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

Tesi di Laurea

PROGETTO E IMPLEMENTAZIONE IN FACPL DI UN MONITOR A RUNTIME PER IL SUPPORTO AL CONTROLLO CONTINUATIVO DEGLI ACCESSI

DEVISING A RUN-TIME MONITOR FOR CONTINUATIVE ACCESS CONTROLS WITH FACPL

FILIPPO MAMELI

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

Anno Accademico 2014-2015

Filippo Mameli: *Progetto e implementazione in FACPL di un monitor a runtime per il supporto al controllo continuativo degli accessi,* Corso di Laurea in Informatica, © Anno Accademico 2014-2015

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, infatti i dati più sensibili, se diffusi senza una valida autorizzazione, possono arrecare ad esempio danni economici ad una società o nuocere anche gli utenti nel privato. Per far fronte al problema sono stati sviluppati dei modelli per il controllo degli accessi. Tuttavia la quantità di dati e la complessità dei sistemi moderni ha mostrato i limiti delle tecnologie concepite e questo fatto ha portato inevitabilmente allo sviluppo di un nuovo modello: lo Usage Control (UCON)

In questa tesi si descrive un'estensione basata su UCON per Formal Access Control Policy Language (FACPL). Il problema principale da risolvere è la staticità del processo di valutazione del linguaggio. Tutte le richieste di accesso ricevono una risposta attraverso un sistema di verifica sempre uguale e i controlli risultano ridondati e in alcuni casi inutili. Nei capitoli successivi si mostrano i passi che hanno portato allo sviluppo di una nuova gestione delle richieste in grado di rendere dinamico il processo valutativo.

Il resto del documento è così strutturato:

- Nel capitolo 2 si introducono vari modelli di Access Control, si descrive lo UCON e si mostrano due casi di studio basati sul nuovo modello.
- Nel capitolo 3 si descrive la sintassi e la semantica di FACPL, il processo di valutazione e un esempio utilizzando il linguaggio.
- Nel capitolo 4 si mostra l'estensione di FACPL per migliorare la gestione delle richieste.

- Nel capitolo 5 si riportano le modifiche alla libreria Java su cui è basato il linguaggio per adattarlo al nuovo processo valutativo.
- Nel capitolo 6 si riassume il lavoro svolto e si presentato dei possibili sviluppi futuri.

2

ACCESS CONTROL E USAGE CONTROL

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 saranno presentate: Access Control List (ACL) in 2.1, Role Based Access Control (RBAC) in 2.2, Attribute Based Access Control (ABAC) in 2.3, Policy Based Access Control (PBAC) in 2.4. In 2.5 sarà descritto lo UCON e il recente modello UCON_{ABC} poi in 2.6 si esporranno due esempi che utilizzano il nuovo schema.

2.1 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 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.

2.2 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.

In figura 1 si mostra un gruppo di utenti facente parte di un *role* e la connessione tra i permessi e il ruolo.

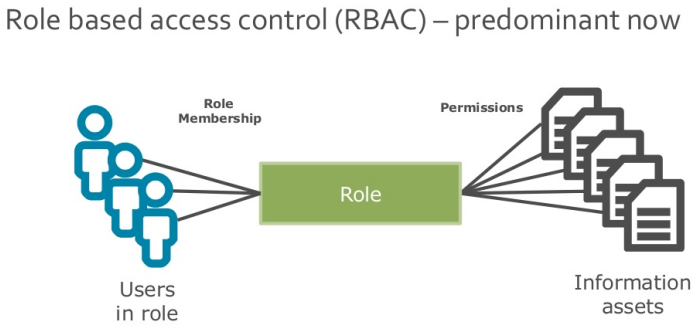

Figura 1: RBAC

2.3 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.

In figura 2 si illustra l'utilizzo da parte del sistema di valutazione(Rules engine) di attributi dell'*utente*(user), del *contesto*(environment) e delle *azioni*, per la decisione sull'autorizzazione.

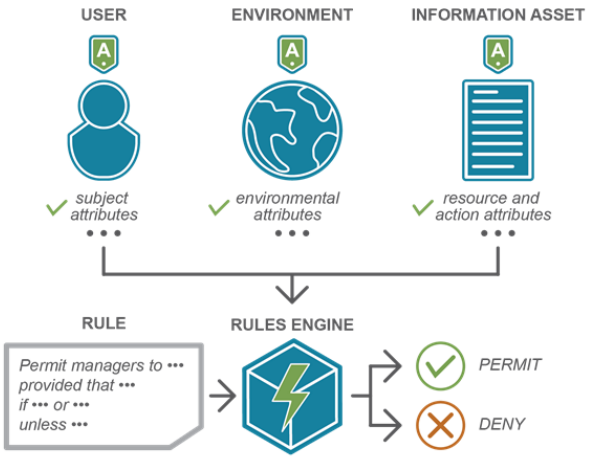

Figura 2: ABAC

Il modello non è utilizzato nei sistemi operativi, dove ACL e RBAC sono i 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 (PBAC) cerca di risolvere i difetti di ABAC.

2.4 POLICY BASED ACCESS CONTROL

PBAC riorganizza il modello ABAC per semplificare la gestione delle regole. Il sistema si basa su *politiche* che non sono altro che insiemi di *rule*. 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. Organization for the Advancement of Structured Information Standards (OASIS) 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.

2.5 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.

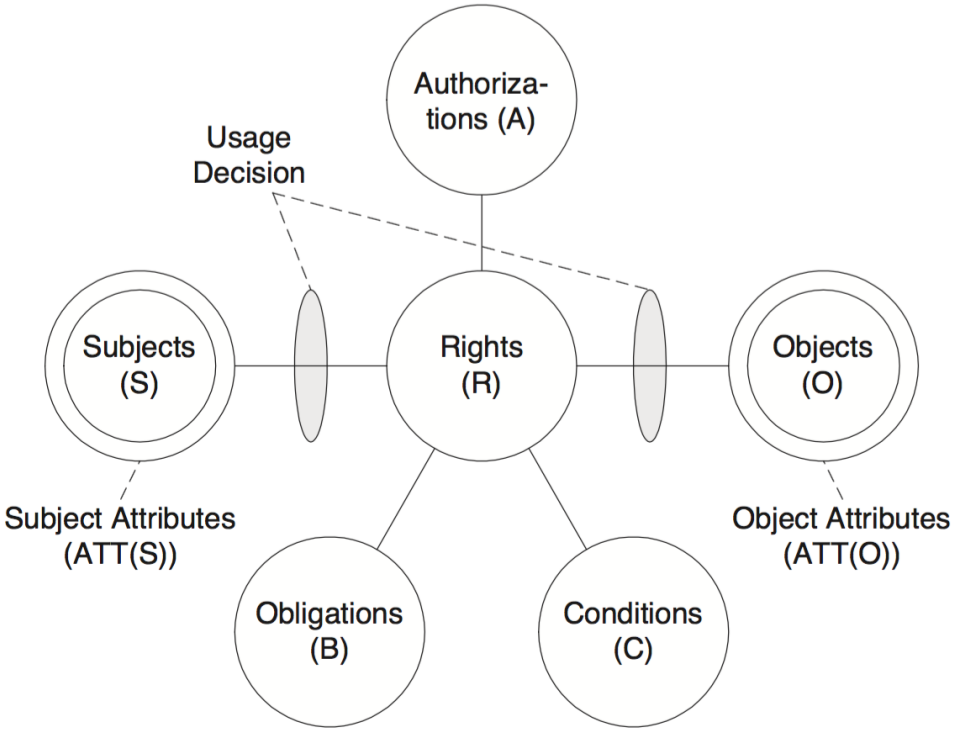

Figura 3: UCON

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*.

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

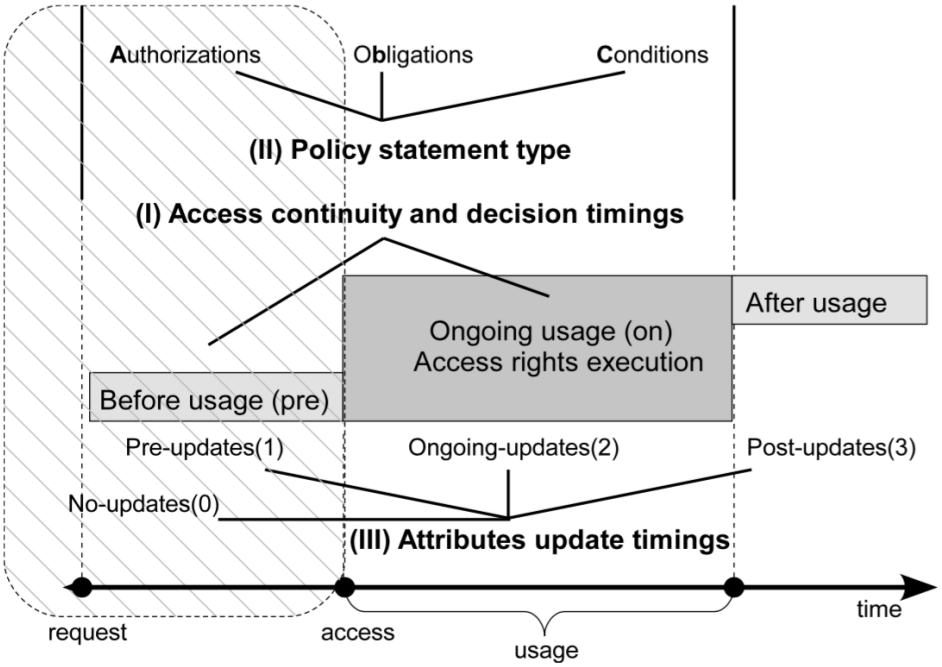

Figura 4: Fasi del processo di decisione

da una richiesta e non ha differenze con il processo valutativo dell'Access Control. Nella seconda invece si utilizzano i nuovi predicati introdotti ed è in questa parte che si affermano i controlli continuativi, le obbligazioni e le condizioni sull'utilizzo.

L'ultima fase 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.

2.6 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 scrittura 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 pubblicità prima di poter fare un'altra richiesta 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 in 2.4, l'organizzazione OASIS ha ideato il linguaggio XACML per sviluppare sistemi basati sul modello PBAC.

eXtensible Markup Language (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.

In questo capitolo si delinea prima di tutto il processo di valutazione in 3.1 poi mostrano le componenti della sistema in 3.2 e il loro significato in 3.3. Infine si mostrerà in 3.4 con un esempio l'utilizzo di FACPL .

3.1 IL PROCESSO DI VALUTAZIONE

In questa sezione si espone la sequenza di azioni che il sistema compie affinchè una richiesta in input sia valutata. In figura 5 si mostrano i passi che vengono eseguiti. Le componenti chiave del processo 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.

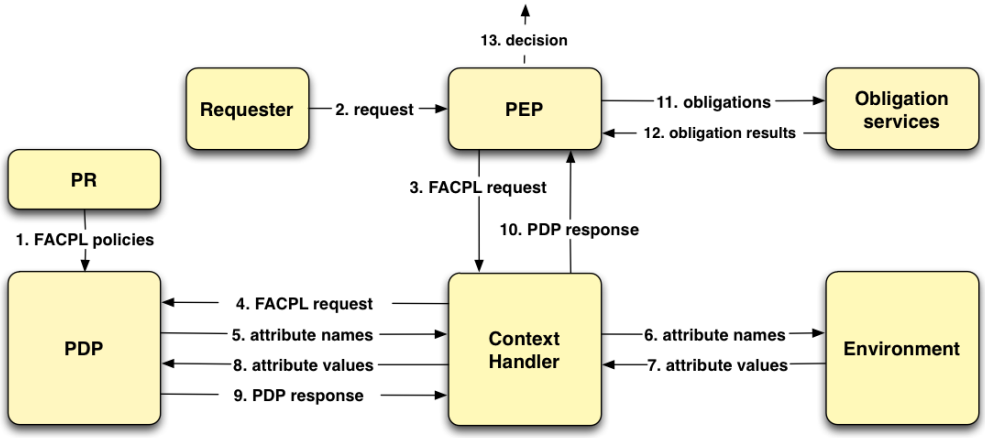

Figura 5: Processo di valutazione

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).

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 Extended Backus-Naur form (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.

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	$Request$ ::=	($Name$, $Value$)$^+$

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

La risposta ad una richiesta FACPL è scritta utilizzando la sintassi ausiliaria riportata in tabella 2. La valutazione in due passi descritta in 3.1 produce due tipi differenti di risposte:

- *PDP Response*
- *Decisions*

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**)]

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

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 algorithms*.

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 *obligations* 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 FACPL

In questo 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)]
}

PAS {
 Combined Decision : false ;
 Extended Indeterminate : false ;
 Java Package : "filePolicy" ;
 Requests To Evaluate : Request1, Request2, Request3, Request4
 pep:  deny- biased
 pdp:  deny- unless- permit
 include filePolicy
}
```

Nel PAS si indicano gli algoritmi usati e le richieste da includere. In questo esempio entrambi gli utenti richiedono sia l'azione di "WRITE" che l'azione di "READ". Qui di seguito si presenta la struttura delle *request*.

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 2.5, 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 un'estensione del linguaggio FACPL basata sul modello UCON. 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 stata necessaria una modifica sul processo di valutazione(3.1) e l'inserimento di nuovi componenti(4.2).

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.

Riprendendo il primo esempio 2.6, 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

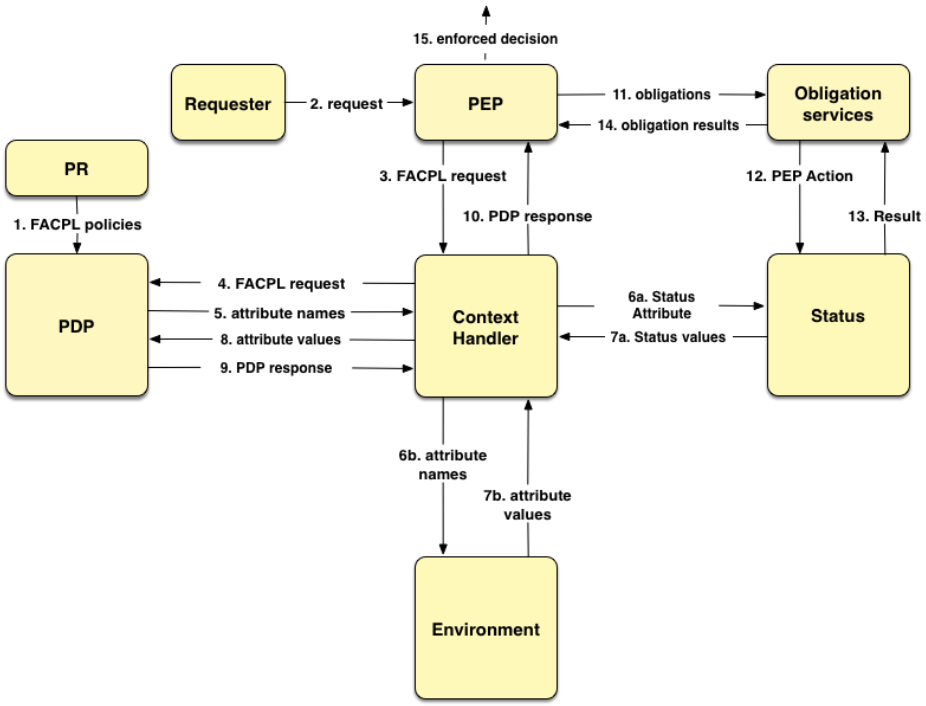

Figura 6: Processo di valutazione Status

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 2.6, 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 bloccano un accesso, è possibile accettare una richiesta esaminando soltanto la tipologia dell'azione 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

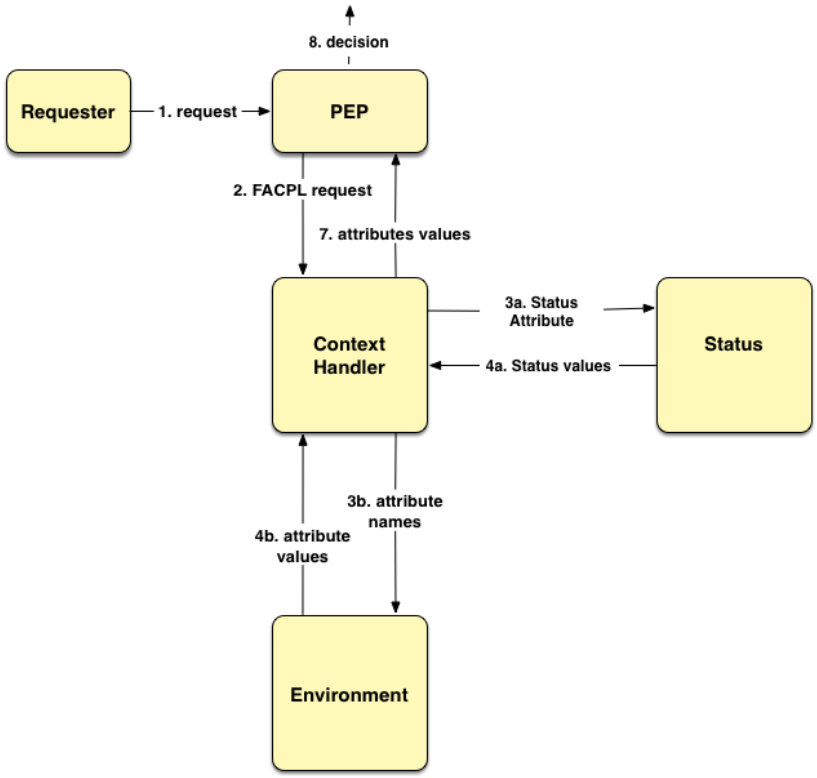

Figura 7: Processo di valutazione PEP-Check

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

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

Tabella 3: Sintassi ausiliaria per le risposte

PDP Responses	*PDPResponse* ::=	⟨ *Decision* *FObligation** ⟩
Decisions	*Decision* ::=	permit | deny | not-app | indet
Fulfilled obligations	*FObligation* ::=	[*ObType* PepAction(*Value**)]
		| [env : *Expr* status :*Expr*]

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:

$$[\texttt{Effect } env : \texttt{Expr } status : \texttt{Expr } expiration : \texttt{Value}^?]$$

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 *expiration:*$\texttt{Value}^?$ della forma:

$$\texttt{Value} ::= \texttt{Int} \mid \texttt{Date}$$

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

La sintassi delle risposte è rimasta quasi interamente invariata. Nella tabella 3 si può vedere che le decisioni non sono cambiate, mentre c'è una nuova produzione per le *Fulfilled obligations*.

La nuova forma indica le *Fulfilled obligations* di tipo Check. Queste hanno solo due *Expression*, una per l'*Environment* e una per lo *Status*. I due termini esprimono i controlli che deve effettuare il PEP prendendo i valori dal *Context Handler*. Le nuove *Fulfilled obligations* sono prive di pepAction in quanto, come enunciato prima, il PEP non può eseguire azioni, ma solo verifiche sugli attributi.

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 effetture 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 cambierà 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 ( equal("read", action/id)) , (
         equal("contabilita.xlsx", resource/id))]
>
```

Listing 4.2: Esempio PAS

```
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 4.3: Esempio per la sintassi

```
    [ permit ( 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. Come si vede, a differenza delle altre obbligations, non è specificato il tipo M o O (mandatory o optional) in quanto tutte le obligation check devono essere eseguite in ogni caso.

Tabella 4: 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* env : *Expr* status : *Expr* exp : *Value*$^?$]
PepAction	*PepAction* ::=	add(*Attribute*, int) | flag(*Attribute*, boolean) | sumDate(*Attribute*, date) | div(*Attribute*, int) | add(*Attribute*, float) | mul(*Attribute*, float) | 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*
Literal Values	*Value* ::=	true | false | *Double* | *String* | *Date*
Requests	*Request* ::=	(*Name*, *Value*)$^+$

4.3 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 *PepAction*, 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 4.4: Esempio status attributes

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

Il PEP oltre a verificare che le PepAction 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 4.5: Esempio Obligation Check

```
    [ permit ( 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.

4.4 ESEMPI

In questa sessione si mostrano gli esempio descritti in 2.6 utilizzando le nuove funzionalità.

4.4.1 *Accessi in lettura e scrittura di file*

Nel sistema ci sono tre utenti: Alice, Bob e Charlie. Alice e Charlie fanno parte del gruppo degli Administators, mentre Bob appartine ad un gruppo di default non specificato. Solo gli amministratori possono scrivere, però tutti possono leggere dei file precedentemente specificati.

Se Alice o Charlie fanno una richiesta di scrittura, hanno l'obbligo di cambiare il booleano *isWriting* associato al file in `true`. Se finiscono di scrivere invece devono fare una richiesta di *stopWrite* e assegnare `false` all'attributo. Ovviamente se non stanno scrivendo non possono richiedere la fine dell'azione quindi *stopWrite* è accettabile solo quando isWriting è `true`.

Tutti gli utenti possono leggere i file solo se nessuno sta scrivendo su questi e se i file appartengono ad un determinato insieme. Nella policy che riguarda le letture si specifica che i file debbano essere "thesis.tex" oppure "facpl.pdf", inoltre viene utilizzato il nuovo tipo di obbligation.

Il PAS è differenze dalla versione usata in 3.4 solo nella definizione dello status in cui si inizializza isWriting di "thesis.tex" e di "facpl.pdf" a false.

Listing 4.6: 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 ) &&
      equal ( status / isWritingFacpl , false )
      )
    obl:
   [  permit M  equal ("read",action/id) , in ( file/id , set
      ("thesis.tex" , "facpl.pdf")) ]
  }
   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),(boolean isWritingFacpl
     = false) ]
  include exampleReadWrite
}
```

Di seguito si mostrano 9 richieste per descrivere tutti i possibili scenari. Le prime tre *request* richiedono una lettura sia da parte di Bob che di Alice. Queste tre richieste sono simili tra loro, differiscono solo per i file richiesti e tutte saranno accette perché rispettano Read_Policy. La *request1* anche se essenzialmente uguale alle altre due passa per un processo di valutazione diverso. Infatti solo la prima richiesta sarà valutata dal PDP e dal PEP mentre le altre solo da quest'ultimo. Nella prima si controlla che l'azione sia "read", il file sia nell'insieme {"thesis.tex","facpl.pdf"} e che i due Boolean isWriting siano falsi. Nella seconda e nella terza invece il PEP verifica solo che l'azione sia una lettura e che il file richiesto sia valido. I controlli del PEP si esprimono in questa riga delle policy:

Listing 4.7: Policy per esempio lettura e scrittura

```
[  permit M  equal ("read",action/id) , in ( file/id , set
    ("thesis.tex" , "facpl.pdf")) ]
```

Listing 4.8: 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")
}
```

La quarta richiesta interrompe il processo valutativo ridotto in quanto l'azione è una "write" e si ritorna alla valutazione completa includendo i controlli del PDP. La *request5* e la *request6* non vengono accettate perché l'amministratore sta scrivendo sul file e solo successivamente alla richiesta di "stopWrite" di Alice, Bob può leggere il documento e Charlie può modificarlo. L'ultima richiesta è una lettura, ed essendo successiva ad un'altra read, si adotterà nuovamente il processo di valutazione ridotto.

Listing 4.9: Richieste per esempio lettura e scrittura

```
 Request:{ Request4
   (name / id , "Alice")
   (group / id , "Administrator")
   (action / id, "write")
   (file / id, "thesis.tex")
}
 Request:{ Request5
   (name / id , "Bob")
   (action / id, "read")
   (file / id, "thesis.tex")
}
 Request:{ Request6
   (name / id , "Charlie")
   (group / id , "Administrator")
   (action / id, "write")
   (file / id, "thesis.tex")
}
 Request:{ Request7
   (name / id , "Alice")
   (group / id , "Administrator")
   (action / id, "stopWrite")
   (file / id, "thesis.tex")
}
 Request:{ Request8
   (name / id , "Bob")
   (action / id, "read")
   (file / id, "thesis.tex")
}
 Request:{ Request9
   (name / id , "Alice")
   (action / id, "read")
   (file / id, "facpl.pdf")
}
```

4.4.2 *Servizio di streaming*

L'esempio seguente è simile al primo, ma in questo caso si fa uso anche delle *Obligations Check* limitate dal tempo. Ci sono due utenti. Alice è un utente *Premium* e può ascoltare le canzoni senza limiti una volta fatto il login. Bob è un utente standard e dopo un certo lasso di tempo non può

più fare richieste di ascolto e dovrà sentire la pubblicità prima di poter ascoltare di nuovo un brano.

Le "listen" dei due tipi di utenti usano obligations check differenti. Gli ascolti degli utenti premium sono associati ad una obbligazione persistente. Possono quindi richiedere i brani senza essere limitati per tutta la durata del login. Gli utenti standard invece possono ascoltare le canzoni per quindici minuti di fila come viene definito in questa riga:

Listing 4.10: Policy per esempio servizio streaming

```
[  permit M  equal(name/id, "Bob"),
    equal(status/streamingBob, true),"00:15:00"],
```

Tutti e due gli ascolti sono gestiti interamente dal PEP a partire dalla seconda richiesta consecutiva, come avveniva per le letture nell'esempio precedente.

Listing 4.11: 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
}
```

Le prime due *request* non sono accettate perché non è possibile richiedere gli ascolti prima di fare un login. Dopo le richieste di login, dove si usa un attributo password per validare l'accesso, i due utenti possono ascoltare i brani.

Si suppone che dopo tre richieste di ascolto, Bob alla *request8* abbia superato il limite di tempo di quindici minuti. Tutte le sue possibili richieste di ascolto sono rifiutate finchè non richiede di sentire la pubblicità. Dopo una richiesta di "listenCommercials" Bob ha di nuovo altri quindici minuti di ascolto possibili.

Listing 4.12: Richieste per esempio servizio streaming

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

5

ESTENSIONE DELLA LIBRERIA FACPL

La librearia FACPL è basata su Java. Per implementare le nuove funzioni sono state aggiunte varie classi e sono state modificate alcune parti per adattare il comportamento 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(5.1) consiste nel creare una classe PEP che può gestire le richieste e adattare il processo di valutazione. La seconda parte(5.2 e 5.3) invece comprende l'implementazione di una gerarchia di *Obbligations* che possono essere sfruttate dal PEP per la nuova gestione delle *request*.

Nella sezione(5.4) si descrive le classi java associate ai file FACPL dell'esempio in 4.4.2, inoltre si mostrano i miglioramenti prestazionali utilizzando il caso sviluppato in 4.4.1.

5.1 LA CLASSE PEPCHECK

Il normale flusso del processo valutativo passa prima dal PEP. Questo riceve la richiesta e poi la manda al PDP affinchè possa valutarla. Infine la risposta del PDP è passata al PEP che controlla le obbligazioni e elabora la decisione conclusiva. Il processo di valutazione ridotto invece avviene interamente nel PEP che usa solo il *Context Handler* per ricavare gli attributi dell'ambiente e dello stato.

Avendo bisogno di entrambe le modalità di valutazione, una per avere la decisione completa e l'altra per eliminare la ridondanza di verifiche da parte del PDP su richieste ripetute, la soluzione più appropriata è stata quella di creare un monitor in cui si potesse valutare l'esigenza d'uso di uno o dell'alto flusso valutativo.

La classe PEPCheck è il monitor che esegue i controlli per determinare la gestione più adatta al contesto. Questa estende la classe PEP e contiene come campo privato il PDP e la sua autorizzazione.

Listing 5.1: Costruttore

```
public class PEPCheck extends PEP {
  private List<AbstractFulfilledObligationCheck> checkObl;
  protected IEvaluableAlgorithmCheck checkAlg;
  private PDP pdp;
  private AuthorisationPDP authPDP;

  public PEPCheck(EnforcementAlgorithm alg,
      IEvaluableAlgorithmCheck combiningAlgorithm, PDP pdp) {
    super(alg);
    checkObl = new LinkedList<AbstractFulfilledObligationCheck>();
    checkAlg = combiningAlgorithm;
    this.pdp = pdp;
    authPDP = null;
  }
```

L'elemento che contraddistingue di più la nuova classe dal precedente PEP è la lista di *AbstractFulfilledObligationCheck* che è usata nel metodo *doAuthorisation* per prendere la decisione sul flusso valutativo. I casi da controllare sono tre:

Non ci sono *obligation check* in lista
Non si apporta nessun cambiamento al normale flusso del processo valutativo. Si invia la richiesta al PDP che poi la passa la sua decisione al PEP. Il codice successivo mostra la condizione che rende possibile il paradigma.

Listing 5.2: Processo senza obligation check

```
    if (checkObl.size() == 0) {
      /*
       * PDP Evaluation -> PEP ENFORCEMENT
       */
      // ENFORCEMENT BY PEP");
      authPDP = pdp.doAuthorisation(cxtReq);
      return this.doEnforcement(authPDP, cxtReq);
```

Nella lista c'è almeno una *obligation check*
Il sistema valuta le *Expression* associate alle obbligazioni con il

metodo *doPEPCheck*(vedi 4.1 per la struttura). Nel caso in cui la decisione sia `PERMIT` oppure `DENY`, il risultato viene subito restituito senza i controlli del PDP.

Listing 5.3: Processo con obligation check

```
/*
 * PEP EVALUATION
 */
// EVALUATING CHECK OBLIGATION");
result = this.doPEPCheck(cxtReq);
StandardDecision dec = result.getDecision();
l.debug("CHECK RESULT: " + dec);
if (dec == StandardDecision.PERMIT || dec ==
    StandardDecision.DENY) {
  return result;
```

Il controllo sulla lista restituisce error/not_applicable

Se la decisione non ricade nei casi precedenti, si è verificato un errore nella valutazione delle espressioni. Se ci sono problemi nei controlli, il PEPCheck, prima di inviare la risposta, esegue una valutazione completa facendo il *doAuthorisation* sul PDP e *doEnforcement*.

Listing 5.4: error/NOT_APPLICABLE durante il processo

```
if (dec == StandardDecision.PERMIT || dec ==
    StandardDecision.DENY) {
  return result;
} else {
  /*
   * if check obligation returns an error -> PDP
       Evaluation -> PEP
   * Enforcement
   */
  l.debug("BACK TO PDP");
  authPDP = pdp.doAuthorisation(cxtReq);
  this.clearAllObligations();
  return this.doEnforcement(authPDP, cxtReq);
}
```

Il *doEnforcement* nel codice 5.4 del PEPCheck richiama il metodo implementato in PEP. Nella linea `super.doEnforcement(authPDP, cxtReq);`

come si mostra di seguito.

Listing 5.5: Invocazione metodo doEnforcement su PEP

```
public AuthorisationPEP doEnforcement(AuthorisationPDP authPDP,
    ContextRequest_Status cxtReq) {
  /*
   * normal PEP enforcement + addiction of CheckObligation
   */
  AuthorisationPEP first_enforcement;
  Logger l = LoggerFactory.getLogger(PEPCheck.class);
  first_enforcement = super.doEnforcement(authPDP, cxtReq); //
      enforcement
  StandardDecision dec = first_enforcement.getDecision();
  l.debug("FIRST ENFORCEMENT COMPLETED, DECISION: " + dec);
  if (dec != StandardDecision.PERMIT) {
    return new AuthorisationPEP(first_enforcement.getId(), dec);
  }
```

Se la decisione non è `PERMIT` viene restituita subito la risposta negativa sull'autorizzazione e si conclude la valutazione. Se invece la decisione è positiva si esegue un controllo sulle *FulfilledObligations* e nel caso in cui queste siano FulfilledObligationCheck oppure FulfilledObligationTimeCheck, sono aggiunte alla lista di AbstractFulfilledObligationCheck di PEPCheck. Solo dopo l'inserimento delle obbligazioni si restituisce la decisione ricavata.

Listing 5.6: Aggiunta delle FulfilledObligationCheck

```
  l.debug("ADDING CHECK OBLIGATION TO PEP...");
  Iterator<AbstractFulfilledObligation> iterator =
      authPDP.getObligationIterator();
  while (iterator.hasNext()) {
    AbstractFulfilledObligation o = iterator.next();
    if (o instanceof FulfilledObligationCheck) {
      FulfilledObligationCheck temp = null;
      try {
        temp = (FulfilledObligationCheck)
            ((FulfilledObligationCheck) o).clone();
      } catch (CloneNotSupportedException e) {
        e.printStackTrace();
      }
      if (!checkObl.contains(temp)) {
        l.debug("ADDED: " + temp);
```

```
            checkObl.add(temp);
          }
        }else if (o instanceof FulfilledObligationTimeCheck) {
          FulfilledObligationTimeCheck temp = null;
          try {
            temp = (FulfilledObligationTimeCheck)
                ((FulfilledObligationTimeCheck) o).clone();
          } catch (CloneNotSupportedException e) {
            e.printStackTrace();
          }
          if (!checkObl.contains(temp)) {
            l.debug("ADDED: " + temp);
            checkObl.add(temp);
          }}}
      l.debug("...CHECK OBLIGATION ADDED");
      return first_enforcement;
    }
```

La lista modificata in 5.6 è utilizzata nel metodo *doPEPCheck*(nel listato 5.7) per eseguire la valutazione di ogni singola obligation check senza passare dal PDP. Questo metodo controlla che i valori siano corretti confrontandoli con gli attributi del contesto. Se non ci sono errori e le espressioni hanno superato i controlli, si ritiene la richiesta verificata utilizzando il processo di valutazione ridotto.

Listing 5.7: Valutazione FulfilledObligationCheck

```
    l.debug("DOING PEP CHECK:");
    AuthorisationPEP r = new AuthorisationPEP();
    StandardDecision dec;
    LinkedList<StandardDecision> decisionList = new
        LinkedList<StandardDecision>();
    for (AbstractFulfilledObligationCheck obl : checkObl) {
      if (obl instanceof FulfilledObligationCheck){
        dec =
            ((FulfilledObligationCheck)obl).getObligationResult(ctxRequest);
        decisionList.add(dec);
        if (StandardDecision.NOT_APPLICABLE.equals(dec)) {
          r.setDecision(StandardDecision.NOT_APPLICABLE);
          return r;
        }
      }else if (obl instanceof FulfilledObligationTimeCheck){
        dec =
```

```
            ((FulfilledObligationTimeCheck)obl).getObligationResult(ctxRequest);
        decisionList.add(dec);
        if (StandardDecision.NOT_APPLICABLE.equals(dec)) {
          r.setDecision(StandardDecision.NOT_APPLICABLE);
          return r;
        }
      }
    }
    r = checkAlg.evaluate(decisionList, ctxRequest);
    checkAlg.resetAlg();
    return r;
```

Nella figura 8 si mostra il grafico UML della classe PEPCheck e i relativi elementi associati.

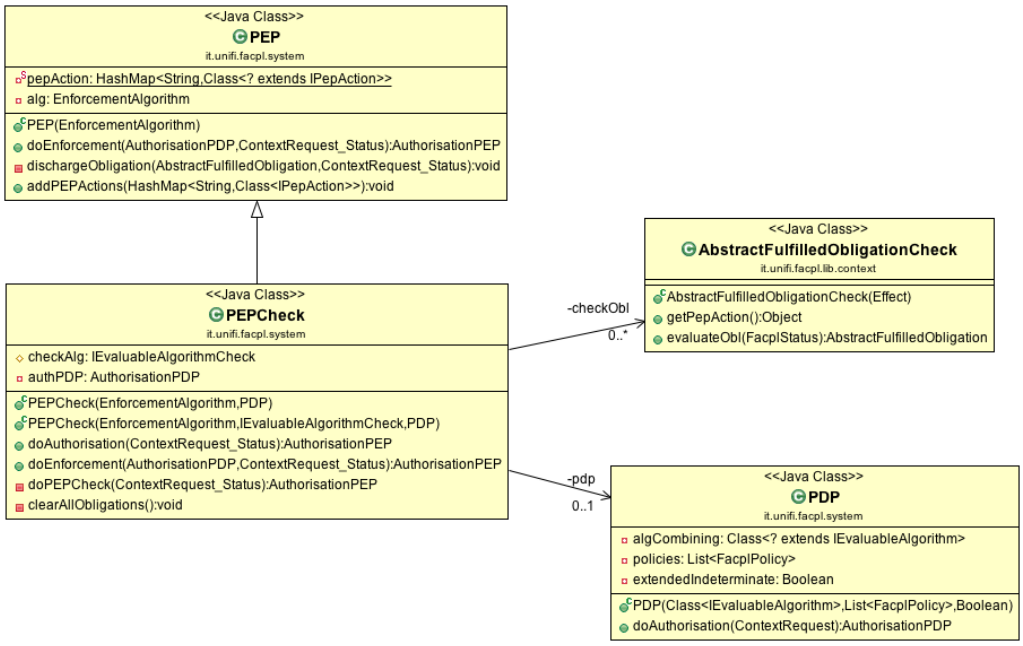

Figura 8: Diagramma PEPCheck

Nella sezione successiva 5.2 si descrive le Obligations utilizzate nel metodo *doPEPCheck* listato in 5.7.

5.2 LE OBLIGATIONS CHECK

La libreria è stata estesa con le *Obbligations Check* perché c'era bisogno di obbligazioni che avessero solo *Expressions* da verificare senza avere nella struttura le pepAction. La *Obbligation Check* è una classe che estende

la preesistente *AbstractObligation*. Nel codice 5.8 si mostra che la classe ha come campi due alberi di espressioni booleane, che possono a loro volta contenere altre *expressions*, due classi per indicare la scadenza e una variabile che determina il tipo di obligation check.

Listing 5.8: Costruttore e campi

```
public class ObligationCheck extends AbstractObligation {
  private ExpressionBooleanTree target;
  private ExpressionBooleanTree status_target;
  private int expiration;
  private FacplDate expirationTime;
  private CheckObligationType type;
  /*
   * four constructor for all combination of Expression:
   * 1: ExpressionFunction, ExpressionFunction
   * 2: ExpressionBooleanTree, ExpressionBooleanTree
   * 3: ExpressionBooleanTree, ExpressionFunction
   * 4: ExpressionFunction, ExpresisonBooleanTree
   */
  public ObligationCheck(Effect evaluatedOn, ExpressionFunction
      target,
       ExpressionFunction status_target, int expiration) {
    super(evaluatedOn);
    this.target = new ExpressionBooleanTree(target);
    this.status_target = new ExpressionBooleanTree(status_target);
    this.init(expiration);
  }
```

La scadenza, come già detto in 4.3, può essere di tre tipi: persistente, limitata temporalmente oppure limitata dal numero di richieste gestite. I tipi sono indicati con "P" per le obbligation Persistenti, "T" per le Temporali e "N" per quelle Numeriche come si mostra nell'`enum` in 5.9.

Listing 5.9: Enum delle Obligations

```
package it.unifi.facpl.lib.enums;

public enum CheckObligationType {
  P, N ,T
}
```

Per ogni obbligazione ci sono quattro struttori in modo tale da poter creare l'oggetto per ogni combinazione di *Expressions* in input. In 5.10

si mostra un construttore per le Obligation Temporali che prende in input due ExpressionBooleanTree, la scadenza determinata da *FacplDate expiration* e l'effetto (che può essere `PERMIT` oppure `DENY`).

Listing 5.10: Costruttore Obligation Temporale

```
public ObligationCheck(Effect evaluatedOn, ExpressionBooleanTree
    target,
     ExpressionBooleanTree status_target, FacplDate expiration) {
  super(evaluatedOn);
  this.target = target;
  this.status_target = status_target;
  this.init(expiration);
}
```

Il metodo *init* in 5.11, presente in tutti i costruttori, inizializza i campi della classe a seconda del tipo di scadenza data in input. Se la scadenza non è presenste si utilizza il metodo init per creare le obbligazioni di tipo Persistente.

Listing 5.11: init

```
private void init(int expiration) {
  this.expiration = expiration;
  this.pepAction = "CHECK";
  this.type = CheckObligationType.N;
}

private void init(FacplDate expiration) {
  this.expirationTime = expiration;
  this.pepAction = "CHECK";
  this.type = CheckObligationType.T;
}

private void init() {
  this.pepAction = "CHECK";
  this.type = CheckObligationType.P;
}
```

In *createObligation* in 5.12 si utilizzano i campi assegnati dai costruttori e da init per la creazione delle *FulfilledObligationCheck* che sono descritte più approfonditamente in 5.3.

Listing 5.12: Creazione di FulfilledObligationCheck

```
@Override
protected AbstractFulfilledObligation createObligation() {
  /*
   * this method create a fulfilledobligationcheck
   */
  if (this.type == CheckObligationType.N) {
    return new FulfilledObligationCheck(this.evaluatedOn,
        this.target, this.status_target,
         this.expiration);
  } else if (this.type == CheckObligationType.T){
    return new FulfilledObligationTimeCheck(this.evaluatedOn,
        this.target, this.status_target,
         this.expirationTime);
  } else{
    return new
        FulfilledObligationCheckPersistent(this.evaluatedOn,
        this.target,
         this.status_target);
  }
}
```

Nella figura 9 si mostra il diagramma della classe ObligationCheck.

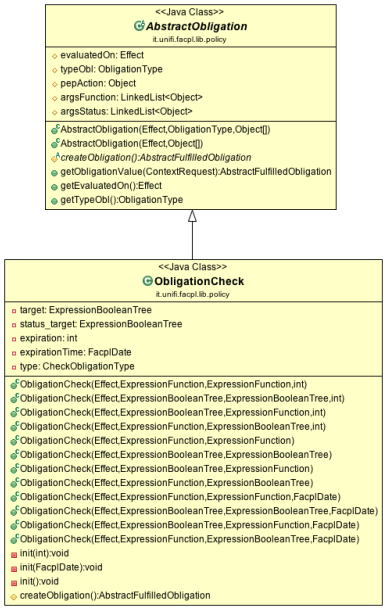

Figura 9: Diagramma ObligationCheck

5.3 FULFILLED OBLIGATION CHECK

Le *Fulfilled Obligation Check* sono una gerarchia di classi che è stata aggiunta come estensione alla *AbstractFulfilledObligation* già presente nella libreria. Ogni tipo di Obligation check ha una Fulfilled Obligation Check associata. Quindi sono state create tre tipi di Fulfilled:

1. FulfilledObligationNumCheck
2. FulfilledObligationCheckPersistent
3. FulfilledObligationTimeCheck

Le tre *Fulfilled Obligation Check* sono molto simili tra loro e l'unica differenza è la loro gestione della scadenza.

In figura 10 si mostra il diagramma della gerarchia di classi.

Figura 10: Diagramma FulfilledObligationCheck

Listing 5.13: FulfilledObligationCheck N

```
public FulfilledObligationNumCheck(Effect evaluatedOn,
    ExpressionFunction target,
     ExpressionBooleanTree status_target, int expiration) {
  super(evaluatedOn);
  this.target = new ExpressionBooleanTree(target);
  this.status_target = status_target;
  this.expiration = expiration;
  this.hasExpired = false;
  this.originalExpiration = expiration;
}
```

Quando si crea una Fulfilled Obligation con una scadenza determinata da un numero intero, si utilizza la *FulfilledObligationNumCheck*. In 5.13 si mostra il costruttore di questo tipo di FulfilledObligation nel caso in cui la prima espressione sia una *ExpressionFunction*, la seconda una *ExpressionBooleanTree* e la scadenza sia, ovviamente, un intero.

In 5.14 e in 5.15 si mostra il funzionamento di *GetObligationResult*. Questo metodo ha il compito, sia di verificare il valore di verità delle espressioni associate, sia di gestire il valore di *expiration*.

Expiration valido

Se l'intero che identifica la scadenza è maggiore di zero, si possono valutare le *Expressions* e richiamare il metodo *subExpiration*(listato 5.16) per aggiornare il valore.

Listing 5.14: Condizione di expiration valido

```
if (this.getExpiration() > 0 || this.getExpiration()==
    -1) {
  //if not expired -> evaluate target
  l.debug("EVALUATING EXPRESSION OF OBLIGATION: " +
     "\r\n");
  result_target =
     target.evaluateExpressionTree(cxtRequest);
  result_target_status =
     status_target.evaluateExpressionTree(cxtRequest);
  this.subExpiration(1);
  l.debug("RESULT_TARGET: " + result_target + " ||
     RESULT_TARGET_STATUS: " + result_target_status);
```

Expiration scaduta

Se si raggiunge la scadenza(in questo caso quando expiration è

zero), le expressions non sono valutate e si assegna direttamente il valore ERROR.

Listing 5.15: Condizione di expiration scaduta

```
} else if (this.getExpiration() == 0) {
  l.debug("OBLIGATION CHECK HAS EXPIRED");
  result_target = ExpressionValue.ERROR;
  result_target_status = ExpressionValue.ERROR;
}
```

In 5.14 si richiama *subExpiration*. Il metodo è utilizzato per aggiornare il valore della scadenza. Come si vede nel listato 5.16 se il valore è -1 non si modifica expiration (spiegherà il motivo in seguito nel paragrafo delle FulfilledObligationCheckPersistent). Negli altri casi si sottrae una unità a *expiration* e se si raggiunge il valore zero, si assegna al campo hasExpired `true`.

Listing 5.16: subExpiration

```
public void subExpiration(int i) {
  Logger l =
      LoggerFactory.getLogger(FulfilledObligationNumCheck.class);
  if (expiration == -1){
    l.debug("EXPIRATION PERSISTENT");
  }
  if (expiration > 0) {
    expiration -= i;
    l.debug("NEW EXPIRATION: " + this.toString());
    if (expiration == 0) {
      this.setExpired();
    }
  }
}
```

La Fulfilled persistente estende la classe FulfilledObligationNumCheck. Il costruttore richiama la classe superiore e inizializza la scadenza numerica con il valore speciale -1. Come si è mostrato in nel codice 5.16, se *expiration= -1* il metodo non modifica mai il valore. In questo modo in *GetObligationResult* di FulfilledObligationNumCheck non si raggiungerà il limite, le funzioni non restituiranno mai il risultato `ExpressionValue.ERROR` associato ad un obbligazione scaduta e si valuteranno sempre le *Expressions*.

Nel codice 5.17 si mostra il costruttore per FulfilledObligationCheck di tipo "P".

Listing 5.17: FulfilledObligationCheck P

```
public class FulfilledObligationCheckPersistent extends
    FulfilledObligationNumCheck {
  /*
   * same constructor of FulfilledObligationCheck,
   * but expiration is initialized with -1
   */
  public FulfilledObligationCheckPersistent(Effect evaluatedOn,
      ExpressionBooleanTree target,
       ExpressionBooleanTree status_target) {
    super(evaluatedOn, target, status_target, -1);
  }
```

Infine se la scadenza è di tipo *FACPLDate* allora si utilizzerà le FulfilledObligationTimeCheck. Queste si comportano in modo analogo alle *FulfilledObligationNumCheck*. La differenza che le contraddistingue è l'utilizzo dei metodi *before* e *after* per i controlli sulla scadenza.

Listing 5.18: getObligationResult per le tipo T

```
    FacplDate TIME = new FacplDate();
    if (TIME.before(this.getExpiration())) {
      l.debug("TIME "+TIME.toString());
      l.debug("NOT EXPIRED");
      //if not expired -> evaluate target
      l.debug("EVALUATING EXPRESSION OF OBLIGATION: " +
          "\r\n");
      result_target = target.evaluateExpressionTree(cxtRequest);
      result_target_status =
          status_target.evaluateExpressionTree(cxtRequest);
      this.checkExpiration();
      l.debug("RESULT_TARGET: " + result_target + " ||
          RESULT_TARGET_STATUS: " + result_target_status);
    } else if (TIME.after(this.getExpiration()) ||
        TIME.equals(this.getExpiration()) ) {
      this.checkExpiration();
      l.debug("TIME "+TIME.toString());
      l.debug("OBLIGATION CHECK HAS EXPIRED");
      this.checkExpiration();
      result_target = ExpressionValue.ERROR;
```

```
      result_target_status = ExpressionValue.ERROR;
18  }
```

Inoltre, al posto del metodo *subExpiration* in *GetObligationResult*, si richiama *checkExpiration*. La funzione è utilizzata in questo caso solo per richiamare il metodo di debug per mostrare la differenza tra il tempo relativo alla determinata richiesta e la scadenza. In 5.19 si mostra come viene calcolato il divario tra la scadenza e il tempo.

Listing 5.19: Time difference

```
  timeDiff=this.expirationTime.getDate().get(Calendar.SECOND)
2                      -TIME.getDate().get(Calendar.SECOND);
  if (TIME.before(this.expirationTime)) {
4   l.debug("TIME TO EXPIRATION: " + timeDiff);
```

5.4 ESEMPIO IN JAVA

In questa sezione si ripropongono gli esempi descritti in 4.4 mostrando le classi java relative ai file FACPL.

5.4.1 *Esempio Streaming*

Le classi associate all'esempio di servizio streaming (4.4.1) sono StatusStreaming,le ContextRequest degli utenti Alice e Bob, PolicySet_Streaming e il MainFACPL.

La classe StatusStreaming contiene FacplStatus. Si fa notare che nel codice 5.20, lo status viene creato una sola volta e sarà unico per tutte le richieste. Se non è inizializzato si creano gli attributi e si aggiungono al campo FacplStatus, mentre se è già inizializzato il metodo `getStatus()` restituirà direttamente `status`.

Listing 5.20: Status

```
1 public class StatusStreaming {
  private static FacplStatus status;
3
    public StatusStreaming() {
5   }

7   public FacplStatus getStatus() {
      if (status==null){
```

```
        HashMap<StatusAttribute, Object> attributes = new
            HashMap<StatusAttribute, Object>();
        attributes.put(new StatusAttribute("loginBob",
            FacplStatusType.STRING),"noLogin");
        attributes.put(new StatusAttribute("loginAlice",
            FacplStatusType.STRING),"noLogin");
        attributes.put(new StatusAttribute("passwordAlice",
            FacplStatusType.STRING), "123456");
        attributes.put(new StatusAttribute("passwordBob",
            FacplStatusType.STRING), "abcdef");
        attributes.put(new StatusAttribute("streamingAlice",
            FacplStatusType.BOOLEAN), false);
        attributes.put(new StatusAttribute("streamingBob",
            FacplStatusType.BOOLEAN), false);
        attributes.put(new StatusAttribute("passwordAlice",
            FacplStatusType.STRING), "123456");
        attributes.put(new StatusAttribute("passwordAlice",
            FacplStatusType.STRING), "123456");
        attributes.put(new StatusAttribute("commercialsBob",
            FacplStatusType.BOOLEAN), false);
        status = new FacplStatus(attributes,
            this.getClass().getName());
        return status;
      }
      return status;
    }
}
```

Le richieste di ascolto e di login sono simili tra loro. Queste si differenziano in base agli attributi associati. Nel codice 5.21 si mostra la richiesta di ascolto di Alice come esempio di *request*. Per prima cosa si creano due attributi, uno per il nome dell'utente(Alice) e uno per il tipo di azione(listen). Dopo la creazione, i due *attributes* sono aggiunti alla *request*. Infine si associa il contesto della richiesta allo Status con il metodo `setStatus()` che prende come attributo FacplStatus usando `getStatus()` di StatusStreaming.

Listing 5.21: Request

```
public class ContextRequest_ListenPremiumAlice {
  private static ContextRequest_Status CxtReq;

  public static ContextRequest_Status getContextReq() {
```

```
    if (CxtReq != null) {
      return CxtReq;
    }
    // create map for each category
    HashMap<String, Object> req_category_attribute_name = new
        HashMap<String, Object>();
    HashMap<String, Object> req_category_attribute_action = new
        HashMap<String, Object>();
    // add attribute's values
    req_category_attribute_name.put("id", "Alice");
    req_category_attribute_action.put("id", "listen");
    // add attributes to request
    Request req = new Request("listen alice");
    req.addAttribute("name", req_category_attribute_name);
    req.addAttribute("action", req_category_attribute_action);
    // context stub: default-one
    CxtReq = new ContextRequest_Status(req,
        ContextStub_Default.getInstance());
    StatusStreaming st = new StatusStreaming();
    CxtReq.setStatus(st.getStatus());
    return CxtReq;
  }
}
```

Il codice 5.22 è relativo alla Policy di ascolto dell'utente standard Bob. Il target della policy è composto da due *Expressions*. La prima è associata all'azione che deve essere una `liste`, la seconda invece si riferisce al nome dell'utente che deve essere `Bob`.

Si fa notare la creazione di una FacplDate (new FacplDate("00:15:00")) di quindici minuti per assegnare la scadenza all'obligation check. Nell'obbligazione si valuta inoltre che l'utente sia ancora `Bob` e che il booleano `streamingBob`(inizializzato a vero solo dopo il login) sia true.

Listing 5.22: Policy Set

```
  private class PolicySet_ListenBob extends PolicySet {
    public PolicySet_ListenBob() {
      addId("ListenBob_Policy");
      // Algorithm Combining
      addCombiningAlg(it.unifi.facpl.lib.algorithm.DenyUnlessPermitGreedy.class);
      // Target
      ExpressionFunction e1=new
          ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
```

```
        "listen",
         new AttributeName("action", "id"));
    ExpressionFunction e2=new
        ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
        "Bob",
         new AttributeName("name", "id"));
    ExpressionBooleanTree ebt = new
        ExpressionBooleanTree(ExprBooleanConnector.AND, e1, e2);
    addTarget(ebt);
    // PolElements
    addPolicyElement(new Rule_ListenBob());
    // Obligation
    addObligation(
         new ObligationCheck(Effect.PERMIT,
             new
                 ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.c
                 "Bob",
                  new AttributeName("name", "id")),
             new
                 ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.c
                       new StatusAttribute("streamingBob",
                           FacplStatusType.BOOLEAN),
                  true),
             new FacplDate("00:15:00")));
}
```

La regola da valutare associata alla PolicySet è stata aggiunta con il metodo `addPolicyElement(new Rule_ListenBob())`. Nella `Rule_ListenBob`, mostrata nel listato 5.23, si controlla che Bob abbia fatto il login e che non debba ascoltare la pubblicità. Dopo il controllo si modifica il booleano `commercialsBob` a true in modo tale che, dopo la scadenza della ObligationCheck, Bob non possa fare un'altra richiesta di ascolto.

Listing 5.23: Policy Set

```
private class Rule_ListenBob extends Rule {
  Rule_ListenBob() {
    addId("ListenBob_Rule");
    // Effect
    addEffect(Effect.PERMIT);
    ExpressionFunction e1=new
        ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
         new StatusAttribute("loginBob",
```

```
                FacplStatusType.STRING),
            "STANDARD");
        ExpressionFunction e2=new
            ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
            new StatusAttribute("commercialsBob",
                FacplStatusType.BOOLEAN),
            false);
        ExpressionBooleanTree ebt = new
            ExpressionBooleanTree(ExprBooleanConnector.AND, e1, e2);
        addTarget(ebt);
        addObligation(
            new ObligationStatus(
                new FlagStatus(),
                Effect.PERMIT,
                ObligationType.M,
                new StatusAttribute("commercialsBob",
                    FacplStatusType.BOOLEAN), true)
              );
    }
```

La classe MainFACPL nel codice 5.24 incorpora PDP, PEPCheck e il PolicySet. Nel costruttore si inizializzano il PDP con un Combining Algorithm (in questo caso PermitUnlessDenyGreedy) e l'insieme di policy, il PEP con l'algoritmo di Enforcement (`DENY_BIASED`), Combining Algorithm (`DenyOverridesCheck`) e il PDP.

Listing 5.24: Policy Set

```
public class MainFACPL {

  private PDP pdp;
  private PEPCheck pep;

  public MainFACPL() throws Exception {
    // defined list of policies included in the PDP
    LinkedList<FacplPolicy> policies = new
        LinkedList<FacplPolicy>();

    policies.add(new PolicySet_Streaming());
    this.pdp = new
        PDP(it.unifi.facpl.lib.algorithm.PermitUnlessDenyGreedy.class,
        policies, false);

```

```
        this.pep = new PEPCheck(EnforcementAlgorithm.DENY_BIASED, new
            DenyOverridesCheck(), this.pdp);

14

        this.pep.addPEPActions(PEPAction.getPepActions());
16  }
```

Nel metodo `main` si aggiungono le richieste nella lista `requests`, queste poi verrano valutate nel ciclo proposto di seguito in 5.25.

Listing 5.25: Policy Set

```
1       // Initialise Authorisation System
        MainFACPL system = new MainFACPL();
3       // Result log
        StringBuffer result = new StringBuffer();
5       requests.add(ContextRequest_ListenStandardBob.getContextReq());//deny
        requests.add(ContextRequest_CommercialsBob.getContextReq());//commercials
7       requests.add(ContextRequest_ListenStandardBob.getContextReq());//permit
        requests.add(ContextRequest_ListenStandardBob.getContextReq());//permit

9
        for (ContextRequest_Status rcxt : requests) {
11        result.append(system.pep.doAuthorisation(rcxt));
        }
13      System.out.println(result.toString());
```

5.4.2 *Esempio accesso file*

In questa sezione si mostrano le parti di codice java più significative associate all'esempio descritto in 4.4.1, inoltre si presentano i miglioramenti a livello prestazionale attraverso l'uso delle *ObligationCheck* nel processo valutativo.

I codici per le richieste e per la costruzione dello status in questo caso non saranno trattati in quanto hanno la stessa struttura dei file precedenti e cambiano soltanto il nome degli attributi e il loro tipo.

Si presenta invece la policy sulle letture che utilizza l'obbligazione di tipo permanente. A differenza della delle regole sull'ascolto di Bob mostrato nel codice 5.22, l'*ObligationCheck* della lettura in 5.26 non ha nessuna scadenza, ma solo due *Expressions* in cui si indica il controllo sull' insieme dei file in cui è possibile fare la lettura e l'azione concessa(`read`). Con Questo tipo di struttura l'obbligaziona non raggiunge mai

la scadenza fintantoché gli attributi delle richieste passino la verifica delle due espressioni.

Listing 5.26: Policy di lettura

```
public PolicySet_Read() {
  addId("Read_Policy");
  // Algorithm Combining
  addCombiningAlg(it.unifi.facpl.lib.algorithm.DenyUnlessPermitGreedy.class);
  // Target
  ExpressionFunction file1 = new
      ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
      "thesis.tex",
       new AttributeName("file", "id"));
  ExpressionFunction file2 = new
      ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
      "facpl.pdf",
       new AttributeName("file", "id"));
  ExpressionBooleanTree setFiles = new
      ExpressionBooleanTree(ExprBooleanConnector.OR, file1,
      file2);
  // Expression ObligationCheck
  ExpressionFunction read = new
      ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
      "read",
       new AttributeName("action", "id"));

  addTarget(setFiles);
  // PolElements
  addPolicyElement(new Rule_read());
  // Obligation
  addObligation(
       new ObligationCheck(Effect.PERMIT,
            read,
            setFiles
            ));
```

Si mostrano adesso le differenze tra i tempi computazioni relativi a un insieme di policy che utilizza le *ObligationCheck* e uno che non le utilizza.

Nel codice 5.27 si presenza la policy duale di 5.26, in questo caso le regole non fanno uso del nuovo tipo di obbligazione.

Listing 5.27: Policy di lettura senza ObligationCheck

```
private class PolicySet_Read extends PolicySet {

  public PolicySet_Read() {
    addId("Read_Policy");
    // Algorithm Combining
    addCombiningAlg(it.unifi.facpl.lib.algorithm.DenyUnlessPermitGreedy.class);
    // Target
    ExpressionFunction file1 = new
        ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
        "thesis.tex",
         new AttributeName("file", "id"));
    ExpressionFunction file2 = new
        ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
        "facpl.pdf",
         new AttributeName("file", "id"));
    ExpressionBooleanTree setFiles = new
        ExpressionBooleanTree(ExprBooleanConnector.OR, file1,
        file2);
    // Expression ObligationCheck
    addTarget(setFiles);
    // PolElements
    addPolicyElement(new Rule_read());
    // No ObligationCheck
  }
```

Nella classe MainFACPL si utilizzano i due insiemi di regole per controllare due liste di richieste differenti. In 5.28 si mostra il caso in cui si utilizza il PolicySet senza ObligationCheck per la prima lista. Tutti i tempi di computazione vengono scritti nel file "testNoCheck.txt".

Listing 5.28: Main per i benchmarks

```
LinkedList<ContextRequest_Status> requests = new
    LinkedList<ContextRequest_Status>();
LinkedList<ContextRequest_Status> requests2 = new
    LinkedList<ContextRequest_Status>();

MainFACPL system = new MainFACPL(true);
MainFACPL systemNoCheck = new MainFACPL(false);
long start;
long end;
try {
  PrintWriter writer = new
```

```
        PrintWriter("/Users/mameli/Desktop/testNoCheck.txt",
        "UTF-8");
    for (int k=1;k<=100;k++){
      start=System.currentTimeMillis();
      for (ContextRequest_Status rcxt : requests) {
        result.append(systemNoCheck.pep.doAuthorisation(rcxt));
      }
      end=System.currentTimeMillis();
      writer.println(end-start);
    }
    writer.close();
```

Nella prima lista sono prensenti cento richieste di lettura, nella seconda il numero totale di richieste è lo stesso, ma ogni otto letture c'è una scrittura e una domanda di interruzione delle modifiche. Utilizzando il set di policy senza ObligationCheck il PDP controlla sempre ogni richiesta. Se invece si usa l'insieme di regole con la nuova obbligazione nella prima lista di richieste, dopo il primo controllo, il PDP non sarà più parte del processo di valutazione, nella seconda lista invece il PDP controllerà il 30% delle richieste. Ogni lista di richieste è stata verificata cento volte

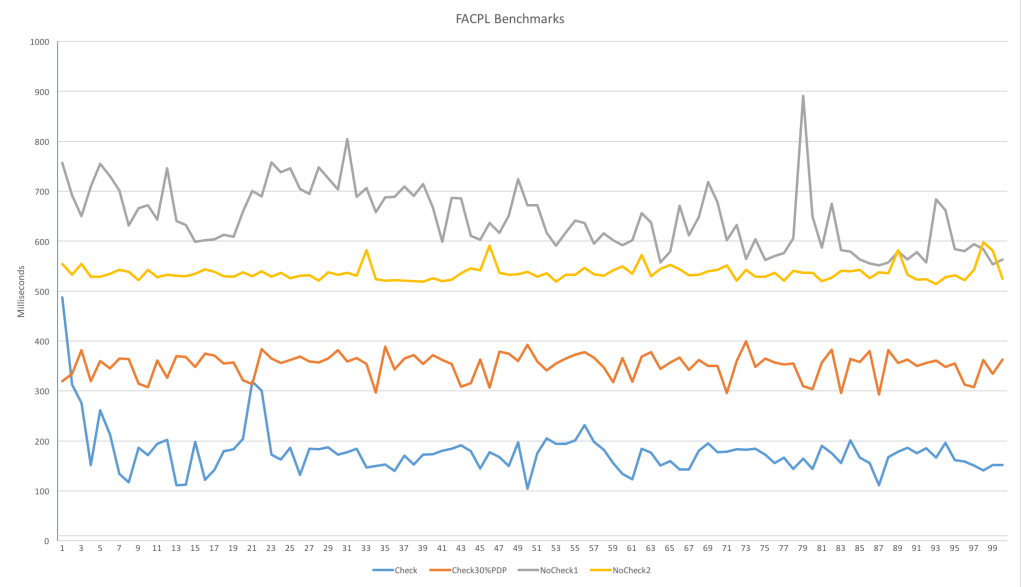

Figura 11: Computazione

per entrambi i tipi di Policy per un totale di diecimila valutazioni.

In figura 11 le unità in verticale indicano i millisecondi impiegati per controllare cento richieste. La linea blu e quella arancione indicano i tempi

per la computazione utilizzando il PolicySet con le ObligationCheck mentre le altre due indicano i tempi per l'insieme di regole che non utilizzano il nuovo paradigma. Analizzando i tempi totali per valutare diecimila richieste, in figura 12 si può vedere che per la prima lista (solo richieste di lettura) il sistema impiega tre volte meno tempo utilizzando le ObligationCheck, mentre per la seconda lista c'è un miglioramento del 35% circa.

Come si voleva provare, se le richieste di scrittura sono minori rispetto a quelle di lettura, la nuova gestione alleggerisce il processo valutativo.

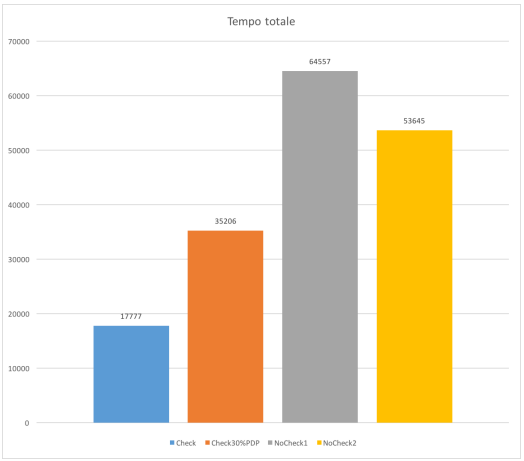

Figura 12: Tempo totale

6

CONCLUSIONI

In questa tesi è stata sviluppata un'estensione del linguaggio FACPL per migliorare la gestione delle richieste di accesso. Nella prima parte sono state descritte le strutture su cui si è basato lo sviluppo. Sono stati introdotti i primi modelli dell'Access Control partendo dal più semplice ACL arrivando fino a PBAC. In seguito è stato mostrato lo Usage Control descrivendo il nuovo modello UCON_{ABC} creato da Jaehong Park e Ravi Sandhu.

Successivamente è stato esaminato il linguaggio FACPL su cui si è basato implementazione del monitor a runtime. È stata mostrata la sintassi, la semantica e il processo di valutazione per presentare i lati positivi e soprattutto i limiti del linguaggio su cui si è lavorato per migliorare il controllo continuativo degli accessi.

Sono state aggiunte nuove produzioni alla sintassi ed è stata modificata la risposta dell'autorizzazione per poter essere utilizzata nel nuovo processo valutativo, in cui si elimina i controlli superflui per un determinato tipo di richieste. Alle modifiche apportate sulla sintassi sono seguite le implementazioni di nuove strutture nella libreria Java su cui si basa FACPL. Infine sono stati proposti esempi per presentare le operazioni possibili e i miglioramenti nell'uso delle risorse computative.

6.1 SVILUPPI FUTURI

In questo documento sono stati esposti due esempi, in uno si usa il numero di richieste, nell'altro il tempo. La librerie di FACPL è semplice da estendere ed è facile pensare a nuove possibili controlli su cui le autorizzazioni possono basarsi. Si può associare le verifiche della richiesta alla posizione geografica dell'utente, oppure alla congestione della rete aggiungendo la gestione di un nuovo parametro nella creazione delle obbligazioni e le rispettive funzioni che ne controllano il valore.

ACRONIMI

ACL Access Control List

RBAC Role Based Access Control

ABAC Attribute Based Access Control

PBAC Policy Based Access Control

UCON Usage Control

TM Trust Management

DRM Digital Rights Management

PAS Policy Authorization System

PR Policy Repository

PDP Policy Decision Point

PEP Policy Enforcement Point

OASIS Organization for the Advancement of Structured Information Standards

XACML eXtensible Access Control Markup Language

XML eXtensible Markup Language

FACPL Formal Access Control Policy Language

EBNF Extended Backus-Naur form

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 9.)

[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*