PROYECTO FIN DE CARRERA

Título:	Desarrollo de una Plataforma Inteligente de Automatización de Tareas
Título (inglés):	Development of an Intelligent Platform for Task Automation
Autor:	Carlos Crespo González-Calero
Tutor:	Miguel Coronado Barrios
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Gregorio Fernández Fernández
Vocal:	Mercedes Garijo Ayestarán
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	Marifeli Sedano Ruiz

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

PROYECTO FIN DE CARRERA

DEVELOPMENT OF AN INTELLIGENT PLATFORM FOR TASK AUTOMATION

Alumno: Carlos Crespo González-Calero

Tutor: Miguel Coronado Barrios

Ponente: Dr. Carlos Ángel Iglesias Fernández

Diciembre 2013

Tempora mutantur et
nos mutamur in illis

Resumen

Esta memoria es el resultado de un proyecto cuyo objetivo es desarrollar una plataforma inteligente para automatización de tareas.

Dicha plataforma tiene la facultad de efectuar acciones en función de determinados eventos o entradas que le llegan a la plataforma. Las dos principales cualidades de esta plataforma son el razonamiento temporal en la parte izquierda de las reglas y el modelado semántico de los elementos de la plataforma.

En primer lugar se presenta la motivación del proyecto seguida por un análisis de la situación actual en los distintos temas en los que este proyecto está envuelto.

Después de éste, se presentan los requisitos planteados para una plataforma que cumpla nuestros objetivos. Una vez aceptada la visión global del proyecto se procede a describir la arquitectura propuesta a un alto nivel de detalle, analizando con exhaustividad cada módulo de la plataforma.

Esta arquitectura ha sido llevada a la práctica a través del desarrollo de la plataforma DrEWE, una plataforma completa que evidencia la efectividad de la solución propuesta.

Por último, se presentan las conclusiones extraídas del trabajo, las posibles líneas de continuación del proyecto así como los siguientes pasos en cuanto a desarrollo y aprovechamiento de la plataforma.

Palabras clave: Automatización de Tareas, Web de Eventos, EWE, Procesado de Eventos Complejos, Web Semántica.

Abstract

This thesis is the result of a project whose objective has been to develop an intelligent platform for task automation.

The developed platform has the faculty of performing actions caused by specific events or inputs that are pushed into the platform depending on some established rules. The most relevant features of this platform are temporal reasoning at the left part of the rule and semantic modelling for the elements of the platform allowing interoperability with other platforms.

Firstly, we present the motivation that has driven us to build this project and secondly we provide a deep analysis of the current situation regarding all the issues involved on the project.

Furthermore, we propose a set of requirements in order to implement a platform that eventually achieve the desired goals. Once this global view is given, we proceed to describe the chosen architecture, analysing exhaustively each module and tier that forms the platform.

This architecture has been put into practice through the development of DrEWE, a complete platform that proves the effectiveness of the proposed solution.

Finally, we gather the extracted conclusions plus a compound of lessons learned, the possible lines of work regarding the continuance of the platform as well as the next step regarding development and exploitation of the service.

Keywords:Task Automation, Evented Web, EWE, Complex Event Processing, Semantic Web.

Acknowledgement

When I arrived at the group of intelligent systems at my university, I had not any programming-related special skills, in fact, I was more than what could be considered an average engineer. Now I realize, that this project has only been the consequence of what is really worth to mention, a fruitful path plagued by challenges and steps that had been climbed.

I really have to thank every person that I have met during this five years in Madrid, each of them has supposed an essential part of my development towards what I am nowadays. I hope to have been a good padawan, particularly of Miguel, the one that really had to suffer me and carry on with every problem we encountered on the way.

Finally I want to dedicate this work to the whole GSI (group of intelligent systems), the one has really made the difference during my days at university.

Agradecimientos

A mis padres, que aunque no vayan a entender nada de lo que hay dentro de este texto siempre es bueno saber que en cierto modo son responsables de él.

A mis compañeros fatigas de la escuela, a pesar de que cada uno ha salido disparado en direcciones de lo más dispares posibles, siempre queda el camino recorrido juntos.

A Carlos y Miguel, que son los verdaderos visionarios que han hecho posible este proyecto.

A todos los que se sientan identificados con este texto y con el hecho de que llegue a cumplir el reto de ser ingeniero. Sin dar nombres para no dejarme a nadie, pero cualquiera que esté leyendo estas líneas sabe si tengo que estar agradecido.

Contents

Listings

List of Figures

Chapter 1

Introduction

"Ideas are the beginning points of all fortunes."

—Napoleon Hill

1.1 Rationale

For the last few years, the number of web services have increased markedly. Many of these services, especially those of them already mature, offers a public API *(Application programming interface)* in order to interconnect them with other services, making them easily accessible. Thanks to this, third party application and services are appearing making use of these APIs. This new world offers plenty of possibilities, however, performing this connection can be complex, requires advanced skills and the learning curve is exponential to the number of services involved. These reasons are what are not allowing the user to benefit from these features.

In this scenario, plenty of new companies are emerging as task automation platforms. These tools like IFTTT[1] or Zapier[2] are in charge of allowing the user to automatically perform tasks such as *"When I am mentioned in Twitter, send me an email"*, in other words, these platforms are able to trigger rules that produce actions, when certain events are received connecting multiple services. These rules are triggered by user's events which

[1] `http://www.ifttt.com`
[2] `http://www.zapier.com`

means that they work only with personal accounts from each services.

However, the current format for these rules are event-condition-action so they do not benefit from all the potential that these platform could offer. In this project, we tackle this problem with the objective of provide an enhanced platform allowing the user to write more complex rules.

This project has been developed in parallel with the development of the EWE ontology, a standardized data schema designed to describe elements within Task Automation Services enabling rule interoperability.

These services are a rising trend among the upcoming new web internet services, what suppose an incoming need of an ontology to assure interoperability between them. Hereby we present DrEWE, a Task Automation Platform with two big special features:

EWE ontology support
: Which means that our platform sets an example of viability of the EWE ontology and could become, in a future, the first platform to standardize the others.

Complex event processing
: This feature allows the when part or left part of the rule to have temporal reasoning. One example of that should be: *"When a meeting is scheduled, if the corresponding attendees enter their Id cards at the entrance during ten minutes before the start time: generate an event"*

Among the advantages and possibilities of this new platform that has been developed, we could note the following:

- A semantic platform for task automation. Which means that it could use a vocabulary like EWE, common to others platforms. All the advantages that a semantic service offers such as knowledge management, classification of information, composition of complex system and information filtering.

- Event aggregation through complex event processing. Possibility of creating more elaborated rules than the classic ones thanks to temporal reasoning that this feature provides.

- Extends the simple rule model including temporal reasoning. In addition to typical when-then rules, temporal reasoning allows us to compose more complex and accurate rules.

- Non-limited when-side of the rules. This platform is able to manage multiple conditions from multiple events that trigger the rules, unlike classic event-condition-action rules that only supports one condition of one event.
- Multiple sources of events. In order to prove the potential of this platform, it is able to receive both events coming from the internet such as a new email or coming from the environment using physical sensors like light level.

1.2 Goals

In the long term, this project aims to provide an extensible and scalable platform for task automation, which will ensure interoperability among platforms through the EWE ontology. This includes, but is not limited to, accessible event network , semantic rule engine, complex event processing rule engine, web handlers for other services, software to manage physical sensors. In a bigger picture, this also includes rule's composer in order to make the interaction with the platform easier.

Among the main goals of this project we find:

- Deepen the knowledge and usage of technologies covered in this project such as: rule engines, event networks and ontology management.
- Build a rule engine that allows both, complex event processing and semantic performance.
- Deploy a sensor network that suits our needs.
- Develop the software in order to wrap third party web services.
- Design some physical sensors to provide events generated from information retrieved of the environment.
- Integrate suitable middleware to handle those physical sensors.
- Develop a communication protocol to connect all the modules.

Other general aims for this project are:

- Study and extend the current state of the state of the art of task automation.
- Explore the capabilities of such technologies for inclusion in intelligent systems.

- Explore and exploit the possibilities of a semantic rule engine.
- Demonstrate viability of the EWE ontology.

1.3 Structure of the document

In order to make it easier for the reader to go through this document, here is a guideline of the contents of each chapter and the connection to each other:

- Chapter 1 gives an overview of the context and rationale of this thesis as well as the relation to the EWE ontology project.
- Chapter 2 first shows the evolution of the technologies related to this thesis, including rule technologies and task automation. This information helps contextualize this thesis and understand its importance and reason.
- Chapter 3 describes the most relevant technologies that have allowed us to achieve these goals. In this chapter we explained technologies such as EWE itself, rule technologies and Global Sensor Network.
- Chapter 4 shows a series of case scenarios that helped shape the requirements of the architecture proposed in Chapter 5.
- Chapter 5 explains in details layered architecture, both functional and modular. In this chapter the relationship between components are described, and how the information flows in the system.
- Chapter 6 exemplifies the architecture explained in the previous chapter, explaining in depth the main use case and describes in detail the implementation of each module.
- Chapter 7 sums up the findings and conclusions found throughout the document and gives a hint about future development to continue the work done for this master thesis.
- Finally, the appendix provide useful related information, especially covering the installation and configuration of the tools used in this thesis.

Chapter 2

State of the Art

"Any fool can know. The point is to understand."

— Albert Einstein

2.1 Introduction

In this chapter we will talk about the history and current state of some concepts involved in this thesis. As it is certainly impossible to understand the implications and essence of this project without knowing them. First we will introduce what has been called the internet of things, then we will give some approach to the global picture of rule technologies and complex event processing and finally, we will analyse some existing task automation platforms.

2.2 The internet of things

The Internet of Things (IoT) is a novel paradigm that is rapidly gaining ground in the scenario of modern wireless telecommunications. The basic idea of this concept is the pervasive presence around us of a variety of things or objects – such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc. – which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbours to reach common goals.

In this context, domotics, assisted living [1], e-health [2, 3], enhanced learning [4] are only a few examples of possible application scenarios in which the new paradigm will play a leading role in the near future. Similarly, from the perspective of business users, the most apparent consequences will be equally visible in fields such as, automation and industrial manufacturing [5], logistics [6, 7], business/process management [8], intelligent transportation of people and goods.

Unquestionably, the main strength of the IoT idea is the high impact it will have on several aspects of everyday-life and behaviour of potential users. From the point of view of a private user, the most obvious effects of the IoT introduction will be visible in both working and domestic fields. In this context, domotics, assisted living, e-health, enhanced learning are only a few examples of possible application scenarios in which the new paradigm will play a leading role in the near future. Similarly, from the perspective of business users, the most apparent consequences will be equally visible in fields such as, automation and industrial manufacturing, logistics, business/process management, intelligent transportation of people and goods.

Actually, many challenging issues still need to be addressed and both technological as well as social knots have to be untied before the IoT idea being widely accepted. Central issues are making a full interoperability of interconnected devices possible, providing them with an always higher degree of smartness by enabling their adaptation and autonomous behaviour, while guaranteeing trust, privacy, and security. Also, the IoT idea poses several new problems concerning the networking aspects. In fact, the things composing the IoT will be characterized by low resources in terms of both computation and energy capacity. Accordingly, the proposed solutions need to pay special attention to resource efficiency besides the obvious scalability problems.

The exact point where this paradigm suits in our project is detailed in section 5.3. Our platform allows physical sensors to retrieve information from the environment, packet them in events and send them via our event network in order to make them available to any point of the network. In order words, we provide a system that can both retrieve information from the environment such as light level and perform actions using the sensors such us take a photo.

2.3 Rule technologies

A rule engine is the computer program that delivers knowledge representation and reasoning to the developer. At a high level it has three components:

- Rules
- Knowledge representation.
- Data

Firstly, we use a knowledge representation for our "things" that could use records or Java classes or full-blown OWL based ontologies. The rules perform the reasoning, i.e., they facilitate "thinking". The distinction between rules and ontologies blurs a little with OWL based ontologies, whose richness is rule based.

The term "rules engine" is quite ambiguous in that it can be any system that uses rules, in any form, that can be applied to data to produce outcomes. This includes simple systems like form validation and dynamic expression engines. For this project, we have considered two different options: Clips and Drools.

2.3.1 Clips

CLIPS is an expert system tool originally developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. Since its first release in 1986, CLIPS has undergone continual refinement and improvement. It is now used by thousands of people around the world.

CLIPS is designed to facilitate the development of software to model human knowledge or expertise. There are three ways to represent knowledge in CLIPS:

- Rules, which are primarily intended for heuristic knowledge based on experience.
- Deffunctions and generic functions, which are primarily intended for procedural knowledge.
- Object-oriented programming, also primarily intended for procedural knowledge. The five generally accepted features of object-oriented programming are supported: classes, message-handlers, abstraction, encapsulation, inheritance, polymorphism. Rules may pattern match on objects and facts.

You can develop software using only rules, only objects, or a mixture of objects and rules. CLIPS has also been designed for integration with other languages such as C and Java. In fact, CLIPS is an acronym for C Language Integrated Production System. Rules and objects form an integrated system too since rules can pattern-match on facts and objects. In addition

to being used as a stand-alone tool, CLIPS can be called from a procedural language, perform its function, and then return control back to the calling program. Likewise, procedural code can be defined as external functions and called from CLIPS. When the external code completes execution, control returns to CLIPS.

2.3.2 Drools

Drools started life as a specific type of rule engine called a Production Rule System (PRS) and was based around the Rete algorithm. The Rete algorithm forms the brain of a Production Rule System and is able to scale to a large number of rules and facts. A Production Rule is a two-part structure: the engine matches facts and data against Production Rules - also called Productions or just Rules - to infer conclusions which result in actions.

The process of matching the new or existing facts against Production Rules is called pattern matching, which is performed by the inference engine. Actions execute in response to changes in data, like a database trigger; we say this is a data driven approach to reasoning. The actions themselves can change data, which in turn could match against other rules causing them to fire; this is referred to as *forward chaining*.

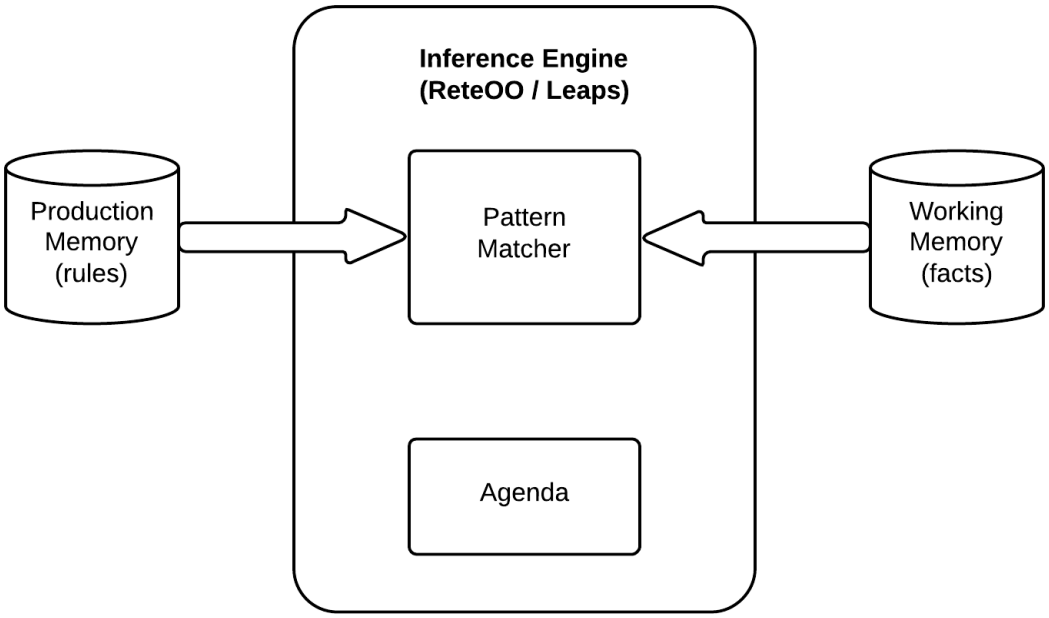

Figure 2.1: High-level view of a production rule system

Drools implements and extends the Rete algorithm. The Drools Rete implementation

is called ReteOO, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for object oriented systems. This algorithm is fully detailed under section 3.4.1.

As we can see in figure 2.1, the Rules are stored in the Production Memory and the facts that the Inference Engine matches against are kept in the Working Memory. Facts are asserted into the Working Memory where they may then be modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion; these rules are said to be in conflict. The Agenda manages the execution order of these conflicting rules using a Conflict Resolution strategy.

2.4 Complex Event Processing

The current understanding of what Complex Event Processing is may be briefly described as the following quote from Wikipedia:

"Complex Event Processing, or CEP, is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes."

Nevertheless, there isn't up to date any broadly accepted definition on the term Complex Event Processing. The term Event by itself is frequently overloaded and used to refer to several different things, depending on the context it is used. Defining terms is not the goal of this guide and as so, lets adopt a loose definition that, although not formal, will allow us to proceed with a common understanding. So, in the scope of this document: Event, is a record of a significant change of state in the application domain.

For instance, on a Stock Broker application (which is one of the most popular uses of this technology), when a sell operation is executed, it causes a change of state in the domain. This change of state can be observed on several entities in the domain, like the price of the securities that changed to match the value of the operation, the owner of the individual traded assets that change from the seller to the buyer, the balance of the accounts from both seller and buyer that are credited and debited, etc. Depending on how the domain is modelled, this change of state may be represented by a single event, multiple atomic events or even hierarchies of correlated events. In any case, in the context of this guide, Event is the record of the change on a particular data in the domain.

Events are processed by computer systems since they were invented, and throughout the history, systems responsible for that were given different names and different methodologies were employed. It wasn't until the 90's though, that a more focused work started on EDA (Event Driven Architecture) with a more formal definition on the requirements and goals for event processing. Old messaging systems started to change to address such requirements and new systems started to be developed with the single purpose of event processing. Two trends were born under the names of Event Stream Processing and Complex Event Processing.

In the very beginnings, Event Stream Processing was focused on the capabilities of processing streams of events in (near) real time, where the main focus of Complex Event Processing was on the correlation and composition of atomic events into complex (compound) events. An important (maybe the most important) milestone was the publishing of the Dr. David Luckham's book "The Power of Events" in 2002. In the book, Dr Luckham introduces the concept of Complex Event Processing and how it can be used to enhance systems that deal with events. Over the years, both trends converged to a common understanding and today these systems are all referred as CEP systems.

In other words, CEP is about detecting and selecting the interesting events (and only them) from an event cloud, finding their relationships and inferring new data from them and their relationships.

Our implementation of complex event processing has focused on developing a rule engine that supports this type of reasoning in order to assist a higher level engine. It is based on the Fusion packet of the Drools suite that is explained in 3.4.1.2.

2.5 Task Automation Services

Since the beginning of the computers era, the term automation has been one of the main topics to keep in mind. Automation means to use a control system in order to save time, labor and effort, by programming a sequence of actions, also called task, to do it without any interaction with the user.

As we all know, a number of prominent web sites, mobile and desktop applications feature rule-based task automation. Typically, these services provide users the ability to define which action should be executed when some event is triggered. Some examples of this simple task automation could be “When I am tagged in Facebook, send me an email”, “When I am within 500 meters from this place, check-in in Foursquare”, or “Turn Bluetooth on when I leave work”. We call them Task Automation Service (TAS). Some TASs allow

users to share the rules they have developed, so that other users can reuse these tools and tailor them to their own preferences.

Task Automation is a rising area: recently lots of different web services and mobile-apps focus their business on this topic. Although the concept is not new, several changes on the state of technology support the success of these services and applications. Among them, the massive publishing of third-party APIs on the Cloud, providing access to their services is a key factor that unchained this mushrooming. To illustrate this, we will analyse three of them: IFTTT, Zapier and Cloudwork.

2.5.1 IFTTT

IFTTT is web service that lets you create when-then rules with one simple statement: *if this then that*. If we had to highlight only one feature of this service we would say simplicity. It offers a user-friendly interface that allows the users compose the rules in a very simple way. As we can see in figure 2.2, they offer an easy way to put the internet to work for you.

Figure 2.2: IFTTT slogan at its landing page

In IFTTT, each service is called a channel, for example: Facebook channel wraps all the possible interactions with the facebook service. Each channel also has triggers and actions. A trigger is the when part or left part of the rule, which means, is the configurable event that triggers the rule, for example: a new photo has been posted on facebook and I am tagged on it. On the other hand, actions are the right side of the rule, the actions themselves, for example: posting something to facebook. Finally, both actions and triggers have parameters, that in IFTTT have been called ingredients.

Composing a rule means: selecting a trigger and setting its ingredients and selecting an action and configuring its ingredients. Each rule in IFTTT is called recipe and are public and accessible to every user by default representing an important growing community.

IFTTT is the proof of the importance of this type of platforms: in its short life, this

company has raised 8.5[1] Millions of dollars and it is in continuous development and growth. With over 8000 recipes and 71 channels by now, it is, by now, the most active task automation service.

2.5.2 Zapier

In a similar way to IFTTT, Zapier is able to automate when-then rules and provides an easy to use interface. In this service, rules are called zaps and are composed by triggers and actions but are the same as the ones explained at the previous subsection.

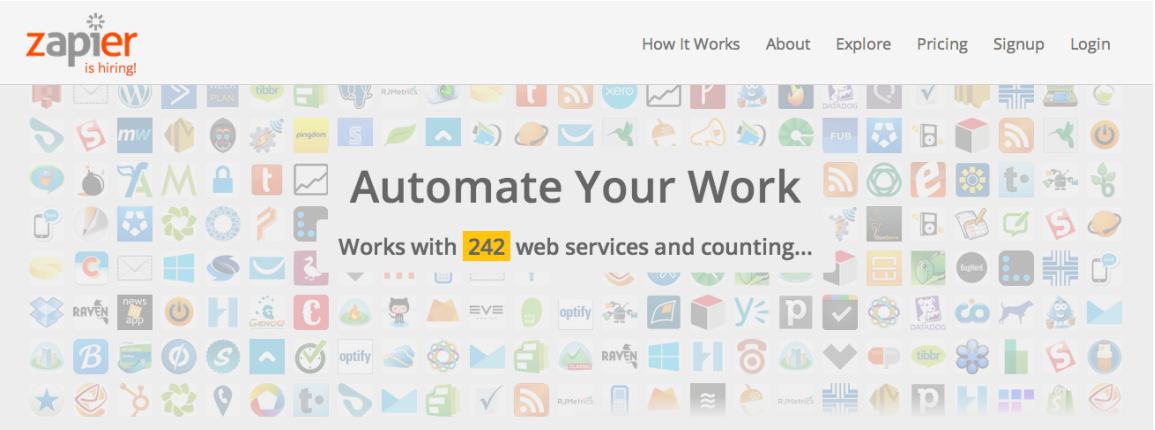

Figure 2.3: Zapier's landing page

It supports more services than IFTTT but it has been less successful, maybe due to its pricing, which is one of the main differences with the previous one. While you can set unlimited free rules in IFTTT, you have to pay monthly for them in zapier . As we can see in figure 2.3, its main strength remains in the number of channels or applications it supports.

At any rate, Zapier is also an important upcoming service to watch, it has raised 1.2 Millions of dollars[2] until the date and it is only two years old. It has been called "An IFTTT for business users" which seems to mean that this two platforms competing for the task automation hegemony can leave together, each one focusing in a type of public.

2.5.3 CloudWork

CloudWork is an integration as a service platform, iPaaS, that allows anyone to build connections between business and social media apps. It has been created more recently that the others and supports less services but is also a service to consider.

[1] http://www.crunchbase.com/company/if-this-then-that
[2] http://www.crunchbase.com/company/zapier

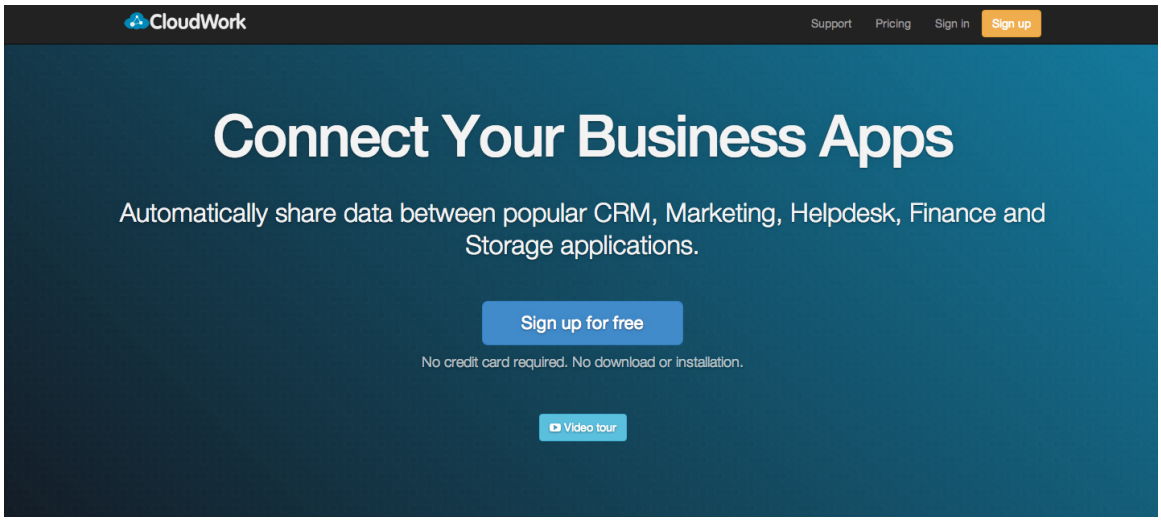

Figure 2.4: Cloudwork's landing page

As a result businesses are able to automate repetitive tasks and receive important notifications in a single feed. CloudWork saves time and increases productivity with just a few clicks. CloudWork takes an approach to integration. It connects data from previously siloed cloud apps (Google Apps, Zoho, Highrise, Capsule CRM, Zendesk, Freshbooks, MailChimp, Salesforce, Desk.com, Campaign Monitor, Twitter etc.) to automate business processes and deliver notifications where you need it.

CloudWork offers a large catalogue of pre-built tried and tested integrations. No technical skills, no big upfront investment or complicated setup are required, like the previous platforms. As they tell in their website: CloudWork is the modern operations center for "all-in cloud" businesses.

This is an example of an only-business oriented platform as figure 2.4 suggests. Unlike Zapier, Cloudwork only focus in the services that have application to productivity or work. It has also raised an important amount of funding: 1.2[3] Millions of dollars during its notorious short life.

[3] `http://www.crunchbase.com/company/cloudwork`

Chapter 3

Enabling Technologies

"The expectations of life depend upon diligence; the mechanic that would perfect his work must first sharpen his tools."

— Chinese proverb

3.1 Introduction

Under this chapter, we describe the technologies that have been chosen for the development of this project. Each technology explanation, far from being exhaustive, focuses in those features that are relevant to the project. This chapter is not supposed to replace the formal specifications of each technology but to highlight the most important features of each technology.

3.2 GSN

The GSN project[9], acronym for *Global Sensor Network*, is a software middleware designed to facilitate the deployment and programming of sensor networks. GSN is a Java environment that runs on one or more computers composing the backbone of the acquisition network. A set of wrappers allow to feed live data into the system. Then, the data streams are processed according to XML specification files. The system is built upon a concept of sensors (real sensors or virtual sensors, that is a new data source created from live data)

that are connected together in order to built the required processing path. For example, one can imagine an anemometer that would sent its data into GSN through a wrapper (various wrappers are already available and writing new ones is quick), then that data stream could be sent to an averaging mote, the output of this mote could then be split and sent for one part to a database for recording and to a web site for displaying the average measured wind in real time. All of this example could be done by editing only a few XML files in order to connect the various motes together.

The original goal of GSN was to provide a reusable software platform for the processing of data streams generated by wireless sensor networks. Once this goal was achieved, and was later reoriented towards a generic stream processing platform.

GSN acquires data, filters it with an intuitive, enriched SQL syntax, runs customisable algorithms on the results of the query, and outputs the generated data with its notification subsystem.

GSN can be configured to acquire data from various data sources. The high number of data sources in GSN allows for sophisticated data processing scenarios. GSN also offers advanced data filtering functionalities through an enhanced SQL syntax.

3.2.1 GSN Architecture

GSN uses a container-based architecture for hosting virtual sensors. Similar to application servers, GSN provides an environment in which sensor networks can easily and flexibly be specified and deployed by hiding most of the system complexity in the GSN Server. Using the declarative specifications, virtual sensors can be deployed and reconfigured in GSN Servers at runtime. Communication and processing among different GSN Servers is performed in a peer-to-peer style through standard Internet and Web Services protocols. By viewing GSN Servers as cooperating peers in a decentralized system, we tried avoid some of the intrinsic scalability problems of many other systems which rely on a centralized or hierarchical architecture. Targeting a ”Sensor Internet” as the long-term goal we also need to take into account that such a system will consist of ”Autonomous Sensor Systems” with a large degree of freedom and only limited possibilities of control, similarly as in the Internet. Figure 3.1 shows the layered architecture of a GSN Server that is detailed below.

Each GSN server hosts a number of virtual sensors it is responsible for. The virtual sensor manager (VSM) is responsible for providing access to the virtual sensors, managing the delivery of sensor data, and providing the necessary administrative infrastructure. The VSM has two subcomponents: The life-cycle manager provides and manages the resources

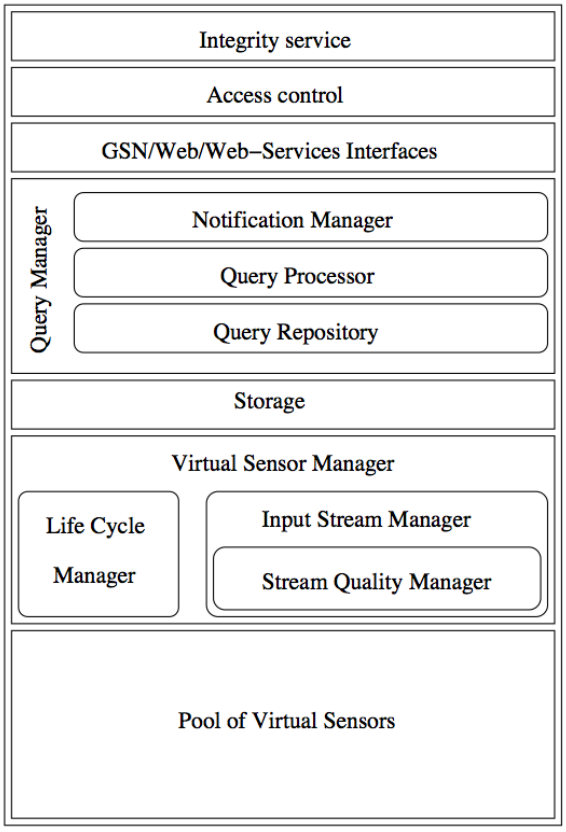

Figure 3.1: GSN architecture

provided to a virtual sensor and manages the interactions with a virtual sensor. The input stream manager is responsible for managing the streams, allocating resources to them, and enabling resource sharing among them while its stream quality manager subcomponent ensures the QoS of streams.

All data from/to the VSM passes through the storage layer which is in charge of providing and managing persistent storage for data streams. Query processing in turn relies on all of the above layers and is done by the query manager which includes the query processor being in charge of SQL parsing, query planning, and execution of queries . The query repository manages all registered queries (subscriptions) and defines and maintains the set of currently active queries for the query processor. The notification manager deals with the delivery of events and query results to registered, local or remote virtual sensors.

The top three layers of the architecture deal with access to the GSN server like access via HTTP and access via other GSN servers.

3.2.2 Data acquisition: GSN Wrappers

GSN can receive data from various data sources. This is done by using so called wrappers. They are used to encapsulate the data received from the data source into the standard GSN data model, called a StreamElement. A StreamElement is an object representing a row of a SQL table. Each wrapper is a Java class that extends the AbstractWrapper parent class. Usually a wrapper initializes a specialized third-party library in its constructor. It also provides a method which is called each time the library receives data from the monitored device. This method will extract the interesting data, optionally parse it, and create one or more StreamElement(s) with one or more columns.

From this point on, the received data has been mapped to a SQL data structure with fields that have a name and a type. GSN is then able to filter this using its enhanced SQL-like syntax.

3.2.3 Data processing: Virtual sensors

The key abstraction in GSN is the virtual sensor. Virtual sensors abstract from the implementation details of the data source to sensor data and correspond either to a data stream received directly from sensors or to a data stream derived from other virtual sensors. The specification of a virtual sensor provides all necessary information required for deploying and using it, including:

1. metadata used for identification and discovery
2. the details of the data streams which the virtual sensor consumes and produces
3. an SQL-based specification of the stream processing (filtering and integration) performed in a virtual sensor
4. the processing class which performs the more advanced and complex data processing (if needed) on the output stream before releasing it
5. functional properties related to persistence, error handling, life-cycle, management, and physical deployment.

To support rapid deployment, the virtual sensors are provided in human readable declarative forms (XML). We have developed three of them as an example and are fully recorded at appendix B.

3.3 Semantic Rule Description: EWE ontology

Evented WEb Ontology (EWE)[10] is a standardized data schema (also referred as "ontology" or "vocabulary") designed to describe elements within Task Automation Services such as the ones detailed in section 2.5 enabling rule interoperability. According to [**?**], the main goals of the EWE ontology to achieve are:

- Provide a common model to represent TAS's rules so that it enable rule interoperability
- Provide a base vocabulary for building domain specific vocabularies e.g. Twitter Task Ontology or Evernote Task Ontology
- Enable semantic rule enhancement with the connection to Linked Open Data (LOD) cloud
- Enable loose-coupled semantic rules -rules that do not bind at leas one of the specific channels to use on design time and leave it to the engine to decide the best match at runtime.

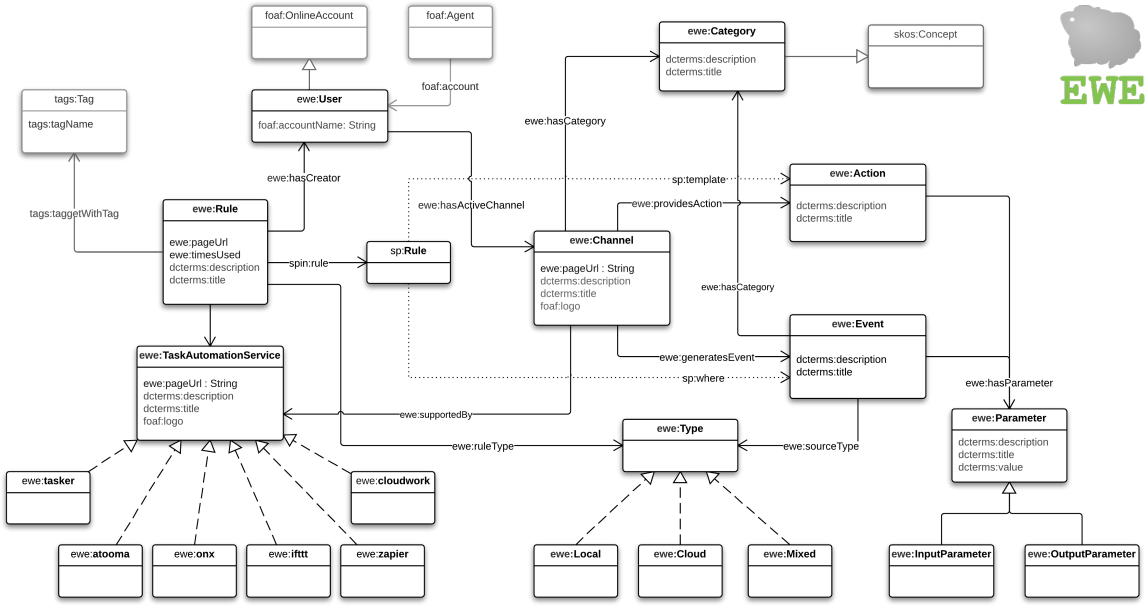

Figure 3.2: EWE ontology at a glance

A very basic example below shows a single Channel described using EWE vocabulary. It defines a new subclass of Channel that outlines how GoogleTalk Service works. As defined below, Gmail Channel generates two events and provide one single action. These are "Any

new email" and "New email from" events and "Send an email" action (their RDF description is not shown in the example below for sake of simplicity).

Listing 3.3 is a real example of class definition scrapped from ifttt.com. Note that the current version of channel description at ifttt.com may differ from the description shown here, due to ifttt is in continuous expansion, remodeling their channels, so new events or actions may have been added since this example was written down.

```
    <owl:Class rdf:about="https://ifttt.com/gmail">
    <rdfs:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe
        /ns#Channel"/>

    <!-- Administrative properties -->
    <dcterms:title>Gmail</dcterms:title>
    <dcterms:description>
      Gmail is a free, advertising-supported webmail, POP3, and IMAP
          service provided by Google.
    </dcterms:description>
    <foaf:logo>https://ifttt.com/images/channels/gmail_lrg.png</foaf:
        logo>

    <!-- Categorization -->
    <ewe:hasCategory rdfs:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns#email">

    <!-- Functionalities -->
    <ewe:generatesEvent rdf:resource="https://ifttt.com/channels/gmail/
        triggers/85"/>
    <ewe:generatesEvent rdf:resource="https://ifttt.com/channels/gmail/
        triggers/86"/>
    <ewe:hasAction rdf:resource="https://ifttt.com/channels/gmail/
        actions/34"/>
  </owl:Class>
```

Listing 3.1: Channel implementation in EWE ontology

In the former example, events and actions are included as external references. This is the preferred way for describing channels, since it is easier to read, and offers a more modular view of the model. However, as in any other RDF graph, RDF entities can be nested, thus we can include the Event or Action definition nested within the Channel definition. We could even add them as black entities or nodes if there is no need to reference them from

the outside (this is without relating them to the Channel that defines them), although not common and it is discouraged.

The example below, in listing 3.3 ,provides the description of the "New Email from" Event referenced at the Gmail Channel definition from the example above. The event shown presents one input parameter -the email address of the sender- and three output parameters -the email address of the sender, the subject of the email, and the body of the message in plain text.

In this case, parameters are included as nested elements instead of being referenced as external resources. Moreover, the reference-to-external-resources is also an acceptable approach.

```
    <owl:Class rdf:about="https://ifttt.com/channels/gmail/triggers/86
        ">
  <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2012/9/
      ewe.owl#Event"/>
  <dcterms:title>New email from</dcterms:title>
  <dcterms:description>
    This Trigger fires every time a new email arrives in your inbox
       from the address you specify.
  </dcterms:description>

  <!-- Input Parameters -->
  <ewe:hasInputParameter>
    <ewe:InputParameter>
      <dcterms:title>EmailAddress</dcterms:title>
    </ewe:InputParameter>
  </ewe:hasInputParameter>

  <!-- Output Parameters -->
  <ewe:hasOutputParameter>
    <ewe:OutputParameter>
      <dcterms:title>FromAddress</dcterms:title>
      <dcterms:description>Email address of sender.</dcterms:
          description>
    </ewe:OutputParameter>
  </ewe:hasOutputParameter>
  <ewe:hasOutputParameter>
    <ewe:OutputParameter>
      <dcterms:title>Subject</dcterms:title>
      <dcterms:description>Email subject line.</dcterms:description>
    </ewe:OutputParameter>
  </ewe:hasOutputParameter>
```

```
    <ewe:hasOutputParameter>
      <ewe:OutputParameter>
        <dcterms:title>BodyPlain</dcterms:title>
        <dcterms:description>Plain text email body.</dcterms:description>
      </ewe:OutputParameter>
    </ewe:hasOutputParameter>

  </owl:Class>
```

Listing 3.2: Event implementation in EWE ontology

The former example uses the properties ewe:hasInputParameter and ewe:hasOutputParameter to reference the Input and Output Parameters. However, the parent hasParameter could have also been used with OutputParameter or InputParameter. When the inference engine is available, both approaches are equivalent and any query that matches one of them will also match the other. Nevertheless, when it is not, the procedure shown is more explicit, thus preferred.

3.4 Rule Engines

In order to process the rules, handling events and generating actions, we need a rule engine. The one that has been implemented for this project uses two different technologies: Drools Expert rule engine and SPIN inferencing notation. The first one is a powerful engine that allows us to effectively trigger the rules and provide complex event processing, and the other one is responsible of the semantic side of our rule engine.

3.4.1 Drools

Drools has been our weapon of choice to handle low level events and provide complex event processing. In the next two subsection we explain the two main highlights that has make us decide for this technology, the Rete algorithm that it uses for triggering the rules and his CEP packet, Drools Fusion

3.4.1.1 Rete algorithm

The Rete algorithm[11] describes how the Rules in the Production Memory are processed to generate an efficient discrimination network. In non-technical terms, a discrimination network is used to filter data as it propagates through the network. The nodes at the top of the network would have many matches, and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. We can see the basic nodes at figure 3.3: root, 1-input, 2-input and terminal.

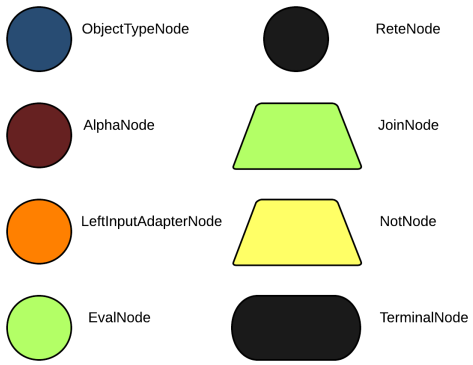

Figure 3.3: Rete type of nodes

The root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode. The purpose of the ObjectTypeNode is to make sure the engine doesn't do more work than it needs to. For example, say we have 2 objects: Account and Order. If the rule engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the engine should only pass the object to the nodes that match the object type. The easiest way to do this is to create an ObjectTypeNode and have all 1-input and 2-input nodes descend from it. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object. In Drools when an object is asserted it retrieves a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class; if this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an instanceof check.

ObjectTypeNodes can propagate to AlphaNodes, LeftInputAdapterNodes and BetaNodes. AlphaNodes are used to evaluate literal conditions. Although the original paper only covers equality conditions, many RETE implementations support other operations. For example, Account.name == "Mr Trout" is a literal condition. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an

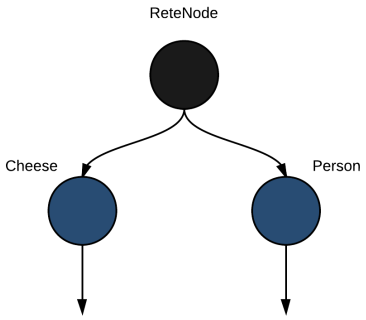

Figure 3.4: ObjectTypeNodes

application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode. Figure 3.5 shows the AlphaNode combinations for Cheese(name == "cheddar", strength == "strong"):

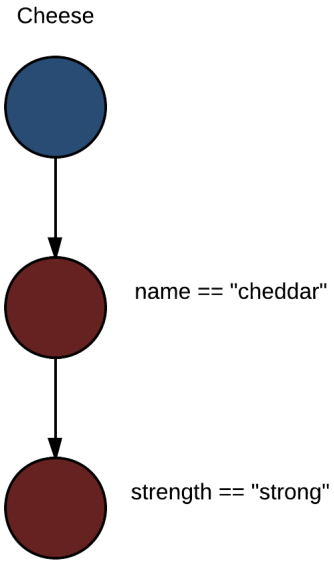

Figure 3.5: AlphaNodes

Drools extends Rete by optimizing the propagation from ObjectTypeNode to AlphaNode using hashing. Each time an AlphaNode is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap,thereby avoiding unnecessary literal checks.

There are two two-input nodes, JoinNode and NotNode, and both are types of BetaNodes. BetaNodes are used to compare 2 objects, and their fields, to each other. The objects may be the same or different types. By convention we refer to the two inputs as left and right. The left input for a BetaNode is generally a list of objects; in Drools this is a Tuple. The right input is a single object. Two Nodes can be used to implement 'exists' checks. BetaNodes also have memory. The left input is called the Beta Memory and remembers all incoming tuples. The right input is called the Alpha Memory and remembers all incoming objects. Drools extends Rete by performing indexing on the BetaNodes. For instance, if we know that a BetaNode is performing a check on a String field, as each object enters we can do a hash lookup on that String value. This means when facts enter from the opposite side, instead of iterating over all the facts to find valid joins, we do a lookup returning potentially valid candidates. At any point a valid join is found the Tuple is joined with the Object; which is referred to as a partial match; and then propagated to the next node.

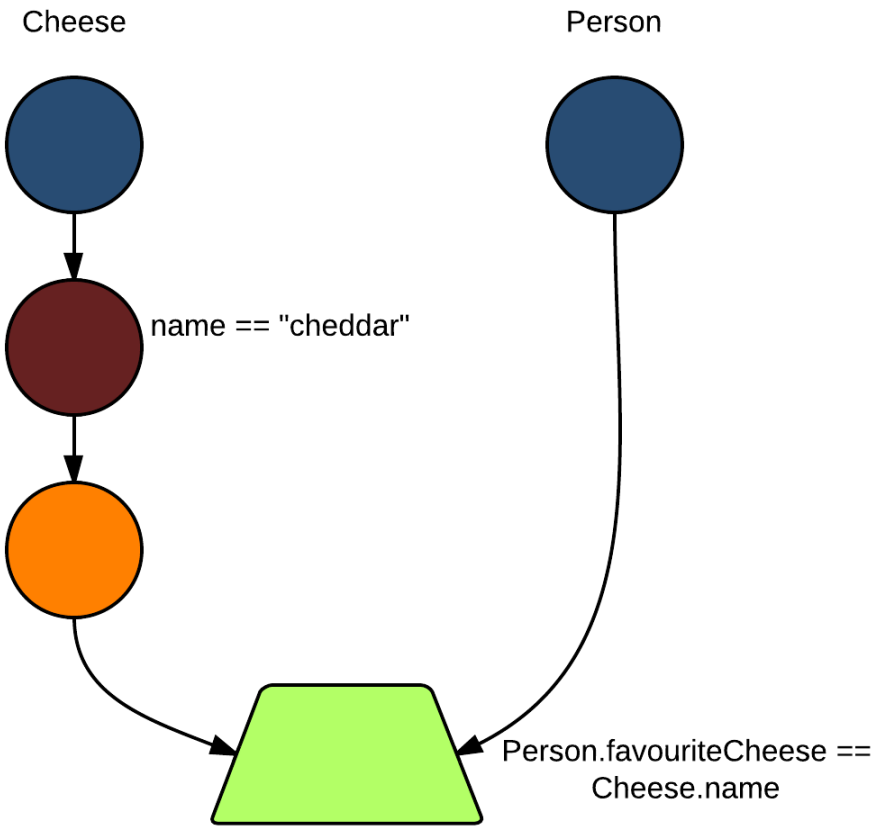

Figure 3.6: JoinNode

To enable the first Object, in the above case Cheese, to enter the network we use a LeftInputNodeAdapter - this takes an Object as an input and propagates a single Object Tuple.

Terminal nodes are used to indicate a single rule having matched all its conditions; at this point we say the rule has a full match. A rule with an 'or' conditional disjunctive connective results in subrule generation for each possible logically branch; thus one rule can have multiple terminal nodes.

Drools also performs node sharing. Many rules repeat the same patterns, and node sharing allows us to collapse those patterns so that they don't have to be re-evaluated for every single instance. The following two rules share the first pattern, but not the last:

Listing 3.3: Node sharing first rule

```
rule
  when
    Color(color : name == "blue")
    person : Person(favouriteColor == blue)
  then
    System.out.println(person.getName() + " likes blue");
end
```

Listing 3.4: Node sharing second rule

```
rule
  when
    Color(color : name == "blue")
    person : Person(favouriteColor != blue)
  then
    System.out.println(person.getName() + " does not like blue");
end
```

As you can see in Figure 3.7, the compiled Rete network shows that the alpha node is shared, but the beta nodes are not. Each beta node has its own TerminalNode. Had the

second pattern been the same it would have also been shared.

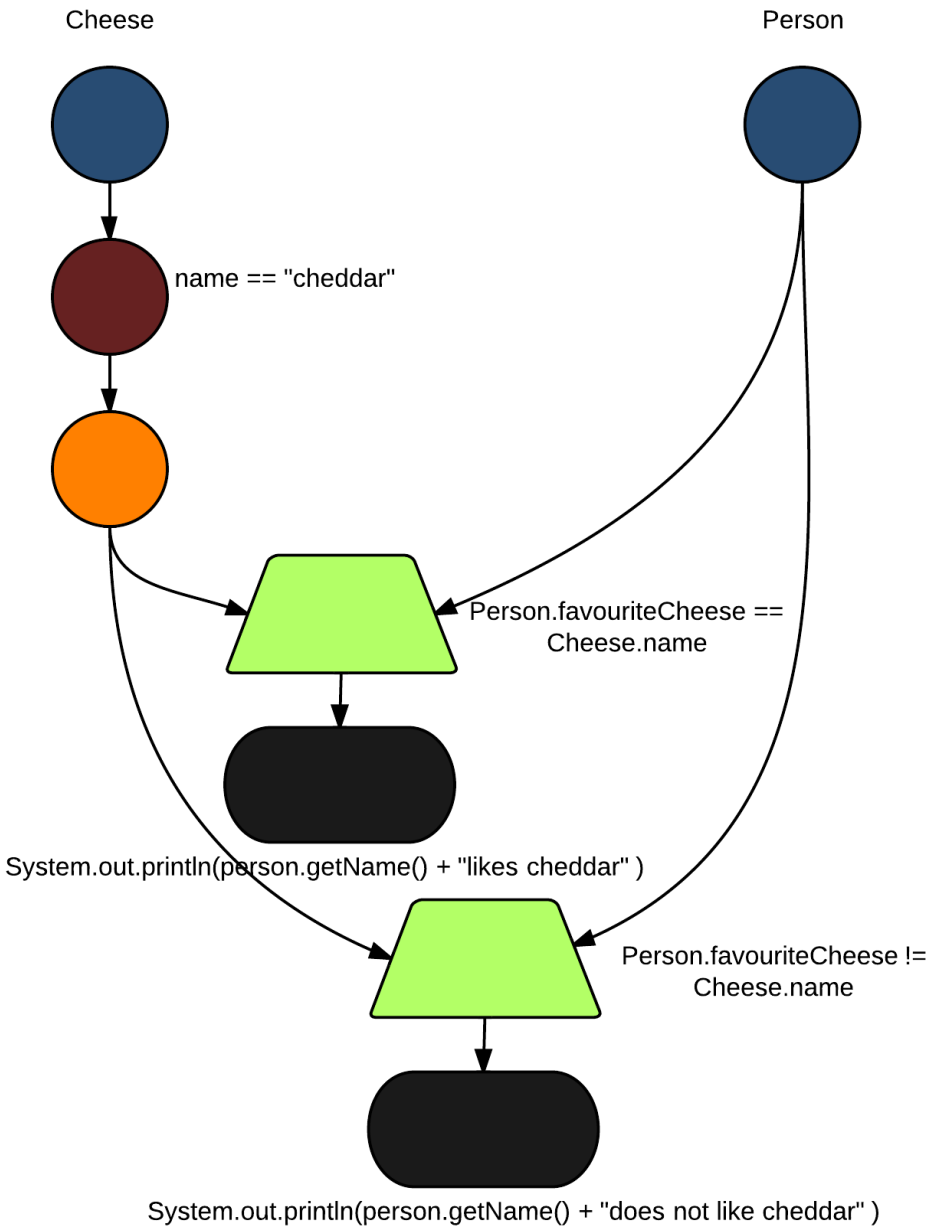

Figure 3.7: NodeSharing

3.4.1.2 Drools Fusion

Drools Fusion is the module responsible for adding event processing capabilities into the platform, supporting Complex Event Processing. This module has a defined a set of goals to be achieved in order to support Complex Event Processing appropriately:

- Support Events, with their proper semantics, as first class citizens.
- Allow detection, correlation, aggregation and composition of events.
- Support processing of Streams of events.
- Support temporal constraints in order to model the temporal relationships between events.
- Support sliding windows of interesting events.
- Support a session scoped unified clock.
- Support the required volumes of events for CEP use
- cases.
- Support to (re)active rules.
- Support adapters for event input into the engine (pipeline).

The above list of goals are the requirements not covered by Drools Expert itself, but they are also the main features that have made us to choose this technology. This way, Drools Fusion is born with enterprise grade features like Pattern Matching, that is paramount to a CEP product, but that is already provided by Drools Expert. In the same way, all features provided by Drools Fusion are leveraged by Drools Flow (and vice-versa) making process management aware of event processing and vice-versa.

3.4.2 SPIN, the semantic rule engine

SPIN[1] is a W3C Member Submission that has become the de-facto industry standard to represent SPARQL[**?**] rules and constraints on Semantic Web models. SPIN also provides meta-modelling capabilities that allow users to define their own SPARQL functions and query templates [**?**, **?**]. Finally, SPIN includes a ready to use library of common functions[**?**].

[1] `http://spinrdf.org/`

SPIN is a way to represent a wide range of business rules. With SPIN, rules are expressed in SPARQL. SPARQL is a well-established W3C standard implemented by many industrial-strength RDF APIs and all databases. This means that rules can run directly on RDF data without a need for materialization. An overview of the various technologies involved in this family of languages is provided in figure 3.8

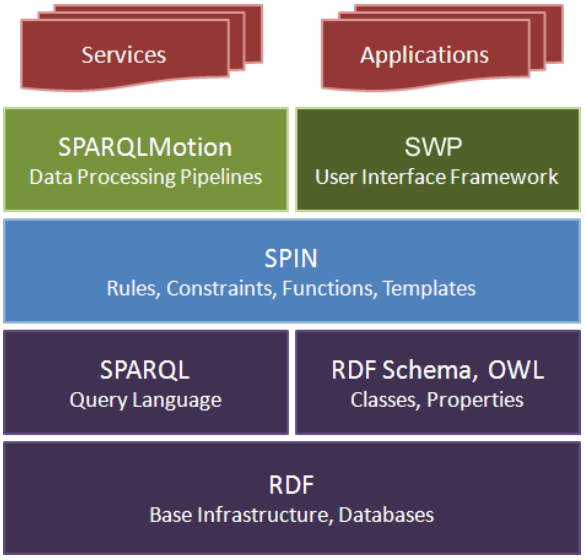

Figure 3.8: spin-stack

SPIN can be used to:

- Calculate the value of a property based on other properties - for example, area of a geometric figure as a product of its height and width, age of a person as a difference between today's date and person's birthday, a display name as a concatenation of the first and last names.
- Isolate a set of rules to be executed under certain conditions - for example, to support incremental reasoning, to initialize certain values when a resource is first created, or to drive interactive applications.

These rules are implemented using SPARQL CONSTRUCT or SPARQL UPDATE requests (INSERT and DELETE). SPIN Templates also make it possible to define such rules in higher-level domain specific languages so that rule designers do not need to work with SPARQL directly.

Another common need in applications is to check validity of the data. For example, you may want to require that a field is entered and/or that the string entered follows your format requirements.

SPIN offers a way to do constraint checking with closed world semantics and automatically raise inconsistency flags when currently available information does not fit the specified integrity constraints. Constraints are specified using SPARQL ASK or CONSTRUCT queries, or corresponding SPIN Templates.

SPIN combines concepts from object oriented languages, query languages, and rule-based systems to describe object behavior on the web of data. One of the key ideas of SPIN is to link class definitions with SPARQL queries to capture rules and constraints that formalize the expected behavior of those classes. To do so, SPIN defines a light-weight collection of RDF properties.

Chapter 4

Requirements Analysis

"The ultimate authority must always rest with the individual's own reason and critical analysis."

—Dalai Lama

In order to make sure the final solution will suit in real life applications, it is important to perform requirements analysis. The main goal of this phase is to stablish all the demands that must be fulfilled by each module using the technologies described in the previous chapter.

We present a set of use cases that aims to cover all the scenarios and possible variables involved in the design of a semantic platform for task automation. Then, at the end of this chapter, we have collected all the requirements that we have concluded from this analysis.

4.1 Actor library

Actor ID	Role	Description
ACT-1	User	The final user
ACT-2	Third party web service	Google Calendar Service
ACT-3	Web service handler	Google Calendar Handler
ACT-4	Physical sensor handler	Dni-e reader handler
ACT-5	Sensor network	Sensor network

ACT-6	User	User B, second user involved in the use case.
ACT-7	User	User C, third user involved in the use case.
ACT-8	Rule engine	DrEWE's rule engine
ACT-9	User	Administrator with permission to create rules and modify the system
ACT-10	Rule engine	CEP rule engine, part of ACT-8
ACT-11	Rule engine	Semantic rule engine, part of ACT-8
ACT-12	Web service handler	Twitter handler, the one in charge of communicate with the twitter service

Table 4.1: List of actors involved in the uses cases

4.2 Use cases

4.2.1 UC-1: Schedule a meeting (third party service)

As we have discussed before, our platform makes use of external APIs in the same way IFTTT and Zapier does. This use case relates the sequence of scheduling a meeting via an external services: Google Calendar.

ID	UC-1
Name	Schedule a meeting.
Description	The user schedules a meeting through google calendar's interface.
Actors	User (ACT-1) and Google Calendar (ACT-2)
Preconditions	-
Basic Flux	1. The user navigates to google calendar's web site on his web browser. 2. The user sign in google and authenticates himself. 3. The user schedules a meeting through google calendar's interface and invite other two attendees to join the meeting.

Alternative Flux	3. The user reschedules a meeting already programmed and sets a new hour.
Post-conditions	A meeting has been scheduled

Table 4.2: Use case for scheduling a meeting

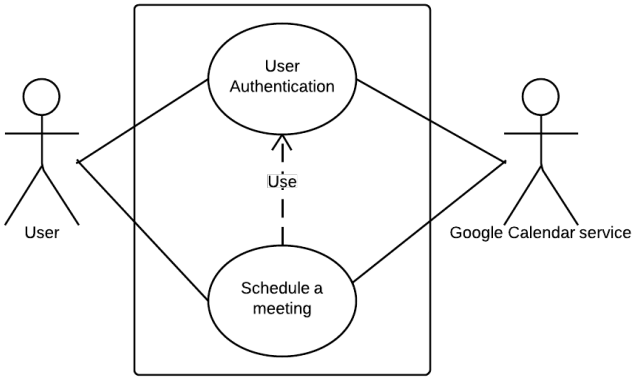

Figure 4.1: Diagram representation for UC-1.

4.2.2 UC-2: New meeting detected

Once the new meeting has been inserted, the specific web handler will notice this new entry by periodically checking the calendar. After that, it will wrap the new entry in a convenient way and push it to the rule engine, which will receive it, convert it to an event and push it into its knowledge base.

ID	UC-2
Name	Detecting scheduled meeting and pushing it to rule engine.
Description	The new meeting is detected, pushed into the rule engine and included in its knowledge base.
Actors	Google Calendar Handler (ACT-3) and Rule engine (ACT-8)
Preconditions	A meeting has been scheduled.

Basic Flux	1. The event is detected and encapsulated in a message. 2. The message is sent to the rule engine. 3. The rule engine receives the message and creates an event. 4. The event is inserted into its knowledge base.
Alternative Flux	4. The meeting date is posterior to current date, the event is rejected.
Post-conditions	New meeting event has been included.

Table 4.3: Use case for detect and push a new meeting

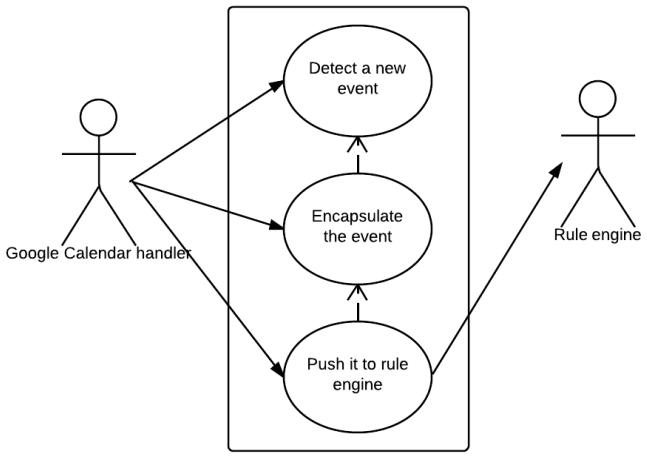

Figure 4.2: Diagram representation for UC-2.

4.2.3 UC-3: Inserting dni-e at meeting entrance

In order to have a reliable log of the incoming employers to the laboratory, we have installed a dni-e reader and have made mandatory to insert the dni-e when the enter the lab. This log is used by our the platform to check the arrival of the meeting's attendees.

ID	UC-3
Name	Introducing dni-e at entrance
Description	An attendee arrives and introduces his dni at the dni-e reader at the entrance
Actors	User A (ACT-1), dni-e reader handler (ACT-4) and sensor network (ACT-5)
Preconditions	The module in charge of handling the dni reader is up and running and the scheduled meeting starts in ten minutes or less.
Basic Flux	1. The attendee arrives at the meeting and introduce its dni-e in the reader the entrance. 2. The dni-e handler detects a change in dni-e reader's log, packets the information in a message and push it to the sensor network.
Alternative Flux	-
Post-conditions	An attendees has arrived in time and a new entry for this event has been recorded at the sensors network.

Table 4.4: Use case for an attendee arriving at the meeting

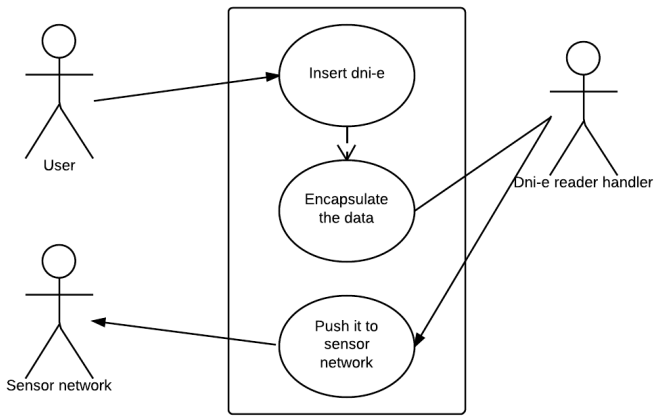

Figure 4.3: Diagram representation for UC-3.

4.2.4 UC-4: Meeting attendees arrived

The three actors that were summoned to the meeting, have just entered the laboratory and the meeting is about to be started.

ID	UC-4
Name	Meeting attendees arrived
Description	As attendees arrive at the scheduled location, they insert their dni-e at the id reader at the entrance in order to confirm their assistance. All the attendees supposed to be at the meeting arrive.
Actors	User A (ACT-1), user B (ACT-6), user C (ACT-7), dni-e reader handler (ACT-4) and sensor network (ACT-5)
Preconditions	The module in charge of handling the dni reader is up and running and the scheduled meeting starts in ten minutes or less.
Basic Flux	1. User A arrives at the meeting, introduces its dni-e in the reader the entrance and the information is pushed into the sensors network. 2. User B arrives at the meeting, introduces its dni-e in the reader the entrance and the information is pushed into the sensors network. 3. User C arrives at the meeting, introduces its dni-e in the reader the entrance and the information is pushed into the sensors network.
Alternative Flux	-
Post-conditions	The three attendees have arrived in time

Table 4.5: Use case for all the attendees arriving at the meeting

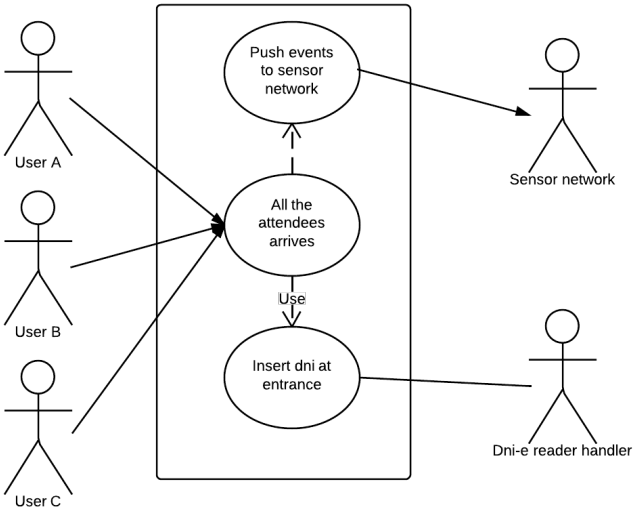

Figure 4.4: Diagram representation for UC-4.

4.2.5 UC-5: Retrieving events from the sensor network

This use case is related to the communication between the sensor network and the rule engine, when any new entries are detected at the sensor network, they are correctly handled and pushed to the rule engine.

ID	UC-4
Name	Retrieving events from the sensor network
Description	The system is continuously checking the sensor network in order to search new events from the physical sensors. It packets them in messages and push them into the rule engine.
Actors	Sensor Network (ACT-5) and Rule Engine (ACT-8)
Preconditions	There are new events in the sensor network
Basic Flux	1. New entries are detected in the sensor network 2. They are packed in message and sent to the rule engine. 3. The rule engine receives them, create events and push them to its knowledge base.

Alternative Flux	3. The event's date is posterior to current date, the event is rejected.
Post-conditions	New events coming from physical sensors have been pushed to the rule engine.

Table 4.6: Use case for pushing events from the sensor network

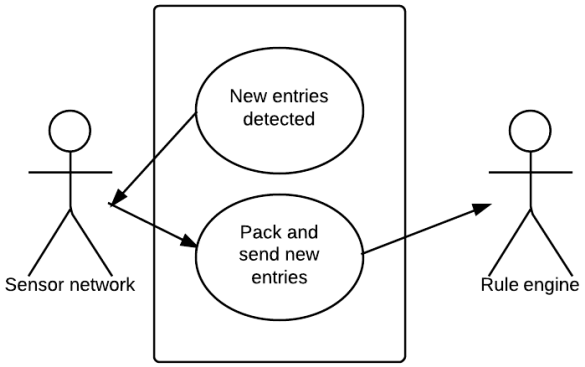

Figure 4.5: Diagram representation for UC-5.

4.2.6 UC-6: Set a CEP rule and a SPIN rule

The administrator sets up rules to be triggered for both engines, CEP and semantic engines. The rules are stored in each rules database and will be loaded by the rule engines in their next checking for new rules.

ID	UC-6
Name	Setting a CEP rule and a SPIN rule.
Description	The system administrator write a rule for complex event processing, another one for the semantic engine and save them.
Actors	Administrator (ACT-9).
Preconditions	The administrator has authenticated himself.

Basic Flux	1. The administrator writes a rule for complex event engine whose output is to generate a high level event. 2. The administrator writes a rule for the semantic engine whose input is the high level event. 3. The administrator saves them in the platform.
Alternative Flux	1. The administrator only writes one rule for complex event engine. 1. The administrator only writes one rule for the semantic engine.
Post-conditions	New rules have been inserted in the platform.

Table 4.7: Use case for creating rules

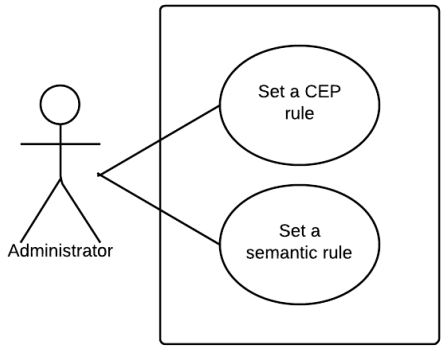

Figure 4.6: Diagram representation for UC-6.

4.2.7 UC-7: CEP rule is triggered

This use case illustrates how the CEP rule engine is in charge of processing low level events and generate high level ones that will be handled by the semantic rule engine, which only accepts this type of events.

ID	UC-7
Name	CEP rule is triggered.
Description	After the three dni events and the meeting event had been introduced, a CEP rule is triggered producing a high level event and pushing it to the semantic rule engine.
Actors	CEP rule engine (ACT-10) and semantic rule engine (ACT-11)
Preconditions	Three dni events and a meeting event has been pushed into the rule engine. There is a rule that matches this pattern and it produces a high level event when triggered.
Basic Flux	1. A CEP rule is triggered. 2. The then part of the rule is executed and it generates a high level event "Meeting started". 3. The high level event is pushed to the semantic rule engine.
Alternative Flux	-
Post-conditions	A high level event has been pushed into the semantic rule engine.

Table 4.8: Use case for triggering a CEP rule

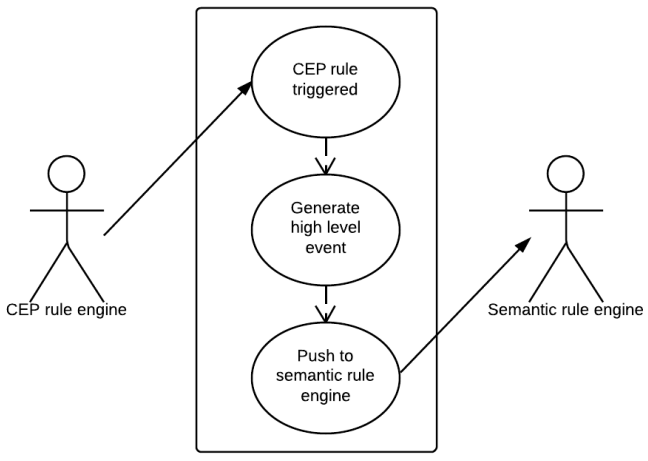

Figure 4.7: Diagram representation for UC-7.

4.2.8 UC-8: SPIN rule is triggered

This one sets an example of how the semantic engine works. This engine processes high level events through when-then rules and is the one in charge of commanding actions to be performed.

ID	UC-8
Name	SPIN rule is triggered.
Description	After pushing the high level event into the semantic rule engine, a rule is triggered and an action intent is thrown.
Actors	Semantic rule engine (ACT-11)
Preconditions	High level event ”Meeting started” has been pushed to the semantic rule engine.
Basic Flux	1. The event matches the condition of one rule. 2. A semantic rule is triggered. 3. The then part of the rule commands an action. 4. An intent for the desired action is thrown.
Alternative Flux	-
Post-conditions	An action intent has been thrown.

Table 4.9: Use case for triggering a SPIN rule

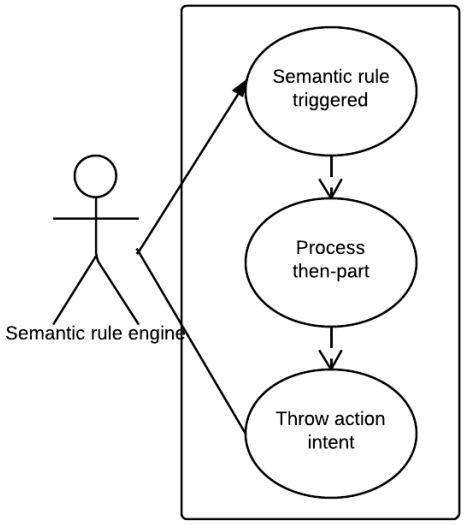

Figure 4.8: Diagram representation for UC-8.

4.2.9 UC-9: Perform an action

Finally, an action is performed. In this case, after occurring a sequence of events, a tweet will be posted in order to inform that the meeting is going on.

ID	UC-9
Name	Perform an action.
Description	In order to perform an action, an intent is thrown by the semantic rule. The correspondent handler catches the intent and perform the action, in this case, post a tweet mentioning the users.
Actors	User A (ACT-1), user B (ACT-6), user C (ACT-7) and twitter handler (ACT-12).
Preconditions	An action intent addressed to the twitter service has been thrown.

Basic Flux	1. The module in charge of handling the twitter service notices that a twitter action has to be performed. 2. The handler compose the tweet from the information inside the intent. 3. A tweet is posted mentioning the three users that attended the meeting.
Alternative Flux	-
Post-conditions	An action has been performed.

Table 4.10: Use case for performing an action

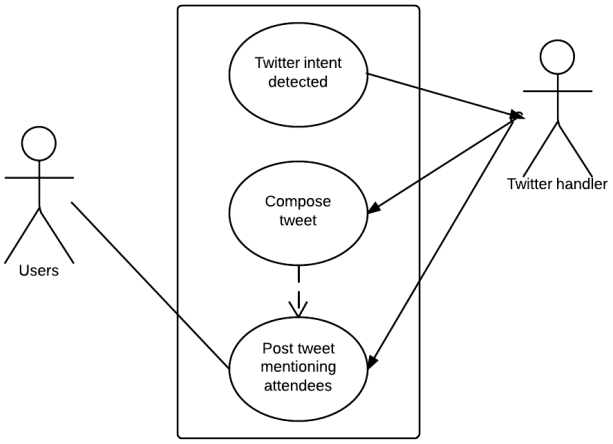

Figure 4.9: Diagram representation for UC-9.

4.2.10 Summary diagram of the use cases

To sum up, we have represented all the use case in only one diagram. This suppose a global use case that include all the previous ones when they happen consequently.

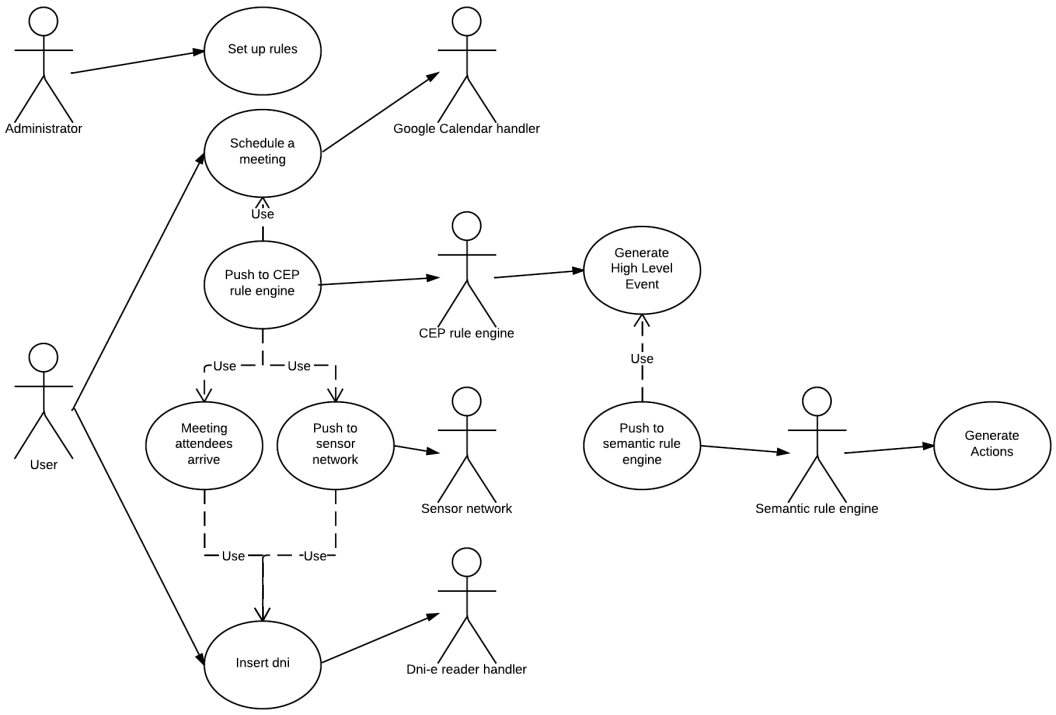

Figure 4.10: Diagram representation for the Global Use Case.

4.3 Requirements summary

After analysing the previous use cases, some clear requirements seem to stand out:

- The architecture must allow the connection of several and heterogeneous modules.
- The architecture ought to be divided in layers or tiers in order to separate the important number of functionalities.
- The platform must have a complex event processing rule engine in order to process low level events and generate high level events.
- The system must provide a semantic rule engine that works in parallel with the CEP engine, to handle the high level events and perform the actions.
- There must be a sensor network that handle the events from physical sensors and ensures they are available from any point of the network or at least, available for the engines.

- Each type of services, both web services and physical sensors, must be wrapped by a handler module independently.
- Language agnostic, the final system will likely incorporate different languages and the corresponding design must allow this.
- Scalable, with an architecture that allows new modules to be inserted.
- Interconnection throw existing web protocols such as HTTP.
- Flexible topology.
- High connectivity between elements.
- Simplicity, to make the development of new components as easy as possible, because the value of this type of platform increases exponentially with the number of implemented services.

To conclude, it is worth to mention that the previous list is not as extensive as other requirements analysis use to be. The main reason of this rely on the novelty of the field of task automation platform, as well as the ever-evolving nature of the technologies involved. In spite of this, this list should be taken as a solid guideline for the design and development of the desired architecture.

Chapter 5

Architecture

"Knowing how things work is the basis for appreciation, and is thus a source of civilized delight."

—William Satire

In this chapter, we address the description of the design phase of the project. Firstly, we present the functional model of the project divided in layers and then, we present a global description of the architecture that can give the reader a general view of the whole project. Under each subsection, we treat each sub-module in depth talking about the functionalities and the purpose of each one of them, giving a special attention to how they work as a part of a synchronized machine with a defined purpose and how they communicate with each other.

On the other hand, we have put special interest to structure the project as independent and complete modules. So each of them can be developed on a separate line of work, be part of other projects and be deployed by themselves providing actual services.

We have also put special effort in implementing a layered structure in order to allocate each responsibility separately in each module. This structure facilitates the development of the project but also, it represents a notable advantage for scalability, meaning that big remarkable changes in one layer makes slight repercussion in next layers.

5.1 Functional model

A three-layered structure is defined: processing layer, transport layer and user layer, as we can see in figure 5.1:

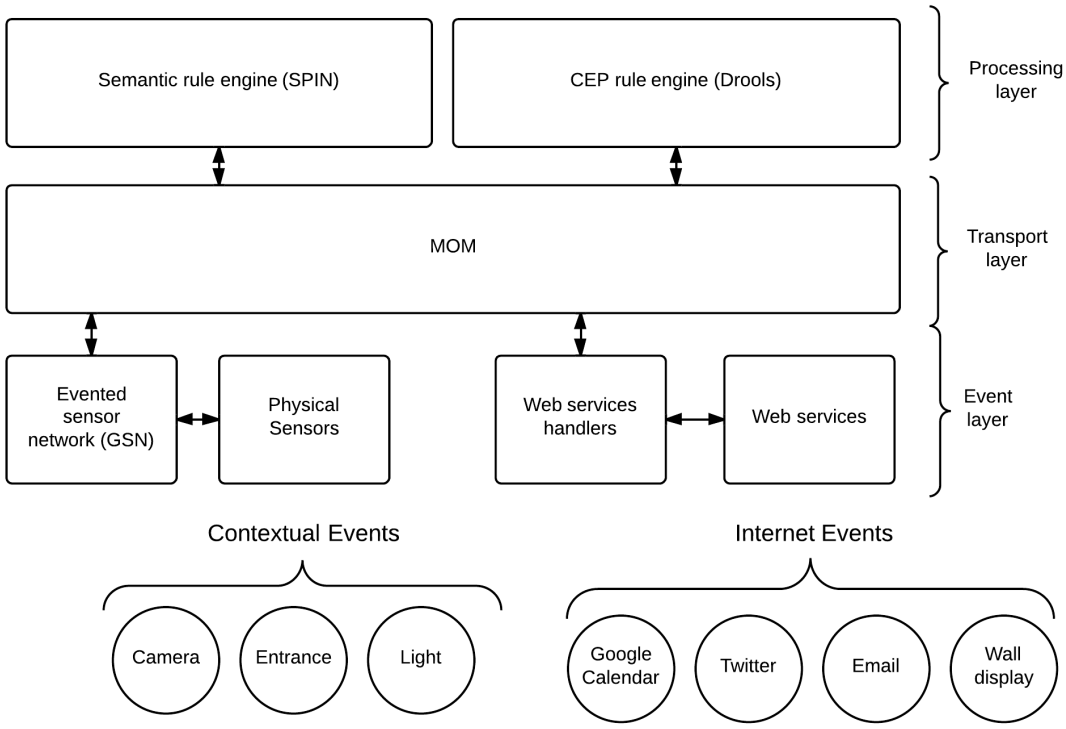

Figure 5.1: Layered structure of DrEWE.

1. *Processing layer*. This layer listens for the events dispatched by the transport layer,it is responsible for generating high level events from the low level ones, handle them to trigger the rules and decide which actions should be performed. On one hand, CEP engine will perform complex rules in order to handle low level events (for example, light-level events) and generate high level events (for example, a light-on event), but on the other hand, SPIN rule engine is responsible for triggering classic when-then rules, with semantic notation. Moreover, they both work together when it comes to low level events scenarios: the CEP engine aggregates the events in high level ones and push them to the semantic engine, which finally trigger the rules.

2. *Transport layer*. Composed by MOM[1] or Message-oriented middleware, as the infrastructure supporting sending and receiving messages between distributed systems.

[1] `http://docs.oracle.com/cd/E19340-01/820-6424/aeraq/index.html`

MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols.This layer is the one responsible for connecting the processing layer with the user and also, this MOM, is in charge of passively receive the events and make them accessible from all over the network, keep a timed log and tag them depending of their network.

3. *Event layer*. Finally we find the layer which will make direct contact with the user. We have two ways to generate events: from the environment and from third-party web services. Starting by the left, we have the Evented network sensor (GSN), that orchestrate the physical sensors and provide several features to distribute these events, and on the right, we can see the web services, handled by a dedicated module that is in charge of addressing and managing third party web service. On the left side, we can see the physical sensors that are already implemented: the raspberry camera, the dni-e[2] card handler and the light sensor. Finally, in the web service side, we have developed GCalendar-DrEWE, that handles a Google Calendar an transform its scheduled meetings in events, and we have also integrated Twitter, Email and a Wall display for meeting purposes. It is worth to mention that these are only examples that illustrate this architecture's potential and what would fit in this layer.

5.2 Global description

The overall platform has been divided into several modules that could be working as a functional whole. In this way, some modules share functionalities of different layers of those described in section 5.1. This division add clarity and loose coupling, which means that each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. As we can see in in Fig. 5.2, there are four main modules or four main classes represented by the figure. These classes group together different implementations with the same functionalities:

1. *Berries-DrEWE*. Under this class, we find the necessary software to handle the physical sensors. Their goal is to retrieve information from the environment, pack it in events and push them to the event network. In fact, Berries-DrEWE is a group of scripts designed to be executed on a Raspberry and both, generate events and perform actions related to sensors. Until the date, we have developed sensor handlers related to the light level, the dni reader and the camera, but these are only examples. As we can

[2] `http://www.dnielectronico.es/`

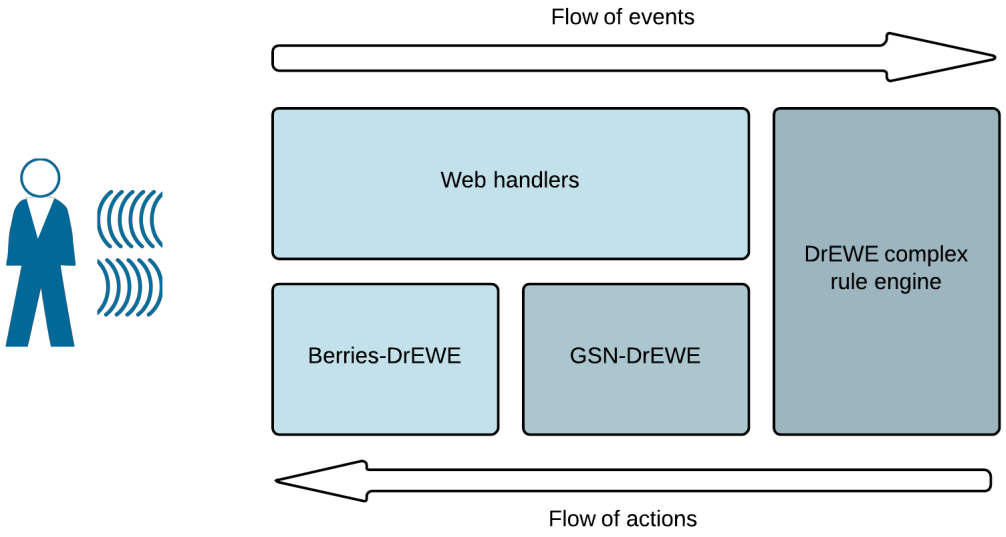

Figure 5.2: Flow of events and actions.

see in chapter 7, there are more physical sensor handlers or event generators to be implemented in a further work.

2. *Web handlers.* In a similar way to the previous one, Web handlers group together as the module with the necessary software to both, perform actions and generate events that comes from web services. As we can see in the figure 5.2, this module directly push and receive data from the rule engine, unlike Berries-DrEWE which makes use of another module to insert the events into the rule engine. This module and the previous one are part of the event layer, the one that makes contact with the user, and both modules handle events and actions but, as you can see in the figure 5.2, it is worth to mention that the flow of actions goes in the opposite direction of events' flow.

3. *GSN-DrEWE.* This module represents the event network and the persistence module for the physical sensor that we mentioned in Berries-DrEWE. Its purpose is to receive the events from the previous module, address the different events to their listeners, tag them for a later handling and make sure that they persist. For this purpose we have used the European project GSN[3] and adapted it to be suitable for our requirements. It also provide persistence by default, so every event generated before will be available after it is produced.

[3] `http://sourceforge.net/apps/trac/gsn/`

4. *DrEWE complex rule engine.* Under the DrEWE complex rule engine, lies the processing module. It is in charge of process the incoming events and generate intents that will become executed actions. We have integrated two different rule engines that works perfectly together Drools Rule Engine and SPIN, the second one will add a semantic point to this task automation platform. As the first is in charge of general event handling and complex event processing, the second one takes the high level events generated by the first one and perform high level actions. It is worth to mention that rules, actions and events from the SPIN engine, are based on the EWE ontology which we have talked before in section 3.3.

5.3 Berries-DrEWE

A task automation platform would not be anything if it had not the software to feed it with events nor the software to perform actions. Under this subsection we will analyse the module we have developed to read information from the environment and convert it to readable, processable and timed events. We have developed handlers,so called berries, for three physical sensors: light sensor, dni-e sensor and camera. They have been developed as an example of event generator modules but they could have been as many as we had needed. Web handlers can also introduce into the system by reading information from from third party services, we will describe them in section 5.5.

Berries module generates events retrieving data from actual sensors and web handlers, as we will see in section 5.5.1, generate events from a third-party web service. This whole project but specially this part belongs to which has been called for long time ago *"The internet of things"*, as we have previously explained in section 2.2.

The way that events are captured or created depends on each module and will talk about it on each sub-module's section but all the event generators have something in common that is the structure of their output.

Once our event network is deployed,these events will be accepted in a passive way, this means that GSN will be able to receive events under HTTP PUT requests. But this request has a defined and strict structure that has to be followed, otherwise, they will be rejected. We talk about it inside each type of event that we generate.

Firstly, these request must follow the uri:

`http://<gsn-url>:22001/streaming/`

Where *<gsn-url>* is the url where gsn is deployed. By default, the streaming port is

the 22001 but it can be changed at GSN configuration, see GSN at section 5.4 for further information.

The request also must have the following header:

Content-Type: application/x-www-form-urlencoded

And it must have the following parameters:

Notification-id: This parameter is a number that must coincide with the one defined at each virtual sensor. By doing this, GSN will be able to address each event depending on its source. For example, for light events we use the notification-id 1.2.

Data: This parameter is where we must put the data we want to put to GSN and it must have exactly the same fields with the same type of value that has been defined in its corresponding virtual sensor. As a example, we present the data field generated by a calendar event:

```
<stream-element timestamp="2013-9-9 0:38:49 CEST">
   <field name="title" type="string">Meeting</field>
   <field name="attendees" type="string">botgsi@gmail.com#
      carlos.crespog@gmail.com</field>
   <field name="start" type="string">2013-09-02
      17:30:00+02:00</field>
</stream-element>
```

Listing 5.1: GSN PUT request data parameter example

- **timestamp:** this will be the timestamp that GSN will set as the time this event has been generated
- **name:** this is an unique name to identify each event's parameters. This event has three attributes so we have defined three different unique name: title, attendees and start.
- **type:** this must be equals to the field's type defined for this attribute, in this case, all of them are string but, for example, dni events has a numeric field: the dni number itself.

As we have mentioned before, this module is composed by software to be executed inside a raspberry. We have three functionalities to be mentioned: retrieve the light level, retrieve the dni log and handle the camera.

5.3.1 Retrieve the light level

This Python script sends the light level to a GSN server. The data acquisition is made via a RC circuit attached to a given pin, because of the lack of analog inputs in the raspberry, this script sets a given entry as low and counts the loop's cycles that it spends discharging.

The trick is to time how long it takes a point in the circuit the reach a voltage that is great enough to register as a "High" on a GPIO pin. For this purpose, we use the following circuit:

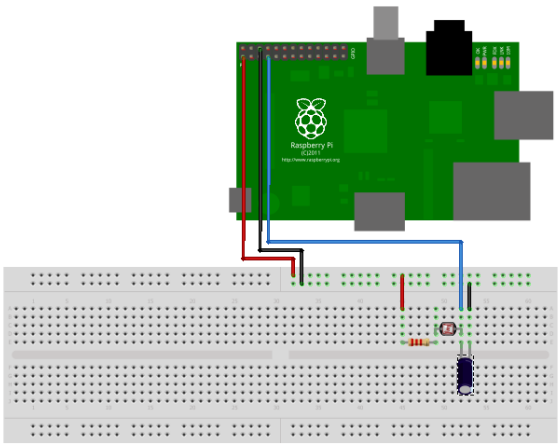

Figure 5.3: Raspberry circuit for retrieving the light level

Once we have retrieved the light level from the light sensor, this module will generate an event to be pushed to GSN as the following:

```
<stream-element timestamp='2013-7-11 7:13:52 CET'>
  <field name='value' type='numeric'>2935</field>
</stream-element>
```

Listing 5.2: GSN PUT request data parameter for light sensor

5.3.2 Retrieve the dni log

Since we need to have a log of who enters the laboratory and when this happens, we have installed a dni-e reader at the entrance. Instead of having keys or other solution, we have integrated the door lock with this reader in order to make necessary to enter your dni-e to open the door. As we talked in the internet of things section, there are more options: we could have also implemented other solutions as for example RFID tags or cards [6], but we decided to use this one because every user already had a dni-e card.

The script in charge of this task is dni.sh. This script checks the last person that has inserted its dni at the laboratory door, sends the info to the GSN server at the given url and echoes the info via terminal, this is part of the use case we have studied and is detailed under chapter 6. In order to implement this scripts, we count with an electronic dni[4] reader that we have installed at the laboratory door.

The raspberry pi and the dni server must be known ssh hosts of each other. In order to make this task iterative we execute it by dniLoop.sh. This script is necessary due to the nature of the dni server, that changes its logs in an unpleasant way.

After collecting the dni information this module will generate an event to be pushed to GSN as the following:

```
<stream-element timestamp='2013-12-11 7:13:52 CET'>
  <field name='numer' type='numeric'>78900012</field>
  <field name='name' type='string'>Carlos Crespo</field>
</stream-element>
```

Listing 5.3: GSN PUT request data parameter for dni sensor

5.3.3 Handle the camera

Inside the berries suite, we also find the necessary software to handle the camera. We use a modified version of the motion packet that goes for the name of motion-mmal[5]. This packet automatically sets up a http server that controls the camera and several other features.

The motion packet allows us to take photos and record video but it also brings a func-

[4] http://www.dnielectronico.es/

[5] http://github.com/dozencrows/motion/tree/mmal-test

tionality that is worth to mention: movement detection. Using this features, we can generate events when something changes around the environment or when someone approaches to the Raspberry.

Motion provides several methods to control the camera, all of them documented at the Motion http API. For example, if you want to force a raspberry snapshot from any other machine in the network (assume that raspberry's ip is 192.168.1.132)

GET `http://192.168.1.132:8080/0/action/snapshot`

Motion provides various amazing features to control the camera, one of them is the video streaming. However we only needed motion to take pictures, nothing to do with video.

We also have developed a simple script that takes the last picture taken by motion and serves it to the world (in our case, just to the network) via http, creating a http server with python module BaseHTTPServer.

Once deployed, in order to get the last picture (by default, port is 8088 and we assume the raspberry's ip is the same as above)

GET `http://192.168.1.132:8088/`

5.4 GSN module

GSN is a middleware (extensible software infrastructure) for rapid deployment and integration of heterogeneous wireless sensor networks. In this project, we have implemented it as an Event Network that retrieve information from our physical sensors and make it accessible for the rule engines.

In DrEWE project, we use GSN as an event network that is a little bit different than the typical sensor network GSN used to be. Some features that an event network may have:

- *Entry points:* for that purpose, we use the GSN remote push wrapper. This wrapper deploys an entry point via HTTP request PUT, so the modules in charge of generate events simply have to make this type of request and the events will be pushed into our event network.
- *Accessible exit points:* one of the main features of GSN, is that all the data can be retrieved, by default, via HTTP request from any point of the network by simple HTTP GET request that are detailed under the GSN subtree.
- *Timed database:* one of the features of events is that they are timed. By default, GSN

provides a timestamp column for each type of data that it receives. This timestamp is used by the following modules (the rule engines) for complex event reasoning in order to provide extreme potential.

- *Directionable access:* this means that one application in the network is able to subscribe to one or more channels and retrieve only the information that it needs. For example, one application that only wants to know who enters the laboratory and doesn't care about the light level. Furthermore, this represents an abstraction layer for the next step: inserting the events into a CEP rule engine.

5.4.1 Virtual sensors

As we explained in section 3.2 and mentioned in section 5.3, virtual sensors are the ones responsible for receiving, processing and addressing the data from the previous modules. They are xml configuration files that are automatically deployed once GSN is started and we have implemented three of them that can be found at the appendix B. But under this section we present the structure and implementation of only one of them: the dni virtual sensor. We can see the full implementation of the virtual sensor right below:

```
1  <virtual-sensor name="RemoteDniVS" priority="11">
2
3    <processing-class>
4      <class-name>gsn.vsensor.BridgeVirtualSensor</class-name
           >
5      <output-structure>
6        <field name="number" type="int" />
7        <field name="name" type="varchar(60)" />
8      </output-structure>
9    </processing-class>
10   <description>Get data from dni sensor</description>
11   <life-cycle pool-size="100" />
12   <addressing>
13     <predicate key="geographical">Not yet specified</
           predicate>
14     <predicate key="eventType">DniEvent</predicate>
15   </addressing>
16   <storage history-size="2h" />
```

```
17    <streams>
18      <stream name="dniInputStream">
19        <source alias="dniSourceStream" sampling-rate="1"
              storage-size="1">
20        <address wrapper="remote-direct">
21          <predicate key="notification-id">1.3</predicate>
22          <output-structure>
23            <field name="number" type="int" />
24            <field name="name" type="varchar(60)" />
25          </output-structure>
26        </address>
27        <query>select * from wrapper</query>
28      </source>
29      <query>select * from dniSourceStream</query>
30    </stream>
31  </streams>
```

Listing 5.4: Virtual sensor full implementation

A virtual sensor has a unique name (the name attribute in line 1 at listing 5.4.1) and the priority field that controls the processing priority. Beneath the first line, there is the processing-class tag where we define which one of the existing processing class is going to process the data that this virtual sensor retrieves. For example, in this one we define that our processing class is going to be *BridgeVirtualSensor* (line 4) that is the usual choice when you only want to use the SQL filtering mechanism, without any data transformation.

The output-structure (line 5) inside the processing-class tag actually defines how the output is going to be, in our case, two fields are defined from the dni-e sensor: number and name. The first one is an integer so we put the keyword int in the field type, and the second one is, in fact, the user name, so it is a string.

At the lines 10 and 11, we have a description tag and the life-cycle pool-size, which enables the control and management of resources provided to a virtual sensor such as the maximum number of threads/queues available for processing.

Each virtual sensor can be equipped with a set of key-value pairs representing the logical addressing of the virtual sensor (lines 12 to 15), i.e., associated with metadata. The

addressing information can be registered and discovered in GSN and other virtual sensors can use either the unique name or logical addressing based on the metadata to refer to a virtual sensor.

Please notice that one of the predicate field with the key: eventType (line 14), is the one responsible for addressing the events in next modules. Finally, at line (16) we can see, the <storage> element which allows the user to control how output stream data is persistently stored.

At the other half of the virtual sensor (lines 17 to 31) we meet the stream definition, in this case, the virtual sensor only have one stream that is defined as follow:

The main point on this snippet is the address-wrapper at line 20 when we define which wrapper we are using, wrappers are detailed in section 3.2 and they are used to encapsulate the data received from the data source into the standard GSN data model, called a StreamElement. By writing remote-direct we specify that we use the *Direct Remote Push Wrapper* which is detailed on next section.

5.4.2 Direct Remote Push Wrapper

As we have said before, each virtual sensor needs a processing class, which is called a wrapper. The previous example uses the Direct Remote Push Wrapper, which comes with the GSN default distribution. This wrapper passively listens for pushed data from a specific remote sensor. Typically it can be used to retrieve data from devices that are not always connected or that may change their IP address often. The structure has to be defined in the xml file as in the example at listing 5.4.1. The notification-id is the only predicate value in the wrapper definition and is used as the key for identifying the remote sensor.

When this wrapper is used by a deployed virtual sensor, it will open an entry point where data can be pushed via HTTP by following the exact way detailed in section 5.3, otherwise, the data will be rejected.

Once deployed, GSN data is accesible from any point of the network. By default, GSN will be deployed at port 22001 so, for example, to access data from the dni virtual sensor:

GET `http://localhost:22001/gsn?request=114&name=RemoteDniVS&window=40`

request parameter is for the type of request, 114 type returns a stream of GSN data limited by the window parameter

name parameter is for the name field of each virtual sensor, it is defined inside each virtual

sensor as we have seen at the previous subsection.

To sum up, we push data to GSN depending on the notification-id parameter in a PUT request, we retrieve data from GSN depending on the name parameter in a GET request

5.5 Web handlers

Web handlers are in charge of connecting web services and our system, and despite we only describe some of them under this section, an unlimited number of them can be implemented. For further implementation details please go to chapter 6. Web handlers are the one responsible for pushing events to the system and performing actions outside the system, although those events are only the ones from web service, not from physical sensors. At both cases, pushing events and performing actions, handlers are highly related to the service they handle. Events represents happenings at the application they wrap, these events are inserted into the system, for example a new events at your calendar, a new article related with a preconfigured topic or a new email at your inbox. Actions are commands that are executed making use of the possibilities each service provides: send an email, post a tweet or schedule an event at your calendar. Both actions and events can be configured.

Usually, web services that are used by these handlers, provide a public API which suppose a reliable and long-lasting implementation. However, for that services that does not provide an API, we use some other type of techniques like scrappers plus polling for generating events, or form-filling for executing actions.

Every web handler is represented by an ewe:channel as we refer in section 6.7, that describes the available events and actions. In this way, we have semantic descriptions of each handler which open some possibilities like user-based recommendation or facet-based search. Moreover, this semantic description allows DrEWE's semantic rule engine to execute these channels.

5.5.1 Google Calendar module

Google Calendar module is an event generator that produces internet events by retrieving information from a web service: Google Calendar. Despite the previous event generators, it does not need GSN because it push events directly into the rule engines.

Firstly, he make the needed request to the third-party service, which in our case is Google Calendar, secondly, it gathers the information obtained from that request in order to form a

message and and finally, it packs that message when the necessary fields to be transformed into an event and push it to the rule engine. This process is explained in figure 5.4

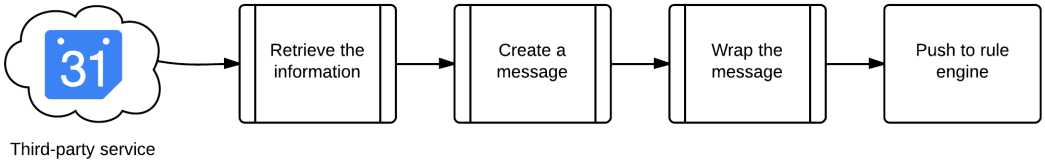

Figure 5.4: General process to generate events from third-party services

Google Calendar will provide us information relative to DrEWE's meeting events. One meeting event in our platform corresponds with one event at some calendar from google services, so we retrieve the information of each google calendar's event in order to create a message, but, as these events have more fields than we need, we only use the following:

Property name	Value	Description
id	string	Identifier of the event.
summary	string	Title of the event.
description	string	Description of the event.
location	string	Geographic location of the event as free-form text.
start	nested object	The (inclusive) start time of the event. For a recurring event, this is the start time of the first instance.
end	nested object	The (exclusive) end time of the event. For a recurring event, this is the end time of the first instance.
attendees[ ]	list	The attendees of the event.
created	datetime	Creation time of the event (as a RFC 3339 timestamp). Read-only.

Table 5.1: Google Calendar related fields

The message wrapping consist in adding two more fields in order to allow the rule engine to create and insert an event. These two fields are a timestamp and the event type.

Timestamp

As we use Complex Event Processing, every event has to be timed. Because of that, we create and fill the timestamp field before inserting it into the rule engine so as to allow the engine to know where it has been created. We make this as norm with every

event.

Event type

Although each stream of events is separated and can be identified by where it comes, it is necessary to indicate in which exactly class will become this event once it is inserted in the engine. We do not need this at the other event generators because, as we will see on the next section, GSN is in charge of that. The main reason why this field exist, is that our rule engine are written in a strong typified language which obligate us to specify a class or type for each event type.

5.5.2 Twitter module

This module is the one which wraps the well-known twitter service. As we detailed in the chapter 6, this module connects to the twitter service, authenticate itself and perform API calls that turn into twitter actions. Under this subsection we will detail the information about one concrete action: post a tweet into twitter.

Unlike Google calendar's scenario, the twitter module perform an action from a given message. So, once a rule has been triggered and a twitter intent has been thrown, this module will receive a message with the needed parameters to perform an action. Hence, it will make the correspondent request to twitter API in order to perform the action. This process is detailed in figure 5.5.

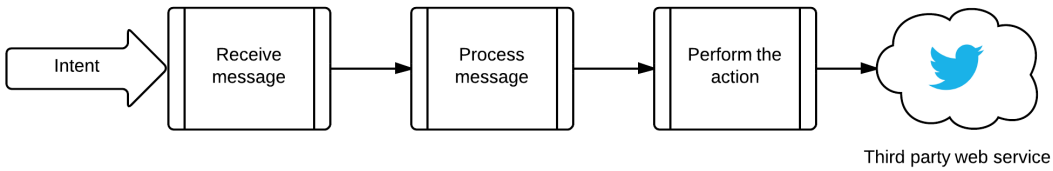

Figure 5.5: General process to perform actions using third party web services

The Twitter API is a very extensive and complete one and the number and type of fields vary depending on each request. This fields are part of the message received by this module. For example, in order to post a tweet, only two fields are needed:

Message

The message to be posted. It has to be shorter than 140 characters.

User credentials

Access data to perform the authentication. Composed by four parameters: Consumer key, consumer secret, access token key and access token secret.

5.6 DrEWE complex rule engine

Under this section lays the module in charge of processing the events, firing the rules and throwing intents to perform actions, it is composed by two rule engines that work together: Semantic rule engine (SPIN) and CEP rule engine (Drools).

As we see in figures 5.6 and 5.7, there are two ways this module can work. In the first one, we use the power of the CEP rule engine in order to correlate low level events and produce high level events. Then, this high level events are handled by the semantic rule engine to perform the actions.

In the second figure, high level events are directly being handled by the semantic rule engine. This scenario is the typical when-then rule and does not require any further processing. In this case the rule engine only addresses each event to its correspondent action following a predefined ewe rule, detailed in section 3.3.

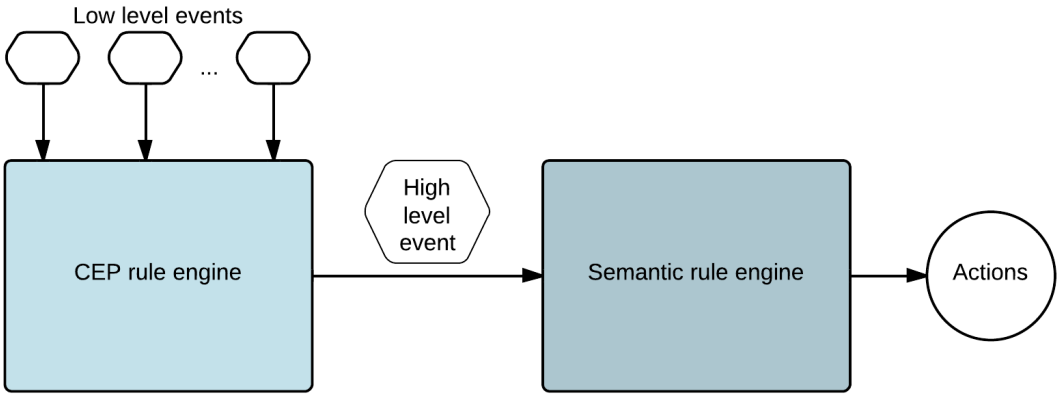

Figure 5.6: DrEWE complex rule engine working with low level events

5.6.1 CEP rule engine

CEP rule engine is the one in charge of handling the low events, mostly the events coming from the sensors, and aggregating them by generating high level actions. It uses the Drools rule engine with its fusion packet, which provides the complex event processing features that

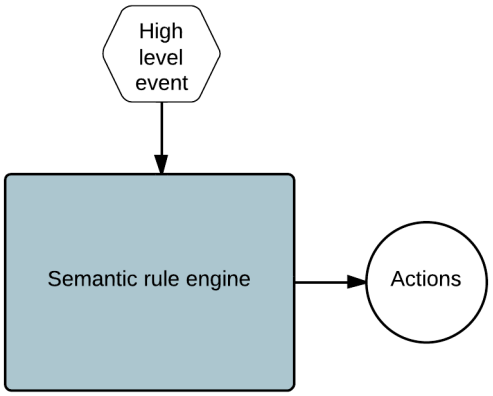

Figure 5.7: DrEWE complex rule engine working with highlevel events

are needed to handle the low level events coming from the sensors, like light level or dni-e entrance log.

One example of complex event processing should be the one related to the light event sensor. As we only receive a numeric value from the light sensor, we need to mark a threshold to decide when the light is on and where is off. Also, as the electronics of the light circuit are quite basic, it is also interesting to take the mean of last five events in order to avoid unpleasant noise-related variation. The rule described in listing 5.6.1 can be seen as follows: *"For the last five light event, calculate the mean and only if it is above the threshold, fire the rule and generate a high level event called light on"*

```
rule "LIGHT_RULE"
        when
          Number( $count : intValue,intValue>=5)
                from accumulate ($light : LightEvent(lightLevel
<= 1000 ) from window LightWindow,
                count($light))
        then
          System.out.println("Light␣on");
          insert(new Event("light_on"));
end
```

Listing 5.5: Drools example rule

The drools rule in listing 5.6.1, sets an example of complex event processing using one of its features: the sliding window. The sliding windows are a way to scope the events of interest by defining a window that is constantly moving i.e.: ”give the latest events and perform operations on them”. Despite it is not the most common way to use the CEP rule engine, there is the possibility of run complex event processing on high level events like for example: ”If someone post an article with a hashtag, and in the following two hours twitter receive over 1000 tweets with the same hashtag: generate the event starred article”

5.6.2 Semantic rule engine

As we have seen in section 3.4.2, SPIN is an inferencing notation with representable web models for inference. So in order to build a semantic rule engine from SPIN we need the following:

1. Transform an event into its semantic rdf form
2. Insert it into a model
3. Continuously check the model looking for new events
4. Continuously run the inference rules
5. Put the inferenced instances in a new model
6. Continuously check the new model for new instances
7. Perform the actions

We can see the process of how a event is inserted in figure 5.8, which corresponds whith points from three to seven:

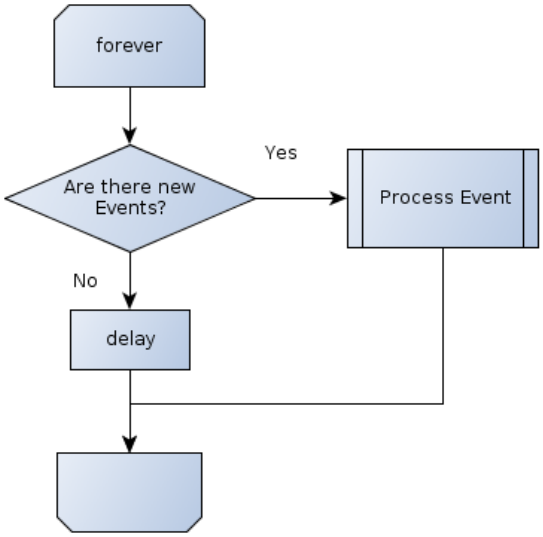

Figure 5.8: DrEWE's semantic rule engine processing incoming events

The Process Event box can be divided in more subprocesses in order to offer a more detailed view of how this engine works. It is represented in figure 5.9 and goes as follows: a new event arrives, it is transformed to rdf and stored, if any rule has been triggered, inferencing rules are executed, this will fill a model with inferred actions and finally, this actions will be executed.

SPIN rules are in fact SPARQL statements with CONSTRUCT and WHERE parts like the following:

```
CONSTRUCT{
        ewe:action dcterms:title "bot" .
        ewe:action dcterms:description ?description .
}
WHERE {
        ?event dcterms:title "meeting" .
        ?event dcterms:description ?description .
}
```

Listing 5.6: SPARQL statement convertible into SPIN rule

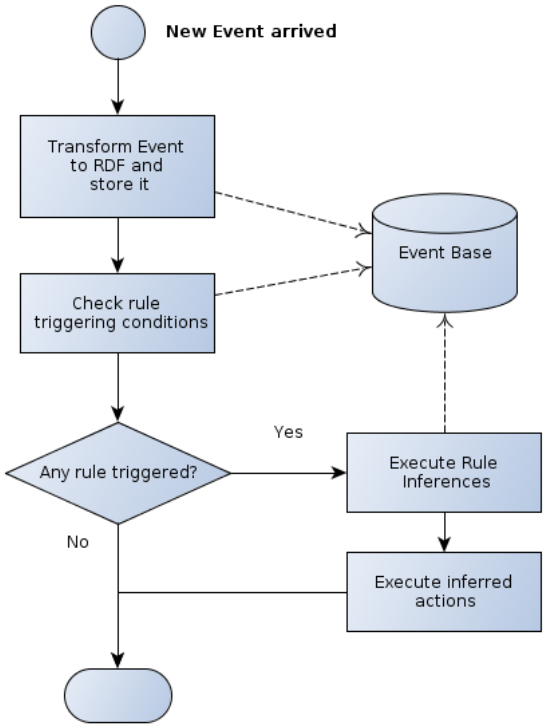

Figure 5.9: Detailed processing of events.

The rule stated in listing 6.6 can be seen as a when then rule that goes like this: *"If a meeting event is received, make the bot (wall display) say something "*. That is the reason why some dressing is needed in order to create a semantic rule engine from the SPIN inference notation.

Chapter 6

Case Study

"Difficile est tenere quae acceperis nisi exerceas"

— Plinia the Elder

6.1 General description

6.1.1 Introduction

In this chapter, we detail the use case that illustrates the capabilities of the architecture described in the previous chapter. Firstly, we explain the use case giving a big picture of what can this platform do and secondly, we get into details for each module in particular. In the last section (6.7), we explain the design of the channels involved in this case.

The purpose of this platform is to be constantly running on our laboratory, as a standalone service to automate tasks that improve productivity or take advantage of the available resources at our workplace. This task are automated by both, CEP and SPIN rules that can be written by the user and saved permanently in the system.

Thus, in the next section we present the sequence of steps to be followed to schedule a meeting using DrEWE (from setting up the rules to the actions that occurs when the meeting is taking place). We show how our platform can assist the users on that process.

6.1.2 Case study

Firstly one user named Tim wants to set up a meeting in order to talk about the results achieved during the previous week and plan the next week. So he goes to the Google Calendar web page and schedules a meeting inviting two other people involved in his current project. The attendees Robin and Michel receive their invitations and decide to go to the meeting.

Once the meeting is about to start, the three attendees (Tim, Robin and Michel) arrive and introduce their dni-e cards into the reader at the entrance of the meeting room. When they arrive and, only if their id cards accredit that they are the three attendees supposed to be at the meeting and they do it on time, the *meeting started* event will be thrown.

Once this event is thrown, some actions will be performed: take a photo, post a tweet mentioning the attendees, show the photo via the meeting wall display and send an email. To implement this case we have to implement two rules represented in figures 6.1 and 6.2

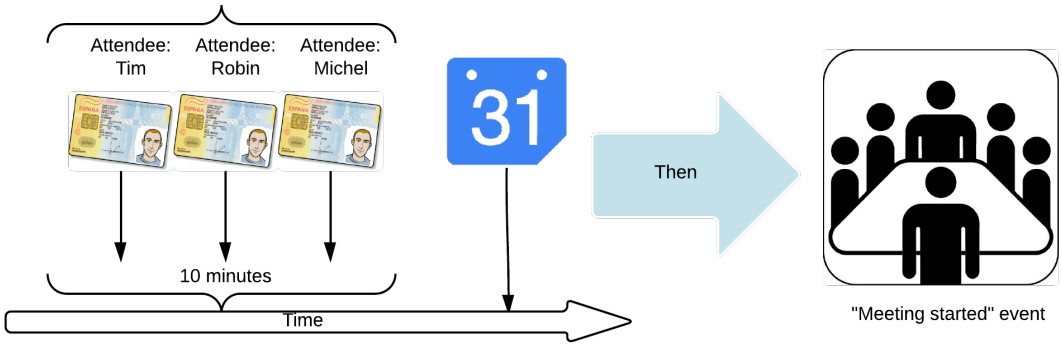

Figure 6.1: CEP part of the rule for the case study

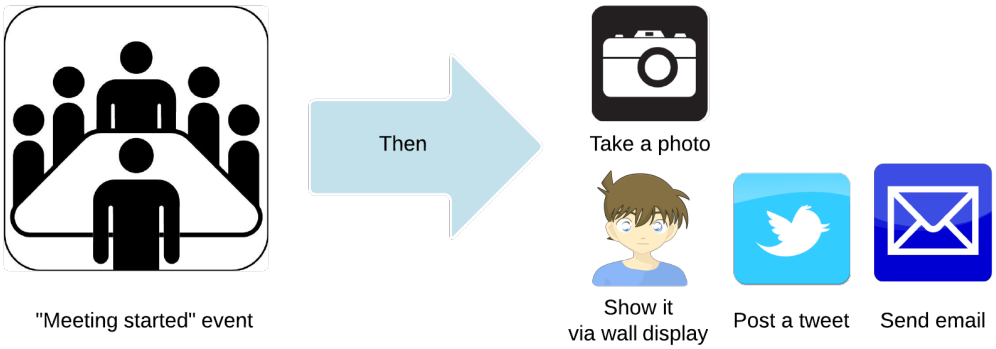

Figure 6.2: High-level part of the rule for the case study

6.2 Google Calendar handler

In order to create an event, the user goes to Google Calendar service using his web browser. Inserts a title, the start date, the location and the attendees, other fields are optional for our system.

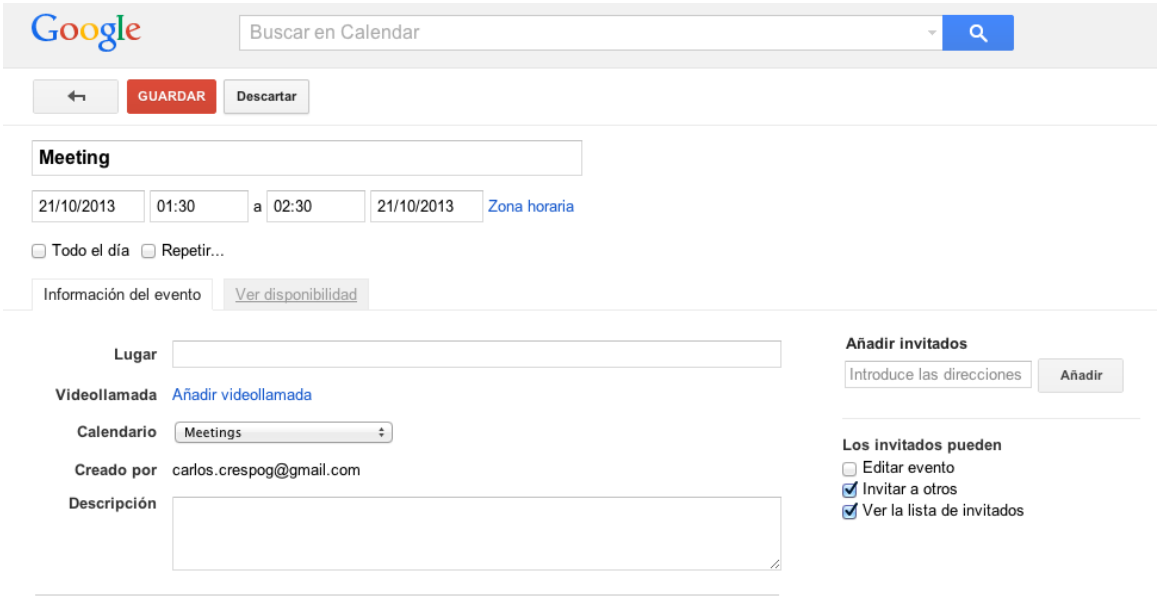

Figure 6.3: Screenshot of google calendar's interface

Google Calendar handler is an node application that runs in background and periodically checks if there are any new scheduled meetings. When it notice a new calendar entry, it will retrieve the meeting information, packet the data and push it to the rule engine.This sequence of how this handler works is represented in figure 6.4.

In order to perform each check, this module has to access to a google service, what means, ask a third party service. In this case, Google offers an extensive API[1] that offers a big set of HTTP petitions for almost everything related with Google Calendar. In this case we want some specific information that is protected, for example: the list of attendees. Due to Google policies this application has to perform authentication by the method Google provides: OAuth 2.0[2]. The details of this type of authentication exceeds the bounds of this document so we will only detail the needed parameters:

- consumer key: Client ID for your project, you'll obtain it once you've registered your project in the Cloud Console[3]

[1] `https://developers.google.com/google-apps/calendar/`
[2] `https://developers.google.com/google-apps/calendar/auth`
[3] `http://cloud.google.com/console/project`

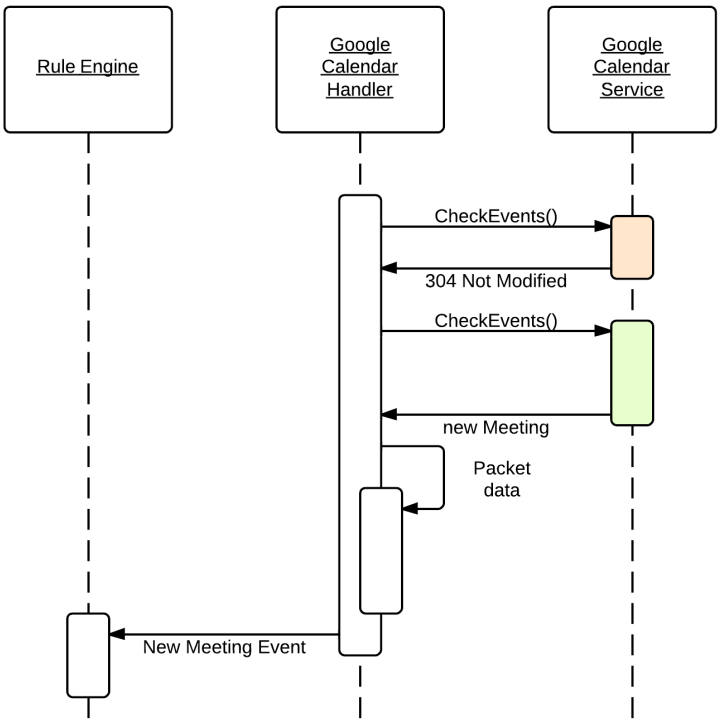

Figure 6.4: Sequence of Google Calendar handler retrieving new events

- consumer secret: Client secret for your project, same as above.
- redirect url: you must grant access to this url in the Cloud Console.
- access token: access token for your application.
- refresh token: refresh token for your application.

6.3 DNI event handler

Once the calendar entry is created and the meeting event is pushed into the rule engine, the users might come at meeting start date. So as to check who is inside the laboratory and when did he enter, we have installed a dni-e (Spanish Id card) reader connected to a server that keeps a log of every entrance through the door. This is also our access mechanism so that if the door is closed, you can open it and access the laboratory only with your identity card. This is a similar solution than RFID cards to grant access but since everyone carry these cards this was our decision.

Our handler accesses periodically to this servers and checks if anybody has introduced

its dni. As we saw in section 5.3.2 this handler will have an output as the one in listing 6.3 and will push it via HTTP PUT to the GSN server at the next url:

`http://localhost:22001/streaming`

```
<stream-element timestamp='2013-12-11 7:13:52 CET'>
  <field name='numer' type='numeric'>78900012</field>
  <field name='name' type='string'>Carlos Crespo</field>
</stream-element>
```

Listing 6.1: GSN PUT request data parameter for dni sensor

6.4 Sensor network: GSN

In order to receive the previous PUT request correctly, GSN should have the correspondent virtual sensor deployed. In this case is the one in the appendix B.2 (RemoteDniVS) and has the following parameters:

- name: name of the person who enters the laboratory
- number: Id card number

Once this virtual sensor is deployed, all the dni-e related data will that have been pushed into GSN will be available by the following request:

GET `http://localhost:22001/gsn?REQUEST=114&name=remotevs&window=40`

Which returns the stream in listing 6.4. It has the same parameters that the virtual sensor plus the "timed" parameter: which is the exact date of the entrance.

```
<result>
  <stream-element>
    <field name="NUMBER">821212</field>
    <field name="NAME">CARLOS ANGEL</field>
    <field name="timed">27/09/2013 13:14:34 +0200</field>
  </stream-element>
  <stream-element>
```

```
    <field name="NUMBER">461212</field>
    <field name="NAME">MIGUEL CORONADO</field>
    <field name="timed">27/09/2013 13:14:19 +0200</field>
  </stream-element>
  <stream-element>
    <field name="NUMBER">7012121</field>
    <field name="NAME">CARLOS CRESPO</field>
    <field name="timed">27/09/2013 13:14:02 +0200</field>
  </stream-element>
</result>
```

Listing 6.2: Output from Dni remote Virtual sensor

As we mentioned when describing the architecture, our sensor network has been designed to be totally passive. So once it has all the events from the sensor, it should be checked by the next module: the CEP rule engine. The sequence of the dni event handler and GSN goes as follows in figure 6.5: the dni handler periodically checks the state of the log of the dni server and its last entries, when it finds a new dni entry, it compress the information in a suitable message for GSN and sends the message to it. This change on the entries of GSN is noticed by the periodically checks of the rule engine and finally, the dni event is retrieved and pushed into it.

6.5 CEP rule engine: Drools

As we already detailed how this engine works, we will only present the implementation referent to this case study. The CEP rule in charge of achieving the task of completing our desired goal is written in listing 6.5

```
rule "Use case meeting 3 people"
        when
                $newEventReunion: CalendarEvent() from
                entry-point entrada
                $newEventDni : DniEvent(this during
                $newEventReunion) from entry-point entrada
```

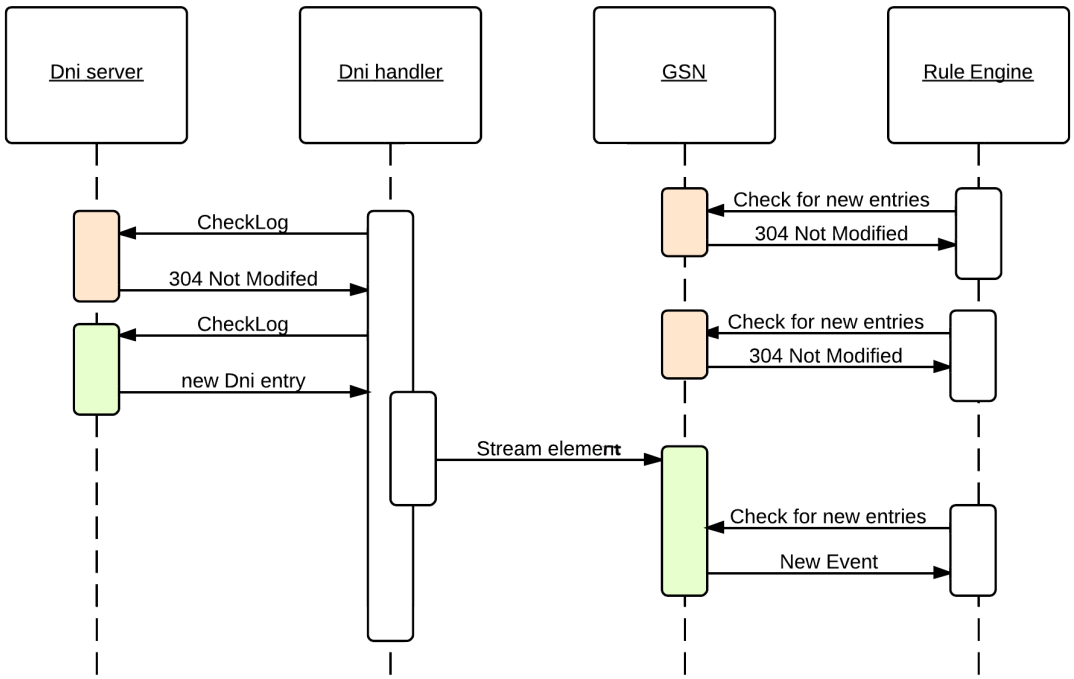

Figure 6.5: Sequence for dni interaction with the rule engine

```
            $newEventDni2: DniEvent(this after[ 1s,20m ]
             $newEventDni ) from entry-point entrada
            $newEventDni3: DniEvent(this after[ 1s,20m ]
             $newEventDni2 ) from entry-point entrada

            eval(checkAttendees($newEventReunion,$newEventDni,
            $newEventDni2,$newEventDni3))

    then

            insert(new SPINEvent("meeting","Meeting started
            with "+$newEventDni.getName()+", "
            +$newEventDni2.getName()+" and "
            +$newEventDni3.getName()));

            System.out.println("Use case rule triggered");
```

```
end
```

Listing 6.3: Drools example rule

The function checkAttendees() will collate the information of the three expected attendees with the data from the three events from dni and will only fire when it matches. Then it will be generate a event to be handled by SPIN.

6.6 Semantic rule engine: SPIN

The semantic rule engine based on SPIN follows its own sequence and has been detailed in section 5.6.2 the SPARQL rules that will perform our desired actions are the following:

```
CONSTRUCT{
        ewe:action dcterms:title "wallDisplay" .
        ewe:action dcterms:description ?description .
}
WHERE {
        ?event dcterms:title "meeting" .
        ?event dcterms:description ?description .
}
```

Listing 6.4: SPIN rule for using the wall display

```
CONSTRUCT{
        ewe:action dcterms:title "email" .
        ewe:action dcterms:description ?description .
}
WHERE {
        ?event dcterms:title "meeting" .
        ?event dcterms:description ?description .
```

```
}
```

Listing 6.5: SPIN rule for sending an email

```
CONSTRUCT{
        ewe:action dcterms:title "tweet" .
        ewe:action dcterms:description ?description .
}
WHERE {
        ?event dcterms:title "meeting" .
        ?event dcterms:description ?description .
}
```

Listing 6.6: SPIN rule for posting a tweet

As we explained at the chapter 5, these rules only will create an rdf instance in a inferenced model, then this model will be checked and actions will be performed depending on its fields. The three other modules related to this case study: wall display, twitter handler and email handler are web service handlers like the explained for google calendar service.

6.7 EWE channel design

In a similar way as the services we discussed in section 2.5, our architecture serves as a task automation platform in which semantically we can describe all the elements with an ontology. One of the main goals of this project is to provide a compatible platform for the use of the EWE ontology we explained in section 3.3. This goal is achieved thanks to the semantic rule engine based on SPIN that allows us to implement ewe rules that generates ewe actions triggered by ewe events.

Under this section, we present four channels following the ewe ontology scheme: WallDisplay, MeetingChannel, TwitterChannel and Google Calendar Channel. As we said in section 3.3, a channel defines the behavior of a web service or a physical device such as sensors or

actuators. Evernote channel[4] at Ifttt.com, is an example of a web service based channel. WeMo Switch channel[5], on the other hand, involves physical devices and the Belkin WeMo switches for home automation[6]. The complete implementation of each channel can be found at the end of this book on appendix A.

6.7.1 Wall display channel

This channel, orchestrated by DrEWE, represents the actions that can be taken by a meeting wall, which is a screen that continuously displays information about the meeting and allow the user to perform meeting-related tasks. For example, shows who is arriving at the meeting or perform the action add meeting. In figure 6.6, we can see the graphic representation of this channel.

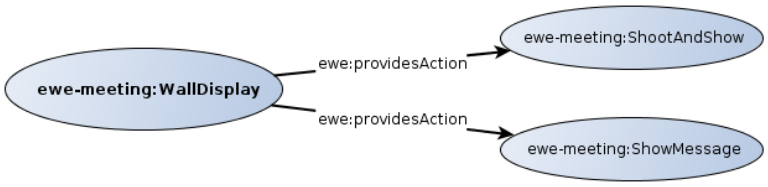

Figure 6.6: Graphic representation of Wall Display channel

It is grouped among those channels involving physical devices. This channel provides two actions, "Shoot and Show" and "Show Message". Both include a configuration parameter to select the device, i.e. the screen in which the action will be executed. The user needs to select among those devices that are registered to be used with its account. This way, multiple users are allowed. It follows the same design pattern as WeMo Switch[7] or WeMo Motion[8] channels, that allow the user to interact with different sensors and actuator.

ewe-meeting:ShootAndShow

This actions takes a picture with the camera and displays it on the screen. The camera used is that associated to the screen. If there is none, no action is taken.

```
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
    meeting-channel/ShootAndShow">
```

[4] https://ifttt.com/evernote
[5] https://ifttt.com/wemo_switch
[6] http://www.belkin.com/us/Products/home-automation/c/wemo-home-automation
[7] https://ifttt.com/wemo_switch
[8] https://ifttt.com/wemo_motion

```
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/Action"/>
    <dcterms:title>Shoot and show picture</dcterms:title>
    <dcterms:description>This actions takes a pictute with the
        camera and displays it on the screen...</dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/WhichDisplay">
  </owl:Class>
```

Listing 6.7: ShootAndShow action implementation

- **ewe-meeting:WhichDisplay**

  WhichDisplay is an input parameter for both actions which aims to be a configuration parameter that decide in which display this photo will be shown.

ewe-meeting:ShowMessage

This actions shows a message on the screen. It takes two input parameters: the display to use (like the previous one) and the message itself.

```
    <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
        meeting-channel/ShowMessage">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/Action"/>
    <dcterms:title>Display message</dcterms:title>
    <dcterms:description>This actions shows a message on the screen<
        /dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/WhichDisplay">
    <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/message">
  </owl:Class>
```

Listing 6.8: ShowMessage action implementation

- **ewe-meeting:WhichDisplay**

  WhichDisplay, in this case, is a configuration parameter that decide in which display the message will be shown.

- **ewe-meeting:Message**
  The message to be displayed.

6.7.2 Meeting channel

The meeting channel models a meeting arrangement web service. It offers all the information related to meeting events and provides several basic actions. This channel, orchestrated by DrEWE, offers a service to schedule and cancel meetings from our platform without any interaction with the third-party web service.

Events and actions contained in this channel also have a *ewe-meeting:Location* configuration parameter which associate them with a meeting room and a access mechanism, for example, in our case study, this mechanism is a dni-e reader but it can be anything that allows the system to know who is coming and who is not. We can see a detailed view of this channel at figure 6.7

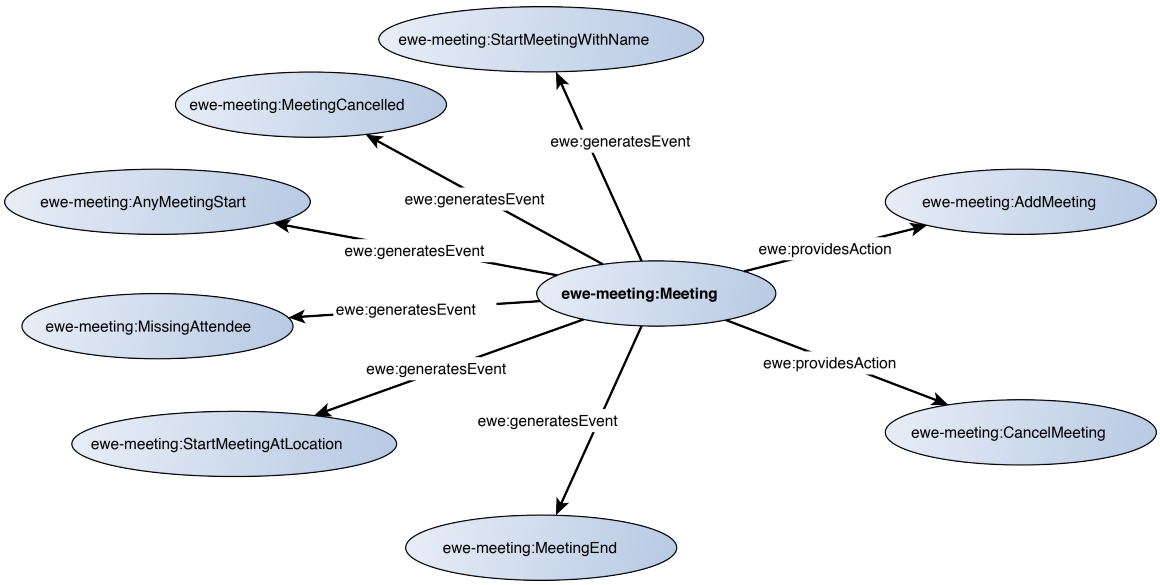

Figure 6.7: Graphic representation of Meeting channel

6.7.2.1 Events

ewe-meeting:StartMeetingWithName

This trigger fires every time a new meeting with the name given starts. A meeting starts when the time of the meeting arrived and all the attendees have already logged in at the meeting room.

```
    <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
        meeting-channel/StartStartWithName">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/Event"/>
    <dcterms:title>Meeting with name starts</dcterms:title>
    <dcterms:description>This trigger fires every time a new meeting
         with the name given starts...</dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter>
      <owl:Class>
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/InputParameter"/>
          <dcterms:title>Which meeting?</dcterms:title>
          <dcterms:description>The name of the meeting. This
              parameter filters the meetings that triggers the event.
              </dcterms:description>
          <dcterms:type>string</dcterms:type>
        </owl:Class>
    </ewe:hasInputParameter>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
  </owl:Class>
```

Listing 6.9: StartMeetingWithName event implementation

Parameters:

- **ewe-meeting:Attendees**
  List of attendees for the meeting
- **ewe-meeting:StartTime**
  The start time of the meeting

- **ewe-meeting:EndTime**
  The ending time of the meeting
- **ewe-meeting:Location**
  The meeting location, this parameter is also used to link an access control mechanism with the meeting, in order to let DrEWE know who is already at the meeting and who is missing.
- **ewe-meeting:whichMeeting**
  The name to check in order to find a meeting that coincide with the given parameter.

ewe-meeting:AnyMeetingStart

This trigger fires every time a new meeting starts. A meeting starts when the time of the meeting arrived and all the attendees have already logged in at the meeting room.

```
        <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
            meeting-channel/AnyMeetingStart">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/Event"/>
    <dcterms:title>The meeting starts</dcterms:title>
    <dcterms:description>This trigger fires every time a new meeting
         starts. A meeting starts when the time of the meeting
        arrived and all the attendees have already logged in at the
        meeting room.</dcterms:description>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
</owl:Class>
```

Listing 6.10: AnyMeetingStart event implementation

Parameters: No additional parameters

ewe-meeting:StartMeetingAtLocation

This trigger fires every time a new meeting at the location given starts. A meeting starts when the time of the meeting arrived and all the attendees have already logged in at the meeting room.

```
        <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
            meeting-channel/StartMeetingAtLocation">
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/Event"/>
        <dcterms:title>Start meeting at location</dcterms:title>
        <dcterms:description>This trigger fires every time a new meeting
             at the location given starts...</dcterms:description>
        <!-- Configuration -->
        <ewe:hasInputParameter>
          <owl:Class>
            <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/InputParameter"/>
            <dcterms:title>Meeting location</dcterms:title>
            <dcterms:description>The location of the meeting. This
                parameter filters the meetings that triggers the event</
                dcterms:description>
            <dcterms:type>string</dcterms:type>
          </owl:Class>
        </ewe:hasInputParameter>
        <!-- Output parameter list -->
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Attendees">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/StartTime">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/EndTime">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Description">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Title">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Location">
      </owl:Class>
```

Listing 6.11: StartMeetingAtLocation event implementation

Parameters:

- **ewe-meeting:WhichLocation**
  This given location to trigger the event, if a meeting is started at this location, the event will be triggered.

ewe-meeting:MissingAttendee

This trigger fires every time there is an attendee is missing a meeting. It is considered that an attendee is missing a meeting when he/she has not logged in at the meeting room after 10 minutes of courtesy past the meeting time.

```
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
    meeting-channel/MissingAttendee">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/Event"/>
    <dcterms:title>There is a missing attendee</dcterms:title>
    <dcterms:description>This trigger fires every time there is an
        attendee is missing a meeting...</dcterms:description>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/OutputParameter"/>
        <dcterms:title>Missing attendees</dcterms:title>
        <dcterms:description>The list of attendees that didn't
            attended the meeting</dcterms:description>
        <dcterms:type>string</dcterms:type>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
```

```
        </owl:Class>
```

Listing 6.12: MissingAttendee event implementation

Parameters:

- **ewe-meeting:MissingAttendees**
  List of attendees that are missing the meeting

ewe-meeting:MeetingCancelled

This trigger fires when a meeting has been cancelled.

```
        <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
            meeting-channel/MeetingCancelled">
            <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
                ewe/Event"/>
            <dcterms:title>A meeting was cancelled</dcterms:title>
            <dcterms:description>This trigger fires when a meeting has been
                cancelled.</dcterms:description>
            <!-- Output parameter list -->
            <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/meeting-channel/params/Attendees">
            <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/meeting-channel/params/StartTime">
            <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/meeting-channel/params/EndTime">
            <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/meeting-channel/params/Description">
            <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/meeting-channel/params/Title">
            <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/meeting-channel/params/Location">
        </owl:Class>
```

Listing 6.13: MeetingCancelled event implementation

Parameters: No additional parameters.

ewe-meeting:MeetingEnd

This trigger fires when a meeting, that has already started, ends. A meeting ends, when all the participants log out the meeting room.

```
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
    meeting-channel/MeetingEnd">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
      ewe/Event"/>
  <dcterms:title>A meeting ended</dcterms:title>
  <dcterms:description>This trigger fires when a meeting, that has
       already started, ends...</dcterms:description>
  <!-- Output parameter list -->
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Attendees">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/StartTime">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/EndTime">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Description">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Title">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Location">
</owl:Class>
```

Listing 6.14: MeetingEnd event implementation

Parameters: No additional parameters.

ewe-meeting:MeetingEndingTime

This trigger fires at the ending time of a meeting that previously started, the main difference between this one and the previous one is that the previous one fires when every person has logged out and this fires when the ending time comes.

```
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
    meeting-channel/MeetingEndingTime">
<rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
    ewe/Event"/>
<dcterms:title>Meeting ending time</dcterms:title>
<dcterms:description>This trigger fires at the ending time of a
    meeting that previously started.</dcterms:description>
```

```
        <!-- Output parameter list -->
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Attendees">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/StartTime">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/EndTime">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Description">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Title">
        <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Location">
    </owl:Class>
```

Listing 6.15: MeetingEndingTime event implementation

Parameters: No additional parameters.

6.7.2.2 Actions

ewe-meeting:AddMeeting

This action schedules a meeting with the data given

```
    <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
        meeting-channel/AddMeeting">
      <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
          ewe/Action"/>
      <dcterms:title>Add a new meeting</dcterms:title>
      <dcterms:description>This schedules a meeting with the data
          given</dcterms:description>
      <!-- Configuration -->
      <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
          ontologies/ewe/ns/meeting-channel/params/Attendees">
      <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
          ontologies/ewe/ns/meeting-channel/params/StartTime">
      <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
          ontologies/ewe/ns/meeting-channel/params/EndTime">
      <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
          ontologies/ewe/ns/meeting-channel/params/Description">
      <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
          ontologies/ewe/ns/meeting-channel/params/Title">
```

```
        <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/ns/meeting-channel/params/Location">
    </owl:Class>
```

Listing 6.16: MeetingEndingTime event implementation

Parameters: In this case, there are not any additional parameter but it is worth to mention that, in this case, they are input parameters, not output like in the case of events.

ewe-meeting:CancelMeeting

This actions will cancel the schedulled meeting that matches the name given.

```
    <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/
        meeting-channel/CancelMeeting">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/Action"/>
    <dcterms:title>Cancel a meeting by name</dcterms:title>
    <dcterms:description>This actions will cancel the schedulled
        meeting that matches the name given</dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter>
      <owl:Class>
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/
            ontologies/ewe/InputParameter"/>
          <dcterms:title>Which meeting?</dcterms:title>
          <dcterms:description>The name of the meeting to cancel.</
              dcterms:description>
          <dcterms:type>string</dcterms:type>
        </owl:Class>
    </ewe:hasInputParameter>
  </owl:Class>
```

Listing 6.17: CancelMeeting action implementation

Parameters:

ewe-meeting:whichMeeting

The name of the meeting to be cancelled.

6.7.3 Twitter channel

This channel has already been implemented at DrEWE, but it has been extracted from another Task Automation Service: IFTTT. We include it here to highlight the interoperability between platforms thanks to the ewe ontology. This channel only has one action, that coincides with the one that DrEWE currently supports but, of course, the complete twitter channel of IFTTT can be implemented at DrEWE.

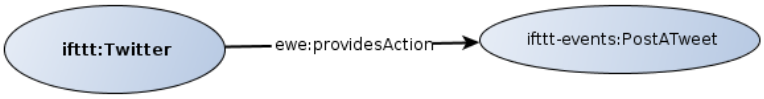

Figure 6.8: Graphic representation of Twitter channel

ifttt-events:PostATweet

This Action will post a new tweet to your Twitter account.

```
<owl:Class rdf:about="https://ifttt.com/channels/twitter/
    actions/PostATweet">
  <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
    ewe/ns/Action"/>
  <dcterms:title>Post a tweet</dcterms:title>
  <dcterms:description>This Action will post a new tweet to your
    Twitter account. NOTE: Please adhereto Twitters Rules and
    Terms of Service.</dcterms:description>
  <ewe:hasInputParameter>
    <owl:Class>
      <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/InputParameter"/>
      <dcterms:title>Whats happening?</dcterms:title>
    </owl:Class>
  </ewe:hasInputParameter>
</owl:Class>
```

Listing 6.18: PostATweet action implementation

Parameters:

- **ifttt-events:WhatsHappening**
  The message to be tweeted

6.7.4 Google Calendar channel

As in the previous one, this channel has been extracted from another Task Automation Service: IFTTT. This is in charge of notify if a meeting has been created on Google Galendar and it is directly used by the meeting channel, in order to avoid any interactions between the user and the end service. In other words, the user can know if a meeting has been scheduled via DrEWE without any contact with the Google Calendar service.

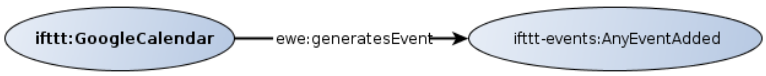

Figure 6.9: Graphic representation of Twitter channel

ifttt-events:AnyNewEventAdded

This Trigger fires every time a new event is added to your Google Calendar.

```
 <owl:Class rdf:about="https://ifttt.com/channels/
     google_calendar/triggers/AnyNewEventAdded">
<rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
    ewe/ns/Event"/>
<dcterms:title>Any new event added</dcterms:title>
<dcterms:description>This Trigger fires every time a new event
    is added to your Google Calendar.</dcterms:description>
<ewe:hasOutputParameter>
  <owl:Class>
    <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/OutputParameter"/>
    <dcterms:title>Title</dcterms:title>
    <dcterms:description>The event's title.</dcterms:description
        >
    <ewe:example>Practice Presentation</ewe:example>
  </owl:Class>
</ewe:hasOutputParameter>
<ewe:hasOutputParameter>
  <owl:Class>
    <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/OutputParameter"/>
    <dcterms:title>Description</dcterms:title>
    <dcterms:description>The event's description.</dcterms:
        description>
    <ewe:example>Make a presentation about new channels on ifttt
        </ewe:example>
```

```
        </owl:Class>
      </ewe:hasOutputParameter>
      <ewe:hasOutputParameter>
        <owl:Class>
          <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
              ontologies/ewe/ns/OutputParameter"/>
          <dcterms:title>Where</dcterms:title>
          <dcterms:description>The location where the event takes
              place.</dcterms:description>
          <ewe:example>Building A, Room 101</ewe:example>
        </owl:Class>
      </ewe:hasOutputParameter>
      <ewe:hasOutputParameter>
        <owl:Class>
          <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
              ontologies/ewe/ns/OutputParameter"/>
          <dcterms:title>Starts</dcterms:title>
          <dcterms:description>Date and time the event starts.</
              dcterms:description>
          <ewe:example>August 23, 2011 at 10:00PM</ewe:example>
        </owl:Class>
      </ewe:hasOutputParameter>
      <ewe:hasOutputParameter>
        <owl:Class>
          <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
              ontologies/ewe/ns/OutputParameter"/>
          <dcterms:title>Ends</dcterms:title>
          <dcterms:description>Date and time the event ends.</dcterms:
              description>
          <ewe:example>August 23, 2011 at 11:00PM</ewe:example>
        </owl:Class>
      </ewe:hasOutputParameter>
      <ewe:hasOutputParameter>
        <owl:Class>
          <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
              ontologies/ewe/ns/OutputParameter"/>
          <dcterms:title>EventUrl</dcterms:title>
          <dcterms:description>A URL to view/edit the event.</dcterms:
              description>
          <ewe:example>https://www.google.com/calendar/event?eid=
              bmpmaDhnMm</ewe:example>
        </owl:Class>
      </ewe:hasOutputParameter>
      <ewe:hasOutputParameter>
        <owl:Class>
```

```
            <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/
                ontologies/ewe/ns/OutputParameter"/>
            <dcterms:title>CreatedAt</dcterms:title>
            <dcterms:description>Date and time the event was created.</
                dcterms:description>
            <ewe:example>August 01, 2013 at 11:00AM</ewe:example>
          </owl:Class>
        </ewe:hasOutputParameter>
      </owl:Class>
```

Listing 6.19: AnyEventAdded action implementation

Parameters:

- **ifttt-events:Title**
  The event's title.
- **ifttt-events:Description**
  The event's description.
- **ifttt-events:Where**
  The location where the event takes place
- **ifttt-events:Starts**
  Date and time the event starts.
- **ifttt-events:Ends**
  Date and time the event ends.
- **ifttt-events:EventUrl**
  A URL to view/edit the event.
- **ifttt-events:CreatedAt**
  Date and time the event was created.

6.8 Conclusions

As we have seen, the implementation goals proposed at the section of requirement analysis, have been fulfilled. In summary, these achieved goals have been gathered in the list below:

- The platform accepts complex rules with temporal restrictions thanks to the CEP rule engine

- Multiple users are supported
- Physical sensors has been implemented and their handlers have been programmed
- It uses several third party web services
- A sensor network has been deployed in order to manage the physical sensors
- A SPIN-based semantic rule engine has been developed in order to work along with the CEP rule engine
- The platform has been developed to work as a standalone service.

To conclude, this use case is detailed and illustrated in the DrEWE's video[9] and all the implementation detailed is available publicly in its repository[10]. Since this project is a platform, this use case is one of the endless possibilities that one task automation service provides. It is also worth to mention that as long as the possible rules or automated tasks depends on the different combinations of triggers and actions, these possibilities grows exponentially with each new implemented service.

[9] `http://www.youtube.com/watch?v=Z7DfvX7VRpQ`
[10] `https://github.com/gsi-upm/DrEWE`

Chapter 7

Conclusion and future work

"It is always wise to look ahead, but difficult to look further than you can see."

— Winston Churchill

7.1 Conclusions

After review the state of the art of Task Automation Service, we identified the necessity and the opportunity for include rules with temporal restriction and reasoning (complex event processing). This platform provides support for this type of rules which functionality has been proved regarding the next step of complexity on task automation.

The event network implemented in this platform allows to model physical sensors from different sources, integrate them and provide easy access to the information they generate. Since modularity has been one of our top priorities, this platform has been designed to grow and to provide quick integration with new services.

7.2 Achieved goals

At the very first of this document, section 1.2, we established a set of goals to be achieved with this thesis. To wrap up, here is a summary of the final outcomes:

- Deepen the knowledge and usage of technologies covered in this project. This has

been achieved during the develop of the project, learning all that has been necessary to fulfilled these goals..

- Build a rule engine that allows both, complex event processing and semantic performance. DrEwe's complex rule engine is, in fact, composed by two different rule engines that works perfectly together: CEP rule engine and Semantic rule engine. See section 5.6
- Deploy a sensor network that suits our needs. Achieved by the use of GSN, see section 6.4
- Develop the software in order to wrap third party web services. As an example, we have developed handlers for Twitter, Google Calendar, Email and a custom meeting Wall display.
- Design some physical sensors to provide events generated from information retrieved of the environment. That has been light, dni-e and a camera.
- Integrate suitable middleware to handle those physical sensors. This middleware is the sensor network that we mention above: GSN.
- Develop a communication protocol to connect all the modules. Detailed along the architecture description in chapter 5

And for the more general aims for this project:

- Study and extend the current state of the state of the art of task automation.
- Explore the capabilities of such technologies for inclusion in intelligent systems.
- Explore and exploit the possibilities of a semantic rule engine.
- Demonstrate viability of the EWE ontology.

7.3 Future work

There are plenty of lines that can be followed to continue this work. As always happens with engineering projects, there is a compromise between time and goals to achieve so some aspects has to be postponed because they land out of the scope of a master thesis.

Now that the concept has been proven to work and be very promising, it is time to continue growing implementing more and more handlers (for sensors and web services) in

order to cover more areas and become a more complete platform because the more services this platform covers, the more possibilities and the more capable it is. In the following sections some fields of study or improvement are presented to the reader.

7.3.1 Create a complex rule composer

At the moment, the only way to create new rules is to code them separately and this requires some skills with a slow learning curve, specially when we are talking about complex event processing. Moreover, you have to go to the plain text files that contains the rules and edit them. So the next step in usability improvement and user experience enhancement should be create a composer or a interface that graphically allows you to create rules selecting triggers, actions and processing logic.

Regarding this goal, some tries were made but we found a challenging task, modelling temporal reasoning without compromising difficulty on composing for the final user. Thus it will be the next big step for this project once it has been decided to go on.

7.3.2 Integrate more web services

It is a fact that the more possibilities a platform offers, the more valuable is for the final user. By developing the handlers indicated in this document, we have defined an interface and a communication protocol to develop more and more handlers. Plus, since almost every notable web service on the internet provides a well-documented API, it is only a matter of time and work to integrate more services in this platform.

7.3.3 Integrate more physical sensors

To integrate more physical sensors is a challenging task but rewarding as well. The possibilities of the internet of things paradigm are endless, from the wide field of domotics to sensor networks. Also, the GSN project integrated in this platforms allows to easily add more sensors and the ability to interconnect with other GSN distributions around the world, which means that any other networks, for example a weather station, could provide data to our platform in order to easily add more channels.

7.3.4 Enhanced user management

At the moment, DrEWE connects to an internal database to manage all the information related to the users, but rules happens to every user for equal. It will be a good line of work to have complete user management from the rule composer to the actions, which will provide situations like rules that are only activated for a limited group users, rules recommendation and more flexibility. Another approach would be to directly to integrate the existing user management system as one service or channel of the whole platform that serves the information when needed.

Appendix A

Complete rdf channel implementation

This appendix presents the complete implementation of the rdf channels explained at section 6.7. The follow the ewe ontology and represents a good example about the usage and importance of this ontology.

A.1 WallDisplay Channel

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
   xmlns:dcterms="http://purl.org/dc/terms/"
   xmlns:ewe="http://gsi.dit.upm.es/ontologies/ewe/ns/"
   xmlns:ewe-meeting="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
       channel/"
   xmlns:foaf="http://xmlns.com/foaf/0.1/"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdfs= "http://www.w3.org/2000/01/rdf-schema#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/WallDisplay">
    <rdfs:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
       ns/Channel"/>
    <dcterms:description>This channel.</dcterms:description>
    <dcterms:title>Wall Display Channel</dcterms:title>
```

```
    <ewe:supportedBy rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/DrEwe" />
    <!-- Action list -->
    <ewe:providesAction rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/ShootAndShow"/>
    <ewe:providesAction rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/DisplayMessage"/>
  </owl:Class>

  <!-- Action definitions -->
  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/ShootAndShow">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Action"/>
    <dcterms:title>Shoot and show picture</dcterms:title>
    <dcterms:description>This actions takes a pictute with the camera and
         displays it on the screen. The camera used is that associated to
        the screen. If there is none, no action is taken.</dcterms:
        description>
    <!-- Configuration -->
    <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
        /ewe/ns/meeting-channel/params/WhichDisplay">
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/DisplayMessage">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Action"/>
    <dcterms:title>Display message</dcterms:title>
    <dcterms:description>This actions shows a message on the screen</
        dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
        /ewe/ns/meeting-channel/params/WhichDisplay">
    <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
        /ewe/ns/meeting-channel/params/message">
  </owl:Class>


  <!-- Common Parameters -->
  <owl:Class>
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        InputParameter"/>
    <dcterms:title>Which display?</dcterms:title>
```

```
    <dcterms:description>The name of the display to use.</dcterms:
        description>
    <dcterms:type>string</dcterms:type>
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/params/Description">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
        /Parameter"/>
    <dcterms:title>message</dcterms:title>
    <dcterms:description>The message to show on the screen.</dcterms:
        description>
    <dcterms:type>string</dcterms:type>
  </owl:Class>

</rdf:RDF>
```

Listing A.1: Meeting channel rdf complete specification

A.2 Meeting Channel

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
   xmlns:dcterms="http://purl.org/dc/terms/"
   xmlns:ewe="http://gsi.dit.upm.es/ontologies/ewe/ns/"
   xmlns:ewe-meeting="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
       channel/"
   xmlns:foaf="http://xmlns.com/foaf/0.1/"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:rdfs= "http://www.w3.org/2000/01/rdf-schema#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/MeetingChannel">
    <rdfs:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/Channel"/>
    <dcterms:description>This channel provides general real-time
        informaton about the meeting and some actions to schedule or
        cancel a meeting. </dcterms:description>
    <dcterms:title>Meeting Channel</dcterms:title>
```

```
    <ewe:supportedBy rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/DrEwe" />
    <!-- Event list -->
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/AnyMeetingStart"/>
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/StartStartWithName"/>
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/StartMeetingAtLocation"/>
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/MissingAttende"/>
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/MeetingEnd"/>
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/MeetingCancelled"/>
    <ewe:generatesEvent rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/MeetingEndingTime"/>
    <!-- Action list -->
    <ewe:providesAction rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/AddMeeting"/>
    <ewe:providesAction rdf:resource="http://gsi.dit.upm.es/ontologies/
        ewe/ns/meeting-channel/CancelMeeting"/>
  </owl:Class>

  <!-- Event definitions -->
  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/AnyMeetingStart">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Event"/>
    <dcterms:title>The meeting starts</dcterms:title>
    <dcterms:description>This trigger fires every time a new meeting
        starts. A meeting starts when the time of the meeting arrived and
        all the attendees have already logged in at the meeting room.</
        dcterms:description>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
```

```
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/StartStartWithName">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Event"/>
    <dcterms:title>Meeting with name starts</dcterms:title>
    <dcterms:description>This trigger fires every time a new meeting with
         the name given starts. A meeting starts when the time of the
        meeting arrived and all the attendees have already logged in at
        the meeting room.</dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter>
      <owl:Class>
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/InputParameter"/>
          <dcterms:title>Which meeting?</dcterms:title>
          <dcterms:description>The name of the meeting. This parameter
              filters the meetings that triggers the event.</dcterms:
              description>
          <dcterms:type>string</dcterms:type>
        </owl:Class>
    </ewe:hasInputParameter>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/StartMeetingAtLocation">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Event"/>
    <dcterms:title>Start meeting at location</dcterms:title>
```

```
    <dcterms:description>This trigger fires every time a new meeting at
        the location given starts. A meeting starts when the time of the
        meeting arrived and all the attendees have already logged in at
        the meeting room.</dcterms:description>
    <!-- Configuration -->
    <ewe:hasInputParameter>
      <owl:Class>
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/InputParameter"/>
        <dcterms:title>Meeting location</dcterms:title>
        <dcterms:description>The location of the meeting. This parameter
            filters the meetings that triggers the event</dcterms:
            description>
        <dcterms:type>string</dcterms:type>
      </owl:Class>
    </ewe:hasInputParameter>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/MissingAttende">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Event"/>
    <dcterms:title>There is a missing attendee</dcterms:title>
    <dcterms:description>This trigger fires every time there is an
        attendee is missing a meeting. It is considered that an attendee
        is missing a meeting when he/she has not logged in at the meeting
        room after 10 minutes of courtesy past the meeting time.</dcterms:
        description>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter>
      <owl:Class>
```

```
        <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/OutputParameter"/>
        <dcterms:title>Missing attendees</dcterms:title>
        <dcterms:description>The list of attendees that didn't attended
            the meeting</dcterms:description>
        <dcterms:type>string</dcterms:type>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/MeetingCancelled">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        Event"/>
    <dcterms:title>A meeting was cancelled</dcterms:title>
    <dcterms:description>This trigger fires when a meeting has been
        cancelled.</dcterms:description>
    <!-- Output parameter list -->
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Attendees">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/StartTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/EndTime">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Description">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Title">
    <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
        ontologies/ewe/ns/meeting-channel/params/Location">
  </owl:Class>
```

```
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/MeetingEnd">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
      Event"/>
  <dcterms:title>A meeting ended</dcterms:title>
  <dcterms:description>This trigger fires when a meeting, that has
      already started, ends. A meeting ends, when all the participants
      log out the meeting room.</dcterms:description>
  <!-- Output parameter list -->
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Attendees">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/StartTime">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/EndTime">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Description">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Title">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Location">
</owl:Class>

<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/MeetingEndingTime">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
      Event"/>
  <dcterms:title>It's come Meeting ending time</dcterms:title>
  <dcterms:description>This trigger fires at the ending time of a
      meeting that previously started.</dcterms:description>
  <!-- Output parameter list -->
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Attendees">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/StartTime">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/EndTime">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Description">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Title">
  <ewe:hasOutputParameter rdf:resource="http://gsi.dit.upm.es/
      ontologies/ewe/ns/meeting-channel/params/Location">
</owl:Class>
```

```
<!-- Action definitions -->
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/AddMeeting">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
      Action"/>
  <dcterms:title>Add a new meeting</dcterms:title>
  <dcterms:description>This schedules a meeting with the data given</
      dcterms:description>
  <!-- Configuration -->
  <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
      /ewe/ns/meeting-channel/params/Attendees">
  <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
      /ewe/ns/meeting-channel/params/StartTime">
  <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
      /ewe/ns/meeting-channel/params/EndTime">
  <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
      /ewe/ns/meeting-channel/params/Description">
  <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
      /ewe/ns/meeting-channel/params/Title">
  <ewe:hasInputParameter rdf:resource="http://gsi.dit.upm.es/ontologies
      /ewe/ns/meeting-channel/params/Location">
</owl:Class>

<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/CancelMeeting">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
      Action"/>
  <dcterms:title>Cancel a meeting by name</dcterms:title>
  <dcterms:description>This actions will cancel the schedulled meeting
      that matches the name given</dcterms:description>
  <!-- Configuration -->
  <ewe:hasInputParameter>
    <owl:Class>
      <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
          ewe/InputParameter"/>
        <dcterms:title>Which meeting?</dcterms:title>
        <dcterms:description>The name of the meeting to cancel.</
            dcterms:description>
        <dcterms:type>string</dcterms:type>
      </owl:Class>
  </ewe:hasInputParameter>
</owl:Class>


<!-- Common Parameters -->
```

```
<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/params/Attendees">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
      /Parameter"/>
  <dcterms:title>attendees</dcterms:title>
  <dcterms:description>List of attendees for the meeting.</dcterms:
      description>
  <dcterms:type>list</dcterms:type>
</owl:Class>

<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/params/StartTime">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
      /Parameter"/>
  <dcterms:title>start time</dcterms:title>
  <dcterms:description>The start time of the meeting</dcterms:
      description>
  <dcterms:type>datetime</dcterms:type>
</owl:Class>

<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/params/EndTime">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
      /Parameter"/>
  <dcterms:title>end time</dcterms:title>
  <dcterms:description>The ending time of the meeting</dcterms:
      description>
  <dcterms:type>datetime</dcterms:type>
</owl:Class>

<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/params/Description">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
      /Parameter"/>
  <dcterms:title>description</dcterms:title>
  <dcterms:description>A description for the meeting</dcterms:
      description>
  <dcterms:type>string</dcterms:type>
</owl:Class>

<owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
    channel/params/Title">
  <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
      /Parameter"/>
  <dcterms:title>title</dcterms:title>
```

```
    <dcterms:description>The title of the meeting</dcterms:description>
    <dcterms:type>string</dcterms:type>
  </owl:Class>

  <owl:Class rdf:about="http://gsi.dit.upm.es/ontologies/ewe/ns/meeting-
      channel/params/Location">
    <rdf:subClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/ns
        /Parameter"/>
    <dcterms:title>location</dcterms:title>
    <dcterms:description>The location of the meeting</dcterms:description
        >
    <dcterms:type>string</dcterms:type>
  </owl:Class>

</rdf:RDF>
```

Listing A.2: Meeting channel rdf complete specification

A.3 Twitter Channel

```
<!DOCTYPE rdf:RDF system "http://www.w3.org/XML/9710rdf-dtd/rdf.dtd">

<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ewe="http://gsi.dit.upm.es/ontologies/ewe/ns/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:tags="http://www.holygoat.co.uk/owl/redwood/0.1/tags/"
  xmlns="http://gsi.dit.upm.es/ontologies/ewe/ns/">

  <owl:Class rdf:about="https://ifttt.com/twitter">
    <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/Channel"/>
    <dcterms:title>Twitter</dcterms:title>
    <dcterms:description>Twitter is a social networking and microblogging
         service that enables itsusers to send and read messages called
        tweets. Tweets are text-based posts ofup to 140 characters
        displayed on the authors profile page and delivered tothe authors
        subscribers, who are known as followers.</dcterms:description>
```

```
    <foaf:url>http://twitter.com</foaf:url>
    <foaf:logo>https://ifttt.com/images/channels/twitter_lrg.png</foaf:
        logo>
    <ewe:supportedBy rdf:resource="http://ifttt.com" />
    <ewe:supportedBy rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/DrEwe" />
    <!-- Event and action references -->
    <ewe:hasAction rdf:resource="https://ifttt.com/channels/twitter/
        actions/PostATweet" />
  </owl:Class>

  <owl:Class rdf:about="https://ifttt.com/channels/twitter/actions/
      PostATweet">
    <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/Action"/>
    <dcterms:title>Post a tweet</dcterms:title>
    <dcterms:description>This Action will post a new tweet to your
        Twitter account. NOTE: Please adhereto Twitters Rules and Terms of
         Service.</dcterms:description>
    <ewe:hasInputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/InputParameter"/>
        <dcterms:title>Whats happening?</dcterms:title>
      </owl:Class>
    </ewe:hasInputParameter>
  </owl:Class>

</rdf:RDF>
```

Listing A.3: Twitter channel rdf complete specification

A.4 Google Calendar Channel

```
<!DOCTYPE rdf:RDF system "http://www.w3.org/XML/9710rdf-dtd/rdf.dtd">

<rdf:RDF
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ewe="http://gsi.dit.upm.es/ontologies/ewe/ns/"
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:tags="http://www.holygoat.co.uk/owl/redwood/0.1/tags/"
  xmlns="http://gsi.dit.upm.es/ontologies/ewe/ns/">

  <owl:Class rdf:about="https://ifttt.com/google_calendar">
    <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/Channel"/>
    <dcterms:title>Google Calendar</dcterms:title>
    <dcterms:description>Google Calendar is a free time-management web
        application offered by Google.</dcterms:description>
    <foaf:url>https://www.google.com/calendar</foaf:url>
    <foaf:logo>https://ifttt.com/images/channels/google_calendar_lrg.png<
        /foaf:logo>
    <ewe:supportedBy rdf:resource="http://www.ifttt.com" />
    <ewe:supportedBy rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/DrEwe" />
    <!-- Event and action references -->
    <ewe:generatesEvent rdf:resource="https://ifttt.com/channels/
        google_calendar/triggers/AnyNewEventAdded" />
  </owl:Class>

  <owl:Class rdf:about="https://ifttt.com/channels/google_calendar/
      triggers/AnyNewEventAdded">
    <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/ewe/
        ns/Event"/>
    <dcterms:title>Any new event added</dcterms:title>
    <dcterms:description>This Trigger fires every time a new event is
        added to your Google Calendar.</dcterms:description>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>Title</dcterms:title>
        <dcterms:description>The event's title.</dcterms:description>
        <ewe:example>Practice Presentation</ewe:example>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>Description</dcterms:title>
```

```
        <dcterms:description>The event's description.</dcterms:
            description>
        <ewe:example>Make a presentation about new channels on ifttt</ewe
            :example>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>Where</dcterms:title>
        <dcterms:description>The location where the event takes place.</
            dcterms:description>
        <ewe:example>Building A, Room 101</ewe:example>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>Starts</dcterms:title>
        <dcterms:description>Date and time the event starts.</dcterms:
            description>
        <ewe:example>August 23, 2011 at 10:00PM</ewe:example>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>Ends</dcterms:title>
        <dcterms:description>Date and time the event ends.</dcterms:
            description>
        <ewe:example>August 23, 2011 at 11:00PM</ewe:example>
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>EventUrl</dcterms:title>
        <dcterms:description>A URL to view/edit the event.</dcterms:
            description>
        <ewe:example>https://www.google.com/calendar/event?eid=bmpmaDhnMm
            </ewe:example>
```

```
      </owl:Class>
    </ewe:hasOutputParameter>
    <ewe:hasOutputParameter>
      <owl:Class>
        <rdfs:SubClassOf rdf:resource="http://gsi.dit.upm.es/ontologies/
            ewe/ns/OutputParameter"/>
        <dcterms:title>CreatedAt</dcterms:title>
        <dcterms:description>Date and time the event was created.</
            dcterms:description>
        <ewe:example>August 01, 2013 at 11:00AM</ewe:example>
      </owl:Class>
    </ewe:hasOutputParameter>
  </owl:Class>

</rdf:RDF>
```

Listing A.4: Google Calendar channel rdf complete specification

Appendix B

Virtual sensor implementation

This appendix presents the complete implementation of the three virtual sensors used as example in DrEWE: RemotelightVS to control the light sensor, DniVS to handle the electronic dni reader and CalendarVS to pipe and address the meeting events

B.1 RemotelightVS

```
<virtual-sensor name="RemotelightVS" priority="11">

  <processing-class>
    <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
    <output-structure>
      <field name="value" type="int" />
    </output-structure>
  </processing-class>
  <description>Get data from remote light sensor</description
     >
  <life-cycle pool-size="100" />
  <addressing>
    <predicate key="geographical">Not yet specified</
       predicate>
```

```
    <predicate key="eventType">LightEvent</predicate>
  </addressing>
  <storage history-size="2h" />
  <streams>
    <stream name="input1">
      <source alias="source1" sampling-rate="1" storage-size=
          "1">
        <address wrapper="remote-direct">
          <predicate key="notification-id">1.2</predicate>
          <output-structure>
            <field name="value" type="int" />
          </output-structure>
        </address>
        <query>select * from wrapper</query>
      </source>
      <query>select * from source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

Listing B.1: RemotelightVS implementation

B.2 RemoteDniVS

```
<virtual-sensor name="RemoteDniVS" priority="11">

  <processing-class>
    <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
    <output-structure>
      <field name="number" type="int" />
      <field name="name" type="varchar(60)" />
    </output-structure>
  </processing-class>
  <description>Get data from dni sensor</description>
```

```
  <life-cycle pool-size="100" />
  <addressing>
    <predicate key="geographical">Not yet specified</
       predicate>
    <predicate key="eventType">DniEvent</predicate>
  </addressing>
  <storage history-size="2h" />
  <streams>
    <stream name="input1">
      <source alias="source1" sampling-rate="1" storage-size=
         "1">
        <address wrapper="remote-direct">
          <predicate key="notification-id">1.3</predicate>
          <output-structure>
            <field name="number" type="int" />
            <field name="name" type="varchar(60)" />
          </output-structure>
        </address>
        <query>select * from wrapper</query>
      </source>
      <query>select * from source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

Listing B.2: RemoteDniVS implementation

B.3 CalendarVS

```
<virtual-sensor name="CalendarVS" priority="11">

  <processing-class>
    <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
```

```
    <output-structure>
      <field name="title" type="varchar(60)" />
      <field name="attendees" type="varchar(200)" />
      <field name="start" type="varchar(60)" />
    </output-structure>
  </processing-class>
  <description>Get data from calendar</description>
  <life-cycle pool-size="100" />
  <addressing>
    <predicate key="geographical">Not yet specified</
       predicate>
    <predicate key="eventType">CalendarEvent</predicate>
  </addressing>
  <storage history-size="2h" />
  <streams>
    <stream name="input1">
      <source alias="source1" sampling-rate="1" storage-size=
         "1">
        <address wrapper="remote-direct">
          <predicate key="notification-id">1.4</predicate>
          <output-structure>
            <field name="title" type="varchar(60)" />
            <field name="attendees" type="varchar(200)" />
            <field name="start" type="varchar(60)" />

          </output-structure>
        </address>
        <query>select * from wrapper</query>
      </source>
      <query>select * from source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

Listing B.3: CalendarVS implementation

Appendix C

Developers manual

In the repository[1] you can find the detailed information to make the platform works. Since it is a big project that uses several technologies the process of installation in a developer's way can be a little bit tedious.

In terms of implementation, the project has been divided in five different subprojects that coincide with the subtrees of the repository. This projects are: Berries-DrEWE, Drools-DrEWE, GCalendar-DrEWE, GSN-DrEWE and NodeEvented.

C.1 Berries-DrEWE

Raspberry's scripts and modules that communicate with GSN and/or SPIN to produce events, make actions and handle requests.

Retrieving the dni log

The script in charge of this task is dni.sh. This script checks the last person that has inserted its dni at the laboratory door, sends the info to the GSN server at the given url and echoes the info via terminal.

The raspberry pi and the dni machine must be known ssh hosts of each other. In order to make this task iterative we execute it by dniLoop.sh. This script is necessary due to the nature of the dni server, that changes its logs in an unpleasant way.

[1] `https://github.com/gsi-upm/DrEWE`

```
Execution: ./dniLoop.sh
```

Retrieving the light level

This Python script sends the light level to a GSN server. The data acquisition is made via a RC circuit attached to a given pin, because of the lack of analog inputs in the raspberry, this script sets a given entry as low and counts the loop's cycles that it spends discharging.

Regarding the circuit, the trick is to time how long it takes a point in the circuit the reach a voltage that is great enough to register as a "High" on a GPIO pin.

A detailed explanation can be found in `http://www.raspberrypi-spy.co.uk/2012/08/reading-analogue-sensors-with-one-gpio-pin/`

```
Execution: sudo ./python light.py &
```

Motion

In order to handle the camera we use a modified version of the motion packet that can be found here:`https://github.com/dozencrows/motion/tree/mmal-test`. This packet automatically sets up a http server that controls the camera and several other features. Regarding the installation of the packet:

```
$ sudo apt-get install -y libjpeg62 libjpeg62-dev libavformat53 libavformat-dev libavcodec53
$ wget https://www.dropbox.com/s/xdfcxm5hu71s97d/motion-mmal.tar.gz
$ tar zxvf motion-mmal.tar.gz
```

And regarding the configuration:

Replace motion-mmalcam.conf with the one given at this repository. With the given configuration, it will deploy an http server at port 8080 that allows us to control the camera remotely, will take snapshots periodically and place them in the imageServer.py directory, in order to serve the latest snapshot via http (accesible from any point of the network).

In terms of control, Motion provides several methods to control the camera, all of them documented at the Motion http API(`http://www.lavrsen.dk/foswiki/bin/view/Motion/MotionHttpAPI`). For example, if you want to force a raspberry snapshot from any other machine in the network (assume that raspberry's ip is 192.168.1.132)

GET 192.168.1.132:8080/0/action/snapshot

```
Execution: $ sudo ./motion -n -c motion-mmalcam.conf &
```

Image Server

Motion provides several amazing features to control the camera, one of them is the video straming. However we only needed motion to take pictures, nothing to do with video.

imageServer.py is a simple script that takes the last picture taken by motion and serves it to the world (in our case, just to the network) via http, creating a http server with python module BaseHTTPServer.

```
Execution: python imageServer.py &
```

C.2 Drools-DrEWE

Drools module for DrEWE project. It launches the drools environment, the SPIN module and the GSNToExpert module. It also is in charge of loading both drools and SPIN rules.

GSN To Expert This module is in charge of retrieving events from GSN and inserting them into the drools rule engine. All the data in GSN is accessible via HTTP, so it is retrieved via GET requests and inserted in the Drools engine via an 'entry point', which is is one of the features of Drools Fusion, the CEP (complex event processing) module of the Drools suite.

Drools module

Once the events has been inserted into the drools engine, the rules can be triggered at real time. This rules can produce other events, direct actions that are sent to the next module or SPIN events that will be handle by the SPIN module.

SPIN module

The SPIN notation is supported by the EWE ontology[10], which is used by DrEWE to represent events, actions and rules.

Installation

This project can be installed as a tipical eclipse project, but there is some points that require special attention: Drools environment and SPIN API

Firstly, you need to set up the drools environment in your machine. A detailed set of instructions can be found at `http://docs.jboss.org/drools/release/5.2.0.Final/droolsjbpm-introduction-docs/html/ch03.html`

The SPIN api by Topbraid can be easily specified and integrated as a Maven Dependency.

So if your eclipse has the m2e plugin, you just have to mark this project as a 'maven project' in order to retrieve the dependencies at the pom.xml file.

Under the 'es.upm.dit.gsi.DrEwe.Main' packet we can find two Init classes.

- DroolsInit.java: will launch this module that needs GSN to be running at the default but configurable port
- DroolsInitTest.java: will launch the module without the need of GSN to be running. Furthermore, I've developed a suite of test for the drools' fusion rules that comes in form of a sequence of insertion of events and management of the drools' pseudoclock

C.3 GCalendar-DrEWE

This module is a Node.js module that simplifies the use of RESTful Google Calendar API without any interaction with the user, retrieve all events on a given calendar and send them to a GSN server. Before sending the events, it checks if it has already been added

How to Install:

```
npm install
```

How to use:

```
node GCalendar.js
```

Inside the config.js you can find six important parameters:

- consumer key: Client ID for your project, you'll obtain it once you've registered your project in the API Console
- consumer secret: Client secret for your project, same as above
- redirect url: you must grant access to this url in the API Console,
- access token: access token for your application,
- refresh token: refresh token for your application,
- calendar Id: your calendar's ID, it can be obtained from google calendar normal service
- refresh time: time in milliseconds to check for new events

Despite there are several methods to retrieve the token, I hardly encourage to go to Google Oauth Playground at `https://developers.google.com/oauthplayground/` and configure it to "Use your own OAuth credentials"

C.4 GSN-DrEWE

Installing to Run and debug GSN in Eclipse. Here they are the installing steps to get GSN up and running. You can find a more detailed documentation at `http://sourceforge.net/apps/trac/gsn/wiki/install-gsn`

- Download and install Eclipse SDK.
- Start Eclipse.
- Download and install the Subclipse (http://subclipse.tigris.org/install.html)
- File -> Import -> Other -> Checkout Projects from SVN.
- Check “Create a new repository location”.
- Paste the repository location http://gsn.svn.sourceforge.net/svnroot/gsn
- Select "trunk" and click "Next".
- Select "Check out project configured using the New Projects Wizard" and click "Finish".
- In the New Projects Wizard select "Java Project" and click "Next".
- In the New Java Projects Wizard.
- Enter the project name, select "Create new project in workspace".
- Select "Create separate folders . . . " and click on "Configure default".
- If the project doesn't contains a "src" directory (depends on the Eclipse version you are using), assure to create one by clicking on the "Create new s ource folder".
- In Build Path preferences select "Folders", enter "build/classes" as the output folder name and click "OK".
- Click "Finish".
- Click "Ok" to confirm overwrite of non standard resources.

- Wait for the files to be downloaded from the repository.
- To add library files to the build path. - Project -> Properties. - In the Properties dialog select "Java Build Path". - In the Java Build Path dialog, select the "Libraries" tab. - On the Libraries tab, click on "Add jars . . . ". - Add only the .jar files in the lib directory and its sub-folders ; do not add any LICENSE text files
- check whether there are any build errors, which prevent the java compiler from building GSN - note: some errors are safe to ignore, but need compiler configuration changes (namely "Access restriction" errors) - in the main menu select "Window" > "Preferences". a configuration popup should appear - in the menu of the popup select "Java -> Compiler -> Errors/Warnings". you should see a list of possible warning/error levels for the java compiler - in the list go to "Deprecated and restricted API -> Forbidden reference (access rules)" and change it to "Warning"

GSN is now ready to Run.

C.5 NodeEvented

Node.js module in charge of generate events and process actions. Connected with Drools module and GSN module.

It is written in node and uses the main advantage of this modern programming language: the low latency. So once a rule is triggered under the Drools or SPIN engine, it only will take fractions of a second to perform high level actions such as post a tweet, order the camera to take a photo or make the bot talk something.

How to Install: npm install

How to use: node app.js

This node module is able to:

- deploy a conversational bot at GET /bot
- post tweets to botgsi twitter account at POST /post-tweet?query.tweet=
- send en email to a given set of directions at POST /email?text=""&subject=""&to="... , ... , ..."
- send light events to Drools via socket.io at POST /light?body=

- send events to Drools by pressing buttons via socket.io at GET /
- an old but pretty drools' rules composer at GET /composer that send them to the drools module via socket.io

How to configure:

Inside the config.js you can find six important parameters:

- consumer key: consumer key for twitter,
- consumer secret: consumer secret for twitter,
- access token key: access token for twitter for the user who is going to post tweets,
- access token secret: access token secret, same as above,
- email user: email who will send the message,
- email password: password for the email

Bibliography

[1] A. Dohr, R. Modre-Opsrian, M. Drobics, D. Hayn, and G. Schreier. The internet of things for ambient assisted living. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 804–809, 2010.

[2] Yu-Ju Tu, Wei Zhou, and Selwyn Piramuthu. Identifying rfid-embedded objects in pervasive healthcare applications. *Decision Support Systems*, 46(2):586–593, 2009.

[3] A.J. Jara, M.A. Zamora, and A.F.G. Skarmeta. An architecture based on internet of things to support mobility and security in medical environments. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–5, 2010.

[4] G.R. Gonzalez, M.M. Organero, and C.D. Kloos. Early infrastructure of an internet of things in spaces for learning. In *Advanced Learning Technologies, 2008. ICALT '08. Eighth IEEE International Conference on*, pages 381–383, 2008.

[5] Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. The internet of things in an enterprise context. In *Future Internet–FIS 2008*, pages 14–28. Springer, 2009.

[6] Rao Yuan, Lu Shumin, and Yang Baogang. Value chain oriented rfid system framework and enterprise application, 2007.

[7] Stephan Karpischek, Florian Michahelles, Florian Resatsch, and Elgar Fleisch. Mobile sales assistant-an nfc-based product information system for retailers. In *Near Field Communication, 2009. NFC'09. First International Workshop on*, pages 20–23. IEEE, 2009.

[8] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, LMSD Souza, and Vlad Trifa. Soa-based integration of the internet of things in enterprise services. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 968–975. IEEE, 2009.

[9] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Global sensor networks. *EPFL, Lausanne, Tech. Rep*, 2006.

[10] Miguel Coronado and Carlos A. Iglesias. Ewe ontology specification.

[11] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[12] Spin - sparql inferencing notation.