UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI

Corso di Laurea in Sicurezza dei Sistemi
e delle Reti Informatiche

Rilascio selettivo delle Politiche per il Controllo dell'Accesso

RELATORE
Prof.ssa Pierangela Samarati

CORRELATORI
Dott. Claudio A. Ardagna
Dott.ssa Sara Foresti

TESI DI LAUREA DI
Mattia Dossena
Matricola 710509

Anno Accademico 2009/2010

Prefazione

Lo sviluppo di Internet quale mezzo di condivisione di servizi ed informazioni ha richiesto lo sviluppo di soluzioni che regolamentassero l'accesso a tali risorse: questa necessità si concretizza nell'identificazione degli utenti che desiderano accedere ad un servizio e nella possibilità di consentire o precludere l'accesso in base alla loro identità. Esistono, però, alcune problematiche intrinseche che derivano principalmente dallo schema di interazione alla base di Internet, ovvero il modello client-server, e sono legate alle diverse e in alcuni casi opposte esigenze delle due parti. Se da un lato il server preferisce un'identificazione quanto più possibile completa e precisa, dall'altro il client non è sempre disponibile al rilascio di tutti suoi dati e necessita di soluzioni che tutelino la sua privacy. La soluzione più efficiente a questi problemi consiste nella creazione di *politiche per il controllo dell'accesso* ai servizi: il server richiede al client alcune credenziali e, dopo averle valutate, decide se consentirgli o meno l'accesso. Tuttavia, siccome il client non può conoscere a priori tutte le credenziali da fornire, è necessario che tali informazioni siano indicate dal server. A sua volta il server deve poter definire alcune preferenze legate alle scelte e alle modalità adottate nella comunicazione della politica. Da un lato, è del tutto lecito ipotizzare che il server richieda al client unicamente le informazioni necessarie alla valutazione della politica senza dover sentire la necessità di rilasciarla nella sua interezza. Dall'altro lato, la comunicazione integrale della politica per il controllo dell'accesso favorisce la privacy del client: questi, valutata autonomamente la politica, potrebbe decidere di non rilasciare determinate informazioni visto che comunque non soddisferebbero la politica. Data una particolare politica, le scelte alla base del suo rilascio possono, quindi, essere molteplici, ma necessitano comunque di una regolamentazione al fine di garantire sempre la possibilità di valutare la politica stessa.

Il presente progetto di tesi ha, dunque, l'obiettivo di creare un'implementazione che, data una politica per il controllo dell'accesso, consenta di specificare determinati parametri e opzioni lato server in modo da gestire efficacemente la selezione ed il rilascio delle credenziali da richiedere al client. L'applicazione creata consente inoltre la valutazione della coerenza tra l'insieme di informazioni richieste al client e la politica per il controllo dell'accesso originale. Il lavoro è organizzato come segue: vengono presentate (Capitolo 2) le attuali soluzioni adottate nel campo delle politiche per il controllo dell'accesso, evidenziando analogie e differenze con questo lavoro; successivamente (Capitolo 3) vengono descritti gli aspetti teorici alla base della gestione del rilascio selettivo delle politiche stesse; infine (Capitolo 4) viene presentata l'applicazione creata con particolare riguardo alle problematiche affrontate, evidenziando inoltre l'adesione dei metodi implementati agli aspetti teorici precedentemente analizzati.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

In questo capitolo viene descritto lo scenario di partenza su cui si basa il presente lavoro di tesi: si cerca dunque di capire le motivazioni alla base della creazione di politiche di controllo dell'accesso e i diversi aspetti per cui è stato necessario lo sviluppo di proposte dedicate. Successivamente viene analizzato l'aspetto specifico al centro di questa trattazione, ovvero la gestione del rilascio selettivo delle politiche, descrivendo la soluzione realizzata in termini generali. Viene descritta infine l'organizzazione del lavoro con una specifica relazione degli argomenti trattati nei diversi capitoli.

1.1 Scenario di partenza

Lo sviluppo di Internet quale mezzo di condivisione di servizi ed informazioni si è ben presto dovuto confrontare con l'esigenza, derivata dalle più svariate motivazioni, di separare le risorse destinate al pubblico dominio da quelle utilizzabili unicamente da una ristretta e definita cerchia di utenti. A questo scopo si rese utile lo sviluppo di soluzioni che regolamentassero l'accesso a tali risorse: questa regolamentazione si traduce nell'identificazione degli utenti che desiderano accedere ad un servizio e nella possibilità di consentire o precludere l'accesso in base alla loro identità. Lo schema di interazione principale su cui si basa Internet prevede una comunicazione bilaterale tra una parte che richiede un determinato servizio, il client, ed un'altra che lo mette a disposizione, il server. Imporre una regolamentazione a questo tipo di interazione comporta un inevitabile confronto con una serie di problemi, legati alle diverse e in alcuni casi opposte esigenze delle due parti. Da un lato il server preferisce un'identificazione quanto più possibile completa e precisa sul modello dell'identificazione adottata nei sistemi tradizionali che non interagiscono con il Web; dall'altro il client non è sempre disponibile al rilascio di tutti suoi dati e necessita di soluzioni che tutelino la sua privacy e la sua sicurezza non solo dal server, ma anche da eventuali attaccanti che possono cercare di appropriarsi indebitamente di informazioni strettamente personali; questo senza, però, dover continuamente rapportarsi alla creazione di account e alla memorizzazione di password. La soluzione più efficiente a questi problemi consiste nella creazione di *politiche per il controllo dell'accesso* ai servizi: il server richiede al client alcune credenziali e, dopo averle valutate, decide se consentirgli o meno l'accesso; tali credenziali devono ovviamente

essere specificate dal server, in quanto non è plausibile che il client le conosca a priori. Anche in questo contesto, si riscontrano, però, esigenze opposte: poichè il server ha necessità di definire modalità e preferenze legate alla comunicazione della politica al client, sarebbe assolutamente lecito ipotizzare, a questo punto, che il server richieda unicamente le informazioni necessarie alla valutazione della politica stessa senza dover sentire la necessità di rilasciare quest'ultima nella sua interezza. Ciò comporta un ovvio vantaggio in termini di salvaguardia della privacy del server: la politica si costituisce anche da parti che il client non potrà mai conoscere — con tutti i vantaggi che ciò comporta — senza che ci siano ripercussioni sulla valutabilità della politica stessa. Dall'altro lato la comunicazione integrale della politica per il controllo dell'accesso favorisce la privacy del client: questi, valutata autonomamente la politica, potrebbe decidere di non rilasciare determinate informazioni visto che comunque non soddisferebbero la politica. Attualmente esistono diverse soluzioni che consentono ad un server di rilasciare determinate politiche in base alle richieste ricevute; esistono anche soluzioni che si focalizzano sulla gestione della privacy e della sicurezza sia nei confronti delle politiche rilasciate, sia nei confronti degli attributi presentati dal client; altri aspetti presi in considerazione riguardano, poi, le modalità e la gestione dell'interazione tra le due parti del canale di comunicazione. Non sono, invece, state ancora proposte delle soluzioni mirate alla selezione delle condizioni e, in particolare, delle singole informazioni contenute in esse che si vuole rilasciare al client a partire da una particolare politica.

1.2 Soluzione proposta

In questo lavoro sono, quindi, presentate delle proposte teoriche finalizzate al rilascio selettivo di una o più parti di una politica d'accesso, secondo delle preferenze decise in modo indipendente lato server. Data una particolare politica, le scelte alla base del suo rilascio possono essere molteplici: queste influiscono sulla visibilità della politica stessa e consentono la selezione delle informazioni da rilasciare. Non è, però, attuabile la possibilità di esprimere preferenze in modo del tutto arbitrario sulla visibilità di ciascuna condizione inclusa nella politica adottata: ciò potrebbe infatti portare all'impossibilità di valutare la politica a causa della scarsità delle informazioni richieste al client. Si rendono dunque necessarie delle regolamentazioni che riguardano, in particolare, i diversi tipi di informazioni che è possibile rilasciare per la valutazione della politica e, in generale, la coerenza tra le scelte effettuate e la possibilità di eseguire tale valutazione. Il presente progetto di tesi si basa in particolare su alcuni aspetti teorici [1] che consentono lo sviluppo delle regolamentazioni per gestire il rilascio selettivo delle politiche per il controllo dell'accesso, al fine di creare un'implementazione in grado di gestire efficacemente la selezione ed il rilascio degli attributi da richiedere al client.

1.3 Strutturazione del lavoro

Il lavoro parte da una panoramica sulle caratteristiche degli attuali sistemi utilizzati nel campo delle politiche per il controllo dell'accesso (Capitolo 2), analizzando i diversi aspetti su cui si concentrano e identificando eventuali relazioni con questo

progetto. In particolare, sono presi in considerazione lo standard XACML, che consente la specifica di due tipi di linguaggi per la gestione delle politiche d'accesso, e le proposte che consentono di includere gli attributi certificati all'interno delle politiche. Segue la trattazione degli aspetti teorici (Capitolo 3) che è necessario analizzare nelle diverse fasi del processo di rilascio: questi riguardano innanzitutto la gestione della visibilità delle singole condizione in relazione ai tipi di attributi contenuti in esse e specificano quali precauzioni è necessario prendere affinchè la politica che sarà poi rilasciata al client possa essere valutata. Successivamente l'attenzione viene spostata sulle operazioni che è necessario eseguire per offuscare le parti di politica che si desidera non siano rilasciate; l'applicazione e il modo stesso con cui queste operazioni agiscono dipenderanno sia dal tipo di attributi sia dal grado di visibilità precedentemente adottato per essi. La politica risultante dall'applicazione di queste operazioni sarà chiamata *Vista* e sarà quella comunicata al client: si rende, dunque, necessario un ulteriore controllo di coerenza tra questa e la politica per il controllo dell'accesso originale; questo controllo viene eseguito verificando se la Vista gode o meno di determinate proprietà. In base al risultato è possibile controllare se le preferenze adottate riguardanti i gradi di visibilità delle singole condizioni producono una Vista che ha significato per il client e può consentirgli di ottenere l'accesso. Successivamente vengono descritte le caratteristiche dell'implementazione proposta (Capitolo 4): vengono analizzati i metodi usati nella definizione dei gradi di visibilità, nel processo di trasformazione della politica e nell'analisi della coerenza tra la Vista e la politica originale. Infine viene proposto un esempio di esecuzione dell'applicazione al fine di verificare passo per passo l'evolversi del processo computazionale.

Capitolo 2

Le politiche per il controllo dell'accesso

I primi tentativi di applicare il concetto di controllo dell'accesso a sistemi raggiungibili via Web richiesero la totale revisione delle soluzioni adottate fino ad allora per i sistemi tradizionali, in cui gli utenti venivano classificati in base alla loro identità ed il controllo doveva essere effettuato per ogni risorsa a cui si volesse accedere; l'identificazione preliminare dell'utente che effettua la richiesta e la successiva verifica delle autorizzazioni in suo possesso, non sono infatti applicabili nei sistemi che necessitano dell'interazione con il Web, in quanto il server non è in possesso di alcuna informazione preliminare del client. Questo, a sua volta, non ha conoscenza delle condizioni in base alle quali l'accesso è permesso o precluso ed è quindi necessario che tali condizioni siano comunicate dal server stesso.

2.1 XACML

Le giuste intuizioni da cui partivano molte delle soluzioni proposte [7], come la differenziazione tra gli attributi dichiarati dal client e quelli certificati da una terza parte o la provata necessità di adottare un linguaggio logico con cui fosse possibile specificare una politica di controllo dell'accesso e comunicarla al client, si scontravano però con i problemi legati alla loro messa in pratica per ragioni fondamentalmente dovute all'efficienza e alla semplicità d'utilizzo. Lo sviluppo di eXtensible Access Control Markup Language (XACML) [2] risolse, in parte, alcuni di questi problemi. La definizione di XACML si basò su alcune provate esigenze che un linguaggio per definire politiche di sicurezza avrebbe dovuto soddisfare, come la necessità di un metodo per combinare singole regole in una politica da applicare a una particolare richiesta, la rapida identificazione della politica da applicare su una data azione in base al valore degli attributi presentati dall'utente o la specifica di insiemi d'azioni da eseguire in relazione alla valutazione di una politica. XACML v.2.0 (2005) è uno standard OASIS che basa il suo funzionamento sull'interazione tra quattro componenti logici:

- il *Policy Enforcement Point*, che effettua il controllo degli accessi inviando richieste di decisione e facendo rispettare le risposte a tali richieste;

- il *Policy Information Point*, che svolge la funzione di archiviazione dei valori dei diversi attributi riguardanti risorse, azioni e ambienti;
- il *Policy Decision Point*, che valuta le politiche applicabili e produce la risposta alla richiesta di accesso;
- il *Policy Administration Point*, che produce le singole regole, le raggruppa in politiche e le memorizza in un apposito archivio;

La richiesta di accesso ad una risorsa si traduce, quindi, in una richiesta al Policy Enforcement Point; dopo che il Policy Information Point è stato interrogato sugli attributi coinvolti, viene inviata una richiesta XACML al Policy Decision Point. Questo, in base alle politiche messe a disposizione dal Policy Administration Point, ritorna una risposta XACML che viene tradotta ed inviata al Policy Enforcement Point, il quale fa rispettare gli obblighi derivati dalla valutazione della richiesta. L'entità di sistema che converte la richiesta dal suo formato nativo al formato XACML e viceversa e che permette la comunicazione tra tutte le altre componenti del sistema viene chiamato *context handler*.

XACML propone due tipi di linguaggi, entrambi basati su XML: uno viene utilizzato per descrivere i requisiti generali del controllo degli accessi a risorse distribuite, consentendo, quindi, la definizione delle politiche di sicurezza; l'altro viene impiegato nel contesto di richiesta e di risposta: permette quindi di gestire gli accessi alle risorse stesse, consentendo di sapere quando una data azione su di una risorsa può essere compiuta o meno ed interpretando un eventuale risultato. Questi due tipi di linguaggi sono simili, ma, essendo sviluppati partendo da esigenze differenti, presentano alcune sostanziali differenze che si ripercuotono soprattutto sugli elementi XML usati: il linguaggio di Policy utilizza elementi adatti a definire degli insiemi di regole, degli effetti derivati dallo loro applicazione, degli insiemi di risorse su cui si applicano gli effetti e degli algoritmi di combinazione delle diverse regole; il linguaggio per la definizione di richieste e risposte contiene elementi per la specifica di soggetti, risorse, azioni e ambienti contenuti nel contesto di richiesta ed elementi per incapsulare la decisione di autorizzazione prodotta dal Policy Decision Point, insieme ad un elenco di uno o più risultati in base a quante sono state le richieste pervenute. L'Esempio 1 mostra una politica che vieta l'accesso al dominio DTI a tutti gli utenti con dominio `blacklist.org`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:
         schema:os:http://docs.oasis-open.org/xacml/access_control
         -xacml-2.0-policy-schema-os.xsd"
      PolicyId="urn:oasis:names:tc:example:Politica1"
      RuleCombiningAlgId="identifier:rule-combining-algorithm:
         deny-overrides">
   <Description>
      Politica per il controllo dell'accesso al dominio dti.it
   </Description>
```

```
<Target/>
<Rule RuleId= "urn:oasis:names:tc:xacml:2.0:example:Regola1"
      Effect="Deny">
   <Description>
      Ogni soggetto con un'e-mail del dominio
      blacklist.org non puo' effettuare alcun
      tipo di azione.
   </Description>
   <Target>
      <Subjects>
         <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
                  function:rfc822Name-match">
               <AttributeValue DataType=
                     "http://www.w3.org/2001/XMLSchema#string">
                        blacklist.org
               </AttributeValue>
               <SubjectAttributeDesignator
                     AttributeId="urn:oasis:names:tc:xacml:1.0:
                        subject:subject-id"
                     DataType="urn:oasis:names:tc:xacml:1.0:
                        data-type:rfc822Name"/>
            </SubjectMatch>
         </Subject>
      </Subjects>
   </Target>
</Rule>
</Policy>
```

(1)

Il fatto di basarsi su XML rappresenta uno dei punti di forza di XACML, in quanto consente la completa indipendenza dalla piattaforma e l'adesione ad uno standard sempre più utilizzato per lo sviluppo di linguaggi per applicazioni Web. XACML può dunque essere definito un modello nel campo del controllo degli accessi alle risorse: una volta costruito il sistema con tutte le sue parti, risulta facile ed intuitivo creare delle politiche di sicurezza per la protezione delle risorse; ciò si sposa perfettamente con il lavoro di tesi presentato, in quanto la definizione di un ambiente standard finalizzato alla creazione delle politiche di controllo è una solida base da cui sviluppare gli aspetti teorici analizzati e la loro successiva implementazione.

2.1.1 Estensioni di XACML

XACML include delle estensioni che consentono la definizione di nuove funzioni, nuovi tipi di dato e metodi per la combinazione delle politiche, cosa particolarmente utile anche al presente progetto in relazione ad alcuni limiti propri di XACML. Tali estensioni [4] consentono, ad esempio, l'applicazione del controllo degli accessi anche ad attributi certificati, l'utilizzo di astrazioni per categorie di attributi e la gestione

della comunicazione tra client e server al fine di amministrare in modo efficiente per entrambe le parti la richiesta ed il rilascio di singole informazioni. In realtà XACML consente di richiedere che alcuni attributi siano certificati; tale possibilità è limitata, però, ad un piccolo insieme di attributi, come la data o l'ora, e non è sicuramente sufficiente al supporto di controlli su attributi certificati più complessi. Grazie alla specifica di una sintassi formalmente definita è, invece, possibile utilizzare XACML sia per richiedere al client di presentare un certificato indipendentemente dal valore che questo assume sia per confrontare il certificato presentato con un certo valore stabilito: ciò consente di includere condizioni su attributi certificati all'interno delle politiche specificate in sintassi XACML. Il concetto di astrazione trova poi utile applicazione nella gestione delle politiche di controllo dell'accesso, in quanto semplifica la specifica delle condizioni all'interno delle politiche stesse: ad esempio, una condizione su un certificato astratto chiamato "documento" con annessa l'indicazione che tale documento può essere una carta d'identità, un passaporto o una patente, consente una gestione più semplice ed efficiente della politica relativa.

Si noti che, come specificato in [4], le proposte applicate ad XACML sono di carattere astratto e possono essere utilizzate efficacemente con qualsiasi linguaggio che consenta di definire politiche in grado di supportare i requisiti propri delle applicazioni Web più recenti. Tali proposte consentono, inoltre, di adottare il grado di specificità che si assume per buono in questa trattazione: esse consentono di definire in modo particolarmente preciso una qualsiasi politica per il controllo dell'accesso; da questo punto di partenza si sviluppa il discorso riguardante il rilascio selettivo delle politiche, che è appunto al centro di questo progetto.

2.2 Sicurezza nella gestione degli attributi

Alcune proposte ([5], [6]) focalizzano l'attenzione sulla protezione degli attributi certificati e non che il client deve rilasciare nell'interazione con il server, puntando a salvaguardare la sicurezza e la privacy del client stesso nei confronti di potenziali avversari interessati all'acquisizione impropria dei suoi attributi. Nonostante il carattere generale di tali proposte, è possibile riscontrare una motivazione della loro formulazione nella carenza di sicurezza e privacy propria di XACML, perlomeno nella v.2.0. Questa carenza si concretizza nella vulnerabilità del canale di comunicazione utilizzato dai vari attori XACML: le informazioni contenute nei messaggi scambiati possono essere osservate e, in alcuni casi, modificate, compromettendo quindi seriamente sia la sicurezza del sistema sia l'effettivo controllo sugli accessi. Le soluzioni proposte hanno dunque l'obbiettivo di fornire al client diverse modalità per specificare delle preferenze sulle informazioni da rilasciare in base al grado di protezione impostato sulle informazioni stesse; queste proposte partono dal valido presupposto che applicare lo stesso approccio di specifica e regolamentazione del controllo dell'accesso utilizzato lato server all'insieme di informazioni di cui è in possesso il client non sia soddisfacente, in quanto non copre tutte le possibili esigenze che potrebbero essere richieste dal client. In particolare, una di queste proposte [5] utilizza un modello grafico per illustrare l'insieme di informazioni possedute dal client e quello delle richieste effettuate dal server, fornendo diverse modalità con cui il client può specificare il grado di privatezza di ciascun attributo; tale specifica è poi applicata ai grafi inizialmente definiti al fine di verificare le corrispondenze e

minimizzare il numero di informazioni che devono essere rilasciate per la valutazione ed il soddisfacimento della politica relativa.

2.2.1 Utilizzo dei certificati

La necessità e l'applicazione di alcune delle proposte esaminate trovano un'altra giustificazione nel grande sviluppo e diffusione dell'uso di attributi certificati, che consentono di verificare facilmente che la parte dall'altro capo del canale di comunicazione sia in possesso di alcune proprietà validate. Al giorno d'oggi l'uso più significativo di questo tipo di attributi è rappresentato dai certificati definiti dallo standard X.509 v.3.0, che vengono esibiti dai server per confermare la loro legittimità. Tale standard si basa su una terza parte fidata, detta anche *certificatore*, responsabile del rilascio dei certificati(una più completa definizione di certificatore è contenuta nel Paragrafo 3.1): questi associano una chiave pubblica ad un *nome*, che potrebbe anche essere un indirizzo e-mail o un record DNS, in modo da attestarne la validità. X.509 permette, inoltre, alle aziende di utilizzare certificati *root*, che consentono a tutti i dipendenti l'utilizzo di un'unica chiave pubblica, ed include degli standard per definire meccanismi finalizzati alla revoca di certificati precedentemente creati.

Sono inoltre state sviluppate alcune soluzioni [9] finalizzate a garantire e preservare l'anonimato nell'interazione con i sistemi di gestione dei certificati, mediante i quali gli utenti possono ottenere certificati dalle terze parti e dimostrarne il possesso. In questo contesto, assicurare l'anonimato si traduce nell'impossibilità di associare le diverse transizioni portate a termine dallo stesso utente, al fine di garantirne la privacy. In questo modo ciascun utente può dimostrare il possesso di ciascuna credenziale ogni volta che lo desidera senza alcun riferimento alle terze parti coinvolte. Per evitare, inoltre, un abuso dell'anonimato, a discrezione dell'utente è possibile revocarne l'applicazione su determinate transizioni. Viene poi utilizzato uno schema logico, chiamato *all-or-nothing*, per contrastare la condivisione di certificati: se un utente può utilizzare anche un solo certificato di un altro utente, allora può utilizzarli tutti con la possibilità, quindi, di impersonificarlo, cosa ovviamente sconveniente per il possessore originale del certificato condiviso.

2.2.2 La negoziazione fidata

Il rilascio bilaterale di attributi certificati è alla base dell'instaurazione di una negoziazione fidata di informazioni tra il client ed il server. Lo stesso rilascio di certificati può, come si è visto, soffrire di alcuni problemi legati alla sicurezza delle informazioni rilasciate: nonostante l'esistenza di rigide politiche di controllo, alcune informazioni sensibili potrebbero infatti essere dedotte dal comportamento dei partecipanti alla negoziazione; allo scopo di rendere quanto più sicuro possibile questo modello, sono state fatte delle proposte che mirano alla creazione di strutture finalizzate alla protezione delle singole informazioni che di volta in volta ciascuna parte ottiene dall'altra per evitare che queste vengano ricavate senza autorizzazione. Si consideri, come esempio, quello citato in [8]: si supponga che una politica per il controllo dell'accesso ad un risorsa richieda ad un utente di rilasciare un certificato che attesti il suo impiego presso la CIA. Nel caso l'utente possieda questo attributo, allora questo dovrebbe essere protetto, a sua volta, da una politica per il controllo

dell'accesso, che deve essere soddisfatta dal server. Nel caso, invece, in cui l'utente non abbia il certificato, l'utente stesso si troverebbe nella necessità di interrompere la negoziazione, in quanto per lui non ci sarebbe alcuna possibilità di ottenere l'accesso. Il server può quindi dedurre con buona probabilità di certezza se l'utente lavora o meno per la CIA semplicemente considerando la sua risposta, anche nel caso in cui esista una severa politica di controllo dell'accesso per quell'attributo. Gli approcci proposti operano, quindi, su questa correlazione tra informazioni differenti, cercando di romperla e rendendo, quindi, ciascun attributo completamente indipendente oppure cercando di renderla sicura, mediante l'utilizzo di particolari attributi di *acknowledgment*.

Altre soluzioni si concentrano invece sulla protezione generale delle politiche di accesso che possono essere rilasciate durante la negoziazione: le parti in comunicazione sfruttano, infatti, tali politiche per conoscere i requisiti necessari che ciascuna deve presentare all'altra allo scopo di instaurare una negoziazione fidata. Lo schema UniPro [10] (UNIfied scheme for resource PROtection) consente sia la protezione delle diverse risorse sia delle politiche che controllano l'accesso ad esse basandosi su un modello a grafo. Ci possono quindi essere politiche che proteggono politiche che proteggono politiche e così via, ma è necessario che quella all'inizio della catena sia rilasciata nella sua interezza oppure sia totalmente offuscata. Si noti che i concetti di rilascio e di soddisfacimento di una politica sono separati: se una risorsa è protetta da una politica che viene soddisfatta — così come tutte le politiche che la precedono nella catena — allora l'accesso a tale risorsa è consentito, indipendentemente dal fatto che la poltica sia stata rilasciata o meno. Questo tipo di schema consente quindi di poter definire in modo efficiente il grado di portezione di ciascuna risorsa e politica, ma non consente il rilascio selettivo delle politiche stesse. Lo schema PROTUNE (PRovisional TrUst NEgotiation) [11] risolve parzialmente questo problema, consentendo di definire alcune condizioni come "private" in modo da non rilasciarle al client. In questo caso si rende però necessario avvisare il client che l'accesso potrebbe essergli precluso anche se gli attributi da lui forniti soddisfano le condizioni rilasciate su essi; ciò avviene nel caso dell'offuscamento di particolari condizioni necessarie alla valutazione della politica nella sua interezza.

La creazione di soluzioni mirate alla protezione della privacy e della sicurezza degli attributi del client e delle politiche del server prendono in considerazione un aspetto che nel presente lavoro di tesi viene parzialmente tralasciato, ma che ricopre comunque primaria importanza nella realizzazione di proposte pratiche ed efficienti nel campo delle politiche per il controllo dell'accesso. Queste, insieme alle analisi sempre più specifiche riguardanti la creazione e l'utilizzo dei certificati, sono, quindi, da considerarsi complementari agli aspetti teorici qui trattati e la loro trattazione consente un approccio ancora più completo al problema generale.

Esistono, poi, alcune proposte che partono da un approccio differente: la soluzione in [12] consente l'interazione tra client e server minimizzando sia il rilascio dei certificati sia quello delle politiche d'accesso. I protocolli proposti fanno in modo che il server non abbia alcuna conoscenza riguardo i certificati del client, neanche nel caso in cui a questo sia consentito l'accesso. Il client, a sua volta, non ha alcuna informazione riguardo la struttura della politica e la gamma di certificati che gli hanno permesso di ottenere l'accesso. Si può chiaramente vedere come tale approccio non sia relazionabile con quello qui presentato: il presente lavoro ha infatti

l'obbiettivo di permettere e gestire il rilascio selettivo delle politiche, mentre quello appena citato punta a offuscare il più possibile il rilascio stesso al client.

Capitolo 3

La gestione del rilascio selettivo

Prima di procedere allo sviluppo e all'implementazione dell'applicazione si è reso necessario uno studio preliminare riguardante i diversi aspetti legati alle politiche per il controllo dell'accesso ed al loro rilascio al client. Tale studio [1] analizza il processo di trasformazione della politica in base a determinate preferenze, fino al rilascio di una o più parti della politica stessa. In particolare, vengono inizialmente esaminate le motivazione alla base della conversione della politica in una formula booleana e, successivamente, in una struttura ad albero; in secondo luogo sono introdotti dei concetti e dei metodi applicativi per la gestione della visibilità delle diverse condizioni che costituiscono la politica; segue, quindi, la creazione dell'insieme di condizioni da rilasciare al client sulla base della visibilità assegnata alle diverse condizioni. Infine viene proposta un'analisi di tale insieme rispetto alla politica per il controllo dell'accesso originale allo scopo di verificarne la coerenza.

3.1 Concetti preliminari

Una politica per il controllo dell'accesso specifica quali attributi un client deve fornire per ottenere l'accesso ad un dato servizio. Tali attributi possono essere classificati in due diverse tipologie, le *dichiarazioni* ed i *certificati*. Una dichiarazione è un attributo dichiarato dal client stesso, su cui non è stato applicato nessun tipo di controllo o certificazione; teoricamente, quindi, un client potrebbe rilasciare un informazione fasulla riguardo un certo tipo di attributo, senza che il server abbia alcuna possibilità di verificarne la veridicità. Si consideri, come esempio, una semplice richiesta dell'età del client: indipendentemente dalla propria età reale, il client potrebbe dichiararne una qualsiasi; un'eventuale condizione su questo attributo sarà valutata, ovviamente, sull'età dichiarata fornendo poi un risultato che potrebbe essere diverso se la stessa condizione fosse valutata sull'età reale. Un certificato è invece un attributo che è stato garantito da una terza parte. Questa garanzia si traduce, tipicamente, nella firma digitale dell'attributo, che rappresenta un sistema di autenticazione per documenti digitali avente accezione giuridica e basato su algoritmi di cifratura asimmetrica. I responsabili dell'applicazione della firma digitale sono particolari soggetti con determinati requisiti di onorabilità: fra le caratteristiche per svolgere l'attività di certificatore vi è quella, ad esempio, per cui occorre essere una società con capitale sociale non inferiore a quello richiesto per svolgere

l'attività bancaria. Normalmente, quindi, i certificatori non sono singoli soggetti, ma piuttosto grosse società. La certificazione garantisce al server che un particolare attributo presentato dal client sia valido e riconducibile al client stesso, senza alcuna possibilità di fraintendimenti. Si consideri, come esempio, un server che richieda la carta d'identità del client: si rende necessaria, in questo caso, la garanzia che la carta d'identità fornita sia innanzitutto valida ed in secondo luogo che sia effettivamente la carta d'identità di quel particolare client; queste garanzie sono fornite efficacemente dalla certificazione della carta d'identità stessa da parte di un certificatore, a cui il server può inoltre riferirsi per eventuali ed ulteriori verifiche. Più avanti nel testo sarà possibile riscontrare alcune importanti differenze nel modo in cui dichiarazioni e certificati saranno trattati.

Una politica per il controllo dell'accesso può essere considerata come un insieme di condizioni valutabili in base agli attributi forniti dal client; tra le diverse condizioni intercorrono, inoltre, alcune relazioni che sono definite dalla politica stessa. Ciascuna condizione risulta quindi composta da un operatore, indicato con o, e da uno o più operandi, indicati con t e numerati in modo sequenziale. Gli operatori sono solitamente di tipo matematico (ad esempio $>$, $<$ o $=$) e di tipo logico (ad esempio $\vee$ o $\wedge$). Gli operandi possono essere costituiti da dichiarazioni, certificati o valori costanti con cui i primi due tipi possono essere confrontati. Fatte queste considerazioni, è abbastanza intuitivo associare una politica per il controllo dell'accesso ad una corrispondente formula booleana. Si consideri, come esempio, una politica che consenta l'accesso ad utenti che dichiarino di essere cittadini italiani maggiorenni: la formula booleana associata conterrà le due condizioni *(Country = Italy)* e *(Age > 18)*, poste in relazione dall'attributo logico $\wedge$ in quanto entrambe devono essere soddisfatte perchè l'accesso sia garantito. La formula booleana risultante è mostrata nell'Esempio 2.

$$(Country = Italy) \wedge (Age>18) \quad \textbf{(2)}$$

In questo genere di contesto si crea la necessità di distinguere in modo esplicativo le dichiarazioni dai certificati, in quanto il server deve aver la possibilità di specificare al client se un attributo può essere semplicemente dichiarato oppure è necessario che sia certificato da una terza parte. Per convenzione, i certificati saranno quindi espressi dal nome dell'attributo preceduto dal prefisso c, come ad esempio $c.s$; se nella formula booleana esiste la condizione per cui un dato attributo s necessita di un certificato di tipo t, tale condizione sarà espressa attraverso la notazione *(c.type = t)*. Nel caso si utilizzino più certificati nella stessa politica, sarà ovviamente necessario differenziarli: ciò può essere fatto inserendo una numerazione sequenziale dopo il prefisso c. Si consideri ora una politica che consenta l'accesso ad un client che dichiari di essere maggiorenne o che presenti una carta d'identità certificante che sia nato a Crema. La formula booleana associata conterrà la macro-condizione *((c.type = CartaIdentità) $\wedge$ (c.LuogoDiNascita = Crema))* che opera sul certificato e la condizione *(Age > 18)* che opera sull'età dichiarata, poste in relazione dall'operatore logico $\vee$ in quanto basta che una sola sia soddisfatta perchè l'accesso sia consentito. La formula risultante è mostrata nell'Esempio 3.

$$((c.type = CartaIdentità) \wedge (c.LuogoDiNascita = Crema)) \vee (Age > 18) \quad \textbf{(3)}$$

Sia nell'Esempio 2 che nell'Esempio 3 sono stati utilizzati unicamente operatori di tipo binario, che agiscono cioè su due operandi. Per ragioni di semplicità, chiarez-

za e leggibilità, nel resto della trattazione continueranno ad essere usati solo questo tipo di operatori; ciò non toglie, comunque, che gli aspetti teorici analizzati e l'implementazione creata siano regolarmente applicabili anche ad altri tipi di operatori, come in seguito sarà dimostrato.

3.2 Rappresentazione mediante Policy Tree

La rappresentazione di una politica mediante una formula booleana consente di evidenziare le singole condizioni che saranno valutate per consentire o precludere l'accesso. Per garantire maggiore efficienza, si desidererebbe, inoltre, che tale formula fosse specificata in *DNF* (Disjunctive Normal Form), ovvero come disgiunzione di congiunzioni. Esprimendo la formula in questo modo è immediatamente possibile individuare diversi insieme distinti di condizioni posti in relazione da un operatore disgiuntivo: ciascuno di essi, preso singolarmente, consente la valutazione della politica, indipendentemente dal soddisfacimento degli altri insiemi di condizioni. Si noti che nonostante sia conveniente adottare questa notazione, gli aspetti teorici sviluppati e l'implementazione ricavata da essi non sono comunque subordinati al suo utilizzo.

Nel caso di politiche complesse, in cui compaiono un grande numero di condizioni, o di politiche che consentono l'accesso anche se sono soddisfatte soltanto un sottoinsieme delle condizioni stabilite, l'analisi mediante la sola formula booleana può risultare scomoda ed inefficiente. Si dimostra, invece, molto più pratica ai fini della trattazione e dell'implementazione la rappresentazione della formula e, quindi, della politica in una struttura ad albero, in cui sono messe in risalto non solo le singole condizioni, ma anche le relazioni che intercorrono fra di esse. Questa struttura — descritta in modo formale in [1] — associa un nodo ad ogni operatore o operando, denotandolo con un etichetta che conterrà l'informazione corrispondente. Nel caso l'etichetta di un nodo consista in un operatore, quel nodo avrà una serie di nodi figli a cui è collegato e le cui etichette consistono negli operandi dell'operatore stesso. La struttura creata ha, quindi, un punto di partenza in un nodo radice, la cui etichettà rappresenta un operatore logico (ad eccezione del caso in cui la politica è costituita da una singola condizione: in questo caso il nodo radice avrà un etichetta rappresentante un operatore matematico). Si noti che il nodo radice è l'unico a non avere un nodo genitore. Da questo nodo si svilupperà quindi la ramificazione della struttura mediante il collegamento con i nodi figli, le cui etichette possono contenenere a loro volta operatori logici che mettono in relazione altre condizioni. Ciascun ramo terminerà infine con una condizione riguardante gli attributi che saranno richiesti al client e valutabile mediante un operatore matematico. L'utilizzo di una struttura di questo genere, che sarà chiamata *Policy Tree*, comporta diversi vantaggi: si può notare che attributi e valori costanti con cui questi sono eventualmente confrontati occupano sempre i nodi foglia, mentre gli operandi occupano i nodi interni. Considerando, quindi, le relazioni tra i diversi nodi interni è possibile comprendere facilmente quali condizioni devono per forza essere valutate insieme ad altre e quali possono, invece, permettere o precludere l'accesso in modo autonomo. Questa struttura, inoltre, rispecchia perfettamente la struttura dati che sarà successivamente utilizzata per l'implementazione dell'applicazione, come si vedrà nel Paragrafo 4.2. Si noti che la scelta adottata di utilizzare solo operatori binari ha l'effetto di produrre unicamente

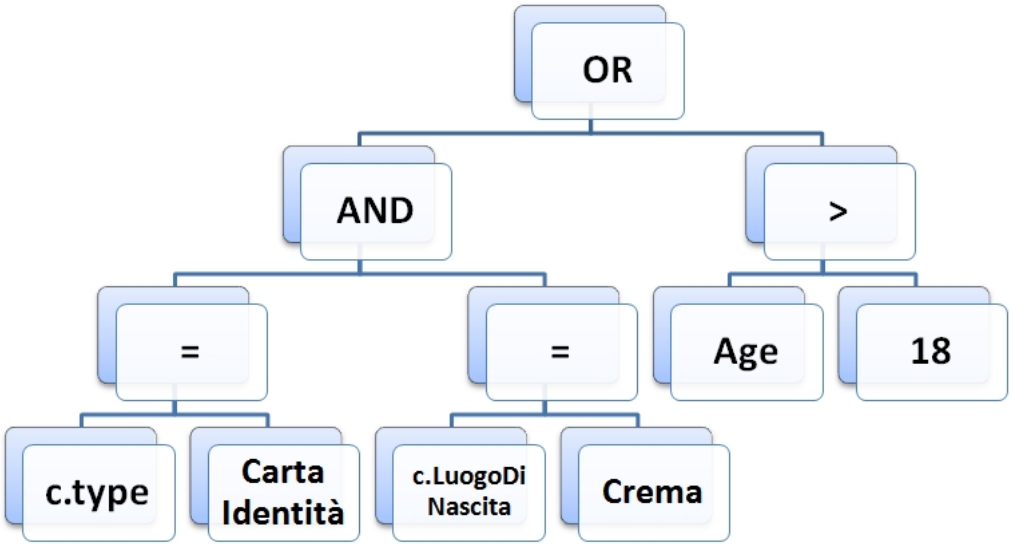


Figura 3.1: Policy Tree

Policy Tree binari, in cui, cioè, ogni nodo ha esattamente 0 o 2 nodi figli e 0 o 4 nodi nipoti. In Figura 3.1 viene mostrato il Policy Tree ottenuto dalla formula utilizzata nell'Esempio 3.

3.3 Gestione della visibilità: il processo di colorazione

Nell'operazione di rilascio della politica per il controllo dell'accesso al client incide la volontà del server di considerare riservata una o più parti di essa. I motivi alla base di questa decisione possono essere molteplici e dipendono da diversi fattori, quali il tipo di servizio offerto o lo specifico tipo di condizione. Si consideri, come esempio, una politica che consente o proibisce l'accesso in base all'età dichiarata; il server potrebbe non avere problemi a rilasciare il valore costante con cui l'attributo *Age* richiesto all'utente è confrontato. Per contro, se l'accesso fosse proibito ad utenti di un certo paese, il server potrebbe invece non voler rilasciare tale informazione, richiedendo semplicemente il paese d'origine e valutando poi l'attributo fornito. Dall'esempio si deduce, inoltre, un importante aspetto: non è sufficiente valutare la visibilità di un'intera condizione, ma è necessario gestire in modo specifico il rilascio delle singole informazioni. In questa prospettiva, la struttura ad albero utilizzata, in cui ogni operatore ed operando sono mantenuti in nodi separati, offre il grande vantaggio di poter effettivamente regolare il rilascio dei singoli nodi: rendere un nodo non visibile si traduce, in pratica, nell'offuscare l'etichetta del nodo stesso.

É importante sottolineare la differenza di gestione della visibilità di un nodo in base alla posizione da questo occupata all'interno del Policy Tree: l'offuscamento di un nodo foglia o del nodo genitore di un nodo foglia ha il solo effetto di modificare il modo in cui una singola condizione viene rilasciata, rendendola visibile per intero, offuscandone una parte o nascondendola completamente. La modifica della visibilità di un nodo interno ha, invece, ripercussioni sulla creazione degli insiemi

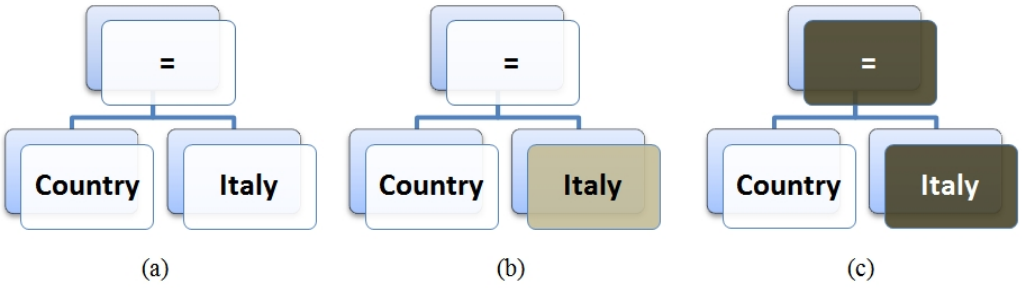


Figura 3.2: Esempi di colorazioni

di condizioni da rilasciare per la valutazione della politica: offuscare un nodo che specifica se due condizioni debbono essere entrambe soddisfatte oppure se è sufficiente il soddisfacimento di una di esse non ha, infatti, effetto sulla visibilità delle condizioni stesse, ma impedisce al client di conoscere la relazione che sussite fra esse.

Offuscare l'etichetta di un nodo si traduce, quando è possibile, nell'eliminazione del nodo dalla struttura dell'albero e, quindi, nella riconfigurazione della struttura stessa. Ci possono essere casi, però, in cui si vorrebbe offuscare un nodo in modo da non rilasciare l'informazione contenuta nella sua etichetta, ma nello stesso tempo mantenere inalterata la struttura dell'albero conservando il nodo. Proprio per questo, non è sufficiente basarsi unicamente su due gradi di visibilità — nodo visibile ed informazione rilasciata / nodo eliminato dall'albero ed informazione non rilasciata — ma è necessario introdurne un terzo, che consenta, appunto, di non rilasciare l'informazione senza eliminare il nodo. Questo problema trova una valida soluzione nella colorazione dei singoli nodi mediante l'uso di tre colori (rosso, giallo, verde), a ciascuno dei quali è associato un diverso grado di visibilità. Nelle immagini di contorno al testo le colorazioni verde, gialla e rossa sono sostituite rispettivamente da quella bianca, grigia chiara e grigia scura.

- **Colore verde**. Un nodo verde è visibile e la sua informazione viene rilasciata. Come esempio si consideri la condizione rappresentata in Figura 3.2(a). Dato che tutti i nodi sono verdi, la condizione è rilasciata nella sua interezza, ovvero *(Country = Italy)*.

- **Colore giallo**. Un nodo giallo è offuscato, la sua informazione non viene rilasciata, ma viene mantenuta la sua presenza all'interno del Policy Tree. Come esempio si consideri la condizione rappresentata in Figura 3.2(b). La condizione viene rilasciata nella forma *(Country = _)* e specifica al client che il paese da lui dichiarato sarà confrontato con un valore, senza specificarlo.

- **Colore rosso**. Un nodo rosso è offuscato, la sua informazione non viene rilasciata e, se è possibile, il nodo viene eliminato dal Policy Tree. Come esempio si consideri la condizione rappresentata in Figura 3.2(c). La condizione viene rilasciata nella forma *(Country _ _)* e specifica semplicemente al client di dichiarare il proprio paese, senza rivelare la condizione su di esso.

In Figura 3.3 viene mostrato una possibile colorazione del Policy Tree visualizzato in Figura 3.1.

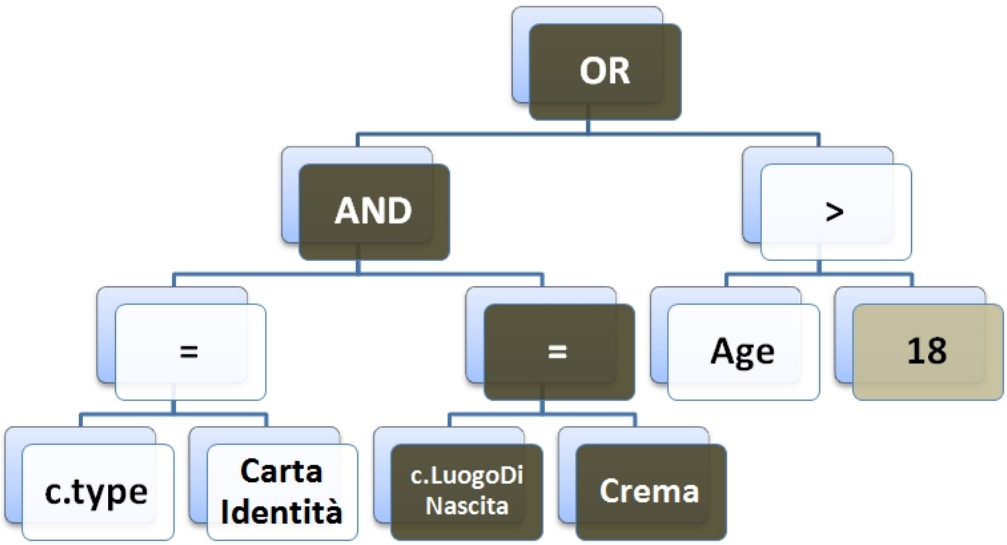


Figura 3.3: Policy Tree colorato

3.3.1 Colorazioni ben definite

La modalità di colorazione adottata permette al server di specificare il colore di ogni singolo nodo e di definire, quindi, un numero arbitrario di funzioni di colorazione, i cui due estremi sono costituiti da un Policy Tree totalmente verde (la politica è rilasciata nella sua interezza) e da uno totalmente rosso (la politica è completamente offuscata). Alcune di queste colorazioni sono, però, prive di significato: di fronte alle corrispondenti informazioni rilasciate il client non sa come comportarsi. Si consideri, come esempio, la condizione *(Age > 18)*, in cui il nodo contenente l'etichetta *Age* e quello contenente l'etichetta $>$ sono entrambi dichiarati rossi. La condizione rilasciata *(_ _ 18)* non ha alcun significato per il client. Si rende, dunque, necessaria una regolamentazione delle funzioni di colorazione, allo scopo di non rilasciare politiche prive di significato. Le regole individuate in [1] sono tre:

- in ogni sottoalbero che include una condizione su un certificato, i nodi devono essere tutti verdi o tutti rossi. La colorazione gialla non è ammessa. Questa regola consente di evitare situazioni in cui il client è informato sul fatto che esiste una condizione su un certificato, senza, però, che sia rilasciato il tipo di certificato su cui insiste la condizione stessa;

- ogni nodo verde la cui etichetta rappresenta un operatore deve avere almeno un nodo figlio non rosso. Questa regola consente di evitare situazioni in cui viene rilasciato unicamente l'operatore senza i suoi operandi;

- ogni nodo foglia verde la cui etichetta rappresenta un valore costante deve avere il nodo fratello anch'esso verde ed il nodo padre non rosso. Questo permette di evitare situazioni simili a quella citata all'inzio del presente paragrafo.

Una colorazione che rispetta queste regole viene detta *colorazione ben definita*.

3.4 Creazione della Vista: il processo di trasformazione

Una volta determinata una funzione di colorazione ben definita, si rende necessario applicare un processo di trasformazione al Policy Tree mirato a semplificarne la struttura, eliminando, ad esempio, i nodi foglia rossi la cui informazione non sarebbe comunque rilasciata. Tale processo è basato sull'applicazione di alcune regole che agiscono sui nodi in base al loro colore e alla loro posizione all'interno dell'albero. L'attuazione di queste regole permette la creazione di una *Vista sul Policy Tree*, una struttura dati ad albero simile al Policy Tree, ma con due importanti differenze. In primo luogo, esiste la possibilità che un nodo abbia un numero di figli superiore a due, cosa che mai si verifica nei Policy Tree binari fino ad ora esaminati; in secondo luogo, i nodi per i quali è stata selezionata la colorazione rossa o gialla saranno effettivamente privi di etichetta. La Vista sul Policy Tree consiste, quindi, nella sezione della politica per il controllo dell'accesso che sarà rilasciata al client.

Data l'importanza centrale del processo di trasformazione, si rende necessaria un'analisi approfondita delle regole applicate. Esse sono distinte in 3 tipologie in base alla funzione svolta:

- la regola di *potatura*, che consente la rimozione dei nodi foglia rossi che non sono necessari;
- la regola del *collasso*, che consente il collasso dei nodi interni rossi in singoli nodi nel caso siano soddisfatte determinate condizioni;
- la regola dell'*offuscamento dell'etichetta*, che consente la rimozione dell'etichetta dai nodi gialli e rossi.

Il processo di trasformazione viene applicato attraversando l'albero con una visita post-order, partendo, dunque, dai nodi foglia e applicando, in ordine, la regola di potatura, la regola del collasso e la regola dell'offuscamento dell'etichetta fino a risalire alla radice. La definizione formale di queste regole e l'algoritmo che le descrive sono anch'esse contenute in [1].

La regola di potatura

La regola di potatura agisce sui nodi interni n tali che n' e n'', figli di n, sono nodi foglia; questa regola è, in realtà, la composizione di due diverse regole, che sono applicate in modo esclusivo a seconda del colore del nodo interno. La regola dell'*operatore rosso* viene applicata se n è rosso e se la sua etichetta consiste in un operatore: nel caso n' o n'' siano rossi, vengono eliminati dall'elenco dei nodi figli di n e dall'albero. Questa regola trova giustificazione nella semplificazione della politica da rilasciare al client: si consideri la condizione *(Age > 18)* e si supponga che il nodo con etichetta $>$ e quello con etichetta *18* siano entrambi rossi; il rilascio della condizione *(Age _ _)* e del solo attributo *(Age)* sono del tutto equivalenti per il client. La regola dei *figli rossi* viene applicata se n non è rosso e se tutti i suoi nodi figli sono rossi: questi vengono rimossi dall'elenco dei nodi figli di n, che diventa, perciò, un nodo foglia, ed il colore di n è impostato a rosso. Anche questa regola è introdotta per semplificare la politica rilasciata: nel caso di due figli rossi, una funzione di colorazione ben definita permette unicamente che n sia giallo, ma

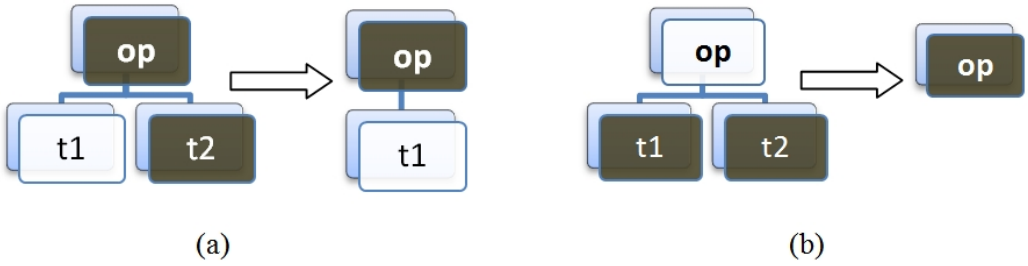


Figura 3.4: Regola di potatura: operatore rosso (a) e figli rossi (b)

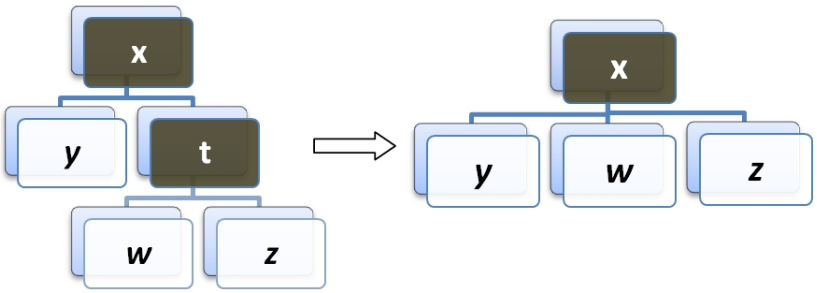


Figura 3.5: Regola del collasso

il rilascio della condizione *(_ _ _)* non ha significato per il client ed è, quindi, equivalente al rilascio di un singolo nodo foglia rosso, a dimostrare che esiste una condizione, ma che questa non è rilasciata al client. In Figura 3.4 viene visualmente mostrata l'applicazione della regola di potatura.

La regola del collasso

La regola del collasso opera sui nodi interni rossi n tali che n', figlio di n, sia rosso e non sia un nodo foglia. Come si è visto nel Paragrafo 3.3 , applicare una colorazione rossa ad un nodo interno ha ripercussioni sulla creazione degli insiemi di condizioni da rilasciare per la valutazione della politica. Dato che n' ha l'etichetta rossa, non è possibile sapere in che modo le condizioni rappresentate dai sottoalberi figli di n' saranno valutate; per questo n' viene eliminato e i sottoalberi figli prendono il suo posto come figli di n. In questo modo le diverse condizioni sono rilasciate separatamente al client; per lui non sarebbe comunque possibile sapere che relazione sussiste tra le valutazioni delle singole condizioni. In Figura 3.5 viene visualmente mostrata l'applicazione della regola del collasso.

La regola dell'offuscamento dell'etichetta

la regola dell'*offuscamento dell'etichetta* è l'unica che si applica effettivamente anche sui nodi foglia. Consente di rimuovere le etichette di quei nodi per i quali

Figura 3.6: Regola dell'offuscamento dell'etichetta

è stata definita una colorazione gialla o rossa. In Figura 3.6 viene visualmente mostrata l'applicazione della regola dell'offuscamento dell'etichetta.

3.4.1 Esempio di trasformazione

Si consideri il Policy Tree colorato mostrato in Figura 3.3. Il processo di trasformazione si applica innanzitutto sul sottoalbero riguardante la condizione *(c.type = CartaIdentità)*, lasciandolo, però, inalterato in quanto tutti i nodi al suo interno sono verdi. Successivamente viene preso in considerazione il sottoalbero *(c.LuogoDiNascita = Crema)*: come prima cosa vengono analizzati i nodi foglia, su entrambi dei quali si applica la regola dell'offuscamento dell'etichetta. Quando l'analisi passa al nodo genitore, l'applicazione della regola dei figli rossi consente di eliminare i nodi foglia, in quanto entrambi nodi rossi. A questo punto viene esaminato il nodo contenente l'etichetta $\wedge$, che mette in relazione le due condizioni sui certificati appena viste. Su questo nodo viene applicata solo la regola dell'offuscamento dell'etichetta, in quanto non sussistono le condizioni per applicare la regola di potatura dato che non si tratta di un nodo genitore esclusivamente di nodi foglia. L'analisi passa, quindi, al sottoalbero contenente la condizione *(Age > 18)*: l'unica regola applicabile è quella dell'offuscamento dell'etichetta sul nodo foglia contenente l'informazione *18*. La valutazione del nodo radice $\vee$ consente, invece, di applicare la regola del collasso in quanto nodo rosso dotato di nodo figlio rosso: questa regola permette l'eliminazione del nodo figlio — quello che conteneva l'etichetta $\wedge$ prima che questa fosse cancellata dalla regola dell'offuscamento dell'etichetta — e rende figli del nodo radice i due sottoalberi contenenti le condizioni sui certificati. Infine, viene applicata la regola dell'offuscamento dell'etichetta sul nodo radice. In Figura 3.7 viene mostrato il risultato dell'applicazione delle regole di trasformazione. Nell'Appendice A viene mostrata l'applicazione delle regole passo per passo mediante degli esempi visuali.

3.5 Proprietà della Vista

L'applicazione delle regole di trasformazione consentono, quindi, l'alterazione della formula booleana rappresentante la politica rispetto alla funzione di colorazione adottata; questo processo viene eseguito senza, però, effettuare alcun controllo sulla coerenza della Vista risultante rispetto al Policy Tree originale. L'assenza di questo genere di analisi può portare a situazioni in cui le informazioni rilasciate dal client soddisfino un numero insufficiente di condizioni al fine della valutazione della politica. Si consideri la politica rappresentata dalla formula mostrata nell'Esempio 4.

$$(Country = Italy) \wedge (Age > 18) \quad \textbf{(4)}$$

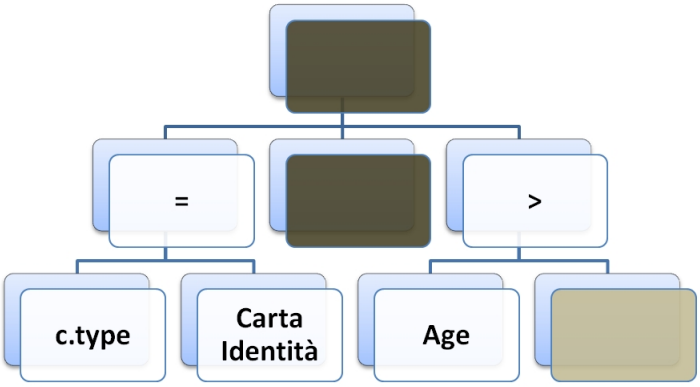


Figura 3.7: Risultato dell'applicazione delle regole di trasformazione

Se la Vista sul Policy Tree ricavata da una funzione di colorazione ben definita prevede il rilascio del solo attributo *Country*, anche nel caso l'utente dichiari di essere cittadino italiano e, quindi, soddisfi la condizione relativa, l'accesso non sarà comunque permesso in quanto la politica non può essere valutata senza l'informazione riguardo l'attributo *Age*. Si rende, dunque, necessario un confronto tra la politica originale per il controllo dell'accesso e quella derivata dalla Vista sul Policy Tree: tale confronto si basa sulla definizione di Set d'informazioni e, a partire da questo, è possibile individuare alcune proprietà che caratterizzano la Vista stessa.

3.5.1 Il concetto di Set

Si consideri la politica rappresentata dalla formula mostrata nell'Esempio 5.

$$(Country = Italy) \vee ((Age > 18) \wedge (Age < 60)) \textbf{ (5)}$$

Nel caso riportato, l'accesso viene consentito se il client si dichiara cittadino italiano oppure se dichiara un'età compresa tra i 18 ed i 60 anni. Non è necessario che siano soddisfatte entrambe le condizioni: basta che una sola di esse sia valutata positivamente perchè l'accesso sia garantito. Si consideri, invece, la politica mostrata nell'Esempio 6, simile a quella mostrata nell'Esempio 5, ma con un'importante differenza.

$$(Country = Italy) \wedge ((Age > 18) \wedge (Age < 60)) \textbf{ (6)}$$

In questo caso è invece necessario soddisfare contemporaneamente entrambe le condizioni affinchè l'accesso sia consentito.

Si definisce allora un Set s come l'insieme di etichette di quegli attributi la cui conoscenza permette la valutazione della politica per il controllo dell'accesso. Nel primo esempio si avranno due differenti Set $s = \{(Country)\}$ e $s' = \{(Age)\}$, mentre nel secondo si avrà un unico Set, ovvero $s = \{(Country), (Age)\}$. Particolare attenzione va data alle condizioni riguardanti i certificati: esse necessitano di un trattamento differente rispetto alle altre in quanto sono, come si è visto, visibili o invisibili nella loro interezza. Al momento della creazione dei Set saranno dunque considerate come attributi atomici complessi e la loro etichetta sarà costituita dalla condizione stessa. Si consideri la formula mostrata nell'Esempio 7.

$$(c.type = CartaIdentità) \wedge (Age > 18) \textbf{ (7)}$$

Essa porterà alla creazione del Set $s = \{(c.type = CartaIdentità),(Age)\}$. Il numero di Set risultanti da una certa politica dipende quindi strettamente dal numero e dell'etichetta dei nodi contenuti nel Policy Tree. Il discorso fatto fino ad ora è valido per ricavare i Set da un qualsiasi Policy Tree colorato; ricavare l'insieme dei Set dalla Vista relativa è un'operazione resa più complicata dal fatto che questa può includere al suo interno nodi interni rossi e gialli, che sono, quindi, senza etichetta. In questo caso, non potendo conoscere le relazioni che intercorrono tra due differenti condizioni, si assume che tali nodi abbiano come etichetta l'operatore logico $\wedge$: così facendo si ha la sicurezza che la politica possa essere valutata.

L'insieme di tutti Set ricavabili da un Policy Tree colorato sarà chiamato *minima collezione di rilascio relativa al Policy Tree* e sarà indicato con α. Con il termine di *minima collezione di rilascio relativa alla Vista sul Policy Tree* sarà invece indicato l'insieme dei Set ricavati dalla Vista; questa collezione sarà indicata con α'.

3.5.2 Analisi della Vista: verifica delle proprietà

Individuare le proprietà caratterizzanti una Vista relativa ad un Policy Tree consente al server di determinare se essa ha significato per il client; questa operazione è fattibile mediante il confronto fra le minime collezioni di rilascio.

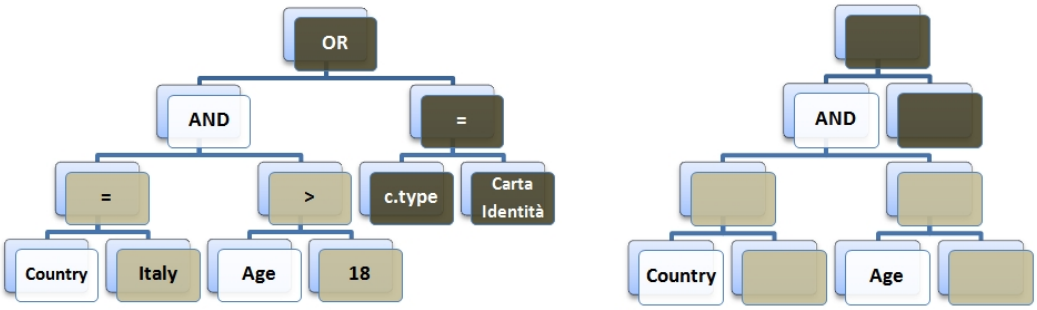
Prima di caratterizzare le diverse proprietà è necessario introdurre una relazione chiamata *copertura* che si rivela particolarmente utile per il confronto dei Set delle minime collezioni di rilascio: un Set s "copre" un altro Set s' se ogni certificato, dichiarazione o attributo complesso presente in s' è anche presente in s. Questo significa, in pratica, che rilasciare le informazioni contenute in s significa rilasciare anche tutte quelle contenute in s', con l'eventuale aggiunta di informazioni superflue.

Vista Equa

Intuitivamente, una Vista è valida e ha significato per il client se α' è un sottoinsieme di α: in questo caso, tutti i Set ricavati dalla Vista permettono di valutare l'originale politica per il controllo dell'accesso. Una vista di questo tipo gode della proprietà di *equità*; la definizione formale di *Vista equa* è contenuta in [1]: riferendosi alla relazione di copertura appena definita, si può verificare che la proprietà di equità è propria di una Vista se per ogni s' appartenente a α' esiste almeno un s appartenente a α tale che s "copra" s' e viceversa. Si consideri, come esempio di Vista equa, la Tabella di Set 3.1.

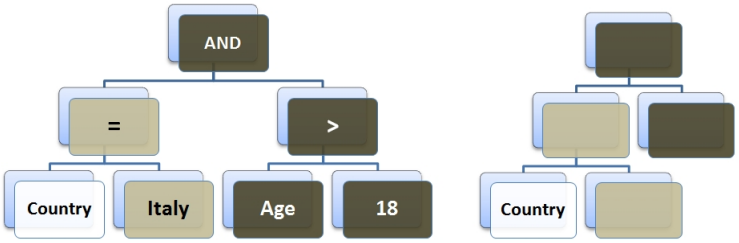
Vista non equa

In modo opposto, una Vista può caratterizzarsi come *non equa* se esiste almeno un Set s' in α' che non "copre" alcun Set s in α. Ciò significa che, nel caso venga rilasciato s' al client, le informazioni richieste non sono in ogni caso sufficienti per la valutazione della politica. Non è, quindi, conveniente rilasciare al client una Vista che gode di questa proprietà. Si consideri, come esempio di Vista non equa, la Tabella di Set 3.2.



α	α'
s^1= {(Country),(Age)}	s'^1= {(Country),(Age)}
s^2= {(c.type = CartaIdentità)}	

Tabella 3.1: Vista equa

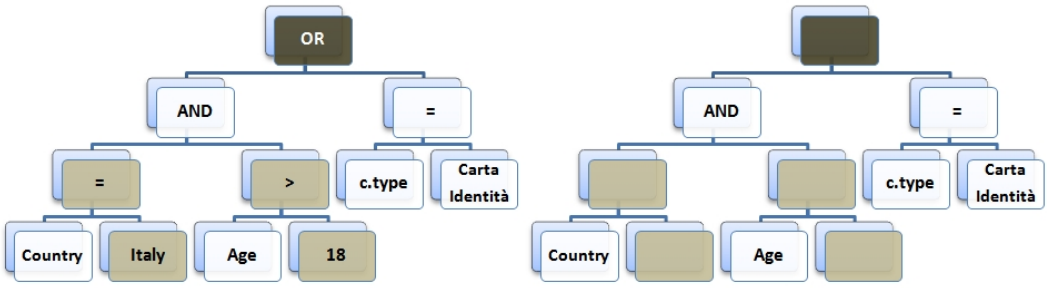


α	α'
s^1= {(Country),(Age)}	s'^1= {(Country)}

Tabella 3.2: Vista non equa

Vista superflua

Una Vista può anche essere caratterizzata come *superflua* se i Set appartenenti a α' richiedono al client più informazioni di quelle necessarie per la valutazione della politica. Questo è di solito una conseguenza della presenza di un nodo interno contenente l'etichetta $\vee$ e colorato di giallo o rosso: nella creazione dei Set a partire dalla Vista, l'etichetta di questo nodo viene considerata come $\wedge$ e le condizioni che dovrebbero essere memorizzate in due Set distinti, sono invece assorbite in un unico Set. Si noti che il client può comunque ottenere l'accesso anche se la Vista a lui rilasciata è caratterizzata come superflua. Si consideri, come esempio di Vista superflua, la Tabella di Set 3.3.

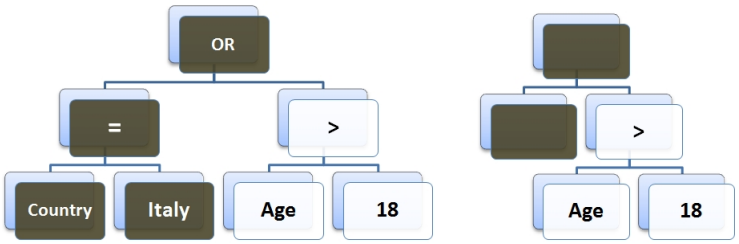


α	α'
s^1= {(Country),(Age)}	s'^1= {(Country),(Age),(c.type = CartaIdenità)}
s^2= {(c.type = CartaIdentità)}	

Tabella 3.3: Vista superflua

Vista pre-analizzabile

Se, infine, il client può valutare autonomamente tutte le condizioni che sono contenute in un Set s' appartenente a α', la Vista si caratterizza come *pre-analizzabile*; in questo caso il client può valutare autonomamente se le condizioni rilasciate dal server saranno valutate positivamente o meno. Il Set s' deve, però, contenere un numero almeno sufficiente di condizioni che consentano la valutazione della politica: la Vista deve, cioè, prima essere equa o superflua per poter poi verificare se è anche pre-analizzabile. Si consideri, come esempio di Vista pre-analizzabile, la Tabella di Set 3.4.



α	α'
s^1= {(Country)}	s'^1= {(Age > 18)}
s^2= {(Age > 18)}	

Tabella 3.4: Vista pre-analizzabile

Capitolo 4

L'implementazione

A partire dagli aspetti teorici alla base del rilascio selettivo delle politiche per il controllo dell'accesso analizzati nel capitolo precedente, è stata sviluppata un applicazione interattiva che gestisce:

- la *creazione del Policy Tree* a partire da una formula booleana inserita dall'utente;
- la *colorazione del Policy Tree* in base alle preferenze dell'utente, ma rispettando comunque le regole per la definizione di una colorazione ben definita;
- la *trasformazione del Policy Tree nella Vista relativa*, che viene poi comunicata all'utente;
- la *creazione di* α *e* α', il loro *confronto* e l'*analisi delle proprietà della Vista*; queste ultime sono comunicate all'utente;

Per dare meglio l'idea di come è stato implementato l'applicativo, nel corso della trattazione saranno utilizzati delle immagini che mostrano un'interfaccia grafica realizzata sulla base del programma, dei brevi frammenti del codice Java realizzato — per una più completa descrizione di quest'ultimo si veda l'Appendice B — e dei brevi esempi grafici che mostrano il funzionamento di alcuni metodi implementati.

L'applicazione realizzata è stata scritta in linguaggio Java utilizzando l'ambiente di sviluppo integrato Eclipse SDK 3.6.1 Helios. Come parser per la formula booleana è stato utilizzato JavaCC, installato come plug-in di Eclipse e, quindi, integrato con esso.

4.1 Parser della formula booleana

L'applicazione ha come punto di partenza la richiesta all'utente della formula booleana rappresentante la politica per il controllo dell'accesso. Questa richiesta viene implementata mediante l'uso di un'interfaccia grafica, mostrata in Figura 4.1, che consente di memorizzare la formula inserita in una variabile di tipo `String`; in alternativa all'utilizzo dell'interfaccia grafica, è possibile ricavare la formula da un file di testo o da riga di comando. Data l'importanza della formula inserita nel prosieguo del processo d'elaborazione, si sono resi necessari opportuni controlli su

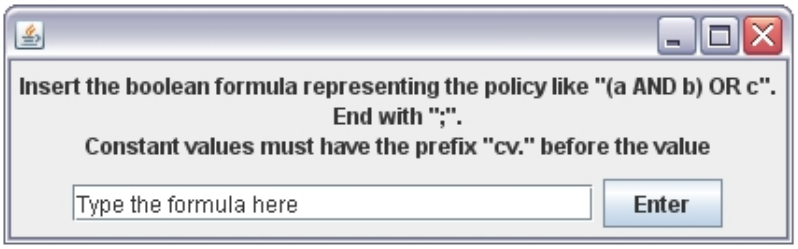


Figura 4.1: Interfaccia grafica per la richiesta della formula

di essa per verificarne la validità sintattica. La variabile in cui è memorizzata la formula viene utilizzata come input per il parser JavaCC, che è l'effettivo autore dei controlli. Essi si basano su una serie di requisiti, alcuni delle quali esplicitamente specificati all'utente prima che questo possa inserire la formula:

- i valori costanti devono contenere il prefisso `cv.`;
- gli operatori logici $\wedge$ e $\vee$ devono essere specificati mediante la notazione `AND` e `OR`;
- le parentesi tonde in cui sono contenute le singole condizioni devono essere annidate correttamente;
- la formula booleana deve terminare con un "`;`" ;

JavaCC permette un'accurata analisi lessicale della formula inserita, consentendo di gestire in maniera efficiente gli operatori, gli operandi, le parentesi e gli spazi vuoti e sollevando interruzioni specifiche nel caso in cui alcuni dei vincoli non siano stati rispettati. Tale analisi mira a distinguere gli elementi inutili (come gli spazi), che non saranno considerati, da quelli necessari, identificando le relazioni gerarchiche che intercorrono tra di essi. In Figura 4.2 è riportato un esempio di eccezione sollevata a causa dell'errato annidamento delle parentesi.

4.2 Creazione del Policy Tree

Una volta che la formula booleana è stata riconosciuta come lessicalmente corretta, il parser gestisce la creazione di un vettore utilizzando una struttura dati di tipo `Vector`: quando riconosce un operatore o un operando, aggiunge al vettore un oggetto di classe `Nodo`, contenente l'informazione corrispondente. Il vettore risultante, che sarà chiamato `Albero`, non è semplicemente un insieme sconnesso di oggetti, ma assume le caratteristiche di una struttura dati ad albero i cui nodi sono, appunto, gli oggetti `Nodo`.

Si rende ora necessaria una breve analisi della classe `Nodo` allo scopo di comprendere in che modo gli oggetti di questa classe possono effettivamente rappresentare i nodi di quello che diventerà il Policy Tree da elaborare. Di seguito sono riportate le variabili istanza contenute in un oggetto di tale classe.

```
private int nodeKey;
```

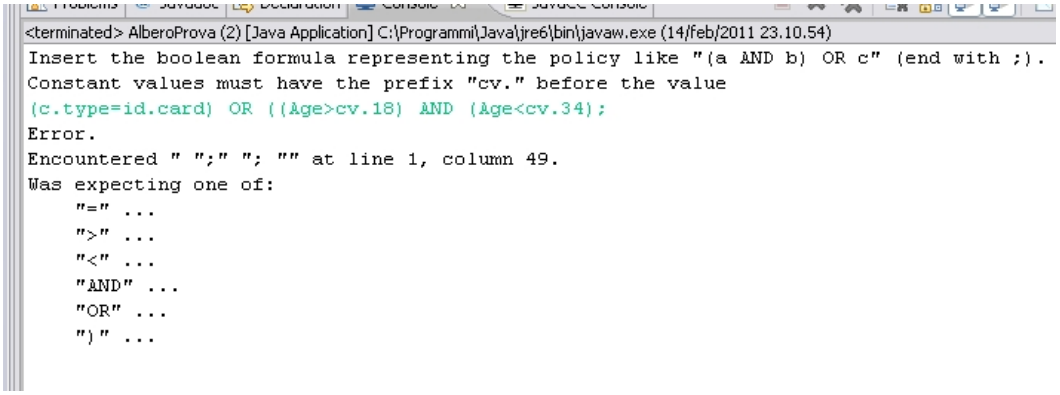


Figura 4.2: Esempio di errore sollevato da JavaCC

```
private static int contatoreNodi = 0;
private String info;
private Vector<Nodo> sons;
private Nodo parent;
private String colour;
```

La variabile statica `contatoreNodi` consente la numerazione sequenziale dei nodi creati: al momento della creazione di un oggetto `Nodo` viene assegnato il suo valore incrementato di un'unità alla variabile `nodeKey`. Come si vedrà in seguito, la numerazione dei nodi sarà essenziale per la gestione dell'albero e per riferirsi in modo efficiente a ciascuno di essi. Nella variabile `info` viene memorizzata l'informazione rappresentata dall'operatore o dall'operando. Tale informazione è memorizzata come stringa di caratteri, anche nel caso di valori costanti utilizzati nei confronti. La variabile `colour` verrà utilizzata per specificare il colore del nodo, in modo da assegnarvi il grado di visibilità voluto. Le variabili istanza che assumono maggiore importanza all'interno della classe sono la variabile di tipo `Nodo parent`, che rappresenta il riferimento al nodo genitore del nodo creato, ed il vettore di oggetti `Nodo sons`, in cui sono contenuti i riferimenti ai nodi figli del nodo creato. Ovviamente, nel caso di un nodo foglia il vettore `sons` sarà vuoto. L'analisi dell'importanza di queste variabili nella creazione e nella successiva gestione dell'albero è rimandata al Paragrafo 4.2.1. Grazie, inoltre, all'implementazione dei metodi `Getter` e `Setter` per ciascuna di queste variabili, qualsiasi informazione riguardante un particolare oggetto `Nodo` è immediatamente reperibile.

4.2.1 Collegamenti fra i nodi

Si è visto come ciascun oggetto `Nodo` contenga un riferimento sia al nodo genitore sia ai nodi figli. In realtà, al momento della creazione di ciascun oggetto viene istanziata unicamente la variabile `parent`, che si riferisce al nodo genitore. Questa scelta trova giustificazione nell'ordine di creazione dei nodi: analizzando la formula booleana il parser individua, prima di tutto, l'informazione corrispondente al nodo radice, creando un oggetto `Nodo` n e impostando la variabile `parent` di n a n stesso. Subito dopo vengono creati altri due oggetti `Nodo`, n' e n'', corrispondenti ai nodi figli

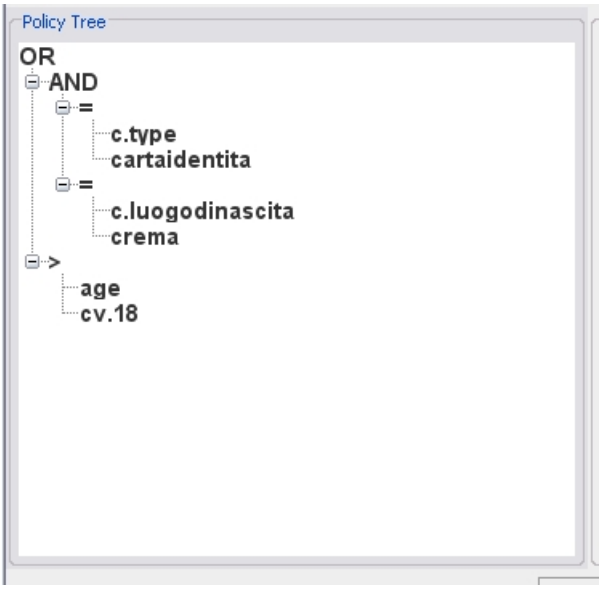


Figura 4.3: Rappresentazione del Policy Tree

di n, nella cui variabile `parent` è impostato il riferimento a n. A questo punto viene effettuato un controllo prima su n' e successivamente su n'' allo scopo di verificare che la variabile `info` non contenga un operatore; se così è, vengono creati altri due nuovi oggetti `Nodo` figli di quello in esame e l'analisi procede sui nodi appena creati. Al momento della creazione di un oggetto non si hanno, perciò, informazioni riguardo i nodi figli, ma unicamente riguardo al nodo genitore. Proprio per questo il riempimento dei vettori `sons` viene effettuato mediante un apposito metodo solo una volta che l'intero albero è stato creato. Si noti che nell'albero creato il vettore `sons` di ciascun nodo contiene esattamente il riferimento a 0 o 2 nodi figli: questo è una conseguenza delle assunzioni fatte nel Paragrafo 3.2 circa la rappresentazione della politica mediante la formula booleana. Una struttura dati di tipo `Vector` per i riferimenti ai nodi figli consente, ad ogni modo, di contenere un numero arbitrario di oggetti; l'efficienza e l'utilità dell'utilizzo di una struttura di questo genere anziché di una serie sconnessa di oggetti `Nodo` sarà evidenziata in modo particolare nel processo di trasformazione.

Il fatto di avere, per ogni oggetto, un riferimento al nodo genitore ed ai nodi figli ha il grande vantaggio di permettere il raggiungimento di qualsiasi nodo del vettore `Albero` a partire da un dato oggetto `Nodo`. Tale vantaggio sarà sfruttato massivamente nei processi di colorazione e trasformazione. In Figura 4.3 è possibile vedere la rappresentazione di un Policy Tree mediante l'interfaccia grafica utilizzata.

4.3 Colorazione del Policy Tree

La scelta progettuale adottata prevede di non specificare nessun colore di default al momento della creazione di un oggetto `Nodo`, ma di richiederlo all'utente una volta conclusa la creazione del vettore `Albero`. I colori che è possibile utilizzare sono tre (`red`, `yellow`, `green`) e la scelta di uno o degli altri ha gli effetti di visibilità

rispettivamente del colore rosso, giallo e verde discussi nel Paragrafo 3.3.

L'algoritmo di colorazione è in realtà composto da due diversi metodi, chiamati `NonLeafNodesColouring` e `LeafNodesColouring`; a partire dal vettore `Albero`, vengono istanziati altri due vettori di oggetti `Nodo`, contenenti rispettivamente i nodi non-foglia ed i nodi foglia. Tale divisione viene effettuata mediante un controllo sul vettore `sons` dei diversi nodi allo scopo di verificare se sia vuoto o meno. Sugli elementi del primo vettore sarà invocato il metodo `NonLeafNodesColouring`, mentre su quelli del secondo il metodo `LeafNodesColouring`.

L'approccio scelto consente, all'interno del metodo `NonLeafNodesColouring`, di avvisare l'utente se ci sono particolari accorgimenti da adottare nella colorazione dei nodi non foglia, al fine di aderire alle regole per ottenere una funzione di colorazione ben definita, ma consentendo comunque di inserire qualsiasi colorazione tra le tre possibili. Se, ad esempio, il `Nodo` n', figlio di n per il quale si sta richiedendo la colorazione, contiene un valore costante nella variabile `info`, l'utente viene avvisato specificando che se n sarà dichiarato `red`, n' non potrà essere dichiarato `green`. L'unica eccezione consiste nel caso in cui il `Nodo` n', figlio di n per il quale si sta richiedendo la colorazione, contiene un attributo certificato: in questo specifica situazione non è permessa la colorazione `yellow` per il `Nodo` n. Quando, in un secondo momento, il processo d'elaborazione invoca il metodo `LeafNodesColouring`, viene impedita la scelta di determinate colorazioni in base alle regole ed ai colori precedentemente scelti per i nodi genitore.

4.3.1 Inserimento della colorazione

Una volta definita la possibile gamma di colorazioni che è possibile specificare per un determinato nodo, viene utilizzata una semplice routine che, mediante l'interfaccia grafica, richiede all'utente di inserire la stringa corrispondente al colore scelto per quel nodo. Si noti che il metodo per l'inserimento dei colori di un nodo può provocare due tipi di errori: il primo viene sollevato se la stringa rappresentante il colore inserito non equivale a `red`, `yellow` o `green` (per motivi di praticità d'utilizzo viene eseguito un controllo non "case sensitive"); in questo caso l'utente viene avvisato che il colore inserito non esiste. Il secondo viene lanciato se la stringa coincide con un colore non permesso per quel nodo, in quanto viola una particolare regola alla base di una funzione di colorazione ben definita; in questo caso l'utente viene avvisato che il colore inserito non è permesso. Se non vengono sollevate errori, la stringa rappresentante il colore viene inserita nella variabile `color` dello specifico oggetto `Nodo`.

Una possibilità alternativa che non da luogo al sollevamento di errori consiste nel gestire la colorazione mediante dei bottoni, che diventano visibili o invisibili in base alla specifiche condizioni che vengono verificate su un particolare attributo.

In Figura 4.4 è mostrata la rappresentazione del vettore `Albero` colorato.

4.3.2 Problematiche relative al processo di colorazione

La prima problematica da affrontare nello sviluppo del codice per la funzione di colorazione riguarda l'individuazione delle condizioni che riguardano dei certificati o dei confronti con valori costanti; questi due tipi di condizioni sono infatti regolate

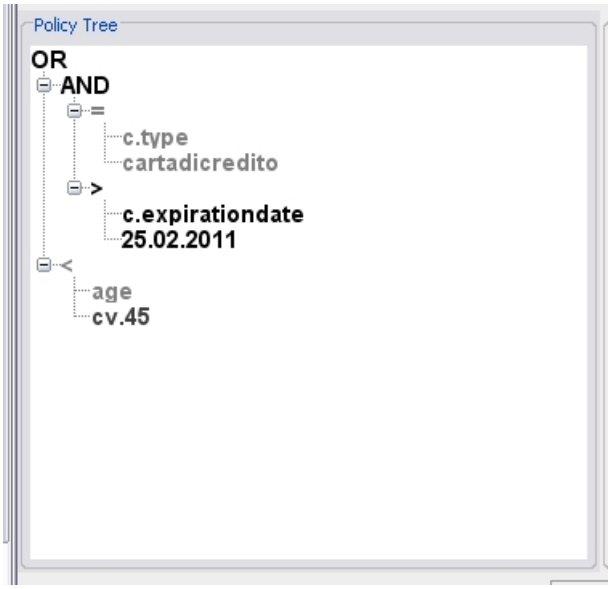


Figura 4.4: Rappresentazione del Policy Tree Colorato

da precise indicazione nella definizione di funzione di colorazione ben definita. La soluzione adottata consiste nell'analisi della variabile `info` dei nodi foglia, la quale è facilmente recuperabile grazie ai riferimenti ai nodi genitori e ai nodi figli presenti negli oggetti `Nodo`. Per verificare che una condizione riguardi attributi certificati, si controlla che tale variabile abbia un prefisso uguale a `c` e contenga la parola chiave `type` al suo interno. Per dimostrare, invece, che una condizione riguardi valori costanti, si controlla che la variabile abbia un prefisso uguale a `cv.`; l'uso di questo prefisso viene esplicitamente richiesto all'utente così come spiegato nel Paragrafo 4.1.

Un secondo problema nasce dalla necessità di settare automaticamente a `green` la variabile `color` del nodo n'', fratello del nodo n' dichiarato `green` e contenente nella variabile `info` un attributo costante. Per motivi di efficienza, si rende doveroso, in questo caso, richiedere sempre prima la colorazione di n'; se questo viene dichiarato `green`, n'' è impostato anch'esso a `green`, in caso contrario si invoca la normale routine di colorazione. Anche questo problema trova facile soluzione grazie ai riferimenti tra nodi parenti: è, infatti, sufficiente riferirsi al nodo genitore, effettuare un controllo per verificare se uno dei nodi figli contiene un valore costante e, in caso affermativo, richiedere subito la colorazione per quel nodo, posticipando o automatizzando quella del nodo fratello. Si chiarisce, quindi, ancora di più l'importanza dei riferimenti agli altri nodi all'interno dell'oggetto `Nodo`: senza di essi si renderebbe ogni volta necessaria la visita di tutto l'albero, unita a complicati controlli incrociati.

4.4 Trasformazione del Policy Tree

La trasformazione del Policy Tree in base alla funzione di colorazione adottata costituisce il nucleo centrale dell'applicazione, in quanto consente la creazione della

Vista che sarà poi valutata e rilasciata al client.

Per le caratteristiche delle regole di trasformazione applicate, non è pensabile applicare tali regole sui nodi in base alla loro numerazione sequenziale. Si rende, invece, necessario un metodo ricorsivo, che nell'applicazione è chiamato `transform`: esso inizialmente prende come argomento il nodo radice e si applica poi ricorsivamente sui nodi figli, applicando i metodi che implementano le regole vere e proprie a partire dai nodi foglia.

Prima di lanciare il processo di trasformazione, si è rivelato utile creare per ciascun oggetto `Nodo` — fatta eccezione per i nodi foglia — un vettore contenente i riferimenti ai propri nodi figli per cui è stata settata la colorazione `red`, in modo da ottimizzarne la ricerca: la struttura dati risultante è sempre un vettore di oggetti `Nodo` chiamato `redSons`.

Si noti che può esserci la possibilità che nel corso del processo di trasformazione un oggetto `Nodo` si trovi con riferimenti a più di due figli nel proprio vettore `sons`; grazie alla struttura dati utilizzata, questa possibilità non rappresenta comunque un problema. Si noti, inoltre, che, ad eccezione della variabile `nodeKey`, che rimane fissa, tutte le variabili istanza degli oggetti `Nodo` possono essere modificate dal processo di trasformazione: il metodo che implementa la regola dei figli rossi può, infatti, modificare la variabile `color`, così come i riferimenti ai nodi parenti e ai nodi figli possono essere modificati dai metodi che implementano la regola del collasso e della potatura. La variabile `info` può infine essere modificata dalla regola dell'offuscamento dell'etichetta. Questa possibilità si traduce nell'assenza di strutture dati superflue che avrebbero significato uno spreco di risorse e di tempo di computazione.

4.4.1 Applicazione delle regole di trasformazione

Si rende innanzitutto necessario analizzare le variabili che dovranno essere passati come argomenti ai metodi che implementano la regola di potatura e la regola del collasso. Oltre al nodo n su cui il metodo dovrà essere applicato, sarà necessario utilizzare il vettore `redSons` di n ed il vettore `Albero`, data l'eventuale necessità di eliminare oggetti `Nodo` da esso.

É importante notare che la regola di potatura e quella del collasso si applicano sui nodi in modo esclusivo: la discriminante è rappresentata dalla condizione di essere nodi genitori di nodi foglia o nodi interni. La soluzione adottata per gestire questa invocazione esclusiva viene spiegata nel Paragrafo 4.4.2.

L'applicabilità della regola di potatura, implementata nel metodo `prune`, sul `Nodo` n è legata alla variabile `color` di n, che deve essere impostata a `red`, oppure al numero di elementi del vettore `redSons` di n, che deve essere uguale a 2. Nel primo caso, mediante un ciclo interno finalizzato a verificare l'effettiva presenza di nodi appartenenti al vettore `redSons`, i nodi figli di n impostati a `red` sono eliminati dal vettore `Albero` e dal vettore `sons` di n. Nel secondo caso è sufficiente richiamare il metodo `clear()` della classe Java `Vector` sul vettore `sons` di n, eliminare i nodi figli dal vettore `Albero` e settare la varianile `color` di n a `red`.

La regola del collasso, implementata nel metodo `collapse`, è quella che richiede maggiore sforzo computazionale, in quanto agisce su un numero di livelli di profondità dell'albero superiore a due e gestisce la copia di interi vettori di oggetti `Nodo`. La discriminante di applicabilità riguarda il colore del nodo n, che deve essere impo-

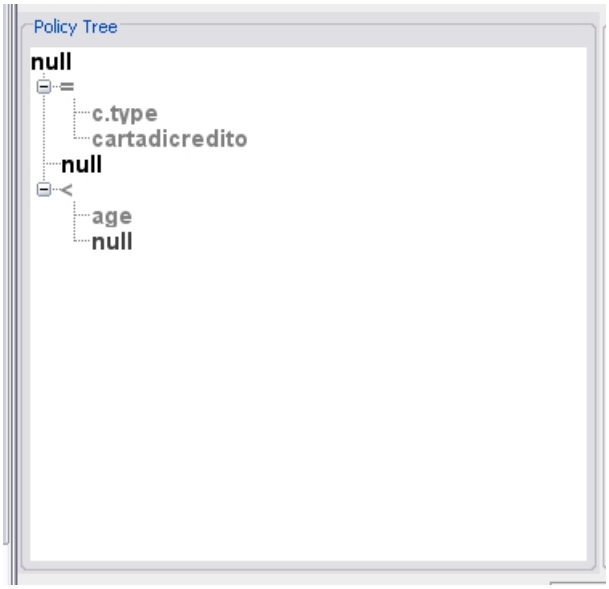


Figura 4.5: Rappresentazione della Vista sul Policy Tree

stato a `red`. Se così è, mediante un ciclo si verifica la presenza di nodi figli anch'essi impostati a `red` e, nel caso esistano, vengono eseguite una serie di operazioni. Prima di tutto il vettore `sons` del figlio rosso n' viene aggiunto al vettore `sons` di n; successivamente n' viene eliminato dal vettore `Albero` e dal vettore `sons` di n; infine viene eseguito un ciclo sul vettore `sons` di n per settare tutte le variabili `parent` a n. Quest'ultima operazione viene effettuata in quanto tutti i nodi nipoti divenuti nodi figli di n contengono ancora il riferimento a n' nella variabile `parent`.

La regola dell'offuscamento dell'etichetta, implementata nel metodo `hideLabel`, è la più immediata: il metodo prende in argomento solo il nodo su cui deve essere applicata, verifica il suo colore e, nel caso quest'ultimo sia `red` o `yellow`, setta la variabile `info` ad uno spazio vuoto. Il motivo di questo assegnamento è spiegato nel Paragrafo 4.4.2.

Applicando i tre metodi specificati sul vettore `Albero` rappresentante il Policy Tree, si ottiene un vettore analogo, che sarà chiamato `Albero`$'$, il quale rappresenta la Vista relativa al Policy Tree. In Figura 4.5 viene mostrata la rappresentazione della Vista mediante l'interfaccia grafica.

4.4.2 Problematiche relative al processo di trasformazione

Le problematiche riscontrabili nell'applicazione delle regole di trasformazione sul vettore `Albero` sono decisamente più complesse rispetto a quelle evidenziate nel processo di colorazione. Oltre a risolvere tali problemi, è inoltre necessario adottare alcuni accorgimenti per ottimizzare il tempo di elaborazione, evitando di verificare condizioni ed effettuare cicli sui nodi dell'albero non interessati dal processo di trasformazione.

Prima di tutto, come si è visto, per ogni nodo non foglia n è necessario valutare se applicare il metodo `prune` oppure il metodo `collapse`. All'interno di `transform` è

stata, perciò, creata una variabile booleana, che viene posta a `true` nel caso almeno uno dei nodi figli di n abbia, a sua volta, dei nodi figli. In base, quindi, al valore della variabile viene applicato sul nodo l'uno o l'altro metodo.

Un'attenta analisi del ciclo di elaborazione consente, poi, di verificare che il metodo `hideLabel` è l'unico che viene effettivamente applicato sui nodi foglia: per ottimizzare la computazione viene fatto, quindi, un controllo preliminare sui nodi per verificare se il vettore `sons` è vuoto: in caso negativo si valuta quale metodo applicare tra `prune` e `collapse`, in caso positivo si applica immediatamente `hideLabel`.

Un'ulteriore problema può derivare da una gestione non attenta dei vettori `sons`: nei diversi cicli contenuti nel metodo `collapse` può capitare di invocare il metodo `Getter` per tali vettori su dei nodi foglia, sollevando, quindi, errori di computazione derivanti da eccezioni `NullPointerException`. Per ovviare a questa problematica è necessario inserire dei controlli in diversi punti che verifichino se questi vettori sono vuoti o meno. Un problema simile è riscontrabile anche nel metodo `hideLabel`: impostare la variabile `info` a `null` non sarebbe conveniente proprio perchè verrebbe sollevata un'eccezione nel caso in cui si cerchi di leggerla. Proprio per questo la soluzione proposta imposta la variabile ad uno spazio vuoto.

4.5 Creazione e confronto delle collezioni di Set

Sebbene il confronto tra le minime collezioni di rilascio per la verifica della relazione di copertura sia visualmente abbastanza semplice ed intuitivo, la sua implementazione in codice è complicata da diversi fattori, primo fra tutti la necessità di utilizzare una struttura dati adatta a contenere le diverse condizioni che, insieme, permettono la valutazione della politica. Non è, infatti, possibile basare il confronto sugli oggetti `Nodo`, in quanto la copiatura in una sola variabile di tipo `String` di più informazioni risulterebbe scomoda ed inefficiente. La scelta progettuale adottata si basa, allora, sulla definizione di una nuova classe `Set` e sulla rappresentazione, mediante gli oggetti di questa classe, dei diversi Set di cui sono composte le collezioni di rilascio.

Prima di passare alla descrizione delle variabili istanza contenute in un oggetto `Set` è opportuno puntualizzare alcune considerazioni che sono state fatte prima dell'implementazione del codice.

La creazione delle minime collezioni di rilascio viene implementata mediante due metodi distinti. Tale distinzione deriva dall'esigenza di dividere il momento di creazione dei singoli Set iniziali da quello di confronto con i Set della stessa collezione, terminante nella creazione dei Set definitivi. Il primo metodo, chiamato `setCreation`, consente la creazione di tanti oggetti `Set` quante sono le condizioni `operando1 operatore operando2` contenute nella politica, dove `operando1` e `operando2` sono nodi foglia. Vengono, quindi, memorizzate in una variabile di tipo `String` contenuta in un vettore chiamato `vectorInfo` — una delle variabili istanza dell'oggetto `Set` — le informazioni contenute nelle variabili `info` degli oggetti `Nodo` corrispondenti, nell'ordine, al nodo figlio sinistro, al nodo genitore e al nodo figlio destro. Si trascende, quindi, dalla colorazione applicata ai nodi, con la sola eccezione dei nodi foglia contenenti dei valori costanti: in questo caso, si assume che se il nodo n', contenente un valore costante, ed il nodo genitore n hanno entrambi una colorazione `green`, allora il vettore `vectorInfo` dell'oggetto `Set` dovrà contenere la

condizione completa; in caso contrario conterrà unicamente l'attributo contenuto nella variabile `info` del nodo n'' fratello di n'. Questa scelta progettuale deriva dalla convinzione che una condizione di questo genere ha significato solo se è rivelata completamente o se è rivelata unicamente l'informazione riguardante l'attributo. Si consideri, come esempio, il Policy Tree colorato mostrato in Figura 4.4. Gli oggetti `Set` creati dal metodo `setCreation` sono tre, ciascuno contenente un'unica variabile `String` nel proprio `vectorInfo`: il primo conterrà `c.type = CartaDiCredito`, il secondo `c.expirationDate > 25.02.2011`, mentre il terzo `age`.

Il secondo metodo, chiamato `setAnalysis`, analizza la variabile `parent` — un'altra variabile istanza della classe `Set` — di ogni oggetto `Set` creato dal metodo `setCreation`. Tale variabile contiene un riferimento al nodo che nel vettore `Albero` rappresenta il genitore del nodo la cui variabile `info` consiste nell'operatore della condizione memorizzata in `vectorInfo`. Il metodo effettua la creazione di un nuovo oggetto `Set` nel caso in cui la variabile `parent` di due oggetti `Set` diversi coincida e contenga nella variabile `info` l'operatore logico $\wedge$: il nuovo oggetto `Set` conterrà all'interno del proprio `vectorInfo` tutto il contenuto dei `vectorInfo` dei Set da cui è generato, che vengono successivamente eliminati dalla collezione. Nel caso in cui, invece, il valore della variabile `info` corrisponda all'operatore logico $\vee$, il metodo permette di mantenere i due oggetti `Set` separati senza più lasciare la possibilità che possano essere uniti in un unico oggetto. Una volta che il confronto è stato effettuato, il riferimento al nodo contenuto nella variabile `parent` viene sovrascritto dal riferimento al nodo genitore. In realtà, il confronto fra due oggetti `Set` viene effettuato anche sulla base di un'altra variabile, chiamata `livello`, la cui trattazione è rimandata al Paragrafo 4.5.1. Il metodo è, poi, invocato ricorsivamente analizzando i nodi genitori dei nodi referenziati nella variabile `parent`, fino ad arrivare al nodo radice del vettore di oggetti `Nodo` su cui è invocato.

In Tabella 4.1 viene mostrato visualmente come opera il metodo `setAnalysis`. Dato che i due `Set` s e s' hanno entrambi il riferimento al nodo n' nella variabile `parent` e n' ha come etichetta l'operatore logico $\wedge$, viene creato un nuovo oggetto `Set` s''', il cui vettore `vectorInfo` conterra due stringhe, `c.type = cartaDiCredito` e `c.expirationDate > 25.02.2011`. La variabile `parent` di s''' conterrà il riferimento a n, genitore di n'. Dopo che s e s' sono stati eliminati dalla collezione, viene eseguito il confronto tra s'' e s''', dato che entrambi contengono il riferimento a n nella variabile `parent`: poichè n ha come etichetta $\vee$, i due Set vengono mantenuti distinti.

Si noti che, nell'invocazione del metodo `setAnalysis` sul vettore `Albero`$'$, nel caso il riferimento contenuto nella variabile `parent` di un oggetto `Set` corrisponda ad un nodo rosso o giallo il metodo opera come se la variabile `info` del nodo stesso contenga l'operatore $\wedge$.

L'implementazione dei metodi appena descritti non sarebbe stata possibile senza l'uso di alcune variabili istanza particolari, che saranno descritte nel Paragrafo 4.5.1.

Vengono infine creati due vettori di oggetti `Set` a cui sono aggiunti rispettivamente gli oggetti derivati dal vettore `Albero` e quelli derivati dal vettore `Albero`$'$. Questi due vettori, che rappresentano le due minime collezioni di rilascio, saranno utilizzati per il confronto e l'analisi delle proprietà della Vista.

Set	*VectorInfo*	*Parent*
s	{`c.type = cartaDiCredito`}	n′(`AND`)
s′	{`c.expirationDate > 25.01.2001`}	n′(`AND`)
s″	{`age`}	n(`OR`)

⇓

Set	*VectorInfo*	*Parent*
s″	{`age`}	n(`OR`)
s‴	{`c.type = cartaDiCredito`},{`c.expirationDate > 25.01.2001`}	n(`OR`)

⇓

Set	*VectorInfo*	*Parent*
s″	{`age`}	`null`
s‴	{`c.type = cartaDiCredito`},{`c.expirationDate > 25.01.2001`}	`null`

Tabella 4.1: Funzionamento del metodo `setAnalysis`

4.5.1 La classe Set

Di seguito sono riportate le variabili istanza contenute in un oggetto di classe `Set`.

```
private int setKey;
private static int contatoreSet = 0;
private Nodo parent;
private Vector<String> vectorInfo;
private int livello = 1;
private Vector<Set> blackList;
private Vector<Set> cover;
```

La variabile statica `contatoreSet` ha una funzione simile a quella svolta dalla variabile `contatoreNodo` negli oggetti `Nodo`: al momento della creazione di un oggetto `Set` viene assegnato il suo valore incrementato di un'unità alla variabile `nodeKey`. La variabile `livello` contiene un numero intero e viene utilizzata per assegnare ad ogni `Set` il livello di profondità rispetto alla radice: questo valore viene calcolato contando quanti nodi —partendo dal nodo radice e conteggiando il nodo radice stesso— si devono attraversare per arrivare al nodo contenente l'informazione relativa all'operatore della condizione, ossia il nodo genitore dei due nodi foglia. Delle variabili `parent` e `vectorInfo` si è già discusso nel Paragrafo 4.5. Il vettore di oggetti `Set blacklist` viene utilizzato per evitare un ulteriore confronto tra due oggetti `Set` su cui questo è già stato eseguito. Il motivo che ha reso necessario l'utilizzo di questa struttura dati è spiegato nel Paragrafo 4.5.4. Il vettore di oggetti `Set cover` viene utilizzato per verificare la relazione di copertura. Come si è già visto,

Set	*VectorInfo*	*Cover*
s^1	{age}	null
s^2	{c.type = cartaDiCredito},{c.expirationDate > 25.01.2001}	null

Set di collectionTree

Set	*VectorInfo*	*Cover*
s'^1	{age},{c.type = cartaDiCredito},{c.expirationDate > 25.01.2001}	s^1, s^2

Set di collectionView

Tabella 4.2: Esempio dell'applicazione del metodo coverVerify

le variabili livello e parent assumono particolare importanza all'interno del metodo setAnalysis: viene, infatti, effettuato il confronto solo tra Set con lo stesso valore di livello e con lo stesso riferimento contenuto in parent. Una volta che il confronto è stato effettuato, il valore della variabile livello viene decrementato di un'unità e, come si è già visto, nella variabile parent il riferimento al nodo n' viene sovrascritto dal riferimento al nodo n, genitore di n'. Sono, infine, stati implementati i metodi Getter e Setter per ciascuna delle variabili istanza, in modo da rendere reperibile qualsiasi informazione su di un particolare oggetto.

4.5.2 Confronto tra le collezioni di Set

Grazie all'invocazione dei metodi setCreation e setAnalysis sono state creati i vettori contenenti le minime collezioni di rilascio collectionTree e collectionView riguardanti rispettivamente il Policy Tree e la Vista relativa.

L'analisi delle proprietà di cui gode la Vista si basa prima di tutto sull'implementazione della relazione di copertura: poiché, secondo la definizione, è necessario verificare tale relazione per ogni set di collectionTree su ogni set di collectionView e viceversa, viene invocato due volte il metodo coverVerify, passando come argomenti collectionTree ed collectionView prima in un ordine e poi in quello invertito. La verifica della relazione di copertura tra due oggetti Set si basa su un doppio ciclo che mette a confronto ciascun dato String, contenuto nel vettore vectorInfo del primo Set s di collectionTree, con tutti quelli del vettore vectorInfo dei Set di collectionView. Ogni volta che si trova una corrispondenza, il ciclo viene interrotto e si passa alla stringa successiva. Nel caso ci sia anche solo una stringa che non sia contenuta nel vettore vectorInfo del Set s' di collectionView, il confronto si interrompe e si passa al set successivo; in caso contrario, significa che s' "copre" s ed il riferimento a quest'ultimo è quindi aggiunto al vettore cover di s', che contiene i riferimenti a tutti i set coperti. La verifica della relazione di copertura viene effettuata per tutti gli oggetti Set appartenenti ad collectionTree e successivamente per tutti quelli appartenenti ad collectionView. In Tabella 4.2 viene mostrato il risultato dell'applicazione del metodo coverVerify.

L'analisi dei vettori cover permette di verificare le proprietà di cui gode la Vi-

sta: analizzando, quindi, tali proprietà, l'utente può decidere se rilasciare la Vista al client oppure operare modifiche sulla funzione di colorazione.

4.5.3 Analisi del risultato

Il primo controllo effettuato punta alla verifica dell'esistenza di almeno un `Set` in `collectionView` tale per cui non è valida la relazione di copertura per alcun `Set` in `collectionTree`. Tale controllo è facilmente eseguibile controllando la dimensione del vettore `cover` di ogni oggetto `set` in `collectionView`: se questa è uguale a 1, significa che il vettore contiene solo il riferimento al se stesso e, quindi, non verifica la relazione di copertura per nessun `Set` di `collectionTree`. In questo caso la Vista gode della proprietà di non equità. Il motivo per cui si inserisce nel vettore `cover` di ogni `Set` il riferimento al `Set` stesso è spiegato nel Paragrafo 4.5.4.

Nel caso in cui il precedente controllo non vada a buon fine, viene verificata la proprietà di equità: ciò avviene operando dei cicli sulle due collezioni al fine di controllare che per ogni `Set` appartenente ad `collectionView` esiste almeno un `Set` in `collectionTree` tale che uno è incluso nel vettore `cover` dell'altro.

Il controllo per verificare che una Vista sia superflua viene effettuato in modo similare confrontando, sempre mediante una serie di cicli, le dimensioni dei vettori `cover` degli oggetti `Set` appartenenenti a `collectionTree` e `collectionView`: se esiste un s' che comprende più `Set` di `collectionTree` nel proprio vettore `cover`, significa che la Vista richiede una serie di informazioni superflue al fine della valutazione della politica.

Si rende, invece, necessario un approccio diverso per quanto riguarda la valutazione della proprietà di pre-analisi: innanzitutto viene valutata una variabile booleana istanziata a `true` nel caso in cui la Vista sia stata riconosciuta come equa o superflua; come si è visto, infatti, se una Vista non verifica una di queste 2 proprietà non può essere pre-analizzabile. In un secondo momento viene effettuato un controllo su `collectionTree` allo scopo di verificare l'esistenza di almeno un `Set` avente ciascuna informazione contenuta nel vettore `vectorInfo` corrispondente ad un nodo impostato a `green` nel vettore `Albero'`. Tale controllo viene eseguito utilizzando riferimenti incrociati tra le variabili di tipo `String` del vettore `vectorInfo` e le variabili `color` dei nodi corrispondenti. In Tabella 4.3 sono mostrati degli esempi di Viste che verificano le diverse proprietà in base alla collezione di set mostrata nella Tabella 4.1.

4.5.4 Problematiche relative alla creazione dei Set

Il problema più impegnativo che è stato necessario affrontare nell'implementazione del codice relativo alla creazione ed al confronto tra collezioni di Set consiste nel verificarsi di una condizione all'interno del metodo `setAnalysis`; essa ha richiesto la creazione di una struttura dati apposita all'interno degli oggetti `Set`, il vettore `blacklist`. Si consideri l'esempio mostrato in Tabella 4.4(a). Viene, prima di tutto, eseguito il confronto tra s' e s'': dato che il nodo genitore n' contiene l'operatore logico $\vee$, la variabile `livello` di s' e s'' viene decrementata di un'unità e nella variabile `parent` viene copiato il riferimento al nodo genitore di n', ovvero n. Successivamente, viene effettuato il confronto tra s, s' e s'', in quanto hanno le variabili `livello` e `parent` uguali. In questo contesto sorge il problema: visualmente, il risultato di

Set	*VectorInfo*	*Cover*
s^1	{c.type = cartaDiCredito}	null

(a)

Set	*VectorInfo*	*Cover*
s^2	{age}	s''

(b)

Set	*VectorInfo*	*Cover*
s^2	{age},{c.type = cartaDiCredito},{c.expirationDate > 25.01.2001}	s'', s'''

(c)

Set	*VectorInfo*	*Cover*
s^2	{c.type = cartaDiCredito},{c.expirationDate > 25.01.2001}	s'''

(d)

Tabella 4.3: Esempi di Viste che soddisfano le diverse proprietà: non-equa (a), equa (b), superflua (c), equa e pre-analizzabile(d)

questo confronto consiste nella creazione di due nuovi oggetti `Set`, il primo contenente i vettori `vectorInfo` di s e s' e il secondo contenente i vettori `vectorInfo` di s e s''. L'eleborazione produce, però, un risultato diverso, creando un unico oggetto `Set` contenente i vettori `vectorInfo` di s, s' e s'': ciò avviene perchè in un primo momento il programma effettua il confronto tra s e s', creando un nuovo oggetto con i loro vettori `vectorInfo`, la stessa variabile `livello` e la stessa variabile `parent`. Successivamente effettua il confronto tra il `Set` appena creato e s'', ottenendo come risultato il `Set` contenente i tre vettori. Per evitare questa doppia unione, viene, quindi, istanziato il vettore di `Set blacklist`: ad esso sono aggiunti i riferimenti a quei Set con cui è stato già fatto un confronto mediante l'operatore logico $\vee$.

Un'altra problematica da affrontare riguarda la gestione della copiatura degli elementi del vettore `vectorInfo` al momento della creazione di un nuovo oggetto `Set`. In esso possono, infatti, venire copiate due informazioni identiche oppure due informazioni che sono una "sottoinformazione" dell'altra. Il primo caso si verifica quando la stessa informazione si trova nel `vectorInfo` di entrambi i set su cui è effettuato il confronto. La soluzione adottata, in questo caso, si appoggia sull'utilizzo di un vettore `trash`, in cui una delle due informazioni viene copiata, dopo aver effettuato un controllo di uguaglianza tra di esse. Tale vettore viene, poi, ripulito al termine del confronto. Il secondo caso è una diretta conseguenza dell'assunzione fatta nel Paragrafo 4.5 circa la gestione della creazione dei set contenenti condizioni riguardanti valori costanti. Tale assunzione può causare la situazione descritta in

Set	*VectorInfo*	*Livello*	*Parent*
s	{`age`}	1	n(`AND`)
s′	{`country`}	2	n′(`OR`)
s″	{`name`}	2	n′(`OR`)

⇓

Set	*VectorInfo*	*Livello*	*Parent*
s	{`age`}	1	n(`AND`)
s′	{`country`}	1	n(`AND`)
s″	{`name`}	1	n(`AND`)

⇓

Set	*VectorInfo*	*Livello*	*Parent*
s‴	{`age`},{`country`},{`name`}	0	`null`

Set	*VectorInfo*	*Livello*	*Parent*
s‴	{`age`},{`country`}	0	`null`
s⁗	{`age`},{`name`}	0	`null`

(a)

Set	*VectorInfo*	*Livello*	*Parent*
s	{`age > 18`}	1	n(`AND`)
s′	{`age`}	1	n(`AND`)

(b)

Tabella 4.4: Problemi relativi alla creazione di Set

Figura 4.4(b), in cui, appunto, l'informazione `age` può essere considerata una "sottoinformazione" dell'informazione `age>18`. Si rende, quindi, necessaria una seconda assunzione: dato che l'informazione `age>18` è più completa, sarà questa quella copiata nel vettore `vectorInfo` del nuovo oggetto `Set`, mentre l'altra sarà scartata.

Anche nell'implementazione di questa parte di codice rimane il problema di gestire le eccezioni di tipo `NullPointerException`: se, però, nella parte riguardante la trasformazione il codice era meno complesso e tale problema è stato risolto mediante l'inserimento di alcuni controlli aggiuntivi, in questa sezione viene adottata un'altra soluzione proprio allo scopo di non appesantire ulteriormente il codice; essa consiste nell'inserire per ogni oggetto `Set` creato, un riferimento a se stesso sia nel vettore `blacklist` sia in quello `cover`, in modo da avere sempre almeno un elemento all'interno di questi vettori e rendendo, di fatto, impossibile il verificarsi dell'eccezione precedentemente citata. E' per questa ragione, ad esempio, che nella verifica della proprietà di non equità si controlla che la dimensione del vettore `cover` sia uguale

a 1 e non a 0, come invece sarebbe immediato pensare.

4.6 Esempio di esecuzione

In questo paragrafo viene mostrata un esempio di esecuzione dell'applicazione passo per passo, mettendo in risalto le operazioni compiute dai metodi che di volta in volta vengono invocati ed i loro effetti sulle variabili utilizzate. Come formula booleana di esempio per la trattazione sarà utilizzata quella mostrata nell'Esempio 8.

$$(c.type = Passport) \vee ((Age > cv.18) \wedge (Age < cv.60)) \quad \textbf{(8)}$$

Per una maggiore leggibilità, ci si riferirà agli oggetti `Nodo` mediante l'informazione contenuta nella variabile `info` e agli oggetti `Set` mediante la o le stringhe contenute nella variabile `vectorInfo`.

Creazione del vettore *Albero*

L'utente inserisce la formula booleana in un campo di testo visualizzato mediante l'interfaccia grafica; le regole sintattiche che devono essere rispettate sono mostrate lui mediante un'etichetta. La formula, memorizzata in una variabile `String`, è quindi passata come argomento al metodo `createTree`, utilizzato dal parser per verificare la validità della formula stessa. Dato che la stringa passata non è vuota, vengono creati tre oggetti `Nodo` (il nodo radice ed i suoi nodi figli) senza però settare le informazioni nelle variabili `info`. Riconoscendo l'apertura di parentesi, il parser crea altri due oggetti `Nodo`, impostandoli come figli del figlio sinistro della radice; le informazioni (`c.type`), (`=`) e (`passport`) sono quindi memorizzate nelle variabili `info` dei rispettivi oggetti. A questo punto viene riconosciuta la chiusura di parentesi e la prossima informazione, ($\vee$), viene, quindi, memorizzata nell'oggetto `Nodo` corrispondente al nodo radice. La successiva apertura di due parentesi implica la creazione immediata di quattro oggetti `Nodo`: i primi due (n' e n'') sono marcati come figli del figlio destro del nodo radice, mentre gli ultimi due sono invece settati come figli di n'. Le informazioni (`age`), (`>`) e (`18`) sono memorizzate rispettivamente nel penultimo oggetto creato, in n' e nell'ultimo oggetto creato. Il parser riconosce, quindi, la chiusura di parentesi, memorizzando linformazione ($\wedge$) nell'oggetto `Nodo` corrispondente al figlio destro del nodo radice. L'apertura di un'altra parentesi provoca la creazione di altri due oggetti `Nodo`, settati come figli di n'': le informazioni che il parser riconosce dopo la parentesi saranno, quindi, memorizzate nel penultimo oggetto creato, in n'' e nell'ultimo oggetto creato. La doppia chiusura di parentesi e il (`;`) finale consentono al parser di terminare efficacemente l'analisi lessicale e di ritornare il vettore `Albero`, a cui sono aggiunti tutti gli oggetti `Nodo` creati. Su questo vengono eseguite due routine: la prima consente di settare i collegamenti ai nodi figli, la seconda imposta il lower case per tutti i caratteri delle variabili `info` contenute nei nodi foglia.

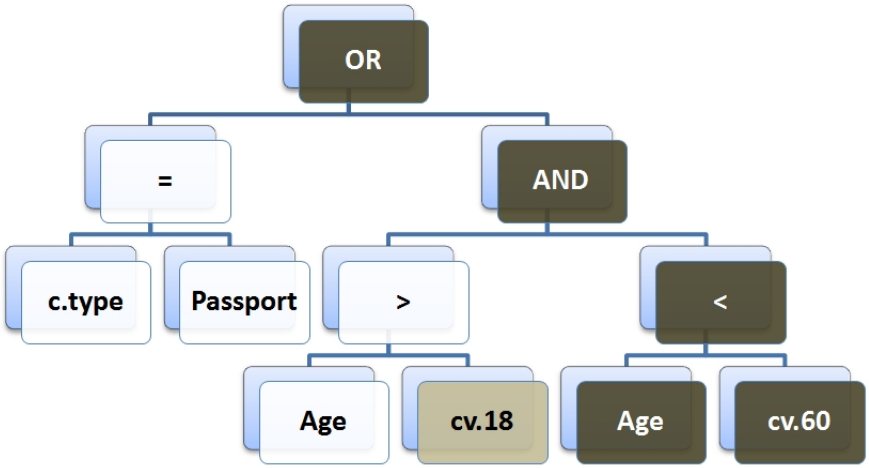
Colorazione del vettore *Albero*

A partire dal vettore `Albero`, mediante una semplice routine sono creati i vettori `LeafNodes`, che rappresenta i nodi foglia, e `NonLeafNodes`, che rappresenta i nodi in-

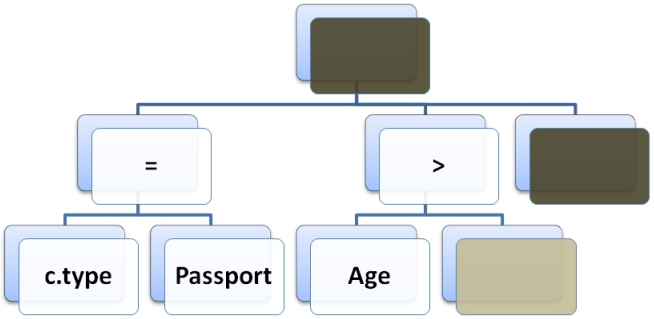
terni: il primo contiene i riferimenti ai nodi (`c.type`), (`passport`), (`age`), (`18`), (`age`) e (`60`); il secondo contiene invece i riferimenti ai nodi (∨), (=), (∧), (>) e (<). In primo luogo viene invocato il metodo `NonLeafNodesColouring` sul vettore `NonLeaf`: questo, partendo dal nodo (∨), richiede all'utente di inserire la colorazione voluta tra `red`, `yellow` e `green`. Nel caso in cui, ad esempio, l'utente inserisca la stringa `black`, viene visualizzato un messaggio di errore e viene nuovamente effettuata la richiesta. Al momento della colorazione del nodo (=) il metodo riconosce che il sottoalbero in esame contiene una condizione su un certificato e non permette, quindi, all'utente di inserire la colorazione `yellow`, visualizzando il relativo messaggio. Nel caso l'utente inserisca comunque la colorazione `yellow` viene visualizzato un ulteriore messaggio che dichiara l'impossibilità di inserire quel colore per quel nodo, effettuando poi nuovamente la richiesta. Quando, invece, passa alla colorazione dei nodi (>) e (<), il metodo riconosce che uno dei due nodi figli è un valore costante e visualizza il messaggio relativo, avvisando l'utente con le relative disposizioni. Successivamente viene invocato il metodo `LeafNodesColouring` sul vettore `LeafNodes`: tale metodo, nel momento in cui richiede all'utente la colorazione di un nodo, controlla anche il nodo fratello e valuta se sussistono particolari condizioni che determinano l'impossibilità di applicare determinate colorazioni. Analizzando la prima coppia (`c.type`) e (`passport`), il metodo riconosce che uno dei nodi contiene un certificato, quindi copia la variabile `color` del nodo padre (=) in quella dei nodi figli, senza alcuna interazione con l'utente. Quando passa alla verifica delle altre due coppie, il metodo riconosce che uno dei nodi contiene un valore costante e richiede subito all'utente la colorazione per quel nodo, specificando eventualmente l'impossibilità di dichiararlo `green` nel caso in cui il nodo genitore sia stato precedentemente dichiarato `red`. Se l'utente setta il nodo con valore costante a `green`, il metodo setta automaticamente il nodo fratello a `green` senza alcuna interazione con l'utente; in caso contrario, procede con la normale routine di colorazione per il nodo fratello. A questo punto dell'elaborazione è stata quindi definita una colorazione per ogni oggetto `Nodo` del vettore `Albero`. In Figura 4.6(a) viene mostrata la colorazione adottata come esempio.

Trasformazione del vettore *Albero*

Viene poi invocato il metodo `transform` passando come argomento il vettore `Albero`; questo metodo si applica ricorsivamente sui nodi figli: i metodi che implementano le operazioni di trasformazione sono quindi applicati in sequenza ai nodi (`c.type`), (`passport`), (=), (`age`), (`18`), (>), (`age`), (`60`), (<), (∧) e (∨). Sui primi quattro oggetti tali metodi non hanno nessun effetto, in quanto hanno tutti la variabile `color` impostata a `green`, mentre l'invocazione del metodo `hideLabel` sul nodo (`18`) ha come conseguenza il settaggio della variabile `info` ad uno spazio bianco. L'invocazione dei metodi sul nodo (>) non ha effetti, poichè questo nodo ha la variabile `color` impostata a `green` e non ha nodi figli rossi. Sui nodi foglia (`age`) e (`60`) agisce il metodo `hideLabel`, mentre sul nodo (<) l'invocazione del metodo `prune` causa l'eliminazione dei due nodi figli, (`age`) e (`34`). Sul nodo (∧) viene invocato il metodo `hideLabel`, mentre l'invocazione del metodo `collpase` sul nodo radice causa l'eliminazione del nodo (∧) e l'assorbimento dei nodi (>) e (<) (a cui è stata cancellata l'etichetta) quali figli di (∧). Il metodo `transform` restituisce quindi



(a)



(b)

Figura 4.6: Colorazione adottata(a) e Vista relativa(b)

un vettore di nodi `Albero'`, che viene memorizzato in una nuova struttura dati di tipo `Vector` e che viene mostrato all'utente mediante l'interfaccia grafica. In Figura 4.6(b) viene mostrato il vettore `Albero'`.

Creazione dei vettori *collectionTree* e *collectionView*

A questo punto si procede con la creazione delle collezioni di Set invocando il metodo `setCreation` sul vettore `Albero` e successivamente sul vettore `Albero'`. Questo metodo prende in considerazione solo gli oggetti `Nodo` genitori di nodi foglia. Nel caso del nodo (`=`), dato che uno dei figli è un certificato, crea un oggetto `Set` contenente la stringa (`c.type = passport`) nel vettore `vectorInfo`. Nel caso del nodo (`>`), visto che uno dei nodi figli contiene un valore costante, verifica la variabile `color`; dato che questa è impostata a `yellow`, crea un nuovo oggetto `Set` con la stringa (`age`) nel vettore `vectorInfo`. Stessa cosa avviene per il nodo (`<`). Al

Set	*VectorInfo*
s^1	{c.type = passport}
s^2	{age}

(a)

Set	*VectorInfo*
s'^1	{c.type = passport},{age}

(b)

Tabella 4.5: Collezioni di set risultanti: `Albero`(a) e `Albero'`(b)

termine dell'invocazione di `setCreation` sul vettore `Albero` si hanno, quindi, tre oggetti `Set`: il primo contiene la stringa (`c.type = passport`), mentre il secondo ed il terzo contengono la stringa (`age`). Quando lo stesso metodo viene invocato sul vettore `Albero'` vengono, invece, creati solo due oggetti `Set`, uno contenente la stringa (`c.type = passport`) e l'altro contenente la stringa (`age`). Gli oggetti `Set` che risultano dall'invocazione del metodo `setCreation` su `Albero` e su `Albero'` vengono aggiunti rispettivamente a `collectionTree` e `collectionView` e per ognuno di essi viene lanciata una routine che calcola la variabile `livello`. A questo punto viene invocato il metodo `setAnalysis` sulle due collezioni, con alcune differenze di elaborazione. Sul vettore `collectionTree` il metodo valuta i confronti trascendendo dal colore del nodo a cui si riferisce la variabile `parent` dei diversi oggetti `Set`: il confronto tra i due oggetti che contengono la stringa (`age`) si risolve con la creazione di un nuovo oggetto `Set`, contenente la stessa stringa, e con l'eliminazione dei due oggetti originali. Successivamente il metodo effettua un confronto tra l'oggetto appena creato e quello contenente la stringa (`c.type = passport`): dato che il riferimento nella variabile `parent` è al nodo radice $\vee$, i due oggetti vengono mantenuti separati. L'invocazione del metodo `setAnalysis` sul vettore `Albero'` non può invece trascendere dal colore dei nodi a cui si riferiscono le variabili `parent`: poichè nel confronto effettuato tra il Set contenente la stringa (`c.type = passport`) e quello contenente la stringa (`age`), la variabile `parent` si riferisce ad un nodo rosso (quindi senza etichetta), il metodo crea un nuovo oggetto `Set` contenente le stringhe dei due oggetto originali, che saranno quindi eliminati. In Tabella 4.5 sono mostrati le stringhe contenute nei Set delle due collezioni.

Verifica delle proprietà del vettore *Albero'*

Una volta create le collezioni di Set definitive, queste vengono confrontate per valutare le proprietà di cui gode il vettore `Albero'`, che rappresenta la Vista sul Policy Tree. Occorre innanzitutto verificare la relazione di copertura: a questo scopo viene invocato il metodo `coverVerify` con `collectionTree` e `collectionView` come argomenti prima in un ordine e poi nell'ordine opposto. Questo metodo compie un'analisi comparativa tra tutti i Set di `collectionTree` con tutti quelli di `collectionView`, allo scopo di verificare se ciascuna stringa contenuta nel vettore

Set	*VectorInfo*	*Cover*
s^1	{`c.type = passport`}	`null`
s^2	{`age`}	`null`

(a)

Set	*VectorInfo*	*Cover*
s'^1	{`c.type = passport`},{`age`}	s^1, s^2

(b)

Tabella 4.6: Collezioni di set con rispettive coperture: `Albero`(a) e `Albero'`(b)

`vectorInfo` di uno è presente anche nell'altro, salvando eventualmente il riferimento al `Set` "coperto" nel vettore `cover`. Il risultato dell'applicazione di questo metodo mostra che il `Set` appartenente alla `collectionView` "copre" entrambi i `Set` appartenenti alla `collectionTree`, ma non il contrario. Successivamente viene invocato il metodo `policyVerify`, che effettua i dovuti controlli sui vettori `cover` degli oggetti `Set` contenuti in `collectionView` allo scopo di determinare le proprietà della Vista: l'applicazione del metodo verifica che il vettore `cover` del Set appartenente alla `collectionView` "copre" più di un Set della `collectionTree`; la Vista è, dunque, superflua e l'utente viene avvisato con un messaggio contenente tale indicazione. In Tabella 4.6 sono mostrati i Set con i riferimenti contenuti nei corrispettivi vettori `cover`.

Capitolo 5

Conclusioni

In quest'ultimo capitolo vengono presentate alcune note riguardanti gli aspetti conclusivi del presente lavoro al fine di verificare se le esigenze a partire dalle quali è stato sviluppato il progetto sono effettivamente state coperte. In secondo luogo vengono proposti alcuni spunti per eventuali progetti futuri da integrare con il presente lavoro di tesi.

5.1 Note conclusive

Gli aspetti teorici considerati consentono di gestire efficacemente la selezione delle parti di politica per il controllo dell'accesso desiderate ed il loro successivo rilascio al client; tale gestione ha il grande vantaggio di consentire la specifica della visibilità delle condizioni contenute all'interno di una particolare politica e, all'interno di ciascuna di esse, di ogni attributo, operatore o valore costante. Il server ha quindi la possibilità di definire delle preferenze sul rilascio al client delle parti di politica.

L'implementazione delle soluzioni adottate nell'applicazione proposta consente all'utente di indicare il grado di visibilità voluto per ogni operatore ed operando e, in base a questo, automatizza il processo di trasformazione della politica. L'applicazione permette inoltre la valutazione automatica della funzione di visibilità adottata, specificando all'utente se questa consente o meno la valutazione della politica per un ipotetico client che in base ad essa rilascia le proprie informazioni. Chiaramente, l'implementazione presentata si propone di essere un semplice punto di partenza nella creazione di un software professionale per il rilascio selettivo delle politiche per il controllo dell'accesso.

5.2 Sviluppi Futuri

Data la già riscontrata necessità di separare l'approccio riguardante la specifica e la regolamentazione del controllo da applicare lato server da quello riguardante le informazioni di cui è in possesso il client (a causa delle diverse esigenze delle due parti), il presente lavoro non vuole essere considerato un modello direttamente applicabile per la specifica delle preferenze di privacy degli attributi del client: un possibile sviluppo di questo progetto potrebbe dunque essere l'applicazione di alcuni principi qui presentati (uniti ad altri aspetti teorici formulati ad hoc) per lo sviluppo

di una soluzione efficiente ed ottimizzata che affronti lo stesso problema lato client. Come si è già detto, inoltre, il modello XACML rappresenta una valida base che consente l'approccio ai diversi aspetti che riguardano il campo delle politiche di accesso: un altro possibile sviluppo di questo progetto potrebbe quindi essere la possibilità di integrare le soluzioni qui presentate all'interno delle definizioni di politiche che è possibile creare con XACML.

Bibliografia

[1] C.A. Ardagna, S. De Capitani di Vimercati, S. Foresti, G. Neven, S. Paraboschi, F.S. Preiss, P. Samarati e M. Verdicchio: *Fine-Grained Disclosure of Access Policies*. In: ICICS 2010: 12º International Conference on Information and Communications Security:. Barcellona, 2010.

[2] *eXtensible Access Control Markup Language (XACML) v.2.0*. Febbraio 2005. http://docs.oasis-open.org/xacml/2.0/access control-xacml-2.0-core-spec-os.pdf

[3] G. Marchi e M. Radenovic. *eXtensible Access Control Markup Language (XACML)*. 2007

[4] C.A. Ardagna, S. De Capitani di Vimercati, S. Paraboschi, E. Pedrini, P. Samarati e M. Verdicchio: *Expressive and deployable access control in open Web service applications*. In: IEEE Transactions on Service Computing (TSC), 2011.

[5] C.A. Ardagna, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi e P. Samarati: *Minimizing disclosure of private information in credential-based interactions: A graph-based approach*. In: SocialCom 2010 : 2º IEEE International conference on Social Computing. PASSAT 2010: 2º IEEE International conference on privacy, security, risk and trust . Minneapolis, 2010.

[6] C.A. Ardagna, S. De Capitani di Vimercati, S. Foresti, S. Paraboschi e P. Samarati: *Supporting privacy preferences in credential-based interactions*. In: CCS 2010: 7º ACM Conference on Computer and Communications Security. Chicago, 2010.

[7] P. Bonatti e P. Samarati: *Logics for authorizations and security*. In: J. Chomicki, R. Van der Meyden e G. Saake *Logics for emerging applications of databases*. Springer-Verlag, 2003.

[8] K. Irwin e T. Yu: *Preventing attribute information leakage in automated trust negotiation*. In: CCS 2005: 12º ACM Conference on Computer and Communications Security, Roma, 2005.

[9] J. Camesnisch e A. Lysyanskaya: *An efficient system for non-transferable anonymous credentials with optional anonymity revocation*. In: EUROCRYPT 2001: 20º International Conference on the Theory and Application of Cryptographic Techniques Advances in Cryptology. Innsbruck, 2001.

[10] T. Yu e M. Winslett: *A unified scheme for resource protection in automated trust negotiation* In: IEEE Symposium on Security and Privacy 2003. Berkley, 2003.

[11] P. Bonatti, D. Olmedilla: *Driving and monitoring provisional trust negotiation with metapolitics* In: POLICY 2005: 6º IEEE International Workshop on Policies for Distributed Systems and Networks. Stoccolma, 2005.

[12] K. Frikken, M. Atallah e J. Li: *Attribute-based access control with hidden policies and hidden credentials*. In: IEEE Transactions on Computers 55(10), pp. 1259-1270. 2006.

[13] Leslie Lamport: *LaTeX: a document preparation system*. Addison-Wesley, 1994.

[14] Marc Baudoin: *Apprends LaTeX*. 1998.

[15] Nadia Garbellini: *LaTeX facile*. 2008.

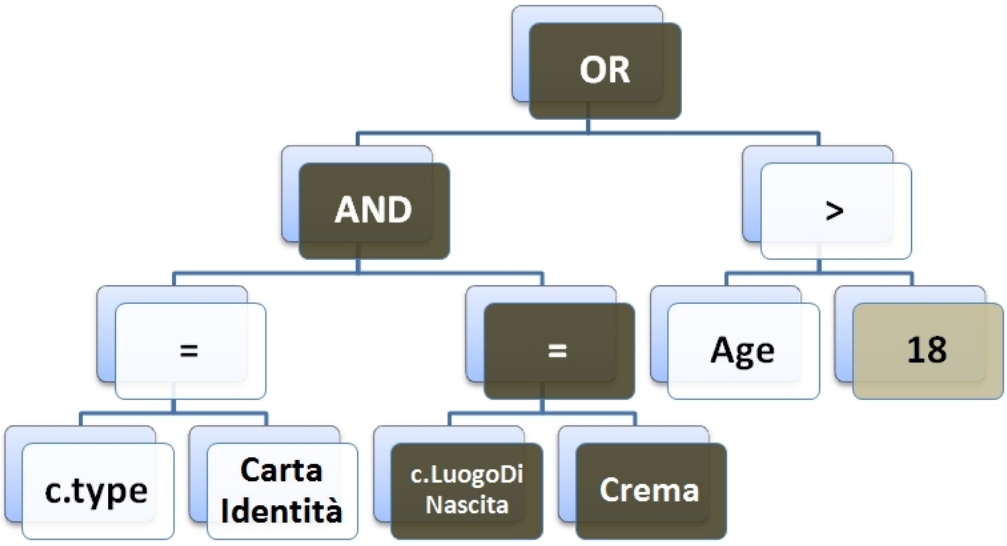
[16] *Java Compiler Compiler (JavaCC) v.0.8* http://http://java.net/projects/javacc

[17] Cay S. Horstmann e Gary Cornell: *Core Java 2*. Pearson, 2005.

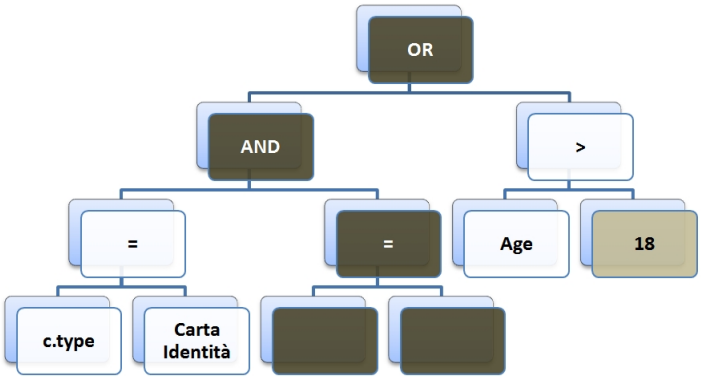
Appendice A

Applicazione visuale della trasformazione

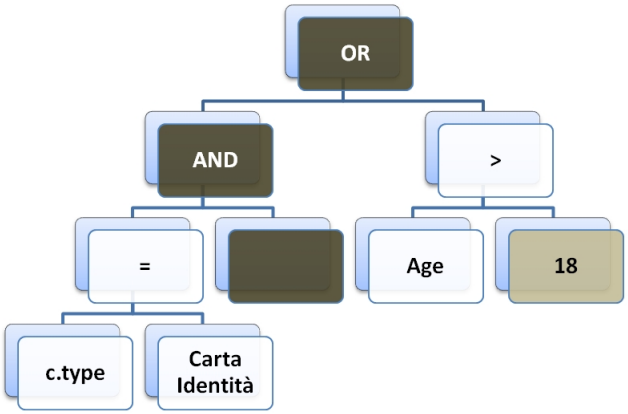
Vengono di seguito proposta visualmente l'applicazione passo per passo delle regole di trasformazione sul Policy Tree descritte nel Paragrafo 3.4.1.



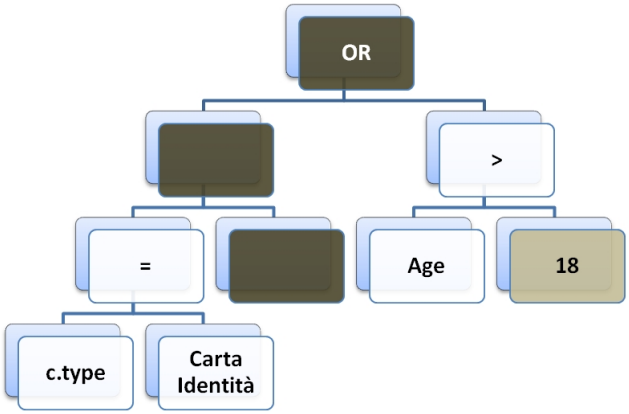
Il Policy Tree colorato originale



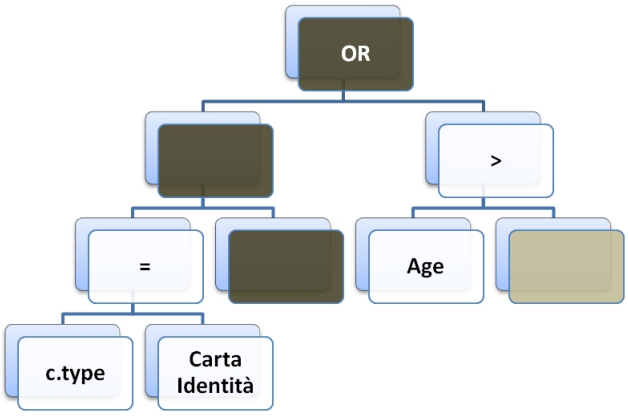
Applicazione della regola dell'offuscamento dell'etichetta sui nodi *(c.LuogoDiNascita)* e *(Crema)*.



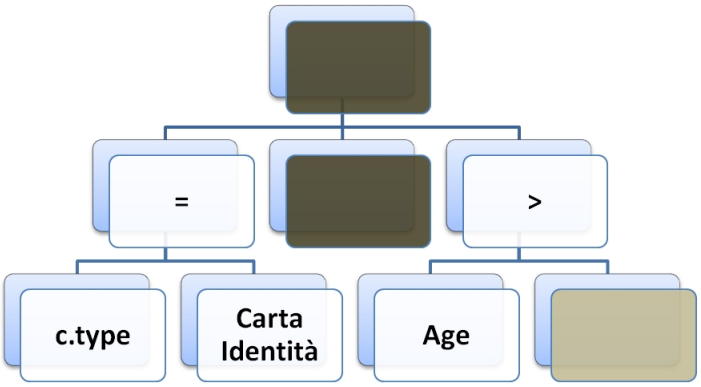
Applicazione delle regole dei figli rossi e dell'offuscamento dell'etichetta sul nodo *(=)*.



Applicazione della regola dell'offuscamento dell'etichetta sul nodo *(AND)*.



Applicazione della regola dell'offuscamento dell'etichetta sul nodo *(18)*.



Applicazione delle regole di collasso e dell'offuscamento dell'etichetta sul nodo *(OR)*.

Appendice B

Codice Sorgente

Vengono di seguito proposti gli estratti di codice Java più significativi realtivi all'implementazione creata.

B.1 Parser della formula booleana

Analisi lessicale

La formula booleana viene analizzata dal punto di vista lessicale per distinguere i diversi componenti ed i caratteri da non considerare.

```
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}
TOKEN:
{
< EQUALS: "=" >
| < MAJOR: ">" >
| < MINOR: "<" >
| < AND : "AND" >
| < OR : "OR" >
}
TOKEN :
{
  < IDENTIFIER :
    < LETTER >
    (
      < LETTER >
    | < DIGIT >
    )*
```

```
  | < DIGIT >
    (
      < LETTER >
    | < DIGIT >
    )* >
}
TOKEN:
{
  < #LETTER : [ "_", ".", "-", ":", "a"-"z", "A"-"Z" ] >
| < #DIGIT : [ "0"-"9" ] >
}
```

Creazione del vettore `Albero`

In base all'analisi lessicale, vengono ricorsivamente invocati i seguenti metodi per la creazione del vettore di oggetti `Nodo Albero`.

```
int Start(Vector<Nodo> albero) :
{}
{
  Expression(albero) ";"
  { return 0; }
| ";"
  { return 1; }
}

void Expression(Vector<Nodo> albero) :
{
  Nodo n = new Nodo();
  n.setParent(n);
  n.setColour("green");
  albero.add(n);
}
{
  Condition(tree, n)
}

void Condition(Vector<Nodo> albero, Nodo p) :
{
  Token t;
  Nodo fs = new Nodo();
  fs.setParent(p);
  fs.setColour("green");
  Nodo fd = new Nodo();
  fd.setParent(p);
  fd.setColour("green");
  albero.add(fs);
```

```
  albero.add(fd);
}
{
  UnaryExpression(albero, fs)
  (
    (
      t = < EQUALS >
    | t = < OR >
    | t = < AND >
    | t = < MAJOR >
    | t = < MINOR >
    )
    {
      p.setInfo(t.image);
    }
  UnaryExpression(albero, fd)
  )*
}

void UnaryExpression(Vector<Nodo> albero, Nodo p) :
{
  Token t;
}
{
  "("  Condition(albero, p) ")"
  | t = < IDENTIFIER >
  {
p.setInfo(t.image);
  }
}
```

B.2 Gestione della colorazione

In questo paragrafo vengono mostrati i metodi per le colorazioni dei nodi foglia: tutti ritornano un valore booleano che viene letto dal metodo principale per verificare l'applicazione della corrispettiva regola di colorazione.

Colorazione dei nodi foglia: `typeRule`

Il metodo per impostare la colorazione dei nodi contenuti nel sottoalbero riguardante una condizione su un attributo certificato.

```
public static boolean typeRule(Nodo n, Nodo m){

      if (((n.getInfo().startsWith("c")) && (n.getInfo().contains("type"))) ||
         ((m.getInfo().startsWith("c")) && (m.getInfo().contains("type"))) ) {
```

```
            m.setColour(n.getParent().getColour());
            n.setColour(n.getParent().getColour());
            return true;
      }
      else
         return false;
}
```

Colorazione dei nodi foglia: `constantRule`

Il metodo per impostare la colorazione dei nodi contenuti nel sottoalbero riguardante una condizione su un attributo certificato.

```
public static boolean constantRule(Nodo n, Nodo m){
      BufferedReader buff = new InputStreamReader(System.in));
      String colour;
      boolean insCorretto = false;
      Nodo cons, other;

      if (n.getInfo().startsWith("cv")){
         cons = n;
         other = m;
      }
      else if (m.getInfo().startsWith("cv")) {
         cons = m;
         other = n;
      }
      else {
         return false;
      }

      if (cons.getParent().getColour().equals("red")){
         System.out.println("Node: " +cons.getKey()
            +"\tValue: \"" +cons.getInfo()
            +"\"\tParent: " +cons.getParent().getInfo());
         System.out.println("This node has a red parent; as"
            +" declared in rule 1, this node must not"
            +" be green");
         System.out.println("Insert node colour (red or "
            +"yellow):");
         while (insCorretto==false){
            try {
               colour = buff.readLine().toLowerCase();
               if ( !((colour.equals("red")) ||
                  (colour.equals("yellow")) ||
                  (colour.equals("green")))){
                  System.out.println("Error: this colour does" +
```

```
                  " not exist.");
               System.out.println("Insert node colour (red or "
                  +"yellow):");
            }
            else if ( colour.equals("green")){
               System.out.println("Error: green colour is" +
                  " not permitted.");
               System.out.println("Insert node colour (red or "
                  +"yellow):");
            }
            else {
               cons.setColour(colour);
               insCorretto = true;
            }
         }
         catch (Exception e){
            System.out.println("Exception: "+e);
         }
      }

      System.out.println("Node: " +other.getKey()
               +"\tValue: \"" +other.getInfo()
               +"\"\tParent: " +other.getParent().getInfo());
      Gestione.insertColour(other);
      return true;
   }

   else {
      System.out.println("Node: " +cons.getKey()
            +"\tValue: \"" +cons.getInfo()
            +"\"\tParent: " +cons.getParent().getInfo());
      System.out.println("If you set this node as green, its"
            +" sibiling will be automatically set as green"
            +" too.");
      Gestione.insertColour(cons);

      if (cons.getColour().equals("green")){
         other.setColour("green");
      }
      else {
         System.out.println("Node: " +other.getKey()
            +"\tValue: \"" +other.getInfo()
            +"\"\tParent: " +other.getParent().getInfo());
         Gestione.insertColour(other);
      }
   }
```

```
      if (n.getInfo().startsWith("cv")){
         n = cons;
         m = other;
      }
      else {
         m = cons;
         n = other;
      }

      return true;
}
```

Colorazione dei nodi foglia: `operatorRule`

Il metodo per impostare la colorazione dei nodi contenuti nel sottoalbero riguardante una condizione con un operatore impostato a `green`.

```
public static boolean operatorRule(Nodo n, Nodo m){
      BufferedReader buff = new BufferedReade(new
            InputStreamReader(System.in));
      boolean insCorretto = false;

      if (n.getParent().getColour().equals("green")){
         System.out.println("Node: " +n.getKey()
            +"\tValue: \"" +n.getInfo()
            +"\"\tParent: " +n.getParent().getInfo());
         System.out.println("This leaf node has a green parent;"
            +"as declared in rule 2, either this node or "
            +"its sibiling\nmust not be red.");
         Gestione.insertColour(n);

         if (n.getColour().equals("red")){
            System.out.println("Node: " +m.getKey()
               +"\tValue: \"" +m.getInfo()
               +"\"\tParent: " +m.getParent().getInfo());
            System.out.println("This node must not be red;");
            System.out.println("Insert node colour (yellow or "
               +"green):");
            while (insCorretto==false){
               try {
                  String colour = buff.readLine().toLowerCase();
                  if ( !((colour.equals("red")) ||
                        (colour.equals("yellow")) ||
                        (colour.equals("green")))){
                     System.out.println("Error: this colour does "
                        +"not exist.");
                     System.out.println("Insert node colour (yellow or "
```

```
                        +"green):");
                  }
                  else if ( colour.equals("red")){
                     System.out.println("Error: red colour is "
                        +"not permitted.");
                     System.out.println("Insert node colour (yellow or "
                        +"green):");
                  }
                  else {
                     m.setColour(colour);
                     insCorretto = true;
                  }
               }
               catch (Exception e){
                  System.out.println("Exception: "+e);
               }
            }
         }
         else {
            System.out.println("Node: " +m.getKey()
               +"\tValue: \"" +m.getInfo()
               +"\"\tParent: " +m.getParent().getInfo());
            Gestione.insertColour(m);
         }
         return true;
      }
      else
         return false;
}
```

B.3 Trasformazione del vettore `Albero`

Il metodo `transform`

L'applicazione ricorsiva del metodo `transform`, la creazione dei vettori `redSons` e la scelta del metodo di trasformazione da applicare in base alla posizione del nodo.

```
public static void transform(Vector<Nodo> albero, Nodo n){
      Vector<Nodo> redSons = new Vector<Nodo>();
      int i;
      boolean control = true;

      if (!n.getSons().isEmpty()){
         for (i=0; i<n.getSons().size(); i++){
            transform(albero, n.getSons().get(i));
         }
      }
```

```
      if (!n.getSons().isEmpty()) {
         for (i=0; i<n.getSons().size(); i++){
            if (n.getSons().get(i).getColour().equals("red")){
               redSons.add(n.getSons().get(i));
            }
            if (!n.getSons().get(i).getSons().isEmpty()){
               control = false;
            }
         }
         if (control){
            albero = Trasformazione.prune(redSons, albero, n);
         }
         if (!control){
            albero = Trasformazione.collapse(redSons, albero, n);
         }
         control = true;
      }

      Trasformazione.hideLabel(n);

}
```

Il metodo prune

```
public static Vector<Nodo> prune(Vector<Nodo> redS,
            Vector<Nodo> albero, Nodo n){
      int i, j;

      if (n.getColour().equals("red")) {
         for (i=0; i<n.getSons().size(); i++){
            for (j=0; j<redS.size(); j++){
               if (n.getSons().get(i).equals(redS.get(j))){
                  albero.remove(n.getSons().get(i));
                  n.getSons().remove(i);
                  n.setSons(n.getSons());
               }
            }
         }
      }

      if ((redS.size())==2){
         albero.removeAll(n.getSons());
         n.getSons().clear();
         n.setSons(n.getSons());
         n.setColour("red");
      }
```

```
      return albero;
}
```

Il metodo `collapse`

```
public static Vector<Nodo> collapse(Vector<Nodo> redS,
            Vector<Nodo> albero, Nodo n){
      int i, j;

      if (n.getColour().equals("red")){
         for (i=0; i<n.getSons().size(); i++){
            if (!n.getSons().get(i).getSons().isEmpty()){
               for (j=0; j<redS.size(); j++){
                  if (n.getSons().get(i).equals(redS.get(j))){
                     n.getSons().addAll(redS.get(j).getSons());
                     albero.remove(n.getSons().get(i));
                     n.getSons().remove(i);
                     n.setSons(n.getSons());
                  }
               }
            }
         }
         for (j=0; j<n.getSons().size(); j++){
            n.getSons().get(j).setParent(n);
         }
      }

      return albero;
}
```

Il metodo `hideLabel`

```
public static void hideLabel(Nodo n){
      if ((n.getColour().equals("red")) ||
         (n.getColour().equals("yellow"))) {
         n.setInfo(" ");
      }
}
```