LINGUAGGIO DI DEFINIZIONE DI REGOLE ATTE AL CONTROLLO DI STATO DI UNA RETE

ROBERTO BELLI

Contents

1. Introduzione

Il progetto prevede la definizione di un linguaggio descrittivo di regole che possano indicare situazioni eccezionali sullo stato del traffico di una rete . Tutto ciò con l'obiettivo di essere direttamente o indirettamente interpretato da un analizzatore di rete qualsiasi periodicamente , ed essere utilizzato da questo per la rilevazione di stati particolari della rete con la conseguente segnalazione di allarmi.

2. Struttura del linguaggio

Nella progettazione del linguaggio é risultata da subito evidente la necessitá di definire gruppi di host a discrezione del definitore delle regole. Proprio per questo è stato introdotto il concetto di *Community* . Una volta definito il concetto di community si è passato all'analisi delle necessità che ci stanno nel descrivere lo stato di una rete. Da questa analisi si è arrivati alla definizione di *Regola* (Rule[1]) come descrizione del rapporto tra due entità diverse e un operatore di comparazione (operatore relazionale , ad esempio maggiore minore uguale ecc..). Questo è risultato un buon punto di partenza per capire che regole complesse con questo metodo non sarebbe stato possibile definirle. Proprio

[1]utilizzeremo in questa trattazione entrambi i termini in modo intercambiabile

per questo è stato pensato allora un nuovo tipo di regola che riuscisse a combinare regole definite come sopra ed a descriverne quindi una complessa. La regola complessa è quindi rappresentabile da un albero che ha come foglie regole semplici e come nodi operatori logici se i figli sono a loro volta regole, relazionali se invece i figli sono riferimenti a community. Una volta stabilito ciò è venuta fuori la necessità di distinguere due diversi casi nella valtazione delle regole: il caso in cui la regola riguardasse valori aggregati o comunque proprietà della comunity stessa oppure il caso in cui questa si riferisse ai singoli host presenti nella comunity. Si è quindi pensato di distinguere il caso del foreach dal caso dell'applicazione di una funzione aggregativa tra gli host. Questa differenza non era da subito venuta fuori dall'analisi delle problematiche del caso in quanto a livello teorico era possibile la definizione di community di un solo host quindi si potevano applicare funzioni aggregative su svariati insiemi di un solo host. Ovviamente all'atto pratico rimane di difficile scrittura e non pratico utilizzo un linguaggio che non preveda questa facility.

2.1. **Community.**

La Community è stata pensata esclusivamente come un insieme di host, e per questo fine si è deciso di descriverla con una lista di indirizzi IP o di CIDR[2]

2.2. **Regola.**

Riassumendo i concetti esplicitati nella introduzione della sezione, la regola può essere quindi di vari tipi:

function(Community, measure) OPERATORE function(Community, measure):
questo tipo di regola esprime il confronto tra due valori ognuno riferito ad una community ed identificato da *Measure*[3].

function(Community, measure) OPERATORE Constant:
in questo caso il confronto del primo argomento si fa con una costante.

foreach(Community, measure) OPERATORE function(Community,measure):
questo tipo di regola esprime il confronto tra la misura (measure) riferita ad ogni singolo host della community confrontandola con il valore risultante dalla funzione applicata al secondo argomento dell'operatore.

foreach(Community, measure) OPERATORE Constant:
in questo caso il confronto della measure relativa ad ogni host si fa con una costante.

Rule BOOLOP Rule:
Regola che compone due regole, è vera a seconda della valutazione delle sottoregole ed alla applicazione di queste di un operatore booleano.

---

[2]notazione che indica un indirizzo di rete dal quale si può evincere un insieme di indirizzi IP appartenenti a quella rete

[3]Misura, rappresenta una tupla di attributi di riferimento per la regola ad esempio < "byte trasferiti" , intervallo porte>

2.2.1. *Grammatica parziale.* Partendo dalle strutture base di regole (precedentemente citate) si è passati alla descrizione della grammatica delle regole stesse:

RULE: = FUNCTION EXPROP FUNCTION |
FUNCTION EXPROP CONSTANT |
FOREACH EXPROP FUNCTION |
FOREACH EXPROP CONSTANT |
RULE BOOLOP RULE ;

EXPROP: = < | > | ≤ | ≥ | == | != ;

BOOLOP: = AND | OR ;

2.2.2. *Timing.*
Siccome un elevato numero di allarmi sarebbe inutile in quanto chi poi li controlla li ignorerebbe si è deciso di introdurre anche il concetto di timing nelle regole. La regola, al momento della valutazione non tiene conto solo lo stato attuale della rete ma prende in considerazione (al fine di lanciare Allarmi) anche le valutazioni precedenti. In pratica, considerando un sistema che fa una valutazione ogni X millisecondi si può imporre tramite il timing che dopo Z valutazioni delle regole (tutte positive) si provveda ad inviare un alert. Con questo escamotage si può quindi arrivare ad esprimere regole che presuppongano il perdurare di una situazione anomala per Z * X millisecondi.

3. Linguaggio di Rappresentazione delle Regole e delle Community

3.1. **Linguaggio ex novo VS tecnologie preesistenti.**

Una volta stabilita la grammatica per la descrizione delle regole è stato necessario scegliere un adeguato metodo per rappresentarle e per rappresentare le community. La prima scelta valutata è stata quella di creare un linguaggio nuovo che rappresentasse per ogni entità sopra descritta gli attributi necessari e opzionali. Il metodo di rappresenzazione delle regole complesse è stato il più attentamente studiato. Le regole complesse strutturate come si era previsto potevano essere analizzate e valutate utilizzando le peculiarità della programmazione ricorsiva che può massimizzare i risultati su strutture dati come alberi. Le regole rappresentate ad albero quindi se la rappresentazione lo poteva permettere potevano essere valutate con agilità.

Una volta creato questo linguaggio però se ne sono notati subito i limiti, nella sua forma benché semplificata costruire un parser[4] avrebbe richiesto duri e poco utili sforzi di programmazione (per oco utili ci si riferisce alla ridotta portabilità del lavoro: qualunque piattaforma si scelga per implementare il parser e realizzare l'interprete risulterebbe oneroso realizzare il porting su altre tecnologie soprattutto se ciò viene creato tramite un linguaggio di scripting). L'alternativa piú logica quindi è risultata essere quindi l'utilizzo di tecnologie gia esistenti e fortemente diffuse: L'unione di questa esigenza con l'esigenza di rappresentare le regole ad albero ha portato all'immediata valutazione di

[4]Programma che verifica la consistenza di una rappresentazione e ne interpreta le informazioni contenute

XML[5] come alternativa. Questa è sembrata da subito la soluzione più semplice al fine di rendere semplice il lavoro di valutazione , validazione e generazione di regole oltre che a rendere il tutto molto efficiente visti gli innumerevoli team di programmatori che lavorano sull'estesissimo campo di parser e validatori di XML.

3.2. **Prototipo XML.**

Una volta individuato come linguaggio di rappresentazione di queste regole un sottoinsieme di XML , ci si è rivolti all'aspetto pratico della cosa prima realizzando un prototipo di file di configurazione. Visto che l'XML permette di rappresentare alberi si è deciso di rappresentare la totalità delle informazioni necessarie nella radice <*configuration*>. qui sono ammessi solo due rami: <*communities*> *e* <*rules*> atti a contenere i relativi insiemi di descrizioni di communities e rules. Ognuno di questi due rami conterrà uno o piú elementi.

3.2.1. *Elementi contenuti in Communities.*

La sezione che descrive le community è molto semplice, una lista di CIDR e una di host in questo ordine, le liste possono anche essere vuote ma ci deve essere almeno una entry tra host e indirizzi di rete.
Per Esempio:

```
<communities>
        <community id="community01" >
                        <cidr>192.168.0.0/24</cidr>
                        <host>192.168.1.1</host>
        </community>
        <community id="community02" >
                        <cidr>192.168.1.0/24</cidr>
                        <host>10.0.0.1</host>
        </community>

</communities>
```

3.2.2. *Elementi contenuti in Rules.*

La descrizione di una regola in XML ha richiesto un grosso lavoro per l'individuazione di tutti gli attributi necessari per definire il piú ampio numero di regole possibile e quindi di lasciare all'utilizzatore finale piú spazio per descrivere le proprie esigenze a prescindere dalla loro complessità. Una regola base ha come attori una combinazione di funzioni foreach e costanti, le prime due hanno un intrinseco bisogno di un riferimento alla comunity al quale vengono applicate: questo è stato realizzato tramite la specifica esplicita di questo riferimento.

Ogni attore relativo a community (<*foreach*> o <*function*>) deve avere uno specifico riferimento alla misura che si intende controllare e questo è specificato all'interno dell'operando stesso tramite i tag <*measure*> che devono contenere come attributi il tipo

[5] eXtendible Markup Language

di misurazione che ci interessa e in caso un opzionale PortRange che ci interessa osservare nelle misurazioni. I possibili tipi di misurazioni specificabili nel tag <*measure*> sono al momento *GENERATED TRAFFIC, RECEIVED TRAFFIC, TOTAL TRAFFIC , OPENED CONNECTION*.

Sia nel caso di aggregazione di valori che nel caso del <*foreach*> è possibile definire opzionalmente un offset percentuale in modo da rendere ancora più personalizzabile l'insieme di regole. Questo offset ha la funzione di incrementare o decrementare del valore indicato il valore che sarà poi confrontato con l'altro operando. Solo nel caso della <*function*> va specificato nell'attributo type il tipo della funzione di aggregazione: per ora quelle previste sono *SUM, MAX, MIN, AVG*.

L'operatore in una regola può essere esclusivamente di due tipi : di confronto o booleano che vengono rappresentati rispettivamente da <*exprOp*> che ha un attributo (type) che indica l'operatore e <*boolOp*> che funziona in modo analogo.

L'operatore definito da <*exprOp*> può quindi appartenere all'insieme *GREATER THAN, LESS THAN, GREATHER THAN OR EQUAL TO, LESS THAN OR EQUAL TO, EQUAL TO, NOT EQUAL TO* mentre l'operatore definito dai tag <*boolOp*> può essere esclusivamente *AND* oppure *NOT*.

Per Esempio:

```
<rules>
# regola con doppia funzione aggregativa
<rule id="rule1" timing="" >
        <function communityId="community01" offsetPercentage="
            100" type="SUM">
        <measure type="Generated Traffic">
                <portRange begin="80" end="80"/>
        </measure>
        </function>
        <exprOp type="GREATHER THAN"/>
        <function communityId="community02" offsetPercentage="
            100" type="MAX" >
        <measure type="Generated Traffic"/>
        </function>
</rule>
#regola con confronto tra foreach e costante
<rule id="rule2" timing="" >
        <foreach communityId="community01" offsetPercentage="
            100" >
        <measure type="Generated Traffic">
                <portRange begin="80" end="80"/>
        </measure>
        </foreach>
        <exprOp type="LESS THAN"/>
        <constant>
                15000
```

```
        </constant>
</rule>
#regole annidate
<rule id="rule3"  timing="" >
        <rule id="rule4" timing="" >
                <function type="SUM" communityId="community01"
                    offsetPercentage="100">
                <measure type="Generated Traffic">
                        <portRange begin="80" end="80"/>
                </measure>
                </function>
                <exprOp type="GREATHER THAN"/>
                <function type="SUM" communityId="community02"
                    offsetPercentage="100" >
                <measure type="Generated Traffic">
                        <portRange begin="80" end="80"/>
                </measure>
                </function>
        </rule>
        <boolOp type="AND"/>
        <rule id="rule5" timing="" >
                <function type="SUM" communityId="community01"
                    offsetPercentage="100" >
                <measure type="Generated Traffic">
                        <portRange begin="80" end="80"/>
                </measure>
                </function>
                <exprOp type="GREATER THAN OR EQUAL TO"/>
                <constant>
                        15000
                </constant>
        </rule>

</rule>


</rules>
```

3.3. **Grammatica e Validazione.**

Realizzato il prototipo il passo successivo è stato pensare a come validarlo . Da queste necessità ed un po' di ricerca si sono scoperte le principali tecnologie per la descrizione di *schemi XML*[6] tra cui compaiono ad esempio "XML SCHEMA" , "DTD"

[6]il suo scopo è delineare quali elementi sono permessi, quali tipi di dati sono ad essi associati e quale relazione gerarchica hanno fra loro gli elementi contenuti in un file XML.

e " RELAXNG". Quest'ultimo è sembrato il piú idoneo in quanto molto semplice da utilizzare, molto diffuso (gran numero di librerie per ogni tipo di piattaforma) e in grado di rappresentare agilmente anche grammatiche ricorsive in pochi passi. Tramite la notazione di RelaxNG quindi è stata riscritta la grammatica completa precedentemente specificata esplicitando ogni possibile attributo.

Collegati a questa tecnologia ovviamente esistono un gran numero di validatori che verificano la corretta formazione di un file xml contenente queste regole. Purtroppo non è stato possibile controllare la completa consistenza della descrizione XML interamente tramite il validatore in quanto o si sarebbe resa inutilmente complicata la notazione da usare nella descrizione o dei controlli sono risultati proprio impossibili da fare con questo metodo.

Per completezza ne riportiamo la descrizione controllo per controllo:

- Validazione degli indirizzi di host
- Validazione degli indirizzi CIDR dei network
- Validazione dei port range
- Controllo univocità di ID delle community
- Controllo univocità ID delle regole
- Controllo di corretto riferimento di attori delle regole alle rispettive community

3.3.1. *RelaxNG Schema.*
Di seguito è riportato lo schema finale che descrive la grammatica del linguaggio specificato.É stato descritto usando la tipica notazione della grammatica che permette di inserire dei riferimenti ad altre strutture. il validatore la utilizzerà partendo dal tag *<start>* e sostituendo ai riferimenti delle sotto strutture le strutture stesse.

```
1  <grammar xmlns="http://relaxng.org/ns/structure/1.0">

3  <start>
   <element name="configuration">
5  <ref name="Communities"/>
   <ref name="Rules"/>
7  </element>
   </start>
9

11 <define name="Communities">
   <element name="communities">
13   <zeroOrMore>
           <ref name="Community"/>
15   </zeroOrMore>
   </element>
17 </define>

19 <define name ="Community">
   <element name="community">
21           <attribute name="id" />
```

```
            <optional>
23        <attribute name="name">
            <text/>
25        </attribute>
          </optional>
27          <zeroOrMore>
                    <element name="cidr">
29                          <text/>
                    </element>
31          </zeroOrMore>
            <zeroOrMore>
33                  <element name="host">
                            <text/>
35                  </element>
            </zeroOrMore>
37  </element>
    </define>
39

41  #define the composition of a set of rule
    <define name="Rules">
43  <element name="rules">
            <oneOrMore>
45                  <ref name="Rule"/>
            </oneOrMore>
47  </element>
    </define>
49
    # define the possible composition and attributes of one rule
51  <define name="Rule">
    <element name="rule">
53  <attribute name="id" />
    <optional>
55  <attribute name="timing" />
    </optional>
57          <choice>
                    <ref name="TRule1"/>
59                  <ref name="TRule2"/>
                    <ref name="TRule3"/>
61          </choice>
    </element>
63  </define>

65  # basic rule type.
    <define name="TRule1">
```

```
67          <group>
                    <ref name="Function"/>
69                  <ref name="ExprOp"/>
                    <choice>
71                          <ref name="Function"/>
                            <ref name="Constant"/>
73                  </choice>
            </group>
75  </define>

77  # rule that merge the result from 2 rules
    <define name="TRule2">
79          <group>
                    <ref name="Rule"/>
81                  <ref name="BoolOp"/>
                    <ref name="Rule"/>
83          </group>
    </define>
85
    # Rule that permit to value a rule foreach host in the
       specified community
87  <define name="TRule3">
            <group>
89                  <ref name="Foreach"/>
                    <ref name="ExprOp"/>
91                  <choice>
                    <ref name="Function"/>
93                  <ref name="Constant"/>
                    </choice>
95          </group>
    </define>
97
    <define name="Foreach">
99          <element name="foreach">
                    <attribute name="communityId" />
101         <optional>
                    <attribute name="offsetPercentage">
103                 <data type="positiveInteger" datatypeLibrary="
                       http://www.w3.org/2001/XMLSchema-datatypes"
                       />
                    </attribute>
105         </optional>
            <ref name="Measure"/>
107         </element>
    </define>
```

```
109
<define name="Function">
111 <element name="function">
        <attribute name="communityId" />
113     <attribute name="type">
            <choice>
115         <value>Max</value>
            <value>Min</value>
117         <value>Avg</value>
            <value>Sum</value>
119         </choice>
        </attribute>
121     <optional>
            <attribute name="offsetPercentage">
123         <data type="positiveInteger" datatypeLibrary="
                http://www.w3.org/2001/XMLSchema-datatypes"
                />
            </attribute>
125     </optional>
        <ref name="Measure"/>
127 </element>
</define>
129
<define name="Measure">
131 <element name="measure">
            <attribute name="type">
133         <choice>
            <value> GENERATED TRAFFIC </value>
135         <value> RECEIVED TRAFFIC </value>
            <value> TOTAL TRAFFIC </value>
137         <value> OPENED CONNECTION </value>
            </choice>
139     </attribute>
        <optional>
141         <element name="portRange">
            <attribute name="begin">
143         <data type="positiveInteger" datatypeLibrary="
                http://www.w3.org/2001/XMLSchema-datatypes">
            <param name="minInclusive">1</param>
145         <param name="maxInclusive">65535</param>
            </data>
147         </attribute>
            <attribute name="end">
149         <data type="positiveInteger" datatypeLibrary="
                http://www.w3.org/2001/XMLSchema-datatypes">
```

```
                    <param name="minInclusive">1</param>
151                 <param name="maxInclusive">65535</param>
                    </data>
153                         </attribute>
                    </element>
155         </optional>
    </element>
157 </define>

159 <define name="ExprOp">
    <element name="exprOp">
161 <attribute name="type">
    <choice>
163         <value>GREATER THAN</value>
            <value>LESS THAN</value>
165         <value>GREATHER THAN OR EQUAL TO</value>
            <value>LESS THAN OR EQUAL TO</value>
167         <value>EQUAL TO</value>
            <value>NOT EQUAL TO</value>
169 </choice>
    </attribute>
171 </element>
    </define>
173
    <define name="BoolOp">
175 <element name="boolOp">
    <attribute name="type">
177 <choice>
            <value>AND</value>
179         <value>OR</value>
    </choice>
181 </attribute>
    </element>
183 </define>

185 <define name="Constant">
    <element name="constant">
187         <data type="positiveInteger" datatypeLibrary="http://
               www.w3.org/2001/XMLSchema-datatypes"/>
    </element>
189 </define>

191 </grammar>
```

4. Algoritmo di analisi dello stato della rete

In questa sezione si descriverà un possibile algoritmo che utilizzi il linguaggio descritto nelle sezioni precedenti. Questa fase è stata ritenuta necessaria per la verifica dell'usabilità del progetto considerando come logica prosecuzione di questo la creazione di un tool che interpreti e valuti le regole. Concettualmente l'analisi dello stato di una rete l'abbiamo interpretato come un evento che avviene ad intervalli di tempo regolari e che non viene fatto direttamente dall'interprete del linguaggio in cui vengono scritte le regole ma quest'ultimo si occupa di interrogare un simil-database che contiene tutte le informazioni di cui necessitiamo.

Possiamo dividere il processo di analisi in piu fasi:

- verifica e validazione del file contenente le regole
- recupero dello stato delle valutazioni di regole eseguite precedentemente
- creazione strutture dati necessarie alla fase di valutazione delle regole
- valutazione delle regole con conseguente lancio allarmi
- salvataggio dello stato attuale

Per non valutare astrattamente questa fase pratica si è preferito dare un contesto ad una possibile implementazione di questo tool. Si è deciso di utilizzare una nuova feature del noto tool di monitoraggio di rete open source *NTOP*[7]. Questa nuova feature permette in pratica di far interpretare script Python[8] direttamente da NTOP e tramite alcune interfacce è possibile, da parte di uno script così interpretato, l'interrogazione di ntop per ricevere dati sullo stato della rete. Oltre a questo è possibile far eseguire questo script da NTOP ad intervalli di tempo regolari.

4.1. **Struttura Script.**

Oltre al file (o più di uno) che contiene il codice dello script, supporremo di avere altri file nel nostro progetto:

config.xml: il file contenente la definizione delle community e delle regole da valutare ad ogni esecuzione

schemaRelaxNG.xml: schema XML che descrive la corretta formazione del file config.xml tramite la notazione RelaxNG

past.save: file contenente la serializzazione delle strutture dati necessarie allo script per valutare lo stato delle valutazioni passate

4.2. **Librerie utilizzabili.**

Per verificare la reale presenza di librerie che giustifichino l'uso di XML , nell'ambito di python la ricerca è stata veloce ed ha portato a diverse possibili soluzioni. Si sono trovate librerie che parsano , interpretano e validano XML tutte utilizzando anche RelaxNg. Alcuni esempi sono *pyxml, xml.dom , sax , lxml* le distinguono solamente fattori

[7] `http://www.ntop.org`
[8] `http://www.python.org`

prestazionali e il modo di interpretare l'xml (ad esempio sax interpreta xml come uno stream di dati ed è efficientissimo in memoria solo che non costruisce l'albero, il dom invece è più pesante: fa un albero complesso e per ogni nodo crea figli sia per i nodi di livello inferiore che per gli attributi.) Nel continuare a dare una consistenza alla strutturazione di una possibile applicazione in python si è scelto di utilizzare lxml come libreria in quanto crea l'albero ma in modo semplificato (attributi e altre cose vengono memorizzati nel nodo stesso) quindi riduce il numero di visite a nodi e soprattutto sembra essere un progetto in fase avanzata tra le cui peculiarità fa dell'efficienza e la semplicità la propria bandiera .

4.3. **Algoritmo nel dettaglio.**

l'esecuzione dello script (come è stato pensato) ha sicuramente delle fasi fondamentali che verranno di seguito illustrate:

Recupero stato precedente: come prima fase vanno recuperati i dati delle elaborazioni precedenti e messi nelle strutture dati opportune. Se non sono presenti queste strutture dati serializzate vanno inizializzate per permette il corretto salvataggio alla fine dello script.

Hash del file di configurazione: va calcolato l'hash del file di configurazione che si andrà a valutare. Questo è necessario per rilevare (confrontandolo con l'hash salvato nella precedente elaborazione) eventuali modifiche del file di configurazione che renderebbero inconsistente lo stato appena recuperato da file. In questo caso va re-inizializzato lo stato recuperato.

Parse di schema e file di configurazione: va recuperato da file sia lo schema che il file di configurazione in modo da permettere alle librerie lxml il parsing e la creazione degli alberi che li rappresentanto in memoria.

Validazione: va validato il file di configurazione tramite lo schema in un primo momento e tramite i controlli non effettuabili tramite lo schema (Sezione 3.3) in un secondo momento.

Suddivisione albero: l'albero ricavato dal file di configurazione va diviso in due alberi, uno contenente esclusivamente le community e uno contentente la root delle regole.

Elaborazione Community: risulta necessario elaborare le community, infatti in questo ambito ntop andrà interrogato host per host quindi andranno espansi i cidr. La soluzione che pare più immediata è la creazione di un array di tuple del tipo <id Comunity, IpRangeList > . IpRangeList è una classe fornita dalla libreria iptools[9] che con le opportune modifiche espande cidr e fornisce iteratori su insiemi di indirizzi ip.

Elaborazione Regole: è necessario in questo caso esplicitare l'algoritmo utilizzando le notazioni tipiche del caso come i cicli:

[9] `http://code.google.com/p/python-iptools/`

```
ciclo
per ogni regola nella root <rules> {
        chiamo un metodo valuta(regola) che la valuta
           ricorsivamente e restituisce o vero o falso
        se regola.timing è diverso da 0
                leggo stato da struttura dati globale
                se è il caso mando allarme
                salvo l'esito in una struttura dati
                   globale
        altrimenti
                mando l'allarme se valuta(regola) è vero
                altrimenti continuo
}
```

Salvataggio stato: Vanno serializzate le strutture dati utilizzate per la valutazione delle regole.

4.4. **Strutture dati.**
una semplice struttura dati che considera l'ordine di valutazione delle regole e che contiene una rappresentazione base degli esiti potrebbe limitarsi a contenere l'hash del file di configurazione precedentemente valutato e un'array di tuple < ID regola , numero di volte che di seguito è stata valutata verificata >. Con un occhio all'efficienza invece si potrebbero serializzare anche gli alberi delle regole e la struttura dati che rappresenta l'insieme delle community interessate senza rieffettuare il parsing e la valutazione in quanto sono sicuramente pesanti a livello computativo. Questo ovviamente solo nel caso che non sia cambiato il file di configurazione.

5. Conclusioni

Il linguaggio in questione purtroppo non è di agile scrittura, infatti per mantenete la notazione pulita e per permettere l'espressione di regole un po' più complesse è stato necessario complicarlo. L'uso di xml però risulta una semplificazione in fase di realizzazione in quanto permette l'agile utilizzo di librerie ormai collaudate. Per rimediare a questa carenza (quella della complessità di notazione) non sarebbe però impossibile come estensione del progetto creare un tool che guidi nella creazione di file di configurazione rendendo all'utente finale invisibile la fase di scrittura in xml del tutto.