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

Tesi di Laurea

ESPRIMERE IN FACPL POLITICHE DI CONTROLLO DEGLI ACCESSI BASATE SUL COMPORTAMENTO PASSATO

EXTENSION OF THE FACPL LANGUAGE IN ORDER TO EXPRESS THE ACCESS POLICIES TO THE RESOURCES OF A SYSTEM BASED ON PAST BEHAVIOUR

FEDERICO SCHIPANI

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

Anno Accademico 2014-2015

Federico Schipani: *Esprimere in FACPL politiche di controllo degli accessi basate sul comportamento passato,* Corso di Laurea in Informatica, © Anno Accademico 2014-2015

INDICE

ELENCO DELLE FIGURE

LISTA DI CODICI

"Stay hungry, stay hungry"
— Paolo Bitta, l'uomo chiamato contratto

1

INTRODUZIONE

prova 123

2

ACCESS CONTROL E USAGE CONTROL

Ai giorni d'oggi esistono moltissimi sistemi capaci di condividere dati e risorse computazionali, ed impedire accessi non autorizzati è diventata una priorità inderogabile. Per esempio molti dati personali possono essere raccolti durante alcune attività quotidiane, e proteggere questi dati da malintenzionati è molto importante. Questo, e moltre altre ragioni, sono il motivo per cui esistono sistemi di Access Control, ovvero dei sistemi definiti da un insieme di condizioni che permettono di creare una prima linea difensiva contro accessi indesiderati.

2.1 STORIA DELL'ACCESS CONTROL

Negli anni sono stati proposti diversi approcci per cercare di definire un modello efficiente e scalabile. Secondo il NIST, in [1], una classificazione dei modelli di Access Control è la seguente. Il primo di questi si chiama *Access Control Lists (ACLs)* ed è stato proposto intorno al 1970 spinto dall'avvento dei primi sistemi multi utente.

Successivamente è nato un nuovo modello chiamato *Role-based Access Control (RBAC)* che modifica alcuni aspetti di ACL in modo da rimuovere molte delle limitazioni di quest'ultimo.

Uno dei problemi di RBAC è l'impossibilità di differenziare membri di uno stesso gruppo in modo da negare o permettere accessi sulla base di singoli attributi, ed è per venire in contro a questa necessità che è stato implementato un nuovo modello chiamato *Attribute Based Access Control (ABAC)*, dove le decisioni vengono prese in base ad un set di attributi legati al richiedente, all'ambiente ed alla risorsa per cui si chiede l'accesso.

Anche questo modello però ha delle limitazioni che vengono fuori quando il numero di risorse da gestire è elevato, motivo per cui na-

sce *Policy-based Access Control (PBAC)*. PBAC migliora e standardizza il modello ABAC combinando attributi dalle risorse, dall'ambiente e dal richiedente con informazioni di un particolare insieme di circostanze sotto le quali la richiesta è stata effettuata.

Le organizzazioni non sono statiche, si evolvono e devono rispondere ad una varietà di stimoli, che possono essere legali, economici, finanziari, di mercato o una varietà di fattori di rischio. Anche tecniche avanzate, come per esempio ABAC e PBAC, non riescono in maniera sufficiente a rispondere ai bisogni di dinamismo e cambiamenti dei livelli di rischio, motivo per cui è nato *Risk-adaptive Access Control (RAdAC)* che fornisce un modello adattabile al settore enterprise.

Access Control Lists (ACLs)

ACL è il più datato e basico modello di controllo agli accessi. Prende piede intorno agli anni 70 grazie all'avvento dei sistemi multi utente i quali necessitano di limitare l'accesso a file e dati condivisi, infatti i primi sistemi ad utilizzare questo modello sono stati sistemi di tipo UNIX.

Con la comparsa della multiutenza per sistemi ad uso personale lo standard ACL è stato implementato in molte più ambienti come sistemi UNIX-Like e Windows.

Nonostante negli anni sono stati sviluppati modelli più complessi ACL viene comunque usato nei sistemi operativi recenti, come si può vedere in figura 1 OS X sfrutta questo standard per la gestione dei permessi sul filesystem. Il concetto dietro ACL è uno dei più semplici, in quanto ogni risorsa del sistema che deve essere controllata ha una sua lista che ad ogni soggetto associa le azioni che può effettuare sulla risorsa ed il sistema operativo, quando viene fatta richiesta decide in base alla lista se dare il permesso o meno.

Per esempio, sempre in figura 1, si può vedere come *test_folder* sia la risorsa da controllare, *federicoschipani*, *staff* e *everyone* siano i soggetti e le azioni associate sono, in questo caso, *Read & Write* al primo soggetto e *Read only* agli altri due.

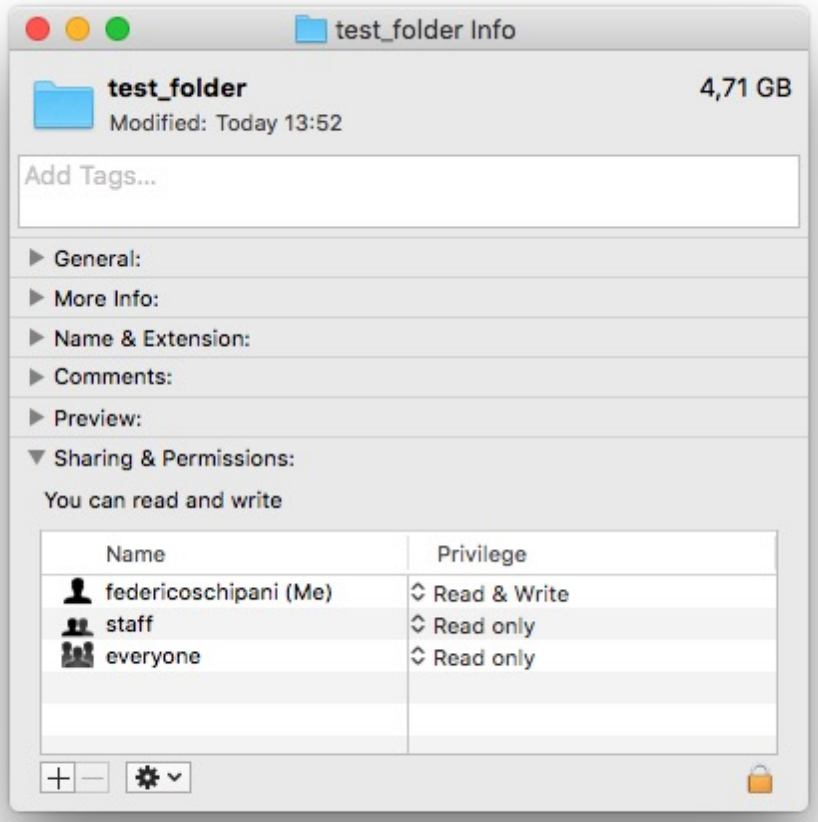

Figura 1: ACL in OS X

La semplicità di questo modello non richiede grandi infrastrutture sottostanti, infatti implementarlo dal punto di vista applicativo risulta abbastanza semplice attraverso l'uso di linguaggi ad alto livello come Python o Java, poiché le strutture che servono per implementare questo standard sono già definite.

Questo elevato grado di relativa facilità di implementazione però ha anche un aspetto negativo che si manifesta quando si ha a che fare con grandi quantità di risorse. Ogni volta che viene richiesto l'accesso ad una risorsa da parte di un entità, utente o applicazione che sia, è necessario verificare nella lista associata, il che lo rende abbastanza oneroso dal punto di vista computazionale.

Un altro lato negativo emerge quando bisogna effettuare modifiche ai permessi di una determinata risorsa, in quanto è necessario andare ad operare sulla lista di quest'ultima, il che rende questo compito incline ad errori ed oneroso dal punto di vista del tempo.

Role-based Access Control (RBAC)

RBAC è un po' l'evoluzione di ACL, in quanto tende a correggerne alcuni, se così si possono chiamare, difetti.

A differenza di ACL il ruolo del richiedente, o la sua funzione, determinerà quando l'accesso sarà garantito o negato. Questo nuovo modello si dedica ad alcuni passi falsi commessi da ACL introducendo nuove ed interessanti funzionalità. Per esempio in ACL ogni utente era trattato come una singola entità distinta da tutte le altre, e questo prevede che ogni utente avesse il suo distinto insieme di permessi per ogni risorsa, il che rende ACL focalizzato sulle risorse.

Un altro difetto che si riscontra in ACL è la sua limitata scalabilità, in quanto impostare un sistema basato su questo standard è un processo che coinvolge tutte le risorse ed i relativi proprietari.

RBAC pone rimedio a questi difetti introducendo il concetto di accesso basato sul ruolo, ovvero può raggruppare diversi utenti in una categoria chiamata ruolo. Questo raggruppamento offre il vantaggio di facilitare la gestione dei permessi, poiché per ogni risorsa non si devono più gestire tutti i singoli utenti, ma basta gestire i permessi associati a queste nuove categorie.

Un utente può anche far parte di più gruppi, per esempio un contabile di un azienda può far parte del gruppo *impiegati* e *contabili* in modo da permettergli l'accesso sia ai documenti riservati ai soli impiegati che quelli riservati ai soli contabili. Come si può vedere in figura 2 e in figura 3 il concetto di gruppo è implementato nei sistemi operativi moderni, in particolare in OS X, Windows e sistemi UNIX-Like.

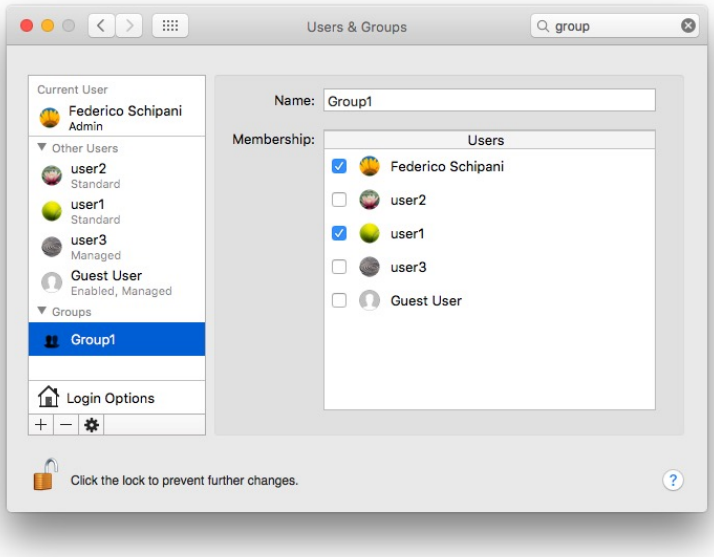

Figura 2: Gruppo in OS X

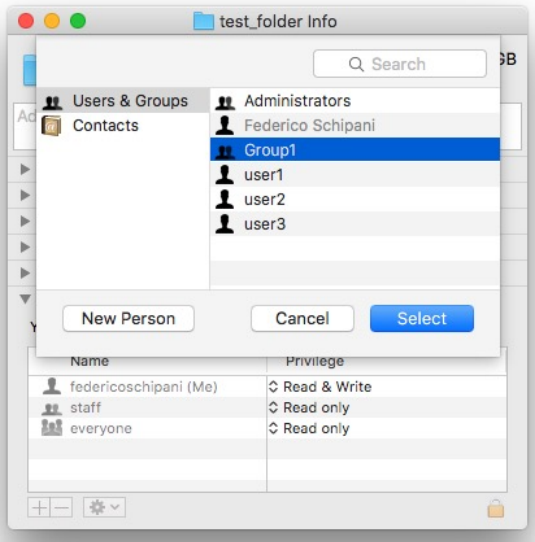

Figura 3: Gruppo in OS X

Non è tutto oro quel che luccica poiché anche RBAC ha i suoi difetti, uno dei più evidenti è l'impossibilità di gestire le autorizzazioni a livello

di singola persona, ed è quindi necessario creare diversi gruppi o trovare altri escamotage per autorizzare, o non autorizzare, singoli utenti appartenti a determinati gruppi. Per questo nasce *Attribute-based Access Control (ABAC)*

Attribute-based Access Control (ABAC)

ABAC è un modello di controllo all'accesso nel quale le decisioni sono prese in base ad un insieme di attributi, associazioni con il richiedente, ambiente e risorsa stessa. Ogni attributo è un campo distinto dagli altri che il *Policy Decision Point (PDP)* compara con un insieme di valori per detrerminare o meno l'accesso alla risorsa. Questi attributi possono provenire da disparate fonti ed essere di svariati tipi. Per esempio nella valutazione di una richiesta possono essere considerati attributi come la data di assunzione di un dipendente ed il suo grado all'interno dell'azienda (Figura **??**). Un vantaggio di ABAC è che non c'è la necessità che il

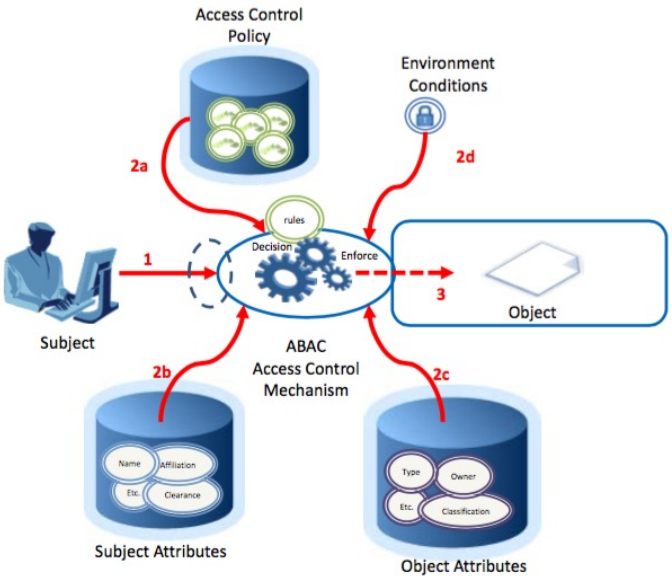

Figura 4: Scenario ABAC base[6]

richiedente conosca in anticipo la risorsa o il sistema a cui dovrà accedere. Finché gli attributi che il richiedente fornisce coincidono con i requisiti l'accesso sarà garantito. ABAC perciò è utilizzato in situazioni in cui i proprietari delle risorse vogliono far accedere utenti che non conoscono direttamente a patto che però rispettino i criteri preposti, il che rende il

tutto molto più dinamico.

Diversamente da RBAC e ACL questo tipo di controllo agli accessi non è implementato nei sistemi operativi, ma è largamente usato a livello applicativo. Spesso si usano applicazioni intermedie per mediare gli accessi da parte degli utenti a specifiche risorse. Implementazioni semplici di questo modello non richiedono grandi *database* o altre infrastrutture,tuttavia in ambienti dove non basta una semplice applicazione c'è necessità di grandi banche di dati.

Una limitazione di ABAC è che in grandi ambienti, con tante risorse, individui e applicazioni ci saranno grandi moli di attributi da gestire.

Policy-based Access Control (PBAC)

PBAC è stato sviluppato per far fronte alle carenze di ABAC, infatti è una sua naturale evoluzione e tende ad uniformare ed armonizzare il sistema di controllo accessi. Questo modello cerca di aiutare le imprese a indirizzarsi verso la necessità di implementare un sistema di controllo agli accessi basato su policy.

PBAC combina attributi dalle risorse, dall'ambiente e dal richiedente con informazioni su determinate circostanze sotto le quali la richiesta è stata effettuata ed inoltre si serve di ruoli per determinare quando l'accesso è garantito.

Nei sistemi ABAC gli attributi richiesti per avere accesso ad una particolare risorsa sono determinati a livello locale e possono variare da organizzazione ad organizzazione. Per esempio, un'unità organizzativa può determinare che l'accesso ad un archivio di documenti sensibili è semplicemente soggetto a richiesta di credenziali e ruolo particolare. Un'altra unità invece, oltre a richiedere credenziali e ruolo, richiede anche un certificato. Se un documento viene trasferito dal secondo al primo archivio perde la protezione fornita da quest'ultimo e sarà soggetto solo alla richiesta di credenziali e ruolo. Con PBAC invece si ha un solo punto dove vengono gestite le policy, e queste policy verranno eseguite ad ogni tentativo di accedere alla risorsa. PBAC quindi è un sistema molto più complicato di ABAC e perciò richiede il dislocamento di infrastrutture molto più onerose dal punto di vista economico che includono *database*, *directory service* e altri applicativi di mediazione e gestione. PBAC non

richiede solo un applicazione per gestire la valutazione delle policy, ma richiede anche un sistema per la scrittura di quest'ultime in modo che non risultino ambigue. Un linguaggio basato su XML si chiama *eXtensible Access Control Markup Language (XACML)*, ed è sviluppato in modo tale da creare policy facilmente leggibili da una macchina.

Sfortunatamente però queste policy non sono facili da scrivere e l'uso di XACML non necessariamente rende facile il processo di creazione, specifica e valutazione corretta di una policy.

Ci vuole anche un modo per assicurare che tutti gli utenti di un sistema utilizzino lo stesso insieme di attributi, che è un compito più facile a dire che farsi. Gli attributi dovrebbero essere forniti da un'entità chiamata *Authoritative Attribute Source (AAS)* che, oltre a fare da sorgente per gli attributi deve anche occuparsi della loro consistenza. In più bisogna instaurare un meccanismo per la verificare che questi attributi provengano realmente dall'AAS.

Come detto prima può sembrare facile fare una cosa del genere, ma bisogna considerare il caso in cui più aziende lavorano insieme e devono implementare un sistema di controllo degli accessi in comune. Un problema si può verificare quando un'azienda valuta la gestione dell'AAS tramite una particolare *repository*, ma un'altra azienda non è d'accordo a questo tipo di soluzione.

Risk-Adaptive Access Control (RAdAC)

Le organizzazioni non sono statiche. Si evolvono costantemente e rispondono ad una varietà di stimoli sempre maggiore. La loro natura dinamica porta ad avere la necessità di policy che si adattino al sistema che le circonda. Con l'avanzare del tempo cambia anche le minacce, ed un organizzazione deve costantamente tenere sott'occhio il rischio, quindi si inizia a parlare anche di livello di rischio.

Anche i più avanzati modelli come ABAC e PBAC non riescono a soddisfare questa necessità di dinamismo e cambiamento del livello del rischio. Per queste situazioni entra in gioco RAdAC che è stato concepito proprio per adattarsi a questi contesti.

RAdAC rappresenta unf ondamentale cambiamento nella gestione del controllo agli accessi, in quanto estende i precedenti modelli con l'introduzione nel processo di valutazione di condizioni ambientali e livello di rischio. RAdAC combina informazioni riguardo l'attendibilità del richiedente, informazioni riguardo le infrastrutture e rischi dell'ambiente circostante per la creazione di una metrica di pericolo e per una corretta valutazione.

Una volta raccolte tutte queste informazioni vengono usate per la valutazione delle policy. Una policy in questo modello può includere direttive su come l'accesso deve essere gestito sotto determinate situazioni e sotto determinati livelli di rischio.

Per esempio, un utente accede ad una determinata risorsa in un determinato momento, e gli vengono richieste delle normali credenziali di accesso. In un secondo momento, quando magari il livello di rischio sale, può essere richiesto anche un certificato.

Le policy definite in RAdAC permettono anche di sovrascrivere il livello di rischio e le varie valutazioni vengono salvate in uno storico. Questo vuol dire che RAdAC usa un approccio euristico per determianre quando l'accesso deve essere garantito o meno.

Ovviamente le infrastrutture per gestire tutto questo sono molto estese e complesse visto il numero di dati che sono richiesti per generare una corretta valutazione, basata anche sul livello di pericolo attuale. Diversi sistemi sono necessari per far funzionare RAdAC , tra cui grandi database.

Implementare un sistema del genere può essere molto frustrante poiché ci sono numero ostacoli da superare per ottenere un risulato quanto meno usabile. Far interagire tutti i sistemi coinvolti in RAdAC può diventare una vera e propria sfida in quanto i dati non sono standardizzati. Il secondo problema è accomunato con PBAC: entrambi i sistemi fanno affidamento su policy per determinare quando garantire o meno un accesso. Questo richiede un modo di standardizzare queste regole, in modo da non renderle ambigue ed agevolare lo scambio tra sistemi differenti. XACML è una possibile soluzione a questo problema, ma è ancora troppo acerbo per essere usato in soluzioni RAdAC .

Terzo problema è l'affidabilità dei dati che vengono forniti al sistema.

Una soluzione possono essere i moduli TPM (Trusted Platform Module), ovvero dei componenti hardware che assicurano la consistenza dei dati, o dei tool di analisi comportamentale. Sfortunatamnete però non sono ancora così affidabili da essere usati in un sistema del genere.

Il quarto problema è legato al dinamismo di RAdAC , in quanto è necessario uno standard per descrivere varie condizioni ambientali necessarie al processo di decisione.

Il quinto problema invece è legato all'affidamento che questo sistema fa sull' euristica per le decisioni. Questo problema, come prima è condizionato dall' immaturità degli algoritmi usati in questo ambito.

2.2 USAGE CONTROL

Come detto ad inizio capitolo, ai giorni d'oggi proteggere l'accesso alle nostre risorse digitali è uno dei problemi fondamentali nell'ambito della sicurezza.

Oggi sono presenti differenti tipi di sistemi diversi che richiedono un modello più flessibile e corposo per gestire la sicurezza. Questa sezione parlerà di un nuovo modello, chiamato *Usage Control* [2].

Usage Control si propone come un nuovo e promettente approccio per l'Access Control, prendendo spunto e migliorando sistemi come *Trust Management* (TM) e *Digital Rights Management*(DRM). In particolare verrà trattato un particolare modello, inizialmente proposto da Sandhu e Park[2], chiamato UCON.

UCON migliora l'access control in due aspetti fondamentali, la mutabilità degli attributi e la continuità delle decisioni sull'accesso. La mutabilità degli attributi significa che questi valori possono cambiare nel corso del tempo, e visto che UCON è basato su quest'ultimi le decisioni di accesso devono essere rivalutate ogni volta che vengono aggiornati.

La continuità delle decisioni invece significa che non vengono più prese decisioni solo a priori, ma anche durante l'accesso. Quindi, se durante l'utilizzo, qualche attributo cambia e la policy non è più soddisfatta viene revocato l'accesso.

Il vantaggio di Usage Control è la sua capacità di esprimere vari scenari, riuscendo così a includere e migliorare sistemi descritti in 2.1. Il passaggio da Access Control a Usage Control è importante soprattutto quando si va a considerare ambienti *network related*, come possono essere il web, il cloud o il grid computing.

Il processo decisionale in Usage Control è diviso in due fasi [3]. la prima fase è una fase di *pre decision* che fondamentalmente è la classica decisione presa in Access Control, questa decisione viene presa al momento in cui è effettuata la prima richiesta per produrre la decisione di accesso. La seconda fase è chiamata *ongoing decision*, ed è un processo che implementa il concetto di continuità. I componenti necessari a questo tipo di processo decisionale sono dei predicati, chiamati *authorizations*, che vengono valutati sul soggetto e sugli attributi dell'oggetto, degli altri predicati, questa volta chiamati *conditions*, valutati sulle variabili d'ambiente, ed infine delle azioni chiamate *obligations* che devono essere eseguite durante l'accesso. Un altro componente di cui necessita UCON è ovviamente un predicato, come nell'access control che viene valutato per l'accesso iniziale, chiamato questa volta *Rights*.

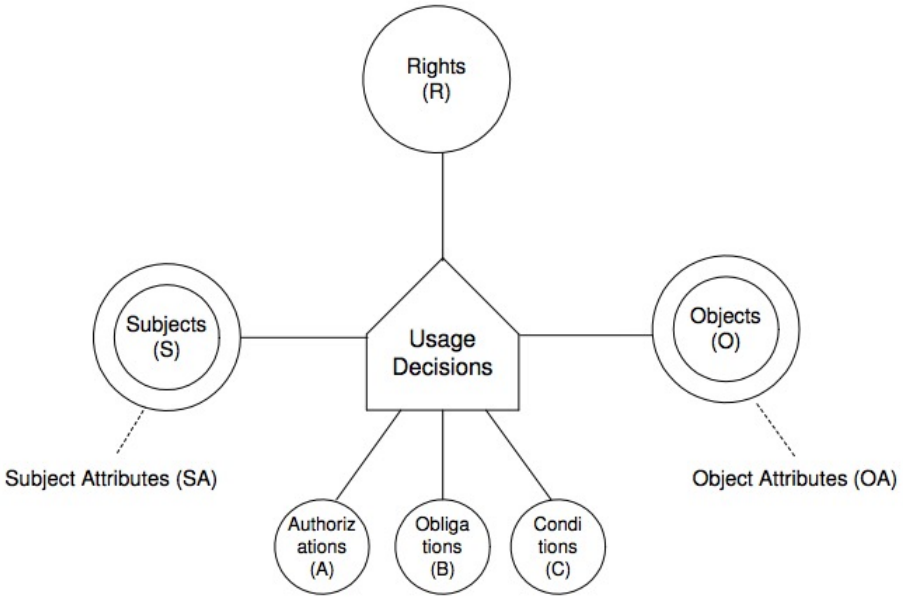

Figura 5: Insieme dei componenti di UCON [4]

Verranno ora proposti alcuni esempi di utilizzo di Usage Control.

Esempio 1: Accesso ai file

Dentro ad un sistema ci sono vari file, ai quali per questione di consistenza, possono accedere massimo due persone in lettura oppure una sola in scrittura.

In un primo momento nessuno sta visualizzando o scrivendo un determinato file, ed un utente genenrico chiederà l'accesso in lettura per questo file, ovviamente il responso sarà positivo in quanto non viola nessuna regola preposta prima.

Dopo un po' di tempo, mentre il primo sta ancora leggendo, un altro utente chiede l'accesso in scrittura, che gli viene negato. In un istante di tempo successivo il primo utente sta continuando a leggere, ed anche il secondo utente vuole leggere. In questo caso viene dato responso positivo.

Infine, entrambi gli utenti smettono di leggere, ma uno di loro vuole apportare una modifica, allora richiede l'accesso in scrittura, che questa volta gli viene consentito poiché nessuno sta leggendo. In Figura 6 viene mostrato un diagramma di flusso che sintetizza e permette di capire meglio quanto detto prima.

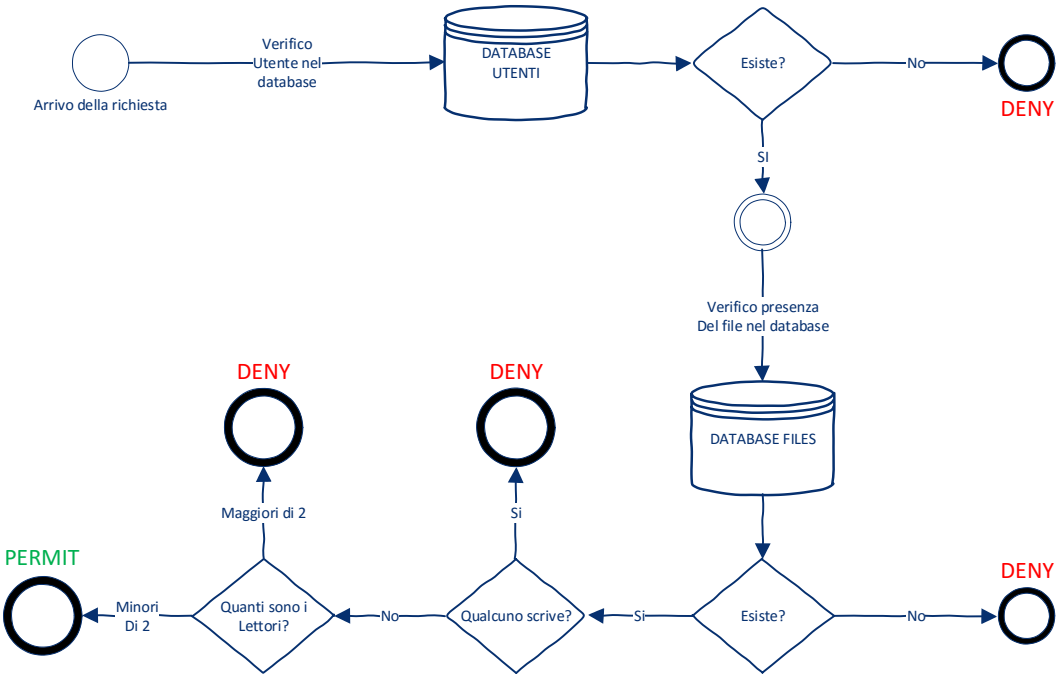

Figura 6: Diagramma di flusso del primo esempio

Esempio 2: Acquisto o noleggio di contenuti

Un altro utilizzo possibile di Usage Control riguarda l'analisi del comportamento passato. Un'azienda fornisce ai propri clienti la possibilità di effettuare noleggi o acquisti di contenuti multimediali (musica, video, film, serie tv e via discorrendo).

In caso il contenuto fosse stato acquistato l'acquirente potrà ottenere l'accesso infinite volte per infinito tempo. Mentre in caso di noleggio saranno presenti delle condizioni, come per esempio il massimo numero di fruizioni del contenuto o una data di scadenza che, una volta oltrepassata, impedirà l'ulteriore visione del contenuto noleggiato in precedenza. Come nell'esempio precedente viene proposto un diagramma di flusso, proposto in Figura 7, che permette di capire meglio il funzionamento questo sistema di *Usage Control*

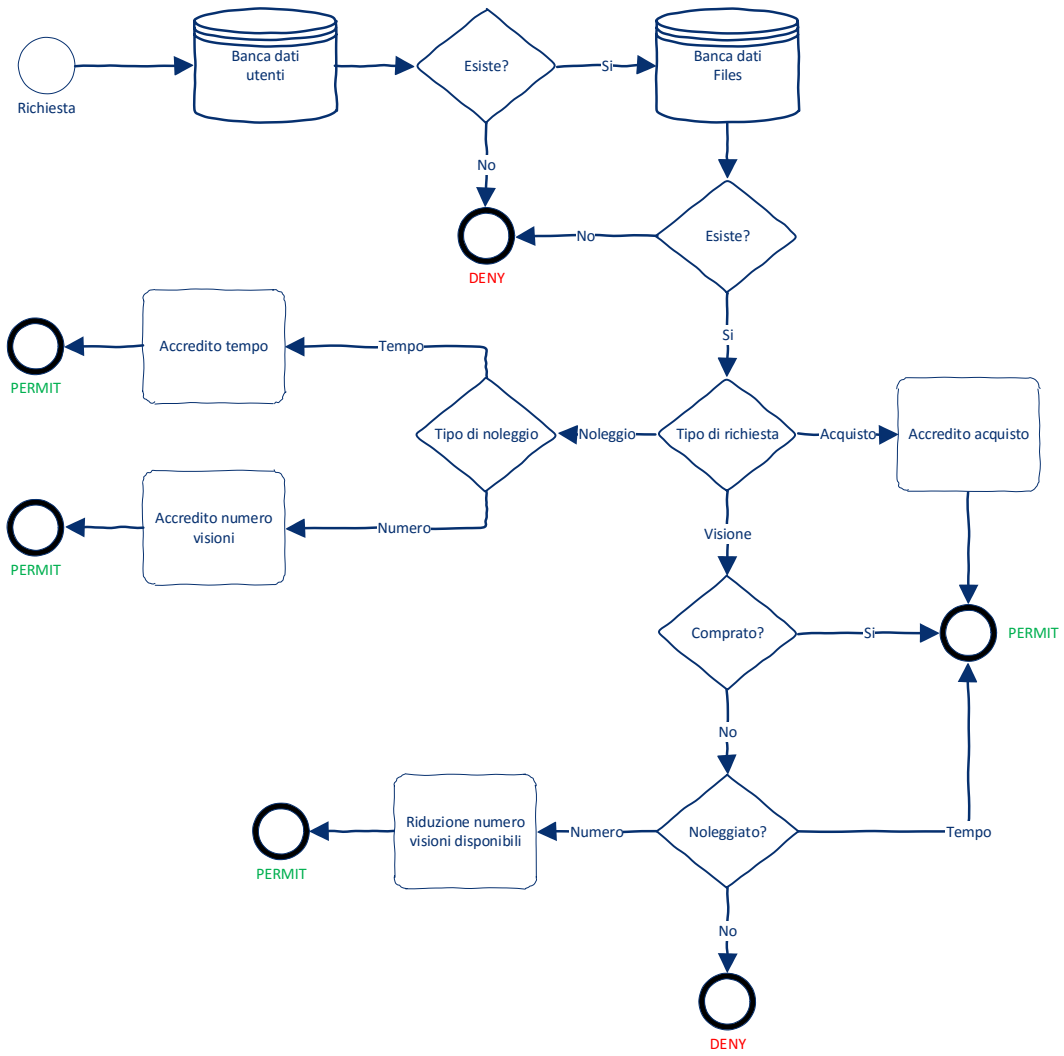

Figura 7: Diagramma di flusso del primo esempio

3

FORMAL ACCESS CONTROL POLICY LANGUAGE

Negli anni molti linguaggi sono stati proposti per definire policy di access control. Uno di questi è stato rilasciato nel 2003 da parte di OASIS ed il suo nome è *eXtensible Access Control Markup Language* (XACML). Questo linguaggio ha una sintassi basata su XML e fornisce caratteristiche avanzate per l'access control. Il problema fondamentale di XACML è che non ha una sintassi facile da leggere e da scrivere.

L'obiettivo di *Formal Access Control Policy Language* (FACPL) è definire una sintassi alternativa per XACML in modo da renderlo più agevole da usare. FACPL quindi è parzialmente inspirato a XACML, ma oltre ad introdurre una nuova sintassi ridefinisce alcuni aspetti aggiungendo nuove caratteristiche. Il suo scopo però non è sostituire XACML, ma fornire un linguaggio compatto ed espressivo per facilitare le tecniche di analisi attraverso tool specifici.

3.1 IL PROCESSO DI VALUTAZIONE DI FACPL

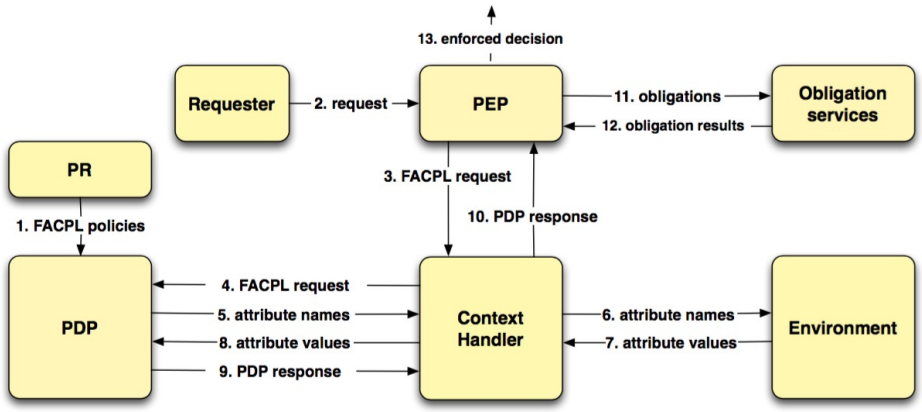

Figura 8: Il processo di valutazione di FACPL

In figura 8 è mostrato il processo di valutazione delle policy definite in FACPL. I componenti principali sono tre:

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

Le policy sono memorizzate nel PR, il quale le rende disponibili al PDP che deciderà, successivamente, se garantire l'accesso o meno (Primo step). Nello step 2, quando il PEP riceve una richiesta, le credenziali di quest'ultima vengono codificate in una sequenza di attributi (ogni attributo è una coppia stringa valore) che, nello step 3, andranno a loro volta a formare una *FACPL Request*. Al quarto step il *context handler* aggiungerà attributi di ambiente (per esempio l'ora di ricezione della richiesta) e manderà la richiesta al PDP. A questo punto il PDP, tra il quinto e l'ottavo step, valuterà la richiesta e fornirà un risultato, il quale può eventualmente contenere delle *obligations*. La decisione del PDP può essere di quattro tipi, *permit*, *deny*, *not-applicable* o *indeterminate*. Il significato delle prime due decisioni è facilmente intuibile, mentre per le ultime due vuol dire che c'è stato un errore durante la valutazione. Gli errori possono essere di diverso tipo, e vengono gestiti attraverso algoritmi che combinano le decisioni delle varie policy per ottenere un risultato finale. Le *obligations* sono azioni, eseguite dal PEP, correlate al

sistema di controllo degli accessi. Queste azioni possono essere di svariati tipi, come per esempio generare un file di log, o mandare una mail. Allo step 13, sulla base del risultato delle *obligations*, il PEP esegue un processo chiamato*Enforcement* il quale restituirà un'altra decisione. Quest'ultima decisione corrisponde alla decisione finale del sistema e può differire da quella del PDP.

3.2 LA SINTASSI DI FACPL

La sintassi di FACPL è definita nella tabella 1. La sintassi è fornita come una grammatica di tipo EBNF, dove il simbolo ? corrisponde ad un elemento opzionale, il simbolo $*$ corrisponde ad una sequenza con un numero arbitrario di elementi (anche 0), ed il simbolo $+$ corrisponde ad una sequenza non vuota con un numero arbitrario di elementi.

Al livello più alto c'è il *Policy Authorisation System (PAS)*, il quale definisce le specifiche del PEP e del PDP. Il PEP è definito semplicemente come un *enforcing algorithm* che sarà applicato per decidere quali decisioni verrà eseguito il processo di *enforcement*.

Il PDP invece è definito come una sequenza (non vuota) di *Policy*, ed un algoritmo di combining che combinerà i risultati di queste policy per ottenere un unico risultato finale.

Una *policy* può essere una semplice *rule* o una *policy set*, quest'ultima avrà al suo interno altre *policy set* o *rule*, ed in questo modo viene formata una gerarchia di policy.

Un *policy set* individua un target, che è una espressione che indica il set di richieste di accesso alla quale si applica la policy, una lista di *obligations*, che definiscono azioni obbligatorie o opzionali che devono essere eseguite nel processo di *enforcement*, una sequenza di altre *policy*, ed un algoritmo per combinarle.

Una *rule* includerà un *effect*, che sarà permit o deny quando la regola è valutata correttamente, un target ed una lista di *obligations*.

Le *Expressions* sono formate da *attribute names* e valori (per esempio boolean, double, strings, date).

Tabella 1: Sintassi di FACPL

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

Un *Attribute Name* indica il valore di un attributo il quale può essere contenuto nella richiesta o nel contesto. FACPL usa per gli *Attribute Name* una forma del tipo *Identifier / Identifier* , dove il primo Identifier indica la categoria, ed il secondo il nome dell'attributo. Per esempio *Action / ID* rappresenta il valore di un attributo ID di categoria Action.

I *Combining Algorithm* implementano diverse strategie che servono per

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**)]

risolvere conflitti tra le varie decisioni, restituendo alla fine un'unica decisione finale.

Una *obligation* ha al suo interno un effect, un tipo, ed una azione eseguita dal PEP con la relativa *Expression*.

Una *request* consiste di una sequenza di attributi organizzati in categorie.

La risposta ad una valutazione di una richiesta FACPL è scritta usando la sintassi riportata in tabella 2. La valutazione in due step, descritta precedentemente in sezione 3.1, produce due tipi di risultati. Il primo è la risposta del PDP, il secondo è una decisione, ovvero una risposta del PEP. La decisione del PDP, nel caso in cui ritorni `permit` o `deny`, viene associata ad una lista, anche vuota, di fulfilled obligations.

Una *fulfilled obligation* è una semplice coppia formata da un tipo (M o O) ed una azione i quali argomenti sono ottenuti dalla valutazione del PDP.

3.3 LA SEMANTICA DI FACPL

Molteplici sono le componenti di FACPL, e la semantica ora verrà informalmente analizzata.[5] Prima verrà presentato il processo che porterà ad una risposta del PDP, successivamente il processo di enforcement del PEP.

Quando il PDP riceve una richiesta, per prima cosa valuta la richiesta sulle basi delle policy disponibili, successivamente determinerà un risultato combinando le decisioni ritornate da queste policy attraverso degli algoritmi di combining.

La valutazione della policy rispetto alla richiesta comincia verificando l'applicabilità alla richiesta, che è fatta valutando un espressione definita *target*.

Si possono valutare due casi distinti:

- Supponiamo che l'applicabilità dia esito positivo, nel caso ci sia una *rule* sarà ritornato il valore risultato dalla valutazione di quest'ultima, mentre se c'è un *policy set* il risultato è ottenuto valutando le policy contenute all'interno, e combinando i loro valori con un algoritmo specificato in fase di creazione del PDP. Successivamente a queste valutazioni verrà effettuato il fulfilment delle obligation contenute all'interno delle policy.
- Supponiamo ora che l'applicabilità non dia esito positivo, ovvero la valutaizone del *target* restituisca `false`. In questo caso il risultato della policy sarà `not-app`. Mentre se *target* restituisce un valore non booleano o ritorna un errore il risultato della policy sarà `indet`.

Valutare le espressioni corrisponde ad applicare degli operatori e risolvere i nomi degli attributi che contengono, e di conseguenza ricavarne un valore.

Se non è possibile trovare un attributo, magari perché non esiste, viene ritornato un valore speciale, chiamato `BOTTOM`. Questo valore può essere usato per implementare diverse strategie per gestire l'assenza di attributi. FACPL gestisce questo valore come una specie di `false`, quindi permette la mancanza di attributi senza la generazione di errori.

La valutazione di un espressione tiene conto anche dei tipi degli argomenti. Se l'argomento è del tipo aspettato l'operatore viene applicato correttamente, sennò, se un argomento è `BOTTOM` e nessun'altro è `error` viene ritornato `BOTTOM`, mentre se almeno uno di essi è `error`, viene ritornato `error`.

Con l'operatore `and` o `or` il trattamento sarà leggermente dievrso, in quanto `BOTTOM` viene ritornato solo se un argomento è tale e nessun'altro è `false` o `error`, mentre in caso contrario viene ritornato `error`.

La valutazione di una policy termina con il fulfillment di tutte le `obligations` le quali hanno il valore di applicabilità coincidente con quello ritornato dalla valutazione della policy. Quest'operazione consisten nel

valutare tutte le espressioni presenti al interno delle `obligations` coinvolte nel processo. Se ci sarà un errore nel processo di fulfilment allora il risultato della policy sarà `indet`, altrimenti il risultato del fulfilment sarà uguale a quello della valutazione del PDP.

Gli algoritmi di combining, come detto prima hanno lo scopo di combinare le decisioni risultanti dalla valutazione delle richieste in accordo con le policy. Un'altra funzione che hanno è ritornare le *obligations* corrette nel caso in cui la valutazione finale risulti `permit` o `deny`. Questa famiglia di algoritmi ha una strategia δ che viene usata per restituire le *obligation*, e può essere di due tipi. Il primo tipo è la strategia `all` (tutto), ovvero richiede la valtuazione di tutte le policy e ritorna le `fulfilled obligation` pertinenti a tutte le decisioni.

Il secondo tipo è la strategia `greedy` (golosa) prescrive che appena è ottenuta una decisione che non può cambiare a causa della valutazione di susseguenti policy nella sequenza di input, l'esecuzione si arresta.

Come ultimo step il risultato del PDP viene mandato al PEP per l'enforcement. Il PEP per effettuare questo processo deve eseguire l'azione all'interno di ogni `fulfilled obligation` e decidere come comportarsi per le decisioni di tipo `not-app` e `indet`.

Per fare questo processo usa delle strategie. In particolare, l'algoritmo `deny-biased` (rispettivamente, `permit-based`) effettua l'enforcement dei `permit` (rispettivamente `deny`) solo quando tutte le corrispondenti obligations sono correttamente scaricate, mentre effettua l'enforcement dei `deny` (rispettivamente `permit`) in tutti gli altri casi. Invece, l'algoritmo di base lascia tutte le decisioni non cambiate ma, in caso di decisioni `permit` e `deny`, effettua l'enforcement di `indet` se un errore occorre quando si stanno rilasciando le `obligations`. Questo evidenzia che le `obligations` non solo influenzano il processo di autorizzazione, ma anche l'enforcement. Gli errori causati dalle `obligations` con tipo O vengono ignorati.

3.4 ESEMPIO DI POLITICA CON FACPL

In questa sezione verrà analizzata una semplice politica scritta in FACPL con delle eventuali richieste.

Codice 3.1: Esempio di politica in FACPL

```
PolicySet fileRule {  permit-overrides
  target:
    equal("458", resource/resource-id)
  policies:
    Rule writeRule (  permit  target:
         equal ("WRITE" , subject/action )
        &&  equal ("ADMINISTRATOR", subject/role)
    )
    Rule writePeronio (  permit  target:
         equal("PERONIO", subject/id)
   )
    Rule denyRule (  deny  target:
         equal("GUEST", subject/role) )
  obl:
   [  deny M action2 (subject / id )]
   [  permit M action1 (subject / id)]
}
```

Con questo codice si vuole ottenere lo scopo di regolare l'accesso ad una risorsa chiamata 458. In questo caso gli utenti che hanno ruolo *GUEST* non possono accedere, mentre gli *ADMINISTRATOR* si, fatta eccezione per l'utente Peronio, che qualunque ruolo abbia può accedere. Le richieste effettuate al sistema vengono mostrate in Codice 3.2, e sono tre. La prima proviene dall'utente Gianfabrizio che fa parte degli *ADMINISTRATOR*, la seconda e la terza rispettivamente dall'utente Gianpietro e Peronio che fanno entrambi parte dei *GUEST*.

Codice 3.2: Richieste per Codice 3.1

```
Request:{ Request1
  (subject/action , "WRITE")
  (subject/role , "ADMINISTRATOR")
  (resource/resource-id , "458")
  (subject/id, "GianFabrizio")
}
Request:{ Request2
  (subject/action , "WRITE")
  (subject/role , "GUEST")
  (resource/resource-id , "458")
  (subject/id, "GianPietro")
}
Request:{ Request3
```

```
  (subject/action , "WRITE")
  (subject/role , "GUEST")
  (resource/resource-id , "458")
  (subject/id, "PERONIO")
}
```

L'output prodotto dalle seguenti richieste è il seguente:

```
Request: Request1
  Authorization Decision: PERMIT
  Obligations: PERMIT M action1([GianFabrizio])
Request: Request2
  Authorization Decision: DENY
  Obligations: DENY M action2([GianPietro])
Request: Request3
  Authorization Decision: PERMIT
  Obligations: PERMIT M action1([PERONIO])
```

Ovviamente alla prima richiesta il risultato è permit, in quanto l'utente è un amministratore. Alla seconda richiesta il risultato è deny poiché l'utente è un ospite, mentre alla terza, nonostante l'utente faccia parte dello stesso gruppo del secondo riesce ad ottenere risultato permit per via della regola che considera il suo nome.

FACPL, come mostrato dall'esempio, permette di fare richieste ed ottenere delle risposte, ma queste richieste sono totalmente indipendenti l'una dall'altra, quindi l'ordine di esecuzione non avrebbe influenzato in alcun modo il risultato finale. In sezione 2.2 sono stati introdotti due esempi i quali non possono, per ora, essere implementati in FACPL poiché manca quest'aspetto che crea dipendenza tra le richieste.. Per creare questa dipendenza tra richieste è necessario che il sistema si ricordi in qualche modo quello che è successo prima, perciò si inizia a parlare di stato. Lo scopo del Capitolo 4 e 5 è proprio permettere a FACPL questo tipo di valutazione.

4

IMPLEMENTARE USAGE CONTROL IN FACPL

FACPL, fino alla versione descritta nel capitolo 3, non aveva la possibilità di essere sfruttato per *Usage Control*.

Grazie a delle nuove strutture implementate insieme al mio collega Filippo Mameli, adesso è possibile usare FACPL per *Usage Control*, introducendo miglioramenti descritti in 2.2.

La nuova funzionalità consiste nel prendere decisioni tenendo conto delle richieste già effettuate. Introdurre questa nuova estensione ha richiesto del lavoro sulla libreria, in quanto è stato necessario aggiungere nuove componenti e di conseguenza modificare il processo di valutazione di una policy. Infine è stato necessario anche introdurre delle modifiche alla sintassi del linguaggio in modo da poterle sfruttare facilmente.

4.1 ESTENSIONE DEL PROCESSO DI VALUTAZIONE

Il processo di valutazione è stato esteso per via delle modifiche introdotte. Rispetto al processo di valutazione standard, descritto in sezione 3.1, sono state aggiunte componenti al grafico, rendendolo così adatto allo *Usage Control*, in particolare alla valutazione di richieste basate sul comportamento passato.

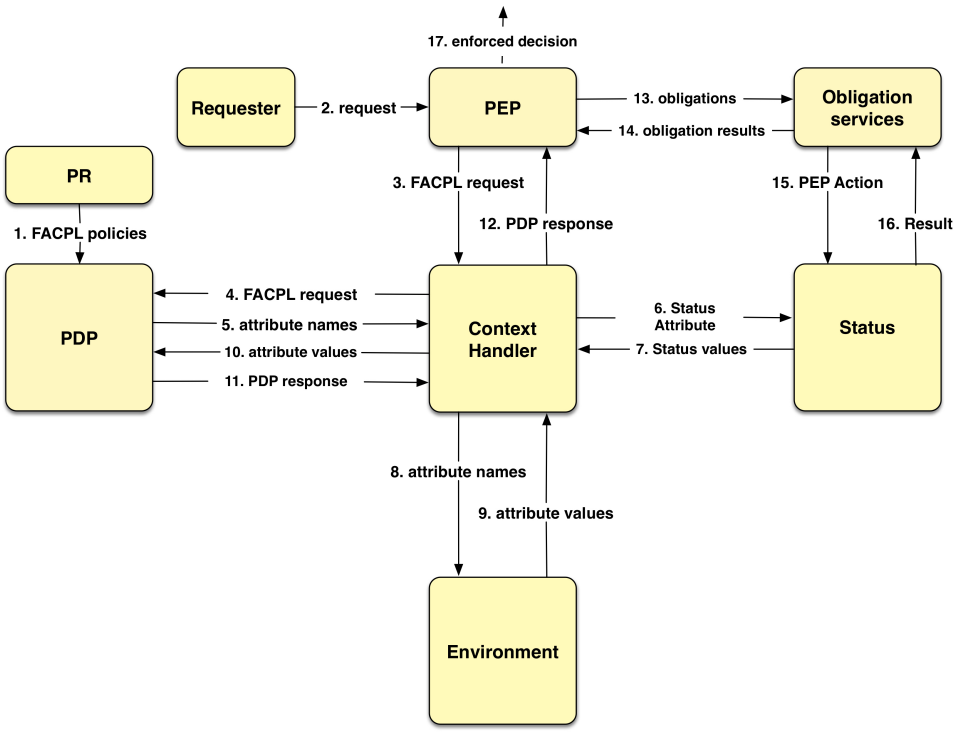

Figura 9: Nuovo processo di valutazione in FACPL

Come si nota in figura **??** è stato aggiunto un componente alla struttura della valutazione. Questo componente è lo *Status* (Stato), ovvero un semplice contenitore di un nuovo tipo di attributi. I nuovi attributi vengono chiamati *Status Attribute*. Ovviamente quest'estensione non modifica il comportamento nel caso di assenza di stato, di conseguenza la valutazione rimane inalterata rispetto a quella descrita precedentemente, mentre viene modificata nel caso in cui lo stato sia presente in modo da gestire correttamente la presenza di esso.

Analizziamo quindi, a scopo esemplificativo, il secondo caso, ovvero quando lo stato è presente. Inizialmente viene definito il sistema, che ora ha quattro componenti principali:

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

Fino al quarto step il comportamento è analogo a quello precedente, mentre cambia negli step successivi.

Al quinto step il *PDP* non necessiterà solo dei normali attributi d'ambiente, ma necessiterà anche degli *Status Attribute* coinvolti nella richiesta effettuata. Il *Context Handler* quindi non andrà solo a fare la ricerca all'interno dell'environment, ma andrà a cercare anche gli *Status Attribute* all'interno dello *Status*.

A questo punto, quando il PDP avrà tutte le informazioni necessarie si potrà passare alla vera e propria valutazione della richiesta che avviene come sempre.

Nel caso in cui viene restituto `Permit` o `Deny` è necessario fare l'enforcement della risposta del PDP. Questo processo differisce dal precedente poiché ora sono state implementate nuove azioni sullo stato che devono essere eseguite dal *PEP* (Passo 13-16). Una volta effettuato l'enforcement viene restituta la decisione finale.

Prendendo il primo esempio citato in sezione 2.2 la valutazione procederebbe in questo modo. Bob richiederà la lettura di un determinato file. Quindi la richiesta conterrà tre attributi, uno che indica il nome dell'utente che effettua la richiesta, il secondo che contiene il nome del file a cui si effettuerà l'accesso, e il terzo che conterrà il nome dell'azione da effettuare. La policy invece sarà strutturata come *"Se il nome è Bob, il file è corretto e nessuno sta scrivendo o ci sono meno di due utenti che leggono, allora permetti, altrimenti nega"* .

Il PDP però ha bisogno di più attributi per valutare la richiesta, in quanto necessita anche di attributi esterni alla policy che riguardano il numero di utenti che stanno accedendo al file richiesto, questi attributi sono gli *Status Attribute*. Per la loro gestione sarà necessario utilizzare la funzionalità che riguarda l'Usage Control, il PDP richiederà al *Context Handler* questi attributi, il quale andrà cercarli nello *Status*. Quest'ultimo li fornirà e verranno direttamente mandati al PDP per la valutazione della richiesta.

Se la richiesta avrà esito positivo, allora vuol dire che Bob avrà accesso al file, e quindi lo stato andrà aggiornato. La risposta del PDP a questo punto andrà al PEP per l'enforcement il quale avrà il compito di aggiornare lo stato. Sostanzialmente lo stato viene aggiornato semplicemente

incrementando l'attributo riguardante il numero di lettori di un'unità.

4.2 ESTENSIONE LINGUISTICA

Per implementare queste nuove funzionalità è stata modificata anche la grammatica di FACPL. Nella grammatica estesa sono state aggiunte nuove regole di produzione e simboli terminali che codificano le nuove funzionalità.

Come è facilmente osservabile dalla consultazione della tabella 3 le aggiunte rispetto alla tabella riporta in sezione 3.2 sono state diverse, vediamo adesso quali sono.

La prima modifica che risulta evidente è nel PAS, ovvero nella definizione del sistema. L'aggiunta è stata lo *Status*, ovvero un contenitore di attributi. Uno *Status* è della forma

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

questo significa che se lo *Status* è presente sarà formato da uno o più *Attribute*.

Passiamo ora a descrivere *Attribute* che è della forma

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

questo tipo particolare di attribute, che è lo *Status Attribute* descritto in precedenza, è formato innanzitutto da un *Type*, dopo il tipo è richiesta una generica stringa chiamata *Identifier*, che sarà un generico nome da dare all'attributo, infine viene richiesto un *Value*, ovvero un valore, che in questo caso è opzionale, all'atto pratico vuol dire che l'attributo di stato potrà essere inizializzato con un valore oppure potrà essere solamente definito, lasciando che il valore sia quello di default.

Type è il tipo che avrà l'attributo di stato, e potrà essere `int, boolean, date o float`.

La regola *PepAction* è stata modificata in modo tale che includesse nuove funzioni per operare matematicamente sugli attributi di stato. Queste nuove funzioni sono:

- *add(Attribute,* `int`*)*

- *add(Attribute,* `float`*)*
- *div(Attribute,* `int`*)*
- *div(Attribute,* `float`*)*
- *sub(Attribute,* `int`*)*
- *sub(Attribute,* `float`*)*
- *mul(Attribute,* `int`*)*
- *mul(Attribute,* `float`*)*
- *flag(Attribute,* `boolean`*)*
- *sumDate(Attribute,* `date`*)*
- *sumString(Attribute,* `string`*)*
- *setValue(Attribute,* `string`*)*
- *setDate(Attribute,* `date`*)*

Infine l'ultima regola di produzione modificata è stata quella riguardante *Attribute Names*, in questo caso è stata semplicemente aggiunto, a fianco di *Identifier/Identifier*, una nuova produzione *Status/Identifier*. Questa nuova produzione serve semplicemente per permettere il confronto tra attributi di stato attraverso le già esistenti *Expression*. La sintassi delle risposte è rimasta invariata. Vediamo ora un esempio di questa nuova sintassi, prenderemo spunto da un caso già trattato in precedenza nella sezione 4.1.

Codice 4.1: Esempio per la sintassi

```
 Policy example <  permit-overrides
  target: equal("Bob",name/id) &&  equal("read", action/id)
 rules:
    Rule access (
     permit  target: less-than(status/counter, 2))
    obl:
     [ permit M add(counter, 1)]
>

PAS {
 Combined Decision : false ;
 Extended Indeterminate : false ;
```

```
  Java Package : "example" ;
  Requests To Evaluate : Request_example ;
  pep:  deny- biased
  pdp:  deny- unless- permit
  status: [(int counter = 0)]
  include example
}
```

In questo esempio (Codice 4.1) si può vedere come nel PAS è stato definito uno stato, con al suo interno uno solo attributo inizializzato con valore 0. Successivamente si può notare nella *Rule* che viene fatto un controllo sul valore di quest'attributo. Infine nella *Obligation* si può notare come viene aggiornato lo stato dell'attributo in base al risultato della valutazione della *Rule*.

4.3 SEMANTICA

Sostanzialmente la semantica di FACPL rimane molto simile a quella descritta in sezione 3.3, quindi verranno di seguito descritte in modo informale solo le novità introdotte.

La prima di queste riguarda la valutazione delle richieste dal PDP. Il PDP ora non si deve più basare solo su richieste totalmente scollegate l'una dall'altra, e quindi è stato introdotto il concetto di *Status*. Lo stato serve per l'appunto per memorizzare il comportamento passato del sistema, e lo fa introducendo una nuova serie di attributi chiamati *Status Attribute*.

Nel linguaggio questo nuovo tipo di attributi viene considerato al pari di normali attributi, quindi si ha la possibilità di effettuare tutte le operazioni di confronto tra di essi, ma in più si deve avere la possibilità di modificarli e memorizzarli in modo da poterli sfruttare per *Usage Control*.

Gli *Status Attribute* vengono inizialmente definiti nel PAS, ed avranno un tipo, un identificatore ed eventualmente un valore. Per esempio uno *Status Attribute*definito in questo modo *(boolean isWriting = false)* codifica un attributo di stato di tipo booleano chiamato "isWriting" ed inizializzato a `false`.

Come detto prima bisogna avere la possibilità di modificare questi

nuovi attributi, e per questo sono state aggiunte delle *Pep Action*, ovvero delle azioni eseguite dal PEP in seguito alla valutazione di *Obligations*.

add(Attribute, int) e add(Attribute, float)

Queste due *Pep Actions* effettuano un'azione su un attributo di stato. Questa *Pep Action* permette, in seguito alla valutazione di una *Obligations*, l'aggiunta di un valore numerico ad uno *Status Attribute* di tipo `float` o `int`. Effettuare questa operazione su un tipo non permesso porterà ad un errore in fase di valutazione.

```
obl:
    [permit M add(counter, 2)]
```

Per esempio l'esecuzione di questa *Obligation* su un'attributo, chiamato *counter*, inizializzato a 0, porterà l'attributo al valore 2. Mentre l'esecuzione di questo

```
obl:
    [permit M add(counter, "foo")]
```

porterà ad un errore per incoerenza sul tipo.

sub(Attribute, int) e sub(Attribute, float)

Questo insieme *Pep Actions* è semplicemente il duale della precedente, ciò vuol dire che invece che effettuare la somma di un valore numerico ad uno *Status Attribute* farà una sottrazione.

Quindi, riprendendendo l'esempio citato per la add:

```
obl:
    [permit M sub(counter, 2)]
```

l'esecuzione corretta di questa *Obligation* porterà nuovamente l'attributo ad il suo valore originale, ovvero 0. Ovviamente riguardo gli errori vale lo stesso discorso fatto in precedenza.

div(Attribute, int) e div(Attribute, float)

Le *Pep Actions* di questa categoria effettuano una divisione tra un attributo ed un numero passatogli come parametro.

```
obl:
    [permit M div(number, 2)]
```

Presupponendo che il valore dell'attributo *number* sia inizialmente 6 l'esecuzione di questa operazione lo porterà a 6/2, ovvero 3.

mul(Attribute, int) e mul(Attribute, float)

Quest'azione, come facilmente intuibile dal nome effettua la moltiplicazione. Considerando l'esempio fatto per la div

```
obl:
    [permit M mul(number, 3)]
```

Visto che il valore dell'attributo, modificato in precedenza dalla div, è 3 l'operazione effettuata sarà $3 * 3$ che darà come risultato 9.

flag(Attribute, boolean)

Quest'operazione, definita solo sul tipo `boolean` modifica il valore di uno *Status Attribute* di tipo booleano.

```
obl:
    [permit M flag(isFoo, true)]
```

L'esecuzione con successo di questa *Obligation* porterà l'attributo *flag* ad avere un valore `true`.

sumDate(Attribute, date)

Anche questa azione, come quella di prima è definita su un solo tipo, ma questa volta il tipo è `Date`. Il tipo `Date` può essere espresso con in tre forme diverse

- `HH:mm:ss`
- `yyyy/MM/dd`
- `yyyy/MM/dd-HH:mm:ss`

Per fare un altro esempio consideriamo uno *Status Attribute* di tipo `date` inizializzato al giorno `2016/04/20` di nome *foo*.

```
obl:
    [permit M sumDate(foo, 24:00:00)]
```

La corretta esecuzione di questa *Obligation* porterà il valoer di *foo* al giorno successivo, ovvero il `21 Aprile 2016`.

sumString(Attribute, string)

L'operazione ora descritta coinvolge il tipo `string`. Passandogli un attributo di tipo `String` ed una stringa effettuerà la concatenazione.

```
obl:
    [permit M sumString(Pablo, " Neruda")]
```

Se `Pablo` avesse il valore *"Pablo"*, il risultato di questa operazione farebbe cambiare valore all'attributo in *"Pablo Neruda"*.

setValue(Attribute, string)

Quest'operazione è molto simile alla precedente, ma invece che concatenare le stringhe sostituisce il valore presente dentro l'attributo con quello passatogli come parametro.

```
obl:
    [permit M setValue(Pablo, "Aghiò Aghiò")]
```

Quindi l'esecuzione corretta di quest'azione porterà l'attributo che precedentemente aveva il valore *"Pablo Neruda"* ad avere il valore *"Aghiò Aghiò"*.

setDate(Attribute, date)

Quest'operazione è uguale alla precedente, ma invece che operare sul tipo stringa opera sul tipo Data.

```
obl:
    [permit M setDate(foo, 2016/12/25)]
```

L'esecuzione di questa *Obligation* modificherà il valore di *foo* al 25 Dicembre 2016.

4.4 ESEMPI

Queste nuove funzionalità introdotte servono allo scopo descritto in sezione 2.2, ovvero l'implementazione di un nuovo modello chiamato *Usage Control*. Mostreremo ora l'implementazione in FACPL dei due esempi trattati in sezione 2.2.

4.4.1 *Accesso ai file*

Codice 4.2: Primo esempio

```
PolicySet ReadWrite_Policy {  deny- unless- permit
 target:  equal ( "Bob" , name / id ) && ("Alice, name/id")
 policies:
 PolicySet Write_Policy {  deny- unless- permit
  target: equal("file", file/id) && ("write", action/id)
  policies:
   Rule write (  permit  target:
    equal ( status / isWriting , false ) &&
    equal ( status / counterReadFile1, 0)
   )
  obl:
 [  permit M  flagStatus(isWriting, true) ]
}
 PolicySet Read_Policy {  deny- unless- permit
  target: equal("file", file/id) && ("read", action/id)
  policies:
   Rule write (  permit  target:
    equal ( status / isWriting , false ) &&
   less-than (status / counterReadFile1, 2)
   )
  obl:
 [  permit M  addStatus(counterReadFile1, 1) ]
}
 PolicySet StopWrite_Policy {  deny- unless- permit
  target: equal("file", file/id) && ("stopWrite", action/id)
  policies:
   Rule write (  permit  target:
    equal ( status / isWriting , true )
   )
  obl:
 [  permit M  flagStatus(isWriting, false) ]
```

```
 }
  PolicySet StopRead_Policy {  deny- unless- permit
   target: equal("file", file/id) && ("stopRead", action/id)
   policies:
    Rule write (  permit  target:
     greater-than ( status / counterReadFile1 , 0 )
     )
   obl:
  [  permit M  subStatus(counterReadFile1, 1) ]
 }
}
PAS {
 Combined Decision : false ;
 Extended Indeterminate : false ;
 Java Package : "example" ;
 Requests To Evaluate : Request1, Request2, Request3, Request4,
    Request5, Request6 ;
 pep:  deny- biased
 pdp:  deny- unless- permit
 status: [(boolean isWriting = false), (int counterReadFile1 = 0), ]
 include example
}
```

Per semplicità, in questo codice, vengono mostrate le policy di accesso per un solo file. L'esempio citato in precedenza metteva una regola sull'accesso ai file, ovvero permetteva un massimo di due persone in contemporanea che potevano effettuare l'accesso in lettura oppure un massimo di una persona che poteva ottenere l'accesso in scrittura.

Tutte le policy del codice 4.2 sono chiuse in un *PolicySet*. Il target del PolicySet `ReadWrite_Policy` verifica che le richieste provengano da utenti che hanno nome *Alice* o *Bob*, in caso contrario il responso sarà `Not Applicable`.

Successivamente sono state create quattro policy per gestire le quattro operazioni possibili, ovvero `read`, `write`, `stopRead` ed infine `stopWrite`.

Prendiamo in considerazione la prima policy, quella per la `write`. Come prima è presente un target, che richiede questa volta due diverse condizioni, la prima riguarda l'id del file richiesto, la seconda invece richiede che l'azione sia `write`. Le parti interessanti di questa policy sono

due, la prima riguarda la *Rule*, la seconda la *Obligation*.

La *Rule* restituisce `permit` se le due condizioni dell'equal sono vere, come si può facilmente notare l'operazione di confronto non viene fatta tra una stringa ed un normale attributo, ma tra una stringa ed uno *Status Attribute*.

L'ultima cosa da notare è l'unica *Obligation* presente per questa policy. Questo tipo particolare di *Obligation*, ha sempre al suo interno un'azione che verrà eseguita dal PEP, ma questa volta l'azione andrà a modificare lo stato del sistema, mettendo il valore `true` all'attributo *isWriting*.

Prendiamo ora una serie di richieste.

Codice 4.3: Richieste del primo esempio

```
Request:{ Request1
  (name / id , "Alice")
  (action / id, "read")
  (file / id, "file1")
}

Request:{ Request2
  (name / id , "Bob")
  (action / id, "Write")
  (file / id, "file1")
}

Request:{ Request3
  (name / id , "Bob")
  (action / id, "read")
  (file / id, "file1")
}
Request:{ Request4
  (name / id , "Alice")
  (action / id, "stopRead")
  (file / id, "file1")
}

Request:{ Request4
  (name / id , "Bob")
  (action / id, "stopRead")
  (file / id, "file1")
```

```
}

Request:{ Request6
  (name / id , "Alice")
  (action / id, "write")
  (file / id, "file1")
}
```

L'output di queste richieste sarà il sguente

```
Request1:
 PDP Decision=
   Decision: PERMIT
   Obligations: PERMIT M AddStatus([INT/counterReadFile1/0, 1])
 PEP Decision= PERMIT
Request2:
 PDP Decision=
   Decision: DENY Obligations:
 PEP Decision= DENY
Request3:
 PDP Decision=
   Decision: PERMIT
   Obligations: PERMIT M AddStatus([INT/counterReadFile1/1, 1])
 PEP Decision= PERMIT
Request4:
 PDP Decision=
   Decision: PERMIT
   Obligations: SubStatus([INT/counterReadFile1/2, 1])
 PEP Decision= PERMIT
Request5
 PDP Decision=
   Decision: PERMIT
   Obligations: PERMIT M SubStatus([INT/counterReadFile1/1, 1])
 PEP Decision= PERMIT
Request6:
 PDP Decision=
   Decision: PERMIT
   Obligations: PERMIT M FlagStatus([BOOLEAN/isWriting/false, true])
 PEP Decision= PERMIT
```

Analizziamo ora il motivo di queste decisioni. Nella prima richiesta ovviamente nessuno sta leggendo o scrivendo, quindi viene tranquillamente

restituito permit, visto che è presente una *obligation* lo stato verrà aggiornato, sommando un'unità al contatore di letture. Alla seconda richiesta l'utente richiede la scrittura, che gli viene negata perché c'è già qualcuno che sta leggendo, però lo stesso utente effettua un'altra richiesta, questa volta in lettura, che gli viene concessa.

La quarta e la quinta richiesta vengono fatte per avvisare il sistema che la lettura è terminata, ovviamente la risposta è permit, e la *obligation* corrispondente decrementerà il contatore. La sesta ed ultima richiesta è una scrittura, che questa volta viene permessa, poiché nessuno sta scrivendo o leggendo.

4.4.2 *Noleggio e acquisto di contenuti*

In questo secondo esempio analizzeremo il caso di un'azienda di distribuzione di contenuti multimediali che vuole regolare l'accesso di quest'ultimi attraverso policy. Faremo un breve esempio con un solo file e due utenti, uno dei due utenti comprerà il file, l'altro lo noleggierà a tempo determinato. Nel codice 4.4 vengono mostrate solo una parte delle policy presenti nel codice completo.

Codice 4.4: Secondo Esempio

```
PolicySet Negozio {  deny- unless- permit
  target:  equal ( "Bob" , name / id ) || ("Alice, name/id")
  policies:

  PolicySet Buy_Policy {  deny- unless- permit
    target: equal("file1", file/id) && ("buy", action/id)
    policies:
      Rule alice_buy (  permit  target: (
        equal ( action / id , "buy" ) &&
        equal ( name / id, "Alice"))
        obl:
         [  permit M setString("accessTypeAlice", "BUY") ]
      )
      Rule bob_buy (  permit  target: (
        equal ( action / id , "buy" ) &&
        equal ( name / id, "Alice"))
        obl:
         [  permit M setString("accessTypeBob", "BUY") ]
      )
```

```
}

 PolicySet NUMBER_Policy {  deny- unless- permit
  target: equal("file1", file/id) && ("number", action/id)
  policies:
   Rule alice_buy (  permit  target: (
    equal ( action / id , "number" ) &&
    equal ( name / id, "Alice"))
    obl:
     [  permit M setString("accessTypeAlice", "NUMBER") ]
     [  permit M  addStatus("aliceFile1viewNumber", 2) ]
   )
   Rule bob_buy (  permit  target: (
    equal ( action / id , "number" ) &&
    equal ( name / id, "Alice"))
    obl:
     [  permit M setString("accessTypeBob", "NUMBER") ]
     [  permit M  addStatus("bobFile1viewNumber", 2) ]
   )
}
```

Queste due policy, e anche le altre che non sono state mostrate, sono racchiuse tutte all'interno del *Policy Set* Negozio il quale come prima cosa verifica se chi ha fatto la richiesta ha un determinato nome, in questo caso *Bob* o *Alice*.

Successivamente, se uno dei due effettua la richiesta di `BUY`, ovvero l'acquisto senza alcun tipo di limitazione, si entra nella prima policy e, tramite le *Obligation* si cambia l'attributo di stato. Invece se un utente decidesse di effettuare il noleggio con un numero limitato di volte si entra nella seconda *Policy Set* la quale, attraverso *Obligations* aumenterà il numero di visioni di due unità. Analogo è il caso del noleggio a tempo.

Per disciplinare la visione è presente un altro *Policy Set*, mostrato anch'esso parzialmente in codice 4.5.

Codice 4.5: Secondo Esempio

```
 PolicySet VIEW {  deny- unless- permit
  target: equal("file1", file/id) && ("view", action/id)
  policies:
   Rule buy (  permit  target: (
    equal ( status / accessTypeBob , "BUY" ) &&
```

```
      equal ( status / accessTypeAlice, "BUY"))
     )
    Rule number_alice (  permit  target: (
     equal ( status / accessTypeAlice, "NUMBER" ) &&
     equal ( name / id, "Alice") && greater-than( status /
       aliceFile1viewNumber, 0))
     obl:
      [  permit M  subStatus("aliceFile1viewNumber", 1) ]
    )
```

Mostriamo ora in Codice 4.6 alcune richieste che possono essere fatte al sistema ed analizziamo le risposte che produrranno.

Codice 4.6: Richieste del Secondo Esempio

```
Request:{ Request1
  (name / id , "Alice")
  (action / id, "view")
  (file / id, "file1")
}

Request:{ Request2
  (name / id , "Bob")
  (action / id, "view")
  (file / id, "file1")
}

Request:{ Request3
  (name / id , "Alice")
  (action / id, "Buy")
  (file / id, "file1")
}
Request:{ Request4
  (name / id , "Alice")
  (action / id, "view")
  (file / id, "file1")
}

Request:{ Request4
  (name / id , "Bob")
  (action / id, "Time")
  (file / id, "file1")
}
```

```
Request:{ Request6
  (name / id , "Bob")
  (action / id, "view")
  (file / id, "file1")
}
```

La prima e la seconda richiesta sono richieste di visione, che ovviamente restituiranno entrambe deny, in quanto nessun utente ha effettuato acquisti o noleggi. Successivamente Alice effettuerà un acquisto ed una visione ed entrambi andranno a buon fine. A questo punto Bob, a cui prima era stata negata la visione effettuerà una richiesta di noleggio e dopo una richiesta di visione, il risultato di entrambe sarà permit.

Tabella 3: Syntax of FACPL

Policy Authorisation Systems	*PAS* ::=	(pep : *EnfAlg* pdp : *PDP* (status : [*Attribute*]$^+$)*)
Attribute	*Attribute* ::=	(*Type Identifier* (= Value)$^?$)
Type	*Type* ::=	int | boolean | date | float
Enforcement algorithms	*EnfAlg* ::=	base | deny-biased | permit-biased
Policy Decision Points	*PDP* ::=	{*Alg* policies : *Policy*$^+$}
Combining algorithms	*Alg* ::=	p-over$_\delta$ | d-over$_\delta$ | d-unless-p$_\delta$ | p-unless-d$_\delta$ | first-app$_\delta$ | one-app$_\delta$ | weak-con$_\delta$ | strong-con$_\delta$
fulfilment strategies	δ ::=	greedy | all
Policies	*Policy* ::=	(*Effect* target : Expr obl : *Obligation**) | {*Alg* target : Expr policies : *Policy*$^+$ obl : *Obligation** }
Effects	*Effect* ::=	permit | deny
Obligations	*Obligation* ::=	[*Effect ObType* PepAction(Expr*)]
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*)$^+$

5

ESTENSIONE DELLA LIBRERIA FACPL

Il linguaggio FACPL è basato interamente su una libreria scritta in Java. Per implementare la valutazione di richieste basate sul comportamnento passato è stato necessario estendere questa libreria con nuove classi e modificarne alcune.

In questo capitolo verranno mostrate le novità introdotte nel capitolo 4 sotto il punto di vista implementativo, per ovvi motivi verranno mostrate solo alcune parti delle modifiche effettuate, ma il codice completo si può comunque trovare su GitHub.

5.1 STATUS E STATUS ATTRIBUTE

Il primo passo per estendere la libreria è stato la creazione di uno *Status*, che è modellato da una semplice classe di cui ne verrà mostrato un pezzo nel codice 5.1.

Codice 5.1: Stralcio della classe Status

```
public class FacplStatus {
  private List<StatusAttribute> attributeList;
  private String statusID;
  public FacplStatus(String statusID) {
    attributeList = new ArrayList<StatusAttribute>();
    this.statusID = statusID;
  }
  public FacplStatus(List<StatusAttribute> attributeList, String
      statusID) {
    this.attributeList = attributeList;
    this.statusID = statusID;
  }
  public FacplStatus() {
```

```
    attributeList = new ArrayList<StatusAttribute>();
    this.statusID = UUID.randomUUID().toString().substring(0, 8);
  }
```

Questa classe ha un campo essenziale per la logica del sistema, ed è una LinkedList di *Status Attribute*. In questa classe, oltre ad i costruttori ed alcuni getter sono stati implementati due metodi mostrati in Codice 5.2, uno per andare a cercare lo *Status Attribute*, e l'altro per restituirne il valore.

Codice 5.2: Metodi per gli *Status Attribute*

```
  public StatusAttribute retrieveAttribute(StatusAttribute
      attribute) throws MissingAttributeException {
    int i = this.attributeList.indexOf(attribute);
    if (i != -1) {
      return this.attributeList.get(i);
    } else {
      throw new MissingAttributeException("attribute doesn't exist in the
          current status");
    }
  }
  public Object getValue(StatusAttribute attribute) throws
      MissingAttributeException {
    return (Object) (this.retrieveAttribute(attribute).getValue());
  }
```

Gli *Status Attribute*sono modellati da una singola classe, anch'essa molto breve e facile da capire. Come facilmente intuibile dai costruttori in Codice 5.3 questa classe ha tre campi, un `id`, un valore, ed un tipo.

Codice 5.3: Costruttori di Status Attribute

```
  public StatusAttribute(String id, FacplStatusType type) {
    this.id = id;
    this.type = type;
    if (type == (FacplStatusType.INT) || type ==
        (FacplStatusType.DOUBLE)) {
      value = "0";
    } else if (type == FacplStatusType.BOOLEAN) {
      value = "false";
    } else if (type == FacplStatusType.DATE) {
      value = "0";
    } else {
```

```
        value = "";
12    }
  }
14  public StatusAttribute(String id, FacplStatusType type, String
      value) {
    this.id = id;
16    this.type = type;
    this.value = value;
18  }
```

Il senso del secondo costruttore è facilmente intuibile, mentre il primo è stato creato appositamente per dare un valore di default all'attributo nel caso non venisse inizializzato.

Di seguito è mostrato un grafico UML che mostra interamente queste due classi e la relazione che intercorre tra di loro.

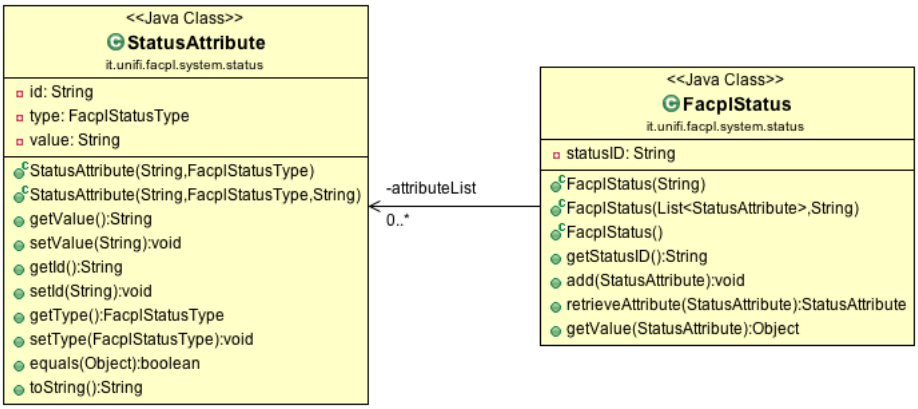

Figura 10: Grafico UML delle classi Status e StatusAttribute

5.2 IMPLEMENTAZIONE DEI COMPARATORI SUGLI STATUS ATTRIBUTE

Nella libreria di FACPL era già presente una solida struttura, basata su un factory, per la comparazione di attributi, quindi è bastato modificare le varie funzioni in modo tale che potessero operare anche su *Status Attribute*. Prendendo in esame la funzione *Equals* vediamo come funziona ora il processo di comparazione.

Codice 5.4: Classe che implementa Equal

```
public class Equal implements IComparisonFunction {
```

```
  public Boolean evaluateFunction(List<Object> args) throws
      Throwable {
    if (args.size() == 2) {
      Object o1, o2;
      o1 = args.get(0) instanceof StatusAttribute ?
          this.convertType((StatusAttribute) args.get(0)) :
          args.get(0);
      o2 = args.get(1) instanceof StatusAttribute ?
          this.convertType((StatusAttribute) args.get(1)) :
          args.get(1);
      IComparisonEvaluator evaluator =
          ComparisonEvaluatorFactory.getInstance().getEvaluator(o1);
      return evaluator.areEquals(o1, o2);
    } else {
      throw new Exception("Illegal number of arguments");
    }
  }
  private Object convertType(StatusAttribute sa) {
    if (sa.getType() == FacplStatusType.BOOLEAN) {
      if (sa.getValue() == "true") {
        return true;
      } else
        return false;
    } else if (sa.getType() == FacplStatusType.DOUBLE) {
      return (Double)Double.parseDouble(sa.getValue());
    } else if (sa.getType() == FacplStatusType.INT) {
      return (Integer)Integer.parseInt(sa.getValue());
    } else if (sa.getType() == FacplStatusType.STRING) {
      return sa.getValue();
    }
    return null;
  }
}
```

Vedendo il codice 5.4 si nota che la classe implementa un interfaccia, quest'interfaccia definisce al suo interno un unico metodo astratto, *public Boolean evaluateFunction(List<Object> args)* che sarà il metodo chiamato in fase di valutazione. La modifica del processo di comparazione è stata fatta in questo metodo, bisognava fare in modo che uno, o entrambi gli argomenti, potessero essere *Status Attribute*, e per questo è stata introdotta un'altra funzione, chiamata *convertType* che, dato uno *Status Attribute*, va a ricavarne il valore.

Quando verrà richiamato il primo metodo verrà effettuato un controllo sul tipo dell'argomento, e in base a questo risultato verrà chiamato il secondo metodo che effettuerà l'operazione descritta in precedenza.

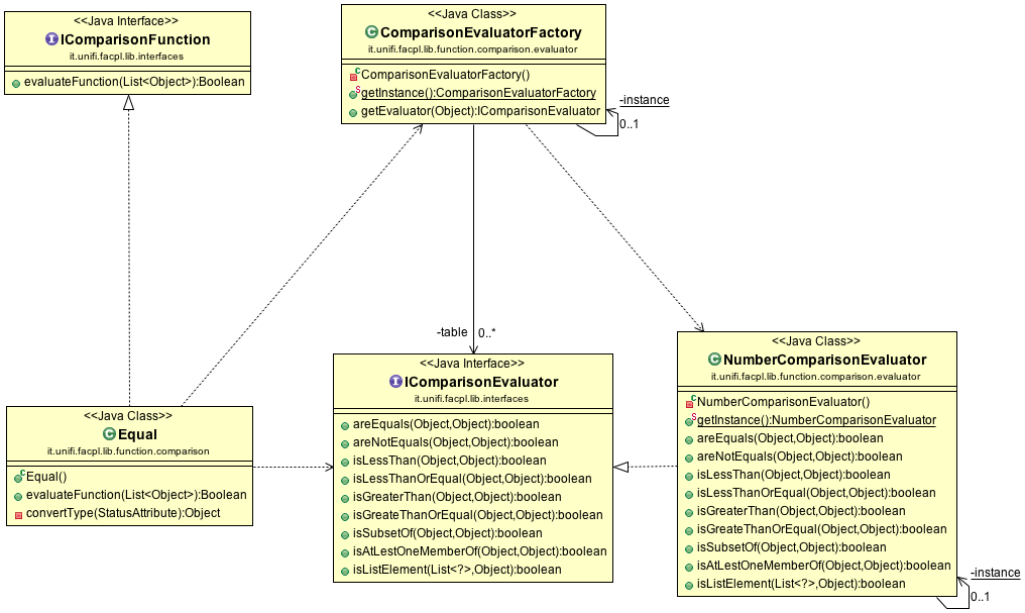

Figura 11: Grafico UML per la gerarchia di classi usate nella comparazione

5.3 FUNZIONI PER LA MODIFICA DEGLI STATUS ATTRIBUTE

Per concetto di *Status Attribute* è richiesto dinamismo, in quanto devono irrimediabilmente cambiare rispetto alla valutazione di una richiesta, quindi sono state implementate funzioni che effettuano queste operazioni di modifica.

Per mantenere la coerenza con il resto della libreria queste funzioni sono state implementate in modo simile a come sono state implementate quelle di comparazione. Ora verrà mostrato il caso di operazioni numeriche. Sono state implementate anche operazioni su stringhe, ma il funzionamento è analogo alla sua controparte numerica.

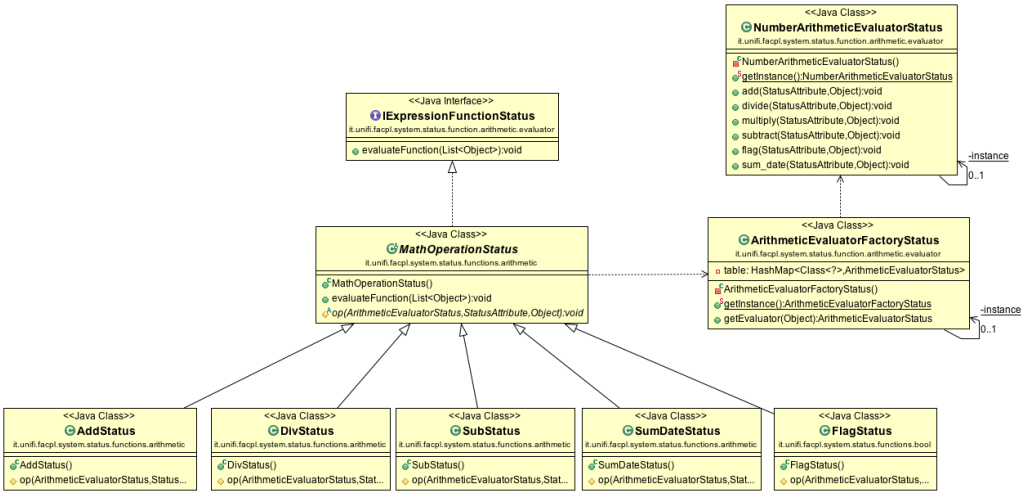

Figura 12: Grafico UML per la gerarchia di funzioni aritmetiche

Tutto parte da un interfaccia, che dovrà essere implementata da tutte le funzioni che andranno a modificare lo stato, quest'interfaccia è composta da un solo metodo che verrà chiamato per l'esecuzione dell'operazione.

Codice 5.5: Interfaccia per le operazioni

```
public interface IExpressionFunctionStatus {

  public void evaluateFunction(List<Object> args) throws Throwable;

}
```

In questo caso, da quest'interfaccia deriva una classe astratta per le operazioni aritmetiche che implementerà il metodo astratto dell'interfaccia ed aggiungerà un altro metodo astratto, che rappresenterà la funzione vera e propria.

Codice 5.6: Metodo implementato dall'interfaccia

```
  public void evaluateFunction(List<Object> args) throws Throwable {
    if (args.size() == 2) {
      StatusAttribute s1;
      if (args.get(0) instanceof StatusAttribute) {
        s1 = (StatusAttribute) args.get(0);
      } else {
        throw new Exception("First argument it's not a Status Attribute");
      }
      Object o2 = args.get(1);
```

```
        ArithmeticEvaluatorStatus evaluator =
            ArithmeticEvaluatorFactoryStatus.getInstance().getEvaluator(o2);
        op(evaluator, s1, o2);
      } else {
        throw new Exception("Illegal number of arguments");
      }
    }
```

Dal questo codice si notano subito molte somglianze con quello proposto in sezione 5.2, questo perché la logica di funzionamento è sostanzialmente la stessa. Inizialmente viene effettuato un controllo sul tipo degli argomenti, successivamente viene richiesto un valutatore corretto per il tipo di dato passatogli e poi è effettuata l'operazione chiamando il metodo *op*, che sfruttando il principio del pattern Template, è implementata nelle varie sottoclassi.

Le classi che estendono *MathOperationStatus* sono molto simili tra di loro, quindi prenderemo in esame solo la classe che effettua l'operazione di somma.

Codice 5.7: Classe per la somma

```
public class AddStatus extends MathOperationStatus {

  @Override
  protected void op(ArithmeticEvaluatorStatus ev, StatusAttribute
      s1, Object o2) throws Throwable {
    ev.add(s1, o2);
  }
}
```

Questa classe implementa semplicemente il metodo astratto, chiamando sul valutatore passatogli in precedenza la funzione di somma.

Il valutatore non è altro che una classe che implementa tutte le operazioni di una determinata categoria, per esempio in questo caso viene restituito un valutatore che effettua le operazioni aritmetiche. A puro scopo esemplificativo mostriamo ora come è implementata l'operazione di somma nel valutatore aritmetico.

Codice 5.8: Metodo implementato dall'interfaccia

```
  public void add(StatusAttribute o1, Object o2) throws Throwable {
```

```
    if (o1.getType() == FacplStatusType.INT) {
      Integer value = Integer.parseInt(o1.getValue());
      Integer newValue = value + (int) o2;
      o1.setValue(newValue.toString());
    } else if (o1.getType() == FacplStatusType.DOUBLE) {
      Double value = Double.parseDouble(o1.getValue());
      Double newValue = value + (double) o2;
      o1.setValue(newValue.toString());
    } else {
      throw new UnsupportedTypeException("Number", "Add");
    }
  }
```

5.4 ESTENSIONE DEL CONTESTO

Lo *Status* creato in 5.1 andrà in qualche modo preso in considerazione durante il processo di valutazione in modo che vada ad influenzare le decisioni. Di conseguenza lo stato dev'essere inglobato dal contesto in cui viene valutata la richiesta.

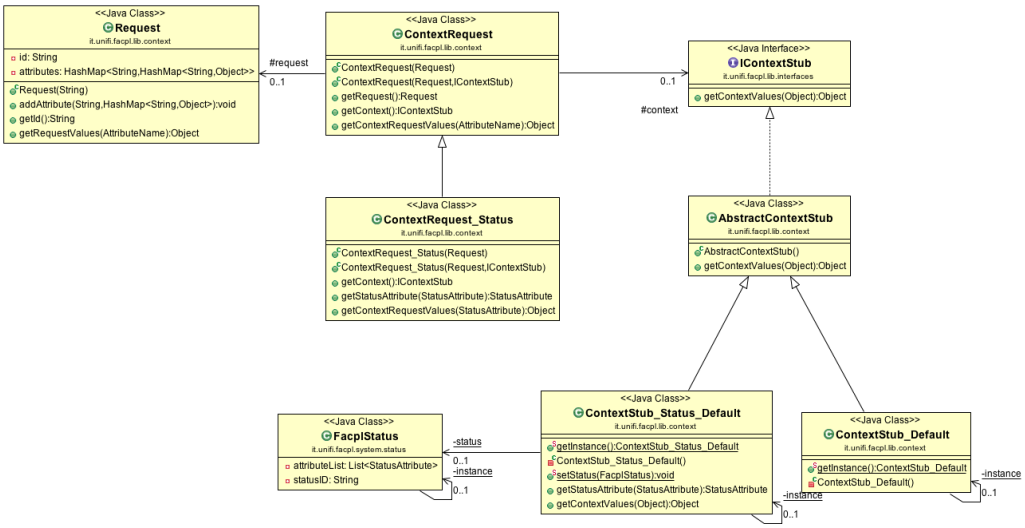

Figura 13: Grafico UML del contesto

Per prima cosa è stato necessario estendere la gerarchia di classi derivanti dall'interfaccia *IContextStub*. Come si può vedere in figura 13 è stata fatta un'operazione di refactoring astraendo alcune parti in comune, ed è stata creata la classe *ContextStub_Default_Status* di cui ora ne verrà

analizzata l'implementazione.

Come si nota dall'immagine 13 la classe è implementata come un *Singleton* e contiene diversi metodi legati a *Status*, uno di questi è il *Setter*, che permette di aggiungere lo stato, l'altro invece è un *Getter* che permette di ricavare un attributo dallo stato. Il metodo più importante invece è quello mostrato in Codice 5.9

Codice 5.9: Classe ContextStub_Default_Status

```
  public Object getContextValues(Object attr) {
    try {
      if (attr instanceof StatusAttribute) {
        return status.getValue((StatusAttribute) attr);
      } else {
        return super.getContextValues(attr);
      }
    } catch (MissingAttributeException e) {
      return null;
    }
  }
}
```

Questo metodo permette di effettuare la ricerca mostrata in figura 9, ovvero datogli un attributo andrà prima a verificare la sua presenza all'interno dello *Status*, dopodiché se non lo trova verificherà la presenza all'interno dell'ambiente.

Successivamente è stata estesa anche alla classe *ContextRequest* con una nuova classe *ContextRequest_Status*, la cui unica differenza è un semplice *Getter* per gli attributi di stato.

5.5 OBLIGATIONS E PEP

Lo stato alla necessità andrà aggiornato, e qua entrano in gioco due componenti fondamentali del sistema, il PEP e le *Obligations*.

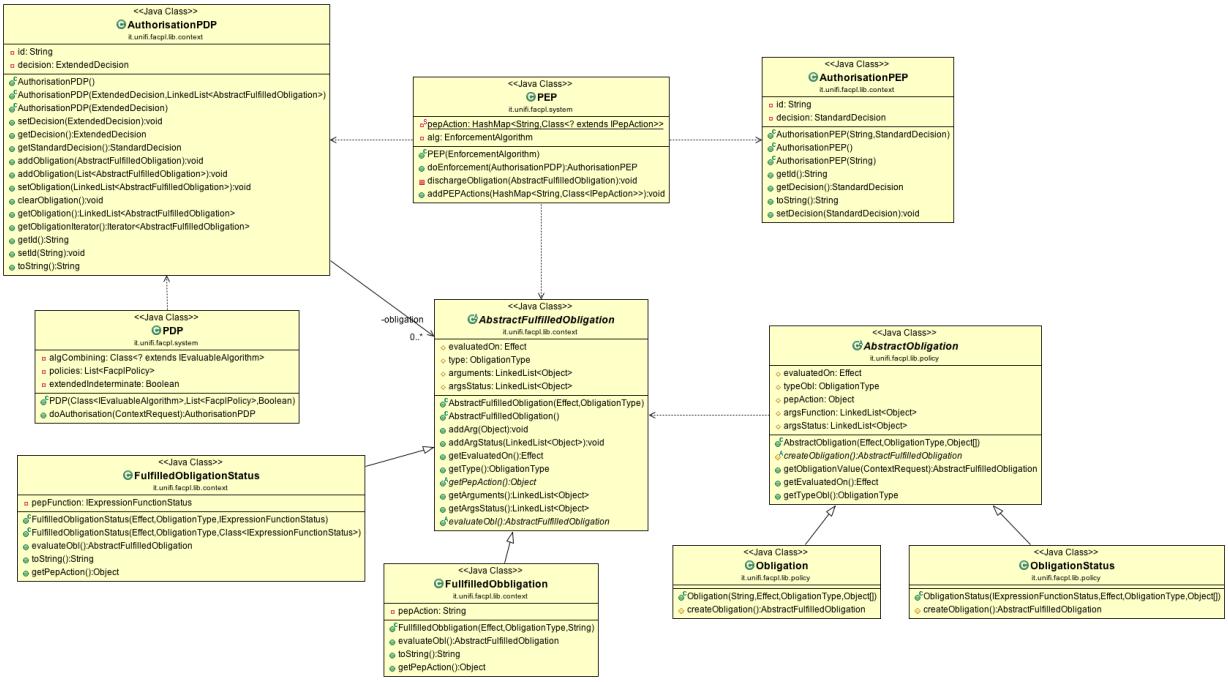

Figura 14: Relazioni tra Obligation e PEP

Anche in questo caso sono state estese le *Obligation* introducendo un nuovo tipo chiamato *Obligation Status*, questo tipo particolare di *Obligation* servono per andare ad eseguire azioni sullo stato. Nel sistema sono presenti due tipo fondamentali di *Obligation*, il primo sono quelle a livello sintattico, le seconde, chiamate *FulfilledObligations* sono quelle pronte ad essere valutate. Vediamo adesso come sono state estese quelle a livello sintattico.

Per eseguire questa estensione è stato reso necessario un refactoring, per prima cosa è stato astratto tutto il comportamento comune in una superclasse astratta, successivamente è stata creata la nuova classe che modella questo nuovo tipo. Il refactoring ha coinvolto anche il metodo che si occupa del *Fulfilling* delle *Obligation* in quanto ora deve creare anche questo nuovo tipo, la scelta più ovvia è stata creare un metodo astratto implementato nelle due sottoclassi che viene chiamato dalla superclasse per creare il tipo corretto.

Codice 5.10: Metodo che si occupa del fulfilling

```
protected abstract AbstractFulfilledObligation createObligation();
@Override
```

```
public AbstractFulfilledObligation
    getObligationValue(ContextRequest cxtRequest) throws
    FulfillmentFailed {
  Logger l = LoggerFactory.getLogger(Obligation.class);
  l.debug("Fulfilling Obligation " + this.pepAction.toString() + "...");
  AbstractFulfilledObligation obl = this.createObligation();
  if (obl instanceof FulfilledObligationCheck) {
    l.debug("...created FulfilledObligationCHECK: " + obl.toString());
    return obl;
  }
  for (Object arg : argsFunction) {
    if (arg instanceof ExpressionFunction) {
      Object res = ((ExpressionFunction)
          arg).evaluateExpression(cxtRequest);
      if (res.equals(ExpressionValue.BOTTOM) ||
          res.equals(ExpressionValue.ERROR)) {
        throw new FulfillmentFailed();
      }
      obl.addArg(res);
    } else if (arg instanceof ExpressionBooleanTree) {
      ExpressionValue res = ((ExpressionBooleanTree)
          arg).evaluateExpressionTree(cxtRequest);
      if (res.equals(ExpressionValue.BOTTOM) ||
          res.equals(ExpressionValue.ERROR)) {
        throw new FulfillmentFailed();
      }
      obl.addArg(res);
    } else if (arg instanceof AttributeName) {
      try {
        obl.addArg(cxtRequest.getContextRequestValues((AttributeName)
            arg));
      } catch (MissingAttributeException e) {
        throw new FulfillmentFailed();
      }
    } else {
      obl.addArg(arg);
    }
  }
  l.debug("...fulfillment completed. Arguments: " +
      obl.getArguments().toString());
  return obl;
}
```

Codice 5.11: CreateObligation nelle status

```
protected AbstractFulfilledObligation createObligation() {
  AbstractFulfilledObligation obl = new
      FulfilledObligationStatus(this.evaluatedOn, this.typeObl,
       (IExpressionFunctionStatus) this.pepAction);
  if (!argsStatus.isEmpty()) {
    obl.addArgStatus(argsStatus);
  }
  return obl;
}
```

Codice 5.12: CreateObligation nelle normali

```
@Override
protected AbstractFulfilledObligation createObligation() {
  return new FullfilledObbligation(this.evaluatedOn,
      this.typeObl, (String) this.pepAction);
}
```

La *Obligation* di stato necessiterà anche di argomenti su cui eseguire l'azione, che le verranno passati in fase di costruzione.

Alla fine della valutazione il PDP crea un oggetto di tipo *AuthorisationPDP* che conterrà la decisione e una lista di *FulfilledObligation*, quest'ultime poi andranno al PEP per la loro valutazione. Vediamo ora come sono state implementate.

Anche in questo caso è stato necessario un refactoring analogo a quello fatto per le prime.

Codice 5.13: Peculiarità della classe FulfilledObligationStatus

```
public class FulfilledObligationStatus extends
   AbstractFulfilledObligation {
  private IExpressionFunctionStatus pepFunction;
  public FulfilledObligationStatus(Effect effect, ObligationType
      typeObl, IExpressionFunctionStatus pepFunction) {
    super(effect, typeObl);
    this.pepFunction = pepFunction;
  }
  public FulfilledObligationStatus(Effect effect, ObligationType
      typeObl,
       Class<? extends IExpressionFunctionStatus> pepFunction) {
```

```
    super(effect, typeObl);
10  }
    @Override
12  public AbstractFulfilledObligation evaluateObl() throws Throwable
        {
      this.pepFunction.evaluateFunction(this.getArgsStatus());
14    return this;
    }
```

Come si può notare, in fase di costruzione, gli verrà passato un oggetto di tipo *IExpressionFunctionStatus* che sarà l'azione che andrà a eseguire sullo stato. Quest'azione andrà realmente ad essere eseguita quando verrà chiamato dal PEP il metodo *evaluateObl*.

Il PEP nella fase di enforcement effettua la valutazione delle *Obligation*, in questo caso le modifiche per permettere al sistema di eseguirle sono state minime, è bastato modificare il metodo *DischargeObligation* in modo che quando gli viene passata una *AbstractFulfilledObligation* chiamasse il metodo *evaluateObl* (Righe 22-25 di codice 5.14).

Codice 5.14: DischargeObligation

```
1   private void dischargeObligation(AbstractFulfilledObligation obl)
        throws Throwable {
      Logger l = LoggerFactory.getLogger(PEP.class);
3     if (obl instanceof FullfilledObbligation) {
        obl = (FullfilledObbligation) obl;
5       try {
          Class<? extends IPepAction> classAction =
              pepAction.get((String) obl.getPepAction());
7         if (classAction == null) {
            l.debug("Undefined PEP action \"" + (String)
                obl.getPepAction() + "\"");
9           throw new Exception("Undefined " + (String)
                obl.getPepAction() + " PEP Action");
          }
11        Class<?> params[] = new Class[1];
          params[0] = List.class;
13        Method eval = classAction.getDeclaredMethod("eval", params);
          Object pepAction = classAction.newInstance();
15        eval.invoke(pepAction, obl.getArguments());
        } catch (Throwable t) {
17        if (obl.getType().equals(ObligationType.M)) {
```

```
            throw t;
19        }
          l.debug("Exception ignored. Obligation is optional");
21      }
    }
23  else if (obl instanceof FulfilledObligationStatus) {
      obl = (FulfilledObligationStatus) obl;
25    obl.evaluateObl();
    }
27
}
```

5.6 ESEMPIO

[TODO: CORREGGERE PARTE SU XTEND E MAGARI ALLUNGARE] Il progetto è basato su Java, ed il codice formalizzato in FACPL viene successivamente convertito in codice Java attraverso un processo di autogenerazione. Per fare questo si usa un linguaggio chiamato XTend.

Figura 15: Logo di Xtend

XTend ha le sue radici in Java, ma si concentra maggiormente su aspetti come una sintassi più concisa, ed altre funzionalità come l'inferenza sui tipi, l'overload degli operatori o l'estensione dei metodi (?). È principalmente un linguaggio *Object-oriented*, ma integra caratteristiche tipiche di un linguaggio funzionale, come ad esempio le Lambda Expression. Il sistema dei tipi di XTend è lo stesso di Java, ed è quindi statico.

In questa sezione verrà mostrato come sarà in Java il corrispettivo del Codice 4.2 e del Codice 4.3.

Tutto parte dalla classe dove è presente il metodo Main (Codice 5.15. In questo stesso modulo viene preparato il sistema all'esecuzione attraverso l'inizializzazione di tutti i componenti necessari al sistema.

I due componenti più importanti sono il PDP e il PEP che vengono inizializzati nel costruttore, in quest'ultimo vengono anche aggiunte le

Policy e creato un contesto. Successivamente all'inizializzazione verranno inserite tutte le richieste in una lista e partirà un ciclo che effettuerà l'operazione di autorizzazione del PDP e di enforcement del PEP.

Codice 5.15: Main di uno scenario FACPL

```
public class MainFACPL {
  private PDP pdp;
  private PEP pep;
  public MainFACPL() throws MissingAttributeException {
    LinkedList<FacplPolicy> policies = new
        LinkedList<FacplPolicy>();
    policies.add(new
        PolicySet_ReadWrite(ContextRequest_WriteRequestAlice.getContextReq()));
    this.pdp = new
        PDP(it.unifi.facpl.lib.algorithm.PermitUnlessDenyGreedy.class,
        policies, false);
    this.pep = new PEP(EnforcementAlgorithm.DENY_BIASED);
    this.pep.addPEPActions(PEPAction.getPepActions());
  }
  public static void main(String[] args) throws
      MissingAttributeException {
    MainFACPL system = new MainFACPL();
    StringBuffer result = new StringBuffer();
    LinkedList<ContextRequest_Status> requests = new
        LinkedList<ContextRequest_Status>();
    requests.add(ContextRequest_ReadRequestAlice.getContextReq());
    requests.add(ContextRequest_WriteRequestBob.getContextReq());
    requests.add(ContextRequest_ReadRequestBob.getContextReq());
    requests.add(ContextRequest_StopReadRequestAlice.getContextReq());
    requests.add(ContextRequest_StopReadRequestBob.getContextReq());
    requests.add(ContextRequest_WriteRequestAlice.getContextReq());
    for (ContextRequest rcxt : requests) {
      AuthorisationPDP resPDP = system.pdp.doAuthorisation(rcxt);
      AuthorisationPEP resPEP = system.pep.doEnforcement(resPDP);
    }
  }
}
```

Per esigienze di comodità il contesto è stato integrato in un'unica classe insieme alle richieste.

Codice 5.16: Richiesta e contesto

```
public class ContextRequest_StopReadRequestAlice {
  private static ContextRequest_Status CxtReq;
  public static ContextRequest_Status getContextReq() {
    if (CxtReq != null) {
      return CxtReq;
    }
    HashMap<String, Object> req_category_attribute_name = new
        HashMap<String, Object>();
    HashMap<String, Object> req_category_attribute_action = new
        HashMap<String, Object>();
    HashMap<String, Object> req_category_attribute_file = new
        HashMap<String, Object>();
    req_category_attribute_name.put("id", "Alice");
    req_category_attribute_action.put("id", "stopRead");
    req_category_attribute_file.put("id", "file1");
    Request req = new Request("stop_read_request");
    req.addAttribute("name", req_category_attribute_name);
    req.addAttribute("action", req_category_attribute_action);
    req.addAttribute("file", req_category_attribute_file);
    CxtReq = new ContextRequest_Status(req,
        ContextStub_Status_Default.getInstance());
    ContextStub_Status_Default.getInstance().setStatus(createStatus());
    return CxtReq;
  }
  private static FacplStatus createStatus() {
    ArrayList<StatusAttribute> attributeList = new
        ArrayList<StatusAttribute>();
    attributeList.add(new StatusAttribute("isWriting",
        FacplStatusType.BOOLEAN, "false"));
    attributeList.add(new StatusAttribute("counterReadFile1",
        FacplStatusType.INT, "0"));
    attributeList.add(new StatusAttribute("counterReadFile2",
        FacplStatusType.INT, "0"));
    FacplStatus status = new FacplStatus(attributeList, "stato");
    return status;
  }
}
```

Il metodo che si occupa della creazione dello stato è *createStatus()*, il quale non farà altro che crearsi una serie di attributi ed inseririrli in un nuovo oggetto di tipo *FacplStatus*, per poi restituirlo. La richiesta invece viene restituita sottoforma di un oggetto di tipo *ContextRequest_Status*, il quale

verrà restituto dal primo dei due metodi. La richiesta è formata da diverse *Hashmap* che sono come le categorie degli attributi che verranno inseriti al loro interno. Queste hashmap poi vengono aggiunte ad un oggetto *Request* insieme ad una stringa che racchiude l'informazione riguardante la categoria dell'oggetto appena inserito. Alice, in Codice 4.3, richiede ad un certo punto un'azione identificata come *(action / id , "stopRead")*, ed insieme a questo attributo manderà anche altri due attributi, uno che rappresenta il suo nome, e l'altro che contiene il nome del file su cui deve essere effettuata l'azione. L'attributo riguardante l'azione viene codificato alle righe 11 e 15.

Come ultima cosa vediamo come viene codificato un set di *Policy*. Come detto in precedenza una *Policy* è formata da un *Target*, eventualmente altre *Policy* o *Rule* e delle *Obligation*. Prendiamo ora in esempio una *Policy* che contiene tutti e tre questi elementi.

Codice 5.17: Policy StopRead

```
1  private class PolicySet_StopRead extends PolicySet {
     protected ContextRequest_Status ctxReq;
3    public PolicySet_StopRead(ContextRequest_Status ctxReq) throws
        MissingAttributeException {
       this.ctxReq = ctxReq;
5      addId("StopRead_Policy");
       addCombiningAlg(it.unifi.facpl.lib.algorithm.DenyUnlessPermitGreedy.class);
7      ExpressionFunction e1 = new
           ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
            "file1",
9           new AttributeName("file", "id")
            );
11     ExpressionFunction e2 = new
           ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
            "stopRead",
13          new AttributeName("action", "id")
            );
15     ExpressionBooleanTree ebt = new
           ExpressionBooleanTree(ExprBooleanConnector.AND, e1, e2);
       addTarget(ebt);
17     addPolicyElement(new Rule_stopRead());
       addObligation(new ObligationStatus(new SubStatus(),
           Effect.PERMIT, ObligationType.M,
19          ctxReq.getStatusAttribute(new
               StatusAttribute("counterReadFile1",
```

```
                FacplStatusType.INT)), 1));
    }
    private class Rule_stopRead extends Rule {
      Rule_stopRead() throws MissingAttributeException {
        addId("stopRead");
        addEffect(Effect.PERMIT);
        addTarget(new
            ExpressionFunction(it.unifi.facpl.lib.function.comparison.GreaterThan.class,
             ctxReq.getStatusAttribute(
                  ctxReq.getStatusAttribute(new
                      StatusAttribute("counterReadFile1",
                      FacplStatusType.INT))),
             0));
      }
    }
  }
```

In particolare è stata presa in considerazione la *Policy* che permette di fermare la lettura, ovvero quella che verrà valutata quando verrà effettuata la richiesta in Codice 5.16.

Tutte le *Policy* derivano da una classe astratta che verrà estesa secondo le necessità. Come detto prima questa *Policy* contiene tutti e tre gli elementi che la caratterizzano, in particolare la *Obligation* che in questo caso è una *ObligationStatus*. Questa *Obligation* viene aggiunta come tutte le altre, ovvero chiamando l'apposito metodo della superclasse, ed è costruita passandogli come parametro un'azione da effettuare e gli argomenti, in questo caso l'azione è la sottrazione e gli argomenti è l'attributo che rappresenta il numero di lettori e l'intero 1.

Codice 5.18: Expression Function che valuta un attributo di stato

```
ExpressionFunction e1=new
    ExpressionFunction(it.unifi.facpl.lib.function.comparison.Equal.class,
     ctxReq.getStatusAttribute(
          ctxReq.getStatusAttribute(new
              StatusAttribute("isWriting",
              FacplStatusType.BOOLEAN))),
     false);
```

Nel Codice 5.18 si può vedere come una *ExpressionFunction* riesca a valutare un attributo di stato.

6

CONCLUSIONI

prova 123

BIBLIOGRAFIA

[1] NIST - *A survey of access Control Models* - http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf (Cited on page 11.)

[2] Aliaksandr Lazouski, Fabio Martinelli, Paolo Mori - *Usage control in computer security: A Survey* (Cited on page 20.)

[3] Aliaksandr Lazouski, Gaetano Mancini, Fabio Martinelli, Paolo Mori - *Usage Control in Cloud Systems* - Istituto di informatica e Telematica, Consiglio Nazionale delle Ricerche. (Cited on page 21.)

[4] Jaehong Park, Ravi Sandhu - *The UCON Usage Control Model* - http://drjae.com/Publications_files/ucon-abc.pdf (Cited on pages 3 and 21.)

[5] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, Francesco Tiezzi - *A Formal Framework for Specification, Analysis and Enforcement of Access Control Policies* (Cited on page 29.)

[6] Eric Chabrow- *NIST Guide Aims to Ease Access Control* - http://www.bankinfosecurity.com/nist-publication-aims-to-ease-access-control-a-6612/op-1 (Cited on pages 3 and 16.)