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

ELENCO DELLE FIGURE

1

INTRODUZIONE

I sistemi informatici, fin dalla loro nascita, sono stati utilizzati per la gestione di dati. Il tipo di informazioni gestite ha reso necessario la creazione 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 approccio: lo Usage Control.

I ricercatori dell'università di San Antonio, Jaehong Park e Ravi Sandhu, hanno descritto il modello Usage Control (UCON) [12] che consente di ottenere controlli continuativi di accesso e di basare le valutazioni di una richiesta sul comportamento passato.

In questa tesi si descrive un'estensione basata su UCON per Formal Access Control Policy Language (FACPL). FACPL è un linguaggio basato su eXtensible Access Control Markup Language (XACML) ed è supportato da una libreria Java. Questo viene utilizzato per lo sviluppo di sistemi basati sul modello per controllo degli accessi Policy Based Access Control.

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 che esegue sempre tutti i controlli. Non modificando mai i passi della valutazione, vari controlli per le richieste potrebbero risultare ridondati e in alcuni casi inutili. Le richieste che non modificano i dati o lo stato del sistema, come ad esempio le letture su file, passano per lo stesso processo di valutazione di tutte le altre richieste e il sistema dei controlli risulta eccessivamente complesso per l'autorizzazione di azioni semplici. Per questo motivo è stato ideato un monitor in grado di scegliere a runtime il processo di verifica delle richieste.

Nella prima parte dello sviluppo è stata modificata la sintassi del

linguaggio per adeguarla ad una nuova gestione delle richieste, in seguito è stata sviluppata un'estensione della libreria Java che supporta FACPL per modificare il processo valutativo così da garantire controlli di accesso continuativi. Nei capitoli successivi si mostrano i passi che hanno portato allo sviluppo della nuova gestione delle richieste in grado di rendere dinamico il processo di valutazione.

Il resto del documento è così strutturato:

- Nel Capitolo 2 si introducono vari modelli di Access Control, si descrive UCON e si mostrano due casi di studio basati sul controllo continuativo degli accessi.
- Nel Capitolo 3 si descrive la sintassi e la semantica di FACPL, il processo di valutazione e un esempio di utilizzo del linguaggio.
- Nel Capitolo 4 si mostra l'estensione di FACPL al fine di supportare controlli continativi di accesso.
- Nel Capitolo 5 si riportano le modifiche alla libreria Java che supporta FACPL per adattarlo al nuovo processo valutativo.
- Nel Capitolo 6 si riassume il lavoro svolto e si presentano alcuni possibili sviluppi futuri.

2

ACCESS CONTROL E USAGE CONTROL

L'esigenza di proteggere i dati ha determinato la necessità di creare strumenti per il controllo degli accessi così da eliminare, o almeno limitare, i rischi derivati dall'accesso non autorizzato alle informazioni.

Nel corso del tempo si sono sviluppati numerosi modelli per i sistemi di controllo degli accessi [11]. Nelle sezioni successive saranno presentati: Access Control List (ACL), Role Based Access Control (RBAC), Attribute Based Access Control (ABAC) e Policy Based Access Control (PBAC). Questi modelli sono introdotti in Sezione 2.1. In Sezione 2.2 è descritto lo UCON e il recente modello UCON_{ABC}, poi in Sezione 2.3 sono esposti due esempi che utilizzano UCON.

2.1 ACCESS CONTROL

Access Control List

ACL è stato creato agli inizi degli anni settanta per la necessità di aver un controllo degli accessi sui sistemi multiutente. Utilizza una lista di utenti con annesse le possibili azioni autorizzate [4]. 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 eccessivo numero di controlli necessari per un singolo accesso.

Role Based Access Control

RBAC è l'evoluzione di ACL. In questo modello vengono introdotti i *ruoli* [15]. Il ruolo è un titolo che definisce un livello di autorità. Più utenti possono avere lo stesso ruolo e quindi avere a disposizione tutte le 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. RBAC è limitato dalla staticità dei permessi e dal concetto di ruolo che non può essere associato a nessun attributo.

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

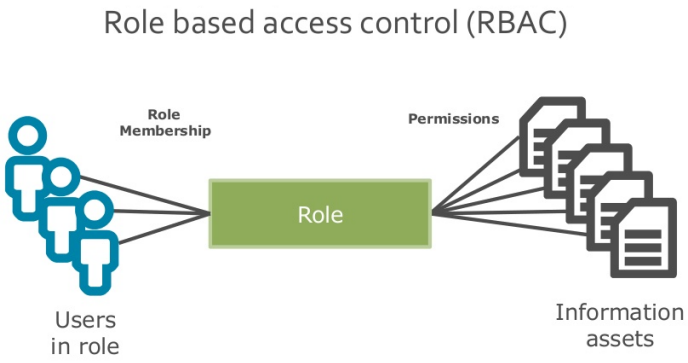

Figura 1: RBAC

Attribute Based Access Control

ABAC si basa sull'utilizzo di attributi associati all'utente, all'azione o al contesto della richiesta [5]. La valutazione di una autorizzazione diventa più specifica rispetto a RBAC, le regole sono più precise per ogni risorsa e inoltre il modello si basa su politiche invece che su permessi statici.

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.

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.

Policy Based Access Control

PBAC riorganizza il modello ABAC per semplificare la gestione delle regole [11]. È una standardizzazione di ABAC e si basa su *politiche* che

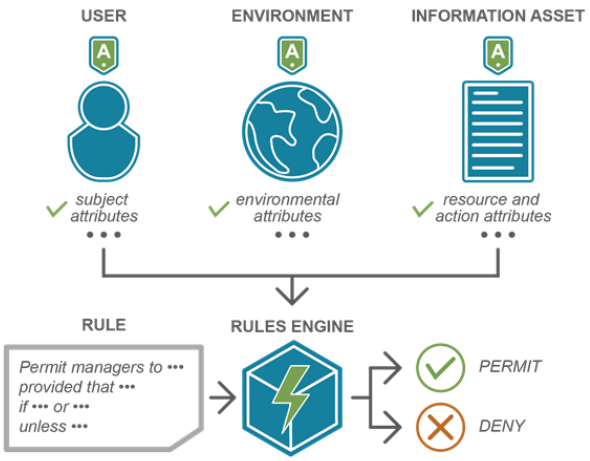

Figura 2: ABAC

non sono altro che insiemi di *rule*. PBAC combina gli attributi della risorsa, dell'ambiente e dell'utente, e li utilizza insieme alle *rule* per determinare l'autorizzazione. 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 però l'utilizzo di un linguaggio adatto allo scopo. Organization for the Advancement of Structured Information Standards (OASIS) ha creato il linguaggio 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 [13].

Il termine Usage Control UCON è stato ripreso da Jaehong Park e Ravi Sandhu per creare il modello UCON_{ABC} [12]. Questo è una generalizzazione dell'Access Control che include obbligazioni, condizioni sull'utilizzo, controlli continuativi e mutabilità [7]. Comprende e migliora i modelli di controllo di accesso tradizionali aggiungendo obblighi da parte dell'utente per accedere ad una risorsa, vincoli sulle azioni permesse, la gestione di attributi variabili nel tempo e la continuità nella

valutazione delle decisioni.

Il modello $UCON_{ABC}$ estende i controlli sull'accesso tradizionali ed è composto da otto componenti fondamentali. Queste sono subject, subject attribute, object, object attribute, right, authorization, obligation e condition.

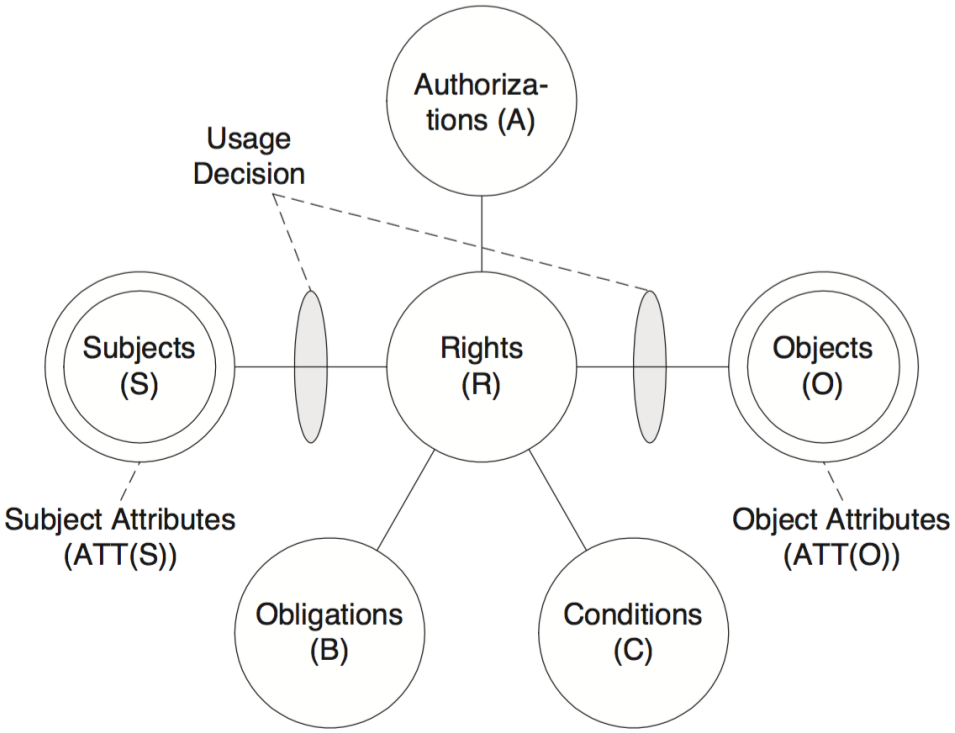

Figura 3: UCON

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

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

I *Right* sono i privilegi che i *Subject* hanno o esercitano sugli *Object*.

I tre fattori Authorization, oBligation e Condition (da cui prende anche il nome il modello $UCON_{ABC}$) sono predicati funzionali che devono essere valutati per le decisioni sull'uso. I tradizionali modelli di Access Control utilizzano solo le Authorization per il processo di decisione,

Obligation e Condition sono i nuovi componenti che entrano a far parte della valutazione [14].

Le *Authorization* 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 come mostrato in Figura 4 : Before usage (pre), Ongoing usage (on) e After usage [8].

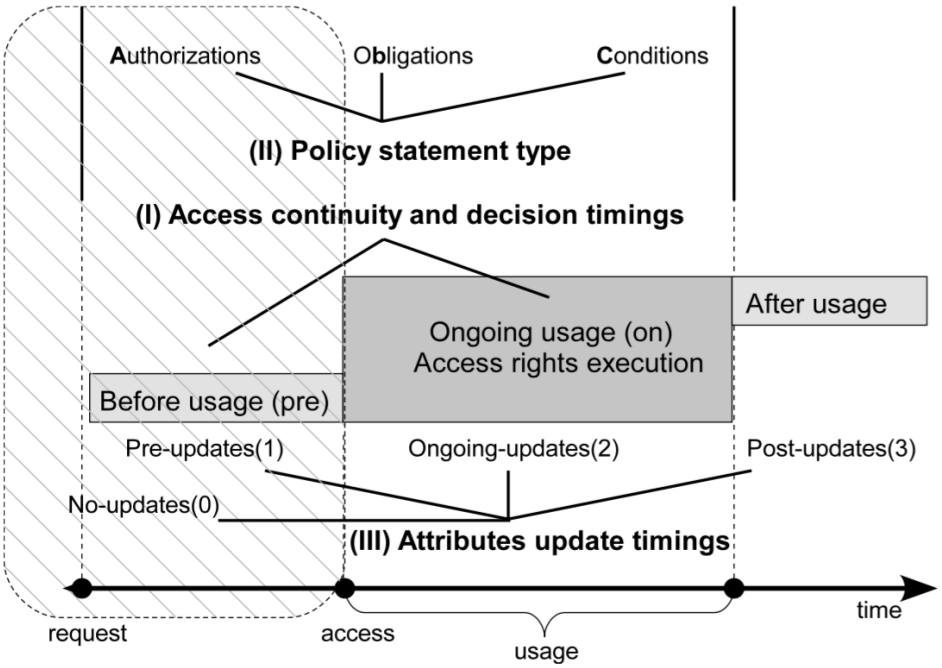

Figura 4: Fasi del processo di decisione

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 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.

2.3 ESEMPI DI USAGE CONTROL

Si propongono adesso due esempi 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 leggere o apportare modifiche nello stesso momento. Una volta finita la scrittura si potrà sbloccare i file e renderli di nuovo disponibili 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 esaurito il quale 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.

I due esempi non possono essere implementati con i modelli di Access Control tradizionali perché basano le valutazioni sul comportamento passato e richieste ripetute di uno stesso tipo sono valutate in modo differente.

3

FORMAL ACCESS CONTROL POLICY LANGUAGE

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. Il linguaggio FACPL (fakpol) è stato creato come alternativa a XACML.

FACPL è ispirato a 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 di FACPL in Sezione 3.1, poi si mostrano le componenti del sistema in Sezione 3.2 e il loro significato in Sezione 3.3. Infine, si mostra in Sezione 3.4 un esempio di 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 di accesso ad una risorsa 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 ad ogni risorsa siano associate una o più politiche, e che queste definiscano le credenziali necessarie per ottenere l'accesso. Il PR

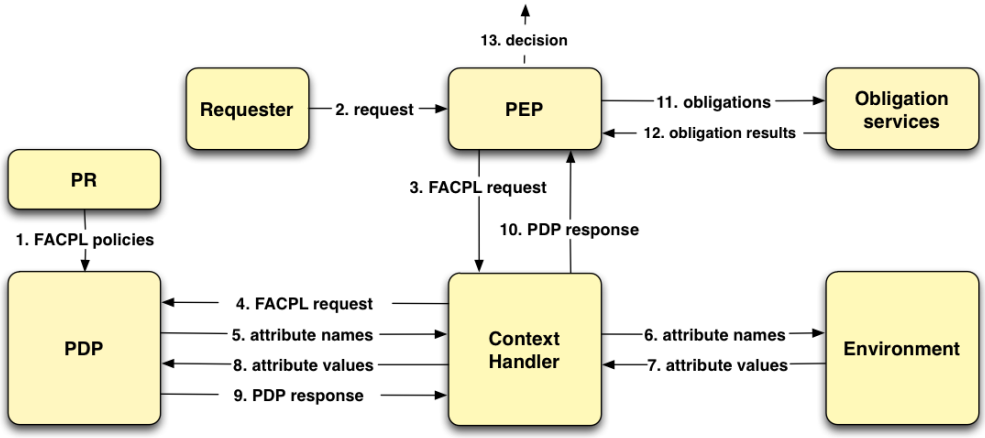

Figura 5: Processo di valutazione

contiene le Policy e le rende disponibili al PDP (Step 1), il quale decide se l'accesso viene autorizzato 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).

Il Context Handler aggiunge attributi dell'ambiente (come la data del sistema) 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 *obligation* 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 l'esecuzione delle *Obligation* 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 restituita dal processo.

3.2 SINTASSI

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.

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$_\delta$
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*)$^+$

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 è 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 della 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 *obligation* cioè le azioni opzionali o obbligatoriche da eseguire, da una sequenza di *Policy* e da un algoritmo per la combinazione dei risultati delle politiche da cui è composto l'insieme.

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

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*

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

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

Tabella 2: Sintassi ausiliaria per le risposte

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

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 sia *rule*, 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.
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.

La valutazione di una *policy* termina con il *fulfilment* di tutte le *obligation*. 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 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, indipendentemente 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 di *fulfilment* scelta, in seguito indicata con δ. 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 migliorano 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 la decisione del PDP sia *permit* e 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 DI POLITICA IN FACPL

Nel Codice 3.1 è riportato un semplice esempio di politica FACPL. Nel dettaglio la politica stabilisce che due utenti possono interagire con il sistema attraverso delle specifiche 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.

Codice 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". Si noti che la sintassi del PAS in Codice 3.1 utilizza delle parole chiave aggiuntive presenti nel tool di supporto. Qui di seguito, nel Codice 3.2, si presenta la struttura delle *request*.

Codice 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
  emph{obligation}: 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 dlla *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 XACML 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 Sezione 2.2, in cui si enfatizza l'uso di controlli continuativi e la gestione di contesti mutabili, non è ancora implementabile. Manca una struttura capace di memorizzare lo stato del sistema che possa essere utilizzata per la valutazione delle richieste e una componente in grado di supportare controlli continuativi di accesso.

Nel capitolo successivo si mostra la nuova struttura 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 del linguaggio è stata modificata di modo che si possa creare delle politiche che si basino 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. La sua tesi descrive l'implementazione della componente status per tenere traccia del comportamento passato e le strutture create per utilizzarla nel processo valutativo.

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 (Sezione 4.1) e l'inserimento di nuovi componenti (Sezione 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 di Figura 5. Come si vede in Figura 6, la nuova componente è Status.

Oltre agli *attribute name* il Context Handler ha anche la possibilità di ricavare gli *status attribute*. Il PDP quindi potrà richiedere entrambi i tipi di attributi per elaborare la sua risposta. La decisione poi passa 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.

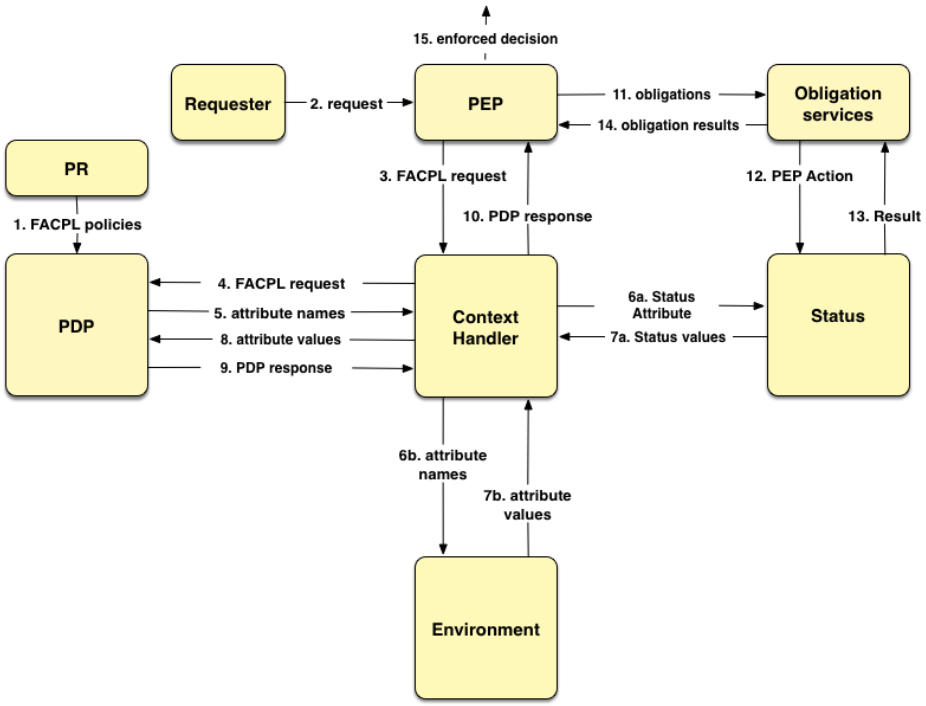

Figura 6: Processo di valutazione Status

Riprendendo il primo esempio di Sezione 2.3, si riporta il seguente requisito

> *Se uno degli amministratori sta scrivendo su un file, nessuno può leggere 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 Sezione 2.3, 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.

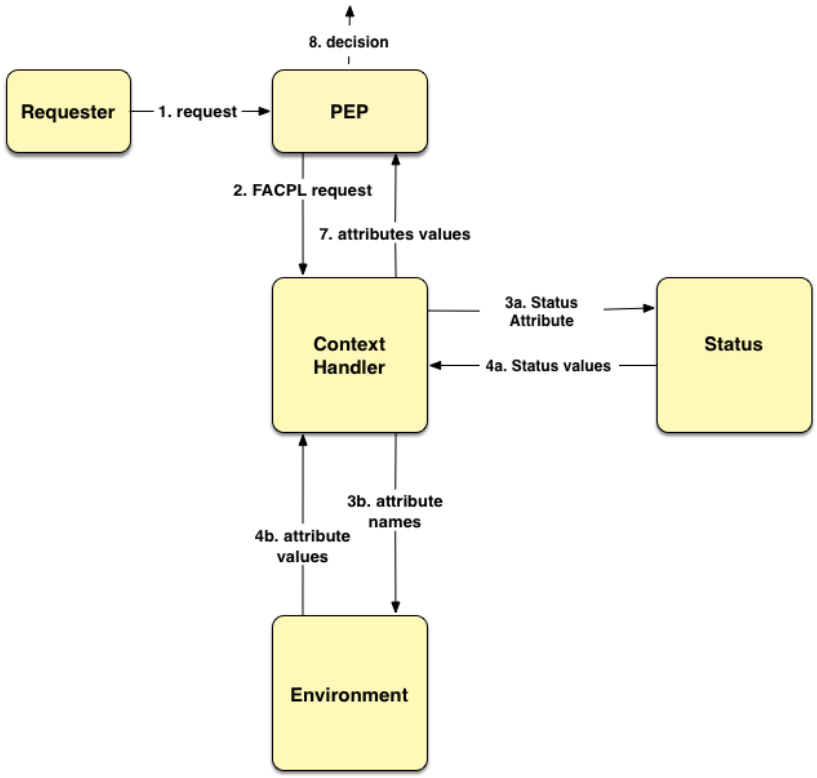

Figura 7: Processo di valutazione PEP-Check

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 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 solo alla prima domanda di accesso. Poi le richieste successive potranno essere gestite dal PEP con soltanto 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.

Si fa notare che il processo di valutazione ridotto di Figura 7 non

sostituisce il processo mostrato in Figura 6, ma il suo utilizzo porta notevoli vantaggi prestazionali e elimina i controlli ridondanti dalle verifiche per le autorizzazioni delle richieste.

4.2 SINTASSI

La modifica nella struttura ha reso necessario apportare dei cambiamenti anche nella grammatica. La nuova sintassi è riportata in Tabella 4.

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:

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

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 Name* è stata aggiunta la produzione *Status/Identifier* per identificare i termini dello Status nelle *Expression*.

La nuova produzione per le *Obligation* della forma:

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

è utilizzata per la nuova gestione delle richieste da parte del PEP ed è denominata *Obligation Check*. 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* affinché il PEP possa fare le proprie verifiche e può esserci un valore opzionale che indica il tipo della scadenza.

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

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

indicherà la scadenza delle *Obligation Check*. Un intero (Int) determina il numero massimo di richieste, mentre una data (Date) il limite temporale. Se *expiration* viene omesso la *obligation* non ha scadenza.

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*]

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 Obligation*.

La nuova forma indica le *Fulfilled Obligation* 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 Obligation* 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 effettuare una lettura solo se nessuno sta scrivendo (come specificato in linea 5 nel Codice 4.1) e vuole utilizzare la risorsa *contabilita.xlsx* (linea 2). Se la richiesta viene accettata si deve cambiare l'attributo *isReading* a `true` (linea 7), 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.

Codice 4.1: Esempio per la sintassi

```
1  Policy readPolicy < permit-overrides
     target: equal("Charlie",name/id) &&  equal("read", action/id) &&
       equal("contabilita.xlsx", resource/id)
3    rules:
      Rule accessReadRule (
5       permit  target:  equal(status/isWriting, false))
     obl:
7     [ permit M flag(status/isReading, true)],
      [ permit ( equal("read", action/id)) , (
          equal("contabilita.xlsx", resource/id))]
9  >
```

Tabella 4: Sintassi di FACPL$_{PB}$

Policy Authorisation Systems	*PAS* ::=	(pep : *EnfAlg* pdp : *PDP* status : [*Attribute*])
Attribute	*Attribute* ::=	(*Type Identifier* (= Value)$^?$)
Type	*Type* ::=	int | boolean | date | double
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$_\delta$
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) | mul(*Attribute*, int) | div(*Attribute*, float) | sub(*Attribute*, int) | sub(*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*
Literal Values	*Value* ::=	true | false | *Double* | *String* | *Date*
Requests	*Request* ::=	(*Name*, *Value*)$^+$

Codice 4.2: Esempio PAS

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

Alla linea 7 di Codice 4.2 si inizializzano i valori dello status *isWriting* e *isReading* a `false`. Si fa notare inoltre che con questa porzione di codice:

```
    [ 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.

4.3 SEMANTICA

La modifica della sintassi ha determinato anche variazioni nella semantica descritta in Sezione 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:

```
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:

```
[ 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 risorsa 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 DI USAGE CONTROL

In questa sezione si mostrano gli esempio presentati in Sezione 2.3 utilizzando le nuove funzionalità.

4.4.1 *Accessi in lettura e scrittura di file*

Nel Codice 4.3 si mostra la policy relativa all'esempio 2.3. Nel sistema ci sono due utenti: Alice e Bob. Alice fa parte del gruppo degli Administrator, mentre Bob appartine ad un gruppo di default non specificato. Solo gli amministratori possono scrivere, però tutti possono leggere file appartenenti ad un insieme precedentemente specificato come si mostra in linea 28.

Se l'Administrator Alice fa una richiesta di scrittura, ha l'obbligo di cambiare il booleano *isWriting* in `true` (linea 14). Se finisce di scrivere

invece deve fare una richiesta di *stopWrite* e assegnare `false` all'attributo (linea 25). Ovviamente se non sta scrivendo non può richiedere la fine dell'azione quindi *stopWrite* è accettabile solo quando isWriting è `true` (linea 21).

Codice 4.3: 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 / isWriting , false )
       )
    obl:
   [  permit M  flagStatus(isWriting, true) ]
 }
   PolicySet StopWrite_Policy {  deny-unless-permit
    target:  equal ("stopWrite", action/id)
    policies:
      Rule stopWrite (  permit  target:
        equal("thesis.tex", file/id) &&
        equal ( status / isWriting , true ) &&
        equal("Administrator", group/id)
       )
    obl:
   [  permit M  flagStatus(isWriting, false) ]
 }
```

Tutti gli utenti possono leggere i file solo se nessuno sta scrivendo su uno di questi (linea 32) e se i file appartengono ad un determinato insieme (linea 28). In *Read_Policy* si specifica che i file debbano essere "thesis.tex" oppure "facpl.pdf", inoltre viene utilizzato il nuovo tipo di obbligation alla linea 35. Il PAS è differente dalla versione usata in Sezione 3.4 solo nella definizione dello status in cui si inizializza isWriting a `false` (linea 45).

```
   PolicySet Read_Policy {  deny-unless-permit
28   target: in ( file/id , set ("thesis.tex" , "facpl.pdf"))
     policies:
30     Rule read (  permit  target:
        equal ("read", action/id) &&
32      equal ( status / isWriting , false )
       )
34   obl:
    [  permit M  equal ("read",action/id) , in ( file/id , set
       ("thesis.tex" , "facpl.pdf")) ]
36 }
}
38 PAS {
  Combined Decision : false ;
40 Extended Indeterminate : false ;
  Java Package : "exampleReadWrite" ;
42 Requests To Evaluate : Request1, Request2, Request3, Request4,
     Request5, Request6 , Request7, Request8;
  pep:  deny-biased
44 pdp:  deny-unless-permit
  status: [(boolean isWriting = false) ]
46 include exampleReadWrite
}
```

Di seguito, nel Codice 4.4 e 4.5, si mostrano otto richieste per descrivere tutti i possibili scenari insieme ai risultati di valutazione.

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 accettatte 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" (linea 31 Codice 4.3), il file sia nell'insieme {"thesis.tex","facpl.pdf"} (linea 28) e che il Boolean isWriting sia falso (linea 32). 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 sono espressi nella linea 35 della policy. Si fa notare la differenza dei tempi di valutazione tra la *Request1* e le altre due.

Codice 4.4: 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 N: 1
REQUEST: read_request Bob
 Decision=
 PERMIT
time 53ms
-------------------------------------------------
REQUEST N: 2
REQUEST: read_request Bob
 Decision=
 PERMIT
time 6ms
-------------------------------------------------
REQUEST N: 3
REQUEST: read_request Alice
 Decision=
 PERMIT
time 9ms
```

La *Request4* 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. La *Request8* è una

richiesta di lettura, ed essendo successiva ad un'altra read (*Request7*), si adotterà nuovamente il processo di valutazione ridotto.

Codice 4.5: 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 , "Alice")
  (group / id , "Administrator")
  (action / id, "stopWrite")
  (file / id, "thesis.tex")
}
 Request:{ Request7
  (name / id , "Bob")
  (action / id, "read")
  (file / id, "thesis.tex")
}
 Request:{ Request8
  (name / id , "Alice")
  (action / id, "read")
  (file / id, "facpl.pdf")
}
```

```
  ------------------------------------------------
  REQUEST N: 4
  REQUEST: write_request Alice
   Decision=
   PERMIT
  time 47ms
  ------------------------------------------------
  REQUEST N: 5
  REQUEST: read_request Bob
```

```
 Decision=
 DENY
time 25ms
-------------------------------------------------
REQUEST N: 6
REQUEST: stop_write_request Alice
 Decision=
 PERMIT
time 44ms
-------------------------------------------------
REQUEST N: 7
REQUEST: read_request Bob
 Decision=
 PERMIT
time 32ms
-------------------------------------------------
REQUEST N: 8
REQUEST: read_request Alice
 Decision=
 PERMIT
time 3ms
```

In Tabella 5 si riassumono i risultati della valutazione, il tipo di processo valutativo (PDP+PEP per quello completo e PEP per quello ridotto) e i tempi di valutazione.

Tabella 5: Risultati della valutazione

	Risultato	**PV**	**Tempo**
Richiesta 1	*PERMIT*	*PDP+PEP*	*53ms*
Richiesta 2	*PERMIT*	*PEP*	*6ms*
Richiesta 3	*PERMIT*	*PEP*	*9ms*
Richiesta 4	*PERMIT*	*PDP+PEP*	*47ms*
Richiesta 5	*DENY*	*PDP+PEP*	*25ms*
Richiesta 6	*PERMIT*	*PDP+PEP*	*44ms*
Richiesta 7	*PERMIT*	*PDP+PEP*	*32ms*
Richiesta 8	*PERMIT*	*PEP*	*3ms*

4.4.2 *Servizio di streaming*

L'esempio seguente è simile al primo, ma in questo caso si fa uso anche delle *Obligation 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, ha lo stesso vincolo di login di Alice, ma 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. Per implementare la "listen" senza limiti di Alice e quella limitata dal tempo di Bob si usano Obligation Check differenti.

Nel Codice 4.6 si mostrano le policy per il "login" e per le "listen" degli utenti Premium. Nella policy di login si controlla che la password della richiesta sia quella salvata nel sistema (linea 8) e se è giusta si assegna all'attributo *loginAlice* "PREMIUM" e a *streamingAlice* `true` (linee 11-12). Gli ascolti sono associati ad una obbligazione persistente (linea 22). Gli utenti possono quindi richiedere i brani senza essere limitati per tutta la durata del login. La prima richiesta di "listen" di Alice è valutata dal PDP e dal PEP eseguendo i controlli riportati nelle linee 16-19. Per le richieste successive, valutate solo dal PEP, si controlla invece l'obbligazione in linea 22.

Codice 4.6: Policy Login e Listen Premium

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

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

```
        equal ( status / loginAlice, "PREMIUM")
20      )
    obl:
22  [  permit M  equal(name/id, "Alice"),
       equal(status/streamingAlice, true)]
}
```

Il login per gli utenti standard è simile a quello dei premium e cambia soltanto nell'assegnamento delle variabili (linee 30-31). Gli ascolti invece sono associati ad una obbligazione limitata dal tempo. Gli utenti standard possono ascoltare le canzoni per quindici minuti di fila, come specificato nell'Obligation in linea 43, ma una volta esaurito il tempo devono ascoltare la pubblicità. Alla prima richiesta di "listen" si assegna al booleano associato al vincolo sulla pubblicità *commercialsBob* `true` affinché, una volta esaurito il tempo, l'utente Bob debba fare una richiesta di "listenCommercials" per richiedere nuovamente un'ascolto. Anche in questo caso le richieste d'ascolto sono gestite interamente dal PEP a partire dalla seconda richiesta consecutiva.

Codice 4.7: Policy Login, Listen e Commercials Standard

```
PolicySet LoginSTANDARD_Policy {  deny-unless-permit
24   target: equal ( name / id, "Bob") &&
       equal ( action/id, "login")
     policies:
26     Rule loginBob(  permit  target:
         equal ( status / passwordBob, password/id)
28     )
     obl:
30  [  permit M  SetString(loginBob, "STANDARD") ],
    [  permit M  flagStatus(streamingBob, true) ]
32 }

34  PolicySet ListenSTANDARD_Policy {  deny-unless-permit
     target: equal ( name / id, "Bob") &&
       equal ( action/id, "listen")
36   policies:
      Rule listenBob(  permit  target:
38      equal ( status / loginBob, "STANDARD") &&
        equal ( status / commercialsBob, false)
40     )
     obl:
42   [  permit M  equal(name/id, "Bob"),
       equal(status/streamingBob, true),"00:15:00"],
```

```
    [  permit M  flagStatus(commercialsBob, true) ]
 }

  PolicySet commercials_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 : "exampleStreaming" ;
 Requests To Evaluate : Request1, Request2, Request3, Request4,
    Request5, Request6 , Request7, Request8, Request9, Request10
 pep:  deny-biased
 pdp:  deny-unless-permit
 status: [ (string loginBob = "NOLOGIN"),
        (string loginAlice = "NOLOGIN"),
        (string passwordBob = "abcdef"),
        (string passwordAlice = "123456"),
        (boolean streamingBob = false),
        (boolean streamingAlice = false),
        (boolean commercialsBob = false) ]
 include exampleStreaming
}
```

Le prime due *request* non sono accettate perché non è possibile richiedere gli ascolti prima di fare un login.

Codice 4.8: Richieste di ascolto prima del login

```
Request:{ Request1
 (name / id , "Alice")
 (action / id, "listen")
}
Request:{ Request2
 (name / id , "Bob")
```

```
  (action / id, "listen")
}
```

```
  ------------------------------------------------
  REQUEST N: 1
  REQUEST: listen alice
   Decision=
   DENY
  time 55ms
  ------------------------------------------------
  REQUEST N: 2
  REQUEST: listen bob
   Decision=
   DENY
  time 39ms
```

Dopo *Request3* e *Request4*, dove si usa un attributo password per validare l'accesso, i due utenti possono ascoltare i brani.

Codice 4.9: Richieste di login

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

```
  ------------------------------------------------
  REQUEST N: 3
  REQUEST: login alice
   Decision=
   PERMIT
  time 24ms
  ------------------------------------------------
  REQUEST N: 4
  REQUEST: login bob
```

```
 Decision=
 PERMIT
time 38ms
```

Dalla *Request5* alla *Request8* gli utenti richiedono degli ascolti e sono tutti autorizzati. Si fa notare anche in questo esempio la differenza tra i tempi di valutazione delle varie richieste. La *Request6* e la *Request8* impiegano meno tempo perché viene adottato il processo valutativo ridotto.

Codice 4.10: Richieste di listen

```
 Request:{ Request5
  (name / id , "Alice")
  (action / id, "listen")
}
 Request:{ Request6
  (name / id , "Alice")
  (action / id, "listen")
}
 Request:{ Request7
  (name / id , "Bob")
  (action / id, "listen")
}
 Request:{ Request8
  (name / id , "Bob")
  (action / id, "listen")
}
```

```
--------------------------------------------------
REQUEST N: 5
REQUEST: listen alice
 Decision=
 PERMIT
time 34ms
--------------------------------------------------
REQUEST N: 6
REQUEST: listen alice
 Decision=
 PERMIT
time 8ms
--------------------------------------------------
REQUEST N: 7
```

```
REQUEST: listen bob
 Decision=
 PERMIT
time 60ms
-------------------------------------------------
REQUEST N: 8
REQUEST: listen bob
 Decision=
 PERMIT
time 7ms
```

Si suppone che dopo tre richieste di ascolto, Bob alla *Request9* abbia superato il limite di tempo di quindici minuti. Tutte le sue possibili richieste di ascolto sono rifiutate finché non richiede di sentire la pubblicità.

Codice 4.11: Richieste di listen Bob

```
Request:{ Request9
  (name / id , "Bob")
  (action / id, "listen")
}
Request:{ Request10
  (name / id , "Bob")
  (action / id, "listen")
}
```

```
-------------------------------------------------
REQUEST N: 9
REQUEST: listen bob
 Decision=
 DENY
time 29ms
-------------------------------------------------
REQUEST N: 10
REQUEST: listen bob
 Decision=
 DENY
time 27ms
```

Dopo una richiesta di "listenCommercials" Bob ha di nuovo altri quindici minuti di ascolto possibili.

Codice 4.12: Richiesta di ascolto della pubblicità di Bob

```
Request:{ Request11
  (name / id , "Bob")
  (action / id, "listenCommercial")
}
Request:{ Request12
  (name / id , "Bob")
  (action / id, "listen")}
```

```
REQUEST N: 11
REQUEST: pubblicità bob
 Decision=
 PERMIT
time 33ms
--------------------------------------------------
REQUEST N: 12
REQUEST: listen bob
 Decision=
 PERMIT
time 36ms
```

Anche in questo esempio si propone una tabella in cui si riassumono i risultati della valutazione.

Tabella 6: Risultati della valutazione

	Risultato	**PV**	**Tempo**
Richiesta 1	*DENY*	*PDP+PEP*	*55ms*
Richiesta 2	*DENY*	*PDP+PEP*	*39ms*
Richiesta 3	*PERMIT*	*PDP+PEP*	*24ms*
Richiesta 4	*PERMIT*	*PDP+PEP*	*38ms*
Richiesta 5	*PERMIT*	*PDP+PEP*	*34ms*
Richiesta 6	*PERMIT*	*PEP*	*8ms*
Richiesta 7	*PERMIT*	*PDP+PEP*	*60ms*
Richiesta 8	*PERMIT*	*PEP*	*7ms*
Richiesta 9	*DENY*	*PDP+PEP*	*29ms*
Richiesta 10	*DENY*	*PDP+PEP*	*27ms*
Richiesta 11	*PERMIT*	*PDP+PEP*	*33ms*
Richiesta 12	*PERMIT*	*PDP+PEP*	*36ms*

In questi due esempi si è mostrato come sia flessibile il processo di valutazione. Il sistema riesce a gestire a runtime il controllo degli accessi e adatta il processo di verifica basandosi sul tipo di richieste. Inoltre sono evidenti i miglioramenti a livello prestazionale. Richieste ripetute sono gestite in modo più efficiente e se valutate solo dal PEP la risposta impiega meno della metà del tempo per essere restituita.

5

ESTENSIONE DELLA LIBRERIA FACPL

La librearia di supporto a FACPL è basata su Java. Per implementare le nuove funzioni sono state aggiunte varie classi e sono state modificate alcune parti già esistenti 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`

nel branch Usage Control. Lo sviluppo dell'estensione si può dividere in due passi principali. La prima parte (Serzione 5.1) consiste nel creare una classe PEP che può gestire le richieste e adattare il processo di valutazione. La seconda parte (Sezioni 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 viene mostrato come è stato esteso il plugin di FACPL per supportare la nuova estensione. Nella Sezione 5.5 si descrivono le classi Java associate ai file FACPL dell'esempio in Sezione 4.4.2. Infine, nella Sezione 5.6 si mostrano i miglioramenti prestazionali utilizzando l'esempio sviluppato in Sezione 4.4.1.

5.1 CLASSE PEPCHECK

Il normale flusso del processo valutativo passa prima dal PEP. Questo riceve la richiesta e poi la manda al PDP affinché 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 avviene invece 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'altro 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. Nella figura 8 si mostra il grafico Unified Modeling Language (UML) della classe PEPCheck e i relativi elementi associati.

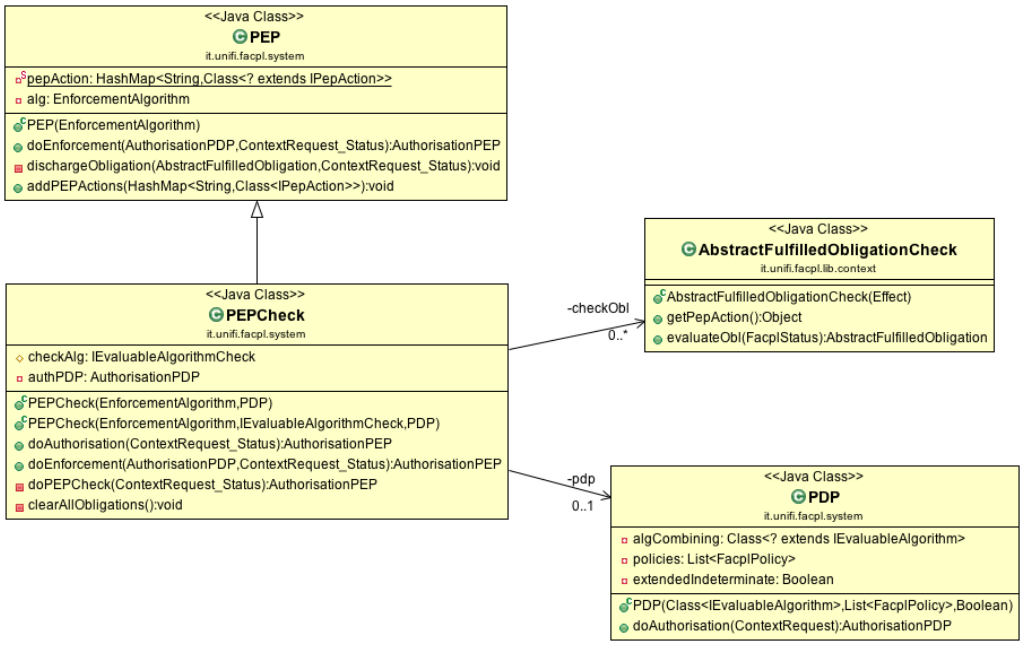

Figura 8: Diagramma PEPCheck

Nel Codice 5.1 si mostrano i campi della classe PEPCheck e il costruttore.

Codice 5.1: Costruttore PEPCheck

```
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>();
```

```
10      checkAlg = combiningAlgorithm;
        this.pdp = pdp;
12      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.

```
14      if (checkObl.size() == 0) {
          /*
16         * PDP Evaluation -> PEP Enforcement
           */
18        authPDP = pdp.doAuthorisation(cxtReq);
          return this.doEnforcement(authPDP, cxtReq);
20      } else {
```

Nella lista c'è almeno una *obligation check*
Il sistema valuta le *Expression* associate alle obbligazioni con il metodo *doPEPCheck* (vedi 3.1 per la struttura). Nel caso in cui la decisione sia `PERMIT` oppure `DENY`, il risultato viene subito restituito senza i controlli del PDP.

```
          /*
22         * PEP Evaluation
           */
24        // Evaluating check obligation
          result = this.doPEPCheck(cxtReq);
26        StandardDecision dec = result.getDecision();
          l.debug("CHECK RESULT: " + dec);
28        if (dec == StandardDecision.PERMIT || dec ==
              StandardDecision.DENY) {
            return result;
30        } else {
```

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 *doAuthorisation* sul PDP e *doEnforcement*.

```
28        if (dec == StandardDecision.PERMIT || dec ==
              StandardDecision.DENY) {
            return result;
30        } else {
            /*
32           * if check obligation returns an error
             * -> PDP Evaluation -> PEP Enforcement
34           */
            l.debug("BACK TO PDP");
36          authPDP = pdp.doAuthorisation(cxtReq);
            this.clearAllObligations();
38          return this.doEnforcement(authPDP, cxtReq);
          }
40      }
```

Il *doEnforcement* nel codice 5.1 del PEPCheck richiama il metodo implementato in PEP nella linea `super.doEnforcement(authPDP, cxtReq);` come si mostra di seguito alla linea 48.

```
    @Override
42  public AuthorisationPEP doEnforcement(AuthorisationPDP authPDP,
        ContextRequest_Status cxtReq) {
      /*
44     * normal PEP enforcement + addiction of CheckObligation
       */
46    AuthorisationPEP first_enforcement;
      Logger l = LoggerFactory.getLogger(PEPCheck.class);
48    first_enforcement = super.doEnforcement(authPDP, cxtReq); //
          enforcement
      StandardDecision dec = first_enforcement.getDecision();
50    l.debug("FIRST ENFORCEMENT COMPLETED, DECISION: " + dec);
      if (dec != StandardDecision.PERMIT) {
52      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 *FulfilledObligation* e nel caso in

cui queste siano FulfilledObligationNumCheck oppure FulfilledObligationTimeCheck, sono aggiunte alla lista di AbstractFulfilledObligationNumCheck di PEPCheck (linee 58-78). Solo dopo l'inserimento delle obbligazioni si restituisce la decisione ricavata (linea 81).

Codice 5.2: Aggiunta delle FulfilledObligationNumCheck

```
54    l.debug("ADDING CHECK OBLIGATION TO PEP...");
      Iterator<AbstractFulfilledObligation> iterator =
          authPDP.getObligationIterator();
56    while (iterator.hasNext()) {
        AbstractFulfilledObligation o = iterator.next();
58      if (o instanceof FulfilledObligationCheck) {
          FulfilledObligationCheck temp = null;
60        try {
            temp = (FulfilledObligationCheck)
                ((FulfilledObligationCheck) o).clone();
62        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
64        }
          if (!checkObl.contains(temp)) {
66          l.debug("ADDED: " + temp);
            checkObl.add(temp);
68        }
        }else if (o instanceof FulfilledObligationTimeCheck) {
70        FulfilledObligationTimeCheck temp = null;
          try {
72          temp = (FulfilledObligationTimeCheck)
                ((FulfilledObligationTimeCheck) o).clone();
          } catch (CloneNotSupportedException e) {
74          e.printStackTrace();
          }
76        if (!checkObl.contains(temp)) {
            l.debug("ADDED: " + temp);
78          checkObl.add(temp);
          }}}
80    l.debug("...CHECK OBLIGATION ADDED");
      return first_enforcement;
82  }
```

La lista modificata nel Codice 5.2 è utilizzata nel metodo *doPEPCheck* (nel Codice 5.3) 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. Nelle linee 90 e 98 si ricava il risultato delle singole obbligazioni con il metodo *getObligationResult* e si inserisce in lista *decisionList*, alla linea 106 invece si utilizza l'algoritmo di combinazione delle decisioni per valutare l'intera lista dei risultati.

Codice 5.3: Valutazione FulfilledObligationCheck

```
  l.debug("DOING PEP CHECK:");
84  AuthorisationPEP r = new AuthorisationPEP();
  StandardDecision dec;
86  LinkedList<StandardDecision> decisionList = new
      LinkedList<StandardDecision>();
  for (AbstractFulfilledObligationCheck obl : checkObl) {
88    if (obl instanceof FulfilledObligationCheck){
        dec =
90      ((FulfilledObligationCheck)obl).getObligationResult(ctxRequest);
        decisionList.add(dec);
92      if (StandardDecision.NOT_APPLICABLE.equals(dec)) {
          r.setDecision(StandardDecision.NOT_APPLICABLE);
94        return r;
        }
96    }else if (obl instanceof FulfilledObligationTimeCheck){
        dec =
98      ((FulfilledObligationTimeCheck)obl).getObligationResult(ctxRequest);
        decisionList.add(dec);
100     if (StandardDecision.NOT_APPLICABLE.equals(dec)) {
          r.setDecision(StandardDecision.NOT_APPLICABLE);
102       return r;
        }
104   }
  }
106 r = checkAlg.evaluate(decisionList, ctxRequest);
  checkAlg.resetAlg();
108 return r;
```

Nella Sezione successiva 5.2 si descrivono le Obligation utilizzate nel metodo *doPEPCheck* Codice 5.3.

5.2 CLASSE OBLIGATION 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*. In Figura 9 si mostra il diagramma UML della classe ObligationCheck. Nel Codice 5.4 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.

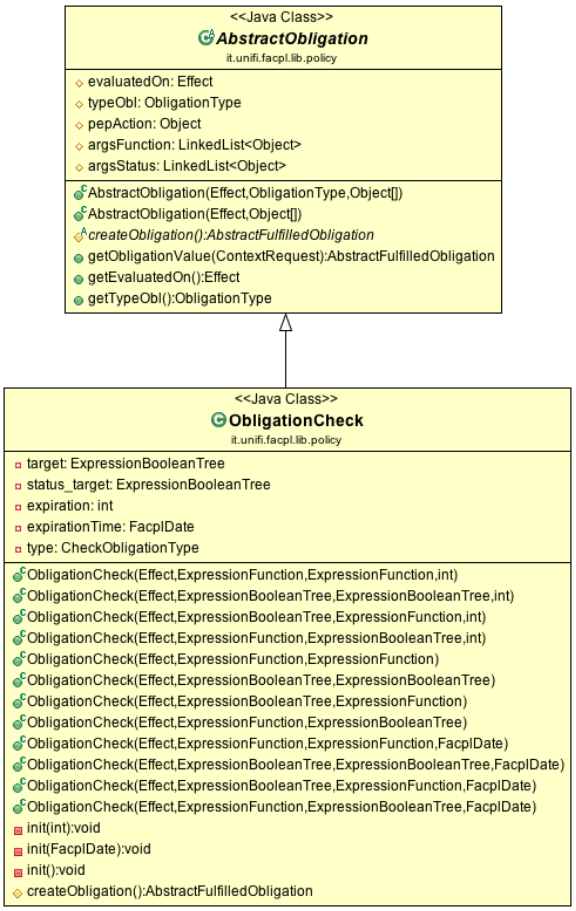

Figura 9: Diagramma ObligationCheck

Codice 5.4: Costruttore e campi

```
public class ObligationCheck extends AbstractObligation {
2   private ExpressionBooleanTree target;
    private ExpressionBooleanTree status_target;
```

```
4   private int expiration;
    private FacplDate expirationTime;
6   private CheckObligationType type;
    /*
8    * four constructor for all combinations of Expression:
     * 1: ExpressionFunction, ExpressionFunction
10   * 2: ExpressionBooleanTree, ExpressionBooleanTree
     * 3: ExpressionBooleanTree, ExpressionFunction
12   * 4: ExpressionFunction, ExpresisonBooleanTree
     */
14  public ObligationCheck(Effect evaluatedOn, ExpressionFunction
        target,
         ExpressionFunction status_target, int expiration) {
16    super(evaluatedOn);
      this.target = new ExpressionBooleanTree(target);
18    this.status_target = new ExpressionBooleanTree(status_target);
      this.init(expiration);
20  }
```

La scadenza, come già detto in Sezione 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 Codice 5.5.

Codice 5.5: Enum delle Obligation

```
package it.unifi.facpl.lib.enums;
2
public enum CheckObligationType {
4   P, N ,T
}
```

Per ogni obbligazione ci sono quattro costruttori in modo tale da poter creare l'oggetto per ogni combinazione di *Expressions* in input. In Sezione 5.6 si mostra un costruttore 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`).

Codice 5.6: Costruttore Obligation Temporale

```
    super(evaluatedOn);
2   this.target = target;
    this.status_target = status_target;
```

```
4       this.init(expiration);
    }
```

Il metodo *init* nel Codice 5.7, presente in tutti i costruttori, inizializza i campi della classe a seconda del tipo di scadenza data in input. Se la scadenza non è presente si utilizza il metodo init per creare le obbligazioni di tipo Persistente.

Codice 5.7: Inizializzazione dei campi della classe

```
    private void init(int expiration) {
8       this.expiration = expiration;
        this.pepAction = "CHECK";
10      this.type = CheckObligationType.N;
    }
12  private void init(FacplDate expiration) {
        this.expirationTime = expiration;
14      this.pepAction = "CHECK";
        this.type = CheckObligationType.T;
16  }
    private void init() {
18      this.pepAction = "CHECK";
        this.type = CheckObligationType.P;
20  }
```

In *createObligation* nel Codice 5.8 si utilizzano i campi assegnati dai costruttori e da init per la creazione delle *FulfilledObligationNumCheck* che sono descritte più approfonditamente in Sezione 5.3.

Codice 5.8: Creazione di FulfilledObligationNumCheck

```
         * this method create a fulfilledobligationcheck
22       */
        if (this.type == CheckObligationType.N) {
24          return new FulfilledObligationCheck(this.evaluatedOn,
                this.target,
                 this.status_target,
26               this.expiration);
        } else if (this.type == CheckObligationType.T){
28          return new FulfilledObligationTimeCheck(this.evaluatedOn,
                this.target,
                 this.status_target,
30               this.expirationTime);
        } else{
```

```
32            return new
                  FulfilledObligationCheckPersistent(this.evaluatedOn,
              this.target,
34            this.status_target);
      }
36  }
```

5.3 CLASSE 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 classi differenti:

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. Quando si crea una Fulfilled Obligation con una scadenza determinata da un numero intero, si utilizza la *FulfilledObligationNumCheck*. Nel Codice 5.9 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.

Codice 5.9: FulfilledObligationNumCheck N

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

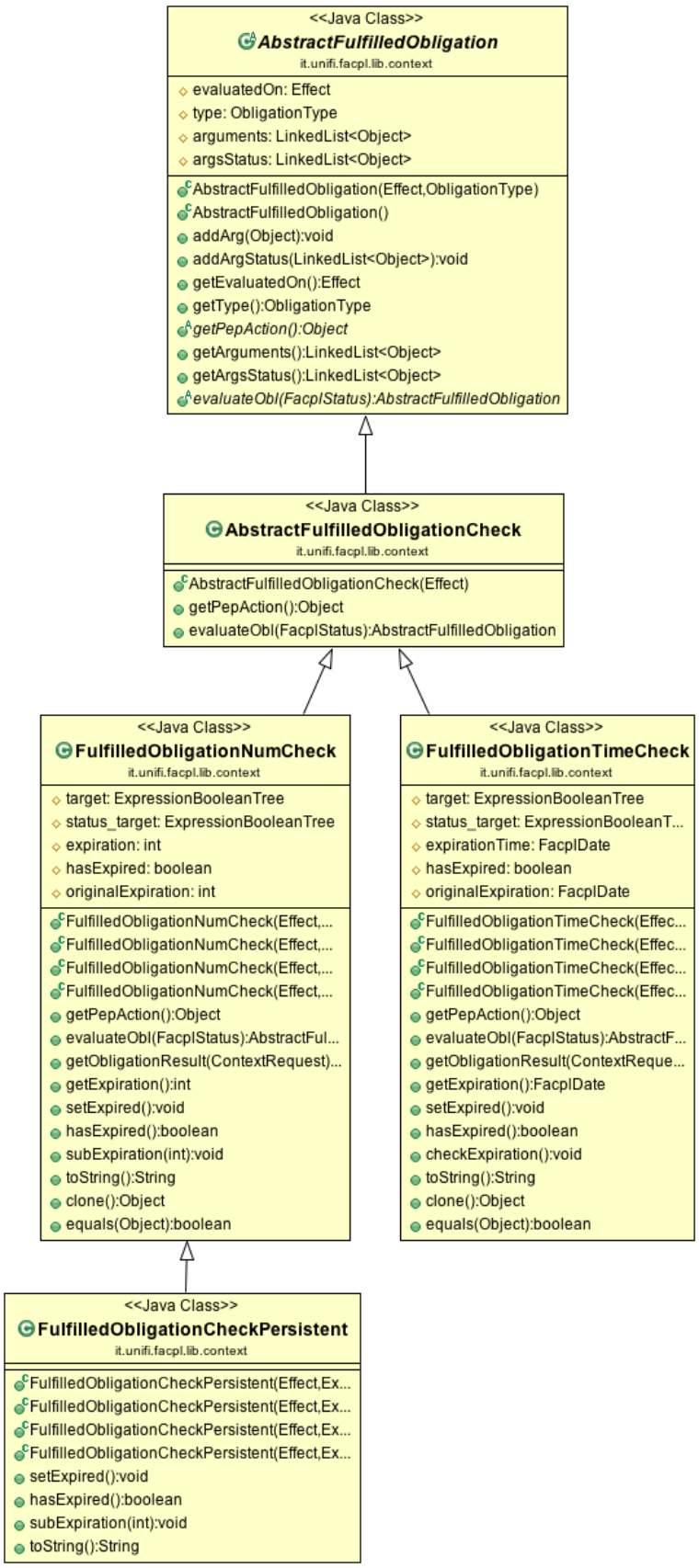

Figura 10: Diagramma FulfilledObligationCheck

Nel Codice 5.10 e 5.11 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 non scaduto

Se l'intero che identifica la scadenza è maggiore di zero, si possono valutare le *Expressions* e richiamare il metodo *subExpiration*(Codice 5.12) per aggiornare il valore.

Codice 5.10: Condizione di expiration valido

```
10      if (this.getExpiration() > 0 || this.getExpiration()==
            -1) {
          //if not expired -> evaluate target
12        l.debug("EVALUATING EXPRESSION OF OBLIGATION: " +
              "\r\n");
          result_target =
              target.evaluateExpressionTree(cxtRequest);
14        result_target_status =
              status_target.evaluateExpressionTree(cxtRequest);
          this.subExpiration(1);
16        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`.

Codice 5.11: Condizione di expiration scaduta

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

Nel Codice 5.10 si richiama *subExpiration*. Il metodo è utilizzato per aggiornare il valore della scadenza. Come si vede nel Codice 5.12 se il valore è -1 non si modifica expiration (si 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`.

Codice 5.12: subExpiration

```
22  public void subExpiration(int i) {
      Logger l =
          LoggerFactory.getLogger(FulfilledObligationNumCheck.class);
24    if (expiration == -1){
        l.debug("EXPIRATION PERSISTENT");
26    }
      if (expiration > 0) {
28      expiration -= i;
        l.debug("NEW EXPIRATION: " + this.toString());
30      if (expiration == 0) {
          this.setExpired();
32      }
      }
34  }
```

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 5.12, se *expiration= -1* il metodo non modifica mai il valore. In questo modo in *GetObligationResult* di FulfilledObligationNumCheck non si raggiunge mai il limite della scadenza, le funzioni non restituiscono mai il risultato `ExpressionValue.ERROR` associato ad un obbligazione scaduta e si valuta sempre le *Expressions*. Nel Codice 5.13 si mostra il costruttore per FulfilledObligationCheck di tipo "P".

Codice 5.13: FulfilledObligationCheck P

```
public class FulfilledObligationCheckPersistent extends
    FulfilledObligationNumCheck {
2   /*
     * same constructor of FulfilledObligationCheck,
4    * but expiration is initialized with -1
     */
6   public FulfilledObligationCheckPersistent(Effect evaluatedOn,
        ExpressionBooleanTree target,
         ExpressionBooleanTree status_target) {
8     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.

Codice 5.14: getObligationResult per le tipo T

```
    FacplDate TIME = new FacplDate();
2   if (TIME.before(this.getExpiration())) {
      l.debug("TIME "+TIME.toString());
4     l.debug("NOT EXPIRED");
      //if not expired -> evaluate target
6     l.debug("EVALUATING EXPRESSION OF OBLIGATION: " + "\r\n");
      result_target = target.evaluateExpressionTree(cxtRequest);
8     result_target_status =
          status_target.evaluateExpressionTree(cxtRequest);
      this.checkExpiration();
10    l.debug("RESULT_TARGET: " + result_target + " ||
          RESULT_TARGET_STATUS: " + result_target_status);
    } else if (TIME.after(this.getExpiration()) ||
        TIME.equals(this.getExpiration()) ) {
12    this.checkExpiration();
      l.debug("TIME "+TIME.toString());
14    l.debug("OBLIGATION CHECK HAS EXPIRED");
      this.checkExpiration();
16    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 5.15 si mostra come viene calcolato il divario tra la scadenza e il tempo.

Codice 5.15: Time difference

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

5.4 PLUGIN ECLIPSE

La sintassi di FACPL è definita in Xtext e viene usato Xtend per la traduzione da FACPL in Java. Xtext [3] è un framework open-source per lo

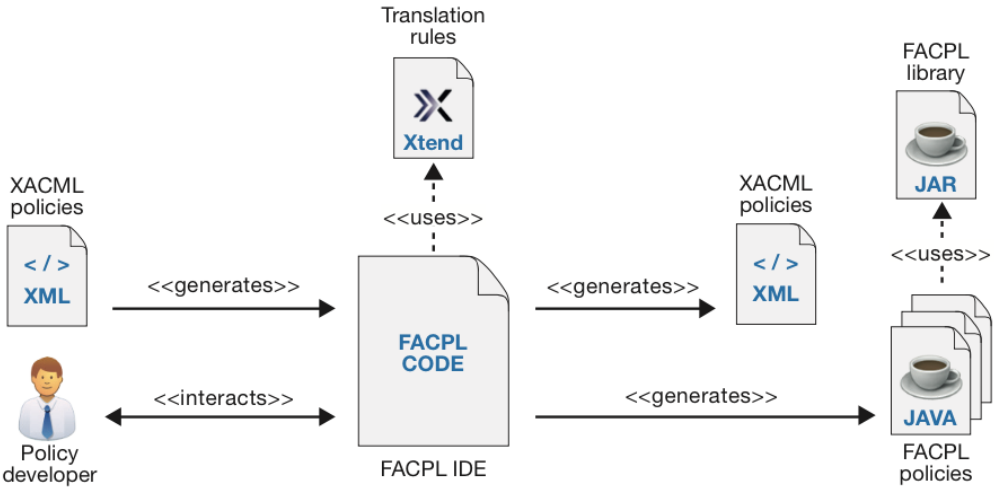

Figura 11: ToolChain di FACPL

sviluppo di linguaggi di programmazione. Xtend [2] è un linguaggio di programmazione basato sintatticamente e semanticamente su Java, ma ha una sintassi più concisa e funzionalità addizionali come: inferenza sui tipi, overload degli operatori e estensione dei metodi. È compilato in codice Java, ha circa lo stesso sistema di tipi statico, ma offre inferenza e integra tutte le librerie Java esistenti. Il plugin utilizza Xtext e Xtend per rendere Eclipse un ambiente di sviluppo per FACPL che offre funzionalità come l'autocompletamento e l'highlighting del codice. In Figura 11 si mostra come da un file FACPL si possa generare il codice Java o XACML utilizzando le regole di traduzione scrtte in Xtend.

```
Obligation:
    '[' EvaluetedOn=Effect typeObl=('M' | 'O')
     (
        ((pepAction=ID '(' (expr+=Expression (',' expr+=Expression)*)* ')')
            | function = PepFunction
        )
     )
     ']'
     |'['
        EvalOn=Effect e1 = Function ',' e2 = Function (',' expiration=Literals)?
     ']'
    ;
```

Figura 12: Produzione per le Obligation

Inizialmente è stata estesa la grammatica aggiungendo le produzioni per i tre tipi di ObligationCheck. Successivamente sono state modificate

```
//-------------------------------------------------
//OBLIGATIONS
//-------------------------------------------------
def compileObligation(Obligation obl) '''
    «IF obl.expiration != null»
    new ObligationCheck(Effect. «obl.evaluetedOn.getName»,
                                «getExpression(obl.e1)»,
                                «getExpression(obl.e2)»,
                                «getExpression(obl.getExpiration())»)
    «ENDIF»
    «IF obl.getE1() != null && obl.getE2() != null && obl.expiration== null »
    new ObligationCheck(Effect. «obl.evaluetedOn.getName»,
                                «getExpression(obl.e1)»,
                                «getExpression(obl.e2)»)
    «ENDIF»
    «IF obl.function == null && obl.pepAction!=null»
    new Obligation("«obl.pepAction»",Effect.
                   «obl.evaluetedOn.getName»,
                   ObligationType.«obl.typeObl»,
                   «obl.expr.getOblExpression»
    «ENDIF»
    «IF obl.function != null»
    new ObligationStatus(    «getPepFunction(obl.function.name)»,
                             Effect.«obl.evaluetedOn.getName»,
                             «generateOblStatusArgs(obl)»)
    «ENDIF»
'''
```

Figura 13: Obligation in Xtend

le regole di traduzione in Xtend in modo da tradurre il codice scritto in FACPL in Java. In Figura 12 si mostra la nuova produzione delle Obligation. Dopo aver esteso la grammatica sono state aggiunte le regole di traduzione. In Figura 13 si può vedere come verrà generata l'obbligazione nel file Java della politica in base al tipo di obbligazione usata nel file FACPL.

In Figura 14 si presenta il plugin. Questo offre funzionalità come l'autocompletamento, il syntax highlighting, auto indentazione e il controllo sull'unicità dei nomi. Inoltre si può generare codice Java o XACML basandosi su un file FACPL usando il menù come si vede in Figura 14. Questo tool insieme ad una guida online si può trovare all'indirizzo:

`http://facpl.sourceforge.net`

5.5 ESEMPI DI USAGE CONTROL IN JAVA

Si ripropongono adesso gli esempi descritti in 4.4 mostrando le classi Java relative ai file FACPL.

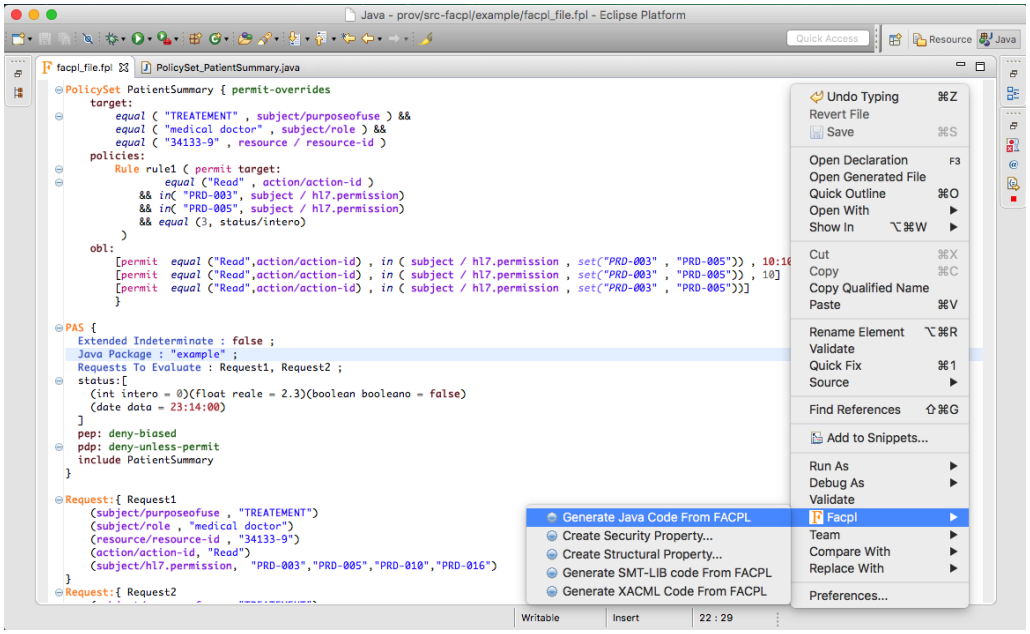

Figura 14: Plugin di FACPL

5.5.1 *Servizio di streaming*

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

La classe StatusStreaming contiene FacplStatus. Si fa notare che nel codice 5.16, 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`.

Codice 5.16: Classe StatusStreaming

```
public class StatusStreaming {
private static FacplStatus status;

  public StatusStreaming() {
  }

  public FacplStatus getStatus() {
    if (status==null){
      HashMap<StatusAttribute, Object> attributes = new
          HashMap<StatusAttribute, Object>();
```

```
10        attributes.put(new StatusAttribute("loginBob",
              FacplStatusType.STRING),"noLogin");
          attributes.put(new StatusAttribute("loginAlice",
              FacplStatusType.STRING),"noLogin");
12        attributes.put(new StatusAttribute("passwordAlice",
              FacplStatusType.STRING), "123456");
          attributes.put(new StatusAttribute("passwordBob",
              FacplStatusType.STRING), "abcdef");
14        attributes.put(new StatusAttribute("streamingAlice",
              FacplStatusType.BOOLEAN), false);
          attributes.put(new StatusAttribute("streamingBob",
              FacplStatusType.BOOLEAN), false);
16        attributes.put(new StatusAttribute("passwordAlice",
              FacplStatusType.STRING), "123456");
          attributes.put(new StatusAttribute("passwordAlice",
              FacplStatusType.STRING), "123456");
18        attributes.put(new StatusAttribute("commercialsBob",
              FacplStatusType.BOOLEAN), false);
          status = new FacplStatus(attributes,
              this.getClass().getName());
20        return status;
        }
22      return status;
    }
24  }
```

Le richieste di ascolto e di login sono simili tra loro. Queste si differenziano in base agli attributi associati. Nel Codice 5.17 si mostra la richiesta di ascolto di Alice come esempio di *request*. Per prima cosa si creano due attributi (linee 9-10), 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* (linee 12-13). Infine si associa il contesto della richiesta allo Status con il metodo `setStatus()` (linea 21) che prende come attributo FacplStatus usando `getStatus()` di StatusStreaming.

Codice 5.17: Request

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

4   public static ContextRequest_Status getContextReq() {
      if (CxtReq != null) {
6       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 5.18 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 `listen`, la seconda invece si riferisce al nome dell'utente che deve essere `Bob` (linee 8-14).

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 `true` solo dopo il login) sia `true` (linee 21-30).

Codice 5.18: 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,
```

```
10          "listen",
            new AttributeName("action", "id"));
12      ExpressionFunction e2=new ExpressionFunction(
            it.unifi.facpl.lib.function.comparison.Equal.class,
14          "Bob",
            new AttributeName("name", "id"));
16      ExpressionBooleanTree ebt = new
            ExpressionBooleanTree(ExprBooleanConnector.AND, e1, e2);
        addTarget(ebt);
18      // PolElements
        addPolicyElement(new Rule_ListenBob());
20      // Obligation
        addObligation(
22          new ObligationCheck(Effect.PERMIT,
                new ExpressionFunction(
24                  it.unifi.facpl.lib.function.comparison.Equal.class,
                    "Bob",
26                  new AttributeName("name", "id")),
                new
                    ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
28                  new StatusAttribute("streamingBob",
                        FacplStatusType.BOOLEAN),
                    true),
30              new FacplDate("00:15:00")
            ));
32  }
```

La regola da valutare associata al PolicySet è stata aggiunta con il metodo `addPolicyElement(new Rule_ListenBob())` (linea 19). Nella `Rule_ListenBob`, mostrata nel Codice 5.19, si controlla che Bob abbia fatto il login e che non debba ascoltare la pubblicità (linee 38-45). 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 (linee 48-53).

Codice 5.19: `Rule_ListenBob`

```
  private class Rule_ListenBob extends Rule {
34    Rule_ListenBob() {
        addId("ListenBob_Rule");
36      // Effect
        addEffect(Effect.PERMIT);
38      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 5.20 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.

Codice 5.20: MainFACPL

```
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.

```
        // Initialise Authorisation System
18      MainFACPL system = new MainFACPL();
        // Result log
20      StringBuffer result = new StringBuffer();
        requests.add(ContextRequest_ListenStandardBob.getContextReq());
22      requests.add(ContextRequest_CommercialsBob.getContextReq());
        requests.add(ContextRequest_ListenStandardBob.getContextReq());
24      requests.add(ContextRequest_ListenStandardBob.getContextReq());

26      for (ContextRequest_Status rcxt : requests) {
          result.append(system.pep.doAuthorisation(rcxt));
28      }
        System.out.println(result.toString());
```

5.5.2 *Accessi in lettura e scrittura di file*

In questa sezione si mostrano le parti di codice Java più significative associate all'esempio descritto in Sezione 4.4.1.

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 5.18, l'*ObligationCheck* della lettura nel Codice 5.21 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`) (linee 25-28). Con questo tipo di struttura l'obbligazione non raggiunge mai la scadenza fintantoché gli attributi delle richieste passano la verifica delle due espressioni.

Codice 5.21: 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
```

5.6 VALUTAZIONE DELLE PRESTAZIONI

In questa sezione si mostrano le differenze tra i tempi computazionali relativi a un insieme di policy che utilizza le *ObligationCheck* e uno che non le utilizza basandosi sull'esempio in Sezione 5.5.2.

In Sezione 4.4.1 è stato mostrato il tempo di esecuzione per ogni singola richiesta e si è notato che le richieste valutate utilizzando solo il PEP ricevevano una risposta molto più velocemente rispetto alle altre.

Per questo motivo è stato ideato un semplice sistema di benchmark per presentare i miglioramenti a livello prestazionale attraverso l'uso delle *ObligationCheck* nel processo valutativo.

Nel Codice 5.22 si presenta la policy duale del Codice 5.21, in questo caso le regole non fanno uso del nuovo tipo di obbligazione.

Codice 5.22: Policy di lettura senza ObligationCheck

```
  private class PolicySet_Read extends PolicySet {
30
    public PolicySet_Read() {
32    addId("Read_Policy");
      // Algorithm Combining
34    addCombiningAlg(
        it.unifi.facpl.lib.algorithm.DenyUnlessPermitGreedy.class);
36    // Target
      ExpressionFunction file1 = new ExpressionFunction(
38      it.unifi.facpl.lib.function.comparison.Equal.class,
        "thesis.tex",
40      new AttributeName("file", "id"));
      ExpressionFunction file2 = new ExpressionFunction(
42      it.unifi.facpl.lib.function.comparison.Equal.class,
        "facpl.pdf",
44      new AttributeName("file", "id"));
      ExpressionBooleanTree setFiles = new
          ExpressionBooleanTree(ExprBooleanConnector.OR, file1,
          file2);
46    // Expression ObligationCheck
      addTarget(setFiles);
48    // PolElements
      addPolicyElement(new Rule_read());
50    // No ObligationCheck
    }
```

Nella classe MainFACPL si utilizzano i due insiemi di regole per controllare due liste di richieste differenti.

Codice 5.23: Main per i benchmarks

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

```
4     MainFACPL system = new MainFACPL(true);
      MainFACPL systemNoCheck = new MainFACPL(false);
6     long start;
      long end;
8     try {
        PrintWriter writer = new
            PrintWriter("/Users/mameli/Desktop/testNoCheck.txt",
            "UTF-8");
10      for (int k=1;k<=100;k++){
          start=System.currentTimeMillis();
12        for (ContextRequest_Status rcxt : requests) {
            result.append(systemNoCheck.pep.doAuthorisation(rcxt));
14        }
          end=System.currentTimeMillis();
16        writer.println(end-start);
        }
18      writer.close();
```

Nella prima lista sono presenti 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.

Nel Codice 5.23 si mostra il caso in cui si utilizza il PolicySet senza ObligationCheck per la prima lista (tutte letture). Tutti i tempi di computazione vengono scritti nel file "testNoCheck.txt" (linea 9).

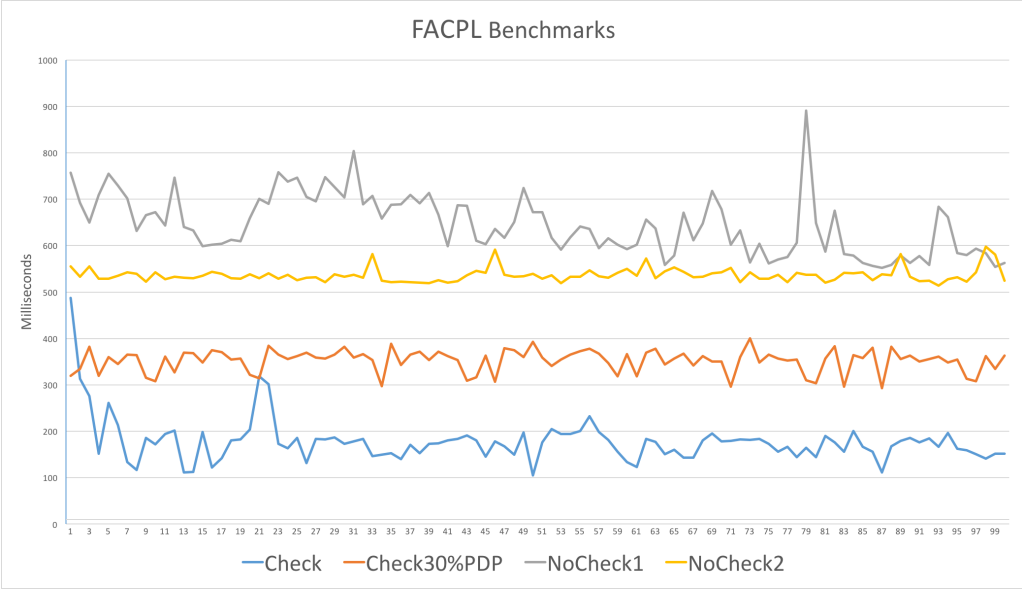

Figura 15: Computazione

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

In Figura 15 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 16 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 era ragionevole attendersi, le misurazioni confermano che, se le richieste di scrittura sono minori rispetto a quelle di lettura, la nuova gestione alleggerisce il processo valutativo.

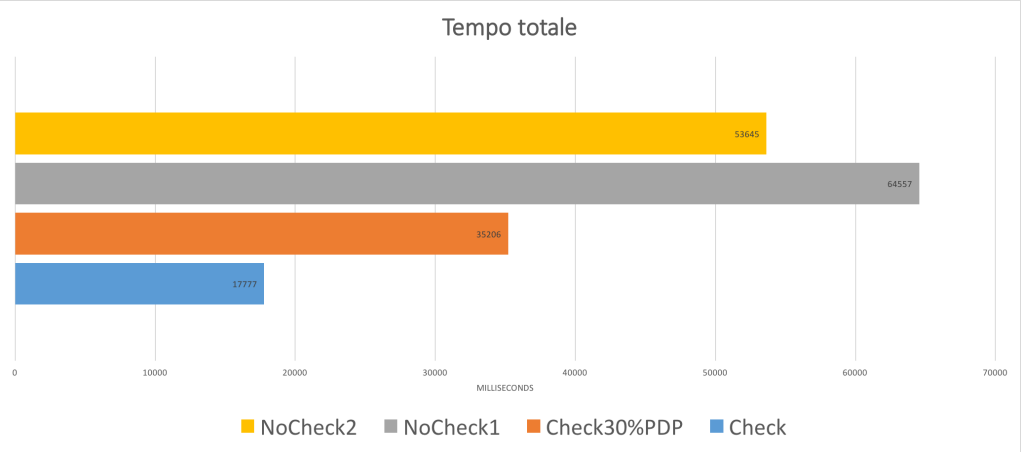

Figura 16: Tempo totale

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 modello UCON_{ABC} ideato da Jaehong Park e Ravi Sandhu.

Successivamente è stato esaminato il linguaggio FACPL su cui si è basata l'implementazione del monitor a runtime. È stata mostrata la sintassi, la semantica e il processo di valutazione del linguaggio. Utilizzando un esempio di politica, sono stati evidenziati i limiti e le strutture mancanti per il supporto al controllo continuativo degli accessi.

Per eliminare i controlli superflui delle verifiche di autorizzazione, è stata modificata la sintassi di FACPL e il processo valutativo. Alle modifiche apportate sono seguite le implementazioni di nuove strutture nella libreria Java che supporta FACPL. Infine, sulla base degli esempi di Usage Control proposti, sono stati mostrati i miglioramenti nell'uso delle risorse computative.

Il lavoro di estensione della libreria è stato svolto insieme al mio collega Federico Schipani. Nella tesi del mio collega si parla di come è stato possibile esprimere in FACPL politiche di controllo degli accessi basate sul comportamento passato. Le modifiche riguardano principalmente il PDP (descritto nel Capitolo 3) e consistono nella creazione di uno stato del sistema. In questa tesi invece si modifica il PEP (descritto ugualmente nel Capitolo 3 e nel Capitolo 4 e 5 nella versione modificata), si implementa un nuovo tipo di obbligazione e si mostrano i miglioramenti dei tempi di valutazione delle richieste.

6.1 SVILUPPI FUTURI

In questo documento sono stati esposti due esempi di Usage Control, uno si basa sul numero di richieste, l'altro sul tempo. La libreria di FACPL è semplice da estendere ed è facile pensare a nuovi possibili controlli su cui le autorizzazioni possono basarsi. Si potrebbe associare le verifiche della richiesta alla posizione geografica dell'utente, oppure alla congestione della rete aggiungendo semplicemente 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

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

UML Unified Modeling Language

BIBLIOGRAFIA

[1] *FACPL guide*. http://facpl.sourceforge.net/guide/facpl_guide.html.

[2] *Xtend Documentation*. http://www.eclipse.org/xtend/documentation/index.html. (Citato a pagina 59.)

[3] *Xtext Documentation*. https://eclipse.org/Xtext/documentation/, [Online; accessed 12-March-2016]. (Citato a pagina 58.)

[4] John F. Barkley. Comparing simple role based access control models and access control lists. In *ACM Workshop on Role-Based Access Control*, pages 127–132, 1997. (Citato a pagina 7.)

[5] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2):85–88, 2015. (Citato a pagina 8.)

[6] Sokratis K. Katsikas, Javier Lopez, and Miguel Soriano, editors. *Trust, Privacy and Security in Digital Business, 7th International Conference, TrustBus 2010, Bilbao, Spain, August 30-31, 2010. Proceedings*, volume 6264 of *Lecture Notes in Computer Science*. Springer, 2010. (Citato a pagina 75.)

[7] Leanid Krautsevich, Aliaksandr Lazouski, Fabio Martinelli, Paolo Mori, and Artsiom Yautsiukhin. Usage control, risk and trust. In Katsikas et al. [6], pages 1–12. (Citato a pagina 9.)

[8] Aliaksandr Lazouski, Fabio Martinelli, and Paolo Mori. Usage control in computer security: A survey. *Computer Science Review*, 4(2):81–99, 2010. (Citato a pagina 11.)

[9] Andrea Margheri. Progetto e realizzazione di un linguaggio formale per il controllo degli accessi basato su politiche, 2011/12. Tesi di laurea magistrale in Informatica.

[10] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. A formal framework for specification, analysis and

enforcement of access control policies. Manoscritto sottoposto per la pubblicazione, 2016.

[11] NIST. A survey of access control models. `http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf`. (Citato alle pagine 7 e 8.)

[12] Jaehong Park and Ravi S. Sandhu. The ucon_{abc} usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004. (Citato alle pagine 5 e 9.)

[13] Jaehong Park, Ravi S. Sandhu, and Yuan Cheng. A user-activity-centric framework for access control in online social networks. *IEEE Internet Computing*, 15(5):62–65, 2011. (Citato a pagina 9.)

[14] Alexander Pretschner, Manuel Hilty, Florian Schütz, Christian Schaefer, and Thomas Walter. Usage control enforcement: Present and future. *IEEE Security & Privacy*, 6(4):44–53, 2008. (Citato a pagina 11.)

[15] Ravi S. Sandhu, David F. Ferraiolo, and D. Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63, 2000. (Citato a pagina 7.)