Data Manipulation Layer

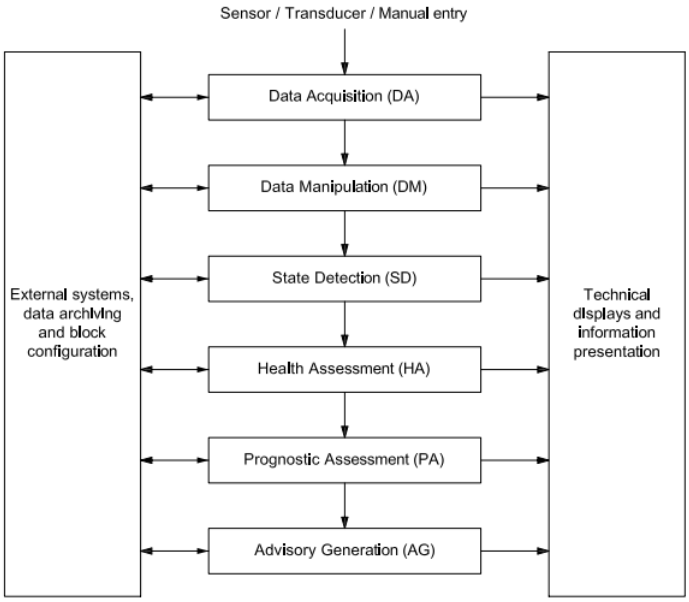
Matteo Minì, Simone Carrea

February 2017

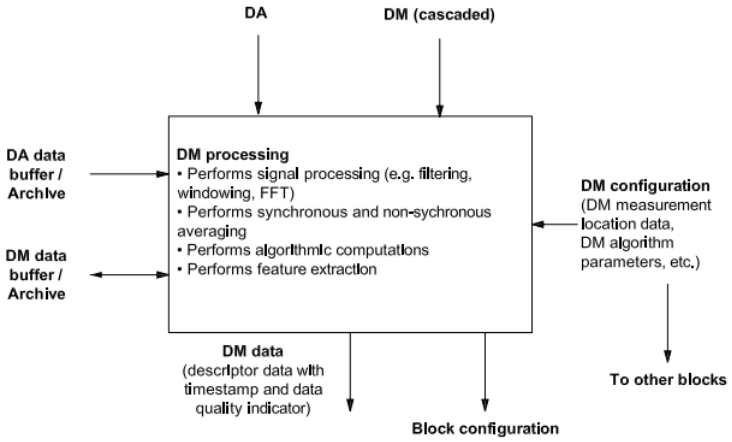
Contents

1 Introduzione

In questo documento si descrive la struttura del Data Manipulation Layer (DM) del sistema diagnostico seguendo le specifiche della normativa **ISO 13374-2** [ISO13374-207].



Il DM si occupa di processare i dati di output provenienti dal livello precedente (Data Acquisition Layer).



1.1 Tipi di dato

Per il livello di manipolazione (DM) abbiamo previsto l'utilizzo di diversi tipi di dato. Questa scelta è dovuta dalla necessità di rappresentare in maniera opportuna le grandezze relative ai sensori del sistema fisico su cui viene eseguita la diagnostica. Qui di seguito la lista dei tipi di dato scelti:

- numero intero a 64 bit (java Long)
- numero in virgola mobile a 64 bit (java Double)
- booleano (java Boolean)
- stringa (java String)
- tupla (java Array)

Abbiamo previsto una restrizione al tipo tupla: esso non può contenere altre tuple (non può essere un Array di Array), ma solamente Long, Double, Boolean e String.

1.2 Regole diagnostiche

Una regola è un processo che, ricevuti dei dati in ingresso elabora un risultato. Il sistema diagnostico permette all'utente di creare la struttura della regola a seconda del caso di utilizzo specifico.
La realizzazione di una regola si divide in due fasi:

- creazione (fase offline)
- esecuzione (fase online)

2 Comunicazione tra i livelli

Si è scelto di utilizzare dei file in formato JSON per la comunicazione tra i vari livelli, per quanto riguarda gli input del DM abbiamo previsto due strutture del file differenti:

- Non geolocalizzati

```
{
        "sensorId": "",
        "value": ,
        "timestamp": 0
}
```

- Geolocalizzati

```
{
        "sensorId": "",
        "value": ,
        "timestamp": 0,
        "GPS": {
            "longitude": 0,
            "latitude": 0,
            "altitude": 0
            }
}
```

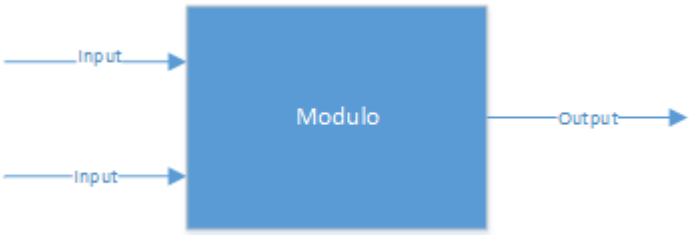
Il campo "sensorId" è una stringa che indica da che sensore proviene il dato; il campo "value", che può essere un intero, un numero in virgola mobile o un booleano, contiene il valore del dato proveniente dal sensore; il campo "timestamp" contiente un intero che indica il tempo in cui è stato registrato il dato dal sistema; il campo "GPS" è un JSON Object contenente i valori "longitude", "latitude"e "altitude" che sono interi indicanti le coordinate geografiche in cui il dato è stato registrato dal sistema.

3 Creazione di una regola

Una delle richieste iniziali del progetto è stata quella di sviluppare un software per la diagnostica che si adattasse a sistemi fisici differenti. Si è deciso quindi di progettare una serie di moduli, ognuno avente un compito specifico, da poter collegare tra di loro a seconda del caso d'uso. Nella fase di creazione l'utente definisce la struttura della regola aggregando diversi moduli.

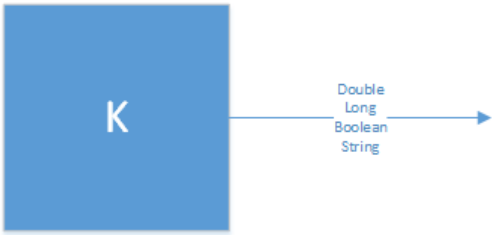
3.1 Moduli

A seconda del tipo, un modulo accetta uno o due dati di input (fuorchè il modulo Costante), può accettare uno o più parametri impostati a priori dall'utente e fornisce un dato di output.
Questi moduli sono collegabili tra di loro, purchè il tipo del dato in uscita dal modulo predecessore sia compatibile con il tipo del dato in ingresso del modulo successore.
La struttura di questi moduli è quella descritta nella figura sottostante:



Costante

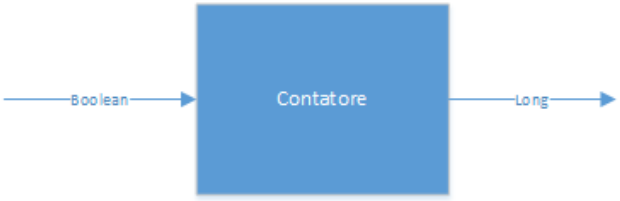
- **Input:** nessuno
- **Parameter:** Double, Long, Boolean, String
- **Output:** Double, Long, Boolean, String



Il modulo Costante è l'unico modulo differente dagli altri poichè non ha input, genera una serie di dati aventi come valore il valore del paramentro impostato a priori.

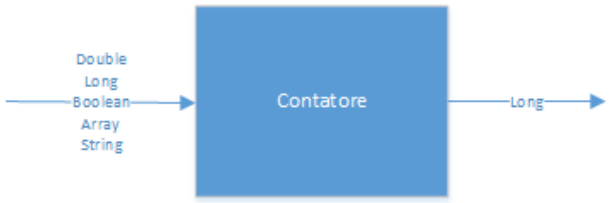
Contatore Ci sono due tipi di contatore differenti:

- **Input:** Boolean
- **Parameter:** Boolean
- **Output:** Long



Questo tipo di contatore può contare il numero di transizioni (da 0 a 1 e viceversa) dei dati di ingresso, oppure tenere traccia del fronte di salita dei dati (per quanto tempo è presente una serie di 1), questi due casi sono gestiti da un parametro booleano che indicherà l'una o l'altra opzione.

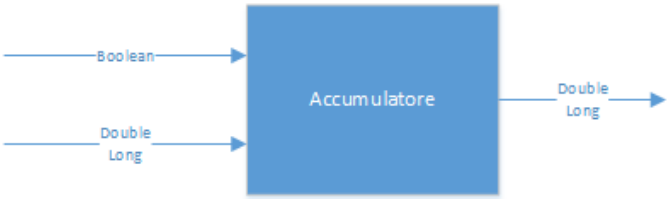
- **Input:** Double, Long, Boolean, Array, String
- **Parameter:** nessuno
- **Output:** Long



Quest'altro tipo di contatore accetta in ingresso una serie di dati e ne conta le occorrenze restituendone il numero totale.

Accumulatore

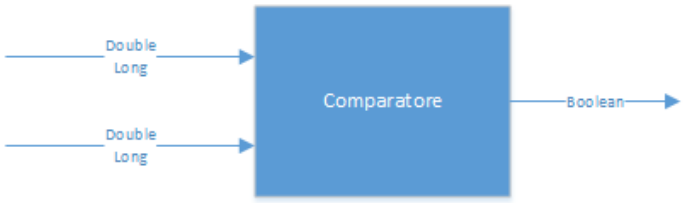
- **Input:**
  1. Double, Long
  2. Boolean
- **Parameter:** Double, Long
- **Output:** Double, Long



Il modulo Accumulatore ha due valori in ingresso: un numero in virgola mobile (o un intero) che verrà sommato ai valori ricevuti precedentemente e un valore booleano che, se impostato a 1, indica la necessità di eseguire un reset. Il parametro, che può essere un numero intero o in virgola mobile, indica il valore da cui far partire la somma una volta effettuato un reset. L'output del modulo sarà un numero intero o in virgola mobile corrispondente alla somma dei valori in ingresso.

Comparatore

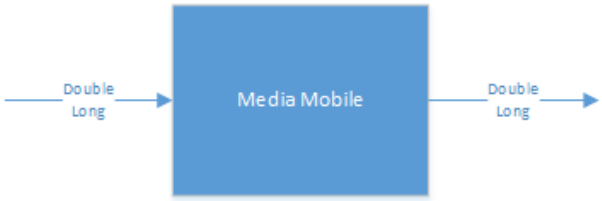
- **Input:**
  1. Double, Long
  2. Double, Long
- **Parameter:** nessuno
- **Output:** Boolean



Questo modulo si occupa di eseguire un confronto del tipo $x_1 < x_2$, dove x_1 e x_2 sono i due valori in ingresso (i quali devono condividere il tipo). L'output è un valore booleano che indicherà se il confronto è verificato o meno.
Per gestire altri tipi di confronto (ad esempio $\leq, >, \geq, =$) sarà necessario connettere in modo opportuno questo modulo ad uno o più moduli logici.

Media mobile

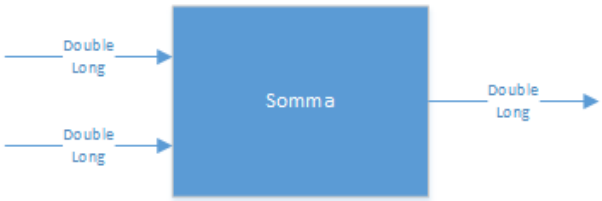
- **Input:** Double, Long
- **Parameter:** Long
- **Output:** Double, Long



All'interno di questo modulo viene calcolata la media mobile dai dati di input. Il parametro indica dopo quanto tempo iniziare a calcolare la media. Se il parametro non viene specificato, il modulo inizia il calcolo della media all'arrivo del primo dato in ingresso.
Input e output devono condividere lo stesso tipo, quindi se in ingresso vi sono numeri interi (in virgola mobile) in uscita si otterrà un numero intero (in virgola mobile).

Somma

- **Input:**
  1. Double, Long
  2. Double, Long
- **Parameter:** nessuno
- **Output:** Double, Long



Questo modulo calcola la somma tra i due dati di input, i quali devono condividere il tipo.

Differenza

- **Input:**
  1. Double, Long
  2. Double, Long
- **Parameter:** nessuno
- **Output:** Double, Long

Questo modulo calcola la differenza tra i due dati di input, i quali devono condividere il tipo.

Integratore

- **Input:**
  1. Double, Long
  2. Boolean
- **Parameter:** nessuno
- **Output:** Double, Long

Questo modulo esegue l'integrale dei dati numerici in ingresso.
L'ingresso booleano ha il compito di definire l'intervallo di integrazione: l'estremo iniziale sarà indicato dal primo fronte di salita della serie di dati in input, mentre quello finale dal successivo fronte di discesa. Possiamo, quindi, distinguere due casi:

1. $\int_a^b f(x)\,dx$, è presente sia il fronte di salita dell'input booleano sia il fronte di discesa.
2. $\int_a^\infty f(x)\,dx$, è presente soltanto il fronte di salita, quindi l'integrale viene calcolato fino all'ultimo dato numerico disponibile.

Dove $f(x)$ rappresenta la funzione descritta dai dati numerici in ingresso mentre a e b i fronti di salita e discesa dell'ingresso booleano.

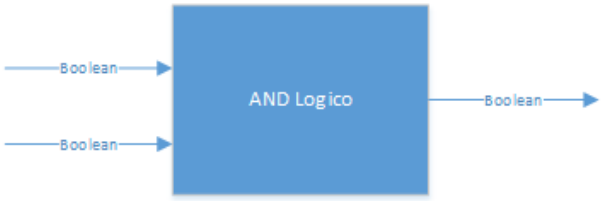
Derivatore

- **Input:**
  1. Double, Long
  2. Boolean
- **Parameter:** nessuno
- **Output:** Double, Long

Questo modulo si occupa di calcolare l'andamento (crescente o decrescente) della funzione rappresentata dai dati numerici di ingresso.
L'ingresso booleano ha il compito di definire l'intervallo su cui calcolare l'andamento: l'estremo iniziale sarà indicato dal primo fronte di salita della serie di dati in input, mentre quello finale dal successivo fronte di discesa.

AND Logico

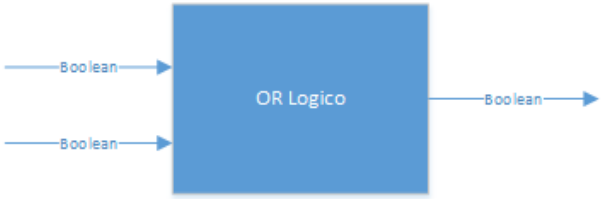
- **Input:**
  1. Boolean
  2. Boolean
- **Parameter:** nessuno
- **Output:** Boolean



Questo modulo esegue l'operazione di AND logico ($\wedge$) tra i due valori booleani d'ingresso.

OR Logico

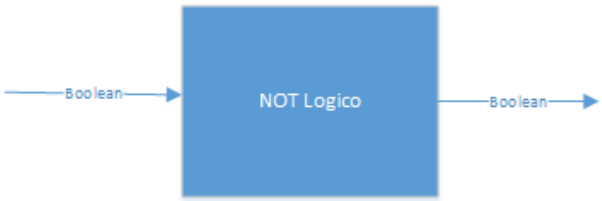
- **Input:**
  1. Boolean
  2. Boolean
- **Parameter:** nessuno
- **Output:** Boolean



All'interno di questo modulo viene implementata l'operazione di OR logico ($\vee$) tra i due valori booleani in ingresso.

NOT Logico

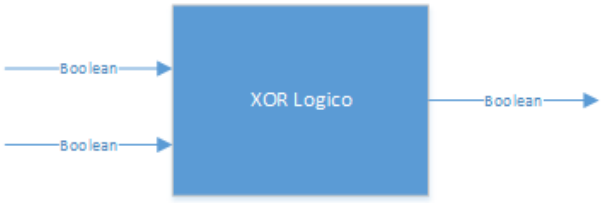
- **Input:** Boolean
- **Parameter:** nessuno
- **Output:** Boolean



Questo modulo implementa l'operazione di NOT logico ($\neg$) andando a negare il valore booleano in ingresso.

XOR logico

- **Input:**
  1. Boolean
  2. Boolean
- **Parameter:** nessuno
- **Output:** Boolean



Questo modulo implementa l'operazione di XOR logico ($\oplus$) tra i due valori booleani di ingresso.

Concatenatore di stringhe

- **Input:**
  1. String
  2. String
- **Parametro:** String
- **Output:** String

Questo modulo accetta in input due stringhe e le concatena separandole con la stringa definita dal parametro.

Disgregatore

- **Input:** Array
- **Parameter:** nessuno
- **Output:** Double, Long, Boolean, String

Il Disgregatore accetta in ingresso una tupla e restituisce in uscita gli elementi contenuti in essa serializzati.

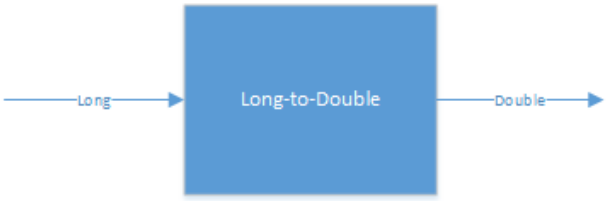
Aggregatore

- **Input:** Double, Long, Boolean, String
- **Parameter:** Long
- **Output:** Array

L'Aggregatore accetta in ingresso una serie di dati e li aggrega in una tupla restituendola in output. Il parametro indica la dimensione della tupla.

Convertitore intero/reale

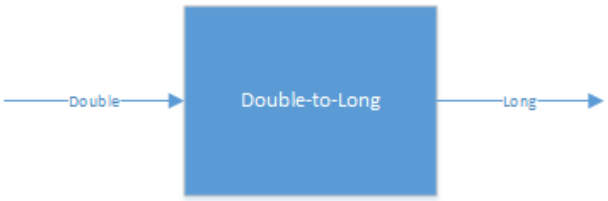
- **Input:** Long
- **Parameter:** nessuno
- **Output:** Double



Questo modulo accetta come input dei numeri interi e li converte in numeri in virgola mobile.

Convertitore reale/intero

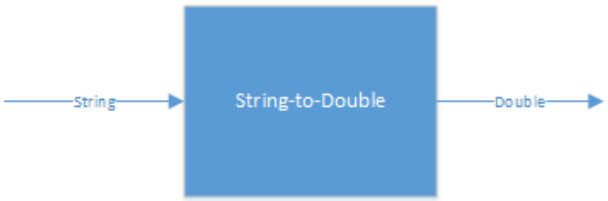
- **Input:** Long
- **Parametro:** Boolean
- **Output:** Double



Questo modulo accetta come input dei numeri in virgola mobile e un parametro che indica il metodo di approssimazione (parte intera superiore o inferiore) ed effettua la conversione in numeri interi.

Convertitore stringa/reale

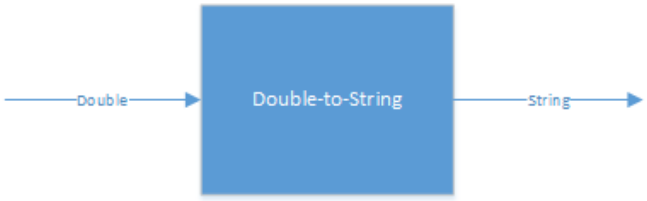
- **Input:** String
- **Parametro:** nessuno
- **Output:** Double



Questo convertitore accetta in input una stringa e la converte in un numero in virgola mobile.

Convertitore reale/stringa

- **Input:** Duoble
- **Parametro:** nessuno
- **Output:** String



Questo convertitore accetta in input un numero in virgola mobile e lo converte in una stringa.

3.2 Editor grafico

Il sistema offre un editor (stile Simulink o PtolemyII) che permette di progettare in modo grafico una o più regole, le quali saranno poi convertite in codice xml e passate ad un compilatore che si occuperà di interpretarle.
L'editor dovrà fornire diverse funzionalità:

- salvataggio di una regola
- caricamento di una regola
- esportazione di una regola in formato XML
- creatione di moduli composti formati dall'unione di moduli semplici
- possibilità di definire una lista di nomi (campo *sensorId* del file JSON), corrispondenti ai sensori su cui eseguire la regola

3.3 Esportazione di una regola

Come illustrato precedentemente, l'editor grafico produrrà un documento XML che definirà la struttura della regola progettata, questo documento verrà poi letto in fase di backend per ricostruire ed eseguire la regola. Per questo compito si è deciso di utilizzare MoML [LN00], un linguaggio di markup XML che permette di specificare interconnessioni tra componenti gerarchici.
Nel nostro caso una regola è sempre formata da: una o più sorgenti, almeno un modulo (complesso o no) e uno o più "sink".
L'elemento *model* definito da MoML [LN00] rappresenta la regola:

```
<model name="rule_n" class="..."></model>
```

l'attributo *name* è una stringa formata dalla parola "rule" e da un numero che identifica in modo univoco la regola.
Sorgenti, moduli e "sink" sono tutti delle *entity* contenute nel *model*.

- **Sorgente:**

  ```
  <entity name="source_n" class="...">
      <property name="sensorId_1" value="..."/>
  </entity>
  ```

  l'attributo *name* dell'entità sorgente è formato dalla stringa **"source"** e da un numero che la identifica in modo univoco. Inoltre una sorgente ha una o più propietà che indicano i nomi dei valori su cui eseguire la regola (campo *sensorId* dei file JSON).

- **Modulo semplice:**

  ```
  <entity name="nomeModulo_n" class="...">
      <property name="nomeParametro" value="..." />
  </entity>
  ```

l'attributo *name* dell'entità modulo semplice è formato da una stringa che indica il nome del modulo e da un numero che lo identifica in modo univoco.
Inoltre questa entità può contenere una proprietà corrispondente ad un eventuale parametro previsto dal modulo.

- **Modulo composto:**

```
<entity name="compositeModel_n" class="...">
</entity>
```

  l'attributo *name* dell'entità modulo composto è formato dalla stringa **"compositeModule"** e da un numero che la identifica in modo univoco.

- **Sink:**

```
<entity name="sink_n" class="...">
</entity>
```

  l'attributo *name* dell'entità sink è formato dalla stringa **"sink"** e da un numero che la identifica in modo univoco.

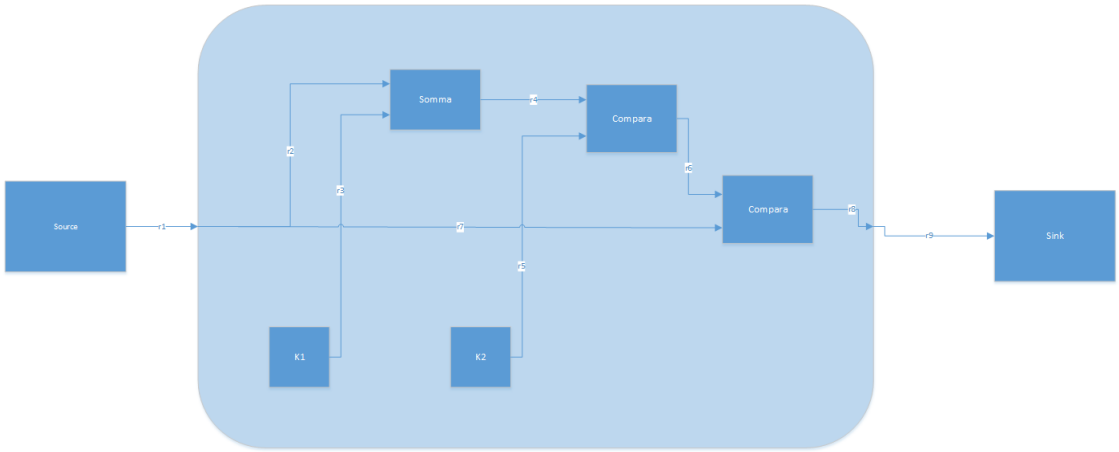
Ogni *entity* ha almeno una porta che la collega con altre entità. Ci sono tre possibili identificativi per le porte, che specificano rispettivamente se la porta è una porta di input, di output oppure entrambe:

```
<port name="in_n" class="..."/>
<port name="out_n" class="..."/>
<port name="inout_n" class="..."/>
```

Anche in questo caso nell'attributo *name* è presente un numero che identifica in modo univoco la porta.

Porte di input/output Queste porte sono specifiche dei moduli composti, in quanto i dati ricevuto in ingresso al modulo dovranno poi essere inviati ai moduli semplici contenuti in esso, viceversa gli output dei moduli semplici saranno ricevuti dal modulo composto che li invierà poi a sua volta.

Esempio Di seguito è presentato un esempio di una semplice regola contenente un modulo composto ed il rispettivo codice XML.




```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE model PUBLIC
        "-//UC Berkeley//DTD MoML 1//EN"
        "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<model name="rule_1" class="package.Rule">
    <entity name="source_1" class="package.Source">
        <property name="sensorId_1" value="id_1"/>
        <property name="sensorId_2" value="id_2"/>

        <port name="out_1" class="package.OPort" />
    </entity>
    <entity name="compositeModel_1" class="package.CompositeModel">
        <port name="inout_1" class="package.IOPort"/>
        <port name="inout_2" class="package.IOPort"/>

        <entity name="constant_1" class="package.Constant">
            <property name="selectedValue" value="5"/>
            <port name="out_1" class="package.OPort"/>
        </entity>
        <entity name="sum_1" class="package.Sum">
            <port name="in_1" class="package.IPort"/>
            <port name="in_2" class="package.IPort"/>
            <port name="out_1" class="package.OPort"/>
        </entity>
        <entity name="constant_2" class="package.Constant">
            <property name="selectedValue" value="10"/>
            <port name="out_1" class="package.OPort"/>
        </entity>
        <entity name="comparator_1" class="package.Comparator">
```

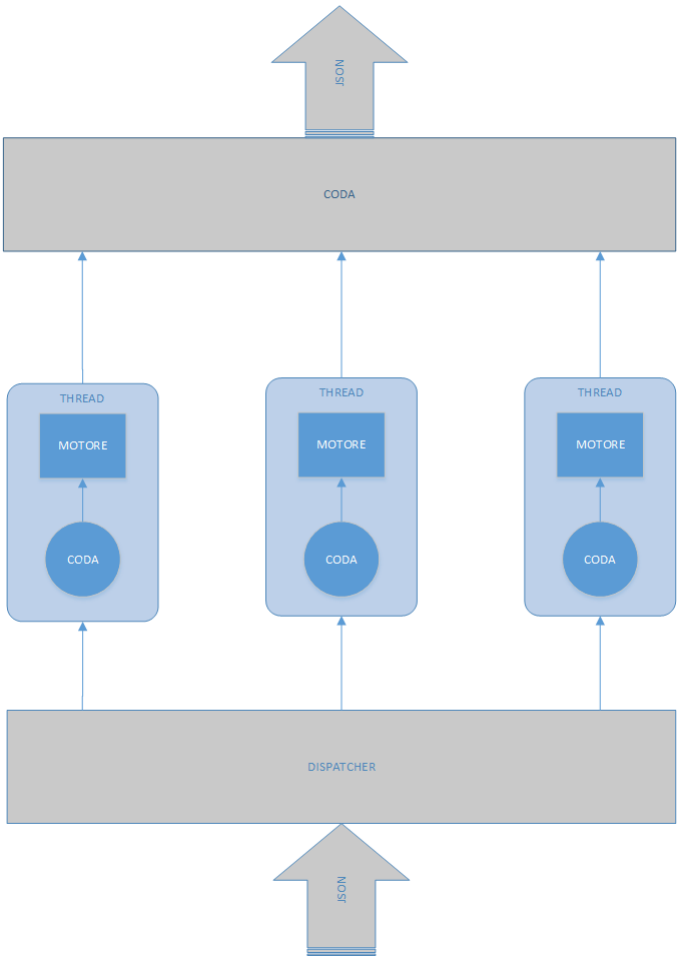
```
            <port name="in_1" class="package.IPort"/>
            <port name="in_2" class="package.IPort"/>
            <port name="out_1" class="package.OPort"/>
        </entity>
        <entity name="integrator_1" class="package.Integrator">
            <port name="in_1" class="package.IPort"/>
            <port name="in_2" class="package.IPort"/>
            <port name="out_1" class="package.OPort"/>
        </entity>
    </entity>
    <entity name="sink_1" class="package.Sink">
        <port name="in_1" class="package.IPort"/>
    </entity>

    <relation name="r_1" class="package.Relation"/>
    <relation name="r_2" class="package.Relation"/>
    <relation name="r_3" class="package.Relation"/>
    <relation name="r_4" class="package.Relation"/>
    <relation name="r_5" class="package.Relation"/>
    <relation name="r_6" class="package.Relation"/>
    <relation name="r_7" class="package.Relation"/>
    <relation name="r_8" class="package.Relation"/>
    <relation name="r_9" class="package.Relation"/>

    <link port="source_1.out_1" relation="r_1"/>
    <link port="compositeModule_1.inout_1" relation="r_1"/>
    <link port="compositeModule_1.inout_1" relation="r_2"/>
    <link port="sum_1.in_1" relation="r_2"/>
    <link port="constant_1.out_1" relation="r_3"/>
    <link port="sum_1.in_2" relation="r_3"/>
    <link port="sum_1.out_1" relation="r_4"/>
    <link port="comparator_1.in_1" relation="r_4"/>
    <link port="constant_2.out_1" relation="r_5"/>
    <link port="comparator_1.in_2" relation="r_5"/>
    <link port="comparator_1.out_1" relation="r_6"/>
    <link port="integrator_1.in_1" relation="r_6"/>
    <link port="compositeModule_1.in_1" relation="r_7"/>
    <link port="integrator_1.in_2" relation="r_7"/>
    <link port="integrator_1.out_1" relation="r_8"/>
    <link port="compositeModule_1.inout_2" relation="r_8"/>
    <link port="compositeModule_1.inout_2" relation="r_9"/>
    <link port="sink_1.in_1" relation="r_9"/>
</model>
```

4 Esecuzione delle regole

In questa fase vengono creati un numero di thread uguale al numero di regole definite dall'utente. Ogni thread contiene il motore che esegue una regola e una coda a priorità ordinata secondo il timestamp, la quale fornirà i dati al motore. Alla base di questa struttura vi è un dispatcher con il compito di ricevere i file JSON dal DA [ISO13374-207] e indirizzare i dati del tipo richiesto da ogni thread nelle rispettive code.
Gli output dei thread sono ricevuti da una struttura (Coda?) con il compito di convertirli in file JSON e inviarli allo SD [ISO13374-207].



References

[ISO13374-207] *ISO 13374-2:2007. Condition monitoring and diagnostics of machines - Data processing, communication and presentation.* Normativa ISO. 2007.

[LN00] Edward A. Lee and Steve Neuendorffer. "MoML - A Modeling Markup Language in XML". In: *Technical Memorandum ERL/UCB M 00/12* (2000).