Simulatore dati per Km4City

Filippo Ermini
filippo.ermini@gmail.com
Università degli Studi di Firenze Scuola di Ingegneria Informatica

Elaborato per il corso di
Knowledge Management and Protection Systems
Prof. Paolo Nesi
Tutor e supervisore Prof. Pierfrancesco Bellini

Aprile 2017

Sommario

Sviluppo di un componente applicativo, per la generazione di dati simulati per la piattaforma KM4City

Indice

1 Introduzione

L'elaborato riguarda lo sviluppo di un componente applicativo, realizzato in Java, per la piattaforma KM4City funzionale alla generazione di triple rappresentanti una campionatura ad un certo istante di un sensore la cui struttura sia già definita all'interno di KM4City ma del quale non abbiamo ancora a completa disposizione i dati.
KM4City è basata su un'ontologia RDF all'interno della quale sono stati definiti i sensori e la loro relativa struttura organizzata in classi e proprietà.
Il componente realizzato è in grado di generare un set specifico di dati a partire da un semplice file di configurazione, il quale definisce, attraverso un metalinguaggio specifico dell'applicazione, la tipologia di dato d'uscita e la relativa funzione di generazione.
Tramite il file di configurazione andiamo dunque a definire, sulla base della struttura che il sensore ha all'interno dell'ontologia, il valore delle triple *<soggetto,predicato,oggetto>* che andranno a simulare il campionamento del sensore ad un certo istante.

2 Le Triple RDF

Gli applicativi di KM4City utilizzano come fonte di dati file di tipo *.n3*, ubicati all'interno di una specifica alberatura di cartelle, contenenti la rilevazione ad un certo istante di tempo di un intero set di sensori (tutti appartenenti alla solita classe).

```
<*/resource/EM0100101> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*/schema#SensorSite> .
<*/resource/EM0100101> <http://purl.org/dc/terms/identifier> "EM0100101" .
<*/resource/EM0100101> <*/schema#hasObservation> <*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*/schema#Observation> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <http://purl.org/dc/terms/identifier> "6e101162-82b3-46c3-bc8f-b66e1608a131" .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <http://purl.org/dc/terms/date> "2016-12-26T12:02:00+01:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#lastAverageSpeed> "25.1286" .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#averageSpeed> "2.1565166" .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#vehicleFlow> "4.540416525955566"^^<http://www.w3.org/2001/XMLSchema#float> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#concentration> "2.1054401"^^<http://www.w3.org/2001/XMLSchema#float> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#measuredTime> <*/resource/2016-12-26T12:02:01+01:00> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#measuredBySensor> <*/resource/EM0100101> .
<*/resource/2016-12-26T12:02:01+01:00> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2006/time#Instant> .
<*/resource/2016-12-26T12:02:01+01:00> <http://purl.org/dc/terms/identifier> "2016-12-26T12:02:01+01:00" .
<*/resource/2016-12-26T12:02:01+01:00> <*/schema#instantObserv> <*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> .

<*/resource/EM0100102> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*/schema#SensorSite> .
<*/resource/EM0100102> <http://purl.org/dc/terms/identifier> "EM0100102" .
<*/resource/EM0100102> <*/schema#hasObservation> <*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*/schema#Observation> .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <http://purl.org/dc/terms/identifier> "9a1cabf2-b4c3-4342-b145-3519f38ad58d" .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <http://purl.org/dc/terms/date> "2016-12-26T12:02:01+01:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <*/schema#lastAverageSpeed> "25.1286" .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <*/schema#averageSpeed> "2.3200924" .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <*/schema#vehicleFlow> "5.6327852913072"^^<http://www.w3.org/2001/XMLSchema#float> .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <*/schema#concentration> "2.427828"^^<http://www.w3.org/2001/XMLSchema#float> .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <*/schema#measuredTime> <*/resource/2016-12-26T12:02:01+01:00> .
<*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> <*/schema#measuredBySensor> <*/resource/EM0100102> .
<*/resource/2016-12-26T12:02:01+01:00> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2006/time#Instant> .
<*/resource/2016-12-26T12:02:01+01:00> <http://purl.org/dc/terms/identifier> "2016-12-26T12:02:01+01:00" .
<*/resource/2016-12-26T12:02:01+01:00> <*/schema#instantObserv> <*/resource/9a1cabf2-b4c3-4342-b145-3519f38ad58d> .
```

Figura 1: *Esempio di un set di triple RDF (per motivi di rappresentazione grafica la stringa* `http://www.disit.org/km4city` *è stata sostituita con **)

Nell'immagine d'esempio sono mostrate due simulazioni relative a due sensori distinti appartenenti alla solita classe (classe dei sensori del traffico). Ogni processo di simulazione genera un blocco di triple per ciascun sensore appartenente ad un set predefinito risultato di un'interrogazione nella base dati RDF.

2.1 Struttura delle Triple

Ogni blocco di triple appartenente ad una stessa classe , come vedremo, ha la medesima struttura. Di conseguenza è stato possibile implementare un algoritmo in grado di valorizzare i tre parametri della tripla che lo compongono (*soggetto, predicato, oggetto*) in modo automatico a partire da alcuni parametri iniziali.
Il blocco di triple che descrivono il sensore è composto da: classi (o istanze), attributi della classe e riferimenti ad altre classi.
Nell'esempio di figura 2a è riportato un blocco di triple suddiviso per classi. Ogni classe possiede obbligatoriamente l'attributo
`http://www.w3.org/1999/02/22-rdf-syntax-ns#type` che identifica la classe alla quale appartiene quella risorsa. Un altro attributo obbligatorio è

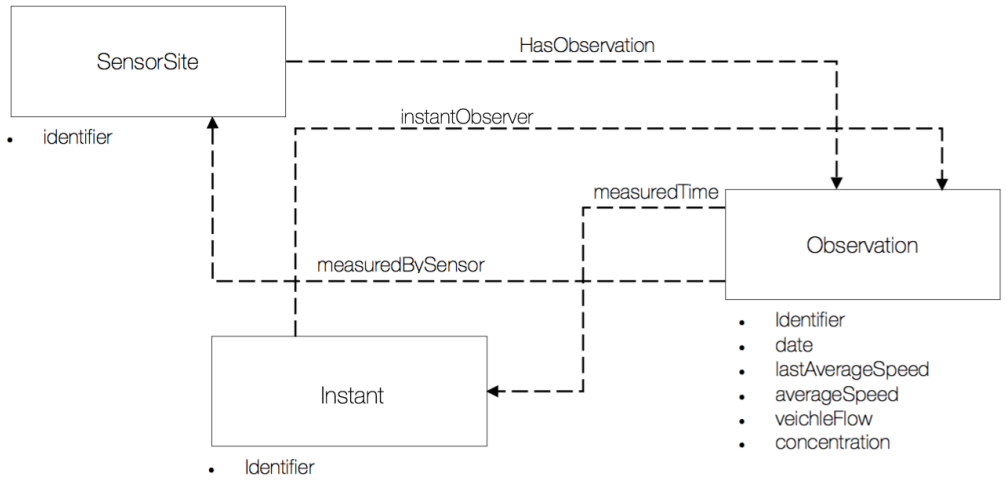
`http://purl.org/dc/terms/identifier`. Il valore di questo attributo identifica l'id della risorsa della classe in questione. Le altre triple della classe, infine, rappresentano proprietà della classe oppure sono riferimenti a risorse di altre classi.

```
<*/resource/EM0100101> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*/schema#SensorSite> .
<*/resource/EM0100101> <http://purl.org/dc/terms/identifier> "EM0100101" .
<*/resource/EM0100101> <*/schema#hasObservation> <*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> .

<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*/schema#Observation> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <http://purl.org/dc/terms/identifier> "6e101162-82b3-46c3-bc8f-b66e1608a131" .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <http://purl.org/dc/terms/date> "2016-12-26T12:02:00+01:00"^^<http://www.w3.org/2001/XMLSchema#dateTime> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#lastAverageSpeed> "25.1286" .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#averageSpeed> "2.1565166" .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#vehicleFlow> "4.540416525955566"^^<http://www.w3.org/2001/XMLSchema#float> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#concentration> "2.1054401"^^<http://www.w3.org/2001/XMLSchema#float> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#measuredTime> <*/resource/2016-12-26T12:02:01+01:00> .
<*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> <*/schema#measuredBySensor> <*/resource/EM0100101> .

<*/resource/2016-12-26T12:02:01+01:00> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2006/time#Instant> .
<*/resource/2016-12-26T12:02:01+01:00> <http://purl.org/dc/terms/identifier> "2016-12-26T12:02:01+01:00" .
<*/resource/2016-12-26T12:02:01+01:00> <*/schema#instantObserv> <*/resource/6e101162-82b3-46c3-bc8f-b66e1608a131> .
```

(a) Esempio di triple suddivise per classi RDF



(b) Schema delle classi RDF riferite alle triple dell'esempio precedente

Figura 2: Triple e schema delle classi RDF

Nelle due immagini è mostrata la struttura di un esempio di triple RDF utilizzate per descrivere l'informazione relativa ad un sensore di traffico. Il tutto è composto da 3 classi ciascuna delle quali è identificata come una risorsa tramite un identificativo. Per ogni risorsa della classe, sono poi, specificati tutti i suoi attributi, mentre i riferimenti verso le altre classi (le frecce tratteggiate nello schema 2b) contengono come valore l'*identifier* della risorsa alla quale fanno riferimento.

3 Il File di Configurazione

A fronte di quanto detto nel paragrafo precedente è stato implementato un algoritmo in grado di generare, in modo sistematico, una simulazione della campionatura in ordine ad un qualsiasi sensore già definito all'interno dell'ontologia a partire da un file nel quale un utente esperto, cioè un utente che ha un'ampia conoscenza dell'ontologia di *KM4City*, specifica come devono essere valorizzate le proprietà delle classi che definiscono il sensore. Tale file è scritto con la sintassi *XML*.

3.1 Struttura del file XML

A seguito è descritta la struttura gerarchica dei nodi e delle loro proprietà che compongono il file di configurazione.

```
<?xml version="1.0"?>
<tree typeId="http://www.w3.org/1999/02/22-rdf-syntax-ns#type" baseUri="http://www.disit.org/km4city/resource/" isStateful="true">
    <instanceIterationQuery>
        <server>http://servicemap.disit.org/WebAppGrafo/sparql</server>
        <query>SELECT DISTINCT ?id WHERE {?s a km4c:SensorSite. ?s dcterms:identifier ?id.filter(!strstarts(?id,"METRO"))}</query>
    </instanceIterationQuery>
    <fileInfo>
        <fileName>sensorsite.n3</fileName>
        <startDirectory>./sensorsite/</startDirectory>
    </fileInfo>
    <iterationElement>
        <instance type="http://www.disit.org/km4city/schema#SensorSite" name="SensorSite" isRoot="true" >
            <properties>
                <prop key="http://purl.org/dc/terms/identifier" isPrimaryKey="true">
                    <type>id</type>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#hasObservation">
                    <type>Observation</type>
                </prop>
            </properties>
        </instance>
        <instance type="http://www.disit.org/km4city/schema#Observation" name="Observation" >
            <properties>
                <prop key="http://purl.org/dc/terms/identifier" isPrimaryKey="true">
                    <type>UUID</type>
                </prop>
                <prop key="http://purl.org/dc/terms/date">
                    <type>DateTime</type>
                    <datatype>http://www.w3.org/2001/XMLSchema#dateTime</datatype>
                </prop>
```

Figura 3: Estratto di un file di configurazione xml

3.1.1 <tree>

È l'elemento radice ed ha le seguenti proprietà:

- **typeId**: [Obbligatorio] Il suo valore indica l'`URI` da usare per indicare la tipologia della classe, ovvero, in altre, parole il valore del complemento della tripla che deve indicare il tipo della classe.
- **isStateful** [Opzionale] Se true crea un file *json* con tutti i valori dell'ultima esecuzione, tale file può essere riutilizzato in esecuzioni successive

per tenere conto dei valori precedentemente generati (le specifiche di questa funzionalità sono descritte a seguire nel paragrafo relativo alla tipologia **previusstate**).

3.1.2 <fileInfo>

Questo elemento definisce le specifiche del file di uscita, e deve contenere i seguenti elementi:

- **FileName** [Obbligatorio] Specifica il nome del file *.n3* d'uscita.
- **StartDirectory** [Obbligatorio] È la cartella all'interno della quale verranno generati i file risultato della simulazione.

A partire dalla cartella specificata sarà creata un'alberatura di cartelle con il seguente formato: `AAAA_MM\GG\HH\mmss\filename.n3`, dove i simboli rappresentano i valori di data e e ora.

3.1.3 <instanceIterationQuery>

Questo elemento contiene la definizione di una query SPARQL. Al suo interno si specifica la query SPARQ, i cui risultati rappresentano la lista di sensori sui quali iterare. La query deve avere una struttura tale da restituire in uscita una lista di identificativi che rappresentano i sensori per i quali si vogliono generare delle simulazioni.
Questo elemento non possiede proprietà ma deve aver valorizzati i seguenti elementi:

- **query** [Obbligatorio] Query sparql
- **server** [Obbligatorio] End-point a cui sottoporre la query
- **bindingValue** [Opzionale] Campo della select su cui andare a recuperare il valore d'uscita (default = id)

3.1.4 <iterationElement>

È l'elemento che contiene le definizioni della struttura RDF e le regole di generazione dei valori del sensore. Per ogni elemento generato dalla query si rieffettua una generazione dei dati secondo le regole definite all'interno di questo elemento.
All'interno di questo elemento si definiscono quindi le istanze e le proprietà del sensore in accordo con la struttura della base dati RDF.
Questo oggetto contiene 2 tipologie di elementi:

- **attributes** [Opzionale] Questo elemento contiene una lista di proprietà che non si riferiscono direttamente alla struttura RDF ma sono degli attributi di supporto alla generazione dei valori delle proprietà del sensore. Talvolta può accadere che il valore di alcuni attributi sia il risultato di una combinazione di fattori la cui generazione risulterebbe complicata se eseguita in una sola volta. L'idea che sta alla base dell'utilizzo di questa funzionalità è di poter demandare la generazione di quei valori, il cui risultato è funzione di più elementi, in questa sezione per poi referenziarli nella proprietà opportuna.
  Un caso emblematico in cui è utile dichiarare una variabile di appoggio in questa sezione, è quello in cui un valore sia funzione del risultato di una query e che in aggiunta debba subire elaborazioni successive. Per casi come questi, a meno che non sia possibile eseguire l'elaborazione del dato direttamente dalla query `SPARQL`, si può dichiarare in questa sezione una variabile che conterrà il risultato della query e nell'apposita proprietà (all'interno dell'istanza di cui fa parte) il valore verrà determinato attraverso una espressione, funzione della variabile che abbiamo dichiarato in questa sezione.

- **list<istance>** [Obbligatorio] Lista delle istanze che definiscono la struttura del sensore. In questa sezione si definiscono le istanze (rappresentanti le classi) e le proprietà ad esse associate necessarie a definire l'intero oggetto (sensore) RDF. Le istanze contengono le informazioni necessarie a definire la tripla. Ognuna di esse contiene una lista di proprietà dove per ognuna di esse sono definite oltre alle informazioni utili a generare la tripla anche le regole di generazione del valore del singolo attributo.

3.1.5 <istance>

Elemento che definisce la struttura e gli attributi di un'istanza RDF.
Proprietà:

- `type`: valore dell'uri *rdf-syntax-ns#type* dell'istanza.

- `name`: nome attribuito all'istanza utilizzato per richiamarla all'interno di altre istanze.

- `isRoot`:se true indica se questa istanza contiene l'elemento su cui inserire il valore generato dalla query.

- `baseUri`: valore del *baseURI* dell'istanza.

Elementi:

- **properties** [Obbligatorio] Elemento che contiene la lista di attributi che definiscono l'istanza.

3.1.6 <properties>

Contiene la lista delle proprietà dell'istanza.

- **list<prop>** [Obbligatorio]

3.1.7 <prop>

Con questo elemento si specificano le singole proprietà dell'istanza e come esse devono essere generate. Per ognuna di esse sono stati definiti dei campi utili alla generazione dei valori della tripla RDF ed altri campi necessari alla generazione del valore dell'attributo. Come vedremo in seguito sono stati individuati alcuni tipi con cui una proprietà può essere definita. Ogni tipo necessita di ulteriori attributi definiti ad hoc per quella tipologia, utili alla generazione del valore di uscita.
Proprietà:

- `key`: valore della uri del predicato di quell'attributo.
- `isPrimary`: se *true* indica che quell'attributo è chiave primaria per l'istanza a cui appartiene (solo un attibuto per istanza può essere chiave primaria).

Elementi:

- **type** [Obbligatorio]: indica il tipo di valore che deve essere generato (ogni tipo può necessitare elementi aggiuntivi).
- **uri** [Obbligatorio]: valore dell'oggetto della tripla RDF.
- **format** : valore contenente la stringa con sintassi "C-like" del formato di uscita (e.g. "%.3f"). Il valore generato verrà dunque formattato in funzione della stringa definita all'interno dell'elemento.

3.2 Lista Tipi

La generazione dei valori può avvenire in molti modi: per cercare di offrire la maggiore capacità espressiva sono state implementate molteplici tipologie per la generazione pseudo-casuale dei valori delle proprietà. Ognuno dei tipi

sotto elencati può essere utilizzato come tipo di valore d'uscita per quella specifica proprietà.Sarà sufficiente specificarlo nell'attributo *type* con a seguito i parametri di cui necessita.

3.2.1 id

Definisce che la proprietà dovrà contenere il valore proveniente dalla query di iterazione, questa proprietà deve essere attribuita all'istanza con *isRoot = true*.
Parametri aggiuntivi: Nessuno

3.2.2 integer

L'attributo restituirà un valore intero generato casualmente in un intervallo.
Parametri aggiuntivi:

- `maxValue`: Estremo superiore dell'intervallo.
- `minValue`: Estremo inferiore dell'intervallo.

3.2.3 float

L'attributo restituirà un valore float generato casualmente in un intervallo.
Parametri aggiuntivi:

- `maxValue`: Estremo superiore dell'intervallo.
- `minValue`: Estremo inferiore dell'intervallo.

3.2.4 md5

Genera il valore md5 di una stringa
Parametri aggiuntivi:

- `md5String`: Contiene la stringa da crittografare

3.2.5 datetime

Tipologia che genera una data in formato ISO 8601, se non specificati gli estremi dell'intervallo la data assume il valore dell'istante in cui è generata, altrimenti genera un valore casuale nell'intervallo di date. Parametri aggiuntivi:

- `maxValue`: Estremo superiore dell'intervallo.
- `minValue`: Estremo inferiore dell'intervallo.

3.2.6 hourdependent

Questa tipologia serve a generare dei valori diversi per ogni ora della giornata in base a dei valori orari di riferimento specificati. Parametri aggiuntivi:

- `hourValue`: Contiene i valori di riferimento per ogni ora. Devono essere inseriti 24 valori separati da ';'.
- `range`: Contiene il range di variazione all'interno del quale possiamo far variare il valore di riferimento indicato. Può essere un valore unico per tutti e 24 i valori oppure può essere definito in modo analogo a hourValue indicando i 24 valori del range.

3.2.7 profiledependent

Con questa tipologia, che è concettualmente molto simile alla precedente, è possibile generare il valore di un attributo a partire da un set di profili normalizzati (tra 0 - 1) definiti per specifici giorni della settimana con valori campionati ad intervalli di 15 minuti (96 campionature) che dovranno essere infine moltiplicati per un valore di riferimento. Questi valori sono specificati in file in formato *.csv* che deve essere importato dal sistema.
All'interno di un file *.csv* possono essere definiti più di un profilo (e.g. "consumo elettrico", "luce", "umidità" ecc...) per ognuno di questi sono definiti i valori giornalieri di riferimento. La distinzione dei giorni può essere fatta

profileId	dayType	timeSlot	value
consumoElettrico	sat	0	0.8
consumoElettrico	sat	1	0.8
consumoElettrico	sat	2	0.8
consumoElettrico	sat	3	0.8
consumoElettrico	sat	4	0.7
consumoElettrico	sat	5	0.6
consumoElettrico	sat	6	0.4
consumoElettrico	sat	7	0.3
consumoElettrico	sat	8	0.2
consumoElettrico	sat	9	0.2
consumoElettrico	sat	10	0.2
consumoElettrico	sat	11	0.2
consumoElettrico	sat	12	0.2
consumoElettrico	sat	13	0.2
consumoElettrico	sat	14	0.2
consumoElettrico	sat	15	0.2
consumoElettrico	sat	16	0.2
consumoElettrico	sat	17	0.2
consumoElettrico	sat	18	0.3
consumoElettrico	sat	19	0.4
consumoElettrico	sat	20	0.5
consumoElettrico	sat	21	0.6
consumoElettrico	sat	22	0.6
consumoElettrico	sat	23	0.6

consumoElettrico	any	24	0.5
consumoElettrico	any	25	0.5
consumoElettrico	any	26	0.6
consumoElettrico	any	27	0.6
consumoElettrico	any	28	0.6
consumoElettrico	any	29	0.6
consumoElettrico	any	30	0.7
consumoElettrico	any	31	0.7
consumoElettrico	any	32	0.7
consumoElettrico	any	33	0.4
consumoElettrico	any	34	0.5
consumoElettrico	any	35	0.6
consumoElettrico	any	36	0.7
consumoElettrico	any	37	0.8
consumoElettrico	any	38	0.5
consumoElettrico	any	39	0.5
consumoElettrico	any	40	0.5
consumoElettrico	any	41	0.7
consumoElettrico	any	42	0.7
consumoElettrico	any	43	0.7
consumoElettrico	any	44	0.7
consumoElettrico	any	45	0.7
consumoElettrico	any	46	0.7
consumoElettrico	any	47	0.7
consumoElettrico	any	48	0.7

luce	any	0	0
luce	any	1	0
luce	any	2	0
luce	any	3	0
luce	any	4	0
luce	any	5	0
luce	any	6	0
luce	any	7	0
luce	any	8	0
luce	any	9	0
luce	any	10	0
luce	any	11	0
luce	any	12	0
luce	any	13	0
luce	any	14	0
luce	any	15	0
luce	any	16	0
luce	any	17	0
luce	any	18	0.1
luce	any	19	0.2
luce	any	20	0.3
luce	any	21	0.6
luce	any	22	0.7
luce	any	23	1
luce	any	24	1

Figura 4: Alcuni estratti del contenuto di un file *.csv*

specificando dei profili diversi per i diversi giorni della settimana oppure specificando un profilo *any* valido per tutti i giorni per i quali non è definito un profilo. Come nel caso precedente anche qui dobbiamo definire un valore che specifica l'entità del rumore da aggiungere al valore di riferimento. Il valore finale è ottenuto quindi moltiplicando il valore ottenuto dal profilo in base all'orario per un valore di riferimento, tutto sommato con una perturbazione randomica di ampiezza specificata.
Parametri aggiuntivi:

- `profilesFile` Contiene il nome del file *.csv* da utilizzare
- `profileId` Indica quale è il profilo da considerare
- `maxValue` Valore base di riferimento
- `variance` Valore della varianza

3.2.8 fromset

Con questa tipologia si estrae un valore a caso da un set di elementi (se il set contiene un solo elemento sarà estratto sempre quello). Parametri aggiuntivi:

- `set` Lista di valori separati da ';'.

3.2.9 query

Il valore dell'attributo è il risultato dell'esecuzione di una query (la query deve restituire un solo valore).

Elemento della stessa tipologia di ***instanceIterationQuery****, con la sola differenza che in questo caso la query deve restituire un solo record e non una lista.*

3.2.10 valueexpression

Definendo un attributo con questa tipologia si permette di definire il valore di un attributo come risultato di una espressione algebrica o logica contenete variabili.
Nel caso in cui si necessiti di calcoli più complessi rispetto ad una semplice espressione algebrica è possibile inserire del codice Javascript da far eseguire al sistema.
Parametri aggiuntivi:

- `valueExpression` Contiene l'espressione algebrica o logica da eseguire. Le variabili devono essere dichiarate con la seguente sintassi ${nome} dove nome è il valore inserito nelle proprietà dell'attributo prop

In caso di espressioni algebriche il valore restituito è sempre di tipo float.
Per una descrizione più approfondita delle variabili ed espressioni rgolari si rimanda al capitolo 3.3 .

3.2.11 foregoing

Attraverso questa tipologia si può andare a referenziare il valore di una variabile generata in passi precedenti all'interno della stessa query di iterazione. Lo scopo di questa tipologia è di permettere di referenziare all'interno di una iterazione un valore generato per un altro sensore sempre all'interno dello stesso ciclo di iterazione corrente.
Parametri aggiuntivi:

- `defaultValue`: Valore da usare nel caso in cui l'attributo interessato non sia presente oppure non sia ancora stato generato.
- `refValue`:contiene la coppia *iterazione - nomeVaribile* da cui prendere il valore. Il valore di iterazione può essere sia un intero, che rappresenta l'indice dell'iterazione da cui prendere il valore, oppure l'id del sensore da cui prendere il valore interessato che viene specificato con la variabile nomeVariabile: il tutto con la sintassi [iterazione]{nomeVariabile}.

3.2.12 previusstate

Questa tipologia ha un funzionamento simile alla precedente, con la differenza che in questo caso si può andare a referenziare valori generati in esecuzioni precedenti e non all'interno della stessa esecuzione. Questa tipologia è utile per generare i valori di quegli attributi che hanno una dipendenza funzionale da stati precedentemente generati. Per fare in modo di poter utilizzare questa tipologia è necessario che sia settato a true la proprietà *isStateful* nell'elemento `<tree>`. Così facendo il programma creerà un dump (in un file esterno in formato json) delle variabili generate ad ogni esecuzione in modo da poter recuperare il valore interessato e poterlo utilizzare nell'iterazione corrente.
Parametri aggiuntivi:

- `defaultValue`: Valore da usare nel caso in cui il valore interessato non sia presente oppure nel caso in cui non esista un'iterazione precedente a quella in oggetto.

- `refValue`:contiene la tripla *indice* - *idSensore* - *nomeVaribile* da cui prendere il valore. L'`indice` specifica quale iterazione precedentemente generata si deve considerare (il conteggio avviene considerando l'iterazione corrente la 0 e la 1 quella precedentemente generata). Ogni iterazione contiene una lista sensore; quindi è necessario specificare a quale sensore si fa riferimento tramite l'`idSensore` ed infine è necessario specificare il `nome` della proprietà che si vuole ottenere. La sintassi da utilizzare per definire questa query di ricerca è la seguente: #[indice][idSensore]{nomeVariabile}.

3.3 Variabili e Espressioni Algebriche

Talvolta accade che i valori di alcune proprietà abbiano delle dipendenze funzionali da altri valori e che se ne debba tenere conto durante la fase di generazione. I valori che sono generati per le proprietà delle classi sono valori estratti casualmente: anche se ne viene definito l'intervallo all'interno del quale variare, sono pur sempre valori casuali. Per quei valori che sono funzione di altri quindi deve essere introdotta una dipendenza dal valore generato casualmente.
Un esempio è il caso dei sensori dei parcheggi, i quali restituiscono informazioni sulle statistiche di utilizzo delle strutture.

```
<*resource/CarParkSensor Alberti> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*schema#CarParkSensor> .
<*resource/CarParkSensor Alberti> <http://purl.org/dc/terms/identifier> "CarParkSensor Alberti" .
<*resource/CarParkSensor Alberti> <*schema#capacity> "313" .
<*resource/CarParkSensor Alberti> <*schema#hasRecord> <*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> .

<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <*schema#SitutatioRecord> .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <http://purl.org/dc/terms/identifier> "8a5b399d-47f8-40ec-9748-262f23b104a9" .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <http://purl.org/dc/terms/date> "2017-03-25T10:07:59+01:00" .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#relatedToSensor> <*resource/CarParkSensor Alberti> .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#observationTime> <*resource/2017-03-25T10:07:59+01:00> .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#validityStatus> "active" .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#occupied> "166" .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#free> "147" .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#parkOccupancy> "53.04" .
<*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> <*schema#carParkStatus> "enoughSpacesAvailable" .

<*resource/2017-03-25T10:07:59+01:00> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2006/time#Instant> .
<*resource/2017-03-25T10:07:59+01:00> <http://purl.org/dc/terms/identifier> "2017-03-25T10:07:59+01:00" .
<*resource/2017-03-25T10:07:59+01:00> <*schema#instantParking> <*resource/8a5b399d-47f8-40ec-9748-262f23b104a9> .
```

Figura 5: Esempio di campionatura per un sensore di parcheggio

Il valore principale che deve essere generato casualmente è il "*numero di posti occupati*", di conseguenza valori come *numero di posti liberi* e *percentuale di occupazione* devono essere funzione del valore casuale generato in precedenza. Per risolvere questo tipo di problematiche è stata introdotta la possibilità di referenziare valori di proprietà già generati attraverso delle variabili e la possibilità di inserire espressioni algebriche.
Per ogni proprietà definita è necessario specificare un nome e per referenziare

quel valore all'interno di altre proprietà si deve utilizzare la seguente sintassi: ${*Nome_Proprietà*} I valori nei quali è inserita una dipendenza funzionale saranno calcolati solo dopo aver generato tutte le loro dipendenze e nei casi in cui è necessario sarà possibile eseguire delle espressioni per determinare il valore di tali proprietà.

3.4 Esempio di file di configurazione

Riprendendo l'esempio dei sensori di parcheggio citato nel paragrafo precedente, in questa sezione sarà mostrato un esempio del file di configurazione necessario per generare la simulazione relativa ad una campionatura di tale sensore (cfr. figura 5).

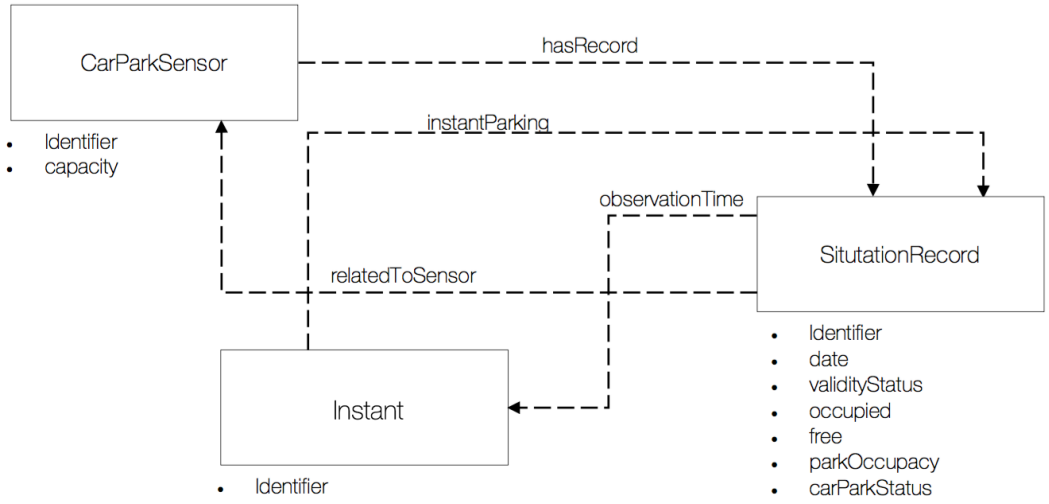


Figura 6: Schema delle classi RDF dei sensori di parcheggio

A partire dallo schema in figura il file *.xml* per la generazione delle triple è il seguente:

```
<?xml version="1.0"?>
<tree   typeId="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
        baseUri="http://www.disit.org/km4city/resource/" isStateful="false">
        <instanceIterationQuery>
                <server>http://servicemap.disit.org/WebAppGrafo/sparql</server>
                <query>SELECT DISTINCT ?id WHERE
                                        {?s a km4c:CarParkSensor.
                                         ?s dcterms:identifier ?id.
                                         ?s km4c:capacity ?o}
                                         limit 100
                </query>
        </instanceIterationQuery>
        <fileInfo>
                <fileName>CarParkSensor.n3</fileName>
                <startDirectory>./parking</startDirectory>
        </fileInfo>
```

Figura 7: Definizione della query di iterazione e de file di output

Nel `tree` si dichiarano rispettivamente la `uri` che conterrà il tipo dell'istanza e con `baseUri`, la stringa che definisce la prima parte del soggetto alla quale

verrà aggiunto poi l'identificativo della risorsa. Per ogni istanza si può definire una `baseUri` diversa specificando tale attributo nella definizione dell'istanza (in caso contrario viene utilizzata quella specificata nel `tree`). La proprietà `isStateful=''false''` specifica che queste simulazioni sono senza memoria.

All'interno dell'elemento `instanceIterationQuery` troviamo la definizione della query sulla quale si dovrà iterare. La query qui definita dovrà restituire sull'attributo `id` una lista di elementi che saranno l'identificativo delle risorse sulle quali verranno generate le simulazioni.

Nell'elemento `fileInfo` andiamo a definire la directory di partenza all'interno della quale creare le cartelle contenenti il file *.n3* e il nome del file di uscita.

Finita la fase iniziale dove sono definite le informazioni generali si passa a specificare la struttura di ogni istanza, seguendo lo schema di figura 5. Per questa simulazione si devono definire tre istanze (*CarParkSensor*, *Instant* e *SituationRecord*).
Le definizioni delle istanze sono inserite all'interno dell'elemento `iterationElement`.

```
<iterationElement>
        <instance type="http://www.disit.org/km4city/schema#CarParkSensor"
                  name="CarParkSensor" isRoot="true">
                <properties>
                        <prop key="http://purl.org/dc/terms/identifier" isPrimaryKey="true">
                                <type>id</type>
                        </prop>
                        <prop key="http://www.disit.org/km4city/schema#capacity">
                                <type>query</type>
                                <name>capacity</name>
                                <queryInfo>
                                        <server>http://servicemap.disit.org/WebAppGrafo/sparql</server>
                                        <query>
                                                SELECT ?capacity WHERE {
                                                ?s a km4c:CarParkSensor.
                                                ?s dcterms:identifier '${id}'.
                                                ?s km4c:capacity ?capacity.
                                                } LIMIT 1
                                        </query>
                                        <bindingValue>capacity</bindingValue>
                                </queryInfo>
                        </prop>
                        <prop key="http://www.disit.org/km4city/schema#hasRecord">
                                <type>SituationRecord</type>
                        </prop>
                </properties>
        </instance>
```

Figura 8: Definizioni dell'istanza CarParkSensor

All'interno dell'elemento `instance` troviamo la definizione: dell'oggetto, il cui predicato identifica il tipo dell'istanza, del nome, dell'istanza e se questa istanza è *root*. A seguito troviamo la definizione di tutte le proprietà della classe (all'interno dell'elemento `properties`). La prima proprietà definisce l'identificativo dell'istanza che essendo *root* conterrà l'elemento estratto dalla query.
La seconda proprietà definisce l'attributo *capacity* che rappresenta la capacità totale del parcheggio. Questo valore è ricavato tramite una query dove viene estratto il valore di capacità del sensore in oggetto. All'interno della query è stato inserito il carattere speciale ${id}[1] che contiene l'identificativo della risorsa nell'iterazione in corso.
Infine definiamo la proprietà *hasRecord* che è un riferimento ad una istanza. Questa definizione viene fatta specificando nell'attributo `type` il nome dell'istanza a cui fare rifermento. Il valore restituito sarà il valore dell'identificativo di questa.

[1]La variabile ${id} insieme alla variabile ${index} sono variabili di sistema e contengono rispettivamente l'id dell'iterazione corrente e l'indice dell'iterazione corrente

```
<instance type="http://www.disit.org/km4city/schema#SitutatioRecord" name="SituationRecord">
        <properties>
                <prop key="http://purl.org/dc/terms/identifier" isPrimaryKey="true">
                        <type>UUID</type>
                </prop>
                <prop key="http://purl.org/dc/terms/date">
                        <type>DateTime</type>
                        <uri>http://www.w3.org/2001/XMLSchema#dateTime</uri>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#relatedToSensor">
                        <type>CarParkSensor</type>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#observationTime">
                        <type>Instant</type>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#validityStatus">
                        <type>fromset</type>
                        <name>validityStatus</name>
                        <set>active</set>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#occupied">
                        <type>integer</type>
                        <name>occupied</name>
                        <uri>http://www.w3.org/2001/XMLSchema#integer</uri>
                        <maxValue>${capacity}</maxValue>
                        <minValue>0</minValue>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#free">
                        <type>valueExpression</type>
                        <name>free</name>
                        <uri>http://www.w3.org/2001/XMLSchema#integer</uri>
                        <valueExpression>${capacity} - ${occupied}</valueExpression>
                        <format>%.0f</format>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#parkOccupancy">
                        <type>valueExpression</type>
                        <valueExpression>( ${occupied} / ${capacity} ) * 100 </valueExpression>
                        <format>%.2f</format>
                </prop>
                <prop key="http://www.disit.org/km4city/schema#carParkStatus">
                        <type>valueExpression</type>
                        <valueExpression>
                                <![CDATA[
                                        function checkStatus(free){
                                                if(free == 0) return 'carParkFull';
                                                else return 'enoughSpacesAvailable';
                                        }
                                        checkStatus(${free});
                                ]]>
                        </valueExpression>
                </prop>
        </properties>
</instance>
```

Figura 9: Definizioni dell'istanza SituationRecord

In figura 9 è mostrata la definizione per l'istanza *SituationRecord* con tutte le rispettive proprietà. In modo analogo al caso precedente anche in questo caso sono state definite la proprietà `identifier` e le proprietà che si riferiscono ad altre istanze (`Instant` e `CarParkSensor`) ed a seguito tutte le altre classi che definiscono questa istanza.

All'interno di questa istanza si fa un uso molto frequente di variabili ed espressioni (dovuto alla natura delle proprietà del sensore). Le variabili possono essere usate sia all'interno di espressioni regolari sia come parametri per la determinazione dei valori come nel caso della proprietà `occupied`, il cui valore deve essere un numero casuale tra 0 e il numero totale di posti di quel parcheggio (per ottenere questo valore si utilizza la variabile ${capacity} che fa riferimento all'omonima proprietà definita nell'istanza precedente).
Per quanto riguarda le espressioni, di particolare interesse è la definizione di *carParkStatus* dove al suo interno è stato definito del codice Javascript.

```
                <instance type="http://www.w3.org/2006/time#Instant" name="Instant">
                        <properties>
                                <prop key="http://purl.org/dc/terms/identifier" isPrimaryKey="true">
                                        <type>DateTime</type>
                                        <uri>http://www.w3.org/2001/XMLSchema#dateTime</uri>
                                </prop>
                                <prop key="http://www.disit.org/km4city/schema#instantParking">
                                        <type>SituationRecord</type>
                                </prop>
                        </properties>
                </instance>
        </iterationElement>
</tree>
```

Figura 10: Definizioni dell'istanza Instant

Infine in figura 10 è mostrata la definizione dell'istanza *Instant* che completa la caratterizzazione di un sensore di parcheggio all'interno dell'ontologia di *km4city*

4 L'applicativo Java

Tutto il sistema descritto nei capitoli precedenti è stato realizzato attraverso un applicativo Java. A partire dal file di configurazione questo applicativo genera il file *.n3* per KM4City. In questo capitolo saranno descritti i componenti principali che compongono l'applicativo.
Il progetto è composto da 4 componenti principali:

1. *Main.* Classe principale, si occupa di avviare il processo di generazione della simulazione e scrivere i risultati su file.
2. *Classi per la gestione del dominio dei dati e dei formati.* In questo applicativo si parte da un informazione di base rappresentata in formato `XML`, inoltre ci sono altri due tipologie di formati che devono essere gestiti: `JSON e CSV`. Per ognuno dei formati sono state definite delle classi per la relativa gestione.
3. *Classi del dominio applicativo.* Sono le classi che si occupano della definizione delle procedure applicative di alto livello.
4. *Classi per la gestione della simulazione generica.* Sono l'insieme delle classi implementate per poter astrarre dalla conoscenza degli aspetti delle diverse tipologie di dato definite e considerare le singole proprietà delle singole istanze come oggetti generici all'interno del processo iterativo di generazione delle triple.

4.0.1 Main

Classe contenente il metodo omonimo che insieme alla classe `DataSimulator.java` si occupa: dell'acquisizione dei parametri iniziali, di fare partire il processo di generazione dei delle triple e di scrivere il file .n3.

```
java -jar sensorGenerator.jar -x sensor.xml -n simulazione_parcheggi
```

Questa è la stringa con la quale viene lanciato il programma da linea di comando, sono necessari due parametri:

1. `-x` indica il file di configurazione xml (obbligatorio)
2. `-n` nome della simulazione (opzionale)

La classe `main` valida gli argomenti passati in ingresso e successivamente istanzia un oggetto della classe `DataSimulator.java`. All'interno di quest'ultima si importa il file di configurazione all'interno di un oggetto e si da il via la processo di generazione delle triple, che una volta concluso, se non si sono verificati errori, genera una stringa con tutte le triple. Una volta generata la stringa con le triple si crea (se non è già stato fatto) le cartelle utilizzando i valori di data e ora (cfr. capitolo 3.1.2). Infine viene creato il file `.n3`.

Nel caso di esecuzione con memoria (cfr. capitolo 3.2.12) i file `.JSON` delle esecuzioni precedenti vengono importati in un'apposita lista al momento dell'importazione del file di configurazione e il file `JSON` relativo all'iterazione corrente viene generato al termine dell'esecuzione nella solita cartella che contiene il file `.n3`.