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

Tesi di Laurea

ESTENSIONE DEL LINGUAGGIO FACPL PER ESPRIMERE POLITICHE DI ACCESSO ALLE RISORSE DI UN SISTEMA DI CALCOLO BASATE SUL COMPORTAMENTO PASSATO

EXTENSION OF FACPL LANGUAGE FOR EXPRESSION OF ACCESS POLICY TO RESOURCE OF A SYSTEM BASED ON THE PAST BEHAVIOUR

FEDERICO SCHIPANI

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

Anno Accademico 2014-2015

Federico Schipani: *Estensione del linguaggio FACPL per esprimere politiche di accesso alle risorse di un sistema di calcolo basate sul comportamento passato*,
Corso di Laurea in Informatica, © Anno Accademico 2014-2015

INDICE

ELENCO DELLE FIGURE

"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 agli anni 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 nasce *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.

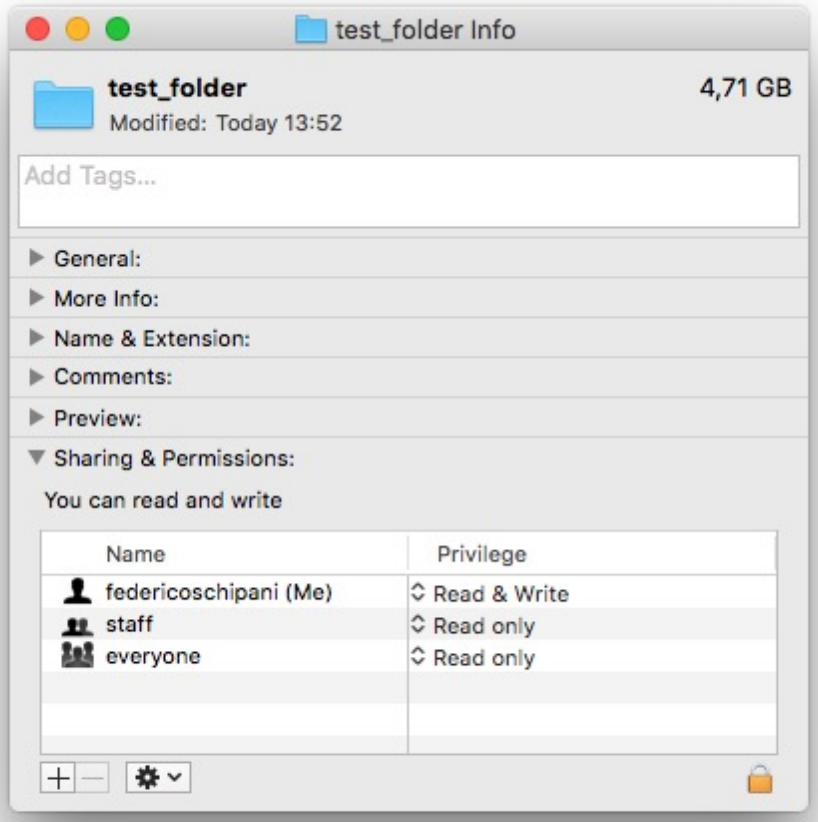

Figura 1: ACL in OS X

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, bisogna 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 bisogna 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.

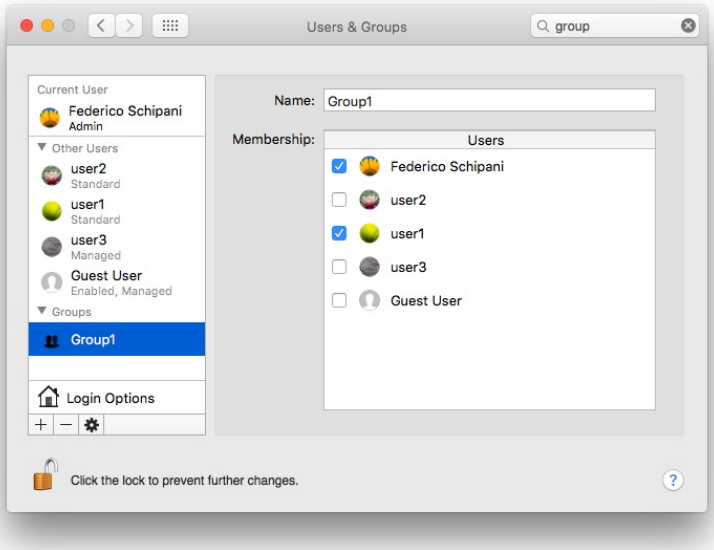

Figura 2: Gruppo in OS X

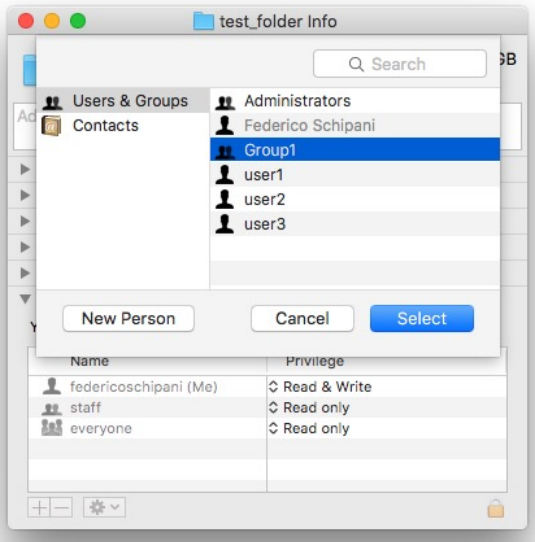

Figura 3: Gruppo in OS X

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 al 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 4). Un vantaggio di ABAC è che non c'è la necessità che il

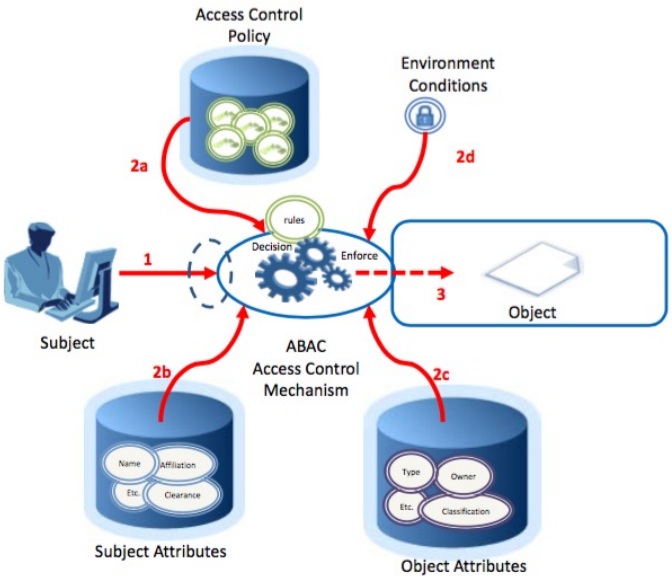

Figura 4: Scenario ABAC base

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 database.
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 controglo 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 fare. 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*.

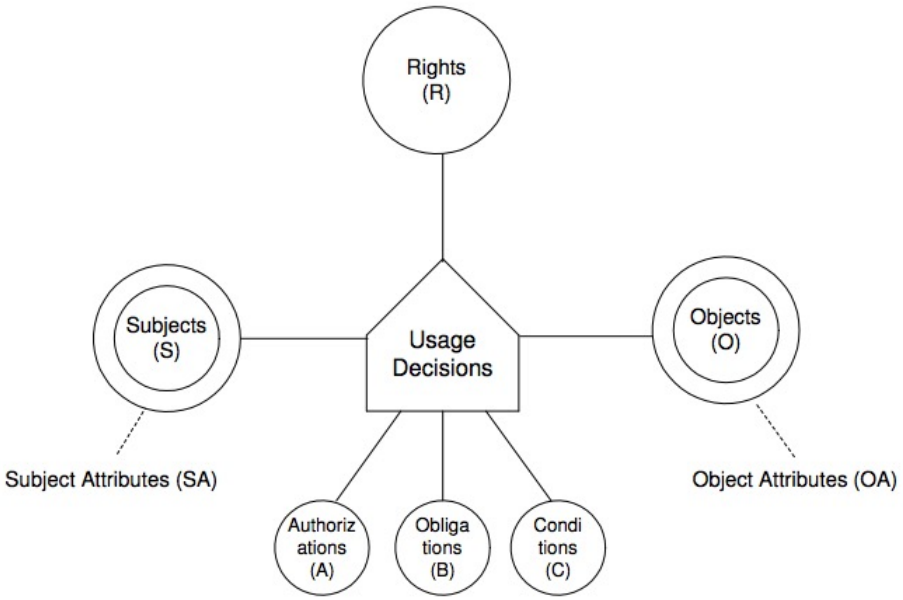

Figura 5: Insieme dei componenti di UCON [4]

Verranno ora proposti alcuni esempi di utilizzo di Usage Control.

Primo Esempio di Usage Control

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.

Secondo Esempio di Usage Control

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.

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

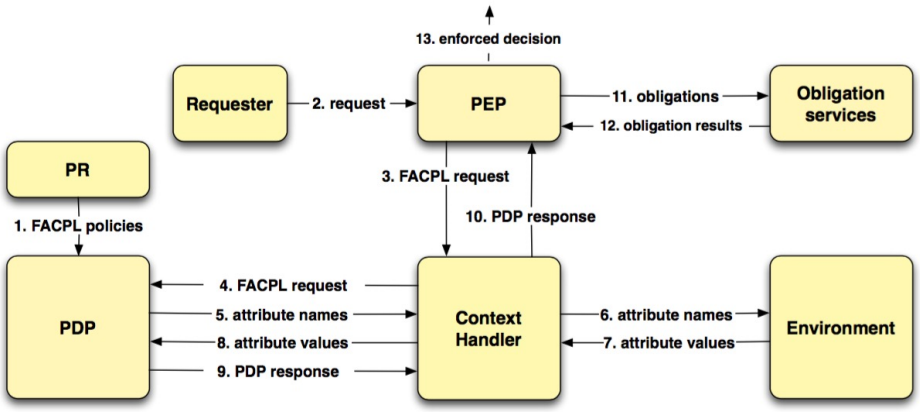

Figura 6: Il processo di valutazione di FACPL

In figura 6 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

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

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

Tabella 2: Sintassi ausiliaria per le risposte

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

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).
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 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 ESEMPI CON FACPL

bla bla bla

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.

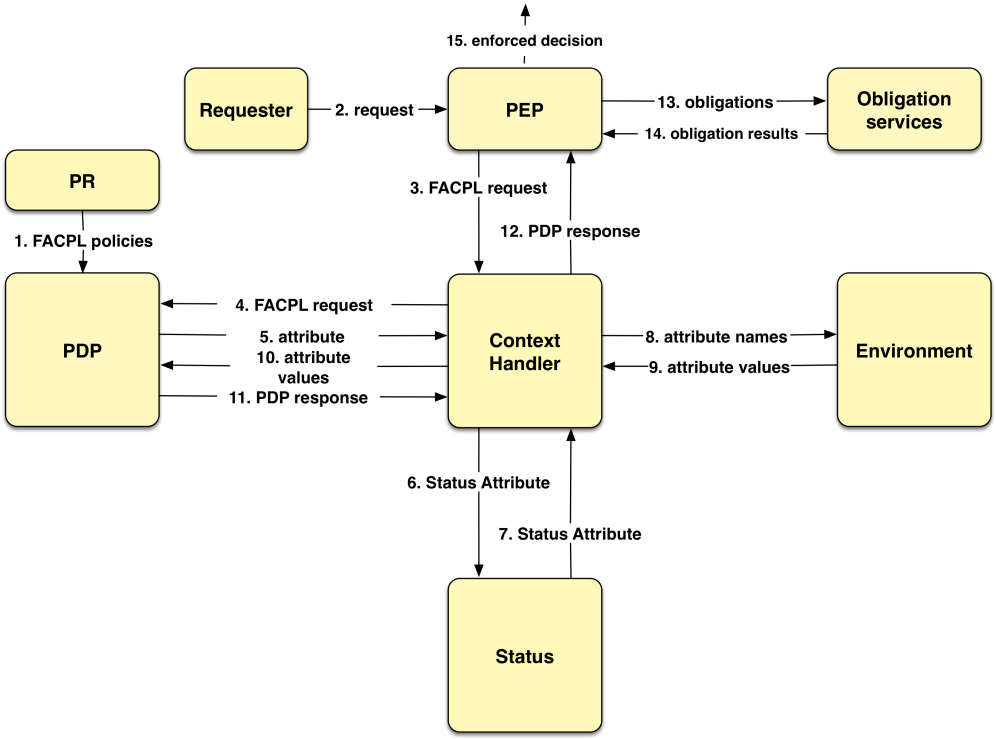

Figura 7: Nuovo processo di valutazione in FACPL

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. 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

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

questo significa che se lo *Status* è presente sarà formato da uno o più *Attribute*.
Passiamo ora a descrivere *Attribute* che è della forma

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

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`)*

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*.

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)
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

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 9.)

[2] Aliaksandr Lazouski, Fabio Martinelli, Paolo Mori - *Usage control in computer security: A Survey* (Cited on page 18.)

[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 18.)

[4] Jaehong Park, Ravi Sandhu - *The UCON Usage Control Model* - http://drjae.com/Publications_files/ucon-abc.pdf (Cited on pages 3 and 19.)

[5] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, Francesco Tiezzi - *A Formal Framework for Specification, Analysis and Enforcement of Access Control Policies* (Cited on page 25.)