Progetto "Casa intelligente per una longevità attiva ed indipendente dell'anziano"
DGR 1464, 7/11/2011

Ambient-**A**ware **LI**fe**S**tyle tutor, **A**iming at a **BET**ter **H**ealth

(Tutoraggio dello stile di vita basato sulla intelligenza ambientale, per una salute migliore)

Risultato D4.2

Metodologia di analisi dati

Rev. 1.0, 25/08/2014

REGIONE MARCHE

INRCA Istituto di Ricovero e Cura a Carattere Scientifico
HEALTH AND SCIENCE ON AGING

Il progetto AALISABETH [0] affronta i temi della prevenzione e monitoraggio di alcune delle patologie a maggiore incidenza sulla popolazione anziana (65+) [1]. All'interno del sistema domotico atto a favorire il benessere della persona anziana, è presente una rete altamente eterogenea di dispositivi [2], tra cui si possono individuare:

- sensori ambientali;
- sensori di parametri clinici e fisiologici;
- sensori personali indossabili.

Risulta evidente come l'ingente quantitativo di dati presenti una natura piuttosto eterogenea. Il progetto fonda, infatti, le proprie radici in diverse aree tecnologiche: da una parte la domotica, la cui crescente diffusione risponde ad esigenze di sicurezza, comfort, intrattenimento e che recentemente ha trovato importanti applicazioni sul tema del risparmio energetico; dall'altra la telemedicina, sviluppata e diffusa per l'erogazione di servizi sanitari (monitoraggio, in prevalenza) distribuiti sul territorio, allo scopo di abilitare meccanismi di cura domiciliari, con potenziale beneficio sia del paziente che del servizio sanitario. A causa dell'eterogeneità delle informazioni provenienti dagli oggetti intelligenti e del complesso ambiente in cui essi vengono istallati, l'insieme dei dati così raccolti necessita, dunque, di una visione e classificazione globale. Pertanto, per monitorare i comportamenti dell'utente all'interno della propria abitazione, si è sviluppata una nuova metodologia di analisi basata sull'uso di una specifica ontologia di sistema, necessaria a rappresentare formalmente i concetti e le relazioni rilevanti per il dominio [3]. Essa facilita l'interoperabilità semantica e le costruzione della inferenze, abilitando l'uso di strumenti, detti reasoner, che consentono di esplicitare la conoscenza implicita. In generale, la costruzione di un'ontologia consiste nella descrizione formale di un'interpretazione condivisa di uno specifico dominio di conoscenza [6]. Occorre cioè prendere in considerazione il sistema globale dell'Ambient Assisted Living (AAL), definendo delle macro categorie, dette propriamente classi, al cui interno si possono definire tutti gli elementi che entrano a far parte dell'ambiente.

La metodologia proposta combina un'ontologia di dominio e un motore Complex Event Process (CEP) [3]. In particolare, l'ontologia, scritta in OWL, presenta una struttura piramidale formata principalmente da componenti, una statica e una dinamica. A causa delle limitazioni del linguaggio OWL, cioè la mancanza di un reasoning temporale, è stato introdotto nel framework un motore CEP.

STRUMENTI E TECNOLOGIE UTILIZZATE

Editor di Ontologie: *Protégé*

Protégé [7] è una piattaforma open-source, basata sul linguaggio di programmazione Java. Offre un ambiente di sviluppo per il progetto di applicazioni e prototipi basati sulla gestione di conoscenze tramite ontologie.

Al suo interno *Protégé* implementa tutto un insieme di strutture per la modellazione delle conoscenze, operazioni per la creazione, visualizzazione e manipolazione di ontologie, che possono essere esportate in vari formati, inclusi RDFS, OWL, e XML-Schema. La piattaforma è estendibile (esistono numerose API e plug-in) e dispone di numerosi ambienti plug-and-play che consentono un rapido sviluppo delle applicazioni.

L'editor *Protégé*-OWL consente di costruire ontologie per il Semantic Web [10], secondo il linguaggio OWL [Ontology Web Language]. Un'ontologia OWL può includere descrizioni di classi, di proprietà e le loro istanze.

L'editor permette di:

- Caricare e salvare ontologie OWL e RDF;
- Modificare e visualizzare classi, proprietà e regole SWRL;
- Definire caratteristiche logiche delle classi come espressioni OEL;
- Eseguire ragionatori quali i classificatori logici;
- Modificare individui OWL per i markup del Semantic Web.

Per il presente progetto l'ontologia che stiamo sviluppando con *Protégé* è stata denominata **OntoAALISABETH**.

SemanticWeb Rule Language

SemanticWeb Rule Language (SWRL) [8] è un linguaggio definito specificatamente per la scrittura di regole di inferenza ed è stato ottenuto dalla combinazione di OWL-DL con il Rule Markup Language. SWRL estende il modello teorico di OWL aggiungendo la logica relativa alle clausole di Horn e rendendo possibile l'applicazione di regole di deduzione ed inferenza sui modelli ontologici.

I vantaggi del linguaggio SWRL sono quelli di avere la possibilità di dichiarare variabili, di esprimere proprietà su più di due oggetti e la possibilità di combinare proprietà e classi.

Ad esempio con SWRL sarà possibile scrivere una semplice regola per asserire che la combinazione delle proprietà hasParent e hasBrother implicano la relazione hasUncle.

Ogni regola scritta con SWRL si presenta nella forma di implicazione tra un antecedente (body) ed un conseguente (head) e corrisponde all'affermazione che se le condizioni specificate nell'antecedente valgono (hold), allora anche quelle specificate nel conseguente devono valere (must also hold).

L'antecedente contiene uno o più predicati unari o binari in congiunzione: quando tutti questi predicati sono verificati allora è possibile dedurre il predicato che si trova alla destra del simbolo di implicazione.

$$\texttt{hasParent(?x1,?x2)} \wedge \texttt{hasBrother(?x2,?x3)} \Rightarrow \texttt{hasUncle(?x1,?x3)}$$

Sia antecedente che conseguente consistono di zero o più atomi (i più piccoli componenti dell'elemento) in congiunzione tra di loro, ciascuno dei quali farà riferimento a classi o proprietà del modello OWL cui la regola si riferisce. Antecedenti e conseguenti vuoti sono rispettivamente interpretati come trivialmente vero e trivialmente falso.

Oltre ai predicati è possibile utilizzare nella parte sinistra delle regole SWRL alcuni built-in, simili nella forma a normali predicati, ma che permettono di effettuare operazioni sui dati: per esempio, è possibile confrontare date oppure eseguire operazioni su valori numerici.

Un reasoner, vale a dire un sistema software che implementa servizi di ragionamento che sia in grado di interpretare le regole SWRL, può quindi usarle non solo per dedurre il conseguente, ma anche per dedurre la negazione dell'antecedente se il conseguente è negato.

Linguaggio D2RQ

Il linguaggio D2RQ [5] è un linguaggio dichiarativo usato per mappare elementi relazionali (**tabelle**, **colonne**, **id**, **join** e così via) su elementi ontologici (**classi**, **relazioni**, **URI**, tipi **XSD**, etc.). Tale mappatura è espressa usando i termini presenti nel D2RQ namespace:

```
http://www.wiwiss.fu-berlin.de/suhl/bizer/D2RQ/0.1#
```

I termini di questo namespace sono formalmente definiti nello schema D2RQ RDF [4].

In generale, la mappatura definisce un virtual RDF graph che contiene informazioni dal database. E' simile al concetto delle viste in SQL, a parte il fatto che si tratta di un grafo virtuale piuttosto che una tabella relazionale virtuale.

Connessione database

All'interno del mapping file possono essere definite più sorgenti relazionali da mappare. Questo perché' ogni connessione a DB è di fatto una istanza di **d2rq:Database** e quindi è possibile definire molteplici istanze di database e riferirle singolarmente all'interno del file per contestualizzare i mapping.

La lista completa dei parametri di configurazione per la connessione si trova online all'indirizzo riportato nel riferimento [mi0]. Di seguito riportiamo solo i parametri principali:

- **d2rq:jdbcDSN** specifica l'URL di connessione al DB (ad esempio, nel caso di MySQL per esempio potrebbe essere qualcosa del tipo "jdbc:mysql://localhost/iswc");
- **d2rq:jdbcDriver** specifica la classe del driver JDBC (ad esempio, per MySQL è "com.mysql.jdbc.Driver");
- **d2rq:username** specifica il nome utente per la connessione (se richiesto);
- **d2rq:password** specifica la password di connessione (se richiesta).

Mapping

Il mapping effettivo consiste essenzialmente nella definizione di due oggetti: **Class Map** e **PropertyBridge** associati.

Un **d2rq:ClassMap** è di fatto l'oggetto che specifica il mapping rispetto ad un certo **d2rq:Database** associando tutti gli identificativi delle istanze a una particolare classe del modello ontologico. A ogni **ClassMap** possono essere associati più **PropertyBridge**. Per una spiegazione approfondita di tutti i parametri di ClassMap si faccia riferimento alla documentazione ufficiale [5].

ClassMap permette solo di definire la modalità di generazione (condizionale o non) degli identificatori delle istanze e delle associazioni rispetto alle classi di modello. Per valorizzare poi le proprietà delle istanze create rispetto alle proprietà/relazioni del modello è necessario associare al ClassMap delle PropertyBridge. In pratica, un **PropertyBridge** è un elemento del mapping che dichiara come valorizzare una certa proprietà di modello (sia **owl:DatatypeProperty** che **owl:ObjectProperty**) per le istanze di una certa classe (associata al**ClassMap**) e in funzione di regole condizionali, join ed eventuali espressioni SQL per i mapping più complessi. Per un approfondimento sui parametri usabili per definire un PropertyBridge si veda la documentazione relativa [5].

Framework JENA

Jena [9] è un framework, ovvero una struttura di supporto allo sviluppo di software, per la costruzione di applicazioni nell'ambito del Semantic Web. Esso è uno strumento open-source, sviluppato da Brian McBride della ditta Hewlett-Packard, e include:

- API per RDF;
- Possibilità di leggere e scrivere RDF in RDF/XML, N3 e N-Triple;
- API per OWL;
- Possibilità di archiviazione in memoria o persistente;
- Un motore di interrogazioni SPARQL.

Jena è scritto in Java, e pur essendovi degli strumenti a riga di comando che permettono la realizzazione di alcuni compiti fondamentali, il suo principale utilizzo è come librerie Java. Attualmente il progetto Jena risiede su sourceforge.

Le ontologie possono essere descritte usando vari linguaggi: dall'OWL Full, il più espressivo, passando per OWL DL, OWL Lite e per finire con RDFS, il meno espressivo. Grazie alla sua *Ontology API*, Jena fornisce una interfaccia di programmazione consistente per sviluppare applicazioni che usano ontologie, in modo indipendente dal linguaggio per ontologie usato: i nomi delle classi Java non fanno cenno al linguaggio sottostante.

Per rappresentare le differenze tra i vari linguaggi che si possono usare per definire l'ontologia, ciascuno dei linguaggi ha un profilo, che elenca i costrutti permessi e i nomi delle classi e proprietà. Il profilo è legato a un modello di ontologia che è una versione estesa della classe Model di Jena.

È interessante notare che Jena non trasforma la struttura dell'ontologia in una struttura di classi. Tutte le informazioni sono mantenute da OntModel come statement RDF. Ogni volta che un'informazione viene ricercata nell'ontologia, si cerca di desumerla analizzando gli statement presenti. Di contro, ogni volta che nuove informazioni vengono inserite, si aggiungono statement RDF. Se all'ontologia è agganciato un reasoner, nuovi statement vengono automaticamente creati ed aggiunti al modello a partire da deduzioni effettuate sugli statement dichiarati. Gli statement dedotti vengono poi utilizzati nella fase di ricerca delle informazioni alla stregua di statement dichiarati.

Un modello di ontologia è una estensione di un modello RDF che fornisce ulteriori capacità di manipolazione di ontologie. I modelli di ontologie sono creati usando la ModelFactory di Jena. Il modo più semplice per creare un modello di ontologia è il seguente:

```
OntModel m = ModelFactory.createOntologyModel();
```

Questa istruzione crea un modello di ontologia con le impostazioni di default, che sono:

- Linguaggio OWL Full;
- Archiviazione in memoria;
- Inferenza RDFS (che produce implicazioni dalle gerarchie di subclass e subproperty).

In generale però le impostazioni di default, soprattutto per quanto riguarda il reasoner, possono essere troppo forti, in altri casi troppo deboli: è bene per creare un `OntModel` decidere un particolare reasoner o profilo di linguaggio. È possibile invocare il metodo `createOntologyModel(OntModelSpec s)` dove il parametro di tipo `OntModelSpec` incapsula la descrizione dei componenti di un modello di ontologia, incluso lo schema di memorizzazione, il reasoner e il profilo del linguaggio.

La classe OntModelSpec permette un controllo completo sulle scelte di configurazione del modello, fornendo un certo numero di costanti che rappresentano le più comuni combinazioni. È possibile creare una OntModelSpecification personalizzata, partendo dal suo costruttore e invocando i metodi per impostare correttamente le proprietà desiderate.

Il listato seguente mostra come si è creato il modello di ontologia con Jena, a partire dall'ontologia del progetto AALISABETH:

```
String owlPath = "resource/OntoAALISABETH.owl";
```

```
OntModel model =
ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM);
model.read(owlPath, null);
```

Reasoner (Pellet)

Pellet [11] è un OWL-DL reasoner con un supporto estensivo per inferenza su individui, datatypes definiti dall'utente, e supporto per il debugging dell'ontologia. Il W3C definisce due tipi di document checker OWL: OWL sintax checker e OWL consistency checker. Pellet è il primo e attualmente unico sistema consistency checker OWL–DL completo e, a livello sintattico, ha una quasi copertura totale dei costrutti OWL; inoltre è anche un syntax checker.

Un OWL consistency checker prende un documento come input e produce come output una risposta relativa alla consistenza o meno del documento, oppure non è in grado di rispondere. Tuttavia pur essendo il consistency checking un importante task, questo non ci permette di fare importanti deduzioni sull'ontologia. Infatti è obbligo per un reasoner mettere a disposizione un insieme standard di servizi di inferenza:

- **Consistency checking:** assicura che l'ontologia non contenga fatti inconsistenti. Nella terminologia DL questa è l'operazione che controlla la consistenza **ABox** rispetto alla **TBox** [
- Tabella 1];
- **Concept Satisfiability:** controlla se per una classe è possibile contenere una qualche istanza. Se la classe è insoddisfacibile allora definire un'istanza di quella classe comporterà un'inconsistenza dell'intera ontologia;
- **Classification:** calcola la tassonomia fra classi, cioè la completa relazione di sottoclassi fra classi;
- **Realization:** trova la *most specific class* a cui un individuo appartiene.

Abbr.	Nome completo	Significato
TBox	Terminological Box	Componente che introduce la terminologia, ovvero il vocabolario del dominio di applicazione. La TBox è un insieme di frasi, assiomi e definizioni che descrivono concetti.
ABox	Assertional Box	Componente che contiene asserzioni su individui. Associa i ruoli e i concetti descritti dalla TBox a istanze (individui) specifiche. Dipende fortemente da TBox
KB	Knowledge Base	La combinazione di ABox e TBox, ovvero l'ontologia OWL completa

Tabella 1 Significato di TBox e ABox nella terminologia DL

I servizi sopra elencati possono essere invocati in Pellet attraverso delle API, come la DIG- interface. A differenza di altri reasoner PELLET è progettato esclusivamente per lavorare con OWL fin dall'inizio, questa scelta di progetto ha un'enorme influenza sull'architettura del sistema. Il principale obiettivo progettuale di Pellet è stato quello di realizzare un modulo core di reasoning suscettibile ad estensione.

La Figura 1 indica l'architettura completa del sistema.

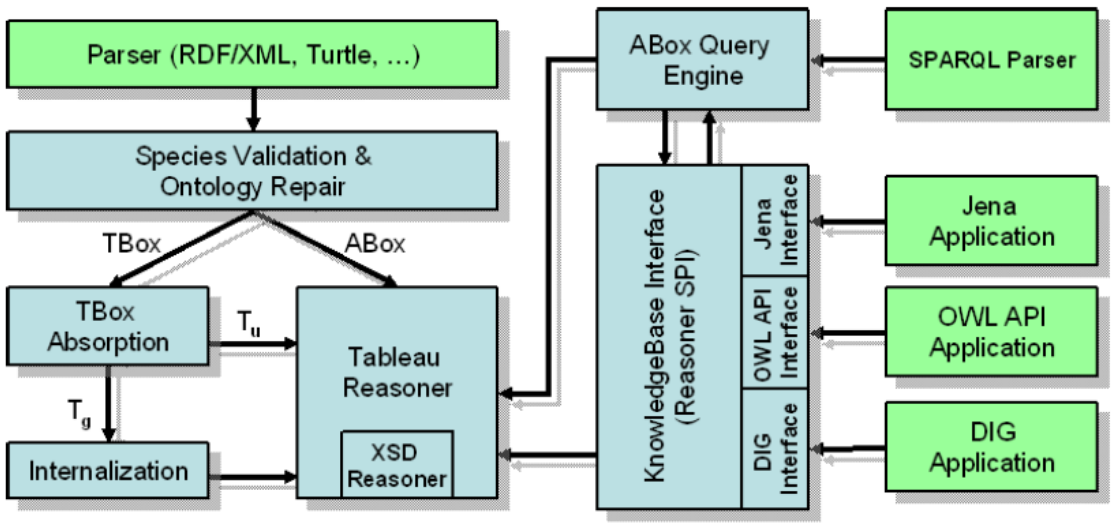


Figura 1: Architettura del ragionatore Pellet

Di seguito descriviamo con più dettaglio i moduli che compongono il sistema.

PARSING AND LOADING: Pellet fornisce differenti interfacce per caricare ontologie; infatti di per sé non ha un parser RDF/OWL ma esso è integrato in differenti toolkits che lo mettono a disposizione. Per tale motivo il ragionatore include differenti sottomoduli che consentono di caricare ontologia a partire da differenti rappresentazioni.

TABLEAUX REASONER: In Pellet il ragionatore ha un'unica funzionalità: fare il check della consistenza di un'ontologia, dove un ontologia è consistente se c'è un'interpretazione che soddisfa tutti i fatti e gli assiomi in essa. Tale interpretazione è chiamata modello. Il tableaux reasoner cerca un modello tramite il processo di completion che prevede la costruzione di un grafo prelevando l'informazione dall'ABox, dove intuitivamente i nodi sono individui e letterali. A loro volta gli archi diretti rappresentano asserzioni, proprietà-valore. Il ragionatore applica ripetutamente le regole di espansione finché non viene trovata una contraddizione oppure il grafo non presenta contraddizioni ma non è applicabile più alcuna regola. Tutti gli altri task d'inferenza possono essere ricondotti al problema del consistency checking. Ad esempio per verificare che un individuo è istanza di un concetto o meno si asserisce che l'individuo è istanza del complemento di quella classe e poi se ne verifica l'inconsistenza applicando il consistency checking.

Per fornire l'estendibilità del sistema, anche il tableaux reasoner è costruito in modo modulare. Infatti l'algoritmo di completion è progettato in modo tale che possono essere implementate differenti strategie di completion con differenti euristiche basate sulle caratteristiche della base di conoscenza data.

DATATYPE REASONER: Esso è responsabile del controllo della consistenza dell'intersezione di tipi di dato. I tipi di dato in OWL sono descritti utilizzando XML Schema che fornisce un ricco insieme di semplici datatypes includendone di numerici, stringhe e tipi date/time. Un'intersezione di tipi di dato è inconsistente quando questi non hanno nessun valore in comune. Ad esempio l'intersezione di

XSD:INTEROPOSITIVO e XSD:INTERONEGATIVO è vuota è quindi non consistente. Anche in questo caso la verifica della consistenza dell'intersezione è verificata tramite la costruzione del grafo di completion.

KNOWLEDGE BASE INTERFACE: Tutti i task di ragionamento possono essere ricondotti alla consistenza della base di conoscenza con un appropriata trasformazione. Tuttavia questa riduzione non sempre è banale e nella maggior parte dei casi piuttosto costosa. Proprio per tale motivo Pellet fornisce una System Progamming Interface (SPI) per nascondere all'utilizzatore i dettagli della trasformazione. Questa Knowledge base interface è costruita sulla libreria Aterm (Annotated Term).

ABOX QUERY ENGINE: La knowledge base interface è strettamente connessa ad un ABox query engine che permette di rispondere a query congiuntive. Questo modulo supporta queries scritte in SPARQL (Simple Protocol and RDF Query Language).

La decisione di descrivere l'architettura di tale DL reasoner deriva dal fatto che abbiamo deciso di utilizzarlo in AALISABETH. Infatti Pellet, pur mostrando un'efficienza minore rispetto a Racer per quanto riguarda il TBox reasoning (Racer è un altro ragionatore compatibile con *Protégé*), permette di ottenere delle performance migliori nel caso di ontologie con un numero elevato d'istanze.

Java programming assistant

Java programming assistant (Javassist) è una libreria Java che permette di manipolare direttamente il "Java bitcode" che è il codice che viene usato dal motore Java per l'esecuzione dell'applicativo [12]. Quindi, Javassist permette di modificare o aggiungere classi a runtime. In particolare, Javassist può essere usato per:

- Specificare il bytecode usando codice sorgente;
- Introdurre nuovi metodi in una classe, questo approccio può essere definito "Aspect-Orientd programming" (AOP);
- Sfruttare la "Reflection" a runtime;
- Invocare un metodo remoto, ad esempio di un oggetto in esecuzione su un Web Server.

Con il concetto di "Reflection" si intende l'abilità di un programma di esaminare e modificare la struttura ed il comportamento del programma durante l'esecuzione.

I metodi di introspezione forniti dalla Java Reflection ci permettono di invocare classi e metodi di cui al momento dello sviluppo del codice non sono conosciuti, che hanno cioè un aspetto dinamico in relazione a dei parametri di input.

ESPER

Esper [13] è un componente per il Complex Event Processing (CEP), disponibile sia per Java che per .NET (NEsper), dotato di capacità di Event Stream Processing, Figura 2, il quale permette un rapido sviluppo di applicazioni che elaborano grandi volumi di eventi in real-time, o semi real-time.

È sostenuto da EsperTech con un modello di doppia licenza sia open-source che professionale.

Il motore di Esper funziona come un database al contrario. Invece di memorizzare dati ed eseguire interrogazioni su di essi, memorizza le *query* e le esegue sul flusso di dati. In questo modo le risposte

avvengono in tempo reale, in quanto le interrogazioni vengono eseguite continuamente invece che solo in fase di registrazione.

Una delle caratteristiche peculiari di Esper è la facilità nel formulare le interrogazioni tramite un linguaggio dichiarativo chiamato EPL (Event Processing Language), che è molto simile al linguaggio SQL. L'EPL differisce dall'SQL in quanto utilizza il concetto di *vista (sliding window)* piuttosto che quello di *tabella*. Tali viste consentono al motore di effettuare le varie operazioni necessarie per strutturare e ricavare i dati in un flusso di eventi. Una volta espresse (in EPL), le interrogazioni dovranno essere registrate nel motore (a *runtime*), in modo che Esper possa verificare i risultati delle interrogazioni sugli eventi in ingresso ed, eventualmente, inviarli in tempo reale ad uno o più moduli sottoscriventi (*listener*). Tale modo di operare consente, qualora fosse necessaria, una modifica abbastanza semplice e veloce delle interrogazioni già registrare e permette di riutilizzare gli eventi.

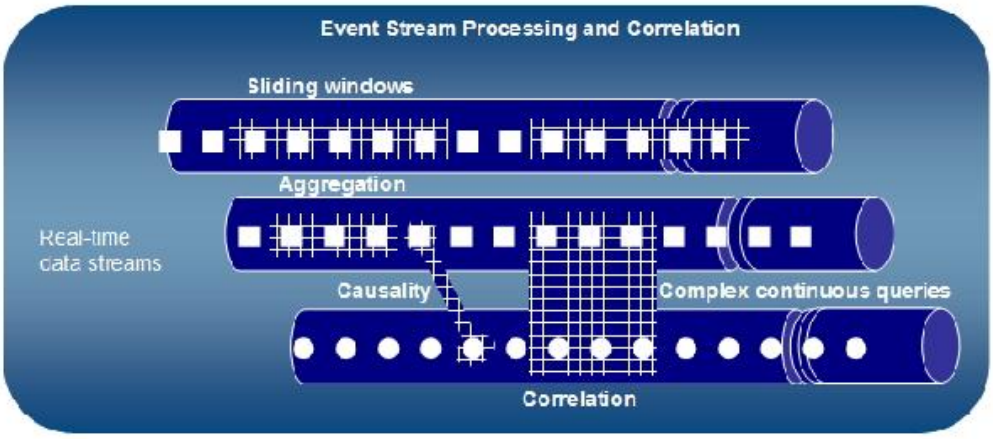


Figura 2: Event Stream Processing

Esper, inoltre, fornisce due metodi principali per processare gli eventi, e sono:

- Event pattern;
- Event stream query.

Il primo si basa su un linguaggio che permette di specificare dei pattern expression-based per il matching degli eventi. Questo metodo di elaborazione si aspetta sequenze di eventi o una loro combinazione, e consente la loro correlazione basata sul tempo.

Il secondo metodo invece, offre la possibilità di definire *query*, secondo la sintassi EPL, che consentono operazioni di *filtering*, *aggregation*, *correlation* (attraverso gli operatori di join) e inoltre mettono a disposizione funzioni di analisi per i flussi di eventi.

Le altre caratteristiche principali sono elencate di seguito:

- Interrogazioni, filtraggio e computazione del risultato in maniera continua;
- Operazioni di Join continue;
- Finestre temporali basate su tempo o numero di tuple;
- Logica Followed by e ricerca di eventi mancanti;
- Join continui di stream e dati storici memorizzati in database relazionali tradizionali, effettuando cache locale;
- Rappresentazione degli eventi attraverso oggetti Java (POJO), .Net, Map o XML.

In Figura 3 è mostrata una rappresentazione schematica dell'architettura di Esper.

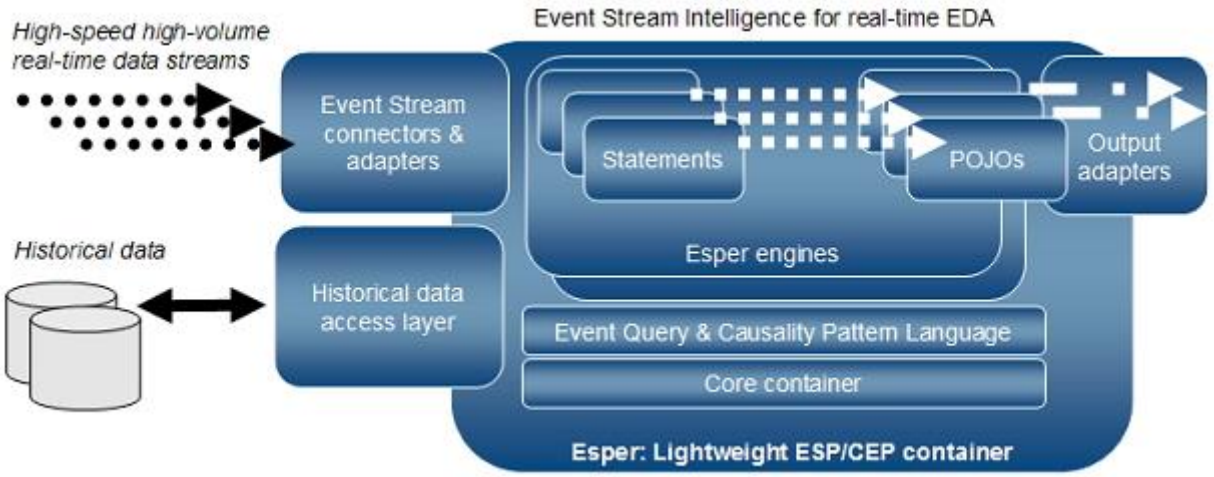


Figura 3: Architettura del motore Esper

In particolare, per quanto riguarda il progetto AALISABETH, la capacità di Esper di specificare dei pattern expression-based per il matching degli eventi e l'analisi del loro flusso, ci consente di rilevare gli scenari comportamentali di sospetto diagnostico identificati, nella fase precedente, dal gruppo coordinato dal Dott. Vespasiani.

METODOLOGIA

Architettura del framework

Motivazioni

La metodologia proposta nasce dalla necessità di gestire una grande quantità di dati eterogenei, memorizzati in un opportuno database. In particolare, sono ben noti sia l'obiettivo finale della analisi che una conoscenza dettagliata dell'intero sistema e dei records acquisiti. In un contesto eterogeneo e ampio, non è sufficiente considerare solo il valore dei singoli dati: è necessario tenere presente il loro significato all'interno del contesto considerato.

Per tener conto delle relazioni e formalizzare la conoscenza del dominio, risulta essere necessaria l'implementazione di un'ontologia. L'approccio generale è illustrato nella Figura 4. In alto, il sistema reale è composto sia conoscenze statiche che dai dati generati dal sistema stesso, i quali sono collezionati in un repository. Poiché in questo step i dati vengono memorizzati come una lista di records, essi mostrano una natura granulare causata dall'eterogeneità dei sensori che li generano e la loro semantica potrebbe essere persa. Inoltre, un'ulteriore situazione che potrebbe verificarsi è la ridondanza dei dati; cioè, ci possono essere diversi dispositivi che apparentemente producono risultati diversi, ma forniscono le stesse informazioni. Quindi, l'ontologia viene introdotta per evitare tali situazioni e per creare un ponte fra il mondo reale e una rappresentazione formale. Infatti, essa è capace di fondere le informazioni statiche e dinamiche mediante classi e istanze. Pertanto, i vantaggi di un tecnica semantica sono due. Infatti, una volta che l'ontologia ha fornito una concettualizzazione e una specifica descrizione del sistema, tali unità guiderà anche la fase di analisi.

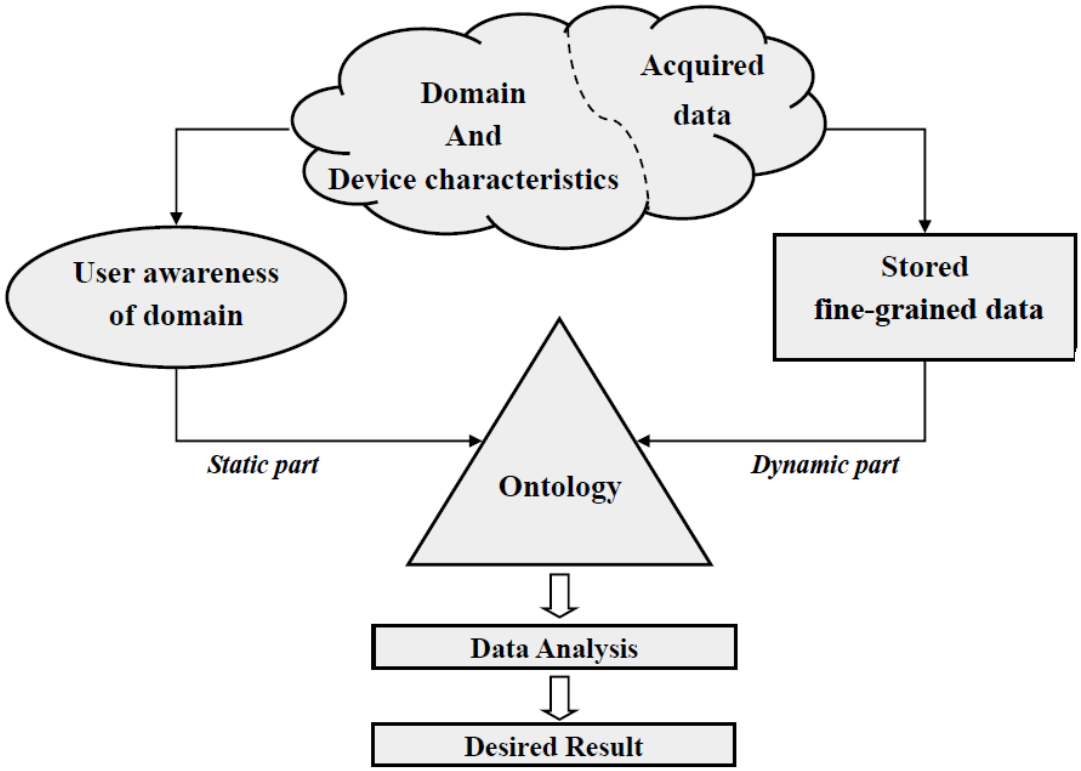


Figura 4: Approccio generale: dal mondo reale all'ontologia

Architettura

L'architettura del framework è mostrata in Figura 5. I dati collezionati nel database di tipo MySQL opportunamente costruito per il progetto [2], vengono mappati in OntoAALISABETH, l'ontologia di dominio appositamente sviluppata per il progetto AALISABETH. Inizialmente si trova una corrispondenza tra gli elementi del DB e quelli dell'ontologia. Tale ontologia è costruito seguendo una precisa struttura che verrà descritta in dettaglio in seguito. Una volta che i dati saranno riorganizzati secondo la loro semantica, l'ontologia avrà un ruolo di pre-processing dei dati. Infatti, l'utente può definire delle query semantiche in modo da estrarre uno specifico aspetto dell'intero complesso contenuto nell'ontologia.

E' interessante notare che questa particolare vista non può essere precedentemente creata a causa della natura altamente frammentata dei dati (privi di significato) contenuti nel database.

Le differenti viste posso essere considerate come l'output di sensori che non sono fisicamente presenti nel sistema, questo tipo di sensori vengono chiamati "sensori virtuali".

Nel momento in cui i vincoli temporali non sono presi in considerazione, una ontologia è sufficiente a classificare ed organizzare i dati prodotti dai sensori, fisici e virtuali. Ricordando che l'obiettivo dell'analisi è monitorare il comportamenti dell'utente è facile rendersi conto come l'output sia strettamente dipendente dal tempo, pertanto abbiamo bisogno di introdurre nel nostro framework un componente capace di gestire vincoli di tempo. Questo problema è stato risolto usando un motore di analisi ad eventi poiché capace di individuare eventi complessi all'interno di flussi di eventi semplici temporizzati.

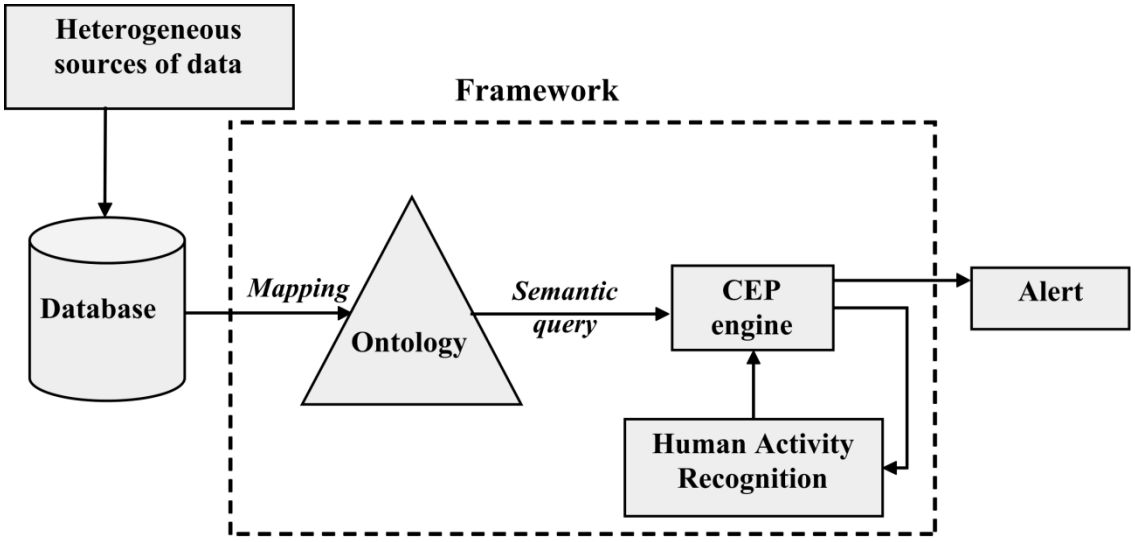


Figura 5: Struttura del framework

Struttura OntoAALISABETH

La principale componente del framework è rappresentata dall'ontologia, OntoAALISABETH, che definisce formalmente il dominio considerato generando una base di conoscenza condivisa per tutte le componenti del dominio. Tale ontologia mostra diversi livelli di astrazione che composti insieme formano una struttura piramidale. L'architettura, come mostrata in Figura 6, è formata da due componenti principali:

- Livello statico (AAL and AAL-Building Specific ontology);
- Livello Dinamico (data and view ontology).

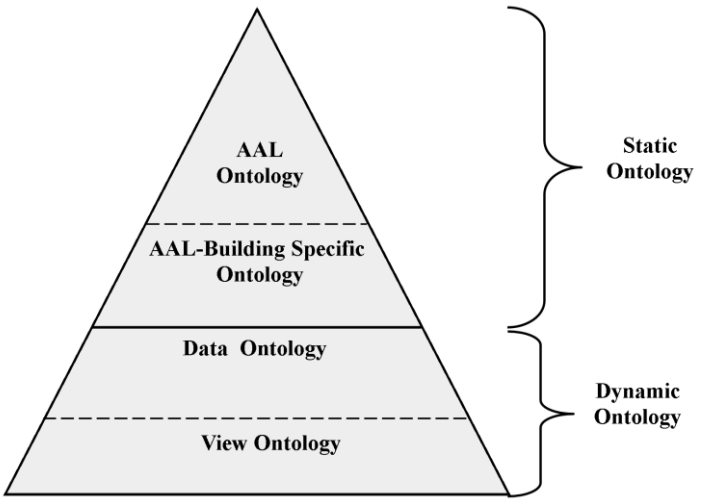


Figura 6: Struttura piramidale dell'ontologia

Ogni parte ha un ruolo specifico al fine di rispondere alle diverse esigenze, come descritto di seguito.

1. *AAL Ontology*. Il primo livello costituisce l'ontologia di dominio superiore, che contiene informazioni statiche. Infatti, vengono formalizzati i concetti generali di contesto AAL pertanto questa parte di OntoAALISABETH può essere riutilizzata in ambienti simili.

   Come suggerisce la letteratura, al fine di velocizzare lo sviluppo dell'ontologia e il tempo di realizzazione, si è scelto di ridefinire ed estendere DogOnt Ontology [8]. Tale ontologia, sviluppata in OWL, è stata implementata dal gruppo e-Lite dell'Università degli Studi di Torino, per descrivere ogni dispositivo di una Smart Home, ovvero un'abitazione dotata di sensori ed attuatori, di hardware e software dedicati, allo scopo di realizzare interazioni di automazione. Infatti, in essa per ogni dispositivo reale viene creata una descrizione dettagliata che ne esplica il funzionamento attraverso l'uso di un linguaggio comune e condiviso. In particolare, l'ontologia OntDog è in grado di descrivere:
   - dov'è è situato un dispositivo domotico;
   - la capacità e la funzionalità di un dispositivo;
   - le specifiche tecnologiche necessarie per interfacciarsi con un dispositivo;
   - le possibili configurazioni che un dispositivo può assumere;
   - come è strutturata l'abitazione (piani, stanze, muri);
   - gli elementi architetturali e di arredamento presenti nell'ambiente.

   La descrizione di ogni dispositivo è quindi composta da due componenti: una parte astratta per descrivere il comportamento del dispositivo ed una parte tecnologica necessaria per la comunicazione. Questo tipo di approccio favorisce l'interoperabilità tra i dispositivi in quanto permette uniformità nel linguaggio utilizzato per la loro descrizione. DogOnt, pur descrivendo dettagliatamente una Smart Home, non soddisfa i nostri obbiettivi, poiché non descrive il tipo di conoscenza che si vuole rappresentare. Infatti, essa non permettere di descrive gli abitanti della casa, come ad esempio il peso corporeo, l'età, la pressione cardiaca, il tasso glicemico, le relazioni familiari che intercorrono, l'indice di saturazione di ossigeno, le frequenze cardiache e respiratorie. Inoltre, attraverso essa non si riesce a descrivere quali sono le abitudini alimentari e le abitudini personali, come ad esempio con quale frequenza l'anziano monitorato effettua attività fisica e, in particolare, se essa è influenza dall'ambiente familiare e/o dalle condizioni meteorologiche. Per le precedenti ragioni, a partire da DogOnt è stata sviluppata OntoAALISABETH. Tale Ontologia eredita le principali classi da DogOnt e, in aggiunta, ne presenta delle nuove. Le classi ereditate sono:

   - ✓ **Building Thing:** per modellare qualsiasi elemento controllabile e elemento non controllabile. Si definisce elemento controllabile (o dispositivo) un elemento che può essere controllato dal sistema domotico. I dispositivi sono suddivisibili in due categorie: quelli appartenenti ad uno degli impianti domestici (la lampada, ad esempio, viene cataloga come appartenente all'impianto elettrico) e gli elettrodomestici. Ogni dispositivo possiede diverse funzionalità (Functionality, rintracciate dalle relazioni hasFunctionality) e può assumere diverse configurazioni di stato (State, identificate attraverso la relazione hasState). La quantità ed il tipo di dati e di funzionalità dipendono ovviamente del dispositivo; in generale, ogni elemento controllabile mette a disposizione almeno una Query-Functionality che permette di fornire informazioni riguardanti la configurazione. Gli elementi non controllabili possono essere di tipo architetturale per la definizione dell'ambiente (muri, pavimenti, ecc.) e di arredamento (mobili, scrivanie, tavoli, ecc);

   - ✓ **Building Enviroment**: per modellare la posizione di un elemento. Contiene una descrizione delle varie tipologie di ambiente: vengono definiti gli ambienti giardino (classe Garden), garage (classe Garage) e appartamento (classe Flat). Ogni piano può contenere una o più stanze (cucina, camera da letto, bagno, ecc.);

- ✓ **Functionality**: per modellare le operazioni che si possono effettuare su un dispositivo. Tutte le funzionalità definiscono i comandi necessari per modificare o per interrogare una specifica proprietà del dispositivo. Sono suddivise in diverse categorie in relazione al loro scopo;

- ✓ **State**: per modellare le configurazioni che un dispositivo può assumere. Come per le funzionalità, anche gli stati vengono classificati in base ai valori che possono assumere: i ContinuosState, che riguardano quelle proprietà che possono assumere valori continui, ed i DiscreteState, associati alle proprietà che assumono sono valori discreti;

- ✓ **Notification**: per modellare tutte le notifiche che i dispositivi possono generare in seguito ad una variazione di stato;

- ✓ **Command**: per modellare tutti i comandi per controllare ed interrogare i dispositivi.

Le classi aggiunte al fine di raggiungere l'obiettivo sono principalmente quelle che descrivono la persona e i suoi comportamenti:

- ✓ **Activity**: per modellare le principali attività della persona può svolgere all'interno della propria abitazione: camminare, camminare velocemente, dormire, riposare, mangiare, stare seduto, o fare altre attività;
- ✓ **ConsumableThing**: per monitorare cosa gli abitanti ingeriscono Medicine, pasti e bevande.
- ✓ **EnviromentProfile:** per modellare le caratteristiche degli abitanti della casa e le condizioni meteorologiche. Tale classe è organizzata in due sottoclassi: **Person** (che descrive le caratteristiche della persona, come età, corporatura) e **Natural**, che ha sua volta, ha due classi-figlie **Season** e **Weather**. Tali classi sono state introdotte con lo scopo di capire le abitudini della persona, ma soprattutto da cosa sono influenzate.
- ✓ **Meal**: per monitorare quando la persona mangia.

In questa parte di OntoAALISABETH sono state introdotte un insieme di regole che sono in grado di formalizzare la conoscenza a priori. Per esempio, se si vuole definire formalmente l'attività di "preparare un pasto", è necessario identificare un insieme di azioni atomiche e uno o più luoghi dove tale azione può avvenire. Questo tipo di informazioni sono ottenute dalla combinazione dei dati prodotti dai sensori e dal loro significato semantico.

2. *AAL-Building Specific ontology.* Il secondo livello, *AAL-Building Specific ontology*, estende diverse proprietà statiche introdotte nel livello precedente. In questa parte si definisce formalmente il particolare dominio che si sta considerando, ovvero vengono descritte le varie componenti della Smart Home considerata: la reale struttura dell'ambiente e la disposizione delle stanze, le informazioni personali relative agli abitanti della casa, quali sensori sono stai istallati nella rete e come cominciano tra di loro. Inoltre, la completa conoscenza del dominio permette allo sviluppatore di aggiungere nell'ontologia nuovi elementi e relazioni, le quali non possono essere descritte nelle tecnologie di memorizzazione dati. A differenza dell'AAL Ontology, questa componente di OntoAALISABETH è sviluppata per modellare uno specifico ambiente. Di conseguenza, essa cambia per ogni particolare dominio considerato e non può essere riutilizzata.

3. *Data ontolog.* Il terzo livello, DataOntology, estende le precedenti livelli di ontologia introducendo il concetto che ogni dispositivo genera dei dati granulari. Nello specifico, le precedenti classi sono popolate dagli individui che presentano una corrispondenza uno a uno con ogni records memorizzati nel data repository. Questa procedura è permessa da un mapping tra DB e ontologia, attraverso il linguaggio D2RQ. In particolare, esso permette, come descritto nel paragrafo "Linguaggio D2RQ", di stabilire una connessione fra l'ontologia e diverse sorgenti e di importare i relativi attributi e proprietà. Dunque, i dati memorizzati nel data repository vengono caricati

nell'ontologia secondo la propria semantica. Inoltre, tale acquisizione avviene in un intervallo di tempo costante, permettendo quindi di eseguire un'analisi dinamica in tempo reale. In conclusione, si ottiene i dati con la propria semantica che permette di avere nel livello successivo delle viste personalizzate del sistema.

4. *View ontology.* L'ultimo livello di OntoAALISABETH, view ontology, consiste nella creazione di viste alternative della conoscenza. Questa fase è basata sulla procedura di prepocessing, dove il reasoner gioca un ruolo principale. Infatti, esso è in grado di inferire conseguenze logiche da un insieme di fatti o di assiomi. Nel nostro caso, le regole e gli assiomi sono definiti attraverso le regole, con lo sviluppo di opportune builtins, e le classi equivalenti. Infatti, un insieme di builtins, paradigma alternativo per modellare la conoscenza, è introdotto per acquisire nuova conoscenza stabilendo nuove connessione fra entità indipendenti. Le regole, che in grado di estendere l'espressività di OWL, sono valutate periodicamente in fase di esecuzione e nuovi fatti sono aggiunti nell'ontologia. La definizione di classe equivalente è guidata dallo scopo di classificare le istanze in base a determinate proprietà e relazioni; cioè queste classi sono popolate da individui desiderati. Invece, la fase di prepocessing è basata sull'abilità del reasoner di fare query sull'ontologia ed estrarre informazioni necessarie per l'analisi. Si osserva che le query sull'ontologia nello step finale della metodologia corrispondono a selezionare un insieme di dati generati da un sensore virtuale, secondo l'interpretazione dell'utente.

Analisi di Processo

La prima componente del framework impiega le tradizionali tecniche di Semantica del Web, come query e reasoner. Tuttavia, per la gestione un insieme di dati che cambiano dinamicamente nel tempo e nello spazio, tali metodi non sono sufficienti. Pertanto, si introduce anche un'architettura basata sul Complex Event Processing.

L'elemento centrale di un'architettura basata sul Complex Event Processing è il motore, ossia l'entità che effettua sugli event object l'insieme delle operazioni necessarie ad elaborarli: creazione, lettura, trasformazione, aggregazione, correlazione, pattern detection e rimozione. Precisamente il suo ruolo è quello di interpretare, filtrare, combinare eventi primitivi ed individuare gli eventi compositi sulla base di regole specifiche, al fine di notificarne l'occorrenza ai componenti "reattivi", cioè quei componenti il cui compito è quello di attuare determinate operazioni in risposta a determinate situazioni rilevate.

Il motore riceve in ingresso tutte le informazioni necessarie ad elaborare il flusso, quali eventi dati, eventi di controllo e in generale tutti i tipi di eventi significativi nel particolare contesto preso in considerazione.

La prima operazione che effettua un motore CEP è il filtraggio, ossia la rimozione degli eventi da uno stream in base alle loro proprietà (o attributi).

Esempi di filtraggio possono essere i seguenti:

- Prendere in considerazione tutto il traffico relativo ad un determinato protocollo: SSH, http, etc;
- Considerare tutti e soli i dati forniti dai sensori i cui valori sono compresi in un range prefissato.

Attraverso tale fase, il CEP engine è in grado di effettuare una selezione degli eventi di interesse, tagliando fuori tutti quelli privi di significato e non appartenenti dunque al dominio del problema.

Un concetto molto importante su cui i CEP engine come Esper basano l'event processing è quello di *sliding window*, Figura 7. Una finestra è un oggetto che mantiene in memoria, in un certo istante di tempo, un insieme finito di eventi appartenenti ad uno stream.

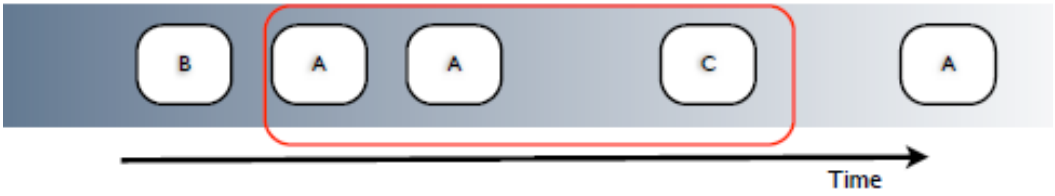


Figura 7: Sliding window

L'utilizzo delle finestre consente al motore di effettuare operazioni fondamentali per l'elaborazione degli eventi, quali ad esempio l'*aggregazione*. Questa operazione consiste nell'aggregare molteplici eventi in un singolo evento, attraverso l'applicazione, allo *stream* di eventi ricevuto in input, di funzioni aritmetiche sui loro attributi (somma, conteggio occorrenze, etc.), statistiche (valor minimo, massimo, medio, deviazione standard, etc.), estraendo in tal modo informazioni "aggregate" atte a rilevare eventuali "scenari" significativi.

L'operazione più importante per un CEP engine, oltre a quelle finora presentate, è sicuramente la *correlazione*. L'elaborazione contemporanea di flussi multipli di eventi consente al motore di correlare dati appartenenti a flussi diversi e "apparentemente" indipendenti tra loro. Riuscire a correlare informazioni semplici ed aggregate vuol dire poter rilevare situazioni significative all'interno di un determinato contesto.

Sebbene la correlazione sia fondamentale nell'individuare scenari significativi per il dominio del problema, essa non consente però facilmente di stabilire relazioni temporali e causali tra eventi. Si rende necessario, quindi, la capacità da parte di un CEP engine di legare gli eventi da un punto di vista logico, causale e temporale, al fine di poter realizzare *pattern detection*. Ecco perché Esper si affida ad un linguaggio *SQL-like* come EPL.

Questa fase può essere vista in realtà come l'applicazione di tutte le altre fasi (filtering, aggregation e correlation), e permette di effettuare un'elaborazione di più alto livello dell'insieme di tutti gli stream in ingresso al motore, riconoscendo pattern complessi nel dominio di interesse.

Esper viene fornito con il loro linguaggio "SQL-Like", chiamato Event Processing Language (EPL) che è molto simile al linguaggio SQL, come già accennato precedentemente, ma che differisce da quest'ultimo per alcuni importanti miglioramenti.

In EPL le query sono chiamate "Statements" e vengono inviate al motore di Esper come stringhe. Ogni evento appartenente ad un flusso di eventi è come una tupla (una riga della tabella). Il flusso di eventi è come la tabella. Le proprietà di un evento sono gli attributi della tupla.

EPL è un linguaggio dichiarativo facente parte della famiglia dei continuous query language, appositamente sviluppato per gestire eventi ordinati nel tempo e con un'alta frequenza di invio, Figura 8.

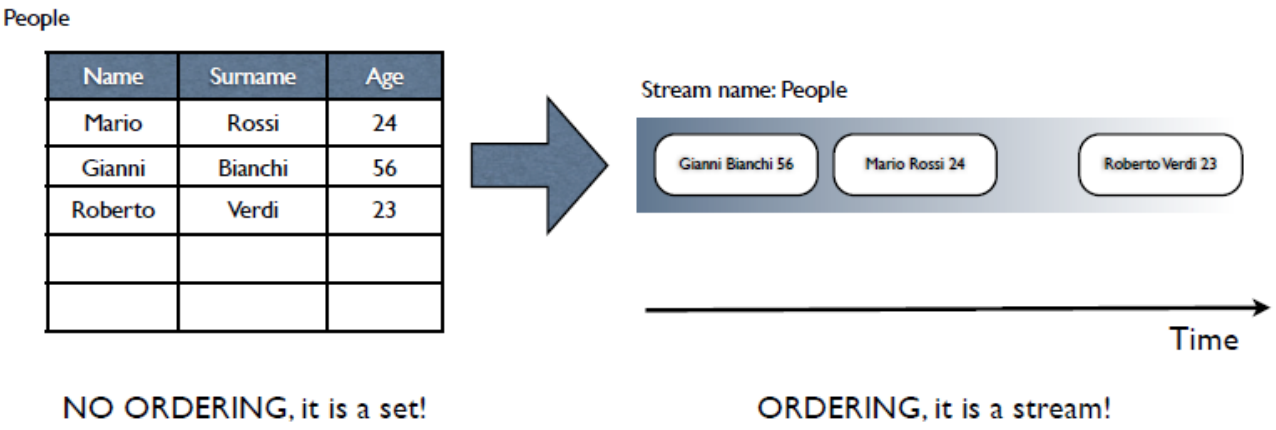


Figura 8: Eventi ordinati nel tempo nel linguaggio EPL

Senza il concetto del tempo i linguaggi SQL ed EPL sono identici. Il miglioramento in EPL è proprio quello di aver tenuto conto del concetto del tempo, permettendo modifiche a *runtime*, l'esecuzione di funzioni definite dall'utente e l'acquisizione di relazioni tra eventi.

Ad ogni query in EPL si collega un Listener (un oggetto Java semplice che contiene un metodo "update"), che verrà innescato ogniqualvolta il CEP incontrerà un messaggio con le stesse caratteristiche riportate nella query: ad esempio, il Listener per eventi relativi ad un sensore di temperatura verrà attivato tutte le volte che in ingresso si rileva un valore superiore alla soglia massima stabilita.

Quando una query (o statement) è soddisfatta, essa restituisce eventi o proprietà che possono essere utilizzati da altre istruzioni creando una rete logica di flussi.

IMPLEMENTAZIONE DEL FRAMEWORK

Il framework, sviluppato in Java per combinare diverse tecniche, ha una particolare struttura, in accordo alla metodologia descritta precedentemente. Infatti, esso può essere diviso in due parti, la prima vede coinvolta l'ontologia di dominio mentre la seconda il motore CEP. In particolare, sono state utilizzate una libreria Java che permette di modellare ontologie e le API di Esper che consentono di utilizzare il motore CEP all'interno dell'ambiente Java.

L'ontologia di dominio, **ontoAALISABETH**, è stata sviluppata e testata in *Protégé 4.3* insieme al reasoning Pellet, che permette la creazione di classi di equivalenza che saranno poi popolate con individui.

Nel nostro framework tale ontologia è caricata usando Jena framework all'interno di un modello ontologico. Questo può essere fatto sviluppando un'interfaccia OntoModel, dove ogni classe dell'ontologia è definita come una OntoClass di Java.

Inoltre, all'interno del modello ontologico sono state introdotte delle rules per aumentare l'espressività dell'ontologia. Esse sono state inserite attraverso un file di testo (.TXT) per integrare la conoscenza che non può essere introdotta usando l'OWL standard. In particolare, la libreria Jena fornisce per implementare le rules un insieme di builtins (funzioni), che non risultano essere sufficienti per i nostri obiettivi in quanto non permette di gestire i tempi e le date. Per questo motivo, è stata sviluppato un insieme di Built-ins per la gestione di date e tempi, come descritto nell'Allegato A, la quale permette di implementare rules volte a calcolare la differenza di due date, il minimo o il massimo di date. In dettaglio, l'input della differenza di date può essere calcolato o tra due date generiche o tra una data generica e la data corrente. Invece, il massimo o il minimo di date può essere calcolato solo tra due date, ma usando le query SPARQL si è in grado di estrarre il minimo o il massimo valore di una classe. Combinando queste regole, si è in grado di valutare i vincoli temporali. La nuova conoscenza descritta dalle rules è introdotta nell'ontologia attraverso il reasoner. In Jena sono state implementati alcuni reasoners con la possibilità di estensione per integrare le Builtins personalizzate.

Successivamente, si stabilisce una connessione fra le classi dell'ontologia e il database, scritto in MySQL. In particolare, un file .ttl, sfruttando il linguaggio D2RQ, è stato opportunamente implementato affinché si stabilisca una connessione e i dati presenti nel database vengono caricati nelle classi dell'ontologia in accordo alla propria semantica.

In seguito, il reasoner Pellet è invocato all'interno di Jena per fare ragionamenti sull'ontologia contenente i dati. In particolare, le query SPARQL sono invocate all'interno di Jena per selezionare gruppi di dati per essere poi analizzati da Esper. Esso non supporta intrinsecamente un accesso all'ontologia OWL, così è necessario mappare gli eventi dell'ontologia all'interno di oggetti Java (POJOs).

Importazione classi dell'ontologia in ambiante Java (creazione Pojo) e caricamento istanze

La struttura ed i dati contenuti nell'ontologia per essere immessi nel motore di analisi di eventi, Esper, devono essere convertiti in strutture dati Java. Infatti per poter sfruttare le funzionalità fornite da Esper i dati devono avere una struttura definita secondo lo standard Plain Old Java Object (POJO), ossia classi contenenti solo metodi relativi alla gestione degli attributi.

Il processo di conversione può essere diviso in due fasi principali:

- Importazione delle classi dell'ontologia in classi Java;
- Creazione delle istanze.

Durante la prima fase, sfruttando le librerie di Jena precedentemente descritte, è possibile leggere l'ontologia con tutte le sue classi. Per ogni classe OWL viene creata una corrispondente classe Java completa di tutti gli attributi e metodi **getter**, per recuperare il valore dell'attributo, e **setter**, per modificarlo.

In questo momento vengono definite delle annotazioni all'interno della classe che verranno poi utilizzate per la mappatura da Individual a Istanze Java nella seconda fase.

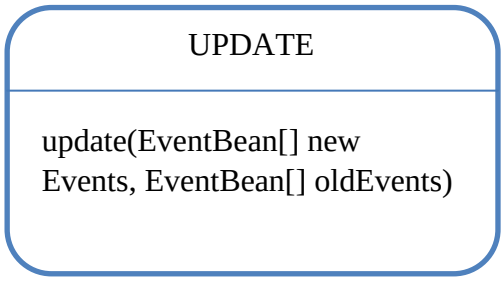
Completata questa prima parte si procede al vero e proprio caricamento dei dati, rappresentati come Individual dell'ontologia. Per ogni Individual dell'ontologia viene creato un oggetto Java inserendo tutti i relativi valori degli attributi che rappresentano il dato finale, ad esempio Timestamp e valore del sensore.

A questo punto questi oggetti Java posso essere analizzati da Esper come un flusso di eventi.

Query con Esper

Il modello di elaborazione in Esper è continuo: l'*Update Listener* è l'entità che riceve dal motore gli eventi e tutte le altre informazioni derivanti dalla elaborazione dei flussi, non appena il motore processa gli eventi per quel determinato statement sul quale il Listener è registrato.

La classe UpdateListener possiede un unico metodo astratto update(), come mostrato in figura, che necessita di essere implementato.



Il motore fornisce i risultati degli statement agli update listener inserendoli in oggetti *com.espertech.esper.client.EventBean.* Un'implementazione tipica di un *update listener* interroga le istanze **EventBean** tramite metodi **get** per ottenere i risultati generati.

Un secondo metodo per la consegna dei risultati, fortemente tipizzato, nativo e altamente performante è l'utilizzo di un *Subscriber Object*, il quale rappresenta un legame diretto dei risultati della query a un oggetto Java. L'oggetto, un POJO, riceve i risultati dello statement tramite l'invocazione di metodi. Per poter implementare un *subscriber object* non è necessario implementare ne un'interfaccia e ne estendere una classe.

Le query EPL si basano sulla seguente sintassi:

```
[annotations]
[expression_declarations]
[context context_name]
insert [istream|rstream] into event_stream_name[(property_name
[,...])]
select [istream | irstream | rstream] [distinct] * |
expression_list ...
from stream_def [.view_spec][(filter_criteria)][as name][,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

Come si può notare, le sole clausole obbligatorie sono la SELECT e la FROM, mentre le altre sono del tutto opzionali. Vediamo ora velocemente il significato delle principali.

La clausola **select** è richiesta in tutti gli statement EPL, e può essere utilizzata per selezionare tutte le proprietà tramite il carattere *, o per specificare un elenco di proprietà o espressioni.

Di seguito un esempio:

```
select symbol, price, sum(volume) as sm
from StockTick.win:time(60 sec)
```

Questo statement prevede la selezione degli attributi symbol e price degli eventi StockTick, e la somma del volume per gli eventi di questo tipo che ricadono in una finestra temporale di 60 secondi.

È importante notare che all'interno della clausola select è anche possibile definire delle espressioni, come mostrato nel seguente statement:

```
select volume * price
from StockTick.win:time_batch(30 sec)
```

nel quale è selezionato il prodotto tra il volume e il prezzo.

La clausola **from** è obbligatoria in tutti gli statement EPL, e permette di specificare uno o più flussi di eventi, ai quali opzionalmente può essere associato un nome attraverso la parola chiave as.

Ovviamente la specifica di più di un flusso ha come obbiettivo la realizzazione di uno o più join, che sono perfettamente supportati dall'EPL.

Attraverso la clausola from è anche possibile realizzare operazioni di filtraggio, il tutto semplicemente prevedendo delle espressioni tra parentesi tonde subito dopo la definizione del flusso di eventi.

```
select *
from RfidEvent(zone=1 and category=10)
```

Attraverso tale statement, sono selezionati tutti gli attributi degli eventi del tipo RfidEvent, i quali presentano l'attributo zone=1 e quello category=10.

La clausola **where** è una clausola facoltativa negli statement EPL, e consente di definire quali flussi di eventi filtrare e su quali flussi effettuare un join.

Gli operatori di confronto utilizzabili sono =, <, >, >=, <=, !=, <>, is null, is not null, ma sono anche possibili delle loro combinazioni logiche attraverso gli operatori AND e OR. Mostriamo di seguito due esempi di utilizzo di tale clausola:

```
select 'IBM stats' as title, avg(price) as avgPric, sum(price) as
sumPric
from StockTickEvent.win:length(10)
where symbol='IBM'
```

Questo effettua un filtraggio dei risultati, prevedendo come esito della query EPL i soli eventi la cui proprietà symbol è pari a IBM

```
select *
from TickEvent.std:unique(symbol) as t,
NewsEvent.std:unique(symbol)as n
where t.symbol = n.symbol
```

Questo invece effettua un join tra i due stream TickEvent e NewsEvent, attraverso la proprietà symbol.

La clausola **group by** è facoltativa in uno statement EPL, e permette di suddividere l'uscita di quest'ultimo in gruppi. È possibile raggruppare a partire da uno o più nomi di proprietà di eventi, o dal risultato di espressioni calcolate.

Ad esempio, la dichiarazione seguente restituisce il prezzo totale per simbolo per gli eventi del tipo StockTickEvent negli ultimi 30 secondi:

```
select symbol, sum (price)
from StockTickEvent.win: time (30 sec)
group by symbol
```

Esper pone le seguenti restrizioni alle espressioni sulla clausola *group by*:

- Le espressioni in group by non possono contenere funzioni di aggregazione.
- Le proprietà degli eventi che vengono utilizzati all'interno di funzioni di aggregazione, nella clausola select, non possono essere utilizzate anche nelle espressioni della group by.
- Quando si raggruppa un flusso non limitato, è necessario assicurarsi che l'espressione definita in group by non restituisca un numero illimitato di valori.

La clausola **having** permette di rifiutare o lasciar passare eventi definiti in group by in maniera molto semplice, e precisamente nello stesso modo in cui nella clausola where si definiscono le condizioni per la select, con l'eccezione che in having è possibile utilizzare funzioni di aggregazione.

Il seguente statement è un esempio di clausola having con una funzione di aggregazione. Esso mostra il prezzo totale per simbolo degli eventi StockTickEvent degli ultimi 30 secondi, ma solo per quelli il cui prezzo totale supera 1000. Quindi tutti quelli il cui prezzo totale è pari o inferiore a 1000 saranno eliminati.

```
select symbol, sum(price)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000
```

La clausola **order by** è facoltativa, e viene utilizzata per ordinare gli eventi in uscita rispetto ad alcune loro proprietà. Ad esempio, lo statement seguente restituisce gli eventi StockTickEvent degli ultimi 60 secondi, ordinati in base al prezzo e al volume:

```
select symbol
from StockTickEvent.win:time(60 sec)
order by price, volume
```

Esper pone una restrizione per tale clausola, e cioè che tutte le funzioni di aggregazione che compaiono in essa, devono apparire anche nell'espressione della select.

Anche la clausola insert into è facoltativa in Esper. Essa può essere utilizzata per rendere disponibili i risultati di uno statement EPL come un flusso di eventi, così da poter essere utilizzato in ulteriori statement. Tale clausola può anche essere usato per unire flussi di eventi multipli per formare un unico flusso di eventi.

Esper pone le seguenti restrizioni alla clausola insert into:

- Il numero di elementi nella clausola select deve corrispondere al numero di elementi nella clausola insert into, se essa specifica un elenco di nomi di proprietà degli eventi.
- Se il nome del flusso di eventi è già stato definito da uno statement precedente e i nomi delle proprietà degli eventi e/o i tipi non corrispondono, viene generata un'eccezione in fase di creazione dello statement.

Il seguente esempio inserisce in un flusso di eventi, denominato *CombinedEvent*, il risultato dello statement:

```
insert into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as
latency
from EventA.win:time(30 min) A, EventB.win:time(30 min) B
where A.txnId = B.txnId
```

Ogni evento in *CombinedEvent* ha due proprietà: "custid" e "latenza". Gli eventi generati da tale statement possono essere utilizzati in ulteriori statement, come ad esempio quello seguente:

```
select custId, sum(latency)
from CombinedEvent.win:time(30 min)
group by custId
```

Come già accennato all'inizio di questa trattazione, gli **Event Pattern** sono dei pattern expression-based per il matching degli eventi. Essi sono costituiti da due elementi fondamentali:

- Pattern atoms: sono gli elementi costitutivi dei pattern, e sono espressioni di filtraggio, osservatori per eventi temporali e plug-in per osservatori personalizzati che osservano gli eventi esterni, cioè quelli che non sono sotto il controllo del motore.
- Pattern operator: controllano il ciclo di vita delle espressioni e combinano i Pattern atom in maniera logica o temporale.

Le pattern expression possono essere innestate con profondità arbitraria includendo l'espressione nidificata in parentesi tonde. Vediamo qualche esempio.

Il pattern individua quando 3 *sensor event* indicano una temperatura maggiore di 50 gradi ininterrottamente per 90 secondi dal primo evento, considerando però gli eventi provenienti dallo stesso sensore.

```
every sample=Sample(temp>50) ->
((Sample(sensor=sample.sensor, temp>50) and not
Sample(sensor=sample.sensor,temp<=50))->
```

```
(Sample(sensor=sample.sensor,temp>50) and not
Sample(sensor=sample.sensor, temp <= 50)))where timer:within(90
seconds))
```

Un aspetto molto importante da notare è che un pattern può apparire ovunque nella clausola from di un'istruzione EPL, e possono essere utilizzati in combinazione con le clausole where, group by, having e insert into.

CONCLUSIONI

Per concludere, in questo documento è stata introdotta una nuova metodologia di analisi dati per il progetto AALISABETH. In particolare, tale metodologia combina un'ontologia di dominio, opportunamente sviluppata, e un motore di analisi di eventi complessi (CEP). L'ontologia rappresenta l'elemento centrale della metodologia presentata e si compone in quattro livelli. Il livello più alto formalizza i concetti generali del dominio Ambient Assisted Living ed è seguito da un livello che formalizza più nello specifico i sensori installati e la disposizione delle stanze in modo tale che i dati possano essere caricati in accordo alla propria semantica (terzo livello dll'ontologia). L'ultimo livello, invece, riorganizza i dati in particolari viste. Di conseguenza, l'ontologia gioca un ruolo di pre-processing poiché riorganizza i dati in base al contesto favorendo una corretta analisi. Il motore CEP, infine, è stato introdotto per riconoscere le attività umane.

Si intende proseguire con l'utilizzo del framework per identificare i comportamenti dell'utente che possono indurre a dei sospetti diagnostici, già definiti [1]. Inoltre, i risultati ottenuti saranno confrontati con quelli ottenuti da esistenti tecniche di riconoscimento delle attività umane, come Bayesian networks, Hidden Markov Models, Learning Machine.

BIBLIOGRAFIA

[0] http://www.aalisabeth.it/

[1] Progetto AALISABETH: Risultato D4.1, Data cloud derivante dal pre-processing dei dati raccolti, data repository
[2] Progetto AALISABETH: Risultato D2.2, Specifica funzionale del sistema AALISABETH

[3] R. Culmone, M. Falcioni, M. Quadrini: "An Ontology-based Framework for Semantic Data Preprocessing Aimed at Human Activity Recognition", Accepted at SEMAPRO 2014 Conference.

[4] http://d2rq.org/terms/d2rq

[5] http://d2rq.org/d2rq-language

[6] T. R. Gruber. (retrieved: March, 2014) What is an Ontology? [Online]. Available: http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[7] (retrieved: June, 2014) Protege. [Online]. Available: http://protege.stanford.edu/

[8] http://www.w3.org/Submission/SWRL/

[9] (retrieved: June, 2014) Apache jena. [Online]. Available: http://jena.sourceforge.net/

[10] http://www.w3.org/standards/semanticweb/

[11] (retrieved: June, 2014) Pellet. [Online]. Available: http://clarkparsia.com/pellet/protege/

[12] http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

[13] http://esper.codehaus.org/

Allegato A:
Libreria di built-ins in jena
Per la gestione del tempo nelle ontologie

Data di creazione: 25.08.2014
Revisione 1.0

LIBRERIA DI BUILT-INS

Le built-ins, funzioni che implementano particolari algoritmi ed invocate all'interno di rules, sono sviluppate all'interno del framework Jena per inferire nuova conoscenza. Tale framework presenta un set di built-ins già sviluppato, con il quale vengono gestiti, per la maggior parte, valori numerici, attribuiti agli individui di un'ontologia. Le built-ins, di default, non permettono, quindi, la gestione dei valori temporali. Per questo motivo, si è pensato di sviluppare una libreria Java capace di estendere le funzionalità di Jena. In questo modo è possibile inferire nuova conoscenza attraverso le rules, lavorando anche su valori temporali.

Pertanto, inizialmente sono trattati i metodi utilizzati per permettere la creazione di nuove built-ins in Jena e vengono descritte le classi Java implementate per raggiungere l'obiettivo. Nell'implementazione del codice, sono state create tante classi java quante sono le built-ins implementate. E' stata realizzata, inoltre, una classe extra che contiene appositi metodi per la gestione di operazioni sulle date.

Custom built-ins

Il framework Jena fornisce un set di built-ins predefinito che può essere esteso implementando nuove built-ins (custom built-ins). Per definire una nuova built-ins è necessario dichiarare una classe che estende BaseBuiltins, la quale implementa (implements) l'interfaccia Builtins. La classe BaseBuiltin fornisce una realizzazione per tutti i metodi astratti dell'interfaccia Builtins. Alcuni metodi della classe BaseBuiltin sono:

- **getNome**: ritorna il nome attribuito alla built-in;
- **getArgLength**: ritorna il numero aspettato di argomenti in input e in output della funzione;
- **bodyCall**: implementa il codice della built-in. Questo metodo è invocato quando la built-in è invocata nel corpo della rule;
- **headAction**: implementa il codice della built-in. Questo metodo è invocato nella testa della rule.

I metodi **bodyCall** e **headAction** ritornano un valore booleano ("true" o "false"), a seconda se il test di confronto abbia risultato positivo o negativo. Invece, l'output del metodo **getNome** è una stringa, mentre quello di **getArgLength** è un intero.

Ogni built-in creata, deve essere registrata attraverso il metodo register della classe *Builtin-Registry*, per poter essere vista e riconosciuta dall'interprete e dall'analizzatore delle rules (reasoner). Questa classe contiene un registro per fare il mapping delle funzioni di oggetti Java che implementano il loro comportamento. Nello specifico, il metodo register permette di registrare un'implementazione della built-in creata.

Di seguito sono descritti, più nello specifico, i metodi utilizzati nelle varie classi e le scelte procedurali ed algoritmiche adottate in ciascuna situazione.

La classe Management.java

La classe "Management.java" contiene l'inizializzazione di un insieme di costanti e metodi necessari per l'implementazione delle built-ins per la gestione del tempo.

Le variabili sono:

- **seconds** = 1000 ;
- **minutes** = seconds * 60 ;

- **hours** = minutes * 60 ;
- **days** = hours * 24 ;
- **months** = 30.44 * days ;
- **years** = months * 12 .

Ad ognuna delle variabili di tipo long, definite come costanti, è stato attribuito un valore in millisecondi.

La variabile **months** è costituita da un valore numerico pari a 30.44 moltiplicato per l'equivalente risultato numerico di un giorno in millisecondi.

Questa costante (30.44) rappresenta il numero medio dei giorni che sono presenti all'interno di un mese. Di conseguenza, non sono presi in considerazione gli anni bisestili. La loro trattazione potrà, però, essere valutata in una seconda fase di sviluppo ed implementazione delle funzionalità del programma. Risulta infatti complesso stabilire, gestendo le date in millisecondi, il numero di giorni effettivi presenti all'interno di un mese, in un determinato anno solare.

Inoltre, nella classe sono stati istanziati diversi metodi, riassunti nella Tabella 3.1. Poniamo di seguito l'attenzione a due di essi, i quali permettono di stabilire una relazione biunivoca fra l'insieme delle date e l'insieme dei numeri reali. Infatti,

- **long dateToMills (Calendar d)**: permette di convertire un oggetto di tipo "Calendar" in numero reale. In Java, il tempo (data e ora) viene gestito tramite le classi java.util.Calendar e java.util.GregorianCalendar. Nella trasformazione in questione è stato utilizzato il metodo getTimeInMillis(), della classe Calendar, il quale restituisce il valore temporale dell'oggetto passato in input.
- **Date millsToDate (long mill):** consente di trasformare un oggetto di tipo "long" in un oggetto "Date", grazie al metodo costruttore della classe "Date". Essa è una classe che permette di specificare un preciso istante temporale, con precisione in milliseconds.

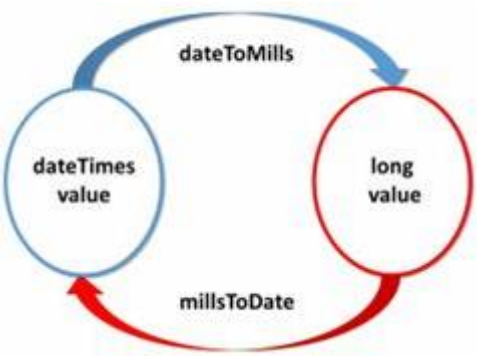


Figura 9: Trasformazione Time to Millisecond

Nella Tabella 2 sono visionabili i metodi della classe appena descritta.

Tipo di ritorno	Nome metodi e parametri	Descrizione
long	dateToMillisecs (Calendar d)	Restituisce il valore in millisecondi, corrispondente alla data di tipo Calendar fornita in input alla funzione.
Date	millToDate (long mill)	Restituisce la data corrispondente al valore in millisecondi passato come parametro.

long	diffDateDays(Calendar d1, Calendar d2)	Dati in input due oggetti di tipo “Calendar”, viene restituito il numero di giorni di differenza tra i due (il risultato è espresso in valore assoluto).
long	diffDateYears(Calendar d1, Calendar d2)	Dati in input due oggetti di tipo “Calendar", viene restituito il numero di anni di differenza tra i due (il risultato è espresso in valore assoluto).
long	diffDateMonths(Calendar d1, Calendar d2)	Dati in input due oggetti di tipo “Calendar”, viene restituito il numero di mesi di differenza tra i due (il risultato è espresso in valore assoluto).
Date	maxDates (Calendar d1, Calendar d2)	Restituisce il Massimo tra i due oggetti di tipo “Calendar” passati come parametri alla funzione.
Date	minDates (Calendar d1, Calendar d2)	Restituisce il minimo tra i due oggetti di tipo “Calendar” passati come parametri alla funzione.
int	compareDatesTo (Calendar d1, Calendar d2)	Stabilisce un confronto tra due date. Il metodo, restituisce -1 nel caso in cui la prima data è più piccola della seconda, 0 nel caso che le due date siano uguali e 1 nel caso in cui la prima data risulta essere maggiore della seconda.
long	nowDateDays (Calendar d1)	Restituisce il numero di giorni di differenza tra la data attuale ed una data passata come parametro alla funzione.
long	nowDateMonths (Calendar d1)	Restituisce il numero di mesi di differenza tra la data attuale ed una data passata come parametro alla funzione.
long	nowDateYears (Calendar d1)	Restituisce il numero di anni di differenza tra la data attuale ed una data passata come parametro alla funzione.

Tabella 2: Metodi di implementazione della classe Management.java

Custom built-ins realizzate

In Jena, non è possibile gestire le date attraverso le built-ins già predefinite nel sistema e, per questo motivo ne sono state implementate di nuove. In questo modo si estendono delle funzionalità non previste, di default, all’interno del framework utilizzato.

Le custom built-ins realizzate sono dodici. Ognuna di esse con caratteristiche e funzioni proprie, le quali permettono di interagire, estrapolando da qualsiasi ontologia in esame, delle informazioni temporali che non possono essere gestite altrimenti.

Nella Tabella 3, sono elencate tutte le built-in implementate, con descrizione e relativi parametri di Input e Output.

Nome built-in	Parametri	Descrizione
diffDateDays	Input: Due date XSDDateTime Output: Un valore long	Prende in input due date (che verranno poi trasformate in millisecondi per la gestione) e restituisce il numero di giorni di differenza tra le due.
diffDateMonths	Input: Due date XSDDateTime Output: Un valore long	Prende in input due date (che verranno poi trasformate in millisecondi per la gestione) e restituisce il numero di mesi di differenza tra le due.
diffDateYears	Input: Due date XSDDateTime Output: Un valore long	Prende in input due date (che verranno poi trasformate in millisecondiper la gestione) e restituisce il numero di anni di differenza tra le due.
maxDate	Input: Due date XSDDateTime Output: Una data XSDDateTime	Prende in input due date (che verranno poi trasformate in millisecondi per la gestione) e restituisce la data che risulta essere più grande tra le due.
minDate	Input: Due date XSDDateTime Output: Una data XSDDateTime	Prende in input due date (che verranno poi trasformate in millisecondi per la gestione) e restituisce la data che risulta essere più piccola tra le due.
nowDiffDays	Input: Una data XSDDateTime Output: Un valore long	Prende in input una data (che verrà poi trasformata in millisecondi per la gestione) e restituisce il numero di giorni di differenza tra essa e la data attuale.
nowDiffMonths	Input: Una data XSDDateTime Output: Un valore long	Prende in input una data (che verrà poi trasformata in millisecondi per la gestione) e restituisce il numero di mesi di differenza tra essa e la data attuale.
nowDiffYears	Input: Una data XSDDateTime Output: Un valore long	Prende in input una data (trasformata in millisecondi) e restituisce il numero di anni di differenza tra essa e la data attuale.
equalsDate	Input: Due date XSDDateTime Output: Un valore int	Prende in input due date (che verranno poi trasformate in millisecondi per la gestione) e restituisce un valore intero -1 se la prima data è più piccola della seconda, 0 se le due date sono uguali, 1 se la prima data è più grande della seconda.
daysMore	Input: Una data XSDDateTime ed un valore intero Output: Una data XSDDateTime.	Prende in input una data ed un valore intero. Restituisce la stessa data, aggiungendo alla parte del giorno, il valore intero passato come

		parametro.
monthsMore	Input: Una data XSDDateTime ed un valore intero Output: Una data XSDDateTime.	Prende in input una data ed un valore intero. Restituisce la stessa data, aggiungendo alla parte del mese, il valore intero passato come parametro.
yearsMore	Input: Una data XSDDateTime ed un valore intero Output: Una data XSDDateTime.	Prende in input una data ed un valore intero. Restituisce la stessa data, aggiungendo alla parte dell'anno, il valore intero passato come parametro.

Tabella 3: Built-ins implementate

Un esempio: la built-in DiffDateYears.java

"DiffDateYears.java" rappresenta un esempio di classe che implementa una nuova built-in, diffDateYears. Essa restituisce la differenza, espressa in anni, tra due date. L'implementazione di questa classe segue la struttura implementata. Di seguito sono analizzate alcune delle parti di codice:

- **public String getName():** ritorna "diffDateYears";

```
@Override
public String getName ()
{
      return " diffDateYears ";
}
```

- **public int getArgLength():** ritorna il valore numerico tre (riceve due parametri di input ed uno di output)

```
@Override
public int getArgLength () {
      return 3;
}
```

- **public boolean bodyCall(Node[] args, int length, RuleContext context):** questa funzione definisce il corpo della built-in, nella quale sono settate tutte le sue funzionalità. Riceve come parametri il numero di oggetti che gli vengono passati in fase di esecuzione ed il contesto nel quale essa viene elaborata. Il valore di ritorno stabilisce se la built-in è stata eseguita in modo corretto (true) o meno (false).
  Come si può osservare, all'interno del corpo delle built-ins, sono stati istanziati oggetti di tipo "GregorianCalendar" per la gestione di date. Questa scelta è stata dettata dalla limitazione offerta agli oggetti XSDDateTime, tipo di dato riconosciuto in Jena per la definizione di informazioni temporali.
  Dopo aver definito il nome della built-in ed il relativo numero di argomenti, è necessario recuperare i valori passati alla funzione attraverso le seguenti istruzioni.

```
Node n1 = getArg (0, args , context )
Node n1 = getArg (0, args , context )
```

Vengono inizializzati tanti oggetti di tipo Node quanti sono i valori in input passati alla funzione. La prima operazione che si fa, una volta entrati nel corpo della classe è quella di recuperare i due valori (riconosciuti come "Nodi") sui quali si deve poi andare a lavorare.
Segue un controllo per vericare che gli oggetti istanziati siano di tipo Literal affinché sia possibile effettuare operazioni numeriche (come mostrato nel Listato 1).

Listato 1: Oggetti Literal

```
if (n1. isLiteral () && n2. isLiteral ())
{
Object v1 = n1. getLiteralValue ();
Object v2 = n2. getLiteralValue ();
Node differenza = null;
}
```

Successivamente, nel caso specifico, vengono create istanze di oggetti di tipo "XSDDate-Time"per recuperare il valore contenuto nei due metodi precedentemente creati. Queste istruzioni sono riportate nel Listato 2.

Listato 2: Istanze di oggetti XSDDateTime

```
if (v1 instanceof XSDDateTime && v2
instanceof XSDDateTime ) {
XSDDateTime nv1 = ( XSDDateTime ) v1;
XSDDateTime nv2 = ( XSDDateTime ) v2;
```

Si procede con l'implementazione dell'algoritmo che permette di ottenere le funzionalità desiderate. Nell'esempio che si prende in considerazione (Listato 3) viene calcolata laLibreria di Built-ins differenza in anni tra due date, richiamando il metodo diffDateYears, della classe Management.java precedentemente realizzata.

Listato 3: Calcolo della differenza tra due date

```
GregorianCalendar data1 =
new GregorianCalendar (nv1. getYears (),
nv1. getMonths (), nv1. getDays ());
GregorianCalendar data2 =
new GregorianCalendar (nv2. getYears (),
nv2. getMonths (), nv2. getDays ());
Management manage = new Management ();
long diff = manage . diffDateYears (data2 ,
data1);
```

Infine, è necessario creare un nodo (che può essere di quattro tipi: long, int, double, list), all'interno del quale sarà memorizzato il risultato nale della built-in (Listato 4). Esso sarà poi restituito come parametro finale all'interno della rule.

Listato 4: Creazione di un nodo

```
differenza = Util. makeLongNode (diff);
return env.bind(args [2], differenza );
```

Per creare custom built-ins è necessario procedere come nell'esempio appena descritto. Pertanto, ogni classe creata ha la medesima struttura ed estende BaseBuiltin. E' possibile, in questo modo, ampliare ed utilizzare i metodi definiti in tale classe.