Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

ESTENSIONE DEL LINGUAGGIO FACPL PER ESPRIMERE POLITICHE DI GESTIONE DELL'UTILIZZO CONTINUATIVO DELLE RISORSE DI UN SISTEMA DI CALCOLO

EXTENSION OF LANGUAGE FACPL TO USE ACCESS CONTROL POLICIES BASED ON CONTINUATIVE USE OF RESOUCES

FILIPPO MAMELI

Relatore: *Rosario Pugliese*
Correlatore: *Andrea Margheri*

Anno Accademico 2015-2016

INDICE

1

INTRODUZIONE

Dalla loro nascita i sistemi informatici hanno avuto il ruolo di gestore di dati. Il tipo di queste informazioni ha reso necessario l' utilizzo di un sistema che le proteggesse. I dati più sensibili se diffusi senza una valida autorizzazione possono arrecare danni economici ad una società o anche nuocere gli utenti nel privato. Lo sviluppo del web ha generato un interconnessione ancora più forte tra i sistemi e questo ha messo ancora più a rischio le informazioni più critiche.

1.1 ESTENSIONE DEL LINGUAGGIO

2

ACCESS CONTROL E USAGE CONTROL

2.1 CONTROLLO DEGLI ACCESSI

La protezione dei dati ha determinato la necessità di creare strumenti per il controllo degli accessi che potevano eliminare ,o almeno limitare, i rischi derivati dalla perdita delle informazioni.

Nel corso del tempo si sono sviluppati alcuni modelli per i sistemi del controllo degli accessi. A seconda delle esigenze sono stati adottati numerosi tipi di tecnologie[1]. Nelle sezioni successive se ne presentano alcune.

2.1.1 *Access Control List*

Access Control List(ACL) è stato creato agli inizi degli anni settanta per la necessità di un controllo degli accessi sui sistemi multiutente. Utilizza una lista di utenti con annesse le possibili azioni autorizzate. Il modello è molto semplice, ma ha molte limitazioni. Quando nel sistema ci sono numerosi utenti o risorse, la quantità di dati da verificare diventa difficile da gestire. Questo può portare a errori di assegnazione di autorizzazioni e ad un eccesivo numero di controlli necessari per un singolo accesso.

2.1.2 *Role Based Access Control*

Role Based Access Control (RBAC) è l' evoluzione di ACL. In questo modello vengono introdotti i *ruoli*. Più utenti possono avere lo stesso ruolo e quindi avere a disposizione le tutte risorse connesse a questo. Il modello diventa scalabile e più facile da gestire, inoltre si possono anche creare delle gerarchie per facilitare l' assegnamento di risorse in base alla classificazione dell' utente.

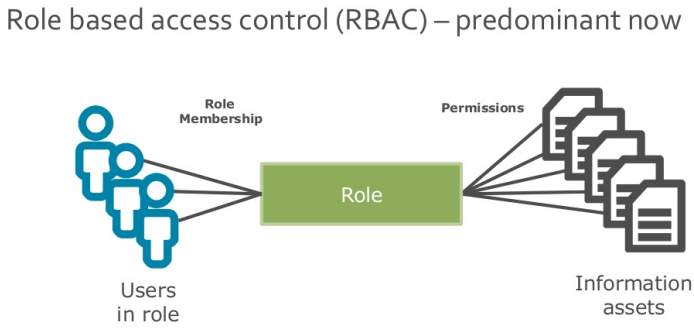

Figura 1: RBAC

2.1.3 *Attribute Based Access Control*

Attribute Based Access Control (ABAC) si basa sull'utilizzo di attributi associati all'utente, all'azione o al contesto della richiesta. La valutazione di una autorizzazione diventa più specifica e le regole sono più precise per ogni risorsa. Questo tipo di modello non è utilizzato nei sistemi operativi,dove ACL e RBAC sono i modelli più diffusi, ma è sviluppatto spesso a livello applicativo. Il problema fondamentale di questo paradigma è che le regole non sono uniformi e se il numero di risorse è consistente, la gestione di queste diventa complicata. Il modello Policy Based Access Control cerca di risolvere i difetti di ABAC.

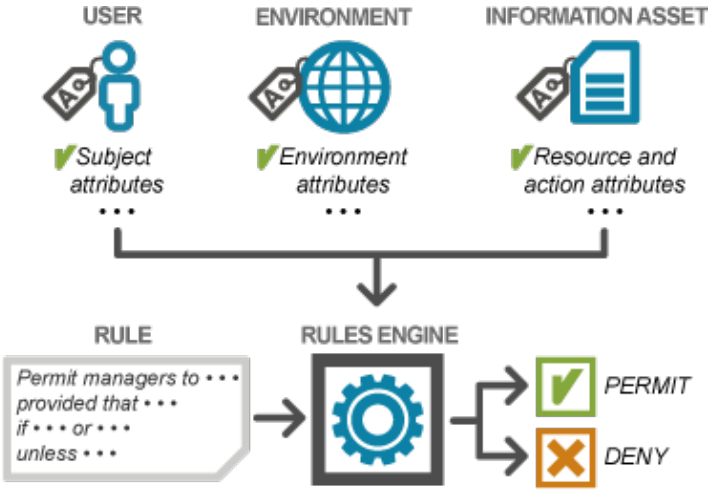

Figura 2: ABAC

2.1.4 *Policy Based Access Control*

Policy Based Access Control riorganizza il modello ABAC per semplificare la gestione delle regole. Il sistema si basa su *politiche* che non sono altro che insiemi di *regole*. A ogni regola è associato un attributo che l'utente deve avere, e ogni politica valuta tutte le regole nel suo insieme per creare la risposta sull'autorizzazione. Anche le politiche possono essere messe insieme per creare gruppi di politiche, in questo modo il sistema diventa scalabile e di più facile utilizzo.

Per costruire un sistema di controllo degli accessi basato sul modello PBAC è necessario l'utilizzo di un linguaggio adatto allo scopo. L'organizzazione OASIS (Organization for the Advancement of Structured Information Standards) ha creato il linguaggio eXtensible Access Control Markup Language (XACML) che è diventato lo standard per lo sviluppo di un sistema costruito sul modello PBAC.

2.2 USAGE CONTROL

Dopo quaranta anni di studi sul controllo degli accessi i modelli sviluppati si sono consolidati e sono largamente utilizzati su sistemi operativi o applicazioni. Tuttavia la complessità e la varietà degli ambienti informatici moderni va oltre i limiti dei modelli creati.

Il termine Usage Control (UCON) è stato ripreso da Jaehong Park e Ravi Sandhu per creare il modello UCON_{ABC}[2], questo è una generalizzazione dell'Access Control che include obbligazioni, condizioni sull'utilizzo, controlli continuativi e mutabilità. Comprende e migliora i modelli di controllo di accesso tradizionali, quali Trust Management (TM) e Digital Rights Management (DRM) aggiungendo la gestione di attributi variabili e la continuità nella valutazione delle decisioni per l'accesso. Il modello UCON_{ABC} estende i controlli sull'accesso tradizionali ed è composto da otto componenti fondamentali. Queste sono subjects,subject attributes, objects,objects attributes, rights, authorizations, obligations e conditions.

I *Subjects* sono entità a cui si associano degli attributi e hanno o esercitano *Rights* sugli *Objects*. Possiamo per semplicità associare i Subjects ad un singolo individuo umano.

Gli *Objects* sono insiemi di entità su cui i *Subjects* possono avere dei *Rights*, questi possono essere usati o vi si può fare accesso. Possono essere associati ad esempio a un libro, o a una qualsiasi risorsa.

I *Rights* sono i privilegi che i *Subjects* hanno o esercitano sugli *Objects*.

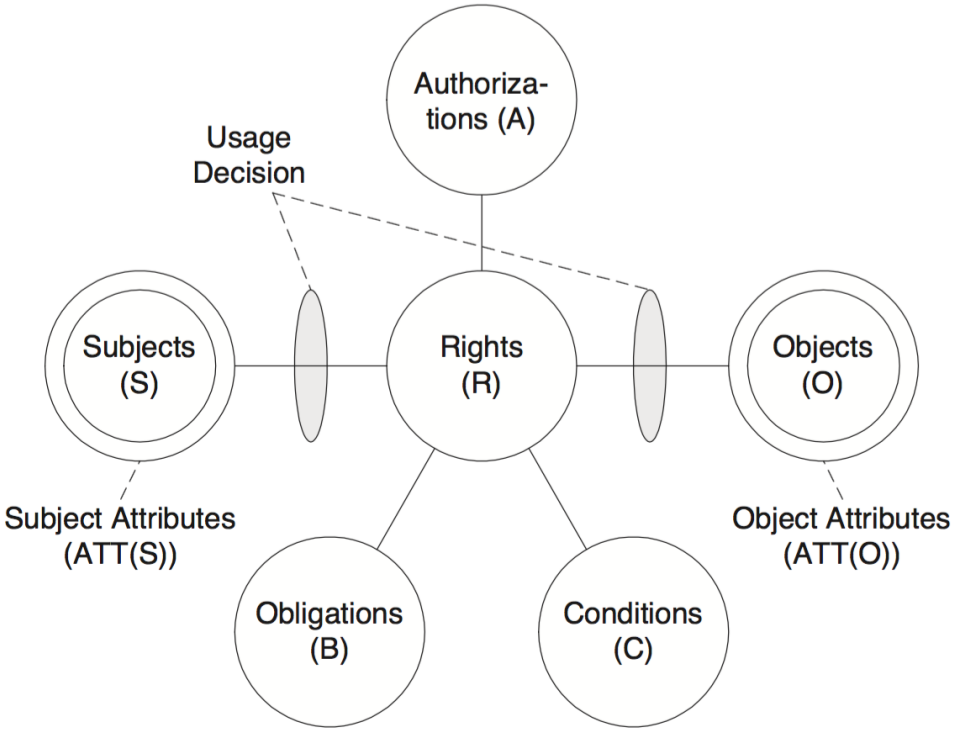

Figura 3: UCON

I tre fattori Authorizations, oBligations e Conditions(da cui prende anche il nome il modello $UCON_{ABC}$) sono predicati funzionali che devono essere valutati per le decisioni sull'uso. I tradizionali Access Controls utilizzano solo le Authorizations per il processo di decisione, Obligations e Conditions sono i nuovi componenti che entrano a far parte della valutazione.

Le *Authorizations* devono valutare la decisione sull'uso. Queste danno un responso positivo o negativo a seconda che la domanda di un Subject sia accettata o meno.

Le *Obligation* verificano i requisiti obbligatori che un Subject deve eseguire prima o durante l'utilizzo di una risorsa.

Infine le *Condition* restituiscono true o false in base alle variabili dell'ambiente o allo stato del sistema.

Il processo di decisione è diviso in tre fasi[3]:Before usage(pre), Ongoing usage(on) e After usage. La valutazione della prima parte inizia da una richiesta e non ha differenze con il processo valutativo dell'Access Control. Nella seconda invece si utilizzano i nuovi predicati introdotti ed

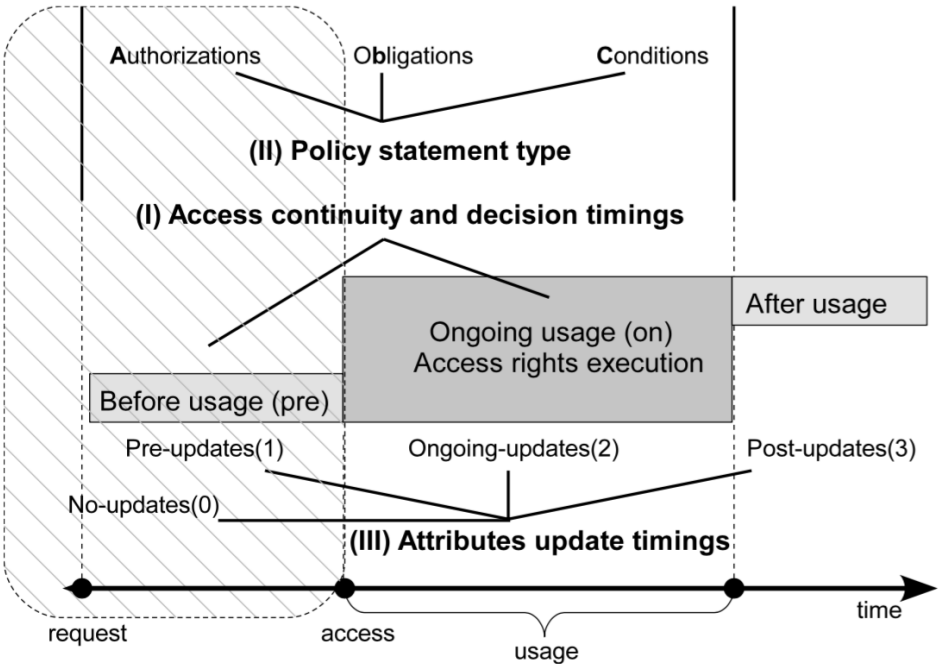

Figura 4: Fasi del processo di decisione

è in questa parte che si affermano i controlli continuativi, le obbligazioni e le condizioni sull'utilizzo.

L'ultima parte varia in base agli eventi delle fasi precedenti. Ad esempio se il Subject che ha richiesto un accesso ha violato una policy oltre al non aver ricevuto l'autorizzazione potrebbe anche essere ammonito e il sistema potrebbe non accettare più nessuna sua richiesta.

2.2.1 *Esempi di Usage Control*

Si propongono adesso due esempio in cui si utilizza lo Usage Control

Accessi in lettura e scrittura di file

In un sistema ci sono alcuni file e gli utenti sono divisi in gruppi, si suppone inoltre che le richieste di lettura siano più comuni rispetto a quelle di scrittura.

Tutti possono leggere i file, ma solo gli utenti nel gruppo degli amministratori possono modificare un elemento. Se uno degli amministratori sta scrivendo su un file, nessuno può leggerlo o apportare modifiche nello

stesso momento. Una volta finita la scrittua si potrà sbloccare il file e renderlo di nuovo disponibile a tutti.

Letture ripetute, su uno stesso insieme di file, vengono gestite in modo efficiente.

Servizio di streaming

In un sistema gli utenti, dopo aver fatto il login, possono fare una richiesta di ascolto di un brano. I clienti sono divisi in due tipologie. Chi effettua il login come utente premium può fare richieste di ascolto senza nessuna restrizione. I clienti stardard hanno invece un tempo limite e una volta esaurito devono ascoltare la pubblità prima di poter fare un'altra richiestad di ascolto

In questo caso si assumono le richieste di ascolto più numerose rispetto ai login o ai logout.

3

FORMAL ACCESS CONTROL POLICY LANGUAGE

Il linguaggio FACPL (fakpol) è stato creato come alternativa a XACML. Come accennato nel capitolo precedente, l'organizzazione OASIS ha ideato il linguaggio XACML per sviluppare sistemi basati sul modello PBAC.

XML è utilizzato da XACML per definire le sue politiche. Questo linguaggio di markup è molto usato per lo scambio di dati tra sistemi, ma rende le politiche difficili da comprendere. Infatti anche le regole più semplici sono prolisse e questo rende la lettura problematica per un utente.

FACPL nasce dalle idee di XACML e ha l'obiettivo di semplificare la scrittura di politiche e di definire un framework costruito sopra basi formali solide, in modo da permettere agli sviluppatori di specificare e verificare automaticamente delle proprietà.

3.1 IL PROCESSO DI VALUTAZIONE

In figura 5 si mostra il processo di valutazione delle policy e delle richieste. Le componenti chiave sono tre:

- Policy Repository (PR)
- Policy Decision Point (PDP)
- Policy Enforcement Point (PEP)

Si assume che un insieme di risorse sia in coppia con le Policy, e che queste definiscano le credenziali necessarie per ottenere l'accesso. Il PR contiene le Policy e le rende disponibili al PDP (Step 1), il quale decide se l'accesso viene garantito o meno.

Quando la richiesta è ricevuta dal PEP (Step 2), le sue credenziali vengono codificate in una sequenza di attributi(una coppia nome-valore) per poi utilizzare quest'ultima per creare la richiesta FACPL(Step 3).

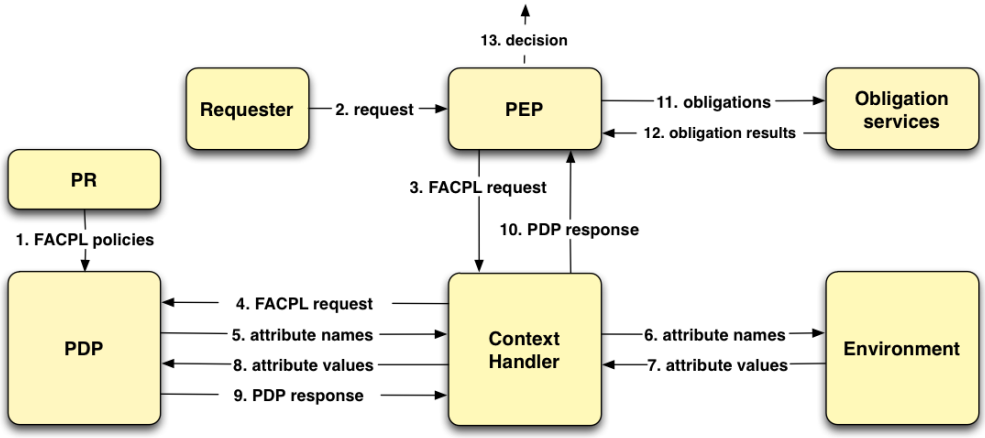

Figura 5: Processo di valutazione

Il Context Handler aggiunge attributi dell'ambiente (come la temperatura esterna) alla richiesta e poi invierà questa al PDP (Step 4).

Il processo di autorizzazione del PDP restituisce una *risposta* verificando che gli attributi, che possono far parte della richiesta o del contesto, siano conformi ai controlli delle policy (Step 5-8). La *risposta* del PDP contiene una *decisione* e una possibile *obbligazione*(Step 9-10).

La *decisione* può essere di quattro tipi:

- Permit
- Deny
- Not-applicable
- Indeterminate

I primi due tipi indicano rispettivamente richiesta accettata e richiesta non accettata. Se viene restituito Not-applicable non ci sono policy su cui si poteva valutare la decisione, se ci sono altri errori la risposta è Indeterminate. Le *Policy* gestiscono automaticamente il risultato Indeterminate combinandolo con le altre decisioni a seconda della strategia del *Combining Algorithm* associato alla Policy stessa.

Le *Obligations* sono azioni addizionali che devono essere eseguite dopo la restituzione di una decisione. Solitamente corrispondono all'aggiornamento di un file, l'invio di un messaggio o l'esecuzione di un comando. Il PEP ha compito di controllare lo svolgimento delle *Obligations* tramite l'*obligation service* (Step 11-12). Il processo di *enforcement* eseguito dal PEP

determina l'*enforced decision* in base al risultato delle obbligazioni. Questa decisione può differire da quella del PDP e corrisponde alla valutazione finale di tutto il processo.

3.2 COMPONENTI DEL SISTEMA

Nella tabella 1 si mostra la sintassi di FACPL. Questa è data da una grammatica di tipo EBNF dove il simbolo ? indica elementi opzionali, * sequenze (anche vuote) e + sequenze non vuote.

Al livello più alto troviamo il termine Policy Authorization System (PAS) che comprende le specifiche del PEP e del PDP.

Il PEP è definito con un *enforcing algorithm* che sarà applicato per stabilire quali sono le decisioni che devono passare al processo di *enforcement*.

Il PDP è definito come una sequenza di $Policy^+$ e da un algoritmo *Alg* per combinare i risultati delle valutazione delle policy.

Una *Policy* può essere una *Rule* semplice oppure un *Policy Set* cioè un insieme di rule o altri policy set. In questo modo si possono creare regole singole, ma anche gerarchie di regole.

Un *Policy Set* è definito da un *target* che indica l'insieme di richieste di accesso al quale la policy viene applicata, da una lista di *Obligations* cioè le azioni opzionali o obbligatoriche da eseguire, da una sequenza di *Policy* e da un algoritmo per la combinazione.

Una *Rule* è specificata da un *effect* , che può essere permit o deny , da un *target* e da una lista di *Obligations* (che può essere anche vuota).

Le *Expressions* sono costituite da *attribute names* e da valori letterali(per esempio booleani, stringhe, date).

Un *attribute name* indica il valore di un attributo. Questo può essere contenuto in una richiesta o nel contesto. La struttura di un attribute name è della forma *Identifier*/*Identifier*. Dove il primo elemento indica la categoria e il secondo il nome dell'attributo. Per esempio Name / ID rappresenta il valore di un attributo ID di categoria Name.

Un *combining algorithm* ha lo scopo di risolvere conflitti delle decisioni date dalle valutazioni delle policy. Ad esempio permit-overrides assegna la precedenza alle decisioni con effetto permit rispetto a quelle di tipo deny.

Una *Obligation* è definita da un effetto, da un tipo (M per obbligatorio e O per opzionale) e da un'azione e le relative espressioni come argomento.

Una richiesta consiste in una sequenza di *attribute* organizzati in categorie.

Tabella 1: Sintassi di FACPL

Policy Authorisation Systems	PAS ::=	(pep : $EnfAlg$ pdp : PDP)
Enforcement algorithms	$EnfAlg$ ::=	base | deny-biased | permit-biased
Policy Decision Points	PDP ::=	{Alg policies : $Policy^+$}
Combining algorithms	Alg ::=	p-over$_\delta$ | d-over$_\delta$ | d-unless-p$_\delta$ | p-unless-d$_\delta$ | first-app$_\delta$ | one-app$_\delta$ | weak-con$_\delta$ | strong-con$_\delta$
fulfilment strategies	δ ::=	greedy | all
Policies	$Policy$ ::=	($Effect$ target : Expr obl : $Obligation^*$) | {Alg target : Expr policies : $Policy^+$ obl : $Obligation^*$ }
Effects	$Effect$ ::=	permit | deny
Obligations	$Obligation$ ::=	[$Effect$ $ObType$ PepAction(Expr*)]
Obligation Types	$ObType$ ::=	M | O
Expressions	$Expr$ ::=	$Name$ | $Value$ | and($Expr$, $Expr$) | or($Expr$, $Expr$) | not($Expr$) | equal($Expr$, $Expr$) | in($Expr$, $Expr$) | greater-than($Expr$, $Expr$) | add($Expr$, $Expr$) | subtract($Expr$, $Expr$) | divide($Expr$, $Expr$) | multiply($Expr$, $Expr$)
Attribute Names	$Name$ ::=	$Identifier$/$Identifier$
Literal Values	$Value$ ::=	true | false | $Double$ | $String$ | $Date$
Requests	$Request$ ::=	($Name$, $Value$)$^+$

La risposta ad una richiesta FACPL è scritta utilizzando la sintassi ausiliaria riportata in tabella 2. La valutazione in due passi descritta in 4.1 produce due tipi differenti di risposte:

- *PDP Response*
- *Decisions*

La prima nel caso in cui la decisione sia permit o deny si associa a una sequenza (anche vuota) di fulfilled obligations.

Tabella 2: Sintassi ausiliaria per le risposte

PDP Responses	*PDPResponse* ::=	$\langle$ *Decision* *FObligation** $\rangle$
Decisions	*Decision* ::=	permit | deny | not-app | indet
Fulfilled obligations	*FObligation* ::=	[*ObType* PepAction(*Value**)]

Una *Fulfilled Obligation* è una coppia formata da un tipo e da un'azione con i rispettivi argomenti risultato della valutazione del PDP.

3.3 SEMANTICA

Si presenta adesso informalmente il processo di autorizzazione del PDP e poi quello di enforcement del PEP.

Quando il PDP riceve una richiesta di accesso, valuta la richiesta in base alle *policy* disponibili poi determina il risultato unendo le decisioni restituite dalle policy con l'utilizzo dei *combining algorithm*.

La valutazione di una policy inizia dalla verifica dell'applicabilità della richiesta, che è compiuta valutando l'espressione definita dal *target* . Ci sono due casi:

La verifica dà un risultato positivo. Nel caso in cui ci siano *rules*, l'effetto della regola viene restituito. Nel caso di *Policy Set* , il risultato è ottenuto dalla valutazione delle *policy* contenute e dalla combinazione di queste. tramite l'algoritmo specificato dal PDP. In entrambi i casi in seguito si procederà al *fulfilment* delle *obligation* da parte del PEP.

La verifica dà un risultato negativo. Nel caso in cui la valutazione sia determinata `false`, viene restituito `not-app`. Nel caso di *error* o di un valore non booleano, si restituisce `indet`.

Valutare le espressioni corrisponde ad applicare gli operatori, a determinare le occorrenze degli *attribute names* e a ricavarne il valore associato.

Se questo non è possibile, ad esempio l'attibuto è mancante e non può essere recuperato dal *context handler*, si restituisce il valore speciale `BOTTOM`. Questo valore è usato per implementare strategie diverse per gestire la mancanza di un attributo. `BOTTOM` è trattato da FACPL in modo simile a `false`, si gestisce così gli attributi mancanti senza generare errori.

Gli operatori tengono conto degli argomenti e dei valori speciali come `BOTTOM` e *error*. Se gli argomenti sono `true` oppure `false` gli operatori sono applicati in modo regolare. Se c'è un argomento `BOTTOM` e non ci sono *error* si restituisce `BOTTOM`, *error* altrimenti. Gli operatori *and* e *or* trattano diversamente i valori speciali. Specificatamente, *and* restituisce `true` se entrambi gli operandi sono `true`, `false` se almeno uno è `false`, `BOTTOM` se almeno uno è `BOTTOM` e nessun altro è `false` o *error*, *error* negli altri casi. L'operatore *or* è il duale di *and* e ha priorità minore. L'operatore unario *not* cambia i valori di `true` e `false`, ma non quelli di `BOTTOM` e *error*.

La valutazione di una *policy* termina con il *fulfilment* di tutte le *Obligations*. Questo consiste nel valutare tutti gli argomenti dell'azione corrispondente all'*obligation*. All'occorrenza di un errore, la decisione della *policy* sarà modificata in `indet`. Negli altri casi la decisione non cambierà e sarà quella del PDP prima del *fulfillment*.

Per valutare gli insiemi di *policy* si devono applicare dei *combining algorithm* specifici. Data una sequenza di policy in input gli algoritmi stabiliscono una sequenza di valutazioni per le *policy* date. Si propone in seguito l' algoritmo *permit-overrides* come esempio.

PERMIT-OVERRIDES Se la valutazione di una *policy* restituisce *permit*, allora il risultato è *permit*. In altre parole, *permit* ha la precedenza, indipendente dal risultato delle altre *policy*. Invece, se c'è almeno una *policy* che restituisce *deny* e tutte le altre restituiscono `not-app` o *deny*, allora il risultato è *deny*. Se tutte le policy restituiscono `not-app`, allora il risultato è `not-app`. In tutti gli altri casi, il risultato è `indet`.

Se il risultato della decisione è *permit* o *deny*, ogni algoritmo restituisce una sequenza di *fulfilled obbligations* conforme alla strategia δ di *fulfilment* scelta. Ci sono due possibili strategie:

All La strategia *all* richiede la valutazione di tutte le *policy* appartenenti alla sequenza in input e restituisce le *fulfilled obligation* relative a tutte le decisioni.

Greedy La strategia *greedy* stabilisce che, se ad un certo punto, la valutazione della sequenza di *policy* in input non può più cambiare, si può non esaminare le altre policy e terminare l'esecuzione. In questo modo si migliora le prestazioni della valutazione in quanto non si sprecano risorse computazionali per valutazioni di policy che non avrebbero alcun impatto sul risultato finale.

L'ultimo passo consiste nell'inviare la risposta del PDP al PEP per l'*enforcement*. A questo scopo, il PEP verifica che tutti gli eventuali obblighi imposti dal PDP al richiedente siano soddisfatti e decide, in base all' algoritmo di enforcement scelto, il comportamento per le decisioni di tipo `not-app` e `indet`. Gli algoritmi sono:

BASE Il PEP mantiene tutte le decisioni, ma se c'è un errore nella verifica degli obblighi il risultato è `indet`.

DENY-BIASED Il PEP concede l'accesso solo nel caso in cui questa sia la decisione del PDP e che gli eventuali obblighi imposti dal PDP siano stati soddisfatti. In tutti gli altri casi, il PEP nega l'accesso.

PERMIT-BIASED Questo algoritmo è il duale di deny-biased.

Questi algoritmi evidenziano il fatto che le *obligation* non solo influenzano il processo di autorizzazione, ma anche l'enforcement. Si nota infine che gli errori causati da obbligazioni opzionali sono ignorati.

3.4 ESEMPIO

Ora si propone un semplice esempio di politica in FACPL. Due utenti possono interagire con il sistema attraverso delle richieste. John può scrivere sulla risorsa "file.txt", ma non vi è specificata nessuna regola per la lettura. Invece Tom può leggere il "file.txt", ma non può scriverci.

Listing 3.1: Esempio di politica in FACPL

```
PolicySet filePolicy {  permit-overrides
  target:
    equal("file.txt", file_name/resource-id)
  policies:
    Rule writeRuleJ (  permit  target:
         equal ("WRITE" , subject/action )
        &&  equal ("John", subject/id)
    )
    Rule readRuleT (  permit  target:
         equal ("READ" , subject/action )
        &&  equal ("Tom", subject/id)
    )
    Rule writeRuleT (  deny  target:
         equal ("WRITE" , subject/action )
        &&  equal ("Tom", subject/id)
```

```
    )
   obl:
    [  deny M log_deny (subject / id )]
    [  permit M log_permit (subject / id)]
}
```

Ci sono 4 richieste. Entrambi gli utenti richiedono sia l'azione di "WRITE" che l'azione di "READ".

Listing 3.2: Esempio di richieste in FACPL

```
 Request:{ Request1
  (subject/action , "WRITE")
  (file_name/resource-id , "file.txt")
  (subject/id, "John")
}
 Request:{ Request2
  (subject/action , "READ")
  (file_name/resource-id , "file.txt")
  (subject/id, "John")
}
 Request:{ Request3
  (subject/action , "READ")
  (file_name/resource-id , "file.txt")
  (subject/id, "Tom")
}
 Request:{ Request4
  (subject/action , "WRITE")
  (file_name/resource-id , "file.txt")
  (subject/id, "Tom")
}
```

L'output dopo l'esecuzione sarà il seguente:

```
  Request: Request1

    Authorization Decision: PERMIT
    Obligations: PERMIT M log_permit([John])

  Request: Request2

    Authorization Decision: NOT_APPLICABLE
    Obligations:
```

```
Request: Request3

  Authorization Decision: PERMIT
  Obligations: PERMIT M log_permit([Tom])

Request: Request4

Authorization Decision: DENY
Obligations: DENY M log_deny([Tom])
```

La prima richiesta di scrittura da parte di John viene chiaramente accettata, la sua richiesta di lettura però ha come risultato *NOT APPLICABLE* in quanto non ci sono regole che possono essere usate per dare una valida risposta. La richiesta di lettuta di Tom invece viene accettata, mentre la sua ultima richiesta di scrittura riceve un deny perché è bloccata alla *rule* writeRuleT.

Da questo esempio si può vedere la semplicità con cui si possono scrivere le policy e le richieste. Le stesse politiche scritte in XML oltre a risultare prolisse a confronto, sono difficili da comprendere o analizzare.

Infine si fa notare che in questa versione, le richieste in ingresso sono completamente indipendenti tra loro e la parte dello Usage Control descritta in 2.2, in cui si enfatizza l'uso di controlli continuativi e la gestione di contesti mutabili, non è ancora implementata. Nel capitolo successivo si mostra la nuova struttura per per uno sviluppo basato proprio sullo Usage Control.

4

USAGE CONTROL IN FACPL

In questo capitolo si descrive l'estensione del linguaggio FACPL per implementare lo Usage Control. Più precisamente lo sviluppo è stato diviso in due parti. In primis la struttura è stata modificata in modo in cui si potesse creare delle politiche che si basassero sul comportamento passato. In seguito il linguaggio è stato ulteriormente esteso per migliorare la gestione dell'utilizzo continuativo di risorse .

In questo capitolo e nel succesivo si espone il secondo sviluppo, per la prima parte dell'estensione si rimanda alla tesi del mio collega Federico Schipani, con cui ho lavorato per l'implementazione dell'intera nuova struttura.

Il linguaggio è stato esteso in modo tale da ottimizzare la gestione di un insieme di richieste sulla stessa risorsa. Per far questo è stato necessaria una modifica sul processo di valutazione e l'inserimento di nuovi componenti.

4.1 IL PROCESSO DI VALUTAZIONE

Per memorizzare il comportamento passato, è stato essenziale l'aggiunta di un altro elemento nel processo di valutazione. Come si vede in figura 6, la nuova componente è Status.

Oltre agli *attribute names* il Context Handler avrà anche la possibilità di ricavare gli *status attributes*. Il PDP quindi potrà richiedere entrambi i tipi di attributi per elaborare la sua risposta. La decisione poi passerà al PEP come avveniva precedentemente.

Inoltre sono state aggiunte un tipo di obbligazioni che possono modificare lo Status. Il PEP quindi può far eseguire questo tipo di azioni per cambiare gli attributi.

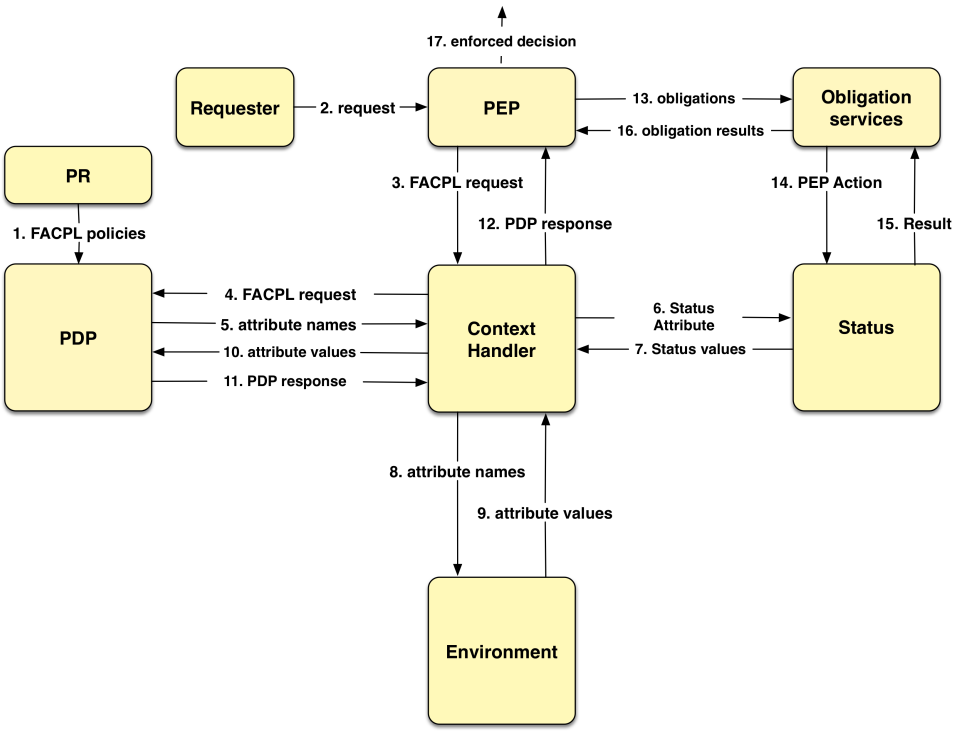

Figura 6: Processo di valutazione Status

Riprendendo il primo esempio 2.2.1, si riporta *Se uno degli amministratori sta scrivendo su un file, nessuno può leggerlo o apportare modifiche nello stesso momento*. Questa regola è realizzabile solo con l'utilizzo di uno status attribute(e.g. un booleano isWriting) che viene modificato da un'azione del PEP dopo che una richiesta di scrittura è stata accettata. Il PDP nelle successive richieste di lettura o di scritturà farà un controllo sull'attributo e negherà di conseguenza l'accesso.

Il passo successivo consiste nell'eliminazione della ridondanza dei controlli da parte del PDP.

Quando in 2.2.1, si scrive: *Letture ripetute, su uno stesso insieme di file, vengono gestite in modo efficiente*; Si fa riferimento al nuovo metodo per il controllo di richieste uniformi su una categoria di risorse e in figura 7 si mostra il processo di valutazione ridotto.

Si può definire un certo tipo di richieste che verrà interamente gestito dal PEP dopo una prima, e unica, valutazione del PDP. Se un sistema deve gestire numerose richieste di lettura, e non ci sono elementi che bloc-

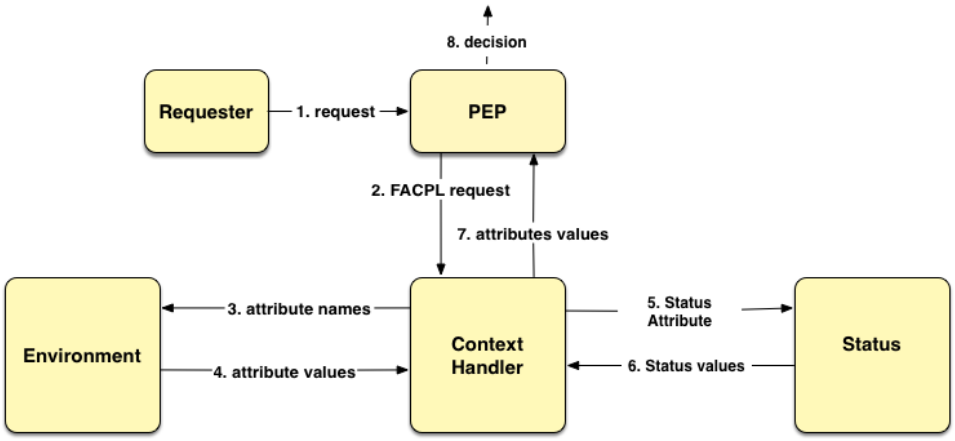

Figura 7: Processo di valutazione PEP-Check

cano un accesso, è possibile accettare una richiesta esaminando soltanto la tipologia e l'appartenenza di una risorsa ad un insieme predefinito.

Prendiamo per esempio un sistema che gestisce dei file dentro delle cartelle. Supponiamo che ci sia una cartella "Share" dove tutti gli utenti possono leggere solo se non ci sono utenti che scrivono. Il sistema può gestire richieste di lettura ripetute, facendo un unico controllo per la verifica dell'assenza di utenti che stanno scrivendo, alla prima domanda di accesso. Poi le richieste successive potranno essere gestite dal PEP con solo due controlli, uno sul tipo di azione, in questo caso una lettura, l'altro sull'appartenza alla cartella "Share". Se i due controlli non sono validi si ritorna alla valutazione completa con il PDP.

4.2 SINTASSI E SEMANTICA

La modifica nella struttura ha reso necessario apportare dei cambiamenti anche nella grammatica.

Al PAS è stato aggiunto *Status* della forma:

$$(\mathtt{status} : \mathtt{Attribute}^{+})^{?}$$

quindi lo *Status* è un elemento opzionale e può essere formato da almeno un *Attribute*.

Attribute invece è della forma:

$$(\texttt{Type Identifier}(= \texttt{Value})^?)$$

quindi è formato da un tipo, che può essere *int*, *float*, *boolean* o *date*, da una stringa identificatrice e da un valore.

Le *PepAction* sono state modificate in modo tale da eseguire operazioni sugli attributi dello stato e sono della forma: *nomeAzione*(*Attribute*,type) per esempio l'azione di somma di interi è *add*(*Attribute*,int).

Agli *Attribute Names* è stata aggiunta la produzione *Status/Identifier* per identificare i termini dello Status nelle *Expressions*.

La nuova produzione per le *Obligations* della forma:

$$[\texttt{Effect ObType (Expr)}^* \texttt{ CheckValue}^?]$$

è utilizzata per la nuova gestione delle richieste da parte del PEP. La nuova tipologia non ha pepAction, in quanto il PEP può modificare valori dello *Status* solo se il processo di valutazione è quello completo, cioè quello che include anche i controlli eseguiti dal PDP. Inoltre deve esserci un insieme di *Expression* affinchè il PEP possa fare le proprie verifiche e può esserci un valore opzionale che indica il tipo della scadenza. Da ora in poi il nuovo tipo di obbligazione sarà denominato *Obligation Check.*
Infine i *Check Values* della forma:

$$\texttt{cValue} ::= \texttt{Int} \mid \texttt{StringDate}$$

Indicheranno la scadenza delle *Obligation Check*. Int determina il numero massimo di richieste, mentre StringDate il limite temporale. Se CheckType viene omesso la *obligation* non ha scadenza.

Si propone adesso un esempio di una Policy che utilizza il nuovo tipo di obbligazione.

Nella Policy si vuole specificare che un utente di nome *Charlie* può effettuare una lettura solo se nessuno sta scrivendo e vuole utilizzare la risorsa *contabilita.xlsx*. Se la richiesta viene accettata si deve cambiare l' attributo *isReading* a true, oltre a ciò al susseguirsi di un altra richiesta di lettura il sistema cambiarà il tipo di processo valutativo e il PEP effettuerà i controlli solo sul tipo di azione e sul nome della risorsa.

Listing 4.1: Esempio per la sintassi

```
Policy readPolicy <  permit-overrides
  target: equal("Charlie",name/id) &&  equal("read", action/id) &&
     equal("contabilita.xlsx", resource/id)
```

```
   rules:
    Rule accessReadRule (
     permit  target:  equal(status/isWriting, false))
    obl:
    [ permit M flag(status/isReading, true)],
    [ permit M ( equal("read", action/id)) , (
       equal("contabilita.xlsx", resource/id))]
>

PAS {
 Combined Decision : false ;
 Extended Indeterminate : false ;
 Java Package : "readPolicy" ;
 Requests To Evaluate : Request_example ;
 pep:  deny- biased
 pdp:  deny- unless- permit
 status: [(boolean isWriting = false), (boolean isReading = false)]
 include readPolicy
}
```

Si fa notare che con questa porzione di codice:

Listing 4.2: Esempio per la sintassi

```
    [ permit M ( equal("read", action/id)) , (
       equal("contabilita.xlsx", resource/id))]
```

si esprime la nuova struttura per la creazione delle *Obligation Check* ed è questo il costrutto in cui si indicano i controlli che il PEP deve effettuare.

Semantica

La trasformazione della sintassi ha determinato anche variazioni nella semantica descritta in 3.3.

Il PDP fa uso degli *Status Attributes* per la valutazione di una richiesta in input. Questi attributi sono modificabili tramite alcune azioni dette *Pep Action*, come l'operazione di somma *add(Attribute,int)* oppure l'assegnamento di una data con *setDate(Attribute,date)*. Sono definiti nel PAS come descritto nell'esempio precedente con questa struttura:

Listing 4.3: Esempio per la sintassi

```
 status: [(boolean isWriting = false), (boolean isReading = false)]
```

Il PEP oltre a verificare che le Pep Action siano eseguite, con l'aggiunta delle *Obbligation Check*, ha il controllo nella gestione delle richieste continuative. Se nella policy è presente una obbligazione che fa parte del nuovo tipo, il PEP ha il compito di controllare che le *Expression* contenute nell'*Obligation Check* siano vere.

Nell'esempio precedente l'obbligazione è definita con:

Listing 4.4: Esempio per la sintassi

```
[ permit M ( equal("read", action/id)) , (
    equal("contabilita.xlsx", resource/id))]
```

il PEP deve verificare quindi che l'azione richiesta sia una "read" e che la risorsia sia "contabilita.xlsx". Se il controllo dà esito positivo la richiesta è subito accettata, se la richiesta non passa una delle verifiche si continuerà il processo di valutazione ripartendo dal PDP e svolgendo tutti i passi.

Le *Obligation Check* possono essere permanenti, limitate temporalmente oppure limitate sul numero di richieste accettate.

PERMANENTI Se sono del primo tipo e la verifica sulle *Expression* è sempre vera, il PEP continuerà a gestire le richieste senza il controllo del PDP. La gestione cambia solo quando una richiesta non è conforme a quelle accettate.

SCADENZA SUL TEMPO Se sono del secondo tipo, anche se la verifica sulle *Expression* è vera, quando il tempo limite è stato superato, la gestione delle richieste cambia e si riprende il processo valutativo completo.

SCADENZA SUL NUMERO DI RICHIESTE Il terzo tipo di *Obligation Check* è simile al secondo, con l'unica differenza che il limite non è temporale, ma sul numero di richieste elaborate.

4.3 ESEMPI

In questa sessione si mostrano gli esempio descritti in 2.2.1 utilizzando le nuove funzionalità.

4.3.1 *Accessi in lettura e scrittura di file*

Listing 4.5: Policy per esempio lettura e scrittura

```
PolicySet ReadWrite_Policy {  deny- unless- permit
  target: in ( name / id , set ("Alice", "Bob"))
  policies:

  PolicySet Write_Policy {  deny- unless- permit
    target: equal ("write", action/id)
    policies:
      Rule write (  permit  target:
        equal ( group / id, "Administrator") &&
        equal ( file / id, "thesis.tex") &&
        equal ( status / isWritingThesis , false ) &&
       )
    obl:
   [  permit M  flagStatus(isWritingThesis, true) ]
 }
  PolicySet Read_Policy {  deny- unless- permit
    target: in ( file/id) , set ("thesis.tex" , "facpl.pdf"))
    policies:
      Rule read (  permit  target:
        equal ("read", action/id) &&
        equal ( status / isWritingThesis , false )
       )
    obl:
   [  permit M  equal ("read",action/id) ,
       equal ( status / isWritingThesis , false ) ]
 }
  PolicySet StopWrite_Policy {  deny- unless- permit
    target:  equal ("stopWrite", action/id)
    policies:
      Rule stopWrite (  permit  target:
        equal("thesis.tex", file/id) &&
        equal ( status / isWritingThesis , true ) &&
        equal("Administrator", group/id) &&
       )
    obl:
   [  permit M  flagStatus(isWritingThesis, false) ]
 }
}
PAS {
 Combined Decision : false ;
 Extended Indeterminate : false ;
 Java Package : "exampleReadWrite" ;
```

```
 Requests To Evaluate : Request1, Request2, Request3, Request4,
     Request5, Request6 , Request7, Request8, Request9, Request10,
     Request11
 pep:  deny- biased
 pdp:  deny- unless- permit
 status: [(boolean isWritingThesis = false) ]
 include exampleReadWrite
}
```

Listing 4.6: Richieste per esempio lettura e scrittura

```
 Request:{ Request1
  (name / id , "Bob")
  (action / id, "read")
  (file / id, "thesis.tex")
}

 Request:{ Request2
  (name / id , "Bob")
  (action / id, "read")
  (file / id, "thesis.tex")
}

 Request:{ Request3
  (name / id , "Alice")
  (action / id, "read")
  (file / id, "facpl.pdf")
}
 Request:{ Request4
  (name / id , "Bob")
  (action / id, "read")
  (file / id, "thesis.tex")
}

 Request:{ Request5
  (name / id , "Bob")
  (action / id, "read")
  (file / id, "thesis.tex")
}

 Request:{ Request6
  (name / id , "Alice")
  (group / id , "Administrator")
```

```
  (action / id, "write")
  (file / id, "thesis.tex")
}
```

4.3.2 *Servizio di streaming*

Listing 4.7: Policy per esempio servizio streaming

```
PolicySet Streaming_Policy {  deny- unless- permit
  policies:

  PolicySet LoginAlice_Policy {  deny- unless- permit
    target: equal ( name / id, "Alice") &&
       equal ( action/id, "login")
    policies:
      Rule loginAlice(  permit  target:
         equal ( status / passwordAlice, password/id)
      )
    obl:
   [  permit M  SetString(loginAlice, "PREMIUM") ],
   [  permit M  flagStatus(streamingAlice, true) ]
 }

  PolicySet LoginBob_Policy {  deny- unless- permit
    target: equal ( name / id, "Bob") &&
       equal ( action/id, "login")
    policies:
      Rule loginBob(  permit  target:
         equal ( status / passwordBob, password/id)
      )
    obl:
   [  permit M  SetString(loginBob, "STANDARD") ],
   [  permit M  flagStatus(streamingBob, true) ]
 }

  PolicySet ListenAlice_Policy {  deny- unless- permit
    target: equal ( name / id, "Alice") &&
       equal ( action/id, "listen")
    policies:
      Rule listenAlice(  permit  target:
         equal ( status / loginAlice, "PREMIUM")
      )
```

```
    obl:
   [  permit M  equal(name/id, "Alice"),
      equal(status/streamingAlice, true)]
  }

   PolicySet ListenBob_Policy {  deny- unless- permit
    target: equal ( name / id, "Bob") &&
      equal ( action/id, "listen")
    policies:
     Rule listenBob(  permit  target:
        equal ( status / loginBob, "STANDARD") &&
        equal ( status / commercialsBob, false)
     )
    obl:
   [  permit M  equal(name/id, "Bob"),
      equal(status/streamingBob, true),"00:15:00"],
   [  permit M  flagStatus(commercialsBob, true) ]
  }

   PolicySet commercialsBob_Policy {  deny- unless- permit
    target: equal ( name / id, "Bob") &&
      equal ( action/id, "listenCommercials")
    policies:
     Rule listenBob(  permit  target:
        equal ( status / loginBob, "STANDARD") &&
        equal ( status / commercialsBob, true)
     )
    obl:
   [  permit M  flagStatus(commercialsBob, false) ]
  }
}
PAS {
  Combined Decision : false ;
  Extended Indeterminate : false ;
  Java Package : "exampleReadWrite" ;
  Requests To Evaluate : Request1, Request2, Request3, Request4,
      Request5, Request6 , Request7, Request8, Request9, Request10
  pep:  deny- biased
  pdp:  deny- unless- permit
  status: [(boolean isWritingThesis = false) ]
  include exampleReadWrite
}
```

Listing 4.8: Richieste per esempio servizio streaming

```
 Request:{ Request1
   (name / id , "Alice")
   (action / id, "listen")
}
 Request:{ Request2
   (name / id , "Alice")
   (action / id, "login")
   (password / id, "123456")
}
 Request:{ Request3
   (name / id , "Alice")
   (action / id, "listen")
}
 Request:{ Request4
   (name / id , "Bob")
   (action / id, "listen")
}
 Request:{ Request5
   (name / id , "Bob")
   (action / id, "login")
   (password / id, "abcdef")
}
 Request:{ Request6
   (name / id , "Bob")
   (action / id, "listen")
}
 Request:{ Request7
   (name / id , "Bob")
   (action / id, "listen")
}
 Request:{ Request8
   (name / id , "Bob")
   (action / id, "listen")
}
 Request:{ Request9
   (name / id , "Bob")
   (action / id, "listenCommercial")
}
 Request:{ Request10
   (name / id , "Bob")
   (action / id, "listen")}
```

Tabella 3: Syntax of FACPL

Policy Authorisation Systems	*PAS* ::=	(pep : *EnfAlg* pdp : *PDP* (status : [*Attribute*]$^+$)*)
Attribute	*Attribute* ::=	(*Type Identifier* (= Value)$^?$)
Type	*Type* ::=	int | boolean | date | float
Enforcement algorithms	*EnfAlg* ::=	base | deny-biased | permit-biased
Policy Decision Points	*PDP* ::=	{*Alg* policies : *Policy*$^+$}
Combining algorithms	*Alg* ::=	p-over$_\delta$ | d-over$_\delta$ | d-unless-p$_\delta$ | p-unless-d$_\delta$ | first-app$_\delta$ | one-app$_\delta$ | weak-con$_\delta$ | strong-con$_\delta$
fulfilment strategies	δ ::=	greedy | all
Policies	*Policy* ::=	(*Effect* target : Expr obl : *Obligation**) | {*Alg* target : Expr policies : *Policy*$^+$ obl : *Obligation** }
Effects	*Effect* ::=	permit | deny
Obligations	*Obligation* ::=	[*Effect ObType* PepAction(Expr*)] | [*Effect ObType* (*Expr*)* *CheckValue*$^?$]
PepAction	*PepAction* ::=	add(*Attribute*, int) | flag(*Attribute*, boolean) | sumDate(*Attribute*, date) | div(*Attribute*, int) | add(*Attribute*, float) | mul(*Attribute*, float) | sumString(*Attribute*, string) | setValue(*Attribute*, string) | setDate(*Attribute*, date)
Obligation Types	*ObType* ::=	M | O
Expressions	*Expr* ::=	*Name* | *Value* | and(*Expr*, *Expr*) | or(*Expr*, *Expr*) | not(*Expr*) | equal(*Expr*, *Expr*) | in(*Expr*, *Expr*) | greater-than(*Expr*, *Expr*) | add(*Expr*, *Expr*) | subtract(*Expr*, *Expr*) | divide(*Expr*, *Expr*) | multiply(*Expr*, *Expr*) | less-than(*Expr*, *Expr*)
Attribute Names	*Name* ::=	*Identifier*/*Identifier* | *Status*/*Identifier*
Check Values	*cValue* ::=	*Int* | *StringDate*
Literal Values	*Value* ::=	true | false | *Double* | *String* | *Date*
Requests	*Request* ::=	(*Name*, *Value*)$^+$

5

ESTENSIONE DELLA LIBRERIA FACPL

FACPL è basato su Java. Per implementare le nuove funzioni sono state aggiunte varie classi e sono state modificate alcune parti per adattare il comportamento della libreria ai nuovi componenti e alle nuove strutture.

Nelle sezioni successive si descriveranno gli aspetti più rilevanti del codice. L'intera libreria si può trovare su GitHub all'indirizzo

`https://github.com/andreamargheri/FACPL`

Lo sviluppo dell'estensione si può dividere in due passi principali. La prima parte consiste nel creare una classe PEP che può gestire le richieste e adattare il processo di valutazione. La seconda parte invece comprende l'implementazione di una gerarchia di *Obbligations* che possono essere sfruttate dal PEP per la nuova gestione delle richieste.

5.1 LA CLASSE PEPCHECK

5.2 LE OBBLIGATIONS CHECK

6

CONCLUSIONI

6.1 SVILUPPI FUTURI

BIBLIOGRAFIA

[1] NIST - *A survey of access Control Models* - http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf (Cited on page 5.)

[2] Jaehong Park, Ravi Sandhu - *The UCON Usage Control Model* - http://drjae.com/Publications_files/ucon-abc.pdf (Cited on page 7.)

[3] Aliaksandr Lazouski, Fabio Martinelli, Paolo Mori - *Usage control in computer security: A Survey* (Cited on page 8.)

[4] Aliaksandr Lazouski, Gaetano Mancini, Fabio Martinelli, Paolo Mori - *Usage Control in Cloud Systems* - Istituto di informatica e Telematica, Consiglio Nazionale delle Ricerche.

[5] Alexander Pretschner, Manuel Hilty, Florian Schutz, Christian Schaefer, Thomas Wlater - *Usage Control Engorcement*

[6] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, Francesco Tiezzi - *A Formal Framework for Specification, Analysis and Enforcement of Access Control Policies*

[7] Jaehong Park, Ravi Sandhu - *A Position Paper: A Usage Control (UCON) Model for Social Networks Privacy*

[8] Leanid Krautsevich, Aliaksandr Lazouski, Fabio Martinelli, Paolo Mori, Artsiom Yautsiukhin - *Usage Control, Risk and Trust*