UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

SVILUPPO DI STRUMENTI PER SISTEMI REATTIVI BASATI SU REGOLE DI PRODUZIONE

Relatore

Chiar.mo Prof. Ing. A. POGGI

Correlatori

Dott. Ing. M. TOMAIUOLO

Tesi di Laurea di

FABIO GHILARDELLI

Anno Accademico 2007/2008

Ad Angela e Franco

Ringraziamenti

La realizzazione di questa tesi necessita di numerosi ringraziamenti.
Innanzitutto i più grandi ringraziamenti, sia per il supporto tecnico sia per quello morale, vanno al correlatore Dott. Ing. Michele Tomaiuolo: per la pazienza dimostratami nello sviluppo del progetto e nella risoluzione dei problemi.
Devo ringraziare il Prof. Agostino Poggi per la possibilità che mi ha dato nello svolgere questa tesi e per i suggerimenti che mi ha fornito.
Da non dimenticare i colleghi del laboratorio AOT, dimostratisi disponibili per chiarimenti su problemi simili a quelli affrontati in questa tesi.
Ovviamente devo ringraziare la mia famiglia e gli amici che mi hanno spronato in questo sviluppo: in special luogo i miei genitori che mi hanno permesso di frequentare questa università.
Tra gli amici come non citare gli iscritti all'anno accademico 2005/2006 della Facoltà di Ingegneria Informatica per la collaborazione nell'affrontare la difficile carriera universitaria.

Indice

Indice delle Figure

Indice dei Listati

Prefazione

Questa tesi tratta dello sviluppo di strumenti software per l'operatività in ambienti dinamici, in particolare di supporto di sistemi basati su regole di produzione che siano capaci di reagire a vari eventi. Un sistema a regole è un processo software che, ricevendo in ingresso conoscenza anche parziale rispetto ad un problema, è in grado di generare delle soluzioni al problema stesso per mezzo di un ben specifico processo di Deduzione.

La Conoscenza è, di solito, rappresentata con un insieme di fatti e oggetti mentre l'Inferenza è il processo di deduzione mediante il quale da una proposizione accolta come vera, si passa a una proposizione la cui verità è considerata contenuta nella prima. Inferire è quindi trarre una conclusione.

L'operatività in un ambiente dinamico si pone come problema nell'ambito informatico perché, in futuro, sarà fondamentale gestire la cooperazione di una moltitudine di applicazioni che comunicano e modificano in maniera concorrente l'ambiente, per raggiungere la soluzione di un unico problema. Il maggiore sviluppo è richiesto principalmente dai nuovi sistemi che utilizzano il Web, non solo come substrato per reperire informazioni residenti in remoto, ma come piattaforma che consente l'interazione attiva verso altri sistemi presenti in rete. In questa ottica i sistemi a regole risultano il miglior strumento di sviluppo per affrontare problematiche di gestione di mutamenti dello stato di qualche nodo della rete, in quanto si può implementare la logica del sistema senza conoscere la reale gestione dei dati.

La necessità di utilizzare un sistema a regole di produzione per sviluppare un sistema a regole reattive nasce dal fatto che i sistemi a regole reattive non trovano, a tutt'oggi, sbocchi applicativi commerciali poiché molti progetti sono ancora in

fase di sviluppo in ambito universitario. Trovano sviluppi concreti i sistemi a regole di produzione data la loro relativa semplicità di sviluppo rispetto ai sistemi reattivi e alle maggiori possibili applicazioni, anche nell'ambito della risoluzione dei problemi di simulazione, grazie alla facilità di utilizzo della conoscenza a riguardo di un problema per governare la logica applicativa.

Il vantaggio evidente che si evince dall'utilizzo di un motore a regole rispetto all'utilizzo di un paradigma tradizionale basato su procedure è la netta distinzione che si riesce ad ottenere tra la gestione dei dati dell'applicazione e la logica che la governa. Questo non è possibile farlo in una programmazione Object Oriented, dove, evidentemente, la logica ed i dati di un oggetto sono intrinsecamente legati e difficili da separare.

I sistemi a regole si basano, evidentemente, sulla stesura di regole che in, una visione semplicistica, potrebbero assomigliare a condizioni (di solito a seguito della parola chiave "if" nei linguaggi di programmazione) con le relative azioni eseguite solamente nel caso in cui tutte le condizioni risultino verificate. Esistono varie tipologie di regole di cui le più significative in ambito informatico sono: Regole di Derivazione, Regole di Integrità (Vincoli), Regole Reattive (Event-Condition-Action (ECA) Rules), Regole di Produzione e Regole di Trasformazione.

La tesi è strutturata per avere una visione generale delle tipologie di motori a regole e le differenze nei vari approcci. Il capitolo 1 è strutturato appunto per discutere questo argomento: si pone come obiettivo un'introduzione alla programmazione a regole evidenziandone i vantaggi e gli svantaggi rispetto ad una programmazione più tradizionale (a procedure oppure Object Oriented). Si discute anche la metodologia di funzionamento di un generico motore a regole ponendo al centro dell'attenzione gli algoritmi che stanno alla base del processo di Deduzione. Le tipologie di processi di deduzione sono essenzialmente due: Forward Chaining (algoritmo "data-driven" cioè il sistema valuta i Fatti presenti nella Base di Conoscenza del sistema in base alle condizioni espresse nelle Regole) e Backward Chaining (algoritmo "goal-driven" cioè partendo dalla conclusione del problema verifica se qualche stralcio della Conoscenza potrebbe

verificare questa conclusione. Presuppone la conoscenza a priori della conclusione del problema).

Il capitolo 2 sviluppa in dettaglio l'analisi delle regole di produzione e delle regole ECA. In primo luogo si evidenzia la definizione di evento: un'osservabile che cambia stato in modo rilevante nel sistema. Da questa definizione si può dedurre che un evento è notificato al sistema in modo istantaneo cioè non si ha una condizione che perdura nel tempo; nasce quindi il problema della gestione degli eventi da parte di un sistema informatico.

Le regole ECA sono il meccanismo per la gestione di eventi in un sistema reattivo a regole. Sono composte dalle parti: "Event", "Condition" e "Action" dove la parte "Event" rappresenta la condizione riguardo un evento accaduto sul sistema, la parte "Condition" rappresenta l'insieme delle condizioni sui dati persistenti del sistema mentre la parte "Action" ha il compito di aggiornare e modificare lo stato del sistema per portarlo ad un nuovo stato. Le regole di produzione consentono la valutazione di condizioni solamente sui dati persistenti del sistema essendo composte dalle sole parti "Condition" e "Action". Questo implica una gestione semplificata del sistema a regole di produzione ma ovviamente il sistema realizzato non offre la possibilità di amministrare un ambiente dinamico dove la conoscenza non rimane congelata nel tempo ma risulta in continua evoluzione per mezzo di eventi non preventivati.

Il capitolo 3 vuole esaminare il confronto tra i vari motori a regole esistenti per comprenderne il meccanismo di gestione dello stato dell'ambiente. È presentato in modo dettagliato un motore a regole di produzione ben accettato dalla comunità OpenSource, JBoss Drools e lo si pone in confronto ad altri sistemi che implementano la gestione delle regole reattive. Esistono infatti ambienti sviluppati in ambito universitario che cercano di simulare problemi reali e quindi fortemente dinamici.

Infine il capitolo 4 illustra le scelte effettuate per sviluppare un sistema reattivo basandosi su di un sistema a regole di produzione e i dettagli implementativi che hanno portato allo sviluppo di questo sistema. In particolare si è scelto di implementare dei comportamenti reattivi di un sistema a regole utilizzando il sistema JBoss Drools. I comportamenti reattivi significativi che

deve possedere un sistema informatico sono legati essenzialmente alla gestione di eventi di tre grosse tipologie: eventi temporali, eventi riguardanti l'interfaccia grafica ed eventi riguardanti lo scambio di messaggi tra sistemi distribuiti. La gestione di queste tipologie di eventi porta il sistema a reagire ad un largo spettro di situazioni possibili rendendo questa implementazione una utile base per lo sviluppo di classi di sistemi che devono simulare situazioni reali o operare in ambienti fortemente dinamici.

Capitolo 1

Sistemi a Regole

Questo capitolo vuole illustrare la programmazione basata su regole, le motivazioni per sfruttare questa tecnologia nonché una classificazione generale delle tipologie di Sistemi a Regole.

1.1 Motore a Regole (Rules Engine)

L'intelligenza artificiale (A.I. acronimo di Artificial Intelligence) occupa una vasta area di ricerca in campo informatico che si concentra nell'utilizzare i computer per simulare il ragionamento insito nella mente umana. Questa area di ricerca influenza varie discipline che non sono tutte strettamente legate all'informatica come: Reti Neurali, Algoritmi Genetici, Alberi di Decisione e Sistemi Esperti. La Rappresentazione della Conoscenza (KR - Knowledge Representation-) è una branca dell'Intelligenza Artificiale che si occupa della manipolazione e della rappresentazione della conoscenza. I sistemi esperti usano la KR per riuscire a codificare e memorizzare la conoscenza che successivamente è usata per il ragionamento, per esempio si può usare una base della conoscenza per inferire i dati in ingresso, del problema, in conclusioni.

Il motore a regole è un processo software che, ricevendo in ingresso qualche stralcio di conoscenza riguardo un problema, è in grado di generare delle soluzioni al problema stesso.

La Conoscenza è, di solito, rappresentata con un insieme di fatti e oggetti mentre l'Inferenza è il processo mediante il quale da una proposizione accolta come vera, si passa a una proposizione la cui verità è considerata contenuta nella prima.

Inferire è quindi trarre una conclusione. Inferire X significa concludere che X è vero; un'inferenza è la conclusione tratta da un insieme di fatti o circostanze

Conoscenza e Inferenza sono memorizzate in proposizioni chiamate Regole. In altri termini le regole consistono in condizioni e azioni, azioni che sono eseguite solo se le condizioni sono vere.

1.1.1 Funzionamento di un Motore a Regole

Un Motore a Regole ha il compito di produrre delle deduzioni da dei Fatti a sua conoscenza (Conoscenza) utilizzando dei vincoli presenti tra i Fatti (Regole). Questo processo, chiamato Inferenza, necessita, in un sistema informatico, di strutture dati apposite per tenere traccia dei Fatti e dei Regole su questi fatti.

I Fatti saranno inseriti in una Working Memory, che può essere rappresentata da una base di dati fisica se i Fatti sono numerosi oppure da una struttura dati in memoria se è possibile descrivere il problema con pochi Fatti.

Le Regole saranno memorizzate in una Rule Base che deve risultare di rapida consultazione da parte del motore a regole data la criticità dell'esecuzione delle deduzioni.

Il processo di Deduzione può essere eseguito in due modi molto distinti in base alla composizione dei Fatti e delle Regole necessarie alla risoluzione del problema: ***Forward Chaining*** e ***Backward Chaining.***

Forward Chaining è un algoritmo "*data-driven*" cioè il sistema valuta i Fatti presenti nella Working Memory in base alle condizioni espresse nelle Regole e seleziona le Regole che possono essere eseguite. L'esecuzione delle Regole porta

a delle conclusioni che possono influenzare nuovamente i Fatti e quindi si necessita una rivalutazione della Working Memory. Iterativamente il processo prosegue fino a che non si trova la soluzione al problema.

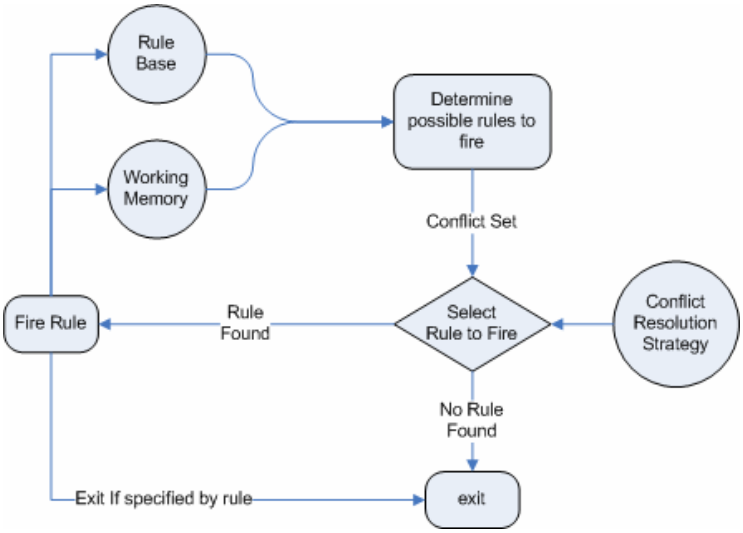

Figura 1.1. Algoritmo Forward Chaining.

Backward Chaining è l'algoritmo agli antipodi del precedente. Si può dire che lavora in modalità "*goal-driven*" cioè partendo dalle conclusioni verifica se qualche dato presente nella Working Memory potrebbe verificare questa conclusione. Lavora praticamente a ritroso rispetto all'algoritmo precedente. Questo procedimento risulta in certi casi vantaggioso in termini computazionali rispetto al precedente perché interroga solo i Fatti che effettivamente partecipano alla deduzione della conclusione finale. Presenta però degli svantaggi rispetto al caso precedente:

- valutazione di azioni che non sono effettivamente sostenute dai fatti: introduce delle iterazioni nell'algoritmo che la soluzione precedente non prenderebbe in considerazione;
- dover conoscere a priori la conclusione finale del problema ed in molti casi non è possibile determinarla.

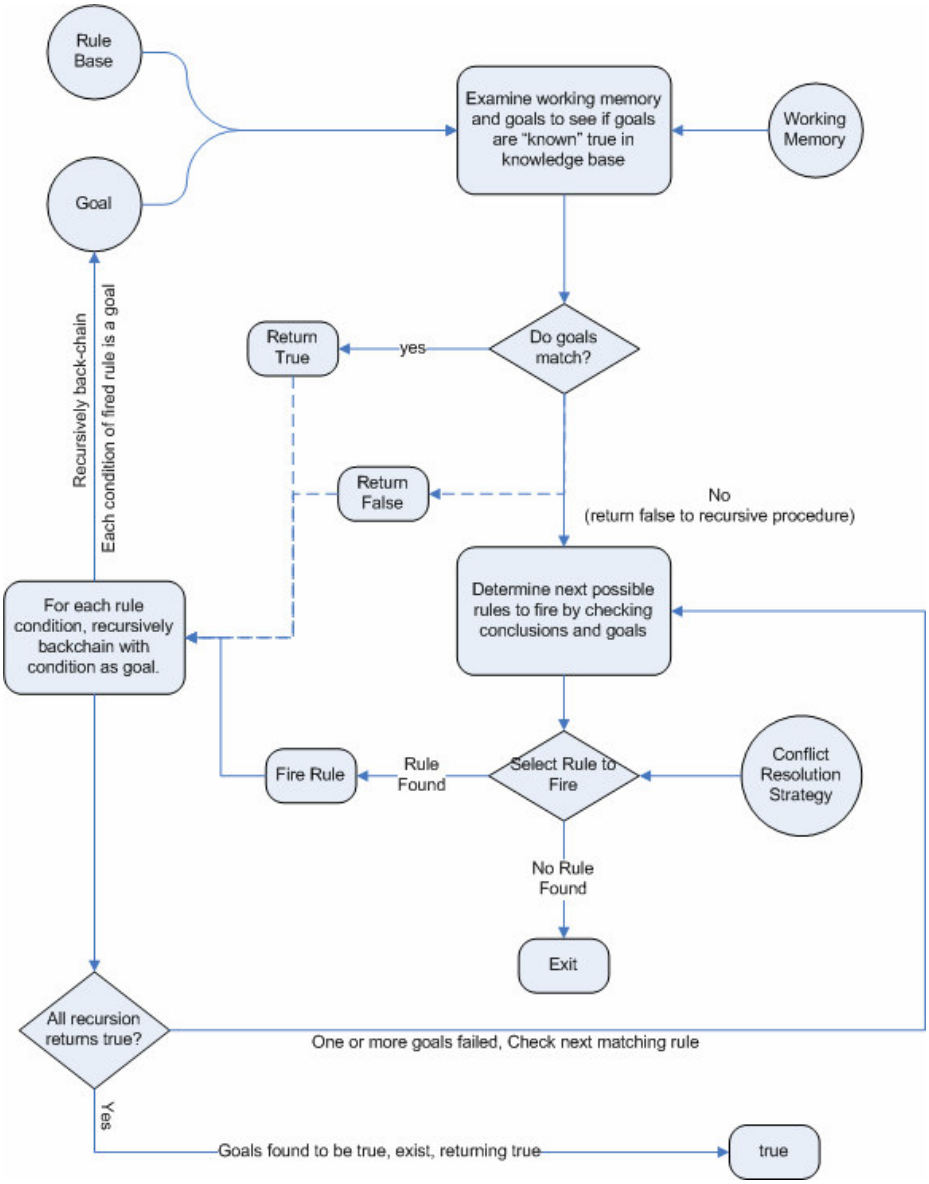

Figura 1.2. Algoritmo Backward Chaining.

1.1.2 Perché utilizzare un Sistema a Regole

La programmazione a regole offre numerosi vantaggi rispetto ad una programmazione "tradizionale" di tipo procedurale:

- **Programmazione Dichiarativa**

  I Rules Engine permettono di concentrarsi sulla risoluzione di "Cosa di deve fare" e non di "Come si deve fare" per risolvere un problema.

  Questo vantaggio è dovuto al fatto che usando le regole si possono risolvere facilmente problemi che richiedono molto lavoro dal punto di vista dell'algoritmo semplicemente esprimendo delle condizioni riguardo all'obiettivo del problema da risolvere.

  I sistemi basati su regole riescono a risolvere problemi molto complessi dal punto di vista logico riuscendo anche a spiegare come si è giunti alla soluzione e perché il sistema ha compiuto queste scelte per giungere alle conclusioni.

- **Separazione della Logica dai Dati del applicazione**

  I dati sono situati nel dominio degli oggetti dell'applicazione mentre la logica è situata nelle regole.

  Risulta fondamentale per spezzare l'accoppiamento tra dati e logica nativo della programmazione orientata agli oggetti può essere considerato un vantaggio oppure uno svantaggio a seconda del problema da risolvere. Molto spesso però applicazioni che richiedono operazioni logiche complesse permettono un buon disaccoppiamento dei dati dalla logica risultando efficiente un approccio orientato alle regole.

  La separazione dei dati dalla logica permette, ovviamente, una migliore riusabilità del codice e una migliore fase di manutenzione e miglioramento dello stesso. Risulta evidente dal fatto che si può agire sulla logica dell'applicazione senza dovere modificare tutte le strutture dati che questa richiede.

  Invece di avere la logica spalmata in tutto il domino dell'applicazione o dei controllori, può essere organizzata in modo molto efficiente in uno o più file di regole che sono esterni all'applicazione stessa.

- **Velocità e Scalabilità**

I Motori a Regole si basano su algoritmi particolari per effettuare il pattern matching delle regole con i fatti della conoscenza come l'algoritmo Rete. Questi algoritmi risultano molto efficienti nella risoluzione delle loro operazioni e sono ottimizzati per processare e inferire in base alle regole soprattutto se le regole e non cambiano rapidamente nel sistema perché gli algoritmi riescono a ricordarsi le associazioni tra dati e conclusioni fatte nel passato.

Per quanto riguarda a scalabilità, le applicazioni che sfruttano i motori a regole risultano molto più scalabili perché, grazie alla separazione tra dati e logica, si può estendere la logica dell'applicazione senza modificare la base di dati e viceversa. Le applicazioni "tradizionali" necessitano invece rielaborazione totale se si deve modificare la logica.

- **Centralità della Conoscenza**

Per usare le regole è necessario creare un contenitore che mantiene le informazioni relative alla regole stesse. Questo significa avere un unico punto dove è contenuta la verità nell'applicazione facilitando la comprensione della logica anche da parte di persone non tecniche.

- **Comprensibilità delle Regole**

Molti Rules Engine esistenti mettono a disposizione delle particolari metodologie di stesura delle regole che risulta essere scritto in un linguaggio molto simile al linguaggio naturale, questo rende le applicazioni basate su regole comprensibili sia dai programmatori sia dalle persone non tecniche che sono interessate allo sviluppo del software.

1.1.3 Quando utilizzare un Sistema a Regole

Scorrendo i vantaggi di questo tipo di approccio alla programmazione risulta evidente valutare quando la programmazione basata su regole trova i maggiori benefici rispetto ad una soluzione procedurale "tradizionale":

- **Il problema da risolvere è difficile da confinare in una soluzione basata sugli algoritmi**.

La complessità del problema obbliga a non trovare una soluzione procedurale per risolverlo

- **La logica del problema cambia in continuazione**

La logica del problema sarebbe semplice da risolvere ma la modifica di questa logica in un numero non ben determinato a priori di comportamenti, porta la risoluzione a regole a trovare la risoluzione più semplice dal punto di vista del codice

- **Disponibilità di persone abili nell'analisi ma non tecniche**

Se il team di sviluppo che si occupa della risoluzione del problema è composto da persone che possiedono una logica molto sviluppata ma non posseggono competenze programmative molto sviluppate, l'uso di Motori a Regole facilita la realizzazione della soluzione nel senso che si può costruire la logica dell'applicazione senza aver basi programmative e scrivendola in linguaggio molto simile a quello naturale.

La programmazione orientata agli oggetti è tipicamente usata per riuscire in modo semplice a costruire oggetti indipendenti che incapsulano la logica e i dati per utilizzarli. La programmazione basata su regole risulta esattamente l'inverso della programmazione OO (Object Oriented). Questo non significa che i capisaldi della programmazione OO sono errati, la programmazione basata su regole molto spesso risulta una parte di un'applicazione complessa che può usare i principi della programmazione OO ma ha il vantaggio di creare indipendenza tra i dati e il comportamento dell'applicazione stessa.

1.1.4 Quando è meglio non utilizzare un Sistema a Regole

Dalla trattazione appena fatta potrebbe apparire la soluzione migliore utilizzare la programmazione basata su regole per risolvere qualunque problema. Ovviamente questa affermazione non è corretta perché tutti i vantaggi introdotti da questa tipologia di programmazione riflette degli svantaggi nascosti ad un'analisi superficiale.

Si può immaginare che un approccio tradizionale risulta molto più efficace in problemi semplici da risolvere e che non richiedono un elevato flusso di elaborazione. Se si usasse in questo caso un Motore a Regole non risulterebbe la soluzione più performante perché l'elaborazione introdotta da un Motore a Regole, anche se ottimizzata negli algoritmi di pattern matching, è sempre molto evidente e molto spesso risulta meno veloce in termini computazionali rispetto ad un approccio più tradizionale.

1.2 Regole, classificazione generale

Le più importanti categorie di regole sono Regole di Derivazione, Regole di Integrità (Vincoli), Regole Reattive, Regole di Produzione e Regole di Trasformazione.

Il principale collegamento tra le varie categorie di regole è il concetto di LogicalFormula che è usata in ognuna delle tipologie.

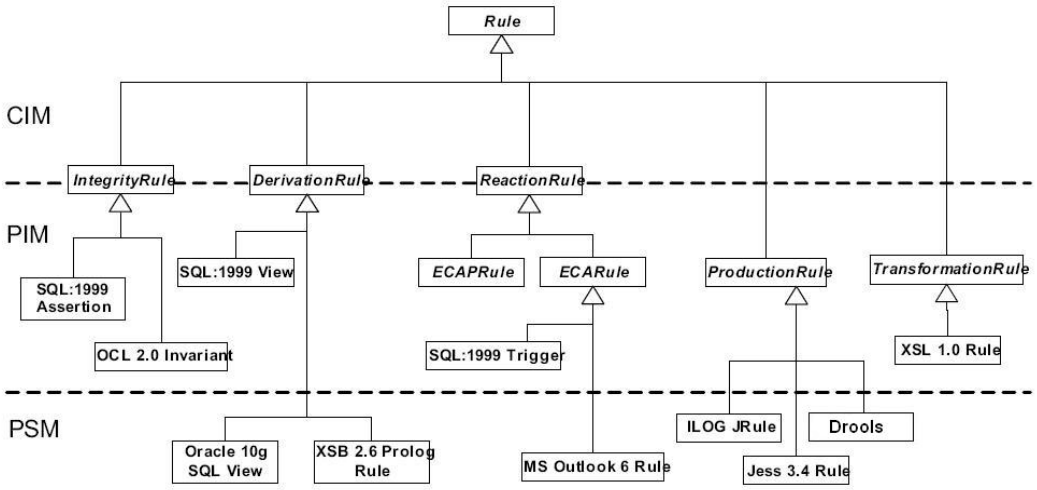

Figura 1.3. Classificazione delle Regole.

1.2.1 LogicalFormula

Una LogicalFormula è un'espressione in un linguaggio di logica formale. Quindi un'esatta definizione di LogicalFormula è diversa per ogni specifico linguaggio logico. Come una regola, la definizione di una formula viene espressa a carattere induttivo: si distingue una classe di espressioni fondamentali, chiamate *atomic formulas*, ed una LogicalFormula risulta costruita dalle formule fondamentali appena costruite.

Per esempio, le formule in una logica preposizionale sono definite come segue.

Se A *e* B *sono formule, allora*

$$(A\&B),\ (A\vee B),\ (A\supset B),\ (\neg A)$$

sono a loro volta delle formule.

Quindi una LogicalFormula è in sostanza un'espressione logica definita come combinazione di proposizioni logiche fondamentali e connettori logici.

1.2.2 Integrity Rules (Regole d'Integrità)

Le regole di integrità, conosciute come Vincoli di Integrità, consistono in una frase logica (espressa in qualche linguaggio logico come: logica dei predicati o logica temporale). I vincoli esprimono delle asserzioni che devono rimanere valide per la completa evoluzione di stati del sistema per cui sono stati definiti.

Esempio di Vincolo di Integrità Statico:

"L'autista di un'automobile **deve** avere almeno 18 anni"

Esempio di Vincolo di Integrità Dinamico:

"La conferma di un noleggio di una automobile **deve** condurre ad un'allocazione di un'autovettura nel gruppo delle vetture preferite dal cliente per la data richiesta ordinate per data"

Come si può notare dagli esempi i vincoli di integrità obbligano il sistema alla verifica della presenza di un vincolo, causa il fallimento dell'intera operazione.

1.2.3 Derivation Rules (Regole di Derivazione)

Le regole di derivazione consistono in una o più condizioni e una o più conclusioni, che sono espresse entrambe in LogicalFormula. Le condizioni sono espresse in formule senza quantificatori logici e con la possibilità di inserire sia deboli che forti negazioni.

La distinzione tra negazione debole e forte non è molto familiare nella logica tradizionale ma diviene una distinzione essenziale nella logica per programmi applicativi. Intuitivamente una negazione debole cattura il fatto dell'assenza di informazioni positive nel predicato logico, mentre una forte negazione cattura le informazioni esplicitamente negative. Nei modelli semantici dei linguaggi di programmazione una negazione debole di solito cattura il concetto di "negation-as-faliure" cioè se qualche parte del software fallisce un'asserzione, la situazione viene interpretata come una negazione. Una negazione debole invece cattura il concetto di "not" cioè delle affermazioni esplicitamente negative.

Una regola di derivazione di solito si presenta con una sola conclusione che prende la forma di un predicato logico che di solito viene valutato in forma atomica cioè al suo interno non sono presenti delle situazioni ambigue che richiedono al sistema di prendere ulteriori decisioni.

Esempi di regole di Derivazione:

"Un cliente importante è un cliente che ha depositato, sul proprio conto corrente, almeno 1,000,000€"

"Un esempio di investimento è il tasso di profitto; il tasso di profitto è valutato solo se le merci sono state comprate più di un anno fa"

1.2.4 Transformation Rules (Regole di Trasformazione)

Le regole di trasformazione consistono in due parti dalla stessa semantica cioè in qualche contesto possono essere intercambiate. Si trovano della forma

lhs ➔ rhs

dove *lhs* e *rhs* sono delle LogicalFormula.

La trasformazione consiste nel fatto che ogniqualvolta si specifica la condizione che è espressa nella parte *lhs*, questa si trasforma nella parte *rhs*.

Sono spesso usate per implementare dizionari elettronici dove l'insieme della conoscenza del problema ha una netta separazione in due blocchi: la lingua da tradurre e la lingua nella quale viene tradotta.

1.2.5 Reactive Rules (Regole Reattive)

Le regole reattive sono un insieme di regole molto importante nell'ambito dei sistemi a regole. Infatti i ricercatori e gli sviluppatori di sistemi reattivi e orientati al Web dinamico, hanno centrato la loro attenzione proprio su questa tipologia di regole e su quelle di produzione, tralasciando lo sviluppo di sistemi a regole d'integrità o di derivazione perché risultano meno efficaci nella risoluzione dei problemi reali. I sistemi che sfruttano le regole d'integrità e di derivazione sono legati allo sviluppo di applicazioni per garantire la persistenza dei dati e il soddisfacimento di vincoli "statici" (nel senso temporale del termine). Ne sono esempi le **asserzioni** presenti in Java, i **contratti** presenti in Eiffel ed **OCL** (Object Constraint Language) [12].

Le regole reattive consistono in un termine "evento", una "condizione" opzionale e un termine "azione" o una "post-condizione" (a volte entrambe).

Mentre la condizione espressa nelle regole reattive ha lo stesso significato della condizione espressa nella regole di derivazione (cioè una LogicalFormula priva di quantificatori), la post-condizione non può presentare condizioni di ambiguità; è quindi ristretta alla congiunzione di espressioni atomiche (anche negative).

I termini evento e azione possono essere specificati in diversi modi. Per esempio in nel nodello espresso in UML la semantica delle azioni potrebbe essere usata per eseguire le azioni in maniera indipendente dalla piattaforma.

Ci sono essenzialmente due tipi si regole reattive: regole di tipo Event-Condition-Action (ECA Rules) (studiate in modo particolareggiato più avanti) e le regole Event-Condition-Action-Post-condition (ECAP Rules).

La post-conzione nelle regole reattive può contenere una formula atomica, la negazione di una formula atomica o una combinazione delle stesse.

1.2.6 Production Rules (Regole di Produzione)

Le regole di produzione consistono in una condizione e un'azione. Sono diventate popolari quando si studiavano tecniche per l'implementazione si sistemi esperti (Expert System) negli anni '80. Comunque, in contrasto con le regole di derivazione, il paradigma di programmazione a regole di produzione manca di precisi fondamenti teorici e non ha una semantica formale. Questo problema è dovuto parzialmente al fatto che le categorie semantiche degli eventi e delle condizioni nella parte "Condition" della regola e le categorie semantiche delle azioni e degli effetti nella parte "Action" della regola sono mescolate tra loro.

Questa tipologia di regole è specificata più avanti in dettaglio e messa in confronto con la tipologia di regole di tipo ECA.

Capitolo 2

Analisi delle Regole ECA e di Produzione

Questo capitolo tratta in modo dettagliato le caratteristiche delle regole ECA e delle regole di Produzione evidenziandone i pregi e i difetti. In conclusione mette in risalto le differenze esistenti tra i due approcci.

2.1 Regole di Tipo ECA (Event-Condition-Action Rules)

Per segnalare un cambiamento dello stato del sistema, in un sistema basato su regole, il metodo più ovvio è la gestione tramite eventi.

Un sistema soggetto a regole di tipo ECA riceve in input, molto spesso asincroni, dall'ambiente esterno: degli eventi, cioè notifica in risposta a situazioni straordinarie.

Il sistema deve reagire gestendo in modo determinato a questi eventi e, siccome il sistema è basato su regole, deve essere in grado di gestire queste situazioni nel paradigma di programmazione a regole.

La segnalazione di un evento può avvenire seguendo due possibili strategie:

- **PUSH**: al cambiamento di stato un nodo del sistema informa tutti i nodi potenzialmente interessati a questo mutamento

- **PULL**: un nodo è in attesa di un evento per mutare il suo stato quindi chiede ad intervalli regolari ad un altro nodo (dove risiede l'informazione soggetta a qualche tipo di evento) se lo stato è cambiato con il metodo a Polling.

Le due strategie hanno ovviamente vantaggi e svantaggi che le rendono adatte a tipologie di applicazioni rispetto ad altre. Risulta però evidente che, nel caso in cui le risorse a disposizione del sistema siano un agente limitante (molto speso è così), il sistema PULL non trova una facile applicabilità dato che "spreca" risorse ogniqualvolta un nodo va a verificare il cambiamento di stato di un atro e non lo trova cambiato.

2.1.1 Struttura e Semantica

La struttura generica di una regola di tipo ECA segue molto spesso questa forma:

ON *Event*
IF *Condition*
DO *Action*

da qui anche il nome ECA.

2.1.1.1 Eventi

E' difficile specificare una definizione di evento ma la più comune può essere: un'osservabile che cambia stato in modo rilevante nel sistema.

I più comuni eventi che deve gestire un sistema informatico sono:

- **update dei dati** (locali o remoti)
- **messaggi** che vengono scambiati dalle esterno verso il sistema oppure messaggi che il sistema invia su se stesso per modificare il suo stato
- **eventi di sistema molto critici** (es. Out of Bounds, Tensione di alimentazione interrotta, CPU occupata ecc.)
- **eventi legati alla nozione di tempo** (es. aggiornamento dei dati ogni giorno alle 6.00)
- **eventi composti** che sono la combinazione di eventi elementari precedenti.

Per la gestione di un evento è necessario che l'oggetto che notifica al sistema l'accaduto, contenga delle informazioni atte a farsi riconoscere per garantire la massima possibilità di gestione da parte del sistema stesso. Ogni oggetto di notifica evento deve incapsulare questi dati:

- **Sender** dell'evento (chi o che cosa ha scatenato la situazione straordinaria);
- **quando** l'evento è stato generato (può succedere che l'istante in cui viene notificato l'evento sia diverso temporalmente da quando effettivamente è stato generato dal Sender (causa code di priorità o ritardi sul canale di comunicazione);
- **quale tipo** di evento è stato generato (questo presuppone una classificazione delle tipologie di eventi possibili che può essere per esempio la suddivisione vista all'inizio del paragrafo;

- **quali dati** sono influenzati dallo scatenarsi dell'evento (la notifica deve contenere i dati da modificare e un percorso univoco per identificare i dati soggetti a cambiamento);

- **motivo** dello scatenarsi dell'evento (quali altri eventi sono responsabili dello scatenarsi dell'evento in questione).

Gli eventi sul sistema appena descritti possono diventare parti di una regola nel momento in cui sono inseriti nella sintassi della regola stessa nella parte "Event". Il sistema di interpretazione delle regole (Interprete) deve garantire un controllo alla base di regole ogni qual volta un evento accade (solo in questo caso) e deve essere in grado di estrarre le informazioni necessarie dall'evento per riuscire ad interpretare se l'evento potrebbe portare alla veridicità di qualche regola inserita nella base di conoscenza. A tal fine la sintassi della parte "Event" di ogni regola di tipo ECA garantisce la possibilità di estrarre i dati dall' "evento" e di utilizzare gli stessi dati per verificare la parte "Condition" della regola.

Per riassumere, in un sistema di gestione a regole di tipo reattivo per riuscire a risolvere i problemi è necessario considerare le seguenti aree di interesse:

- **Estrazione dati**: i dati sono fondamentali per comprendere il mittente della segnalazione e quindi riuscire ad identificarlo

- **Composizione di Eventi**: per risolvere problemi reali il sistema deve essere in grado di gestire eventi creati dalla composizione di eventi elementari (atomici). Il sistema deve riuscire a gestire costrutti del tipo: congiunzione, disgiunzione e negazione di eventi e combinazioni di questi

- **Condizioni temporali**: hanno un ruolo fondamentale nella gestione di un qualunque sistema per risolvere i problemi di consequenzialità di alcune

operazioni atomiche (transazioni bancarie) necessita di un forte concetto di tempo per dare senso all'obiettivo del processo.

- **Eventi Aggregati**: può capitare che alcuni eventi trovano significato solo se presi in forma aggregata: per esempio lo sconto da offrire ad un cliente viene assegnato solo da un numero minimo di unità di merce acquistata. Questo necessita avere un sistema in grado di "memorizzare" eventi della stessa tipologia per poi fornire una risposta aggregata.

2.1.1.2 Condizioni

La parte condizionale di una regola di tipo ECA di solito esprime una richiesta di informazione dalla base della conoscenza del sistema. Con le regole di tipo ECA la parte "Condition" ricopre un duplice ruolo: determinare le regole da attivare ed estrarre i dati dalle variabili create nella parte "Event" per utilizzarle nella reazione alla regola (parte "Azione" della sintassi ECA rules).

La parte condizionale viene eseguita dal sotto-sistema a regole solo se l'evento si è verificato e se anche le condizioni sulla base della conoscenza risultano vere, allora la regola viene schedulata nell'insieme delle possibili regole da eseguire chiamato "Agenda".

2.1.1.3 Azioni

Mentre la parte "Event" e "Condition" delle regole di tipo ECA riescono solo a interrogare lo stato corrente del sistema senza modificarlo, la parte "Action" della regola ha il compito di aggiornare lo stato del sistema e portalo ad un nuovo stato. Le azioni tipiche che un sistema informatico deve essere in grado di gestire possono essere:

- **Aggiornare** la base della conoscenza del sistema

- **Sensibilizzare** nuovi eventi spesso asincroni ad interagire con il sistema

- **Svolgere** procedure di solito sincrone, le regole devono attendere il completamento della procedura in corso

- **Modificare** l'insieme delle regole inserite nel sistema di interpretazione: questo include l'aggiunta, la cancellazione e l'aggiornamento di regole già presenti, molto spesso questi mutamenti devono essere gestiti in modo runtime.

2.2 Regole di Produzione

Una regola di produzione è una parte della conoscenza di un sistema organizzata nella struttura

WHEN *Condition*
DO *Action.*

Lo scopo di questo tipo di regole è far evolvere il sistema attraverso stati ben determinati dopo l'esecuzione della parte "Action" della regola. Per evolvere il sistema in uno stato successivo l'azione viene eseguita solamente quando la condizione, espressa nella parte "Condition" della struttura della regola, è verificata. Lo stato ottenuto dopo l'esecuzione dell'azione può risultare incompatibile con lo stato che il sistema assumeva prima dell'esecuzione della stessa. Programmi a regole logiche si implementano spesso con questi tipi di regole.

Le regole di produzione sono usate spesso nelle applicazioni software per implementare la parte logica dell'applicazione in modo da svincolarla dalla parte di calcolo.

2.2.1 Sistemi a Regole di Produzione

Introdotti da Post nel 1943 come meccanismo di computazione del tutto generale, furono usati per la prima volta in matematica negli algoritmi di Markov, e ripresi successivamente da Chomsky , che li usò in linguistica con il nome di "*Rewrite Rule"*. In Intelligenza Artificiale fanno la loro prima apparizione con Newell e Simon che li usarono, nel 1965, per l'analisi del gioco degli scacchi, e nel l972 per rappresentare il comportamento umano nella soluzione dei problemi. Sono stati successivamente ripresi e utilizzati come modello psicologico della conoscenza umana (Young, Evertz).

Oltre ad essere visti come un meccanismo generale di computazione e quindi in ultima analisi come uno stile di programmazione (si può infatti dimostrare che hanno la potenza di calcolo delle macchine di Turing), sono largamente utilizzati come modo di rappresentare la conoscenza nei sistemi esperti.
I sistemi di produzione seguono uno schema generale composto da tre elementi indipendenti:

- **Regole di produzione**
- **Global Database**
- **Interprete delle regole**

Le regole di produzione sono divise in parte sinistra (LHS) e parte destra (RHS). La parte sinistra rappresenta la premessa, la parte destra l'azione che deve essere eseguita qualora la premessa sia verificata. Talvolta vengono chiamati 'Antecedente' e 'Conseguente'. Si possono immaginare come istruzioni del tipo:

IF *premessa* **THEN** *azione*

Ogni volta che la premessa di una regola è soddisfatta l'azione conseguente deve essere eseguita, e comporterà una modifica o un aggiunta di fatti nel Global Database.

Le regole di produzione offrono una rappresentazione della conoscenza di facile accesso e facilmente modificabile. Per tale motivo ben si addicono ai sistemi disegnati per realizzare approcci incrementali delle competenze.

Il Global database è in generale una collezione di elementi simbolici, e rappresenta lo stato del mondo. Conterrà anche quelle informazioni derivanti dall'applicazione delle regole.

Il lavoro dell'interprete è diviso in una fase di 'riconoscimento', in cui viene effettuato il test di verifica della premessa, che avviene mediante un processo detto di '*Pattern Matching'* in cui la parte sinistra della regola, viene confrontata con il contenuto del Global Database, e una fase di 'azione' in cui, qualora la premessa sia soddisfatta, viene eseguita l'azione conseguente. Ad esempio se si ha una regola del tipo:

IF *"Il corpo x è libero di muoversi"* **AND** *"Il corpo x è soggetto ad una forza"*
THEN *"Il corpo x si muove"*

ed si ha nel Global Database due fatti che corrispondono alle premesse

"Il corpo x è libero di muoversi"
"Il corpo x è soggetto ad una forza"

allora l'effetto dell'applicazione di tale regola è l'aggiunta nel Global Database del fatto

"Il corpo x si muove".

Talvolta, il lavoro dell'interprete viene affiancato da una componente di risoluzione dei conflitti, che entra in azione quando si presentano più regole

applicabili (Meta-livello). Vengono introdotte in questo modo le meta-regole, che permettono di introdurre diverse strategie per risolvere i conflitti.

L'interprete di un sistema di produzione può agire *'forward'* o *'data driven'* in contrapposizione a *'backward'* o *'goal directed'* . Nei primi il problema viene risolto, a partire da quello che si sa, producendo tutto ciò che si può ottenere dalla conoscenza iniziale, utilizzando un procedimento iterativo. Nei secondi il problema viene risolto, a partire dal risultato che si vuole ottenere, cercando di vedere se dall'insieme delle conoscenze iniziali è possibile, ottenere il risultato voluto, utilizzando un procedimento ricorsivo. La ricerca forward solitamente si usa quando da una serie di fatti noti si vuole conoscere tutto ciò che ne scaturisce. Viceversa la ricerca backward parte dalla meta e prende in esame solo le regole che lo interessano, in particolare le regole dal quale può essere dedotta, scartando le altre.

I principali vantaggi offerti dai sistemi a regole di produzione sono:

- **Facilità nell'aggiornamento** (aggiunta e cancellazione di regole e di elementi nel Global database).

- **Semplicità di lettura ed esplorazione** della conoscenza.

- **Alta modularità.** In particolare la base di conoscenza e meccanismo di inferenza sono completamente indipendenti ed esplorabili separatamente.

- **Rapidità nel costruire spiegazioni** (Con l'aggiunta di meta-regole è possibile spiegare il perché della scelta di una particolare regola).

A causa della forte modularità, unita alla difficoltà con cui si segue il flusso della computazione, i sistemi a regole di produzione risultano idonei laddove l'indipendenza tra i vari pezzi di conoscenza è di primaria importanza, ma poco utilizzati quando l'interesse ricade sul flusso della computazione. Un classico esempio in cui si utilizza un sistema a regole di produzione è la costruzione di un dimostratore automatico di teoremi in una qualche teoria formale. Identificheremo

l'insieme degli assiomi della teoria nel Global Database, l'insieme delle regole di deduzione, nell'insieme delle regole di produzione, e l'insieme di procedure per generare nuove prove nell'interprete.

.

2.2.2 Descrizione di un Sistema di Produzione

Si sta trattando un vero e proprio linguaggio di programmazione e, come ogni linguaggio di programmazione, si deve definire un modello dei dati (più o meno implicito). Essendo una tipologia di programmazione essenzialmente reattiva, la programmazione basata su regole deve garantire sia la definizione di un modello dei dati (attraverso specifici costrutti o nel linguaggio a regole stesso), sia la reazione del sistema ai cambiamenti dei dati.

Un sistema a regole di Produzione necessita di concetti diversi da un sistema a regole di tipo ECA per svolgere la sua funzione, mentre un sistema a regole di tipo ECA introduce il concetto di Evento, un sistema a regole di produzione introduce il concetto di "Working Memory" e di "Update" dei dati inseriti nella Working Memory.

La Working Memory è quell'insieme finito di dati (fatti, oggetti, ecc) su cui sono valutate le regole. I dati sono aggiunti e rimossi dalla Working Memory attraverso delle particolari istruzioni che di solito prendono il nome di *assert* e *retract*. Da quando i mutamenti dei dati sono esplicitamente notificati al motore a regole attraverso lo statement "Update", i dati posso seguire un qualsiasi modello dei dati. Alcuni linguaggi a regole di produzione includono un modello dei dati personalizzabile dagli sviluppatori mentre i linguaggi di programmazione a regole più moderni si basano su modelli dei dati esterni al linguaggio di programmazione stesso come può essere Java o XML.

2.2.3 Struttura e Sintassi

Come già evidenziato prima la struttura di una regola di produzione è

WHEN *Condition*
DO *Action*

La parte di "Condition" esprime in quali occasioni la regola potrebbe essere messa in lista di esecuzione mentre la parte "Action" esprime cosa dovrebbe accadere se la regola viene messa in esecuzione.

La condizione della regola contiene l'espressione dei dati che dovrebbero attivare la regola. Quando il motore a regole valuta la parte condizionale della regola cercherà nella Working Memory i dati che sono compatibili (match) con tutti i vincoli espressi nella regola. La natura dei vincoli dipende dal modello dei dati, comunemente si associa ai fatti delle classi di oggetti e i vincoli risultano sugli attributi di questi oggetti.

I vincoli che coinvolgono un singolo fatto della Working Memory vengono chiamati *discrimination tests* mentre i vincoli che coinvolgono più fatti in contemporanea sono detti *join test.*

2.3 Differenze fra i due approcci

Le regole di produzione e le regole ECA rispondono a due parti differenti ma complementari di un'applicazione software. Le regole di tipo ECA sono impiegate naturalmente in un contesto di applicazioni distribuite, mentre le regole di produzione si adattano molto bene ad applicazioni ricche di algoritmi logici.

2.3.1 Applicazioni Distribuite

Dato che nella struttura delle Regole di Produzione non è presente nessuna caratteristica adatta alle applicazioni distribuite, non si riesce ad implementare algoritmi distribuiti usando solo questo tipo di regole ma bisogna basarsi sugli algoritmi standard per applicazioni distribuite. Invece le Regole di tipo ECA sono molto adattabili a questo tipo di applicazioni, (basandosi per l'appunto sullo scambio di eventi (es. messaggi) tra i vari nodi.

2.3.2 Gestione dello Stato

In una applicazione basata su regole si possono distinguere tre livelli differenti del concetto di Stato:

- **Stato definito dagli oggetti e dagli eventi controllato dalle regole.**

Questa concezione di Stato è locale alla regola e prende definizione con la valutazione della regola stessa e termina con l'esecuzione dell'azione associata alla regola in questione.

- **Stato del Motore a Regole.**

Questo stato può essere visto come locale se si pensa ad un applicazione distribuita basata su regole di tipo ECA perché il motore è pensato contenuto in un singolo nodo dove l'applicazione è in esecuzione, ma può essere visto anche globale dal punto di vista delle singole regole elaborate dal motore stesso. Questa è un'importante differenza tra i due tipi di regole in discussione perché lo stesso oggetto è visto in modi diametralmente opposti.

- **Stato dell'intera Applicazione.**

Questo concetto di stato non è previsto nella struttura sia dei sistemi basati su regole di tipo ECA sia per sistemi basati su regole di Produzione perché risulta problematico gestirlo anche per le applicazioni distribuite non basate su sistemi a regole.

2.3.3 Controllo di flusso

La differenza più interessante tra le regole di tipo ECA e quelle di Produzione è probabilmente da cosa è guidata la loro esecuzione. Le regole di tipo ECA sono poste in lista per essere eseguite direttamente allo scatenarsi di un evento mentre le regole di Produzione sono poste in lista per essere eseguite quando alcune condizioni imposte sullo stato del sistema diventano vere. Anche se il mutamento delle condizioni in un sistema a regole di Produzione può avvenire attraverso lo scatenarsi di qualche evento, questi eventi sono impliciti e non utilizzabili nella sintassi della regola di Produzione.

Per fare un esempio, una regola di produzione deve esprimere il fatto che un cliente sia maggiorenne. Quando la condizione diventa vera in generale non si riesce a distinguere se la condizione è vera perché il cliente ha compiuto gli anni oppure perché c'è stata una modifica manuale dei suoi dati anagrafici. Con un sistema a regole di tipo ECA questo controllo diventa immediato ponendo disporre dell'evento scatenante la regola e così riuscire a discriminare se il cliente può ricevere il suo regalo di compleanno oppure no.

Un sistema a regole di tipo ECA offre un controllo del comportamento del sistema molto più fine rispetto ad un sistema basato su regole di produzione. Questo è dovuto al fatto che le regole di tipo ECA sono comandate direttamente dagli eventi del sistema, l'efficacia si paga però in termini di complessità del sistema e del motore che deve processare le regole di tipo ECA.

2.3.4 Rilevazione di Eventi ed Eventi Composti

Molti linguaggi a regole di tipo ECA hanno un buon supporto alla nozione di tempo attraverso costrutti composti da posizionare nella parte "Event" della regola stessa. Ci sono moltissimi tipi di applicazioni software che necessitano il trattamento degli eventi temporali: applicazioni che interagiscono con sensori (es. ogni quanto tempo fare la rilevazione), applicazioni di monitoraggio, applicazioni di controllo dei processi produttivi (es. attendere che un certo numero di eventi paralleli termini) e applicazioni che coinvolgono dei "time-out" (es. dopo un determinato periodo di tempo suona la sirena dell'allarme).

Per i linguaggi per regole di produzione questi concetti temporali sono di difficile implementazione perché non trovano spazio nella sintassi nativa della regola. Comunque certi linguaggi iniziano a garantire estensioni in questo senso.

Capitolo 3

Sistemi Esistenti

Questo capitolo vuole illustrare una panoramica sui Sistemi a Regole per mettere in luce i recenti e continui sviluppi della ricerca in quest'area dell'Intelligenza Artificiale.

3.1 JBoss Drools

JBoss Drools è un sistema in grado di gestire ed elaborare le regole di produzione implementato in Java.

Per il pattern matching sfrutta l'algoritmo Rete migliorato per la gestione di oggetti chiamato ReteOO.

Come tutti i motori a regole di produzione, anche Drools rispetta la sintassi "classica" di una Regola di Produzione.

Le figure ed i Class Diagram presenti in questa sezione sono presi dalla documentazione ufficiale di JBoss Drools che è reperibile al riferimento [4].

3.1.1 Authoring e Runtime

JBoss Drools lavora in due fasi ben distinte: la fase di Authoring e la fase di Runtime.

3.1.1.1 Authoring

Il processo di Authoring richiede la creazione di un file DRL o XML che contiene le regole e la successiva elaborazione di questo file da parte di un parser che esamina la sintassi del file di regole per verificare la presenza di errori sintattici nella stesura delle regole. L'output di questo parser è un file descrittivo intermedio che viene passato al Package Builder che produce uno o più Package. Un Package è un oggetto autonomo ed indipendente che contiene le regole introdotte dal file DRL.

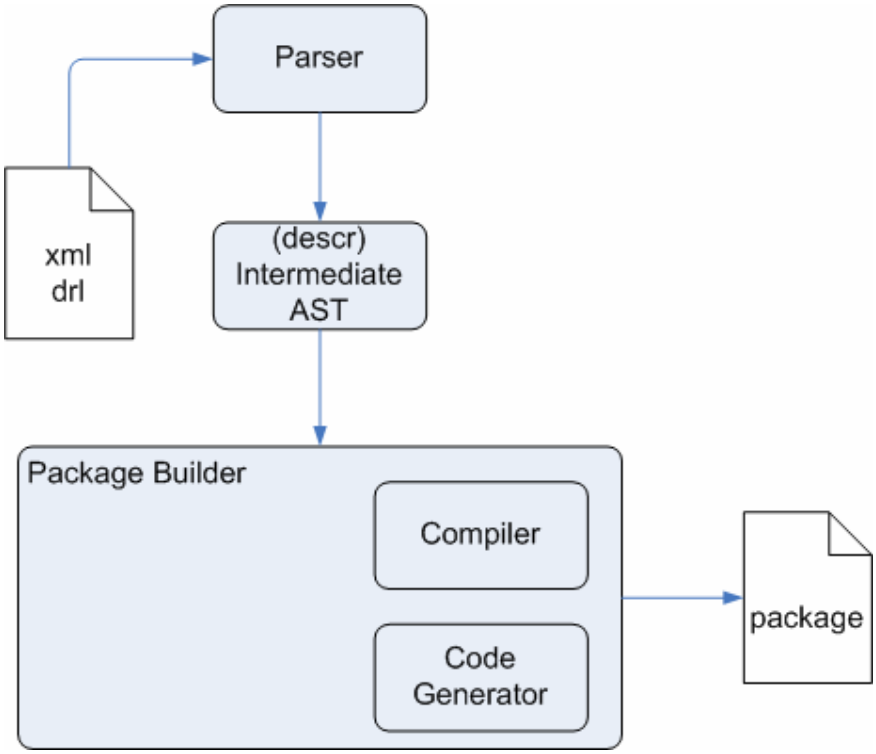

Figura 3.1. Fase di Authoring.

3.1.1.2 Runtime

La fase di Runtime riceve in input i vari Package creati in fase di Authoring. Necessita di una RuleBase che è un contenitore di Package. È un oggetto runtime perché permette l'inserimento e la rimozione di Package in fase di esecuzione dell'applicazione. Una RuleBase può istanziare una o più WorkingMemory. Una

WorkingMemory è composta da un certo numero di sotto-componenti, quali Working Memory Event Support, Truth Maintenance System, Agenda and Agenda Event Support. L'inserimento nella WorkingMemory di Oggetti porta il sistema a creare una o più Activation in seguito a questo evento. L'Agenda è responsabile dell'analisi e della gestione di queste Activation grazie ad una parte dell'EventModel implementato in Drools: Agenda Event Support.

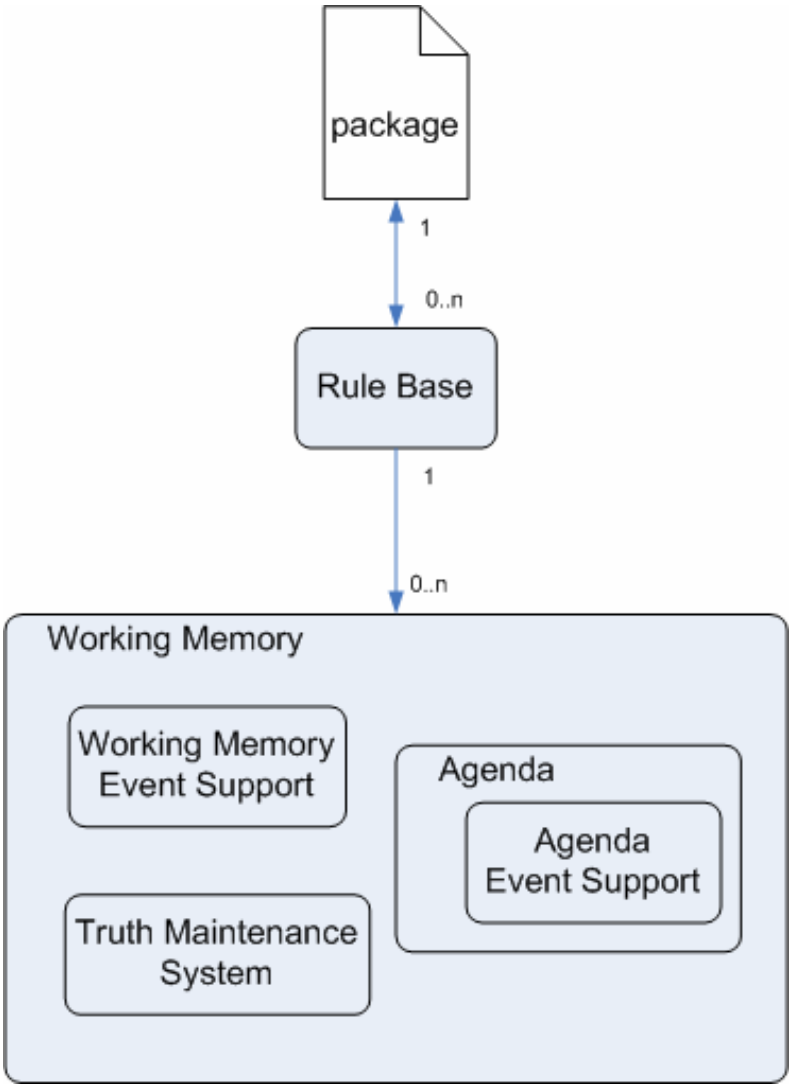

Figura 3.2. Fase di Runtime.

3.1.2 Package

Un Package ha il compito di ricevere in input un file esterno all'applicazione (che può essere DRL oppure XML) e renderlo interpretabile alla RuleBase. Per fare ciò necessita di un parser che compia la traduzione. Drools mette a disposizione del programmatore un'API di facile utilizzo che oscura tutti i passaggi di parsing dei file tramite la classe PackageBuilder. Applicando sull'oggetto di classe PackageBuilder i metodi "addPackageFromDrl" oppure "addPackageFromXml" si riescono facilmente ad aggiungere i file di regole al PackageBuilder. Il Package viene estratto dal Builder così configurato già contenente le regole interpretabili dal sistema.

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl( new InputStreamReader(
      getClass().getResourceAsStream( "package1.drl" ) ) );
builder.addPackageFromXml( new InputStreamReader(
      getClass().getResourceAsStream( "package2.xml" ) ) );
Package pkg = builder.getPackage();
```

Listato 3.1. Creazione PackageBuilder e Package.

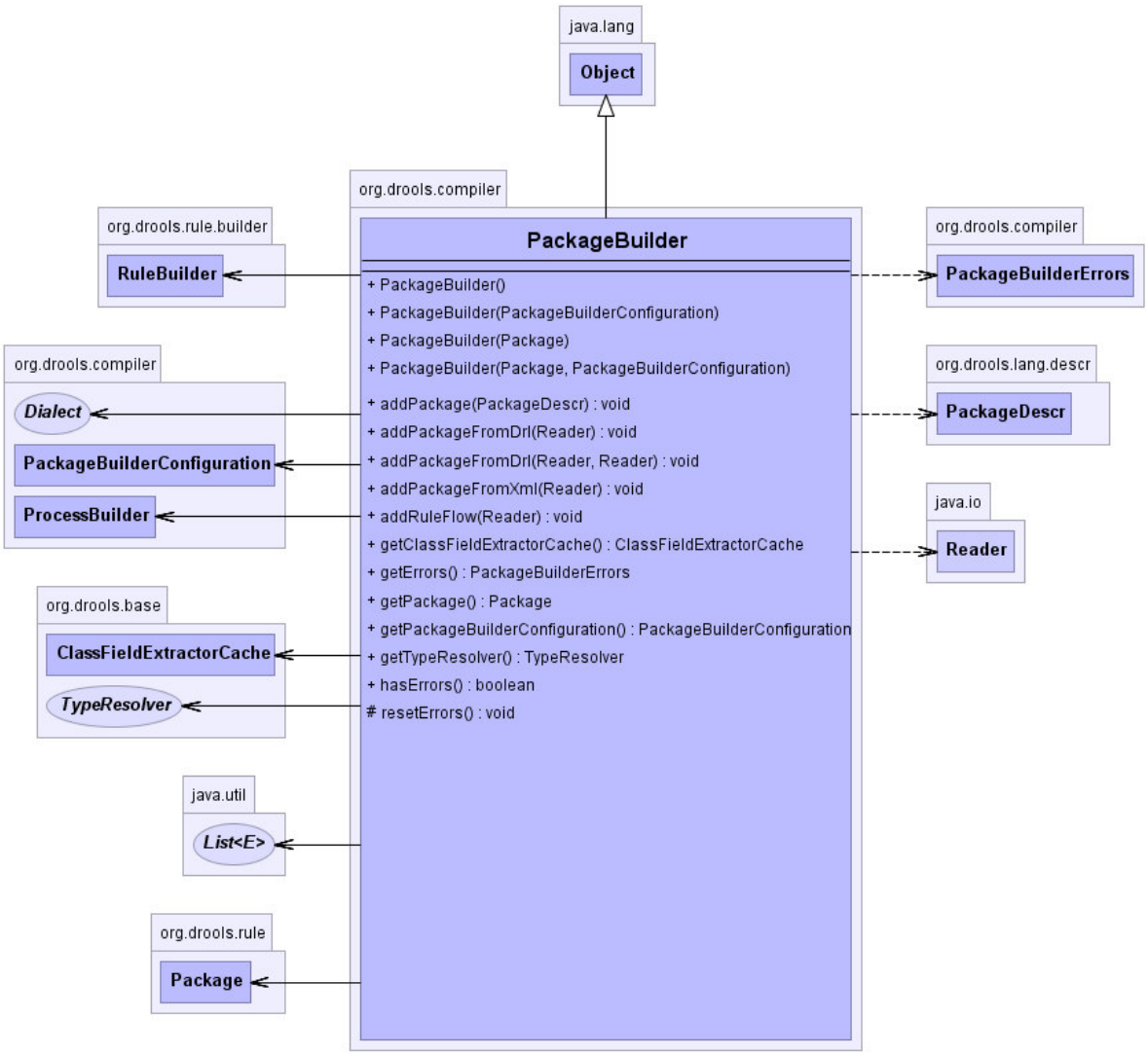

Figura 3.3. PackageBuilder.

3.1.3 RuleBase

La `RuleBase` è l'oggetto più importante necessario nella fase di Runtime. Si istanzia grazie alla classe `RuleBaseFactory` che, di default, ritorna un oggetto `RuleBase` ReteOO cioè che utilizza, per il pattern matching, l'algoritmo ReteOO.

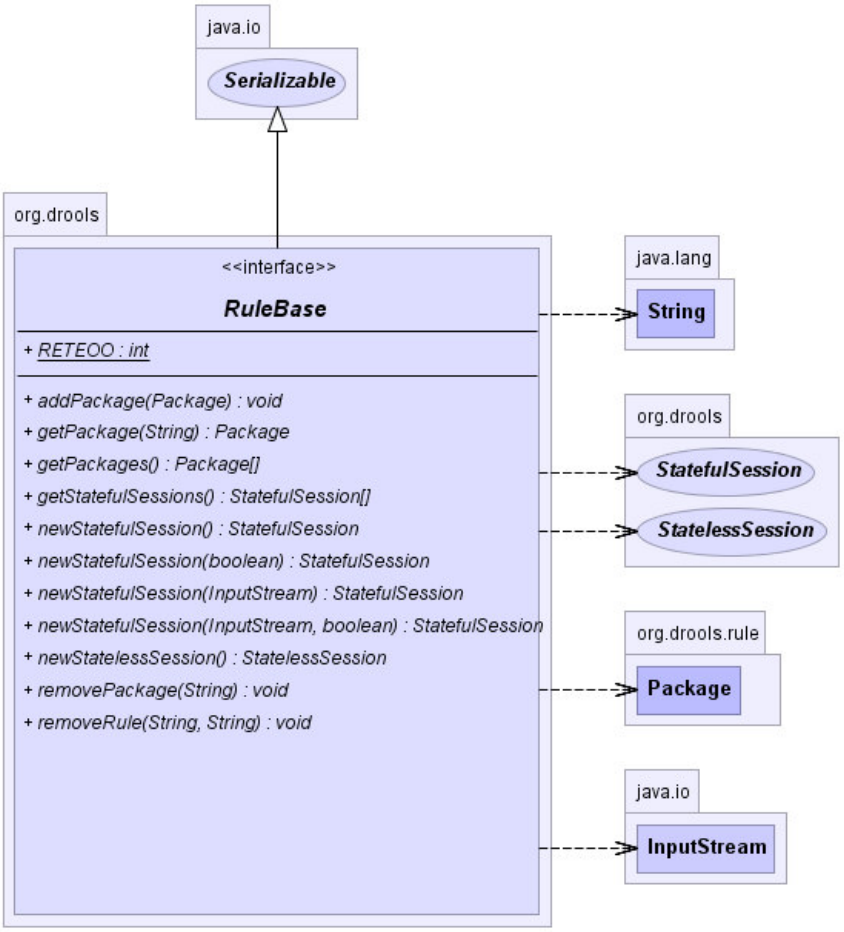

Figura 3.4. RuleBase.

```
RuleBase ruleBase  = RuleBaseFactory.newRuleBase();
ruleBase.addPackage( pkg  );
```

Listato 3.2. Creazione RuleBase ed associazione con il Package.

La più comune operazione che si applica alla `RuleBase` è la creazione di una sessione che può essere Stateful o Stateless.

La differenza tra le due tipologie sta nel fatto che la Stateful Session mantiene lo stato dei fatti e delle regole inserite mentre la Stateless Session il pattern mathing viene fatto su un solo fatto e alla fine della valutazione delle regole la sessione viene terminata. Nella maggioranza dei casi i problemi reali necessitano di un ripetuto utilizzo dei fatti inseriti nella sessione quindi viene praticamente sempre utilizzata la Stateful Session.

3.1.4 Working Memory

L'oggetto `WorkingMemory` (**WM**) mantiene un riferimento a tutti i **Fatti** (sotto forma di oggetti) inseriti al suo interno, risulta quindi un oggetto Stateful. Questo rende molto facile verificare se un Fatti inserito ad un certo istante, a causa dell'inferenza dell'Engine, viene modificato dall'attivazione ed esecuzione delle regole.

I **Fatti** sono a tutti gli effetti degli oggetti Java su cui le regole posso agire e modificare. Il motore a regole non "clona" i fatti che si inseriscono in WM ma ne crea una copia per riferimento. I Fatti sono quindi i dati dell'applicazione e non possono essere dati nativi del linguaggio Java come `String`, `Integer` o `Float`, devono essere oggetti che possiedono i metodi Getter e Setter e quindi oggetti simili a dei JavaBean. Questo è dovuto al fatto che per interagire con i Fatti, il motore a regole deve poter avere accesso allo stato degli oggetti e deve essere in grado di modificarne lo stato stesso.

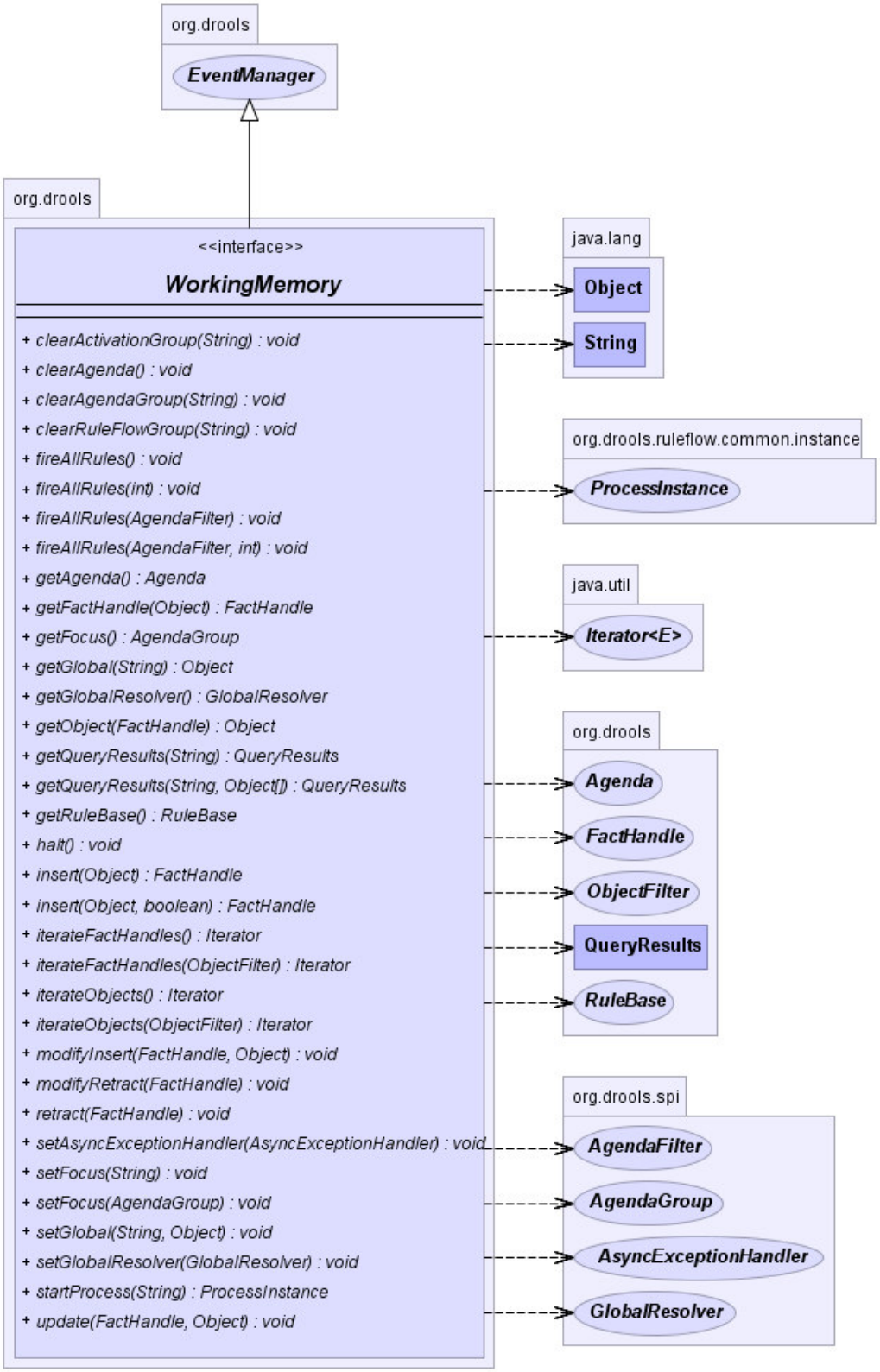

Figura 3.5. WorkingMemory.

Sui Fatti inseriti nella WM si possono eseguire svariate operazioni: **Inserimento**, **Estrazione** e **Modifica.**

L'**Inserimento** consente l'aggiunta di un fatto al sistema a regole. L'inserimento viene tracciato grazie ad un Handler specifico che rimarrà l'unico oggetto, che interagisce con l'applicazione, in grado di avere un vincolo di riferimento con il fatto inserito in WM. Questo riferimento permette anche di ritirare o modificare l'oggetto dal programma applicativo rendendo l'interazione con il motore a regole sempre più dinamica.

All'atto di inserimento di un fatto, il sistema esegue il matching con tutti i Fatti e le regole presenti nella Base della Conoscenza predisponendo un grafo di attivazioni possibili. Si capisce che l'inserimento dei Fatti non comporta nessuna esecuzione regole associate ai Fatti inseriti ma si ha solo la notifica di una possibile attivazione. L'esecuzione effettiva dell'azione espressa nella regola si ha solo dopo l'esecuzione del metodo `fireAllRules()` che attiva effettivamente il motore a regole e modifica lo stato del sistema secondo i vincoli espressi dalle regole.

```
Cheese stilton  = new Cheese( "stilton" );
FactHandle stiltonHandle = session.insert(stilton);
```

Listato 3.3. Inserimento di Fatti nella WorkingMemory.

L'**Estrazione** consiste nella eliminazione di un Fatto precedentemente inserito nella WM. Questa azione porta il Fatto a non essere più valutato nelle decisioni del motore a regole e le regole che si erano attivate per la presenza di questo fatto vengono disattivate. C'è da notare che alcune regole possono valutare la presenza o meno di un Fatto nella WM e l'estrazione di un Fatto può anche provocare l'attivazione di nuove regole. L'Estrazione viene eseguita tramite l'utilizzo dell'Handle apposito per fatto in questione che era stato associato al momento dell'asserzione (Inserimento) nella WM.

```
Cheese stilton  = new Cheese( “stilton” );
FactHandle stiltonHandle = session.insert( stilton );
....
session.retract( stiltonHandle );
```

Listato 3.4. Estrazione di Fatti della WorkingMemory.

La **Modifica,** tradotto dalla parola chiave **Update**, è un’operazione che causa un mutamento di stato di un Fatto precedentemente inserito nella WM. Anche la Modifica causa un “ri-processamento” del Rule Engine con la conseguente rivalutazione di tutti i matching tra le regole ed i Fatti della WM. La Modifica di un Fatto viene eseguita allo stesso modo dell’Estrazione, per mezzo dell’Handler specifico del Fatto.

Internamente al sistema, un’operazione di modifica causa prima l’Estrazione del dato dalla WM con un successivo Inserimento dello stesso Fatto salvo le modifiche apportate.

```
Cheese stilton  = new Cheese( “stilton” );
FactHandle stiltonHandle = session.insert( stilton );
....
stilton.setPrize( 100 );
session.update( stiltonHandler, stilton );
```

Listato 3.5. Modifica di Fatti nella WorkingMemory.

3.1.5 Stateful Session

La classe `StatefulSession` estende il concetto di `WorkingMemory` permettendo anche operazioni asincrone sui Fatti gestiti. Presenta anche un metodo `dispose()` che permette di eliminare le strutture interne alla WM senza creare dei memory leak.

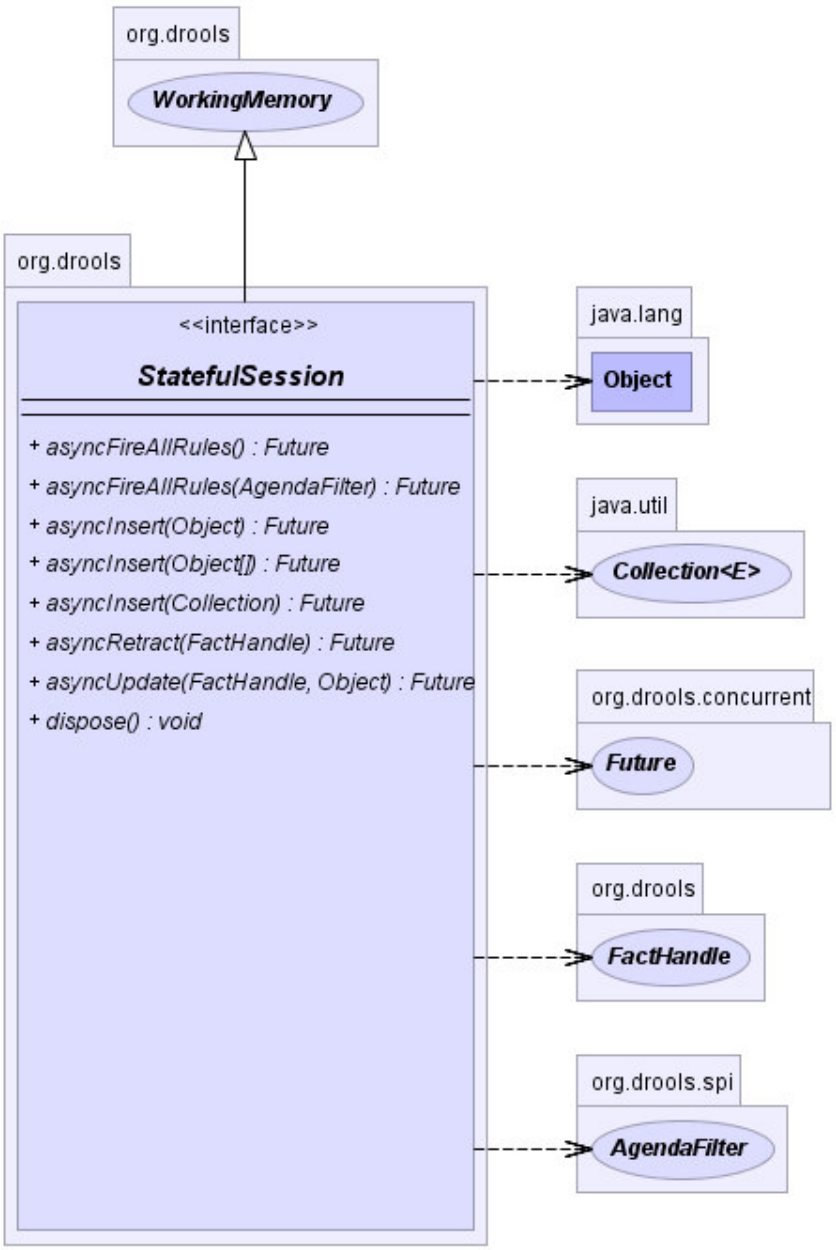

Figura 3.6. StatefulSession.

```
StatefulSession session = ruleBase.newStatefulSession();
.... //insert, retract and upload Facts
session.fireAllRules();
```

Listato 3.6. Creazione della StatefulSession ed esecuzione del Drools Engine.

3.1.6 Stateless Session

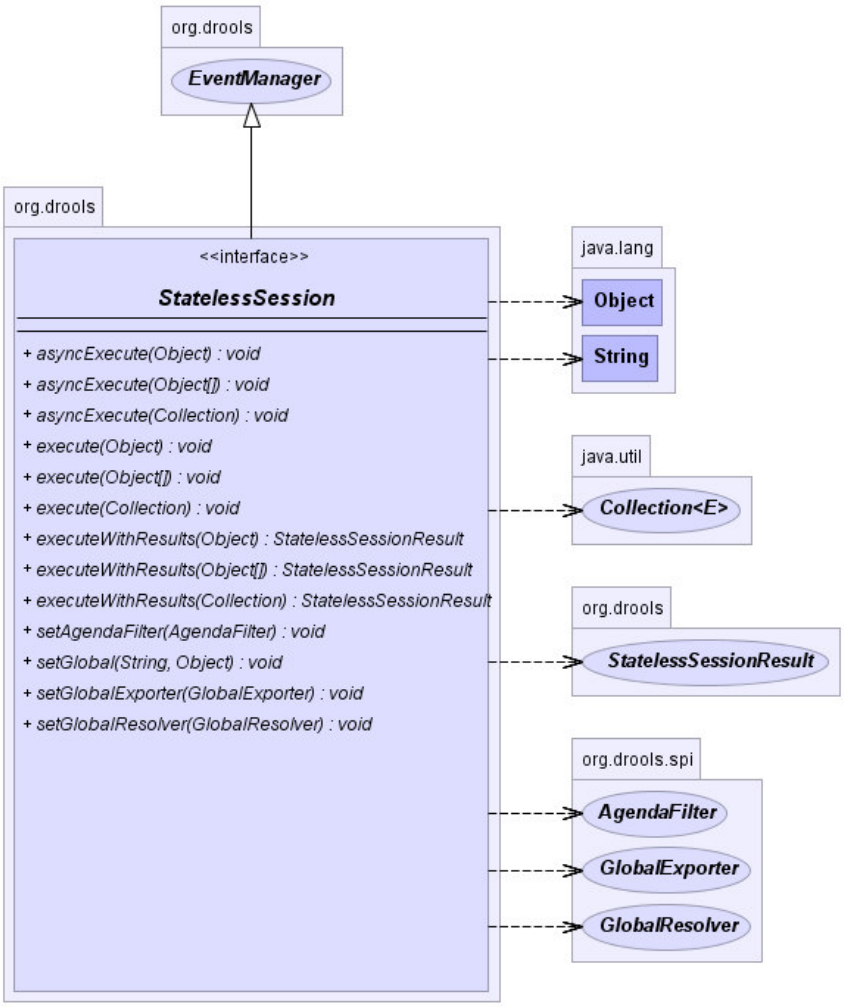

Figura 3.7. StatelessSession.

La classe `StatelessSession` funziona esattamente come la `WorkingMemory` senza estendere nessuna funzione. Presenta un API molto ridotta che permette di risolvere solo problemi semplici, ma offre il vantaggio di non caricare di lavoro eccessivo la macchina su cui opera l'applicazione.La `RuleBase` non mantiene nessun riferimento sulla `StatelessSession` e quindi non serve neanche il metodo `dispose()`. Sulla `StatelessSession` si può utilizzare il metodo `execute()` che riceve un Fatto oppure una lista di Fatti ed esegue internamente un'istruzione `fireAllRules()` che modifica lo

stato del sistema senza avere il controllo dell'istante in cui si modifica. Successivamente la sessione termina e una successiva chiamata di `execute()` genererà una nuova sessione.

```
StatelessSession session = ruleBase.newStatelessSession();
session.execute( new Cheese( "cheddar" ) );
```

Listato 3.7. Creazione della StatelessSession ed esecuzione del Drools Engine.

3.1.7 Agenda

L'Agenda è una caratteristica dell'algoritmo RETE, l'algoritmo di pattern matching implementato in Drools nella versione estesa RETEOO. Durante le azioni della WM, cioè Inserimento, Modifica e Estrazione di Fatti, se la condizione espressa in una regola viene pienamente verificata (fully matched), il sistema crea un'Attivazione e questa viene posta in un oggetto Agenda. L'Agenda ha il compito di decidere l'ordine di esecuzione di queste attivazioni, che possono essere contemporanee, secondo una specifica politica di assegnazione chiamata: **Conflict Resolution strategy**.

Il motore a regole opera in un modello a "2 fasi" ricorsive (*"2-phase model"*):

- **Working Memory Actions**: questa fase è la più complessa ed anche quella più onerosa in termini computazionali. Ad ogni azione eseguita sulla WM il motore determina le regole che si possono attivare dallo stato corrente del sistema. Alla successiva chiamata di `fireAllRules()` il sistema passa nell'altra fase, quella denominata Agenda Evaluation.

- **Agenda Evaluation**: permette di scegliere quale regola eseguire se sono presenti più attivazioni contemporanee. Se non è presente nessuna regola, cioè se nessun Fatto inserito nella WM ha un match pieno con le condizioni espresse nelle regole, allora l'algoritmo termina e non produce nessun cambiamento alla WM. Se invece si trovano regole in grado di

essere eseguite, la loro esecuzione poterà una mutazione dello stato della WM che dovrà ancora essere analizzata. Il processo quindi ritorna nella fase 1 e continua ad iterare fino a che l'Agenda non sarà vuota.

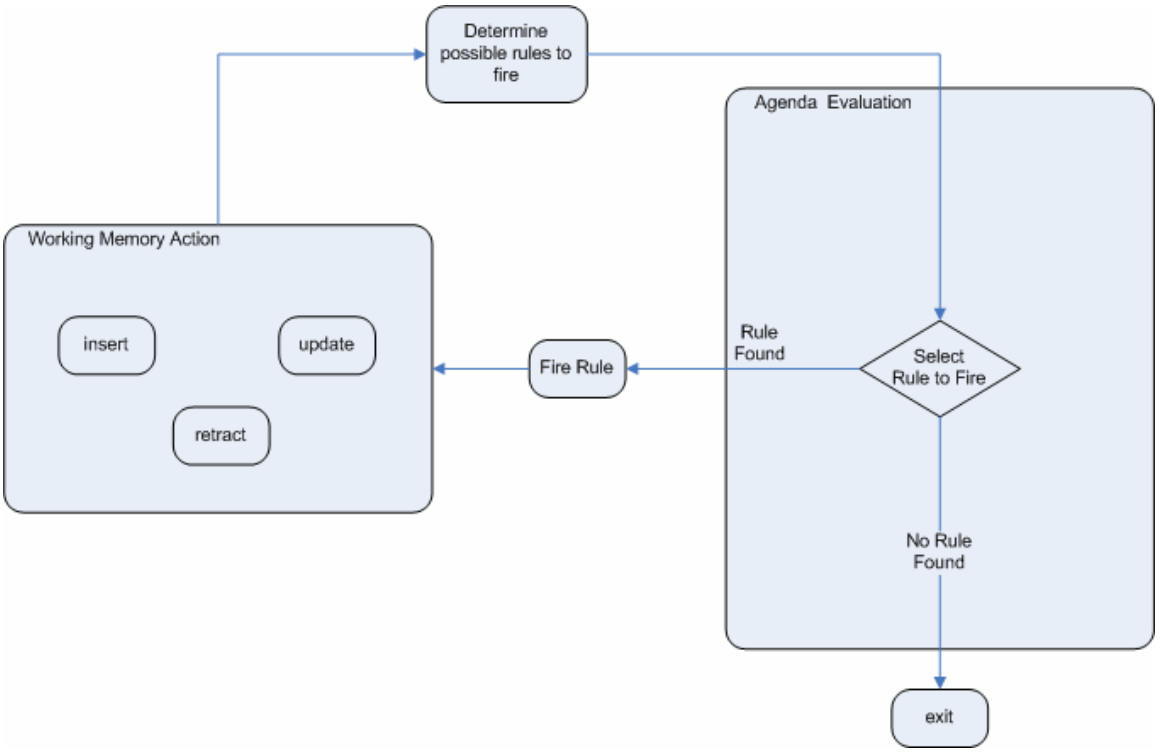

Figura 3.8. "*2-phase model*".

Da questo procedimento si nota la quantità di elaborazione richiesta solo per il semplice inserimento di un Fatto nella WM. Si evidenzia infatti un problema di calcolo computazionale al crescere dei fatti e delle regole del sistema, uno dei limiti di questo approccio alla risoluzione dei problemi reali.

3.1.8 Conflict Resolution

La risoluzione dei conflitti avviene quando in Agenda ci sono più regole che possono essere eseguite. Ci sono dei casi nel quale l'ordine di esecuzione delle regole è di vitale importanza per la perfetta soluzione del problema. Per esempio se si esegue la regolaA, la regolaB viene rimossa dall'agenda. Si può notare che

l'esecuzione della regolaB prima della regolaA causa un incorretto comportamento del sistema che prevedeva la rimozione dalle regole attivate della regolaB quando si verificano le condizioni per la regolaA.

Drools può risolvere questo conflitto in due modi: mediante "Salience" oppure con il metodo LIFO (last in, first out).

Il metodo consigliato è l'utilizzo della "Salience" o priorità. Ad ogni regola lo sviluppatore può assegnare un numero intero che farà da discriminante sull'ordine di esecuzione delle regole (ad un numero più alto corrisponderà una priorità maggiore e quindi l'esecuzione sarà anticipata rispetto alle altre regole).

La priorità di tipo LIFO, invece, assegna un numero incrementale ad ogni attivazione di regole. Questo numero non può essere modificato dall'applicazione e viene eseguita la regola con numero di attivazione più basso. Questo provoca impredicibilità nelle applicazioni perché non si ha un completo controllo sul flusso di regole eseguite che può portare l'applicazione a condizioni di instabilità. A volte questa gestione dei conflitti è un punto di forza per applicazioni specifiche che sono consapevoli dell'imprevedibilità dell'evoluzione dello stato della WM. Per esempio se si vuole introdurre una quantità di imprevedibilità a situazioni che possono avvenire in contemporanea e che producono effetti simili al sistema, una risoluzione dei conflitti di tipo LIFO è l'ideale per questo scopo.

3.1.9 Agenda Filter

Drools permette di porre dei filtri sulle attivazioni delle regole in Agenda. Questi filtri sono opzionali ma permettono allo sviluppatore di avere un controllo quasi totale sull'evoluzione del motore a regole permettendo di bloccare attivazioni di regole che porterebbero l'applicazione in uno stato non previsto.

Si realizza tramite l'implementazione di un'interfaccia di nome `AgendaFilter` che ha il compito di selezionare le attivazioni presenti in agenda. L'operazione che è concessa è il bloccaggio della attivazione che si è scatenata tramite il metodo `accept()`. Quando si scatena questo evento lo si

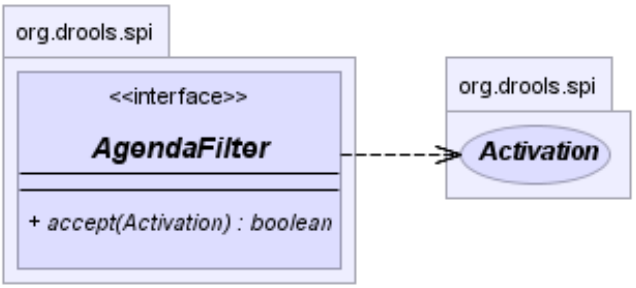

Figura 3.9. AgendaFilter.

può intercettare e bloccare a seconda della politica che si adotta oppure lo si lascia agire sullo stato della WM.

3.1.10 Modello a Eventi

Drools offre la possibilità di controllare le operazione compiute in fase di Running tramite un Modello ad Eventi. Con questo package si possono avere notifiche e agire in conseguenza agli eventi che si scatenano nel Rule Engine come per esempio: l'attivazione o l'esecuzione di una determinata regola, l'asserzione o la modifica di un fatto in Working Memory ecc.. Grazie al Modello ad Eventi si riesce a separare la fase di jogging/auditing dal resto dell'applicazione permettendo, in fase di debug e testing, di avere una visione particolareggiata dell'evoluzione dell'Engine.

Drools mette a disposizione tre tipologie di listener degli eventi:

- **`WorkingMemoryEventListener`**
  Permette di notificare tutti i mutamenti dei Fatti inseriti nella WM.
  Si ha una visione globale di tutte le reazioni del sistema alla modifica della WM per mezzo di oggetti specifici che incapsulano le informazioni necessarie alla gestione dell'evento avvenuto: `ObjectInsertedEvent`, `ObjectRetactedEvent` e `ObjectUpdatedEvent`.

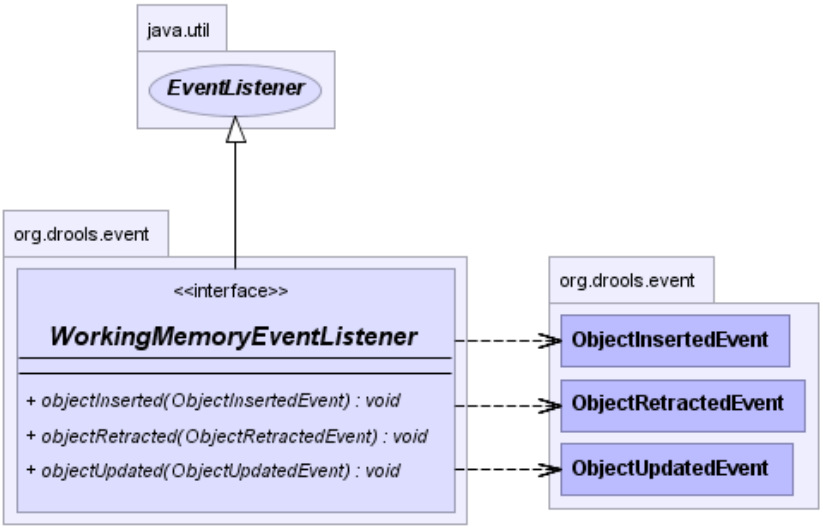

Figura 3.10. WorkingMemoryEventListener.

- **`AgendaEventListener`**

  Permette di notificare tutte le attivazioni delle regole, sia l'istante prima dell'attivazione sia l'instante dell'effettiva esecuzione ed anche i cambiamenti che subisce la WM dopo l'esecuzione delle regole.

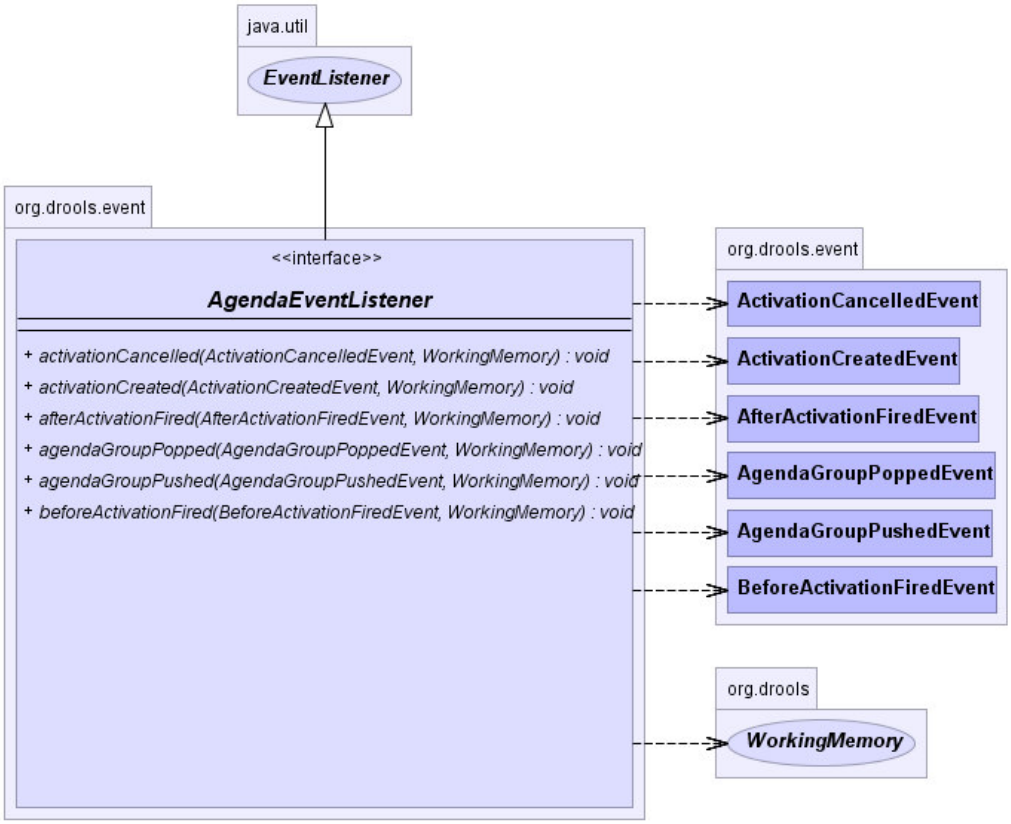

Figura 3.11. AgendaEventListener.

- **`RuleFlowEventListener`**
  Permette di notificare le modifiche al `RuleFlow` (strumento che risulta necessario per gestire i diagrammi di flusso con il Motore a Regole).

Drools offre delle implementazioni di default di queste interfacce che hanno solo lo scopo di debugging dell'applicazione. Ovviamente si può creare un'implementazione particolare che risolve il problema posto.

3.1.11 Regole

Drools ha un linguaggio nativo per scrivere le regole diverso da XML. Questo linguaggio è molto meno preciso di XML in termini di formalità ma permette di usare il linguaggio naturale per esprimere le regole. Risulta così un ottimo vantaggio in termini di facilità di stesura delle regole e del controllo per eventuali errori semantici. È permesso inoltre estendere il linguaggio con dei Domain Specific Languages (DSL) che permettono di adattare la stesure delle regole in base al problema da risolvere (descritti in seguito *par. 3.1.11.7*).

3.1.11.1 File di Regole

In Drools i file di regole sono dei semplici file testuali in estensione *.drl.* In un file drl si possono immagazzinare una molteplicità di regole, di query e di funzioni, nonché delle dichiarazioni di risorse come: import, globals, attibutes che, quando assegnati, si possono utilizzare in tutto il file da tutte le regole.

3.1.11.2 Costruzione delle Regole

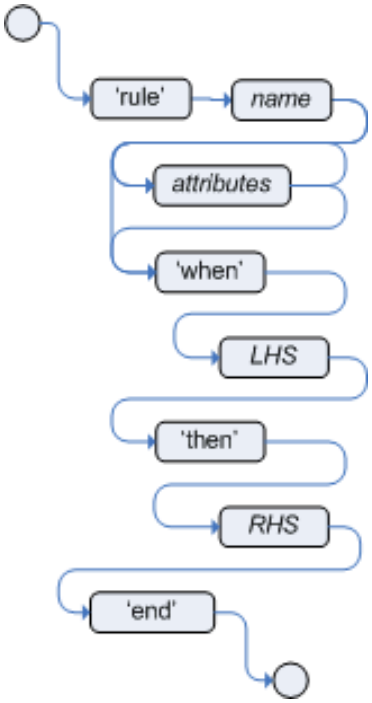

Figura 3.12. Struttura della Regola.

```
rule “<name>”
      <attribute>*
when
      <conditional element>*
then
      <action>*
end
```

Listato 3.8. Sintassi di una generica regola.

Il listato 3.8 evidenzia le seguenti parti:

- **il campo "<name>”** rappresenta il nome mnemonico della regola;
- **i campi <attribute>*** (opzionale e ce ne possono essere più di uno) rappresentano gli attributi che può assumere la regola;

successivamente la regola assume la stessa sintassi di una qualsiasi regola di produzione:

- **la parte <conditional element>*** indica l'elenco delle condizioni che permettono l'attivazione della regola ed assume il significato della parte "Condition" della generica regola di produzione. È spesso indicato con il termine LHS (Left Hand Side).

- **la parte <action>*** rappresenta l'insieme delle azioni che il motore deve eseguire quando le condizioni poste dopo la parola chiave when sono verificate. Questa lista ha la stessa semantica della parte "Action" della della generica regola di produzione. È spesso indicato con il termine RHS (Right Hand Side).

La regola di produzione viene attivata quando tutta la lista di **<conditional element>*** risulta verificata mentre la stessa viene eseguita quando effettivamente viene messa in esecuzione. Questa distinzione è necessaria perché Drools mantiene un'agenda di regole attivabili cioè regole che sono tutte candidate all'esecuzione. Spetta al motore la decisione di quale regola eseguire per prima nell'elenco di regole attivabili tramite dei meccanismi decisionali che possono essere controllati mediante particolari attributi.

3.1.11.3 Attributi

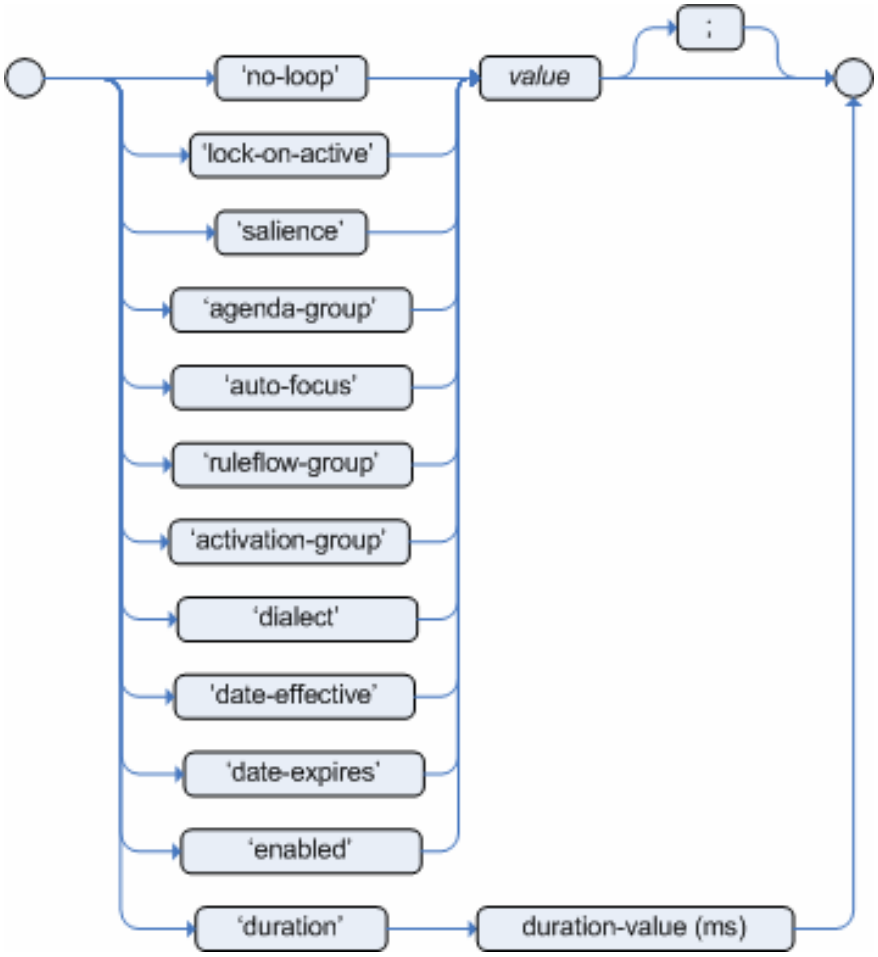

Figura 3.13. Attributi della Regola.

Gli attributi alle regole permettono di impostare delle condizioni preliminari sulla regola e quindi permette al motore di gestire le regole con attributi in modo differenziato in base al tipo di attributo espresso.

Come si evince in figura una regola può presentare uno o più attributi ma anche non esprimerli, se li esprime ogni attributo permette alla regola di comportarsi in una metodologia diversa.

L'attributo più utilizzato è '**salience'** che permette alla regola di prendere una priorità diversa rispetto alle altre se due o più regole dovrebbero andare in esecuzione contemporaneamente. Il valore associabile a '**salience'** è un valore intero positivo o negativo e la priorità è associata alla regola che ha un valore di '**salience'**.

Per la gestione sequenziale delle regole (cioè riuscire a gestire eventi temporali) si possono utilizzare attributi quali **'date-effective'**, '**date-expires'**, '**enabled'** e '**duration'**.

L'attributo **'date-effective'** imposta una condizione preliminare sull'inizio dell'attivazione della regola. La regola potrà essere attivata, cioè controllato il suo elenco di condizioni per verificare se risultano tutte verificate, solo dopo che la data corrente ha superato il valore espresso in **'date-effective'**.

L'attributo **'date-expires'** è simile al funzionamento dell'attributo **'date-effective'** solo che la regola non può essere più attivata fino a che la data corrente non supera il valore espresso nel **'date-expires'**.

L'attributo **'enabled'** ha il compito di riferire al motore a regole se la regola è abilitata o meno. Ovviamente una regola non abilita non viene presa in considerazione dal motore mentre se la regola è abilitata viene elaborata dal motore.

Infine l'attributo **'duration'** permette l'esecuzione della regola ritardata di una quantità di tempo (espresso in millisecondi) a partire dal momento dell'attivazione della stessa. Ovviamente la regola va in esecuzione solo se le condizioni si mantengono "vere" nel periodo impostato dall'attributo **'duration'.**

3.1.11.4 Left Hand Side (Condizioni)

Figura 3.14. ConditionalElement.

Il Left Hand Side (LHS) è il nome standardizzato per la parte condizionale di una regola in Drools. La parte condizionale contiene i costrutti semantici per riuscire a valutare una regola. La parte LHS contiene degli specifiche

dichiarazioni ed ogni dichiarazione può contenere al massimo un fatto inserito nella Working Memory. Dato che le condizioni, spesso, devono agire su più fatti contemporanei, l'LHS contiene dei particolari pattern (descritti in seguito *par. 3.1.11.6*) per permettere a concatenazione di fatti. Il più comune pattern è '**and'** che ovviamente unisce due dichiarazioni con il costrutto logico 'and' (l'LHS risulta vera se tutti le dichiarazioni legate da 'and' sono verificate contemporaneamente). Ovviamente Drools prevede anche l'operazione logica di '**or'** tra le dichiarazioni; questo implica che l'LHS risulta verificata se almeno una delle dichiarazioni è vera.

3.1.11.5 Right Hand Side (Azioni)

Il Right Hand Side (RHS) è il nome standardizzato per definire la parte "Action" di una regola in Drools. La parte di azione contiene il codice da eseguire se le precondizioni (attributi) e le condizioni sono verificate.

È sconsigliabile usare del codice condizionale nella parte RHS della regola perché si vìola la natura atomica della regola stessa - *"when this, then do this"*, non *"when this, maybe do this"*. Le istruzione da inserire nella parte RHS sarebbe meglio fossero poche e se si rende necessario inserire istruzioni di tipo condizionale allora ci sarà sicuramente la possibilità di suddividere il codice condizionale in più regole in modo che ogni parte RHS di ogni regola sia di tipo atomico. Lo scopo principale della parte RHS della regola è di inserire, togliere e modificare l'insieme dei fatti asseriti nella Working Memory. Per fare ciò Drools mette a disposizione dei metodi per modificare i dati nella Working Memory senza far riferimento all'istanza della Working Memory allocata:

- **`update(object, handle);`** informa l'engine che un fatto (che è inserito nella parte LHS della regola) è modificato rispetto ad uno stato iniziale, questo provoca una riconsiderazione delle regole presenti nella Working Memory;

- **`insert(new Something());`** inserisce nella base di conoscenza un nuovo fatto di tipo `Something();`

- **`insertLogical(new Something());`** inserisce il modo logico il fatto di tipo `Something()`. Questo significa che il fatto viene mantenuto in memoria fino a che lo stato del motore a regole è compatibile con le condizioni per cui l'oggetto di tipo `Something()` è stato inserito;

- **`retract(handle);`** rimuove l'oggetto gestito da "`handle`" dalla Woking Memory.

3.1.11.6 Pattern

Il pattern è la componente più importante di un ConditionalElement presente nell'LHS. Il seguente diagramma Entità/Relazione evidenzia la composizione dell'elemento pattern e si nota la vastità di condizioni che si possono esprimere nell'LHS. Ovviamente questo pattern può essere concatenato tramite gli operatori 'and', 'or', 'not', 'exist', ecc. che moltiplicano le possibilità di combinazioni possibili per creare condizioni molto complesse e veramente selettive sui Fatti della Working Memory.

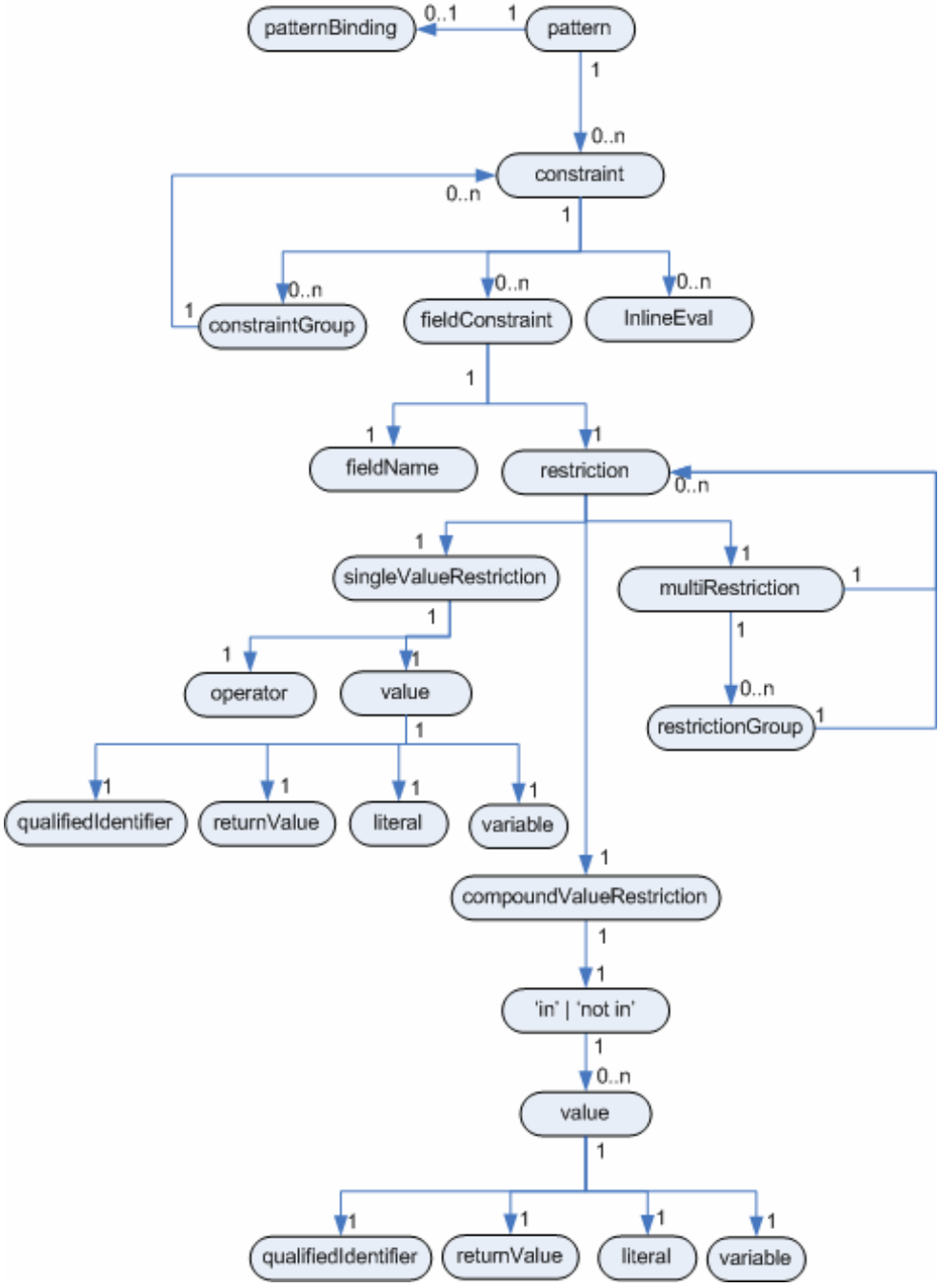

Figura 3.15. Diagramma E/R del Pattern.

3.1.11.7 Domain Specific Languages (DSL)

Come già specificato prima, i DSL sono un metodo per adattare la scrittura delle regole al domino specifico del problema. Sono collegati al linguaggio a

regole standard tramite dei file specifici e permettono di utilizzare tutte le caratteristiche del linguaggio a regole in modo semplice e molto simile al linguaggio naturale.

I DSL sono utilizzati per creare un livello di separazione tra la fase di authoring delle regole e il dominio degli oggetti sul quale opera l'Engine. Possono anche servire come sorta di "template" di condizioni o azioni necessarie in modo ripetuto nelle varie regole in modo da non dover riscrivere le stesse condizioni infinite volte. Sono usati anche per sfruttare sistemi di analisi delle regole che lavorano con dei linguaggi diversi da quello nativo di Drools come Business Analysts. Si può in questo modo compilare le regole con l'ausilio di uno strumento specifico che facilita la stesura ed allo stesso modo rendere disponibili queste regole per un Motore a Regole come può essere quello di Drools.

Con l'approccio ai DSL si può concentrare l'attenzione allo sviluppo delle regole senza interessarsi dei dettagli implementativi dei fatti inseriti nella Working Memory.

I DSL non hanno nessun impatto nell'esecuzione dell'applicazione perché vengono valutati una sola volta prima della compilazione dei file di regole.

Un file *.dsl* è lo strumento necessario per implementare questo linguaggio specifico per il domino dell'applicazione. È una sorta di mappa a due entrate (es. come un dizionario). Esegue l'effettiva traduzione di una frase in linguaggio specifico in una specifica condizione o azione espressa nel linguaggio nativo di Drools.

Il listato seguente mostra la sintassi che si deve seguire per implementare un file DSL.

```
[when]This is {also} valid=Another(something=="{also}")
```

Listato 3.9. Sintassi per l'implementazione di un Domain Specific

Ovviamente nel file di regole si può usare, oltre la sintassi nativa di Drools, le convenzioni in linguaggio diverso espresse nel DSL dopo aver opportunamente associato il file *.dsl* al file di regole. Il file DSL si aggiunge al

motore a regole in fase di dichiarazione del Packge e si deve aggiungere una linea nel file *.drl* (di regole) che effettua il link tra i due file espressi.

```
PackageBuilder builder = new PackageBuilder();
builder.addPackageFromDrl( source, dsl );
```

Listato 3.10. Aggiunta di un file DSL al Package.

```
expander your-expander.dsl
```

Listato 3.11. Link nel file di regole per il file DSL.

3.2 Prova

Prova è un linguaggio di scripting in Java basato su regole ECA. È stato sviluppato per permettere l'integrazione delle informazioni e la programmazione ad agenti sul Web. Prova integra Java con le regole di derivazione e reattive e supporta lo scambio di messaggi con vari protocolli. Il motore a regole implementato in Prova usa l'algoritmo backward chaining per prendere decisioni. L'approccio di Prova alla programmazione a regole è importante per due tipologie di applicazioni: regole di derivazione per la risoluzione di problemi dichiarativi ricchi di logica e le regole reattive per specificare i comportamenti dinamici dei possibili agenti (oppure applicazioni) distribuiti.

3.2.1 Programmazione Dichiarativa

Un esempio tipico di applicazione della programmazione dichiarativa è la risoluzione del problema dell'attraversamento dei grafi. La soluzione di questo problema nella tradizionale programmazione procedurale non è assolutamente banale e richiede un notevole sforzo da parte dei programmatori per implementare una struttura dati specifica per mantenere le informazioni necessarie alla visita e all'elaborazione dei dati contenuti in un grafo.

La programmazione dichiarativa, ponendo i vincoli tipici di questo tipo di sviluppo riesce con facilità di espressione a risolvere tutta la tipologia di problemi.

Prova sfrutta la stessa tecnologia sviluppata per Prolog (linguaggio altamente logico che, in letteratura informatica, è definito come primo linguaggio per lo sviluppo di Intelligenza Artificiale).

L'esempio tipico di applicazione della programmazione dichiarativa è la risoluzione del problema della ***"stessa generazione"***.

La "stessa generazione" è il risultato dell'attraversamento dell'albero genealogico (un albero è una struttura semplificata di un grafo) per trovare i parenti della stessa generazione di una particolare persona presa in esame.

Facendo un ragionamento logico (metodologia di lavoro tipica del linguaggio Prolog e quindi anche di Prova) una persona risulta della stessa generazione di un altra se sono fratelli oppure se i loro genitori appartengono alla stessa generazione.

Esempio: il problema impone la ricerca delle persone della "stessa generazione" di Marta.

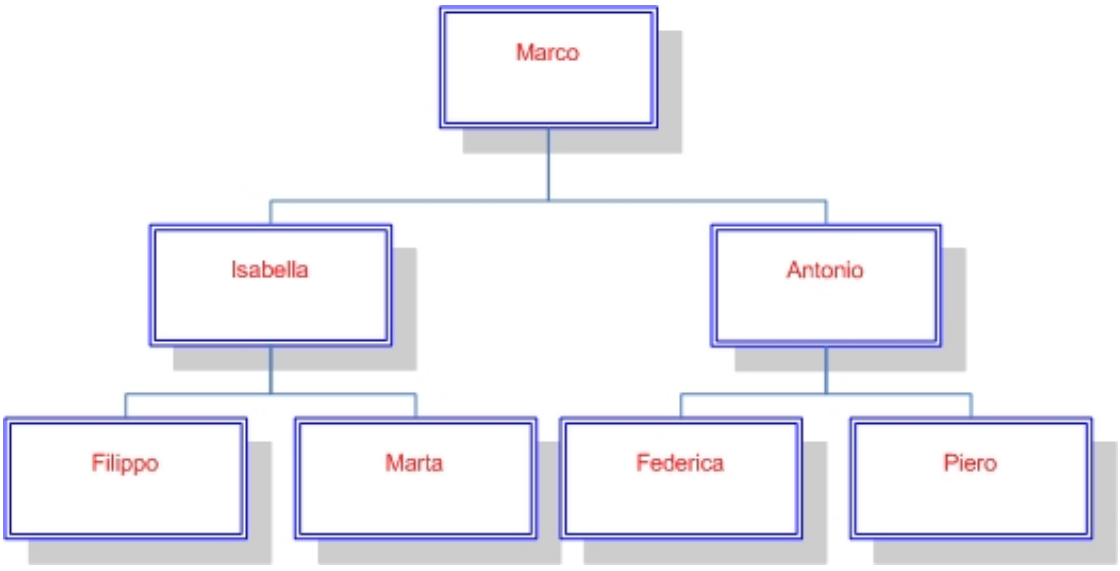

Figura 3.16. Albero Genealogico di Marta.

Per esprimere questa situazione in Prova si produce la stessa sintassi del linguaggio **Prolog** [13].

```
genitore (Marco, Isabella).
genitore (Marco, Antonio).
genitore (Isabella, Filippo).
genitore (Isabella, Marta).
genitore (Antonio, Federica).
genitore (Antonio, Piero).

sg (X,Y) :- genitore (Z,X), genitore (Z,Y).
sg (X,Y) :- genitore (Z1,X), genitore (Z2,Y), sg (Z1,Z2).
```

Listato 3.12. Esempio di programmazione dichiarativa: inserimento dati e algoritmo risolutivo.

Come si evince dal listato 3.x le prime righe di codice rappresentano la base di conoscenza del problema. Le ultime due righe sono effettivamente la ricerca delle persone con la stessa generazione. Esprimono quindi le regole logiche, quindi derivative del problema da risolvere.

Su questo codice, che risolve il problema, il programmatore può svolgere delle query in relazione alla persona di cui si vuole avere le informazioni (nel nostro caso Marta).

```
query :- solve ( sg ( Marta, X ) ).
```

Listato 3.13. Query per risolvere il problema.

L'esecuzione della query espressa nel listato 3.x porta come risultati:

- X = Filippo
- X = Federica
- X = Piero

Che risultano proprio essere i parenti di Marta della stessa generazione.

Prova permette di esprimere regole derivative anche mediante la programmazione orientata agli oggetti tipica di Java. Questo amplia notevolmente il campo di sviluppo di applicazioni.

3.2.2 Programmazione Reattiva

Le regole reattive in Prova sono espresse nella forma canonica Eventi, Condizioni e Azioni. Gli Eventi e le Azioni sono implementati con dei predicati built-in per ricevere e spedire messaggi. I messaggi sono gestiti con diversi protocolli: in special luogo si può implementare una messaggistica basata sul linguaggio per messaggi tra gli agenti **JADE** [10] oppure, se l'applicazione non prevede l'utilizzo di agenti, si può implementare il sistema di comunicazione a messaggi attraverso **JMS** (Java Messaging System) [11]. Gli eventi possono essere generati anche senza l'utilizzo di messaggi grazie alla capacità di Prova di interagire con gli eventi dei componenti **Swing,** cioè gli eventi scatenati dai componenti dell'interfaccia grafica (GUI) di Java. Data la naturale integrazione di Prova con il linguaggio Java, le regole reattive in Prova offrono una via sintattica economica e compatta di rappresentare il comportamento di agenti grazie alle innumerevoli estensioni di Java che introducono efficienza e performance riguardo alle operazioni critiche che possono essere eseguite sugli agenti. Se si sfrutta il sistema JMS si ha il notevole vantaggio che la messaggistica scambiata tra le applicazioni sia consegnata in modo corretto ed efficiente. Intuitivamente questo significa che quando l'applicazione A invia un messaggio all'applicazione B, questa non deve necessariamente essere operativa. Quando B ritornerà "online" il messaggio pendente sarà effettivamente spedito e quindi non sarà perso nella rete di comunicazione.

Prova implementa tre costrutti diversi per abilitare la comunicazione tra applicazioni o tra agenti:

- **Predicati *sendMsg***: si possono usare nel campo Azione sia delle regole reattive sia delle regole di derivazione.

- **Reaction rules**: in questo caso si ha un predicato rcvMsg bloccante nel campo Evento delle regole reattive che fa eseguire la regola dopo la ricezione dell'evento corrispondente specificato.

- **Inline reactions**: sono codificate come le precedenti solo che possono ibridare le regole di derivazione in regole reattive introducendo un predicato rcvMsg bloccante nella parte condizionale delle regole di derivazione.

3.2.3 Confronto con altri sistemi a regole

Prova è un sistema a regole basato essenzialmente su regole ECA. Questo comporta notevoli benefici per quanto riguarda l'applicabilità di questo sistema per risolvere problemi reali che sono molto spesso legati allo scatenarsi di eventi improvvisi e non previsti. Esistono, tuttavia, altri sistemi a regole reattive e si può pensare di confrontare Prova rispetto agli altri sistemi esistenti:

- **Jess**: la differenza sostanziale tra Prova e Jess è la non perfetta integrazione di Jess con le strutture di Java. Anche se sviluppato in questo linguaggio di programmazione, per eseguire l'interazione con basi di dati o lo scambio di messaggi tra applicazioni, Jess non presenta un'interfaccia semplificata per utilizzare questi sotto-sistemi Java in modo efficace ed intuitivo. Prova, dal suo canto, grazie all'introduzione dei predicati, garantisce un utilizzo trasparente e ad un più alto livello della comunicazione tra componenti in Java.

- **XChange**: la caratteristica di miglior impatto che presenta XChange è l'utilizzo di costrutti temporali espliciti per l'utilizzo degli eventi temporali. Da questo punto di vista, Prova risulta un po' carente perché nella sua struttura non prevede la reazione ad eventi temporali, fondamentali nell'implementaizone di un sistema real-time esperto. XChange, però, non è completamente caratterizzato dalla ricchezza

espressiva e dalla robustezza tipiche di Java perché non è completamente implementato sulla gerarchia ad oggetti Java considerando che XChange è stato sviluppato in seguendo le convenzioni e le caratteristiche di XML e HTML.

- **ruleCore**: il motore a regole di ruleCore non è stato sviluppato in un generico linguaggio di programmazione come Prova e Jess ma è stato implementato sulla definizione di situazioni come gli eventi generati dall'albero di decisioni che limita notevolmente il campo di utilizzo di questo motore in situazioni più generali. Anche in questo confronto, ruleCore risulta meno robusto di Prova data la sua implementazione indipendente dal linguaggio di programmazione. Questo può risultare anche un vantaggio rispetto a Prova e Jess quando l'utilizzo di un motore a regole è condizionato dalla indisponibilità di sfruttare la gerarchia Java come per esempio nel controllo di operazioni automatiche di sistemi limitati. Se si deve controllare una macchina operatrice che ha a disposizione dei micro-controllori che non permettono l'installazione di una JVM (Java Virtual Machine) tutti i motori a regole basati su Java non possono essere utilizzati e quindi si può pensare di utilizzare dei rule engine indipendenti dal linguaggio di programmazione adattandoli al firmware disponibile nel micro-controllore in esame.

3.3 Jess

Jess è un rule engine che sfrutta l'algoritmo di forward chaining per le decisioni ed esegue il pattern mathing dei fatti grazie all'algoritmo Rete. Questa definizione rispecchia esattamente la definizione del motore a regole Drools ma Jess, in aggiunta a queste funzioni che riguardano le regole di produzione, è stato implementato per reagire anche agli eventi, rendendolo di fatto un sistema a regole ECA. Jess è stato implementato, come molti di questi motori a regole, in Java, un linguaggio che offre molte potenzialità ai sistemi ma ha lo svantaggio di

richiedere la JVM installata sulle postazioni che richiedono l'utilizzo di un motore a regole. In questo modo si possono utilizzare funzioni Java nelle applicazioni che sfruttano Jess e lo stesso Jess può essere esteso utilizzando del codice Java per adattarlo a problemi specifici. Anche Jess, come Drools ed XChange, eredita da Java tutte le librerie per la gestione dell'input ed output XML rendendo l'interfaccia con questo motore molto semplificata e ben utilizzabile da tutte le applicazioni che riconoscono XML come standard di comunicazione.

Jess prevede la stesura di regole tramite una sintassi precisa che non differisce semanticamente da quella di Drools. La regola ha una parte iniziale che indica il nome mnemonico e univoco della stessa, una parte di condizioni chiamata LHS ed infine una parte di Azioni definita RHS. Le due parti, LHS e RHS, sono separate dal simbolo "=>".

```
Jess> (defrule welcome-toddlers
    "Give a special greeting to young children"
    (person {age < 3})
    =>
    (printout t "Hello, little one!" crlf))
```

Listato 3.14. Sintassi di definizione regola in Jess.

Le regole vengono valutate tramite i Fatti inseriti in una Working Memory, avente lo stesso significato di Drools, ed è vista come una sorta di base di dati dove inserire Fatti che possono essere oggetti Java.

Come in Drools, l'attivazione della regola ha un significato diverso dall'esecuzione della regola. L'attivazione si ha, infatti, al momento della modifica dei dati nella Working Memory che porta al verificarsi in contemporanea di tutte le condizioni espresse in una determinata regola. L'esecuzione della regola si ha al momento dell'invocazione sul sistema a regole del metodo **`run(),`** l'equivalente di **`fireAllRules()`** in Drools.

Nelle condizioni si possono esprimere pattern complessi, come in Drools, e si possono utilizzare anche le espressioni regolari che permettono, con pochi simbolismi, di valutare condizioni complesse specialmente legate alle condizioni su elementi testuali quali oggetti contenenti String.

Se si verificano attivazioni contemporanee sono stati implementati dei meccanismi di risoluzione dei conflitti atti a dirimere il problema delle sequenza di attivazioni da eseguire. È stato realizzato un meccanismo di **salience**, come in Drools, che permette di stabilire la priorità dell'esecuzione di attivazioni contemporanee. Se non specificato come attributo della regola, il sistema esegue le regole in base ad un numero interno assegnato ad ogni attivazione e non gestibile dagli sviluppatori.

La descrizione del sistema implementato in Jess sembra, fino a questo punto, esattamente la stessa del sistema Drools salvo la sintassi delle regole e qualche feature specifica in Jess. La potenzialità di questo sistema, invece, è ben altra: le condizioni delle regole possono essere anche degli eventi. Secondo una tradizionale descrizione di un sistema a regole di tipo ECA si ha una netta distinzione tra Eventi e Condizioni dove la più evidente è la persistenza dei dati espressi negli Eventi e nelle Condizioni. In Jess non si ha una netta distinzione tra evento e condizione ma si inseriscono tutti nella parte condizionale della regola. Per distinguere un evento da una qualsiasi altra condizione si può associare ad un elemento un listener degli eventi usando l'Event Model di Java per indicare la tipologia di evento al quale la regola deve reagire. Per creare questa associazione si definisce il listener dell'evento in una specifica funzione che sarà associata ad un elemento presente nella condizione della regola. Così facendo la regola si trasforma in regola reattiva ed è in grado di gestire gli eventi dell'interfaccia grafica Java (sia **AWT** sia la più recente **Swing**) e gli eventi scatenati all'interno del sistema a regole. In Drools si ha la possibilità nativa di gestire gli eventi generati all'interno del sistema a regole implementando opportunamente alcune interfacce Java che rappresentano il solo modello ad eventi utilizzabile mentre in Jess si possono gestire tutti gli eventi che si possono scatenare in Java ad eccezione degli eventi Temporali.

3.4 XChange

XChange è un motore a regole ancora in fase di sviluppo in ambito universitario e si propone come un ottimo e performante sistema a regole di tipo reattivo per la gestione di problematiche riguardanti lo sviluppo di applicazioni Web dinamiche.

Reattività sul Web è un problema che sta emergendo in questo periodo data l'evoluzione del Web verso una dimensione più dinamica. Il Web non è più visto come un grande "magazzino" di dati dove ogni utente può cercare; si sta sviluppando la possibilità di aumentare l'interazione tra il Web e l'utenza che ne fa uso. A questo scopo lo sviluppo di XChange mira a implementare meccanismi di reattività dei Web Server verso qualsiasi evento in entrata quali l'esecuzione di *update* dei dati dovuti alla modifica di qualche parametro in sistemi esistenti in rete. Il linguaggio XChange segue dei chiari paradigmi che hanno lo scopo di fornire il miglior linguaggio comprensibile da ognuna dei diversi tipi di eventi gestibili.

3.4.1 Paradigmi di XChange

- ***Event vs. Event Query***

Un evento (***Event***), nella gestione del Web, è una situazione accaduta in cui ogni sito web può decidere di reagire in modo particolare o di non reagire affatto. Una definizione del genere di evento risulta un po' vaga, ma l'intenzione, in questo campo, è di enfatizzare che ognuno può considerare ogni sorta di cambiamento sul Web come un evento. Comunque, ogni sistema reattivo Web-based può essere interessato in differenti tipologie di eventi o in differenti combinazioni di questi eventi (come avere un differente ordine temporale nella gestione di queste combinazioni di eventi). Quindi, questo largo spettro di possibili eventi deve essere filtrato in qualche modo per poter soddisfare le specifiche di ogni applicazione Web. Ogni nodo Web deve riuscire ad avere una notifica di un evento accaduto in un altro nodo in rete e quindi ogni evento necessita di una rappresentazione per essere distinto dagli altri.

XChange prevede una rappresentazione per gli eventi in ingresso basata su documenti XML per poter essere gestita dalle varie applicazioni grazie alla standardizzazione della sintassi XML.

Event Queries sono query eseguite sui dati di un evento. Le Event Query contengono variabili per selezionare parti di una rappresentazione di un evento. Molti strumenti atti ad implementare comportamenti reattivi non pongono nessuna differenza tra gli eventi e gli Event Queries. In XChange gli Event Queries hanno due funzioni principali: determinare l'evento d'interesse oppure una combinazione temporale degli stessi e la selezione di informazioni da estrapolare dalla rappresentazione dell'evento stesso.

- ***Volatile vs. Persistent Data***

Lo sviluppo del linguaggio XChange – progettazione ed implementazione – riflette la nuova visione dei dati presenti sul Web che si differenziano tra dati volatili (notifica di eventi sul Web tra applicazioni basate su XChange) e dati persistenti (dati delle risorse Web come le pagine XML e HTML). La netta distinzione tra le tipologie di dati presenti sul Web ha lo scopo di semplificare lo sviluppo di un Web parallelo basato su eventi.

- ***Rule-Based Language***

La reattività del sistema XChange è specificata e realizzata per mezzo di regole reattive. Oltre alle regole reattive, XChange è in grado di gestire anche regole di deduzione (chiamate regole di derivazione) per permettere un migliore sviluppo di sistemi Web "intelligenti".

Un'applicazione XChange è posta su un sito Web e contiene le regole reattive (più precisamente ECA Rules) nella forma:

Event query – Web query – Action.

Ogni evento in ingresso al sistema è filtrato attraverso uno specifico Event query che si occupa del match esatto degli eventi che influenzano il sistema (dalla

definizione precedente, l'Event query agisce solamente sui dati volatili del sistema). Se si è verificato un match esatto e contemporaneamente risultano verificate le condizioni presenti sui dati presistenti (nella parte Web query), allora la parte Action viene eseguita. A seconda della tipologia di evento e della Web query espressa si ha la determinazione di quale regola eseguire che porterà al cambiamento di qualche fatto nel sistema oppure allo scatenarsi di eventi che a loro volta potranno influenzare il sistema in questione e quelli a cui è connesso.

- ***Pattern-Based Approach***

XChange è un linguaggio basato sui pattern (molto similmente a Drools). Ogni tipologia di condizione (sia essa reattiva o su dati persistenti) e condizione, in XChange viene implementata seguendo la guida di determinati template. Questo implica una facilità di comprensione delle caratteristiche espresse nonché una migliore strutturabilità delle condizioni espresse permettendo di sviluppare applicazioni complesse con poco utilizzo di codice.

- ***Processing of Events***

Ogni evento in XChange viene elaborato nel singolo nodo Web. Questo introduce una semplificazione nella gestione degli eventi ma limita fortemente la gestione di eventi composti non dipendenti dal solo nodo in questione. Per dirimere situazioni complesse il singolo nodo valuta gli eventi composti in maniera incrementale e non in un singolo blocco come vorrebbe la sintassi della condizione. Riesce lo stesso ad essere un sistema efficace grazie all'efficienza della valutazione degli eventi singoli che, essendo processati localmente al nodo, risulta molto veloce e quindi valida per un sistema reattivo.

- ***Relationship Between Reactive and Query Languages***

XChange è stato sviluppato non solo per risolvere il problema della gestione delle regole reattive ma riesce anche a gestire un linguaggio a regole d'integrità perché è stato sviluppato partendo da una piattaforma basata su regole d'integrità (Xcerpt – molto simile ad un linguaggio per interrogazioni a database)

3.5 ILog Solver

ILog Solver è una libreria C++ sviluppata per risolvere problemi combinatori complessi in diverse aree di competenza come: pianificazione della produzione, allocazione delle risorse, time tabling, assegnamento delle radio frequenze e molti altri.

ILog Solver è basato su ILog Concert Technology. Concert Technology offre una libreria C++ di classi e di funzioni che premettono la definizione di modelli per i problemi di ottimizzazione e permette di applicare specifici algoritmi basandosi su questi modelli. Concert Technology supporta la programmazione sia basata su vincoli sia basata sulla matematica (inclusa la programmazione lineare, la programmazione quadratica e la programmazione di rete).

Una delle caratteristiche più importanti implementate in ILog Solver è la programmazione a vincoli con l'utilizzo di regole d'integrità. Utilizzando questa tipologia di regole si ha il vantaggio di poter dissociare la rappresentazione del problema, chiamata il **modello** del problema, dalla ricerca dell'algoritmo per **risolvere** il problema. Grazie all'introduzione di un *model layer,* si è in grado di modificare il modello del problema senza la necessità di dover riscrivere tutto il codice.

Questa libreria non è un linguaggio di programmazione: permette di usare le strutture dati e di controllo che sono insite in C++. Così ILog Solver risulta essere una parte dell'applicazione che può essere completamente integrata con il resto dell'applicazione per fornire soluzioni a problemi sempre più complessi.

Per trovare una soluzione ad un problema usando ILog Solver, si usa un metodo a tre passi (*three-stage method*): **describe**, **model** e **solve.**

Il primo passo *describe* è necessario per descrivere il problema in linguaggio naturale. È assolutamente necessario in quanto può capitare di dover risolvere problemi complessi che necessitano di una definizione formale in linguaggio naturale per riuscire ad esprimere tutte le condizioni ed i vincoli necessari.

Successivamente ci usano le classi di Concert Technology per modellizzare il problema (*model*). Il modello è composto da un insieme di **variabili di decisione** e di **vincoli**. Le variabili di decisione rappresentano le informazioni non conosciute a priori nel problema. Ogni variabile di decisione presenta un *domino* di *valori* possibili. I vincoli sono limiti o restrizioni sulle combinazioni di valori che le variabili possono assumere.

L'ultimo passo per trovare la soluzione del problema è l'utilizzo delle classi di ILog Solver per risolvere il problema (*solve*). Risolvere il problema, in questo campo di analisi, significa trovare i valori di tutte le variabili di decisione introdotte in modo da soddisfare in maniera simultanea i vincoli e massimizzare o minimizzare la funzione obiettivo del problema di ottimizzazione.

Il tipo di approccio alla risoluzione dei problemi è specifico di questa tipologia di motore a regole e le sue caratteristiche sono distanti da quelle degli altri sistemi mostrati. Risulta pertanto difficile provare a trovare delle caratteristiche comuni ad altri sistemi per ottenerne un paragone comunque si può valutare la capacità di questo sistema di adattarsi ai vari scenari possibili. L'implementazione in C++ permette l'utilizzo di questo sistema in un area più vasta rispetto ai sistemi basati su Java. Questo è dovuto al fatto che il compilatore C++ è molto più snello della JVM. Porta ovviamente lo svantaggio di non avere una struttura ben definita ed un sistema di gestione eccezioni che Java vanta come suo punto forte.

Il grave limite di questo sistema risulta essere la possibilità di risoluzione di problemi combinatori che non coprono la totalità dei problemi risolubili da un generico motore a regole.

Capitolo 4

Caso di Studio

Questo capitoli tratta del caso di studio considerato. Tramite l'utilizzo di un sistema a regole di produzione, JBoss Drools, si è implementato un sistema reattivo in grado di reagire ad eventi temporali, dell'interfaccia grafica e allo scambio di messaggi con altri sistemi distribuiti.

4.1 Requisiti

Come si evince dalla discussione teorica dei sistemi a regole, i sistemi a regole reattive permettono una migliore flessibilità di utilizzo rispetto a motori a regole di produzione. Questa flessibilità nasce dal fatto che i sistemi informatici moderni devono essere quasi completamente orientati all'interazione dinamica verso gli utenti e alla reazione a comportamenti di altri sistemi a loro collegati.

Data la presenza sul mercato di soli sistemi a regole di produzione (perché più facili da implementare), si è reso necessario l'implementazione di un sistema reattivo partendo da un sistema a regole di produzione già presente sul mercato, con licenza OpenSource e apprezzato da molti: JBoss Drools.

I sistemi presentati nel capitolo 3 sono per lo più reattivi ma la loro implementazione non è ancora disponibile per l'utilizzo oppure ne è disponibile una versione ancora non perfettamente stabile e testata.

Da queste considerazioni nasce l'idea di estendere un sistema di regole di produzione per riuscire ad ampliare i suoi campi di utilizzo rivolti verso le applicazioni "quasi real-time".

La soluzione ad un problema del genere non è semplice specialmente perché il sistema a regole utilizzato, JBoss Drools nella versione 4.0.7, non permette l'interazione con eventi esterni al sistema stesso. Come si è discusso nella sezione 3.1 il motore a regole permette di valutare una situazione statica della conoscenza. Questo significa che per riuscire a trovare una conclusione l'insieme di Fatti inseriti nella `WorkingMemory` è fissato, cioè si applica il metodo `fireAllRules()` una volta inseriti tutti i Fatti nella `WorkingMemory`.

Per creare una sorta di dinamicità a questo motore si devono utilizzare delle estensioni da ricercare nel linguaggio Java ed utilizzare l'Event Model di Drools per avere un aggancio dinamico tra l'applicazione che sfrutta Drools ed il motore stesso.

Gli eventi più comuni in un sistema informatico sono di tre tipologie fondamentali(come accennato nella sezione 2.1): eventi temporali, eventi legati all'interfaccia grafica dell'applicazione e scambio di messaggi tra applicazioni distribuite. Si è cercato, in questo modo, di progettare un sistema che, sfruttando le potenzialità del sistema Drools, riuscisse ad assomigliare ad un sistema reattivo partendo dal presupposto che un sistema progettato direttamente come sistema reattivo sarebbe la migliore soluzione al problema in questione.

4.2 Analisi

Dopo un accurato studio delle potenzialità espresse dal sistema a regole di produzione Drools, si è focalizzata l'attenzione sulle necessità richieste per l'implementazione di un sistema reattivo.

In primo luogo si nota subito una differenza lampante tra la sintassi ma soprattutto la semantica delle regole di produzione rispetto alle regole ECA (si vedano a riguardo le sezioni 2.1 e 2.2).

Per simulare la clausola **ON *Event*** presente nelle regole ECA si dovrebbe modificare internamente il sistema Drools perchè il parser dei file di regole non prevede una clausola del genere nella stesura delle regole. Questo però non risulta l'obiettivo di questa tesi perché, come già precisato in precedenza, si vuole fruttare un sistema già presente sul mercato (quindi già testato e con garanzie di affidabilità) per incrementare le sue possibilità di utilizzo.

Allora si è focalizzata l'attenzione verso l'oggetto `StatefulSession` che estende la più generica `WorkingMemory` nell'ambito della persistenza dei Fatti, assolutamente indispensabile per garantire il corretto funzionamento dell'applicazione reattiva.

Dopo aver generato la `StatefulSession` dalla `RuleBase`, che contiene le informazioni necessarie per l'associazione con i file di regole esterni, si possono aggiungere ad essa dei particolari ***Listener,*** presenti nell'Event Model di Drools, che permettono l'interazione degli eventi interni al sistema con l'applicazione che svolge le operazioni sul sistema stesso. Questi sono i già citati nella sotto-sezione 3.1.10 `WorkingMemoryEventListener` e `AgendaEventListener`.

Implementando in modo opportuno queste interfacce, si rende possibile la gestione dei Fatti inseriti nella `WorkingMemory` mentre vengono valutati dal motore a regole nella fase di **Runtime**.

Ovviamente la gestione delle attivazioni dell'agenda non è sufficiente per sviluppare un sistema di tipo reattivo, infatti in Drools 4.0.7 non è previsto un sistema di gestione temporale delle attivazioni delle regole.

La gestione temporale delle regole, e quindi la reazione agli eventi temporali, è un problema complesso per qualsiasi sistema informatico soprattutto per un sistema fortemente interattivo come può essere un'applicazione Web. Il concetto di tempo e le conseguenti azioni dovute alla conseguenza temporale degli accadimenti, non è di facile implementazione in un sistema fortemente statico

come può essere un sistema a regole di produzione. L'estensione possibile, dato che il motore Drools è sviluppato in Java, è sfruttare le potenzialità offerte dalla libreria Java per assecondare Drools nel reagire anche agli eventi temporali. Java permette una gestione affidabile e efficiente del concetto di tempo grazie alla gerarchia di classi legate al concetto di `Timer`.

In Java un Timer è un oggetto in grado di scatenare eventi allo scadere di un predeterminato intervallo di tempo. Questa funzionalità permette all'applicazione di non terminare la sua esecuzione al termine delle istruzioni definite, ma di persistere nell'ambiente e di gestire un evento non previsto ad ogni intervallo di tempo. Ovviamente il Timer può essere utilizzato per predisporre Fatti specifici da inserire nella `WorkingMemory` che si comporteranno da "evento" di scadenza del Timer.

Drools, essendo essenzialmente una libreria di funzionalità, non si preoccupa della gestione dell'interazione con l'utente ma prevede dei meccanismi di esecuzione di determinate azioni in base allo stato corrente della `WorkingMemory`. Per implementare correttamente la reazione ad eventi legati all'interfaccia grafica (GUI), è necessario far elaborare lo stato corrente della `WorkingMemory` del motore ad ogni evento scatenato dall'interfaccia grafica. Anche in questo caso ci viene in aiuto Java ed in particolar modo la gestione delle interfacce grafiche presenti nel package `javax.swing.*` o `java.awt.*`.

La presenza di un preciso ed efficace **Event Model** che gestisce l'interfaccia grafica Java, permette l'utilizzo dello stesso per modificare in passi successivi la conoscenza presente nella `WorkingMemory` rendendo così il sistema molto agile nell'inseguimento dei comportamenti che l'utente applica al sistema stesso.

Ultimo punto da analizzare resta la reazione del sistema a regole a messaggi. Con il termine messaggio non si deve pensare a solo un sistema che manda per esempio e-mail ad un altro. Un messaggio, formalmente testuale, può contenere codici specifici utilizzabili dal sistema ricevente per modificare il suo stato oppure per eseguire un determinato processo. Messaggio implica quindi una gestione di applicazioni non necessariamente residenti sullo stesso sistema e quindi

distribuite. Lo scambio di messaggi è forse lo strumento più importante per lo sviluppo di sistemi dinamici. Come tratta l'introduzione, i sistemi informatici sono sempre di più rivolti alla interazione remota dei dati. Per sviluppare un sistema basato su regole di produzione in grado di gestire i messaggi è necessario avere il supporto di sistemi in grado di creare canali comunicativi tra diversi nodi di una rete informatica e per fare ciò è presente in Java un'Application Programming Interface (API) in grado di creare, inviare, ricevere e leggere dei messaggi: **JMS** (Java Message Service) [11]. Questo sotto-sistema offre un servizio affidabile e flessibile per la gestione di messaggistica asincrona tra sistemi distribuiti offrendo un Event Model appropriato alla gestione di questi eventi.

Anche per risolvere questa problematica, quindi, si è ricorso all'utilizzo di sistemi già disponibili sul mercato che offrono elevate garanzie di affidabilità, riuso e robustezza del codice.

4.3 Progettazione e Implementazione

Per la progettazione di un sistema del genere si possono ben isolare i tre macro-problemi evidenziati nell'analisi. A questo scopo si è prodotto un insieme di strumenti per implementare un sistema reattivo in base alle effettive esigenze che il problema da risolvere richiede.

L'ossatura generale del sistema è ovviamente l'interfaccia verso il motore a regole di produzione offerto da Drools. Come specificato nella sezione 3.1 si necessita di una serie di oggetti generali per l'impostazione del motore che ne permettono il corretto funzionamento. Per quanto riguarda il file di regole si utilizza il file tipico di Drools con estensione *.drl:* in questo modo si utilizza in modo automatico il parser presente in Drools che elabora direttamente il file di regole e lo mette a disposizione della fase di Runtime del motore per mezzo della `RuleBase`.

4.3.1 Implementazione degli eventi temporali

Per la realizzazione di un sistema in grado di reagire ad eventi temporali si è utilizzato, come già accennato in precedenza, dell'oggetto `Timer` presente nella libreria standard di Java. Ci sono due possibili risvolti nell'utilizzo di questo oggetto: l'utilizzo degli eventi temporali in un sistema che dispone di una GUI oppure l'utilizzo degli eventi temporali senza l'utilizzo dell'interfaccia utente.

Se si sviluppa un sistema senza interfaccia utente, Java permette l'utilizzo del package `java.util` che contiene la classe `Timer` e l'interfaccia `TimerTask`. L'intefaccia `TimerTask` è necessaria perché la classe `Timer`, tramite il metodo `schedule()` necessita l'utilizzo di un task specifico in cui si andranno a notificare gli eventi riguardanti la scadenza dell'intervallo di tempo associato al `Timer` stesso. Quindi si deve eseguire una corretta implementazione dell'interfaccia mediante l'override del metodo `run()` che rappresenta il metodo eseguito ad ogni scadenza dell'intervallo del `Timer`.

In base alla tipologia di problema da risolvere, sarà necessario effettuare tutte le modifiche necessarie alla `WorkingMemory` del motore a regole nel metodo `run()`; ed eseguire il metodo `fireAllRules()` che premette la rivalutazione dell'intera conoscenza del sistema ad ogni iterazione del metodo. In questo modo il sistema è in grado di rispondere, tramite l'esecuzione di opportune regole, a comportamenti ripetuti nel tempo.

Per esemplificare una situazione del genere si è implementato un ipotetico Task Scheduler. Questa applicazione è in grado di gestire l'esecuzione di processi contemporanei di un ipotetico sistema operativo, consentendo di inserire nuovi Task e lasciando al motore a regole la decisione dell'istante in cui mettere in esecuzione questi Task ed eliminarli una volta che hanno completato la loro evoluzione. Questo esempio sfrutta appieno le caratteristiche del motore a regole e degli eventi temporali. Per la gestione del tempo si è pensato all'introduzione nella `WorkingMemory` di un Fatto che mantenesse le informazioni di un `Clock`. Questo `Clock` ha il compito di interfacciare il motore a regole con gli eventi temporali esterni scatenati da un `Timer` parametrizzabile. Ad ogni scadenza dell'intervallo del `Timer` si presuppone il completamento di un ciclo di

Clock ed in base al tempo di esecuzione di ogni singolo Task il motore a regole prende la decisione di eliminare un Task dal sistema solo dopo la terminazione dello stesso.

```
@Override
      public void run() {
            ck.setSlot(ck.getSlot() + 1 );
            ck.setActive( true );
            session.update( factHandleCk, ck );
            session.fireAllRules( agendaFilter );

            ck.setActive( false );
            session.update(factHandleCk, ck );
            session.fireAllRules( agendaFilter );
      }
```

Listato 4.1. Meccanismo di gestione degli eventi temporali .

Il listato 4.1 evidenzia l'override del metodo `run()` presente nella classe che gestisce gli eventi temporali: ad ogni scadenza del `Timer` viene incrementato lo slot che rappresenta l'effettivo istante di `Clock` del sistema. La modifica si deve ripercuotere sul sistema a regole mediante il preventivo inserimento dell'oggetto `Clock` e la successiva modifica mediante l'`update`. Ovviamente il sistema a regole necessita di una rivalutazione globale del suo stato per mezzo dell'esecuzione del metodo `fireAllRules()` dopo ogni aggiornamento del `Clock`. Si può notare che, per rendere effettivamente istantanea la notifica al sistema dell'evento temporale accaduto, è necessario compiere la modifica all'oggetto `Clock` per poi riportarlo nel suo stato originale (cioè non attivo) perché il sistema a regole valuta i dati persistenti nella `WorkingMemory`. Essendo il `Clock` a tutti gli effetti un dato persistente, deve rispecchiare fedelmente lo stato del `Timer`. Infatti il `Clock` è attivo solo nell'istante in cui notifica l'evento per tutti gli altri istanti di tempo deve rimanere in stato non attivo per non scatenare le regole che valutano opportunamente lo stato corrente del `Clock`.

Sono presenti e necessarie specifiche regole che si preoccupano della perfetta gestione dell'avvenuto mutamento.

```
rule "Clock Activation"
      salience 10
      when
            $ck : Clock( active == true );
            $th : MyThread(  );
      then
            $th.setElapsedTime ($th.getElapsedTime() + 1);
end
```

Listato 4.2. Regola che intercetta l'eventuale attivazione del Clock .

La regola descritta nel listato 4.2 mette in evidenza una delle regole di gestione degli eventi temporali, in special modo la regola che si preoccupa della gestione dell'istante di attivazione del `Clock`. La regola viene eseguita solamente quando riconosce nella `WokingMemory` un oggetto di tipo `Clock` che risulta attivo e per ogni `Thread` presente nell'ipotetico sistema operativo incrementa il tempo di esecuzione trascorso.

Il controllo presente sull'attivazione del `Clock` impedisce al sistema a regole di incrementare il tempo di esecuzione trascorso dei processi quando non è il `Clock` a stabilirlo.

Quando un processo ha finito la sua esecuzione lo segnala al sistema tramite la modifica del suo stato ponendo un flag di segnalazione opportuno al valore **`true`**. Nel listato 4.3 la regola presentata serve ad eliminare dal sistema gli eventuali processi già terminati. In questo modo l'applicazione funziona correttamente anche con un numero molto elevato di processi perché alla loro terminazione provvede a toglierli dalla valutazione che compie il sistema a regole.

```
rule "Kill Process"
      when
            $ck : Clock ( active == true );
            $th : MyThread (  );
            eval ( $th.isEnded() == true );
      then
            System.out.println("The " + $th + " is killed at = "
                  + $ck.getSlot() + " Slot Time");
            retract ( $th );
end
```

Listato 4.3. Regola che controlla la terminazione di un processo.

Se si deve creare, invece, un'applicazione che presenta una GUI, allora si può utilizzare l'oggetto `Timer` già presente nel package `java.swing`. questo `Timer` è solo utilizzabile con la presenza di un'interfaccia grafica e non richiede alcun Task aggiuntivo perché sfrutta la gestione degli eventi già presenti nelle interfacce grafiche Java. Presenta un metodo `start()` per decidere l'istante di partenza del `Timer` e ad esso si può associare un `ActionListener` apposito per la gestione degli eventi scatenati. La gestione dell'`ActionListener` è molto simile alla gestione del metodo `run()` presente in `java.util.Timer`.

4.3.2 Implementazione degli eventi legati all'interfaccia grafica

Per quanto riguarda l'implementazione degli eventi dell'interfaccia utente, si tratta di una semplice estensione del concetto espresso in precedenza. Java mette a disposizione una ricca gerarchia di eventi scatenati dall'interfaccia grafica è si sfrutta il meccanismo di segnalazione di Java per compiere le modifiche necessarie alla `WorkingMemory`. Bisogna sempre avere l'accortezza di invocare il metodo `fireAllRule()` di Drools alla fine di ogni modifica: in questo modo si riesce a gestire tramite il motore a regole i cambiamenti di stato della finestra di visualizzazione.

Per semplificare questa situazione si è creato un interfaccia grafica di un minimale negozio di abiti. Per mezzo di una finestra di selezione, il venditore può

segnalare al motore a regole l'acquisto dei vari prodotti da parte di un ipotetico cliente. Il sistema, ricevendo la segnalazione di questi eventi, modifica la `WorkingMemory` inserendo gli opportuni Fatti che segnalano l'acquisto della merce. Successivamente, il motore a regole, valuta la possibilità di eseguire uno sconto al cliente in base al prezzo degli acquisti effettuati grazie all'implementazione di opportune regole in grado di gestire l'insieme delle transazioni effettuate dal singolo cliente.

Figura 4.1. Esempio di applicazione con GUI.

L'applicazione è stata sviluppata seguendo il design pattern Model-View-Controller in modo da isolare il problema della visualizzazione dal problema della gestione dei dati. Ovviamente il Controller dell'applicazione, che segue il design pattern Singleton, definisce tutti gli oggetti per la gestione del motore a regole Drools (mostrate nella sezione 3.1) oltre che a gestire la parte visuale (View) ed i dati (Model).

Per sviluppare la reazione del sistema a regole in base agli eventi della GUI si è sfruttato il modello ad eventi presente in Java. La parte View dell'applicazione si preoccupa della notifica degli eventi dell'interfaccia grafica che vengono gestiti dal Controller che effettuerà le modifiche opportune alla base della conoscenza che informerà l'utente dell'avvenuta transazione.

```
JButton buyHat = new JButton();
buyHat.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent arg0) {
            try{
                  Controller.getController().requestBuyHat();
            }
            catch(Exception e){}
      }
});
```

Listato 4.4. Associazione di un Listener di Java all'evento di pressione del bottone `buyHat`.

```
public void requestBuyHat() {
      if( state != MAIN_STATE ){
            System.err.println("STATE ERROR");
            System.exit(-1);
      }
      session.insert( new Purchase(client, hatProduct) );
      session.fireAllRules();
}
```

Listato 4.5. Procedura di gestione dell'evento di acquisto di un cappello.

I listati 4.4 e 4.5 esemplificano una possibile interazione da parte dell'utente con il sistema. Alla pressione del bottone buyHat la parte View dell'applicazione notifica l'evento al Controller che lo gestisce inserendo nalla WorkingMemory del sistema a regole un acquisto associato al cliente corrente e al tipo specifico di prodotto acquistato. Il sistema elabora queste nuovi Fatti durante l'esecuzione del metodo fireAllRules() che tramite opportune regole è in grado di stabilire se il cliente ha il diritto ad un eventuale sconto sugli acquisti fatti e ne determina l'importo. La regola che calcola questa situazione è illustrata nel listato 4.6.

```
rule "Apply 10% discount"
      no-loop true
      dialect "java"
      when
            $c : Customer ( )
            $total : Double(doubleValue  > 100) from accumulate
                  (Purchase (customer == $c, $price :
                        product.price ), sum( $price ) )
      then
            $c.setDiscount( $total / 10 );
            insertLogical( new Discount($c, ( $total / 10 )) );
            System.out.println( "Customer" + $c.getName() +
                  " has spent " + $total + "$" );
end

rule "Notify discount"
      when
            $c : Customer ( )
            $d : Discount( customer == $c )
      then
            System.out.println( "Customer " + $c.name +
                  " now has a discount of " + $s.sum  + "$");
end
```

Listato 4.6. Regole per identificare lo sconto e notificarlo al cliente.

Si può notare che l'aggiunta dello sconto da attribuire al cliente viene eseguita tramite l'utilizzo del metodo insertLogical che permette l'inserimento di un Fatto nella WorkingMemory in modo logico. Questo significa che il Fatto viene automaticamente ritirato dalla WorkingMemory appena non esistono più attivazioni in Agenda che prevedono il suo utilizzo. Così facendo lo sconto è inserito ogni qual volta si verificano le condizioni presenti nella regola ma è eliminato se lo stato del sistema prevede l'attivazione di regole che interagiscono con l'oggetto in questione.

4.3.3 Implementazione della gestione di messaggi

Per l'implementazione dei un sistema reattivo ai messaggi si è sviluppato un'infrastruttura legata al sistema **JMS**. Con JMS si è in grado di creare applicazioni che comunicano tramite messaggi. Si crea un canale comunicativo mediante una coda che viene posta in comunicazione agli applicativi che devono comunicare. In questa coda un processo può scrivere e leggere i messaggi provenienti dall'altro lato della comunicazione. Ci sono due metodologie di invio e ricezione dei messaggi: sincrona e asincrona. La comunicazione sincrona impegna i processi ad una comunicazione real-time cioè il destinatario resta in attesa del messaggio del mittente e non procede con l'esecuzione. La comunicazione asincrona prevede un principio di comunicazione simile a quello dei messaggi e-mail: il mittente invia il messaggio alla coda, JMS provvede alla garanzia di ricezione asincrona e il destinatario, a sua discrezione, può scegliere il momento più opportuno per ricevere il messaggio.

La seconda strategia è molto usata nell'ambito di applicazioni distribuite basate sul Web perché non è possibile determinare se tutti i nodi destinatari sono disponibili alla ricezione al momento dell'invio del messaggio. Questa situazione non impegna neanche il mittente del messaggio in quanto, dopo aver spedito il messaggio, può compiere altre azioni senza attendere un riscontro dalla comunicazione realizzata.

Lo scenario così descritto può essere facilmente utilizzato per creare applicazioni basate su regole di produzione distribuite: ogni nodo di comunicazione avrà la sua base della conoscenza (implementata in Drools con la `WorkingMemory`) e ogni modifica di un nodo che influenza anche gli altri presenti in rete verrà notificata tramite messaggi. In questo modo un sistema a regole di produzione può essere in grado di gestire gli eventi generati da un altro sistema.

L'applicazione realizzata permette di inviare notifiche attraverso messaggi che consentono al sistema ricevente di inserire nella `WorkingMemory` locale un determinato cliente che verrà gestito dal sistema a regole per mezzo di messaggi in formato standard che il mittente invia in precedenza. Nei listati 4.7 e 4.8 si

evidenziano i passi fondamentali per il setup di una connessione in JMS e l'invio di semplici messaggi testuali.

```
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false,
               Session.AUTO_ACKNOWLEDGE);
queueSender = queueSession.createSender(queue);
message = queueSession.createTextMessage();
queueSender.send(message);
. . .
message.setText("+INS+ Customer:" + customer.getID());
queueSender.send(message);
```

Listato 4.7. Procedura di setup di un processo Sender ed invio di un messaggio per l'inserimento di un cliente dalla WorkingMemory di un altro applicativo.

```
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false,
                    Session.AUTO_ACKNOWLEDGE);
queueReceiver = queueSession.createReceiver(queue);
queueConnection.start();
while (true) {
      Message m = queueReceiver.receive(1);
      if (m != null) {
            if (m instanceof TextMessage) {
                  message = (TextMessage) m;
                  switch ( message.getText().substring(0, 15) ){
                        case "+INS+ Customer:" :
                              Customer c = searchCustomer(
                              message.getText().substring(16));
                              session.insert( c );
                              break;
                        . . .
            } else { break; }
      }
}
```

Listato 4.8. Procedura di setup di un processo Receiver, ricezione messaggi e inserimento di un nuovo cliente nella WorkingMemory.

Ovviamente i sistemi `Sender` e `Receiver` possono essere migliorati per la gestione di particolari servizi legati alla messaggistica che influenzano i rispettivi sistemi a regole.

4.4 Testing e Manutenzione

Dopo aver sviluppato l'ossatura generale del sistema si è reso necessario pianificare una serie di test per la verifica della funzionalità delle specifiche introdotte. Il Test del sistema ha seguito un modello, denominato in Ingegneria del Software, white-box testing ossia: ad ogni incremento di funzionalità del sistema si è verificata la sintassi e l'effettiva correttezza del codice. Ha richiesto una valutazione generale del sistema per verificare la compatibilità tra i singoli elementi sviluppati.

Infine per la verifica della correttezza del sistema si sono introdotti un numero considerevole di Fatti valutando l'effettiva funzionalità del sistema in condizioni di carico. Dato che lo sviluppo del sistema ha seguito le regole imposte dall'Ingegneria del Software, il sistema risulta efficiente ed in grado di gestire anche eventi non previsti, limitandosi alla segnalazione dell'anomalia verso lo sviluppatore ma svolgendo correttamente le operazioni necessarie al completamento delle operazioni in corso.

Conclusioni

La gestione informatica di un ambiente reale è un processo molto complesso che non può essere amministrato in modo esatto. Si devono obbligatoriamente tralasciare alcuni aspetti che in alcuni casi possono portare al non prefetto funzionamento della simulazione. L'intelligenza artificiale ha il compito fondamentale di limitare il gap che esiste tra il ragionamento umano e le deduzioni che può eseguire il computer. Per la gestione di problemi complessi si deve procedere con sistemi appositi che semplificano alcuni aspetti di mantenimento delle informazioni. I sistemi a regole offrono numerosi vantaggi rispetto a sistemi tradizionali procedurali o Object Oriented. In special modo offrono la possibilità di scindere il problema in due grossi blocchi da trattare separatamente senza incorrere nel rischio di affrontare algoritmi contorti e poco lineari: la parte logica e la parte di gestione dei dati.

Il controllo di un sistema reale implica la creazione di un sistema reattivo, cioè capace di reagire ad eventi non previsti gestendoli in modo corretto. A questo scopo la tesi tratta in modo dettagliato la gestione degli eventi in un sistema a regole, ponendo al centro dell'attenzione gli eventi che possono accadere più frequentemente in un sistema informatico: eventi temporali, eventi legati all'interfaccia grafica dell'applicazione e gli eventi generati dallo scambio di messaggi tra applicazioni distribuite.

In commercio sono disponibili sistemi a regole non reattivi, che sfruttano i dati persistenti nel sistema per dedurre una soluzione al problema. A motivo di ciò occorre sviluppare un sistema reattivo a regole partendo da un sistema a regole di produzione. Il sistema a regole di produzione scelto è JBoss Drools che offre già

un minimale modello ad eventi per la gestione del cambiamento di stato interno del sistema.

Gli eventi temporali hanno un ruolo fondamentale nella gestione di un qualunque sistema per risolvere i problemi di consequenzialità di alcune operazioni atomiche (per es. transazioni bancarie); si necessita di un forte concetto di tempo per dare senso all'obiettivo del processo. Questo presuppone una gestione esterna al sistema a regole di produzione perché non ne è prevista una gestione interna. Si è utilizzato quindi il modello ad eventi presente in Java per estendere le funzionalità di questo sistema a regole aggiornando, opportunamente secondo le notifiche degli eventi temporali, alcuni dati persistenti del sistema per permettere al motore a regole di produzione di valutare in modo statico cambiamenti di stato che sono essenzialmente dinamici.

L'applicazione implementata simula la gestione di un Task Scheduler. Esiste quindi il concetto di Clock che è simulato nel sistema con un opportuno oggetto che viene aggiornato allo scadere di un Timer. Dalla fase di test di questa applicazione si è dedotto che il sistema a regole gestisce in modo corretto ed efficace il problema portando a terminazione l'esecuzione di ipotetici Task presenti nel sistema.

JBoss Drools sta per essere sviluppato in questo senso, cioè nelle future versioni di questo motore saranno implementate delle particolari condizioni nella stesura delle regole per la gestione del concetto di tempo.

Un altro ambito fondamentale di un sistema dinamico è la gestione degli eventi legati all'interfaccia grafica del sistema. Le applicazioni, nella loro esecuzione, non possono prescindere dall'interazione con l'utente. Le interfacce grafiche permettono un modo semplice di interazione con l'utente ed obbligano gli sviluppatori alla gestione di un vasto campo di eventi generati da questa interazione. Per estendere il funzionamento di un sistema a regole di produzione alla gestione di questi eventi, si può modificare la conoscenza del sistema utilizzando il modello ad eventi implementato in Java che permette di segnalare al sistema l'avvenuta interazione con l'utente. Un opportuno utilizzo di questo modello permette di implementare nel sistema di produzione la reazione a questi

eventi modificando la conoscenza del sistema e costruendo regole apposite per rispondere a questi accadimenti.

Infine si è trattato del problema della gestione dei messaggi. Un'applicazione necessita quasi obbligatoriamente dell'interazione con altre applicazioni per la risoluzione di problemi molto complessi. Ne sono esempi le applicazioni Web che per natura devono stabilire una comunicazione con altri sistemi per il loro corretto funzionamento. Un sistema a regole di produzione è un sistema isolato in modo nativo. Questo presuppone una gestione esterna della messaggistica verso altri sistemi riuscendo successivamente ad utilizzare il sistema a regole per produrre delle conclusioni dai messaggi in ingresso. Per lo sviluppo si è utilizzato un servizio implementato in Java: Java Message Service (JMS). È un servizio che premette di inviare e ricevere messaggi in modo asincrono. Questi messaggi sono gestiti per modificare la conoscenza del sistema a regole e quindi si riesce a produrre modifiche di stato del sistema locale oppure il sistema locale può decidere di inviare notifiche ad altri sistemi per modificare la loro conoscenza.

Il sistema implementato permette di controllare l'invio e la ricezione dei messaggi permettendo la modifica dello stato del sistema a regole fornendo un interfaccia base di comunicazione che può essere ampliata per implementare servizi specifici.

Seguendo le linee guida dell'Ingegneria del Software si riesce in questo modo ad ottenere sistemi in grado di essere aggiornati con facilità e di incrementare le loro funzioni senza stravolgere l'ossatura del sistema.

La riusabilità del codice è implementata mantenendo separati i modelli dei dati dai modelli di logica dell'applicazione, riuscendo a modificare la struttura dei dati e della logica senza influenzare minimamente il corretto funzionamento del sistema.

È possibile estendere le funzionalità del sistema: dato lo sviluppo di strumenti del tutto generici si possono implementare diverse situazioni particolari partendo dalle linee guida discusse in questa tesi.

Bibliografia

[1]P. Yeh, B. Porter, K. Barker, *Mining Transformation Rules for Semantic Matching*, Fonte: http://www.cs.utexas.edu.

[2]AA VV., *XChange: A rule-based language for programming reactive behavior on the Web,* Fonte: http://reactiveweb.org/xchange/intro.html.

[3]M.Proctor, *Drools*, Fonte: http://www.jboss.org/drools/.

[4]M.Proctor, M. Neale, M. Frandsen, S. Griffith Jr., E. Tirelli, F. Meyer, K. Verlaenen, *Drools Documentation*, Fonte: http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html/index.html.

[5]G. Wagner, S. Lukichev, *Visual Rules Modeling*, Brandenburg University of Technology at Cottbus.

[6]U. Chirico, *Schemi di rappresentazione della conoscenza*, Fonte: http://www.ugosweb.com/Documents/Articles/Rap-Con.html

[7]AA. VV., *Rule-based Policy Specification: State of the Art and Future Work*, Dipartimento di Scienze Fisiche, Università di Napoli.

[8]I. Romanenko, *Use Case for Reactivity on the Web: Using ECA Rules for Business Process Modeling*, Intitut für Informatik der Ludwig-Maximilian-Universität München.

[9]A. Paschke, *ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics*, RuleML Reaction Rules Technical Goup, Nov 2005.

[10]TiLab , *JADE*, Fonte: http://jade.tilab.com/.

[11]Sun Microsystems, *Java Message Service (JMS)*, Fonte: http://java.sun.com/products/jms/.

[12]Object Management Group, *OCL (Object Constraint Language) version 2.0*, Fonte: http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

[13] HCS Lab, *SWI-Prolog*, Fonte: http://www.swi-prolog.org/.

[14]B. Berstel, P. Bonnard, F. Bry, M. Eckert, P. Patranjan, *Reactive Rules on the Web*, Fonte: http://www.pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2007-8/PMS-FB-2007-8-reactive-rules-on-the-web.pdf