rappresentante la collezione dati dell'owner che vuole aggiungersi al mercato;
- risulta 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 MOSAICrOWN risulta interessante solamente l'utilizzo dell'algoritmo esteso: cambiamenti portati da un owner non devo avere effetti su dati sui quali non può definire regole.

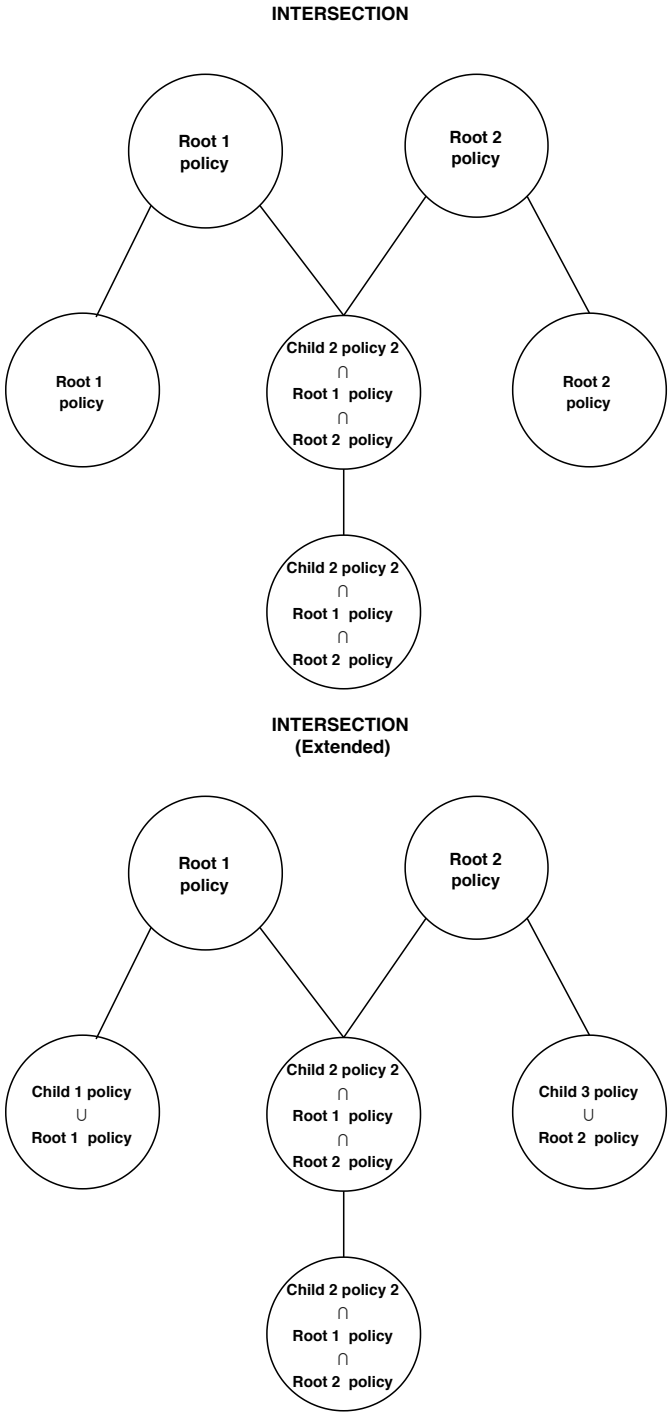

Figura 7.5

7.2 Possibili sviluppi futuri

In questa sezione si discutono quelli che possono essere gli sviluppi futuri del lavoro svolto. In particolare si trattano i seguenti aspetti:

- effettivo sistema per il deployment del tool;
- supporto alle gestione della cronologia delle operazioni;
- supporto ad azioni con **constraint**.

I primi due temi non sono stati approfonditi durante la stesura della tesi poiché non direttamente inerenti al merging di policy. L'ultimo tema, invece, non è stato trattato poiché la generalizzazione dei vincoli relativi ad un'azione dipende dai vari domini possibili.

Possibile deployment del tool
Per sfruttare completamente i vantaggi portati dalla procedura sviluppata risulta necessario effettuare il merging di policy solo all'inserimento di un documento ODRL all'interno del mercato digitale. Tra le possibili serializzazioni RDF vi è il formato JSON-LD. Questo formato può essere sfruttato all'interno di basi di dati NoSQL come ad esempio MongoDB.
Per quanto concerne l'effettivo inserimento di un documento ODRL all'interno della base di dati del mercato digitale si potrebbe esporre un REST Service dove il corpo delle richieste è un documento JSON avente i seguenti campi:

- operazione di merging che si desidera effettuare;
- oggetto JSON contenente la serializzazione JSON-LD di una policy;
- identificativo della collezione al quale si vuole prendere parte.

In questo modo il sistema avrebbe tutte le informazioni necessarie per attuare il merging tra la policy attuale delle collezione esposta nel mercato digitale e quella inviata al servizio REST.

Possibile gestione della cronologia delle operazioni
All'interno del mercato digitale risulta possibile che i vari owner non vogliano solamente aggiungere dati ad una collezione ma modificarne la policy o rimuovere i propri dati dal mercato. Per supportare questo tipo di operazioni è necessaria la rappresentazione della cronologia delle operazioni che hanno portato una policy al suo stato

attuale.
La rappresentazione che viene proposta è riassunta nella seguente immagine:

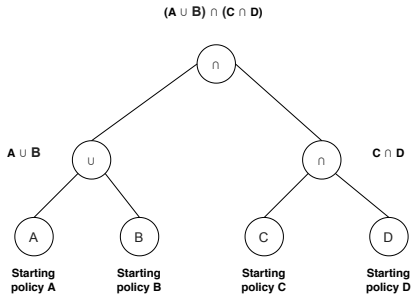

Figura 7.6: Esempio di rappresentazione della cronologia di operazioni che portano allo stato finale di una policy

Questa rappresentazione può essere attuata seguendo due metodi:

- arricchendo la sintassi ODRL inserendo termini relativi al processo di merging. In questo modo si seguono i princici dei *Linked Data* poiché da una policy è possibile raggiungere i vari documenti che hanno contribuito alla sua creazione;
- inserendo in una base di dati record relativi alle operazioni intraprese su una policy. Questo approccio risulta attuabile anche in database relazionali ma è meno vicino ai principi di sviluppo di ODRL.

In ognuno dei due casi se una delle policy originali venisse rimossa o modificata l'unica operazione necessaria per aggiornare i documenti finali sarebbe ripercorrere l'albero delle operazioni. Prendendo ad esempio la figura 7.6: se si volesse rimuovere la policy B si seguirebbe il procedimento riassunto nella seguente figura:

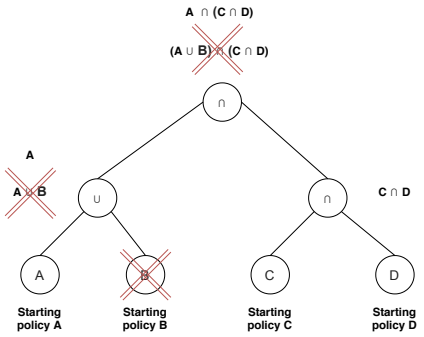

Figura 7.7: Rappresentazione della procedura attuata in seguito alla rimozione della policy B

Trattazione regole con constraint

Come mostrato all'interno della sezione 3.1.2, ODRL permette la definizione di **refinement** sulle azioni regolate da una policy. Questo aspetto non è stato trattato all'interno del lavoro di tesi per le seguenti motivazioni:

- i dati contenuti in queste proprietà possono essere poco strutturati. Si prenda come esempio il listing 3.7 dove la proprietà *unit* è definibile attraverso un link che spiega quale sia l'unità di misura utilizzata. Questo comporta la definizione di procedure specifiche rispetto al dominio e quindi poco generalizzabili;
- come visibile all'interno dell'*ODRL Information Model*[9] la struttura relativa ai *Constraint* è soggetta a cambiamenti; per esempio risultano tra i pochi punti di non compatibilità con le vecchie versioni di ODRL. Per ottenere uno strumento il più stabile possibile si è preferito non lavorare su questa porzione del linguaggio che risulta in fase di definizione.

Alla luce di queste problematiche per riuscire ad utilizzare questi costrutti all'interno di un mercato digitale risulta necessario discretizzare il *range* utilizzabile in queste proprietà. Questo procedimento andrebbe a ridurre la flessibilità del linguaggio ma, allo stesso tempo, permetterebbe di attuare logiche di merging anche su questa porzione di ODRL.

Capitolo 8: Conclusione

La soluzione implementata risolve le problematiche individuate in ODRL in molteplici scenari, compresi quelli individuati in MOSAICrOWN. Tali problemi riguardavano l'assenza di politiche per risolvere conflitti tra policy in un contesto multi-owner e l'inefficienza del linguaggio in fase di interrogazione della policy. Queste problematiche sono state affrontate mediante lo sviluppo di una procedura di *merging* di due policy. Questo processo produce in output una nuova policy che soddisfa i requisiti espressi da entrambi i documenti in input. Il *merging* può essere attuato secondo due modalità: *intersezione* ed *unione*. Questa decisione permette di supportare sia scenari collaborativi che non collaborativi. Lo strumento prodotto non si occupa solamente del *merging* di policy relative a singole risorse ma anche della propagazione dell'operazione sull'intera gerarchia degli asset regolati dalle policy.
Il processo di *merging*, essendo uno sforzo eseguito una tantum, produce in output un documento riutilizzabile che permette un'iterrogazione del modello efficiente in tutti gli scenari d'utilizzo possibili. L'unico lato negativo del processo risulta essere il costo di storage di questa nuova policy prodotta. I vantaggi dal punto di vista computazionale sono stati dimostrati mediante un confronto con un metodo baseline in grado di utilizzare solamente le policy originarie per controllare lo stato di una regola.
La soluzione prodotta non introduce modifiche ad ODRL e risulta in linea con il requisito di MOSAICrOWN di standardizzazione ed applicabilità dei risultati ottenuti. Questo è possibile anche grazie all'implementazione delle componenti per il *parsing* e la *produzione* mediante il framework *Apache Jena*. Il documento prodotto in output risulta infatti essere conforme allo standard ODRL in ogni sua parte.

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 Vocabulary & Expression 2.2.

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

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

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

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

[13] European Union's Horizon 2020 research and innovation programme. Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNer control.

[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.