Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de un dispositivo para el control de instalaciones domóticas mediante reglas de usuario

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: D. Abelardo Codoñer Llopis

Tutores: D. Alberto Bonastre Pina
D. Rafael Ors Carot

2014 - 2015

A Eva, por estar siempre

Resumen

Existen diversos estándares domóticos, de los cuales uno de los de mayor implantación es KNX. El diseño y configuración de las instalaciones KNX se realiza mediante un paquete de software propietario, disponible solo bajo una costosa licencia y destinado a ser usado por instaladores. Así pues, el usuario final de la instalación cuenta con posibilidades limitadas a la hora de configurar el comportamiento de su vivienda, teniendo que recurrir al instalador si desea realizar cualquier cambio.

El presente trabajo tiene como objetivo el diseño e implementación de un dispositivo de reducido coste que permita al usuario de una instalación domótica KNX la definición de un determinado comportamiento automático en su hogar de forma intuitiva y sencilla.

La parte *software* del dispositivo consiste en una aplicación WEB que permite de manera amigable definir reglas de comportamiento de la forma "SI condición ENTONCES acción". Dichas reglas son posteriormente ejecutadas sobre la instalación KNX haciendo uso de la metodología de las Redes de reglas. Como soporte *hardware* del dispositivo, y con el objeto de minimizar el consumo, se emplea un ordenador monoplaca, de tamaño reducido y bajo coste, en el que se ha instalado un servidor WEB.

Palabras clave: sistemas de control, inteligencia ambiental, redes domóticas, Redes KNX, sistemas basados en reglas, redes de reglas.

Abstract

There are several home automation standards, being KNX one of the most widely established. Both the design and configuration of KNX installations are done by means of a proprietary software package only available under an expensive license and intended to be used by installers. Thus, the final user of the installation has limited freedom to configure the behavior of its own home. He depends on the installer if he wishes to make any changes.

The aim of the present work is to design and implement a low cost device to provide the user a simple way to define the desired automatic behavior of its home by means of Rule Nets. It will be attached to a home automation system based on KNX technology.

The software of the device consist on a user-friendly web application that allows the definition of behavior rules such as "IF condition THEN action". These rules are expressed as a Rule Net in order to be run on the home installation. A web server is required to provide the above application. In order to minimize energy consumption, a low cost single board computer has been used to provide this service.

Keywords: control systems, environmental intelligence, home automation networks, KNX networks, rule-based systems, rule nets.

Tabla de contenidos

1 Introducción **15**

1.1 Entorno y justificación *15*

1.2 Objetivos *16*

1.3 Descripción de contenidos *17*

2 Base teórica **19**

2.1 Introducción *19*

2.2 KNX *19*

2.2.1 Topología de la red KNX 19

2.2.2 Acceso al medio 20

2.2.3 Direccionamiento 20

2.2.4 Datapoints 22

2.2.5 Objetos de comunicación 23

2.2.6 Estructura de un telegrama 24

2.2.7 Conectividad con el bus KNX desde el exterior 25

2.2.7.1 La librería Calimero 25

2.3 Introducción a las redes de reglas *28*

2.3.1 Variables y hechos 28

2.3.2 Reglas 28

2.3.3 Representación gráfica 28

2.3.4 Representación textual 29

2.3.5 Representación matricial 29

2.3.6 Sensibilización de una regla 30

2.3.7 Disparo de una regla sensibilizada 30

2.3.8 Verificación de la red de reglas 31

3 Análisis **33**

3.1 Análisis de requerimientos *33*

3.2 Esquema de bloques funcionales *34*

3.3 Selección de software y hardware *35*

3.3.1 Formato de datos 35

3.3.2 Aplicación WEB 36

3.3.3 Ordenadores monoplaca 38

4 Diseño 41

4.1 Introducción 41

4.2 Formato de datos 41

4.3 Representación de la instalación KNX en formato XML 43

4.4 Traducción a variables y hechos 44

4.5 Representación de la red de reglas en XML 46

4.6 Ejecución de las reglas 46

4.7 Diseño de la aplicación WEB 47

4.7.1 Funcionalidades 47

4.7.2 Diseño de la interfaz de usuario 49

4.7.3 Diagrama de navegación 53

5 Implementación 55

5.1 Introducción 55

5.2 Descripción de la página JSF 56

5.3 Configuración del aspecto. 57

5.4 Descripción de los managed bean 58

5.5 Representación en Java de la instalación KNX 61

5.6 Representación en Java de la red de reglas 61

5.7 Conversión Java <-> Xml 61

5.8 Conversión a variables y hechos 66

5.9 Conexión con KNX 68

5.10 Detección de eventos KNX 72

5.11 Ejecución de las reglas 75

5.12 Ejecución de las órdenes KNX 81

5.13 Despliegue de la aplicación WEB en la BeagleBone Black 83

5.13.1 Puesta en marcha 83

5.13.2 Instalación de *Tomcat* versión 7 84

5.13.3 Instalación de jdk versión 1.8.33 86

5.13.4 Despliegue de la aplicación 86

6 Pruebas de uso 89

6.1 Introducción 89

6.2 Identificación KNX 89

6.3 Representación de la instalación en formato XML 93

6.4 Prueba de reglas *95*

6.4.1 Control de iluminación. 95

6.4.2 Oscilaciones........ 96

6.4.3 Múltiples antecedentes 97

6.5 Representación de la red de reglas en XML........ *99*

7 Conclusiones y futuras ampliaciones 101

Bibliografía 103

ANEXO I: Manual del programador 105

Introducción *105*

Ficheros de configuración *105*

Implementación de la interfaz de usuario........ *106*

Mensajes emergentes........ 106

Pantalla principal 107

Editor de variables 116

Editor de reglas 119

ANEXO II: Manual de usuario 127

Introducción *127*

Pantalla principal *127*

Cargar un fichero de instalación *128*

Editar el nombre de las variables *129*

Cargar un fichero de reglas........ *131*

Crear una nueva regla........ *133*

Editar una regla existente *137*

Eliminar una regla existente........ *138*

Ejecutar reglas *138*

Guardar la red de reglas en un fichero........ *141*

Lista de figuras

Figura 1: Áreas de aplicación de la domótica 15
Figura 2: Topología de la red KNX 20
Figura 3: Estructura de una dirección individual en KNX 21
Figura 4: Estructura de una dirección de grupo de tres niveles 21
Figura 5: Estructura de un datapoint en KNX 23
Figura 6: Estructura de un telegrama KNX 24
Figura 7: Dispositivo pasarela entre red IP y KNX WEINZIERL mod. 730 25
Figura 8: Arquitectura de la librería Calimero 26
Figura 9: Representación gráfica de una red de reglas 29
Figura 10: Estructura general del dispositivo a implementar 33
Figura 11: Bloques funcionales del dispositivo a implementar 34
Figura 12: Arquitectura de la aplicación WEB utilizando JSF 37
Figura 13: Diagrama de bloques de la BeagleBone Black 38
Figura 14: Conectores, pulsadores e indicadores en la BeagleBone Black 40
Figura 15: Diagrama de funcionamiento de la ejecución de reglas 47
Figura 16: Prototipo - Pantalla principal 49
Figura 17: Prototipo - Menú de la pantalla principal 50
Figura 18: Prototipo - Pantalla del editor de variables 50
Figura 19: Prototipo - Menú del editor de variables 50
Figura 20: Prototipo - Pantalla del editor de reglas 51
Figura 21: Prototipo - Menú del editor de reglas 51
Figura 22: Prototipo - Diálogo para la carga de un fichero de instalación 51
Figura 23: Prototipo - Diálogo para la carga de un fichero de reglas 52
Figura 24: Prototipo - Diálogo para editar el nombre de una variable 52
Figura 25: Prototipo - Diálogo para establecer antecedentes / consecuentes 52
Figura 26: Prototipo - Diálogo para conexión a KNX y ejecución de reglas 53
Figura 27: Diagrama de navegación de la aplicación WEB 53
Figura 28: Diagrama de clases de los managed bean 60
Figura 29: Diagrama de clases de la instalación KNX 61
Figura 30: Diagrama de clases de la red de reglas 61
Figura 31: Gestor de aplicaciones WEB de Tomcat 87

Lista de tablas

Tabla 1: Ejemplo de direcciones de grupo basadas en la aplicación 22
Tabla 2: Ejemplo de direcciones de grupo basadas en la ubicación...............................22
Tabla 3: Ejemplos de datapoint 23
Tabla 4: Conversión de domoelementos a variables y hechos..45
Tabla 5: Elementos de la instalación KNX para pruebas...93

1 Introducción

1.1 Entorno y justificación

La Asociación Española de Domótica e Inmótica[1] (CEDOM) se refiere a la domótica como el conjunto de tecnologías aplicadas al control y a la automatización inteligente de la vivienda, lo cual facilita una gestión eficiente del uso de la energía, aporta seguridad, confort y además permite la comunicación entre el usuario y el sistema.

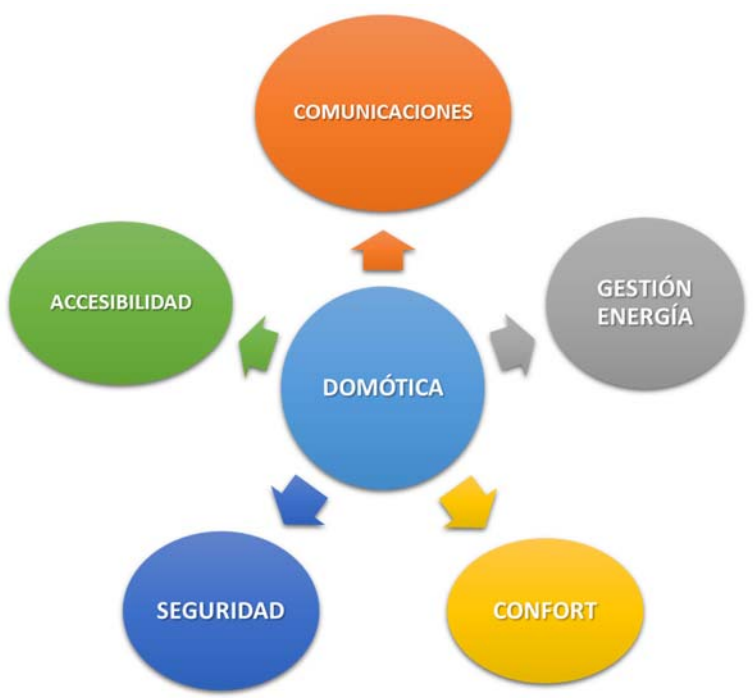


Figura 1: Áreas de aplicación de la domótica

La domótica ayuda al ahorro energético mediante la gestión inteligente de la iluminación, climatización, consumo de agua, etc. Proporciona servicios de teleasistencia para personas que lo necesiten y fomenta la accesibilidad facilitando el manejo de los elementos del hogar a personas discapacitadas. Además, aporta seguridad al proporcionar vigilancia automatizada de personas, animales y bienes mediante controles de intrusión, simulación de presencia, cámaras de vigilancia y a través de alarmas técnicas que permiten detectar incendios, fugas de gas, inundaciones de agua, fallos del suministro eléctrico, etc.

Asimismo, un sistema domótico puede acceder a redes exteriores de comunicación o información, permitiendo al usuario el control y supervisión remota de la vivienda a través de un PC o de dispositivos móviles tales como tabletas o teléfonos inteligentes.

[1] La Inmótica es equivalente a la domótica, pero aplicada a edificios no destinados a vivienda, como hoteles, centros comerciales, escuelas, universidades, hospitales y todos los edificios terciarios.

Existen diversos estándares domóticos de los cuales uno de los de mayor implantación es KNX, siendo España el segundo país del mundo en número de KNX *partners* solo por detrás de Alemania. KNX nace en 1999 de la convergencia de tres estándares previos: *European Installation Bus* (EIB), *European Home System* (EHS) y Batibus. Se trata de un sistema descentralizado que utiliza como medio de conexión entre sus dispositivos un bus común, lo cual simplifica tanto su instalación, como sus posteriores ampliaciones o modificaciones. Existe una gran variedad de dispositivos para aplicaciones de todo tipo que son proporcionados por múltiples fabricantes. Afortunadamente, KNX cuenta con un proceso de certificación que garantiza la interoperabilidad, es decir, el funcionamiento y la comunicación entre los dispositivos con independencia de cuál sea su fabricante.

El diseño y configuración de las instalaciones KNX se realiza mediante la herramienta ETS (*Engineering Tool Software*), un paquete de software propietario disponible bajo una costosa licencia y destinado a ser usado por instaladores. Así pues, el usuario final de la instalación cuenta con posibilidades limitadas a la hora de poder configurar el comportamiento de su vivienda, teniendo que recurrir al instalador si desea realizar cualquier cambio. Sería por tanto de gran utilidad, disponer de un mecanismo que permitiera a un usuario definir y modificar de manera sencilla comportamientos automáticos en su vivienda.

En principio, la forma más natural de plasmar un automatismo consiste en reglas del estilo "SI condición ENTONCES acción". Este tipo de automatismo se puede implementar mediante el uso de una metodología denominada Redes de Reglas, que se sitúa entre los sistemas expertos basados en reglas y los automatismos modelados en forma de Redes de Petri.

1.2 Objetivos

El presente trabajo tiene como objetivo el diseño e implementación de un dispositivo de reducido coste, que permita al usuario de una instalación domótica KNX definir, de manera sencilla, un determinado comportamiento automático en su hogar.

La parte software del dispositivo consistirá en una aplicación que permita de manera amigable definir reglas de funcionamiento en la forma "SI condición ENTONCES acción". Posteriormente se deberá poder ejecutar dichas reglas sobre la instalación haciendo uso de la metodología de las redes de reglas. Para que el usuario pueda utilizar la aplicación desde cualquier tipo de dispositivo, ya sea de manera local o remota, se propone implementar esta parte bajo la forma de una aplicación WEB. Dicha aplicación WEB deberá emplear la pasarela de conexión entre la red IP y el bus KNX para poder examinar el estado de la instalación y actuar sobre ella.

Para la parte hardware del dispositivo, será preciso un ordenador con un servidor WEB instalado que ofrecerá la aplicación antes citada. Con el objeto de minimizar el

consumo de energía, se propone optar por un ordenador monoplaca de tamaño reducido.

1.3 Descripción de contenidos

El restante contenido de la memoria se estructura en los siguientes capítulos y anexos:

Capítulo 2: Base teórica
Se van a tratar los conceptos que sirven de base al presente trabajo. Se comenzará por una descripción de los aspectos más importantes relacionados con el sistema KNX. Para terminar, se realizará una introducción a la metodología de las Redes de reglas para el diseño de automatismos

Capítulo 3: Análisis
Se estudia la manera de abordar el problema y las posibles tecnologías a emplear.

Capítulo 4: Diseño
Se detallan las decisiones de diseño tomadas, abarcando aspectos como la estructura básica del dispositivo, el mecanismo para la ejecución de las reglas o el diseño de la interfaz de usuario.

Capítulo 5: Implementación
Se describe cómo se han puesto en práctica las decisiones de diseño expuestas en el capítulo 4; indicando las herramientas y tecnologías empleadas, describiendo las partes más importantes del código y detallando el proceso para la puesta en marcha de la aplicación WEB sobre el ordenador monoplaca.

Capítulo 6: Pruebas de uso
Se muestra el funcionamiento del dispositivo sobre una instalación KNX.

Capítulo 7: Conclusiones y futuras ampliaciones
Para finalizar, se enuncian las conclusiones del trabajo y se enumeran una serie de posibles ampliaciones.

Anexo I: Manual del programador
Se tratan aspectos relacionados con la implementación de la aplicación WEB que no se han desarrollado en el capítulo 5.

Anexo II: Manual de usuario
Se describe el manejo de las distintas funcionalidades que ofrece el dispositivo implementado.

2 Base teórica

2.1 Introducción

En este capítulo se van a tratar los conceptos que sirven de base al presente trabajo. Se comenzará por una descripción de los aspectos más importantes relacionados con el sistema KNX. Y para terminar, se realizará una introducción a la metodología de las Redes de reglas para el diseño de automatismos.

2.2 KNX

KNX es un sistema tipo bus, descentralizado y controlado por eventos. En él todos los dispositivos tienen su propio microprocesador y electrónica de acceso al medio. Existen principalmente cuatro tipos de dispositivos:

- Fuentes de alimentación.
- Acopladores de línea o de área para interconectar diferentes segmentos de la red.
- Elementos sensores que detectan eventos procedentes de pulsadores o debidos a cambios en la luminosidad, la temperatura, la humedad, movimientos, etc.
- Elementos actuadores que se encargan de ejecutar las órdenes adecuadas, llevando a cabo la activación y regulación de cargas.

2.2.1 Topología de la red KNX

La red KNX tiene una estructura jerárquica, siendo la **línea** la unidad mínima de la instalación. Cada línea requiere de una fuente de alimentación para proporcionar energía a los dispositivos conectados a ella, el número máximo de dispositivos que puede alimentar una fuente es de 64. Es factible conectar más dispositivos añadiendo líneas adicionales que se conectarán a una **línea principal** mediante **acopladores de línea**. Se pueden acoplar hasta 15 líneas en la línea principal, constituyendo así lo que se denomina **área**. A su vez, es posible unir hasta un total de 15 áreas mediante el uso de **acopladores de área** que se conectan a la línea ***backbone***.

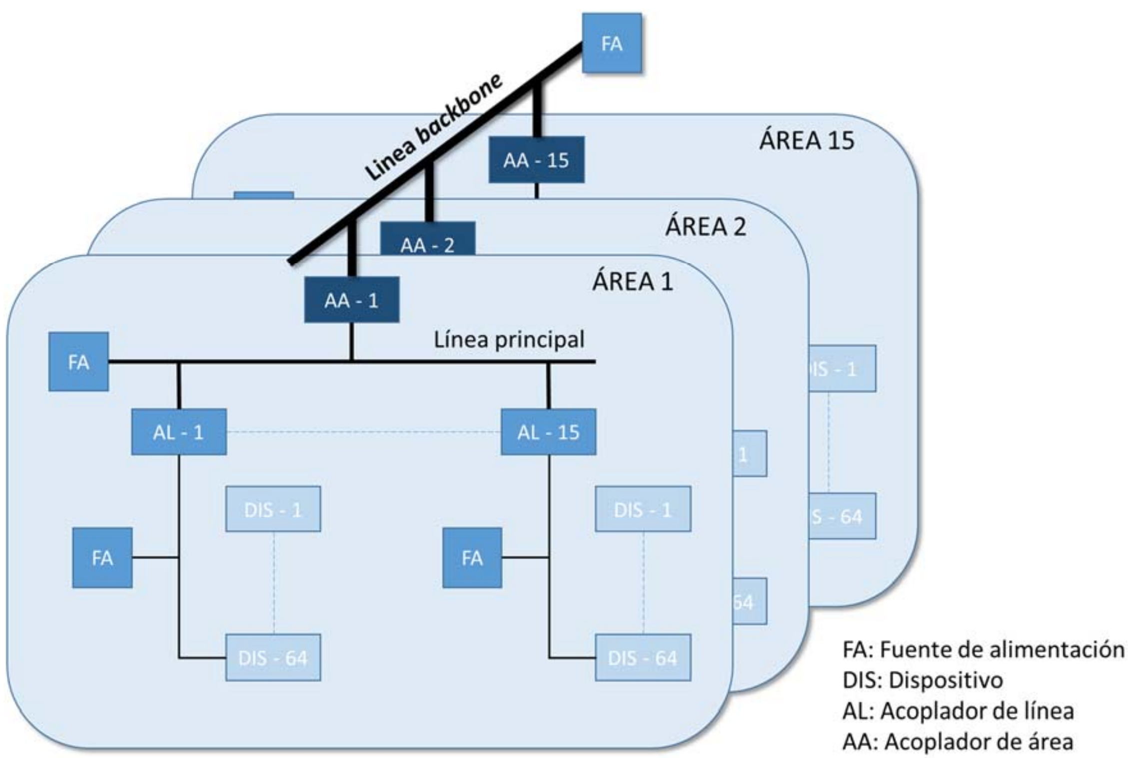


Figura 2: Topología de la red KNX

2.2.2 Acceso al medio

La información se trasmite de forma simétrica por un par trenzado. El "0" lógico se codifica mediante la transmisión de un pulso de tensión, mientras que la no transmisión de ningún pulso codifica el "1" lógico.

Al utilizar un medio de transmisión compartido, se precisa de un mecanismo de arbitraje, que en el caso de KNX es de tipo CSMA/CA. Cuando un dispositivo desea transmitir, primero debe comprobar que el bus está libre; en caso de que el bus esté ocupado, deberá esperar. Si dos dispositivos comienzan a emitir en el mismo instante, se arbitra mediante bits dominantes (0) y recesivos (1), de forma que gana el medio el que primero coloque un bit dominante. En caso de igualdad de prioridad, comenzará aquel cuya dirección física sea más baja.

2.2.3 Direccionamiento

En la red KNX cada dispositivo es identificado mediante una dirección individual y una o varias direcciones de grupo. Ejemplos de direcciones son: 1/3/6, 0/5/224 y 5/78.

La **dirección individual** es única en el sistema e identifica al dispositivo de manera unívoca, además de indicar su posición en la red. Está formada por 16 bits divididos en 4 bits de Área, 4 de Línea y 8 de Dispositivo. Las direcciones que empiezan por cero se reservan para los dispositivos acopladores.

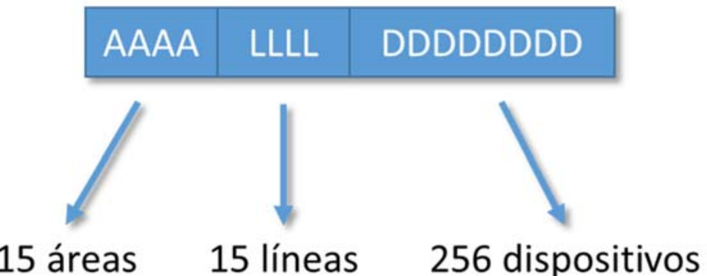


Figura 3: Estructura de una dirección individual en KNX

Las **direcciones de grupo** permiten asociar dispositivos, ya que todos los dispositivos que tengan la misma dirección de grupo reciben los mismos mensajes. Los sensores únicamente podrán enviar telegramas a una dirección de grupo, mientras que los actuadores podrán tener varias direcciones de grupo, lo que les permitirá reaccionar a distintos sensores. Asimismo, varios actuadores podrán tener la misma dirección de grupo y responder por tanto a un mismo mensaje o telegrama. Las direcciones de grupo están formadas por 15 bits y pueden ser de dos o tres niveles, siendo estas últimas las más empleadas:

- Las direcciones de dos niveles se dividen en 4 bits para la dirección de grupo principal y 11 bits para la dirección de grupo secundario.
- Las direcciones de tres niveles se dividen en 4 bits para el grupo principal, 3 bits para el grupo intermedio y 8 bits para el grupo Secundario.

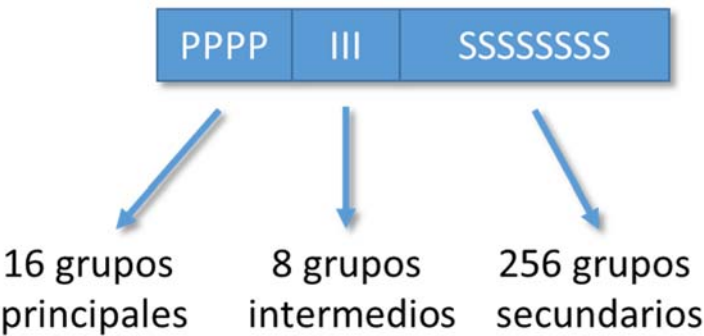


Figura 4: Estructura de una dirección de grupo de tres niveles

Las direcciones de grupo se utilizan para describir y clasificar las diferentes funciones con las que se quiere dotar la instalación. Existen dos tendencias para asociar direcciones de grupo a funciones:

- La primera asocia a los grupos principales las áreas de aplicación, a los intermedios información de ubicación y a los secundarios la zona concreta. Por ejemplo:

Grupo principal	Grupo intermedio	Grupo secundario	Dirección de grupo
1 – Iluminación	2 - Aparcamiento	1 - Hueco del ascensor	1/2/1
		2 - Nivel 1	1/2/2
		.	.
		.	.
2 - Persianas	3 - Oficinas	3 – Oficina z-21	2/3/1
		.	.
		.	.

Tabla 1: Ejemplo de direcciones de grupo basadas en la aplicación

- La segunda tendencia asocia a los grupos principales la información de ubicación, a los intermedios el área de aplicación y a los secundarios la zona concreta. Por ejemplo:

Grupo principal	Grupo intermedio	Grupo secundario	Dirección de grupo
1 – Aparcamiento	2 - Iluminación	1 - Hueco del ascensor	1/2/1
		2 - Nivel 1	1/2/2
		.	.
		.	.
2 - Oficinas	3 - Persianas	3 – Oficina z-21	2/3/1
		.	.
		.	.

Tabla 2: Ejemplo de direcciones de grupo basadas en la ubicación

2.2.4 Datapoints

Para que dos dispositivos puedan comunicarse entre sí, no solo se precisa conocer como localizarse; los datos intercambiados tienen que tener el mismo significado para ambos dispositivos. KNX soluciona este problema con la definición en su estándar de los *datapoint*, los cuales fijan el formato de los datos.

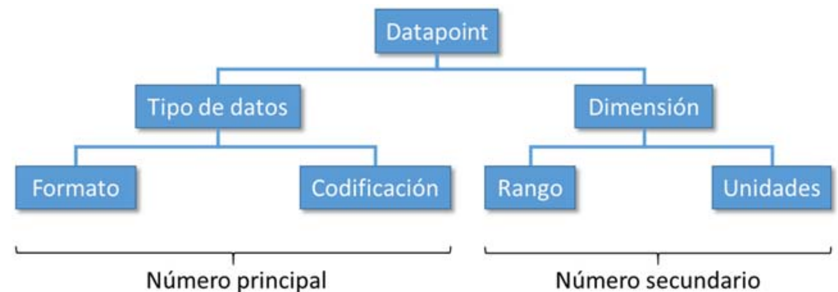


Figura 5: Estructura de un datapoint en KNX

Los tipos de ***datapoint*** vienen definidos como una combinación de formato, codificación, rango y unidades. Se identifican mediante un número principal y un número secundario separados mediante un punto. La siguiente tabla muestra algunos ejemplos de ***datapoint*** definidos en el estándar [1]:

Identificador	**Nombre**	**Codificación**
1.001	*DPT_Switch*	Booleano 0 = Off 1 = On
1.008	*DPT_UpDown*	Booleano 0 = Arriba 1 = Abajo
3.007	*DPT_Control_Dimming*	Booleano + binario 0 = Decrementar brillo 1 = Incrementar brillo 001b...111b: Paso
5.003	*DPT_Angle*	Binario Rango: [0...360] Unidades: °
9.001	*DPT_Value_Temp*	Float Rango: -273 °C ... 670 760 °C Unidades: °C

Tabla 3: Ejemplos de datapoint

2.2.5 Objetos de comunicación

Cada dispositivo, ya sea un sensor o un actuador, puede disponer de uno o más objetos de comunicación. Los objetos de comunicación son direcciones de memoria en los dispositivos que contienen información relevante sobre su estado. Por ejemplo, si una salida está activada o desactivada, la hora y la fecha de un reloj o si una entrada está a 0 o está a 1. El tamaño de estas direcciones de memoria puede variar entre 1 bit y 14 bytes dependiendo de la función que desempeñen. Por ejemplo, una conmutación solo precisará dos estados (0 y 1) por lo que requerirá de objetos de comunicación de 1 bit.

Cada objeto de comunicación puede tener una dirección de grupo asociada. Si se trata de un objeto de comunicación emisor esta dirección será única, mientras que un objeto de comunicación receptor puede tener varias direcciones de grupo. Los objetos de comunicación emisores y receptores se ligan entre sí asociándoles una misma dirección de grupo. Las direcciones de grupo solo pueden tener asociados objetos de comunicación del mismo tamaño.

Todo cambio en un objeto de comunicación emisor conlleva la oportuna difusión del nuevo valor por el bus, de forma que todos los objetos de comunicación receptores, ligados por la misma dirección de grupo, reciben el nuevo valor y actúan en consecuencia.

2.2.6 Estructura de un telegrama

Cada vez que se genera un evento en el sistema, como por ejemplo la activación de un pulsador, el sensor correspondiente se encarga de trasmitir un mensaje o telegrama. Una vez obtiene acceso al bus y finaliza la transmisión sin colisiones, espera a la recepción de un reconocimiento. En el caso de no recibir reconocimiento, se reintentará la transmisión hasta tres veces.

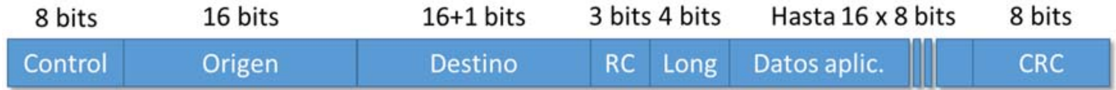


Figura 6: Estructura de un telegrama KNX

Los telegramas contienen información específica acerca del evento acontecido. Constan de siete campos de los cuales seis corresponden al protocolo de red y uno solo contiene datos de aplicación. A continuación se describe la función de cada uno de estos campos:

- **Control**: Indica el tipo de trama e información adicional, como la prioridad o si se trata de una trama repetida.
- **Origen**: Contienen la dirección física del dispositivo que emite el telegrama.
- **Destino:** Contiene la dirección del destinatario del telegrama. La dirección de destino puede ser de dos tipos en función del valor que tome el bit de mayor peso de este campo:
    - Si su valor es ‘0’, se trata de una dirección física y el telegrama va dirigido únicamente a un dispositivo.
    - Si su valor es ‘1’, se trata de una dirección de grupo y el telegrama se dirige a todos los dispositivos que tengan esa dirección de grupo.
- **RC:** Es un contador utilizado para funciones de enrutamiento. Cuenta el número de saltos que ha dado el paquete.
- **Longitud:** Este campo indica cuántos bytes contiene el campo de datos útiles.

- **Datos aplicación**. Contiene el tipo orden a ejecutar o los datos generados por un sensor.
- **CRC**: Permite al dispositivo receptor comprobar si el telegrama es correcto. Si la recepción es correcta se envía un reconocimiento, en caso contrario se manda un reconocimiento negativo para que el emisor repita el envío.

2.2.7 Conectividad con el bus KNX desde el exterior

La manera de establecer una conexión al bus desde el exterior ha ido evolucionando. Al principio se empleaban transceptores TP-UART (*Twisted Pair – Universal Asynchronous Receiver/Transmitter*) que convertían las señales utilizadas en el par trenzado en bytes en un puerto serie RS232.

Posteriormente, aparecieron los dispositivos BCU (*Bus Coupling Unit*) que también permitían acceder al bus por puerto serie. Estos dispositivos incorporaban un microprocesador que implementaba el protocolo de comunicación del bus en el lado KNX, y un protocolo externo (*EMI, External Message Interface*) en el lado RS232.

Por último, han aparecido dispositivos que permiten conectarse al bus desde un puerto USB o desde una red IP. Debido a la naturaleza WEB del dispositivo objetivo del presente trabajo, se empleará como puerta de entrada al bus KNX uno de estos dispositivos pasarela KNX/IP, en concreto el modelo 730 de la marca *WEINZIERL*.

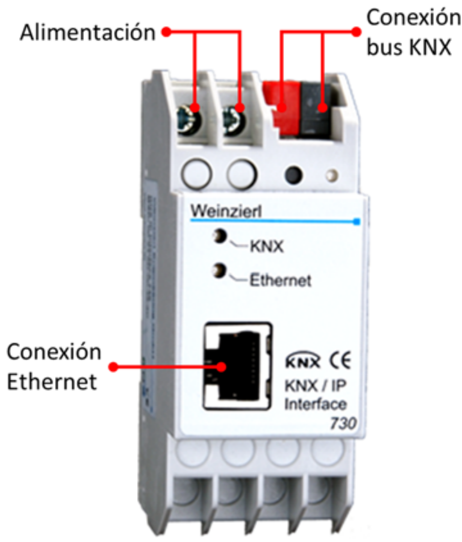


Figura 7: Dispositivo pasarela entre red IP y KNX WEINZIERL mod. 730

2.2.7.1 *La librería Calimero*

Además de un dispositivo *hardware* que proporcione el acceso al bus KNX, es preciso implementar las primitivas necesarias para leer e interpretar la información que circula por el bus. Asimismo, la información que se envíe por el bus debe ser entendible por los dispositivos conectados al mismo.

Calimero es una librería Java creada para poder acceder a sistemas KNX que proporciona una API [2] para la mayoría de los servicios de red y codificaciones de datos definidos en el estándar. La plataforma mínima de Java que requiere es la *Java 2 Micro Edition*, lo cual permite su uso en sistemas con recursos limitados, tales como los sistemas empotrados.

La versión actual de *Calimero* (2.1) se distribuye bajo licencia GPL (*General Public License*) con la excepción *Classpath*; los cambios o extensiones realizadas sobre la librería deben ser gratuitas y de código abierto, mientras que las aplicaciones desarrolladas con la librería pueden tener la licencia que el desarrollador considere. La arquitectura de *Calimero* se divide funcionalmente en tres capas:

- Una capa inferior que proporciona la implementación de los servicios básicos relacionados con los protocolos de red.
- Una capa intermedia que proporciona una interfaz estándar y homogénea para la comunicación con redes KNX.
- Una capa superior que proporciona servicios de alto nivel.

El usuario no está obligado a trabajar con los servicios de alto nivel, sino que puede elegir el nivel de abstracción con el que desea trabajar.

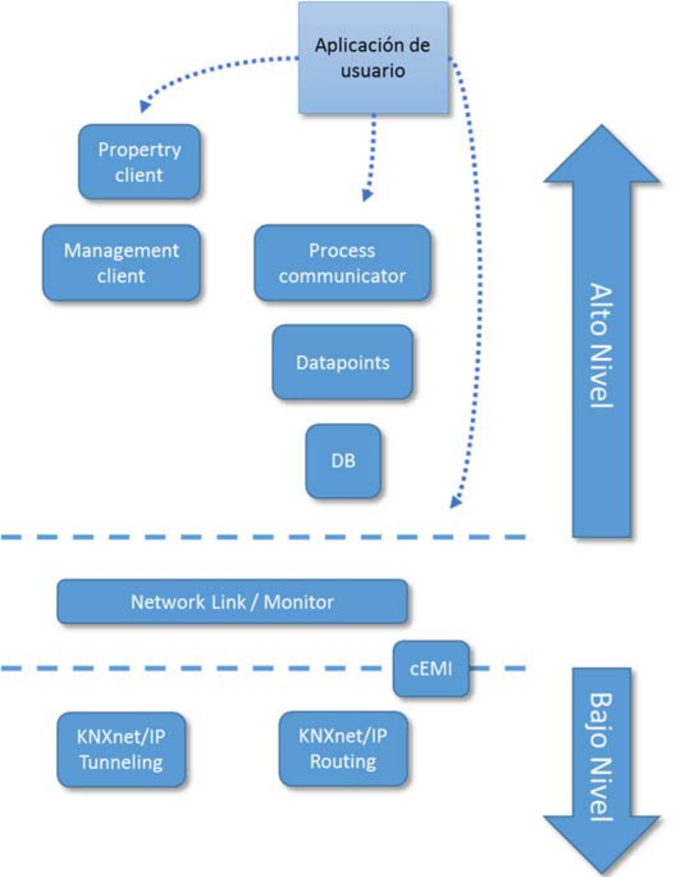


Figura 8: Arquitectura de la librería Calimero

La conexión con la red KNX desde una red IP se realiza mediante un objeto de enlace `KNXNetworkLinkIP`. En el momento de su creación, se proporciona al constructor de dicho objeto todos los elementos necesarios para el acceso a la red KNX, como pueden ser el protocolo de comunicación con el interfaz de red o su dirección IP. La comunicación que se ofrece emplea únicamente UDP. Los servicios de alto nivel hacen uso de este objeto, pudiéndose usar el mismo objeto de enlace para varios servicios de alto nivel.

```
KNXNetworkLinkIP netLink = new KNXNetworkLinkIP(… ;
ManagementClientImpl mngClt = new ManagementClientImpl(netLink);
ProcessCommunicator pc = new ProcessCommunicator (netLink);
```

Entre los servicios de alto nivel proporcionados, cabe destacar el `ProcessComunicator`. Este servicio, ofrece una API bloqueante con métodos que permiten operaciones de lectura y escritura, sobre direcciones de grupo, para distintos tipos de datos.

```
Boolean b = pc.readBool (new GroupAddres("1/2/1"));
pc.write(new GroupAddress("3/2/2", 3.14159f);
```

Para aquellas aplicaciones que deban reaccionar ante cambios en el bus, *Calimero* proporciona el interfaz `NetworkLinkListener`. La clase que implemente este interfaz en la aplicación, deberá sobrescribir el método `indication` del interfaz, el cual se ejecutara cada vez que se detecte un evento. Este método cuenta con un parámetro `FrameEvent` que permite extraer la información del telegrama asociado al evento.

```
byte[] payLoad = ((CEMILData) fe.getFrame()).getPayload();
String GAdd = ((CEMILData) fe.getFrame()).getDestination().toString();
```

Una vez instanciado el objeto `NetworkLinkListener`, deberá asociarse a un enlace previamente creado mediante el método `addLinkListener`:

```
netLink.addLinkListener((NetworkLinkListener) networkListener);
```

Una interesante funcionalidad que ofrece *Calimero* son unos traductores llamados *DPTXlator*. Estos traductores cierran la brecha que existe entre los tipos de datos de KNX, representados mediante los *datapoint,* y los tipos de datos de Java. Todos los *DPTXlator* aceptan y devuelven representaciones de su valor en formato de cadena. El usuario puede trabajar con representaciones de valores tales como "250.4 lx" y el *DPTXlator* se encarga de ofrecer a KNX su representación en forma de un *array* de bytes y viceversa. Para obtener el traductor adecuado, basta con proporcionar el identificador del *datapoint* en el momento de instanciarlo:

```
DPTXlator xlator = new DPTXlatorBoolean("1.001");
```

2.3 Introducción a las redes de reglas

Las redes de reglas son una metodología para el diseño, implementación y verificación de sistemas automáticos de control mediante reglas. Su nombre deriva de la combinación de las redes de Petri y de los sistemas expertos basados en reglas.

2.3.1 Variables y hechos

Para la aplicación de esta metodología, se deben determinar las variables observables de sistema que se quiere controlar, las cuales constituirán las entradas de la red de reglas. También es necesario identificar las variables controladas, que serán las salidas del sistema de control. En cada una de estas variables de entrada o de salida, se deben identificar los posibles estados significativos, indicando para cada uno de ellos las condiciones en los que resulte cierto. Cada uno de estos estados significativos se va a denominar **hecho**.

Para las variables digitales se asignará un hecho para cuando valgan "0" y otro hecho para cuando valgan "1". En el caso de las variables analógicas, deberán ser previamente discretizadas, asignándose un hecho a cada rango de valores de la magnitud.

Una variable solo podrá encontrarse en un estado, por lo tanto, solo uno de sus hechos será cierto en un momento dado; en este caso se dice que el hecho está **marcado**. Al resto de hechos de la variable se les denomina **complementarios** del primero y deberán estar **desmarcados**. El marcado de la red de reglas se expresa mediante un **vector de marcado** que muestra, de manera inequívoca, cuál de los hechos de cada variable es cierto en un determinado momento. Por tanto, el vector de marcado representa el **estado** de la red de reglas.

2.3.2 Reglas

El comportamiento del sistema se expresa creando relaciones entre los hechos de las distintas variables de entrada y salida. En las redes de reglas estas relaciones se establecen mediante reglas de producción de la forma:

SI <Lista de hechos antecedentes> ENTONCES <Lista de hechos consecuentes>

Las redes de reglas pueden representarse de varias formas: gráfica, textual y matricial. A continuación se describen estas formas de representación indicando sus ventajas e inconvenientes.

2.3.3 Representación gráfica

Análogamente a las redes de Petri, una red de reglas se puede representar mediante un grafo dirigido con nodos de dos tipos: los hechos y las reglas. Los hechos se representan mediante círculos de forma similar a los lugares de las redes de Petri, mientras que las reglas se representan como segmentos. A cada regla llegaran arcos dirigidos desde cada uno de sus hechos antecedentes y de ella partirá un arco hasta cada uno de sus hechos

consecuentes. Para indicar que un hecho está marcado se colocará un punto en el nodo correspondiente.

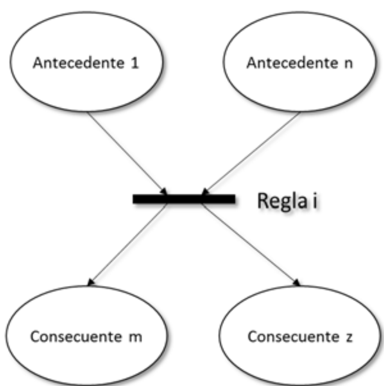


Figura 9: Representación gráfica de una red de reglas

Este modo de representación ofrece una visión bastante intuitiva del sistema; sin embargo, resulta prácticamente inviable su uso cuando el número de reglas o hechos es elevado.

2.3.4 Representación textual

Es la forma más sencilla de representar una red de reglas, consiste en expresar cada una de las reglas mediante secuencias similares a una oración condicional. Su formato es el siguiente:

$$SI\ variable_i\ ESTÁ\ hecho_x\ [Y\ variable_j\ ESTÁ\ hecho_y\ \dots]$$
$$ENTONCES\ variable_m\ DEBE\ ESTAR\ hecho_u\ [Y\ variable_n\ DEBE\ ESTAR\ hecho_v\ \dots]$$

Es tal vez el formato más adecuado para que un usuario especifique la red de reglas, pero resulta poco útil desde el punto de vista computacional.

2.3.5 Representación matricial

Se puede expresar una red de reglas mediante vectores y matrices de elementos binarios. Siendo **h** el número de hechos total y **r** el número de reglas de una red de reglas, se definen las siguientes matrices:

- **Matriz de antecedentes A** de orden r x h donde el elemento $\mathbf{a_{ij}}$ es 1 si el hecho j es antecedente de la regla i, de lo contrario vale 0.
- **Matriz de consecuentes C** de orden r x h donde el elemento $\mathbf{c_{ij}}$ es 1 si el hecho j es consecuente de la regla i, de lo contrario vale 0.
- **Matriz de desmarcado D** de orden r x h donde el elemento $\mathbf{d_{ij}}$ es 0 si el elemento $\mathbf{c_{ik}}$ es 1 y el hecho j es complementario del hecho k, de lo contrario vale 1.
- **Vector de estado S** de orden h donde el elemento $\mathbf{s_i}$ vale 1 si el hecho i es cierto, de lo contrario vale 0. Por tanto, el vector de estado indica en un

momento dado, qué hechos de la red de reglas son ciertos y cuáles no. Se definirá un estado inicial S_0 que representará el estado de reposo en el que arrancará el sistema y del cual dependerán las características y funcionamiento del mismo. Para que un estado sea coherente solo uno de los hechos de cada variable debe ser distinto de cero. El estado inicial S_0 debe ser obligatoriamente coherente.

La representación de una red de reglas mediante matrices binarias resulta muy adecuada para su utilización en sistemas informáticos, ya que todas las operaciones relativas a la evolución de la red pueden implementarse mediante sencillas operaciones lógicas y saltos condicionales.

A continuación se indica, a partir de las matrices binarias, cómo determinar cuándo una regla debe ser aplicada y como aplicarla. Una regla que debe ser aplicada se dice que está **sensibilizada** y la acción de aplicar dicha regla recibe el nombre de **disparo**.

2.3.6 Sensibilización de una regla

Se dice que una regla ***i*** esta sensibilizada en un estado ***S*** si se cumplen las siguientes condiciones:

- Todos sus antecedentes son ciertos
- Al menos uno de sus consecuentes es falso

Se puede demostrar que las condiciones anteriores son equivalentes a que sean ciertas las siguientes igualdades:

$$\bar{S}\ AND\ Ai = 0$$
$$\bar{S}\ AND\ Ci\ \neq 0$$

Siendo ***Ai*** y ***Ci*** las filas i-ésimas de las matrices de antecedentes y consecuentes respectivamente. $\bar{S}$ es el complementario del vector de estado.

2.3.7 Disparo de una regla sensibilizada

El disparo de una regla ***i*** sensibilizada supone la modificación del vector de estado mediante las siguientes acciones:

- Marcar o hacer ciertos todos sus consecuentes.
- Desmarcar o negar todos los complementarios de sus consecuentes.

Se puede demostrar que estas acciones equivalen a la siguiente operación lógica:

$$S_{n+1} = (S_n\ OR\ Ci)\ AND\ Di$$

Siendo ***Ci*** y ***Di*** las filas i-ésimas de las matrices de consecuentes y desmarcado respectivamente.

2.3.8 Verificación de la red de reglas

Para emplear una red de reglas en el control de un sistema, resulta imprescindible garantizar a priori que su comportamiento sea correcto. Se puede realizar una simulación del comportamiento ante cambios en las entradas; si la verificación es exhaustiva, se podrá garantizar el comportamiento de la red en el sistema real. Sin embargo, este proceso de verificación puede simplificarse si se garantiza que el comportamiento de la red de reglas cumple unas determinadas pautas.

El análisis de una red de reglas para estudiar sus propiedades puede realizarse mediante la expansión del **árbol de alcanzabilidad**. Se define el árbol de alcanzabilidad de una red de reglas como un grafo en el que se cumplen las siguientes condiciones:

- Cada nodo representa un vector de estado. Se define como un estado **estable o conclusión,** aquel en el cual no existe ninguna regla sensibilizada. Por otro lado, un estado **transitorio** es aquel en el que existe al menos una regla sensibilizada. El nodo raíz será el estado inicial.
- De cada nodo transitorio salen tantos arcos como reglas **sensibilizadas** haya en el mismo. Cada uno de los arcos se etiqueta con la regla sensibilizada correspondiente, y conduce al nodo del estado obtenido al disparar dicha regla. Si dicho estado es transitorio, tendrá reglas sensibilizadas a su vez, expandiéndose el árbol hasta alcanzar un estado estable o conclusión. A la secuencia de reglas disparadas se le denomina **deducción.** Una RdR se dice que es **única** cuando toda deducción aplicable desde un estado dado lleva a la misma conclusión.
- De cada nodo estable salen tantos arcos como hechos de entrada ante los que el estado sea receptivo, es decir, modifican el marcado de forma que aparece una regla sensibilizada. Cada uno de los arcos se etiqueta con el hecho al cual el estado es receptivo, y conduce al correspondiente nodo transitorio.

Para llevar a cabo el estudio de este árbol, es necesario implementar algoritmos que, recorriendo dicho árbol, sean capaces de detectar ciclos, nodos conclusión que provienen de un mismo padre, cambios repetidos en el valor de una variable e incluso estados de los cuales es imposible salir.

En el caso que nos ocupa no es posible definir un estado inicial, ya que estamos creando una capa de funcionamiento por encima de la ya existente, por lo que sería necesario la comprobación de la red de reglas para todos los estados iniciales posibles. Además, podría darse el caso de tener una red de reglas correcta que, en combinación con la configuración definida en la instalación KNX, diera como resultado un comportamiento anómalo.

De todas las posibles anomalías que podría presentar la red de reglas, en el caso que nos ocupa, sería conveniente evitar la existencia de ciclos que provoquen el cambio repetitivo en un hecho de una variable. Por tanto, habría que detectar esta circunstancia en tiempo de ejecución de la red de reglas y neutralizarla. En cuanto a la formulación individual de cada una de las reglas, se comprobará que estas sean **puras**.

Se dice que una regla es **pura** si se verifica que esta no es **incompleta**, **imposible**, **autocontradictoria** ni **realimentada**. A continuación se definen estos cuatro términos:

- Una regla es incompleta si carece de antecedentes o de consecuentes.
- Una regla es imposible si tiene como antecedente un hecho y alguno de sus complementarios.
- Una regla es autocontradictoria si tiene como consecuente un hecho y alguno de sus complementarios.
- Una regla es realimentada si contienen un hecho que sea simultáneamente antecedente y consecuente de dicha regla.

3 Análisis

3.1 Análisis de requerimientos

En la siguiente figura se muestra la estructura general del dispositivo que se pretende implementar:

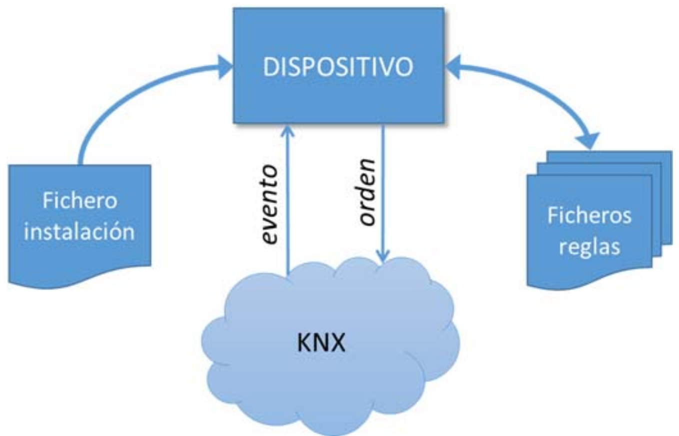


Figura 10: Estructura general del dispositivo a implementar

La parte *software* del dispositivo va a estar constituida por una aplicación WEB que realizará las siguientes tareas:

- Leer un fichero representando una instalación KNX.
- Interpretar dicha instalación y convertirla en variables con sus respectivos hechos.
- Ofrecer al usuario una interfaz amigable que le permita asociar los hechos de las variables mediante reglas.
- Ser capaz de detectar eventos en la instalación KNX y, en función de las reglas definidas por el usuario, generar las órdenes necesarias para producir cambios sobre dicha la instalación.
- Guardar en un fichero la red de reglas, formada por las variables y las reglas, de manera que pueda ser recuperada posteriormente.

Desde el punto de vista hardware, se desea emplear un dispositivo de bajo coste y bajo consumo, de forma que, por un lado, pueda quedar permanentemente encendido - al menos mientras la instalación domótica se encuentre funcionando - mientras que por otro lado, sea lo suficientemente potente para ejecutar las tareas anteriormente descritas. El dispositivo que aúna estas características es un computador monoplaca

3.2 Esquema de bloques funcionales

El dispositivo contará con un servidor WEB a través del cual se ofrezca una aplicación WEB compuesta por los siguientes módulos:

- Intérprete de la instalación KNX: Se encargará de leer un fichero conteniendo la descripción de la instalación KNX y convertir sus elementos en variables con sus respectivos hechos. Se está desarrollando en un TFG paralelo un módulo Identificador instalaciones KNX que permita acceder a la instalación e identificar las direcciones de grupo asociadas a los distintos dispositivos de la misma, incluyendo dicha información en el fichero de instalación.
- Editor de reglas: Permitirá al usuario definir de manera sencilla sus propias reglas de comportamiento para la instalación KNX.
- Ejecutor de redes de reglas: Establecerá una conexión con la instalación KNX a través de la pasarela KNX/IP. A través de dicha conexión, monitorizará la instalación examinando el contenido de los mensajes que circulan por el bus KNX y, en función de las reglas definidas por el usuario, generará las órdenes necesarias para modificar el estado de los dispositivos KNX.
- Gestor de redes de reglas: Permitirá guardar en ficheros las reglas definidas por el usuario. El usuario podrá posteriormente cargar, editar y ejecutar las reglas contenidas en dichos ficheros.

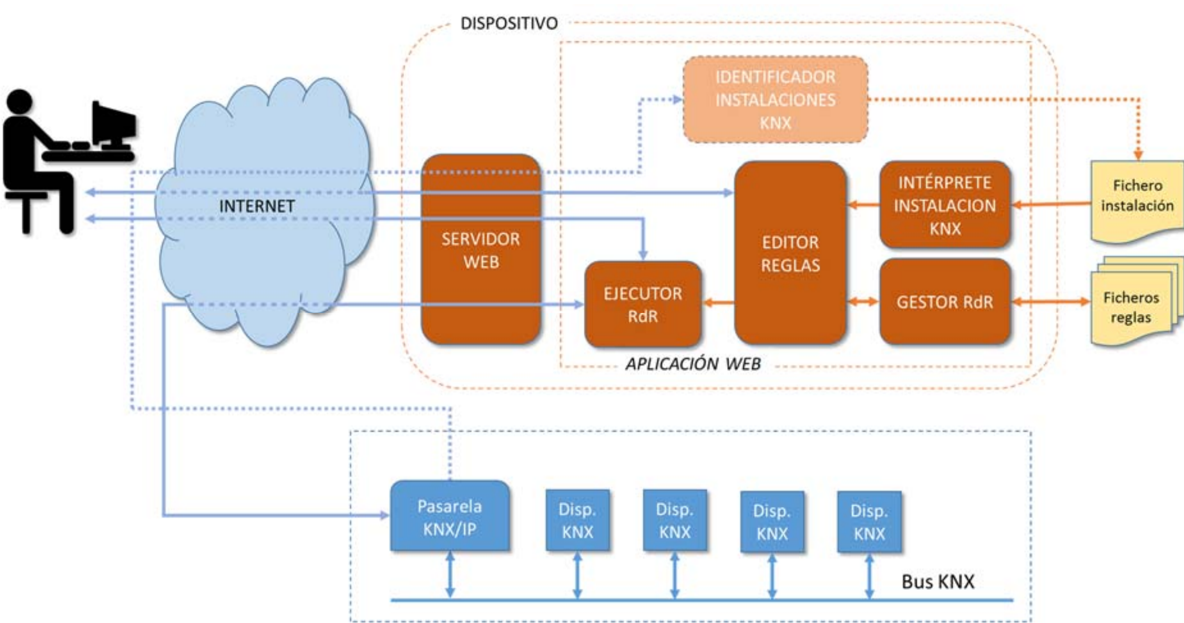


Figura 11: Bloques funcionales del dispositivo a implementar

3.3 Selección de *software* y *hardware*

A partir del análisis de requerimientos, se detalla a continuación la elección de tecnologías, en cuanto a la parte *software* y *hardware*, para la implementación del dispositivo.

3.3.1 Formato de datos

Para que el dispositivo que se pretende implementar pueda funcionar con cualquier instalación KNX, deberá ser capaz de aceptar una descripción de la instalación sobre la que va a operar. La manera más inmediata es a través de un fichero en el que se detallen los dispositivos que componen la instalación. Evidentemente, este fichero ha de estar formateado, debiéndose establecer la sintaxis para la representación de cada dispositivo.

A la hora de buscar una especificación para representar información estructurada, de modo que pueda ser almacenada, transmitida, procesada, visualizada e impresa por muy diversos tipos de aplicaciones y dispositivos, la respuesta es XML. XML (*eXtensible Markup Language*) es un lenguaje de marcas desarrollado por el *World Wide Web Consortium* (W3C) que deriva del lenguaje SGML. Su función principal es describir datos y no mostrarlos, como es el caso de HTML. A continuación se realizará una breve introducción al formato de los documentos XML.

Un documento XML está formado por datos de caracteres y de marcado, que se almacena como texto en un archivo con extensión *.xml*. Las normas a seguir para la construcción de documentos XML son dictadas por el organismo W3C [3]. Entre ellas destacan:

- El nombre de las etiquetas es sensible a las mayúsculas.
- En todo documento debe haber un elemento, llamado raíz de documento, que contenga a los demás.
- Todo elemento tiene que tener su correspondiente etiqueta de inicio y de cierre o una sola etiqueta vacía.
- Todos los elementos deben estar correctamente anidados.
- Todos los valores de los atributos deben ir entre comillas.

Aunque no es obligatorio, los documentos XML pueden empezar con unas líneas llamadas prólogo que describen la versión XML, el tipo de documento, etc.

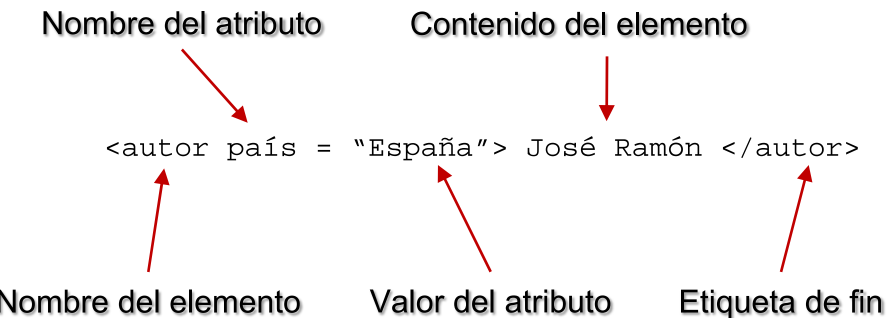
```
<?xml version="1.0" encoding="UTF-8"?>
```

El prólogo de un documento XML contiene:

- Una declaración XML que califica al documento como un documento XML.
- Una declaración de tipo de documento

- Uno o más comentarios e instrucciones de procesamiento.

A diferencia del prólogo, el cuerpo no es opcional en un documento XML. El cuerpo debe contener un único elemento raíz. Los elementos XML pueden tener contenido o bien ser elementos vacíos. Además, los elementos pueden tener atributos para incorporar características o propiedades a los elementos del documento.



El dispositivo también permitirá guardar en ficheros las reglas definidas por el usuario. De este modo, el usuario podría disponer de bibliotecas de reglas. Para mantener la coherencia con los ficheros de entrada, también se utilizará XML para dar formato a estos ficheros de reglas.

3.3.2 Aplicación WEB

En los últimos años, se han popularizado las denominadas aplicaciones RIA (*Rich Internet Applications*). A diferencia de las aplicaciones WEB tradicionales, que se basan en la navegación a través de distintas páginas, este tipo de aplicaciones está constituida por una única página cuyo contenido se va actualizando, tratando así de imitar a las aplicaciones de escritorio. El *framework* oficial de Java EE para la implementación de este tipo de aplicaciones es JSF (*Java Server Faces*).

En JSF, la definición de la interfaz de usuario se realiza mediante páginas XHTML con distintos tipos de etiquetas que representan componentes tales como menús desplegables, paneles, etc. Estas páginas se denominan páginas JSF. Los componentes de la interfaz deben mostrar y recoger datos que se obtienen y se pasan a la aplicación; es lo que se denomina ligado de datos o *data binding*. Para realizar esta tarea, en JSF se utiliza la anotación `@ManagedBean` dentro de una clase Java para definir los llamados *managed bean*.

JSF se ejecuta sobre la tecnología de *Servlets*, por tanto, para la ejecución de aplicaciones JSF únicamente se precisa de un contenedor de *servlets* como pueden ser *Jetty* o *Tomcat*. Cuando el usuario realiza una petición a una página JSF, se realizan las siguientes acciones:

1. En primer lugar, se procesa la página de arriba abajo y se crea un árbol de componentes JSF en forma de objetos Java. Estos objetos son instanciados a partir de clases del *framework* JSF. El árbol se guarda en memoria para que posteriores peticiones a la misma página no tengan que volver a crearlo.

2. A continuación, se obtienen los valores introducidos por el usuario y se actualizan con ellos los atributos de los *managed bean*.

3. Seguidamente, se actualizan los componentes del árbol a partir de los valores procedentes de los atributos de los *managed bean*.

4. Finalmente, a partir de los componentes del árbol, se genera el código HTML que se envía de vuelta al navegador como resultado de la petición. Muchas veces no va a ser necesario generar la página completa, sino que bastará con actualizar únicamente alguno de los componentes.

Existen distintas implementaciones de la especificación JSF, siendo *Mojarra* la de referencia proporcionada por *Sun/Oracle*. Otra implementación muy utilizada es *MyFaces*, creada por la *Apache Software Foundation*. Además, hay disponibles distintas librerías de componentes, entre las más empleadas se encuentran *RichFaces*, *IceFaces* y *Primefaces;* todas con licencia de código abierto. *PrimeFaces* destaca por ser una librería muy liviana, lo que la hace especialmente interesante de cara a utilizarla en el presente trabajo.

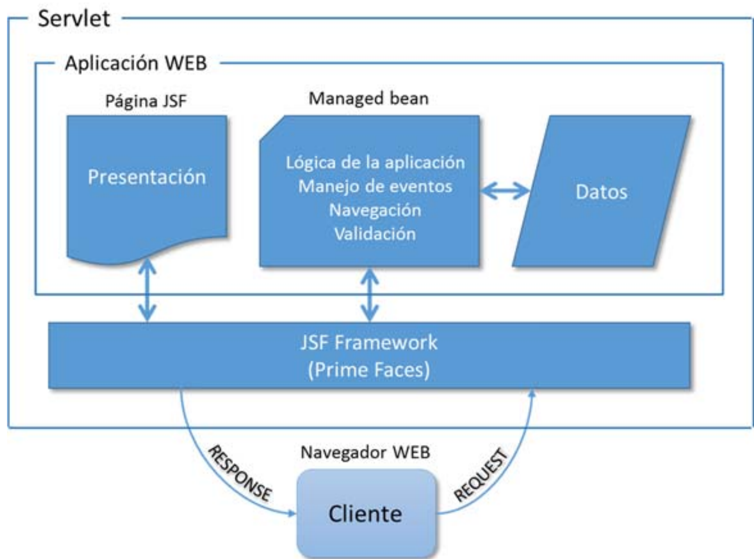


Figura 12: Arquitectura de la aplicación WEB utilizando JSF

3.3.3 Ordenadores monoplaca

Como su nombre indica, los ordenadores monoplaca contienen en una única placa de circuito impreso todos los elementos de un ordenador, es decir, procesador, memoria, almacenamiento, periféricos y demás circuitería.

La aparición de procesadores destinados al mercado de los dispositivos móviles ha propiciado su aplicación en este tipo de ordenadores, permitiendo su reducción en cuanto a coste, consumo y tamaño. En la actualidad es posible disponer de dispositivos del tamaño de una tarjeta de crédito que incluso incorporan un sistema operativo precargado.

En el mercado existen diversas opciones entre las que se ha optado por el último modelo de la serie *Beagle*: la *BeagleBone Black* en su revisión C. Este modelo tiene preinstalada la distribución Linux *Debian* en su versión 7 *«wheezy»* para arquitectura ARM, lo cual simplifica tanto su puesta en marcha como la posterior instalación de paquetes como si de un ordenador convencional se tratara.

La *BeagleBone Black* ha sido concebida bajo la filosofía *Open Source*, estando a disposición de los usuarios toda la información referida al diseño, incluyendo esquemas y ficheros de los PCB. Por tanto, cualquiera puede construir la placa o incorporar las modificaciones que considere oportunas. Además, cuenta con el soporte de una amplia comunidad de usuarios. Las principales características de este dispositivo son las siguientes:

- Procesador: *Texas Instruments Sitara* AM3358BZCZ100 ARM *Cortex* A8 a 1GHz.
- Motor gráfico: SGX530 3D.
- Memoria SDRAM: 512MB DDR3L 800MHZ.
- Memoria Flash: 4GB , 8bit eMMC.

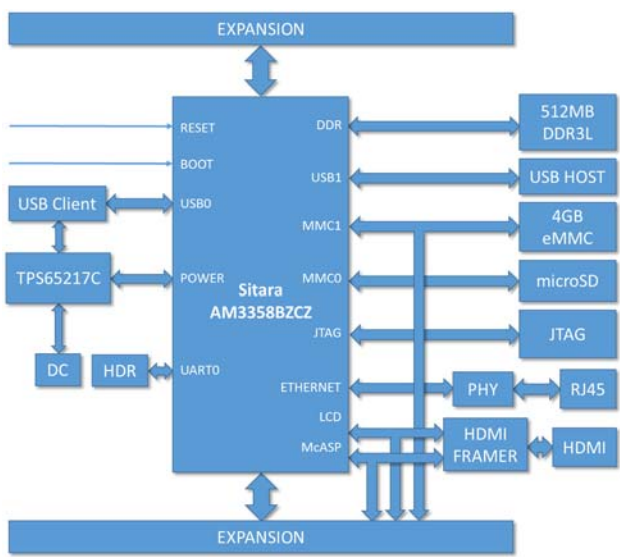


Figura 13: Diagrama de bloques de la BeagleBone Black

La placa dispone de los siguientes conectores:

- Dos puertos de expansión de 46 pines configurables en distintos modos que incorporan los siguientes elementos:
    - Tres buses I2C.
    - Un bus CAN.
    - Un bus SPI.
    - Cuatro temporizadores.
    - Cinco UART.
    - 65 entradas/salidas binarias.
    - Ocho salidas PWM.
    - Siete entradas analógicas de 1.8 voltios con conversores A/D de 12 bits.
- Puerto USB 2.0 tipo miniUSB que permite la comunicación con un PC así como la alimentación del sistema.
- Puerto USB 2.0 *host* tipo A 500mA LS/FS/HS.
- Puerto Ethernet 10/100, RJ45.
- Lector de tarjetas microSD.
- Conector de alimentación 5V 350 mA.
- Salida de video microHDMI de 16 bits con resoluciones 1280x1024, 1024x768, 1280x720, 1440x900 y 1920x1080 a 24Hz.
- Audio estéreo a través del puerto HDMI.

Cuenta con tres pulsadores cuya función se describe a continuación:

- **Reset:** Al pulsarlo se provoca un reinicio del sistema.
- **Boot:** Permite modificar el dispositivo del que arranca el sistema. Por defecto se arranca desde la memoria flash, pero puede cambiarse a la tarjeta microSD o al puerto USB.
- **Power:** Al pulsarlo se realiza un apagado ordenado del sistema. Si se mantiene pulsado por más de ocho segundos, se produce el apagado instantáneo. Si el indicador LED *power* está apagado, al pulsar este botón se enciende el sistema.

La placa dispone de un total de siete indicadores LED cuya función es la siguiente:

- El LED *power* se enciende al aplicar tensión a la placa, indicando que el circuito de manejo de la alimentación está en funcionamiento. Si este indicador

parpadea, significa que hay un exceso de consumo de corriente y que el circuito de manejo de la alimentación está desconectado.

- Cuatro LED ***user*** cuya función puede ser configurada por el usuario. Su funcionamiento por defecto es el siguiente:
    - USR0 parpadea a una frecuencia fija.
    - USR1 se ilumina al acceder a la tarjeta microSD.
    - USR2 se ilumina cuando hay actividad en la CPU.
    - USR3 se ilumina al acceder a la memoria flash interna.
- Dos LED en el conector RJ45 que dan información de estado, el amarillo indica que el enlace está activo y el verde que existe tráfico de información.

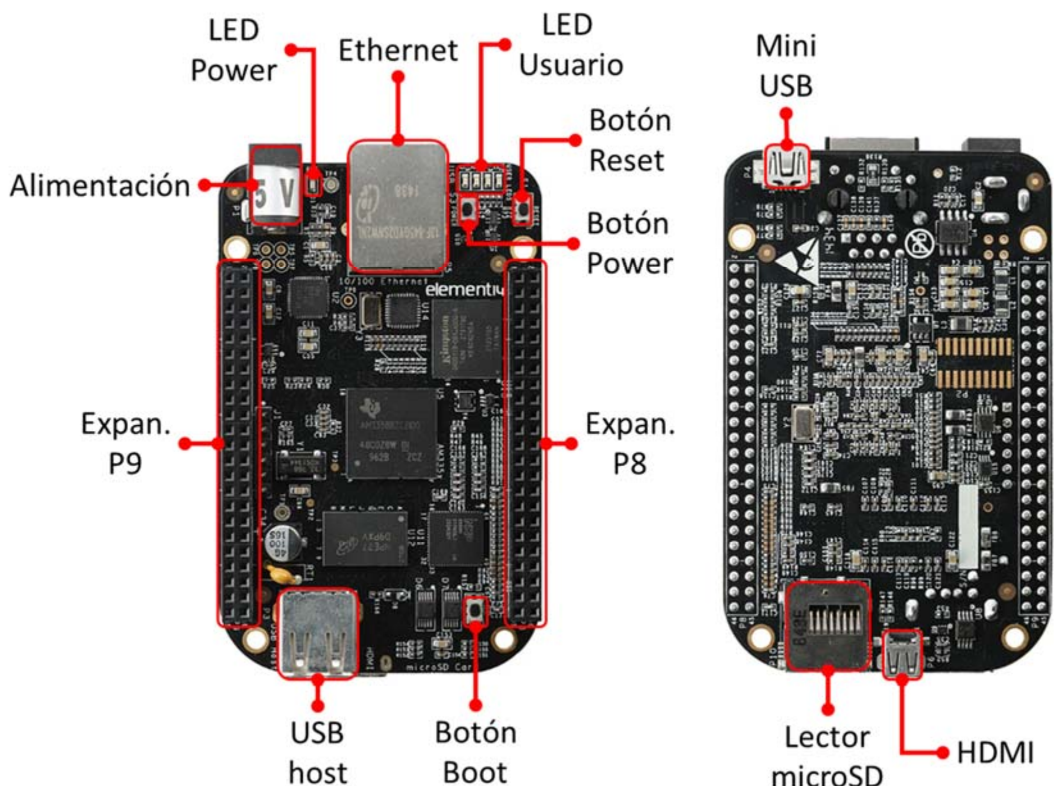


Figura 14: Conectores, pulsadores e indicadores en la BeagleBone Black

4 Diseño

4.1 Introducción

En el presente capítulo se van a abordar los aspectos relacionados con el diseño del dispositivo. En primer lugar, se detallará el formato de los ficheros de entrada y salida. Seguidamente, se describirá el proceso de ejecución de las reglas definidas por el usuario. Finalmente, se abordará el diseño de la aplicación WEB y su interfaz de usuario.

4.2 Formato de datos

Para que el dispositivo que se pretende implementar pueda funcionar con cualquier instalación KNX, deberá ser capaz de aceptar una descripción de la instalación sobre la que va a operar. La manera más inmediata es a través de un fichero en el que se detallen los dispositivos que componen la instalación. Evidentemente, este fichero ha de estar formateado, debiéndose establecer la sintaxis para la representación de cada dispositivo.

A la hora de buscar una especificación para representar información estructurada, de modo que pueda ser almacenada, transmitida, procesada, visualizada e impresa por muy diversos tipos de aplicaciones y dispositivos, la respuesta es XML. XML (*eXtensible Markup Language*) es un lenguaje de marcas desarrollado por el *World Wide Web Consortium* (W3C) que deriva del lenguaje SGML. Su función principal es describir datos y no mostrarlos, como es el caso de HTML.

Un documento XML está formado por datos de caracteres y de marcado, que se almacena como texto en un archivo con extensión *.xml*. Las normas a seguir para la construcción de documentos XML son dictadas por el organismo W3C [3]. Entre ellas destacan:

- El nombre de las etiquetas es sensible a las mayúsculas.
- En todo documento debe haber un elemento, llamado raíz de documento, que contenga a los demás.
- Todo elemento tiene que tener su correspondiente etiqueta de inicio y de cierre o una sola etiqueta vacía.
- Todos los elementos deben estar correctamente anidados.
- Todos los valores de los atributos deben ir entre comillas.

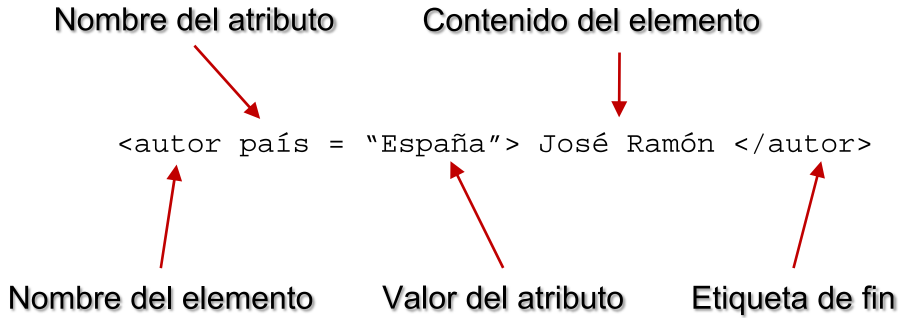
Aunque no es obligatorio, los documentos XML pueden empezar con unas líneas llamadas prólogo que describen la versión XML, el tipo de documento, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
```

El prólogo de un documento XML contiene:

- Una declaración XML que califica al documento como un documento XML.
- Una declaración de tipo de documento
- Uno o más comentarios e instrucciones de procesamiento.

A diferencia del prólogo, el cuerpo no es opcional en un documento XML. El cuerpo debe contener un único elemento raíz. Los elementos XML pueden tener contenido o bien ser elementos vacíos. Además, los elementos pueden tener atributos para incorporar características o propiedades a los elementos del documento.



En cuanto a la manera de describir la instalación, habría dos aproximaciones posibles:

- Basar la descripción en los dispositivos KNX, tales como módulos actuadores, pantallas táctiles, etc.
- Basar la descripción en los elementos que percibe el usuario, como pueden ser las luces, las persianas, los sensores, etc.

Por su mayor legibilidad desde el punto de vista del usuario, se va a optar por la segunda aproximación. Por tanto, hay que definir una biblioteca de posibles elementos domóticos.

El dispositivo también permitirá guardar en ficheros las reglas definidas por el usuario. De este modo, el usuario podría disponer de bibliotecas de reglas. Para mantener la coherencia con los ficheros de entrada, también se utilizará XML para dar formato a estos ficheros de reglas.

4.3 Representación de la instalación KNX en formato XML

En cuanto a la manera de describir la instalación, habría dos aproximaciones posibles:

- Basar la descripción en los dispositivos KNX, tales como módulos actuadores, pantallas táctiles, etc.
- Basar la descripción en los elementos que percibe el usuario, como pueden ser las luces, las persianas, los sensores, etc.

Por su mayor legibilidad desde el punto de vista del usuario, se va a optar por la segunda aproximación. Por tanto, hay que definir una biblioteca de posibles elementos domóticos. La instalación KNX se va a representar en formato XML mediante tres tipos de etiquetas:

- <**topoElement**>: Va a hacer referencia a una ubicación física como puede ser una habitación, una ventana, una puerta, una pared, un bastidor con múltiples focos que se controlan individualmente, etc. Un ***topoelemento*** podrá contener ***domoelementos*** y otros ***topoelementos***. La instalación constituirá el ***topoelemento*** raíz. Dentro del fichero XML este tipo de etiqueta tendrá como atributo el nombre.
- <**domoElement**>: Representará los dispositivos de la instalación, como por ejemplo un punto de luz, una persiana, una caldera, etc. Un ***domoelemento*** podrá contener uno o varios ***datapoints***. Dentro del fichero XML este tipo de etiqueta tendrá como atributos el nombre y el tipo.
- <**dataPoint**>: Indicará los objetos de comunicación de un dispositivo, como por ejemplo leer estado, subir persiana, leer temperatura, etc. Dentro del fichero XML este tipo de etiqueta tendrá como atributos el nombre, el tipo y la dirección de grupo.

Es necesario definir los tipos de ***domoelementos*** con los que se va a trabajar y sus correspondientes ***datapoints***. Dentro del presente trabajo se han definido los siguientes tipos de ***domoelemento***:

- LIGHT_ON_OFF: Representa una luz todo / nada. Dispondrá de dos ***datapoints***:
  - CONTROL: Permitirá el encendido o apagado de la luz.
  - STATUS: Indicará si la luz está encendida o apagada.
- DOOR_SENSOR: Representa un sensor que indica si una puerta o una ventana está abierta o cerrada. Dispondrá de un único ***datapoint*** denominado STATUS que permitirá obtener el estado de la puerta o de la ventana.

- MOTION_SENSOR: Representa un sensor de movimiento. Contará con un único *datapoint* denominado STATUS que permitirá obtener el estado del sensor.
- POWER_SOCKET: Representa un enchufe controlado por un relé. Dispondrá de los siguientes *datapoints*:
  - CONTROL : Que permitirá su conexión o desconexión.
  - STATUS: Que indicará si está conectado o desconectado.
- ALARM: Representa una alarma. Dispondrá de dos *datapoints* :
  - CONTROL : Que permitirá su armado o desarmado.
  - STATUS: Que indicará si la alarma se ha disparado o permanece en reposo.
- TEMP_SENSOR: Representa un sensor de temperatura. Contará con un único *datapoint* denominado VALUE que permitirá obtener el valor de temperatura medido por el sensor.
- BLIND: Representa una persiana motorizada. Dispondrá de tres *datapoints*:
  - UP_DOWN: Que permitirá subir o bajar la persiana.
  - STOP: Que detendrá el movimiento de la persiana.
  - POSITION: Que indicará la posición de cierre de la persiana.

Queda como futuro trabajo la definición de una biblioteca más amplia de *domoelementos*.

4.4 Traducción a variables y hechos

La traducción se va a producir en base a la siguiente correlación entre los *datapoints* y las variables:

- Los *datapoints* que den información de un dispositivo se convertirán en una variable de entrada.
- Los *datapoints* que permitan actuar sobre un dispositivo se convertirán en una variable de salida.

En cuanto a los hechos de cada variable, van a depender del tipo de datos asociado al *datapoint*; el cual dependiendo del dispositivo puede ser booleano, *float*, etc. La manera de proceder será la siguiente:

- Si se trata de un *datapoint* de tipo booleano, la variable tendrá un hecho por cada posible estado. Por ejemplo, abierta, cerrada, ON, OFF, etc.

- Si se trata de un *datapoint* de tipo no booleano, se deberán definir rangos de valores y asociar un hecho a cada rango. Por ejemplo, Temp_Confort = 20ºC a 25ºC, Temp_Calor = > 27ºC, etc.

Por tanto, para los *domoelementos* definidos en el punto 4.1 se obtendrán las siguientes variables y hechos:

Domoelemento	Variables	Tipo	Hechos
LIGHT_ON_OFF	Control	Salida	Apagar Encender
	Estado	Entrada	Apagada Encendida
DOOR_SENSOR	Estado	Entrada	Cerrada Abierta
MOTION_SENSOR	Estado	Entrada	Inactivo Activo
POWER_SOCKET	Control	Salida	Desconectar Conectar
	Estado	Entrada	Desconectado Conectado
TEMP_SENSOR	Valor	Entrada	Frío Confort Calor
ALARM	Control	Salida	Desarmar Armar
	Estado	Entrada	Reposo Disparada
BLIND	Control	Salida	Subir Bajar
	Detener	Salida	DetenerSubir DetenerBajar
	Posición	Entrada	0% 25% 50% 75% 100%

Tabla 4: Conversión de domoelementos a variables y hechos

4.5 Representación de la red de reglas en XML

La red de reglas va a representarse en formato XML a través de las siguientes etiquetas:

- <**variable**>: Representará una variable de la red de reglas. Tendrá los siguientes atributos:
  - *Nombre*: Será el identificador que utilizará el usuario para la variable.
  - *Tipo*: Indicará si la variable es de entrada o de salida.
  - *Etiqueta*: Será el identificador que establecerá la relación entre una variable y sus hechos.
  - *Datapoint*: Contendrá el identificador del tipo de datos asociado a la variable. Se representará según el estándar [1] en la forma *número_principal.número_secundario*.
  - *Dirección de grupo*: Contendrá la dirección de grupo del *datapoint* representado por la variable.
- <**fact**>: Hará referencia a un hecho de una variable, cada variable podrá tener uno o más hechos. Los atributos de esta etiqueta serán el nombre del hecho y el nombre y la etiqueta de la variable.
- <**rule**>: Representará una regla de la red de reglas. Tendrá como atributos el nombre de la regla y su descripción, la descripción de la regla será una representación escrita de la misma. Una regla podrá tener uno o más antecedentes y consecuentes.
- <**antecedent**>: Indicará cada uno de los hechos que sean antecedentes de una regla. Tendrá como atributos el nombre del hecho y el nombre y la etiqueta de la variable asociada.
- <**consequent**>: Hará referencia a cada uno de los hechos que sean antecedentes de una determinada regla. Tendrá los mismos atributos que la etiqueta <antecedent>.

4.6 Ejecución de las reglas

Para la ejecución de las reglas van a utilizarse tres procesos que funcionarán de manera concurrente:

- Un proceso que detecte eventos en la instalación KNX, como pueden ser la activación de un sensor o el apagado de una luz.
- Un proceso que a partir de dichos eventos y, en función de las reglas definidas por el usuario, genere las órdenes pertinentes para ser ejecutadas sobre la instalación KNX. Dichas órdenes podrán ser encender una luz, subir las persianas hasta el 50% de su recorrido, etc.

- Un proceso que ejecute sobre la instalación KNX las órdenes que se van generando.

Con el fin de desacoplar estos tres procesos entre sí, se utilizarán sendas colas de eventos y órdenes.

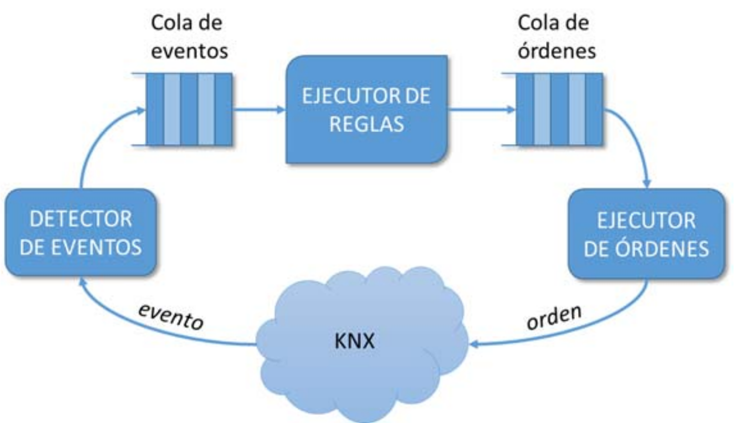


Figura 15: Diagrama de funcionamiento de la ejecución de reglas

El proceso encargado de ejecutar las reglas dispondrá de una representación de la red de reglas en forma de matrices binarias. Cada vez que tome un evento de la cola realizará las siguientes acciones:

- Actualizará el vector de estado en función de la información contenida en el evento. Por ejemplo, en el caso de que el evento indique que un sensor se ha activado, deberá actualizar el estado del hecho correspondiente.
- Examinará las reglas para ver si alguna ha quedado sensibilizada tras la modificación del vector de estado.
- Disparará las reglas sensibilizadas y, por cada cambio en un estado, generará la orden necesaria para que dicho cambio se materialice sobre el dispositivo correspondiente de la instalación KNX.

4.7 Diseño de la aplicación WEB

En los siguientes apartados se detalla el diseño de la aplicación WEB que constituye la parte software del dispositivo, centrándose tanto en las funcionalidades que va a ofrecer como en la interfaz de usuario.

4.7.1 Funcionalidades

A continuación se describen las funcionalidades que va a ofrecer la aplicación:

- **Cargar instalación**: El usuario podrá cargar en la aplicación un fichero en formato XML que represente los elementos de la instalación domótica. Este fichero será interpretado por la aplicación y, a partir de él, creará de manera

automática las variables y los hechos con los que el usuario podrá establecer las reglas de funcionamiento que desee.

- **Editar Variables**: Va a permitir al usuario cambiar el nombre de las variables según su elección. Por defecto el nombre y la etiqueta de las variables será de la forma:

  domoelemento@topoelemento

  Tras realizar cualquier cambio, se actualizará el nombre de las variables en la descripción de las reglas que ya se hubieran definido. Esta opción no modificará la etiqueta de la variable.

- **Crear una nueva regla**: Con esta funcionalidad el usuario podrá definir una nueva regla estableciendo sus antecedentes y consecuentes. Cada regla tendrá un nombre y una descripción que se generarán de manera automática. La descripción contendrá los antecedentes y consecuentes de la regla en la forma:

  SI (ant_1 Y .. Y ant_n) ENTONCES (cons_1 Y .. Y cons_m)

  Esta opción estará inicialmente inhabilitada hasta que el usuario cargue un fichero de instalación o uno de reglas.

- **Ejecutar reglas**: Va a permitir la conexión con la instalación KNX para poder ejecutar sobre ella las reglas definidas por el usuario. Tanto la dirección IP del ordenador monoplaca como la de la pasarela KNX/IP serán detectadas automáticamente por la aplicación. El usuario podrá definir el puerto de comunicación, ofreciéndose por defecto el número 3671.
- **Cargar reglas**: Mediante esta funcionalidad el usuario podrá cargar en la aplicación un fichero en formato XML conteniendo una red de reglas.
- **Guardar reglas**: Esta funcionalidad va a permitir al usuario guardar en un fichero con formato XML la red de reglas creada.
- **Editar una regla existente**: A través de esta funcionalidad el usuario podrá editar una regla tras seleccionarla de una lista, modificando, si lo desea, sus antecedentes y consecuentes. La descripción de la regla se actualizará automáticamente.
- **Eliminar una regla existente**: Esta opción permitirá al usuario eliminar una regla tras seleccionarla previamente.
- **Establecer los antecedentes de una regla**: El usuario podrá elegir de la lista de hechos de entrada, que ofrecen los elementos de la instalación domótica, aquellos que vayan a actuar como antecedentes de una determinada regla. En caso de que la regla presente alguna anomalía, se indicará en la barra de estado y no se permitirá el guardado de la misma.

- **Establecer los antecedentes de una regla**: De manera análoga a la anterior funcionalidad, el usuario podrá escoger de la lista de hechos de salida, que ofrecen los elementos de la instalación domótica, aquellos que vayan a actuar como consecuentes de una determinada regla. Al igual que en la funcionalidad anterior, no se permitirá el guardado de reglas que presenten alguna anomalía.

4.7.2 Diseño de la interfaz de usuario

Se ha pensado dividir la aplicación en las siguientes pantallas:

- **Pantalla principal**: En la que se va a mostrar un menú con las opciones de carga y guardado de ficheros y acceso a la creación, edición y ejecución de reglas. Además, se dispondrá con un listado de las reglas definidas hasta el momento indicando su nombre y descripción. Dicho listado permitirá la selección de un elemento y
- **Editor de variables**: En esta pantalla se va a mostrar la lista de variables indicando su etiqueta y su nombre. La lista permitirá seleccionar un elemento y contará con un botón que mostrará un diálogo para editar el nombre de la variable seleccionada.
- **Editor de reglas**: En esta pantalla se va a disponer de un menú con las opciones para establecer los antecedentes y consecuentes de una regla, así como el guardado de la regla o la cancelación de la operación. Además se va mostrar una representación de la regla que se irá actualizando a medida que se varíen los antecedentes y consecuentes de la misma. También contará con una barra en la parte inferior con información de estado.

A continuación se presentan los prototipos de las pantallas arriba descritas:

http://DIR_IP:8080/rulesnetKNX

Red de reglas

	Nombre	Descripción
○	R0001	Descripción de la regla
○	R0002	Descripción de la regla
◉	R0003	Descripción de la regla

Editar Borrar

Figura 16: Prototipo - Pantalla principal

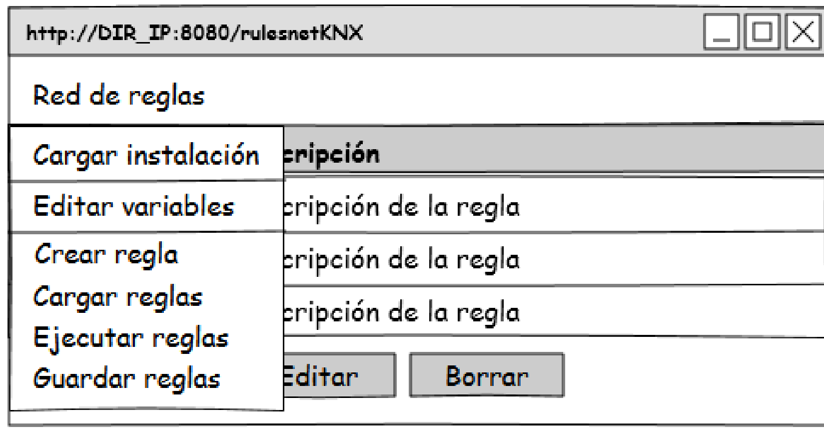


Figura 17: Prototipo - Menú de la pantalla principal

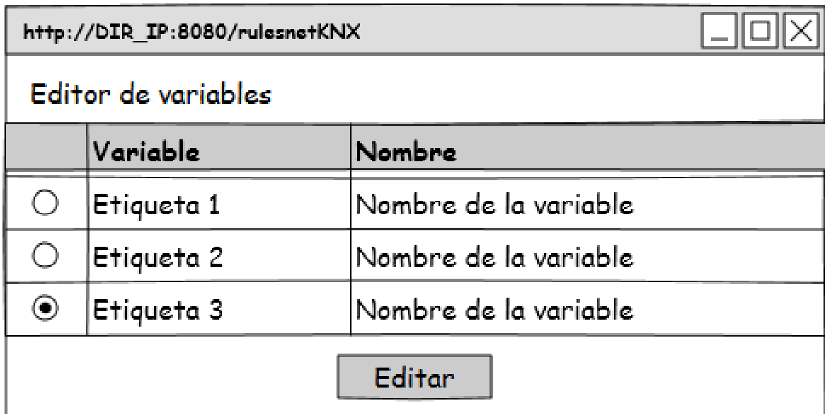


Figura 18: Prototipo - Pantalla del editor de variables

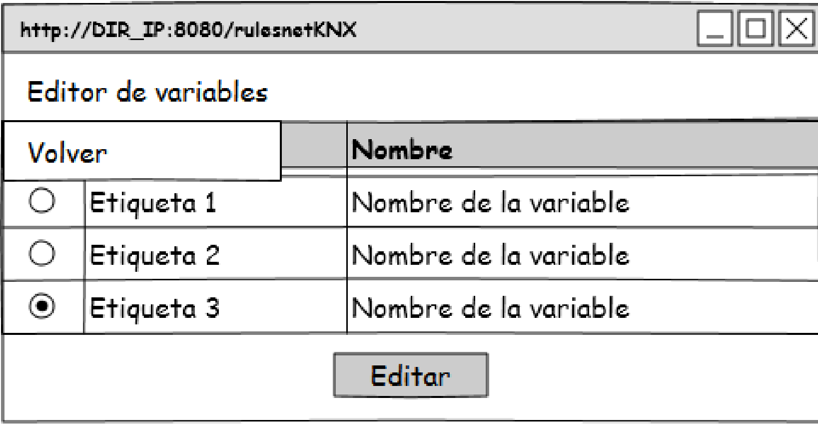


Figura 19: Prototipo - Menú del editor de variables

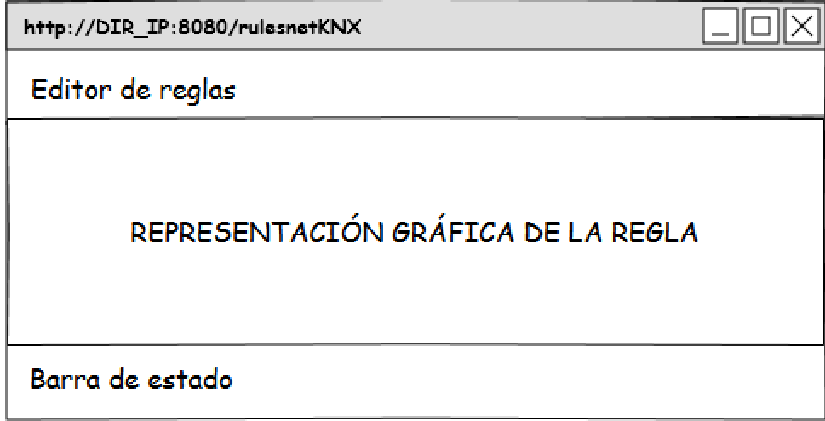


Figura 20: Prototipo - Pantalla del editor de reglas

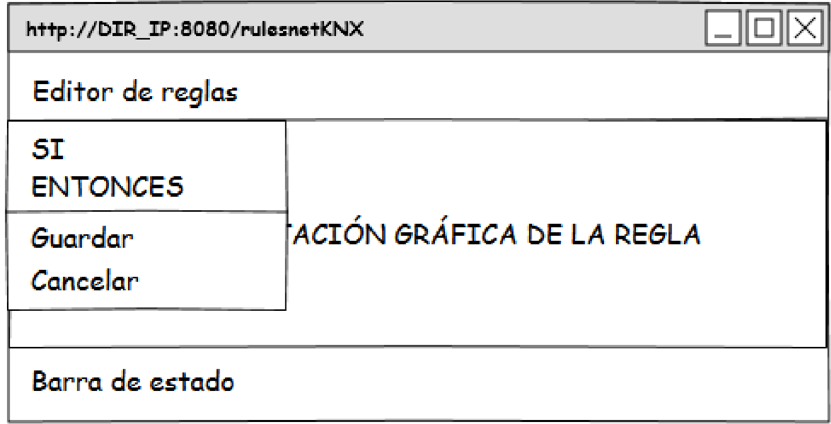


Figura 21: Prototipo - Menú del editor de reglas

Se han diseñado una serie de cuadros de diálogo para las siguientes tareas:

- Cargar instalación.
- Cargar reglas.
- Editar el nombre de una variable
- Establecer los antecedentes y consecuentes de una regla.
- Conectar con la instalación KNX y ejecutar sobre ella las reglas definidas.

A continuación se muestran los prototipos de los diálogos arriba mencionados:

Figura 22: Prototipo - Diálogo para la carga de un fichero de instalación

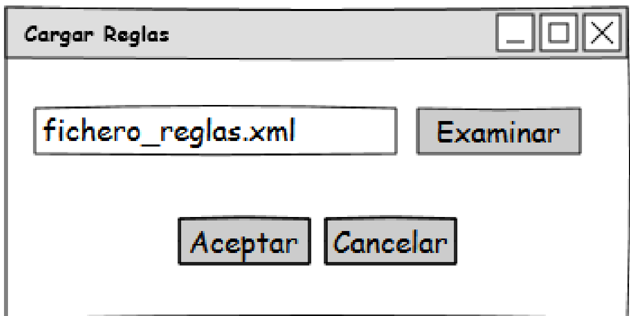


Figura 23: Prototipo - Diálogo para la carga de un fichero de reglas

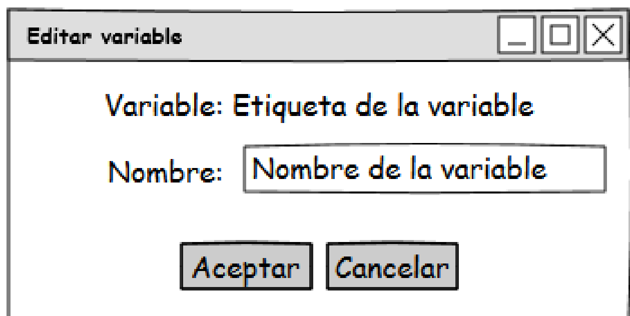


Figura 24: Prototipo - Diálogo para editar el nombre de una variable

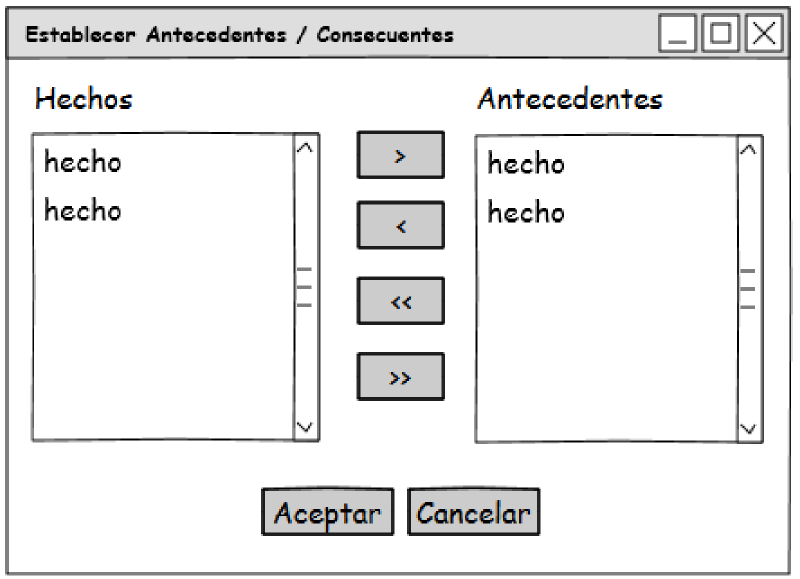


Figura 25: Prototipo - Diálogo para establecer antecedentes / consecuentes

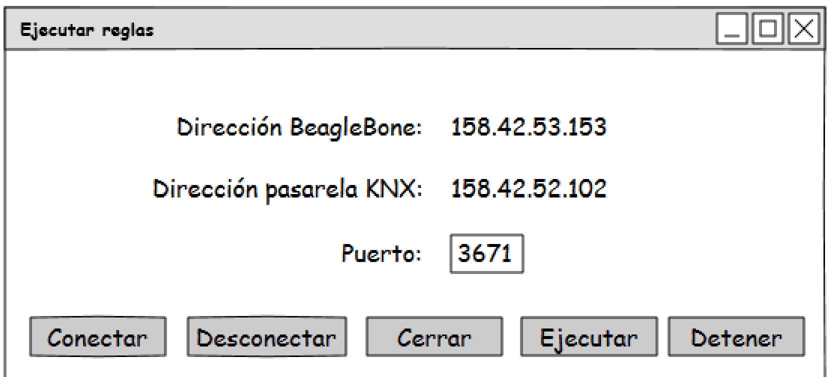


Figura 26: Prototipo - Diálogo para conexión a KNX y ejecución de reglas

4.7.3 Diagrama de navegación

En la siguiente figura se muestra el diagrama de navegación entre las distintas pantallas y diálogos de la aplicación:

Figura 27: Diagrama de navegación de la aplicación WEB

5 Implementación

5.1 Introducción

La aplicación WEB se ha implementado en Java haciendo uso del JDK 1.8. Como entorno de desarrollo se ha utilizado *NetBeans* en su versión 8.0.2. Para simplificar la implementación de la aplicación, se ha hecho uso del *framework* Java Server Faces y concretamente de los componentes enriquecidos que proporciona la librería de código abierto *PrimeFaces* en su versión 5.2. Además de la amplia cantidad de componentes que ofrece, entre las características de dicha librería cabe destacar:

- Soporte de Ajax con actualización parcial, pudiendo determinar qué componentes de la página se actualizarán y cuáles no.
- Componentes para el desarrollo de aplicaciones destinadas a dispositivos móviles.
- Es una librería ligera y fácil de instalar, basta con incluir un *.jar* en el proyecto para hacer uso de todos sus componentes sin necesidad de configuración adicional.
- Cuenta con el soporte de una amplia comunidad de desarrolladores.

El interfaz gráfico de la aplicación se ofrece al usuario a través de una página JSF (`index.xhmtl`). La ruta de acceso a esta página dentro del servidor viene dada en el fichero de configuración `META-INF/context.xml`.

La lógica de la aplicación, el manejo de eventos, etc. Se realiza mediante los *managed bean*. Además, se dispone de las clases correspondientes para implementar los objetos que contienen los datos de la aplicación.

5.2 Descripción de la página JSF

A continuación se muestra la estructura básica de la página JSF encargada de proporcionar el interfaz gráfico a la aplicación:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:p="http://primefaces.org/ui">

<f:view contentType="text/html">
  <h:head>
    <f:facet name="first">
      <meta content='text/html;
            charset=UTF-8' http-equiv="Content-Type"/>
      <title>Editor de Reglas</title>
    </f:facet>
  </h:head>
  <h:body>
    <h:outputStylesheet name="css/styles.css"/>
    <h:form id="reform" enctype="multipart/form-data>
        .
        .
     CONTROLES
        .
        .
    </h:form>
  </h:body>
</f:view>
</html>
```

Al principio de la página se incluyen los espacios de nombre XML necesarios para acceder a las etiquetas ofrecidas por *JSF* y *PrimeFaces*. La etiqueta `<f:view>` permite fijar la manera en que el navegador interpreta la página JSF proporcionada por el servidor, en este caso según el modo *text/html*. Si no se especifica nada, algunos navegadores pueden interpretarla en el modo *application/xhtml+xml*, causando así problemas de funcionamiento. Por último, hacer notar que todos los controles que se utilicen deben ir dentro de un formulario. En el ANEXO II se detallan los controles empleados para implementar la aplicación.

5.3 Configuración del aspecto.

PrimeFaces tiene predefinidos una serie de temas que pueden descargarse desde su repositorio. Para instalar un tema es suficiente con copiarlo en el *classpath* de la aplicación e incluir en el fichero de configuración `WEB-INF/web.xml` el correspondiente parámetro de contexto. En este caso se ha utilizado el tema `bootstrap`.

```
<context-param>
  <param-name>primefaces.THEME</param-name>
  <param-value>bootstrap</param-value>
</context-param>
```

Es posible modificar el aspecto de un tema predefinido creando clases de estilo CSS. Están clases están definidas en el fichero `resources/css/styles.css` y para utilizarlas desde los distintos controles se emplea el atributo `styleClass`.

```
      .
      .
.ui-button-dialog {
    width: 100px;
    margin-left: 2px;
    margin-right: 2px;
}
      .
      .
```

Los iconos que aparecen en la aplicación, pertenecen a una librería de *PrimeFaces* denominada *FontAwesome*. Para poder utilizar esta librería es preciso añadir en el fichero de configuración `WEB-INF/web.xml` el correspondiente parámetro de contexto.

```
<context-param>
  <param-name>primefaces.FONT_AWESOME</param-name>
  <param-value>true</param-value>
</context-param></p:dialog>
```

Después, basta con indicar en el atributo `icon` del control correspondiente, el icono que se desea emplear precedido de los caracteres `fa` y un espacio.

```
      .
      .
<p:menuitem value="Guardar"
                  .
                  .
            icon="fa fa-save"
      .
      .
```

5.4 Descripción de los *managed bean*

Un ***managed bean*** no es más que una clase que se va a encargar de la lógica de la aplicación (manejo de eventos, validación, navegación, etc.). Para que la clase sea reconocida por el ***framework*** JSF como un ***managed bean***, se debe anteponer a su declaración la anotación `@ManagedBean`. Esta anotación admite el atributo `name` para indicar como será referenciado el ***managed bean***. En caso de no indicar el atributo `name`, el ***managed bean*** será referenciado por el nombre de su clase. A continuación puede verse la estructura básica de un ***managed bean***.

```
/** CLAUSULAS import **/

@ManagedBean
@ApplicationScoped
public class MainBean implements Serializable {

/** ATRIBUTOS **/

/** CONSTRUCTOR **/
  public MainBean (){..}

/** MÉTODOS GET y SET **/

/** MANEJADORES DE EVENTOS **/

}
```

La anotación que aparece a continuación de `@ManagedBean` indica el ámbito del ***managed bean*** y por tanto el de sus atributos. Las principales anotaciones que se puede utilizar para definir el ámbito del ***managed bean*** son las siguientes:

- **@RequestScoped:** El ***managed bean*** se crea cuando se produce una petición HTTP y se destruye cuando finaliza la respuesta HTTP asociada.
- **@ViewScoped:** El ***managed bean*** se crea cuando se produce una petición HTTP y permanece vivo mientras el usuario no recargue la página en el navegador.
- **@SessionScoped:** El ***managed bean*** se crea con la primera petición HTTP que lo involucra y permanece vivo mientras no se cierre o invalide la sesión, por ejemplo cerrando completamente el navegador.
- **@ApplicationScoped:** El ***managed bean*** se crea con la primera petición HTTP que lo involucra y permanece vivo mientras lo esté la aplicación WEB.

Si no se especifica ninguna anotación se utiliza por defecto `@RequestScoped`. Se aconseja que cuando se utiliza un ámbito distinto al `@RequestScoped`, la clase del ***managed bean*** implemente el interfaz `Serializable`.

Aunque se puede utilizar un único ***managed bean*** para todos los controles de la aplicación, por cuestiones de claridad y posterior mantenimiento, es conveniente utilizar varios ***managed bean***. JSF permite la inyección directa de un ***managed bean*** dentro de otro, pudiendo así estructurar la lógica de la aplicación en varios ***managed bean***. El ***managed bean*** a inyectar se definirá de la manera habitual:

```
/** CLAUSULAS import **/

@ManagedBean(name = "ruleEditor")
@ApplicationScoped
public class RuleEditorBean implements Serializable{

/** ATRIBUTOS **/

/** CONSTRUCTOR **/

/** MÉTODOS GET y SET **/

/** MANEJADORES DE EVENTOS **/

}
```

En el ***managed bean*** principal se deberá incluir la instanciación del ***managed bean*** a inyectar, precedido por la anotación `@ManagedProperty`. También será imprescindible incluir el método `set` correspondiente:

```
/** CLAUSULAS import **/

@ManagedBean
@ApplicationScoped
public class MainBean implements Serializable {

/** ATRIBUTOS **/

@ManagedProperty(value = "#{ruleEditor}")
private RuleEditorBean ruleEditorBean;

/** CONSTRUCTOR **/

/** MÉTODOS GET y SET **/

public void setRuleEditorBean(RuleEditorBean ruleEditorBean) {
  this.ruleEditorBean = ruleEditorBean;
}

/** MANEJADORES DE EVENTOS **/

}
```

A continuación puede verse la estructura de *managed bean* empleada en la aplicación:

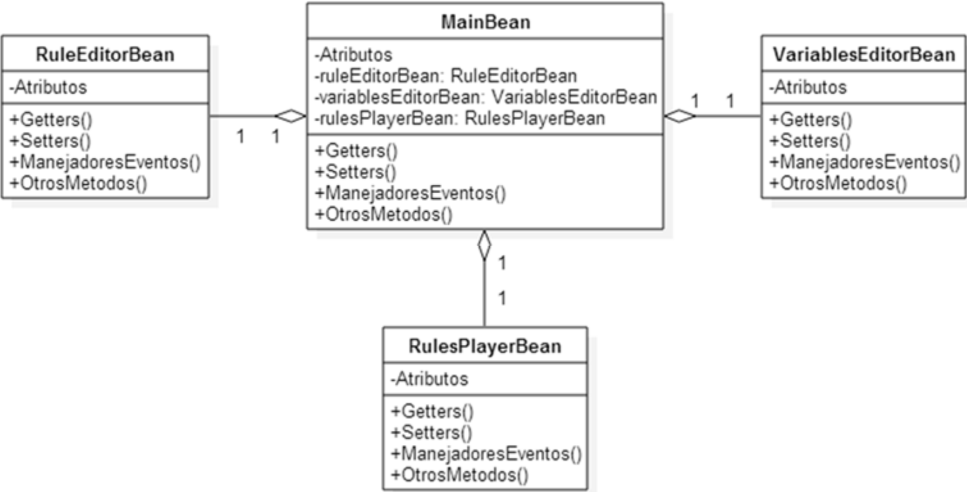


Figura 28: Diagrama de clases de los managed bean

Los métodos `get()` y `set()` permiten a los controles de la página JSF leer y escribir en los distintos atributos del *managed bean*. Cabe mencionar el método `getFileDownload()` que, además de proporcionar al control *FileDownload* el fichero en el que se guardará la red de reglas, realiza la conversión de objeto Java a XML. Dicha conversión será descrita en detalle en el punto 5.7.

```
public StreamedContent getFileDownload() {
  try {
    StringWriter sw = new StringWriter();
    JAXBContext jaxbContext = JAXBContext.newInstance(RulesNetXML.class);
    Marshaller marshaller = jaxbContext.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    marshaller.marshal(rulesNet, sw);
    InputStream stream = new ByteArrayInputStream(sw.toString().getBytes("UTF-8"));
    fileDownload = new DefaultStreamedContent(stream, "text/xml", "rulesNet.xml");
  } catch (JAXBException | UnsupportedEncodingException ex) {
     Logger.getLogger(RulesEditorBean.class.getName()).log(Level.SEVERE, null, ex);
  }
  return fileDownload;
}
```

Los métodos manejadores de eventos se ejecutan cuando el usuario realiza una acción sobre un control, como por ejemplo elegir una opción de un menú o hacer clic en un botón. Además, los *managed bean* pueden contener otros métodos necesarios para implementar la lógica de la aplicación. En el ANEXO II se detalla el contenido de los distintos *managed bean* empleados en la implementación de la aplicación.

5.5 Representación en Java de la instalación KNX

En la siguiente figura se muestra el diagrama de clases con el que se ha representado la instalación KNX. Al leer el fichero *.xml* que contiene la instalación, se rellena con su contenido una instancia de la clase *TopoElement*.

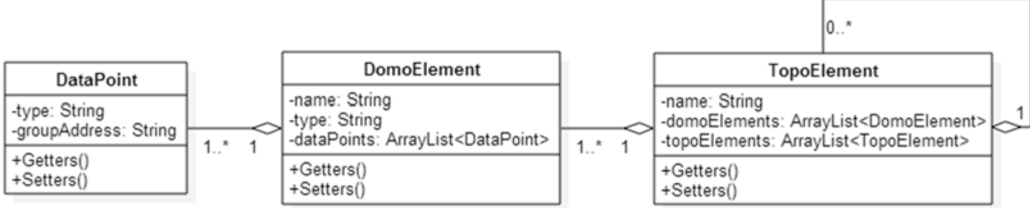


Figura 29: Diagrama de clases de la instalación KNX

5.6 Representación en Java de la red de reglas

En la siguiente figura se muestra el diagrama de clases que representa la red de reglas. Al cargar el fichero *.xml* que contiene una red de reglas definidas por el usuario, se rellena con su contenido un objeto de la clase *RulesNetXML*. Del mismo modo, al guardar la red de reglas en un fichero *.xml*, se vuelca sobre el fichero el contenido de un objeto de dicha clase.

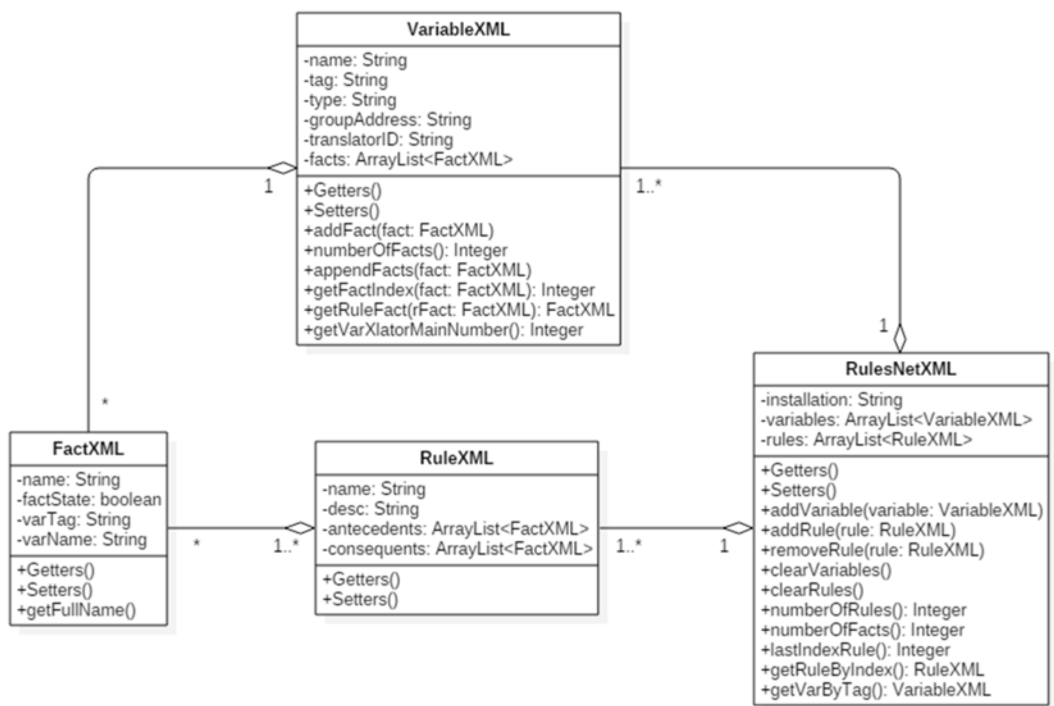


Figura 30: Diagrama de clases de la red de reglas

5.7 Conversión Java <-> Xml

Para la conversión de objetos Java a XML y viceversa, se ha hecho uso de la API JAXB (*Java Architecture for XML Binding*) que se incluye en la plataforma Java EE. Esta API permite almacenar y recuperar datos en memoria en cualquier formato XML, sin la necesidad de implementar un conjunto específico de rutinas de carga y guardado de XML para la estructura de clases del programa.

De una parte se han colocado, en las clases que implementan la instalación y la red de reglas, una serie de anotaciones para establecer las correspondencias entre los atributos de dichas clases y los elementos y atributos XML. De otra parte, se han creado instancias de objetos ***marshaller*** y ***unmarshaller*** para la conversión de objeto Java a XML y viceversa. A continuación se describen las anotaciones empleadas:

- **@XmlRootElement**: Se coloca inmediatamente antes de la declaración de la clase y define el elemento raíz del XML. El nombre del elemento raíz deriva del nombre de la clase, pero puede cambiarse mediante el parámetro `name` de la anotación.

```
      .
      .
import javax.xml.bind.annotation.XmlRootElement;
      .
      .
@XmlRootElement(name = "RulesNet")
public class RulesNetXML {
      .
      .
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RulesNet .. >
      .
      .
</RulesNet>
```

- **@XmlType**: Especifica el orden en que los elementos XML serán generados. Se coloca a continuación de la anotación `@XmlRootElement`.

```
      .
      .
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
      .
      .
@XmlRootElement(name = "RulesNet")
@XmlType(propOrder = {"installation","variables","rules"})
public class RulesNetXML {
      .
      .
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RulesNet installation="Nombre instalación">
  <variables>
      .
      .
  </variables>
  <rules>
      .
      .
  </rules>
</RulesNet>
```

- **@XmlAttribute** y **@XmlElement**: Se colocan delante de los métodos `get()` de aquellos atributos de la clase que deban corresponderse con un atributo o un elemento del XML. El nombre que aparecerá en XML coincidirá con el que tenía el atributo de la clase, a menos que se indique con el parámetro `name`.

```
      .
      .
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
      .
      .
@XmlAttribute
public String getName() {
  return name;
}     .
      .
@XmlElement(name = "domoElement")
public ArrayList<DomoElementXML> getDomoElements() {
  return domoElements;
}     .
      .
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
      .
      .
  <topoElement name="Cocina">
    <domoElement ..>
      .
      .
  </topoElement>
      .
      .
```

- **@XmlElementWrapper**: Permite agrupar en el XML una serie de elementos del mismo tipo.

```
      .
      .
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
      .
      .
@XmlElement(name = "rule")
@XmlElementWrapper(name = "rules")
public ArrayList<RuleXML> getRules() {
  return rules;
}
      .
      .
```

```
      .
      .
<rules>
  <rule name="R0001" desc="..">
      .
      .
  </rule>
  <rule name="R0002" desc="..">
      .
      .
  </rule>
      .
      .
</rules>
      .
      .
```

- **@XmlTransient:** Se utiliza para indicar que no se desea incluir un determinado elemento en el XML o leer su valor de este.

```
      .
      .
import javax.xml.bind.annotation.XmlTransient;
      .
      .
@XmlTransient
public String getFullName() {
  return this.varName +"."+ this.name;
}
      .
      .
```

Para la conversión de objeto Java a XML es necesario instanciar un *marshaller*. Los pasos para crear un *marshaller* son:

1. Instanciar un contexto JAXB indicando como parámetro la clase del objeto que se quiera convertir a XML.
2. Crear el objeto *marshaller* mediante el método `createMarshaller()` del objeto contexto.
3. Fijar en las propiedades del *marshaller* el formato de salida del XML.
4. Invocar el método `marshal` del objeto *marshaller*, pasándole como parámetros el objeto a convertir y el flujo de salida de datos.

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
      .
      .
try {
  StringWriter sw = new StringWriter();
  JAXBContext jaxbContext =   JAXBContext.newInstance(RulesNetXML.class);
  Marshaller marshaller = jaxbContext.createMarshaller();
  marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
  marshaller.marshal(rulesNet, sw);
} catch (JAXBException | UnsupportedEncodingException ex) {
      .
      .
}
```

Para realizar el proceso a la inversa y rellenar un objeto Java con el contenido de un XML, hay que instanciar un *unmarshaller*. Los pasos para su creación son:

1. Instanciar un contexto JAXB indicando como parámetro la clase del objeto que se quiera rellenar a partir del XML.
2. Crear el objeto *unmarshaller* mediante el método `createUnmarshaller()` del objeto contexto.
3. Invocar el método `unmarshal` del objeto *unmarshaller*, pasándole como parámetro el flujo de entrada de datos. El método devuelve un objeto genérico Java, por lo que es necesario realizar el *casting* correspondiente

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller; .
      .
      .
InputStream uploadFile;
try {
  uploadFile = event.getFile().getInputstream();
  JAXBContext jaxbContext = JAXBContext.newInstance(RulesNetXML.class);
  Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
  rulesNet = (RulesNetXML) unmarshaller.unmarshal(uploadFile);
} catch (JAXBException | IOException ex) {
      .
      .
}
```

5.8 Conversión a variables y hechos

Para la conversión de los elementos de la instalación a variables y hechos, se ha creado la clase `VariableFactory`. Tal como se expuso en el apartado 4.4, cada ***datapoint*** será convertido en una variable cuyos hechos dependerán del tipo de ***domoelemento*** al que pertenezca dicho ***datapoint***.

Tras la carga del fichero XML que contiene la instalación, se creará un objeto de esta clase. A continuación, se invocará al método `createVariables`, pasándole como parámetro el objeto que contiene el ***topoelemento*** raíz de la instalación. Finalmente, se cargará el conjunto de variables resultante en el objeto que contiene la red de reglas.

```
private TopoElementXML installation;
      .
      .
installation = (TopoElementXML) unmarshaller.unmarshal(uploadFile);
VariableFactory varFactory = new VariableFactory();
varFactory.createVariables(installation);
rulesNet.setVariables(varFactory.getVariables());
```

El método `createVariables` va recorriendo la estructura contenida en el ***topoelemento*** raíz y, para cada ***domoelemento***, invoca al método de creación de variables específico para ese tipo de ***domoelemento***. A continuación puede verse como ejemplo los métodos encargados de crear las variables y hechos para un ***domoelemento*** de tipo luz todo/nada, el correspondiente al domoelemento persiana y el del sensor de temperatura:

```
public void createVarsLightOnOff(DomoElementXML domoElement,
                                 String topoElement) {
  for (DataPointXML dataPoint : domoElement.getDataPoints()) {
    switch (dataPoint.getType()) {
      case "CONTROL":
        VariableXML var = new VariableXML();
        var.setTag("Control_" + domoElement.getName() + "@"
                   + topoElement);
        var.setName(var.getTag());
        var.setType("OUTPUT");
        var.setTranslatorID(DPTXlatorBoolean.DPT_SWITCH.getID());
        var.setGroupAddress(dataPoint.getGroupAddress());
        var.addFact(new FactXML("Apagar", var.getTag()));
        var.addFact(new FactXML("Encender", var.getTag()));
        variables.add(var);
        break;
            .
```

```
            .
      case "STATUS":
        var = new VariableXML();
        var.setTag("Estado_" + domoElement.getName() + "@"
                   + topoElement);
        var.setName(var.getTag());
        var.setType("INPUT");
        var.setTranslatorID(DPTXlatorBoolean.DPT_SWITCH.getID());
        var.setGroupAddress(dataPoint.getGroupAddress());
        var.addFact(new FactXML("Apagada", var.getTag()));
        var.addFact(new FactXML("Encendida", var.getTag()));
        variables.add(var);
        break;
      default:
        break;
    }
  }
}
```

```
public void createVarsBlind(DomoElementXML domoElement,
                            String topoElement) {
  for (DataPointXML dataPoint : domoElement.getDataPoints()) {
    switch (dataPoint.getType()) {
      case "UP_DOWN":
        VariableXML var = new VariableXML();
        var.setTag("Control_" + domoElement.getName() + "@"
                   + topoElement);
        var.setName(var.getTag());
        var.setType("OUTPUT");
        var.setTranslatorID(DPTXlatorBoolean.DPT_UPDOWN.getID());
        var.setGroupAddress(dataPoint.getGroupAddress());
        var.addFact(new FactXML("Subir", var.getTag()));
        var.addFact(new FactXML("Bajar", var.getTag()));
        variables.add(var);
        break;
      case "STOP":
        var = new VariableXML();
        var.setTag("Detener_" + domoElement.getName() + "@"
                   + topoElement);
        var.setName(var.getTag());
        var.setType("OUTPUT");
        var.setTranslatorID(DPTXlatorBoolean.DPT_TRIGGER.getID());
        var.setGroupAddress(dataPoint.getGroupAddress());
        var.addFact(new FactXML("DetenerSubir", var.getTag()));
        var.addFact(new FactXML("DetenerBajar", var.getTag()));
        variables.add(var);
        break;
          .
```

```
            .
      case "POSITION":
        var = new VariableXML();
        var.setTag("Posicion_" + domoElement.getName() + "@"
                   + topoElement);
        var.setName(var.getTag());
        var.setType("INPUT");
        var.setTranslatorID(DPTXlator8BitUnsigned.
                           DPT_SCALING.getID());
        var.setGroupAddress(dataPoint.getGroupAddress());
        var.addFact(new FactXML("0%", var.getTag()));
        var.addFact(new FactXML("25%", var.getTag()));
        var.addFact(new FactXML("50%", var.getTag()));
        var.addFact(new FactXML("75%", var.getTag()));
        var.addFact(new FactXML("100%", var.getTag()));
        variables.add(var);
        break;
      default:
        break;
      }
    }
  }
}
```

```
public void createVarsTemp(DomoElementXML domoElement,
                           String topoElement) {
  for (DataPointXML dataPoint : domoElement.getDataPoints()) {
    switch (dataPoint.getType()) {
      case "VALUE":
        VariableXML var = new VariableXML();
        var.setTag("Valor_" + domoElement.getName() + "@"
                   + topoElement);
        var.setName(var.getTag());
        var.setType("INPUT");
        var.setTranslatorID(DPTXlator2ByteFloat.
                           DPT_TEMPERATURE.getID());
        var.setGroupAddress(dataPoint.getGroupAddress());
        var.addFact(new FactXML("Frío", var.getTag()));
        var.addFact(new FactXML("Confort", var.getTag()));
        var.addFact(new FactXML("Calor", var.getTag()));
        variables.add(var);
        break;
      default:
        break;
    }
  }
}
```

5.9 Conexión con KNX

Para poder conectar con la pasarela KNX/IP que proporciona el acceso a la instalación KNX, es preciso conocer los siguientes parámetros:

- Dirección IP local, que en nuestro caso será la dirección IP del ordenador monoplaca.
- Dirección IP de la pasarela KNX/IP.

- Puerto empleado para la comunicación.

Tanto la dirección IP del ordenador monoplaca como la de la pasarela KNX/IP se obtendrán de manera automática. El puerto podrá ser establecido por el usuario, fijándose como valor por defecto el 3671.

Se ha creado la clase `HandlerKNX` que es la encargada de proporcionar lo necesario para establecer la conexión con KNX. Para obtener la dirección IP de la *BeagleBone* se dispone del método `discoverBeagleIP` cuyo código se muestra a continuación:

```
private String discoverBeagleIP() {
  String localIP = "0.0.0.0";
  try {
    localIP = InetAddress.getLocalHost().getHostAddress();
  } catch (UnknownHostException ex) {
      Logger.getLogger(HandlerKNX.class.getName()).log(Level.SEVERE,
                        null, ex);
  }
  return localIP;
}
```

Una vez conocida la IP de la *BeagleBone*, el siguiente paso es obtener la dirección IP de la pasarela KNX/IP. Para lo cual, se dispone del método `discoverGatewayIP` al que se le pasan como parámetros la IP de la *BeagleBone* y el puerto de comunicación.

```
private String discoverGatewayIP(String IP, int port) {
  String gwIP = "0.0.0.0";
  InetAddress localIP;
  NetworkInterface netIface;
  Discoverer discover;
  SearchResponse resp[];
  Boolean useNAT = false, useMCAST = false;
  try {
    localIP = InetAddress.getByName(IP);
    netIface = NetworkInterface.getByInetAddress(localIP);
    discover = new Discoverer(localIP, port, useNAT, useMCAST);
    discover.startSearch(netIface, 3, true);
    resp = discover.getSearchResponses();
    if (resp.length != 0) {
      gwIP = resp[0].getControlEndpoint().getAddress().toString();
      gwIP = gwIP.substring(1, gwIP.length());
    }
  } catch (UnknownHostException | SocketException | KNXException |
          InterruptedException ex) {
      Logger.getLogger(HandlerKNX.class.getName()).log(Level.SEVERE,
                        null, ex);
        }
  return gwIP;
}
```

Este método hace uso de un objeto de la clase `Discoverer` de ***Calimero*** que, mediante el método `startSearch`, realiza en la red una búsqueda a través del interfaz de red que se le indique; en este caso el interfaz de red va a ser el de la ***BeagleBone***. Una vez finalizada la búsqueda, se obtiene el resultado mediante el método `getSearchResponses` en la forma de un vector. En el elemento inicial de dicho vector se encuentra almacenada la IP de la pasarela en caso de que haya sido encontrada.

Conocidas ambas direcciones IP y el puerto de comunicaciones, ya es posible establecer una conexión mediante el método `openConnection`.

```
public void openConection() {
  Boolean useNAT = false;
  if (!gatewayIP.equals("0.0.0.0")) {
    try {
      beagleISA = new InetSocketAddress(InetAddress.
                      getByName(beagleIP), portKNX);
      gatewayISA = new InetSocketAddress(InetAddress.
                       getByName(gatewayIP), portKNX);
      link = new KNXNetworkLinkIP(KNXNetworkLinkIP.TUNNELING,
                                  beagleISA, gatewayISA,
                                  useNAT, TPSettings.TP1);
      proCom = new ProcessCommunicatorImpl(link);
      connected=true;
    } catch (UnknownHostException | KNXException |
             InterruptedException ex) {
        Logger.getLogger(HandlerKNX.class.getName()).
                        log(Level.SEVERE, null, ex);
        FacesContext context = FacesContext.getCurrentInstance();
        context.addMessage(null,
                           new FacesMessage(FacesMessage.
                           SEVERITY_ERROR,"ERROR de comunicación",
                           ex.getMessage()));
    }
  }
}
```

En primer lugar, se crea un enlace a KNX mediante un objeto de la clase `KNXNetworkLinkIP` de ***Calimero***. Al constructor de este objeto se le pasan los siguientes parámetros: dos sockets, formados por el puerto y la IP de la ***BeagleBone*** y la pasarela KNX/IP respectivamente; información del tipo de conexión, que en nuestro caso va a ser de tipo *TUNNELING*; información del medio de transmisión, que en nuestro caso será par trenzado tipo *TP1*; indicación de si se va utilizar o no NAT para la conexión, en nuestro caso no utilizaremos NAT.

Una vez establecido el enlace, ya se puede crear un objeto de la clase `ProcessComunicatorImpl` de ***Calimero***, el cual proporcionará métodos para escribir y leer en las direcciones de grupo de los distintos objetos de comunicación. En la clase `HandlerKNX` se han implementado métodos para la lectura de algunos de los tipos de datos más habituales. Una vez establecida la conexión con KNX, se utilizan estos

métodos para conocer el estado actual de los hechos asociados a las variables de entrada.

```
public Boolean readBoolSensor (String grpAddr) {
  Boolean value = false;
  try {
    value = proCom.readBool(new GroupAddress(grpAddr));
  } catch (KNXTimeoutException | KNXRemoteException |
           KNXLinkClosedException | KNXFormatException |
           InterruptedException ex) {
      Logger.getLogger(HandlerKNX.class.getName()).log(Level.SEVERE,
                       null, ex);
  }
  return value;
}

public int readUnsigned8bitSensor(String grpAddr, String scale) {
  int value = 0;
  try {
    value = proCom.readUnsigned(new GroupAddress(grpAddr), scale);
  } catch (KNXTimeoutException | KNXRemoteException |
           KNXLinkClosedException | KNXFormatException |
           InterruptedException ex) {
      Logger.getLogger(HandlerKNX.class.getName()).log(Level.SEVERE,
                       null, ex);
  }
  return value;
}

public Float readFloatSensor (String grpAddr) {
Float value = 0.0f;
  try {
    value = proCom.readFloat(new GroupAddress(grpAddr));
  } catch (KNXTimeoutException | KNXRemoteException |
           KNXLinkClosedException | KNXFormatException |
           InterruptedException ex) {
      Logger.getLogger(HandlerKNX.class.getName()).log(Level.SEVERE,
                       null, ex);
  }
  return value;
}
```

Para cerrar la conexión con KNX, la clase proporciona el método `closeConnection` cuyo código se muestra a continuación:

```
public void closeConnection() {
  link.close();
  connected=false;
  beagleISA = null;
  gatewayISA = null;
  gatewayIP = "0.0.0.0";
}
```

Además, se dispone de los métodos `setEventListener` y `removeEventListener` cuya utilidad se detalla en el apartado 5.10.

5.10 Detección de eventos KNX

Para implementar el hilo de ejecución que se encarga de la detección de eventos en el bus KNX, se ha creado la clase `EventListenerKNX`, la cual implementa el interfaz `NetworkLinkListener` de *Calimero*.

Al constructor de esta clase se le proporciona el objeto que contiene la red de reglas y la cola en la que el hilo deberá ir introduciendo los eventos que va detectando.

```
public EventListenerKNX(RulesNetKNX rulesNet,
                        BlockingQueue<EventKNX> eventQueue) {
  varsDir = new HashMap<>();
  for (VariableXML var : rulesNet.getVariables()) {
    if (var.getType().equals("INPUT")) {
      varsDir.put(var.getGroupAddress(), var);
    }
    if (var.getType().equals("OUTPUT") && !varsDir.containsValue(
                                     var.getGroupAddress())) {
        varsDir.put(var.getGroupAddress(), var);
    }
  }
  this.eventQueue = eventQueue;
}
```

Para arrancar el hilo de ejecución se debe crear un objeto de esta clase y asociarlo al enlace establecido al abrir la conexión con KNX, para ello se emplea el método `setEventListener` de la clase `HandlerKNX`.

```
public void setEventListener(EventListenerKNX eventListener) {
  this.eventListener = eventListener;
  link.addLinkListener((NetworkLinkListener) eventListener);
}
```

```
handlerKNX = new HandlerKNX();
EventListenerKNX eventListener = new EventListenerKNX(varsDirectory,
                                                      eventQueue);
handlerKNX.setEventListener(eventListener);
```

Para detener el hilo de ejecución, habrá que desvincularlo del enlace mediante el método `removeEventListener` de la clase `HandlerKNX`.

```
public void removeEventListener() {
  link.removeLinkListener(eventListener);
}
```

```
handlerKNX.removeEventListener();
```

Cada vez que se produce un evento, se ejecuta el método `indication` del interfaz `NetworkLinkListener` de *Calimero*, por lo que en la case `EventListenerKNX` se ha sobrescrito dicho método para adaptarlo a nuestras necesidades.

```
@Override
public void indication(FrameEvent fe) {
  TDPU = ((CEMILData) fe.getFrame()).getPayload();
  GroupAdress = ((CEMILData)
                fe.getFrame()).getDestination().toString();
  if (varsDir.containsKey(GroupAdress)) {
    VariableXML var = varsDir.get(GroupAdress);
    DPTXlator varXlator;
    switch (var.getVarXlatorMainNumber()) {
      case 1://Boolean
        try {
          varXlator = new DPTXlatorBoolean(var.getTranslatorID());
          varXlator.setData(TDPU);
          String upperValue = varXlator.getType().getUpperValue();
          String eventValue = varXlator.getAllValues()[1];
          Object value = eventValue.equals(upperValue);
          EventKNX event = new EventKNX(GroupAdress, value);
          eventQueue.put(event);
          Logger.getLogger(EventListenerKNX.class.getName()).
                          log(Level.INFO, var.getName()+
                          " - "+GroupAdress+" - "+
                          eventValue+" ("+value+")");
        } catch (InterruptedException ex) {
            Logger.getLogger(EventListenerKNX.class.getName()).
                            log(Level.SEVERE, null, ex);
        }
        break;
      case 5://Unsigned
        try {
          varXlator = new DPTXlator8BitUnsigned(
                          var.getTranslatorID());
          varXlator.setAppendUnit(false);
          varXlator.setData(TDPU);
          String eventValue = varXlator.getAllValues()[2];
          Object value = Integer.parseInt(eventValue);
          EventKNX event = new EventKNX(GroupAdress, value);
          eventQueue.put(event);
          Logger.getLogger(EventListenerKNX.class.getName()).
                          log(Level.INFO, var.getName()+
                          " - "+GroupAdress+" - "+
                          eventValue+"%");
        } catch (KNXFormatException | InterruptedException ex) {
            Logger.getLogger(EventListenerKNX.class.getName()).
                            log(Level.SEVERE, null, ex);
        }
        break;
            .
            .
      default:
        break;
    }
  }
}
```

El método `indication` cuenta con un parámetro que contiene el telegrama asociado al evento. Dicho parámetro dispone de procedimientos que permiten, entre otras acciones, la obtención de la dirección de destino del telegrama y de su carga útil de datos.

A partir de la dirección, localizamos la variable relacionada con el evento y obtenemos su ***DPTXlator***. Este ***DPTXlator*** nos permitirá transformar la carga útil de datos, que está en forma de un vector de bytes, en una cadena de texto representando un valor. Con esta información y, en función de tipo de ***datapoint*** asociado a la variable, se compone el evento y se introduce en la cola.

Para implementar los eventos se ha creado la clase `EventKNX`. Cada evento va a constar de una dirección de grupo y de un valor. Para poder almacenar en el atributo valor cualquier tipo de datos (booleano, entero, coma flotante, etc.), se ha definido con el tipo genérico `Object` de Java.

```
public EventKNX(String groupAddres, Object value) {
  this.groupAddres = groupAddres;
  this.value = value;
}
```

5.11 Ejecución de las reglas

La ejecución de las reglas definidas por el usuario se lleva a cabo mediante las siguientes dos clases:

- `RulesNetExec`: que implementa el hilo de ejecución.
- `RulesNetMatrix`: la cual define las matrices binarias que representan la red de reglas. Además, proporciona los métodos para detectar si una regla está sensibilizada y para proceder a su disparo.

La clase `RulesNetExec` deriva de la clase `Thread` de Java. En ella se ha sobrescrito el método `run` para adecuarlo a las necesidades de la aplicación

```
@Override
public void run() {
  EventKNX event;
  VariableXML var;
  FactXML fact;
  while (keepRunning) {
    try {
      event = eventQueue.take();
      if (inputVarsDir.containsKey(event.getGroupAddres())){
        var = varsDir.get(event.getGroupAddres());
        switch (var.getVarXlatorMainNumber()) {
          case 1://is Boolean
            fact = var.getFacts().get(0);
            int factIndex = rnMatrix.getS().getFactIndex(fact);
            int newValue = (Boolean) event.getValue() ? 0 : 1;
            rnMatrix.getS().putFactIndexToValue(factIndex,newValue);
            fact = var.getFacts().get(1);
            factIndex = rnMatrix.getS().getFactIndex(fact);
            newValue = (Boolean) event.getValue() ? 1 : 0;
            rnMatrix.getS().putFactIndexToValue(factIndex,newValue);
            applyReasoning();
            break;
          case 5
            fact = var.getFacts().get(0);
            factIndex = rnMatrix.getS().getFactIndex(fact);
            newValue = (int) event.getValue()<10 ? 1 : 0;
            rnMatrix.getS().putFactIndexToValue(factIndex,
                                                newValue);
            fact = var.getFacts().get(1);
            factIndex = rnMatrix.getS().getFactIndex(fact);
            newValue = (int) event.getValue()>20 &&
                       (int) event.getValue()<30 ? 1 : 0;
            rnMatrix.getS().putFactIndexToValue(factIndex,
                                                newValue);
            fact = var.getFacts().get(2);
            factIndex = rnMatrix.getS().getFactIndex(fact);
            newValue = (int) event.getValue()>45 &&
                       (int) event.getValue()<55 ? 1 : 0;
            rnMatrix.getS().putFactIndexToValue(factIndex,
                                                newValue);
              .
```

```
          .
        fact = var.getFacts().get(3);
        factIndex = rnMatrix.getS().getFactIndex(fact);
        newValue = (int) event.getValue()>70 &&
                   (int) event.getValue()<80 ? 1 : 0;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        fact = var.getFacts().get(4);
        factIndex = rnMatrix.getS().getFactIndex(fact);
        newValue = (int) event.getValue()>90 ? 1 : 0;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        applyReasoning();
        break;
      case 9:
        fact = var.getFacts().get(0);
        factIndex = rnMatrix.getS().getFactIndex(fact);
        newValue = (float) event.getValue()<20.0 ? 1 : 0;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        fact = var.getFacts().get(1);
        factIndex = rnMatrix.getS().getFactIndex(fact);
        newValue = (float) event.getValue()>20.0 &&
                   (float) event.getValue()<28.0 ? 1 : 0;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        fact = var.getFacts().get(2);
        factIndex = rnMatrix.getS().getFactIndex(fact);
        newValue = (float) event.getValue()>28.0 ? 1 : 0;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        applyReasoning();
        break;
     default:
        break;
    }
  }
  if (outputVarsDir.containsKey(event.getGroupAddres())) {
    var = var = outputVarsDir.get(event.getGroupAddres());
    switch (var.getVarXlatorMainNumber()) {
      case 1:
        fact = var.getFacts().get(0);
        int factIndex = rnMatrix.getS().getFactIndex(fact);
        int newValue = (Boolean) event.getValue() ? 0 : 1;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        fact = var.getFacts().get(1);
        factIndex = rnMatrix.getS().getFactIndex(fact);
        newValue = (Boolean) event.getValue() ? 1 : 0;
        rnMatrix.getS().putFactIndexToValue(factIndex,
                                            newValue);
        applyReasoning();
        break;
          .
```

```
              .
       default:
         break;
       }
     }
     } catch (InterruptedException ex) {
         Logger.getLogger(RulesNetExec.class.getName()).
                         log(Level.SEVERE, null, ex);
     }
   }
}
```

El método `run` toma un evento de la cola, localiza la variable asociada al evento a partir de la dirección de grupo del evento y, en función del tipo de datos de la variable, actualiza convenientemente el vector de estado y llama al método `applyReasoning`. Este método recorre la lista de reglas y, para cada una de ellas, comprueba si está sensibilizada; en caso afirmativo, dispara la regla. A continuación, para los hechos que se hayan afirmado, crea la correspondiente orden KNX y la introduce en la cola de órdenes.

```
public void applyReasoning() {
  boolean changes;
  int numberOfChanges;
  int[] newVectorS;
  numberOfChanges = 0;
  if(!rulesNetLocked) {
    do {
      changes = false;
      for (int r = 0; r < rulesNet.numberOfRules(); r++) {
        if (rnMatrix.isRuleEnabled(r, rnMatrix.getS().
            getVectorNotS())) {
          Logger.getLogger(RulesNetExec.class.getName()).
                          log(Level.INFO,
                              "Se ha disparado la regla {0}",
                              rulesNet.getRules().get(r).getDesc());
          changes = true;
          numberOfChanges++;
          newVectorS = rnMatrix.fireEnabledRule(r, rnMatrix.getS().
                                                   getVectorS());
          for (int f = 0; f < newVectorS.length; f++) {
            if (newVectorS[f] != rnMatrix.getS().getVectorS()[f]) {
              if (newVectorS[f] == 1) {
                Logger.getLogger(RulesNetExec.class.getName()).
                                log(Level.INFO,
                                "Se ha afirmado el hecho {0}",
                                rnMatrix.getS().getFactByIndex(f).
                                getFullName());
            .
```

```
            .
                CommandKNX commandKNX = createCommandKNX(f);
                if (commandKNX != null) {
                  commandQueue.offer(commandKNX);
                  Logger.getLogger(RulesNetExec.class.getName()).
                                  log(Level.INFO,
                                  "new CommandKNX ({0},{1},{2})",
                                  new Object[]{commandKNX.
                                  getGroupAddress().toString(),
                                  commandKNX.getValue(),
                                  commandKNX.getType()});
                }
              }
            }
          }
          for (int f = 0; f < newVectorS.length; f++) {
            rnMatrix.getS().putFactIndexToValue(f, newVectorS[f]);
          }
          break;//Explore rules from the beggining
        }
      }
    } while (changes && numberOfChanges < reasoningMaxNumber);
    if (numberOfChanges == reasoningMaxNumber) {
      rulesNetLocked = true;
      Logger.getLogger(RulesNetMatrix.class.getName()).
                       log(Level.SEVERE,
                   "ERROR: La red de reglas genera oscilaciones");
      Logger.getLogger(RulesNetMatrix.class.getName()).
                       log(Level.WARNING,
                   "AVISO: La red de reglas ha sido desactivada");
    }
  }
}
```

Para detectar posibles oscilaciones, se ha incorporado el atributo `reasoningMaxNumber`, el cual corta el proceso de razonamiento en caso de que se produzcan más cambios que reglas. Cuando el número de cambios máximo es superado, se activa la variable booleana `rulesNetLocked` que inhabilita la red de reglas. Este mecanismo ha resultado efectivo en las pruebas realizadas.

Las órdenes se implementan mediante objetos de la clase `CommandKNX`. Cada orden va a constar de una dirección de grupo, de un valor y de un tipo. Para poder almacenar en el atributo valor cualquier tipo de datos (booleano, entero, coma flotante, etc.), se ha definido con el tipo genérico `Object` de Java. El atributo tipo se utiliza por parte del ejecutor de órdenes para distinguir el tipo de datos y escoger el método `write` adecuado del `ProcessComunicator`.

```
public CommandKNX(String groupAddress, Object value, int type) {
  try {
    this.groupAddress = new GroupAddress(groupAddress);
  } catch (KNXFormatException ex) {
      Logger.getLogger(CommandKNX.class.getName()).
                      log(Level.SEVERE, null, ex);
  }
  this.value = value;
  this.type = type;
}
```

Para crear el hilo de ejecución, en primer lugar se define un objeto de la clase `RulesNetExec`, pasándole como parámetros el objeto que contiene la red de reglas, la cola de eventos y la cola de órdenes que deberá ir rellenando. A continuación, se invocará al método `start` para poner en marcha el hilo. Tras arrancar el hilo, se llama también al método `applyReasoning` para detectar si el estado inicial de la instalación provoca el disparo de alguna regla. Para detener el hilo de ejecución, se invoca al método `ceaseRunning`.

```
  .
  .
execRN = new RulesNetExec(rulesNet, eventQueue, commandQueue);
execRN.setName("Ejecutor de Reglas");
execRN.start();
execRN.applyReasoning();
  .
  .
execRN.ceaseRunning();
```

En la clase `RulesNetMatrix` se definen y rellenan las matrices binarias de antecedentes `A`, consecuentes `C`, desmarcado `D` y el vector de estado `S`. En el caso de las tres matrices, cada fila representará una regla y cada columna un hecho.

Para el vector de estado `S` se ha definido la clase `VectorS` embebida en la clase `RulesNetMatrix`. El vector `S` va a estar compuesto a su vez por tres vectores:

- `factsVector`: Que contendrá los hechos.
- `S`: Que contendrá unos o ceros indicando que hechos están o no marcados en un momento dado.
- `notS`: Que será el complementario del vector `S` y se empleará en las operaciones para comprobar si una regla está o no sensibilizada.

Con el objeto de mantener la coherencia, el valor de estos tres vectores se modificará de manera conjunta mediante el método `putFactIndexToValue`.

```
public void putFactIndexToValue(int factIndex, int newValue){
  S[factIndex]=newValue;
  notS[factIndex]=(newValue==1)?0:1;
  factsVector[factIndex].setFactState((newValue==1));
}
```

Para detectar si una regla está sensibilizada, la clase `RulesNetMatrix` proporciona el método `isRuleEnabled`, el cual comprueba las condiciones vistas en el apartado 2.3.6.

```
private boolean isRuleEnabled(int IndexOfRuleToTest,
                              int[] vectorNotS) {
  boolean condition1 = true;
  boolean condition2 = true;
  int aux = 0;
  int[] subvectorA = extractSubVectorA(IndexOfRuleToTest);
  int[] subvectorC = extractSubVectorC(IndexOfRuleToTest);

  // condition 1: ( NotS and A = 0 )
  for (int f = 0; f < vectorNotS.length; f++) {
    aux = aux + vectorNotS[f] * subvectorA[f];
  }
  if (aux != 0) {
    condition1 = false;
  }

  // condition 2: ( NotS and C != 0 )
  for (int f = 0; f < vectorNotS.length; f++) {
    aux = aux + vectorNotS[f] * subvectorC[f];
  }
  if (aux == 0) {
    condition2 = false;
  }
  return (condition1 & condition2);
}
```

En cuanto al disparo de las reglas, se dispone del método `fireEnableRule`, el cual aplica lo expuesto en el apartado 2.3.7.

```
private int[] fireEnabledRule(int indexOfRuleToFire, int[] vectorS){
  int[] newVectorS = new int[vectorS.length];
  int[] subvectorC = extractSubVectorC(indexOfRuleToFire);
  int[] subvectorD = extractSubVectorD(indexOfRuleToFire);
  for (int f = 0; f < vectorS.length; f++) {
    newVectorS[f] = (vectorS[f] + subvectorC[f]) * subvectorD[f];
    if (newVectorS[f] == 2) {
      newVectorS[f] = 1;
    }
  }
  return newVectorS;
}
```

5.12 Ejecución de las órdenes KNX

El hilo encargado de ir ejecutando sobre KNX las órdenes generadas por el ejecutor de reglas, se implementa mediante una instancia de la clase `CommandExecKNX`. Esta clase deriva de la clase `Thread` de Java y en ella se ha sobrescrito el método `run` para adecuarlo a las necesidades de la aplicación.

```
@Override
public void run() {
  try {
    CommandKNX commandKNX;
    while (keepRunning) {
      commandKNX = commandQueue.take();
      switch (commandKNX.getType()) {
        case 1:
          handlerKNX.getProCom().write(commandKNX.getGroupAddress(),
                                    (Boolean) commandKNX.getValue());
          Logger.getLogger(CommandExecutorKNX.class.getName()).
                          log(Level.INFO, "{0} - {1} - BOOLEAN",
                          new   Object[]
                          {commandKNX.groupAddress.toString(),
                          commandKNX.value});
          break;
            .
            .
        case 9:
          handlerKNX.getProCom().write(commandKNX.getGroupAddress(),
                                      (Float) commandKNX.getValue());
          Logger.getLogger(CommandExecutorKNX.class.getName()).
                          log(Level.INFO, "{0} - {1} - FLOAT",
                          new   Object[]
                          {commandKNX.groupAddress.toString(),
                          commandKNX.value});
          break;

        default:
          break;
      }
    }
    Thread.sleep(1000);
  } catch (InterruptedException | KNXTimeoutException |
           KNXLinkClosedException | KNXFormatException ex) {
      Logger.getLogger(CommandExecutorKNX.class.getName()).
                      log(Level.SEVERE, null, ex);
  }
}
```

Al crear un objeto de la clase `CommandExecKNX`, se le pasa como parámetros la cola, de la que ir tomando las órdenes, y el objeto de la clase `HandlerKNX` ,con el que se haya establecido la conexión con KNX. A continuación, se deberá invocar el método `start` para poner el hilo en ejecución. Para detener el hilo, se invocará el método `ceaseRunning`.

```
execKNX = new CommandExecutorKNX(commandQueue, handlerKNX);
execKNX.setName("Ejecutor de órdenes KNX");
execKNX.start();
       .
       .
execKNX.ceaseRunning();
```

5.13 Despliegue de la aplicación WEB en la *BeagleBone Black*

En los siguientes apartados se describen los pasos seguidos para instalar la aplicación WEB implementada en el ordenador monoplaca *BleagleBone Black*.

5.13.1 Puesta en marcha

Para alimentar y acceder a la *BleagleBone*, solo se necesita conectarla a un puerto USB mediante el cable que se proporciona con ella. En función del sistema operativo que se esté utilizando habrá que realizar unos determinados pasos.

En Windows el dispositivo es detectado como una unidad de almacenamiento USB. Para poder establecer una conexión de terminal, es necesario instalar unos controladores que vienen almacenados en la carpeta *Drivers* → *Windows* del dispositivo. La última versión de estos controladores se puede descargar de *http://beagleboard.org/getting-started*. Habrá que instalar el controlador adecuado a la versión de Windows que se esté utilizando, ya sea de 32 bits o de 64 bits.

En Linux se debe crear en el directorio `/etc/udev/rules.d/` un fichero llamado `73-beaglebone.rules` con el siguiente contenido:

```
ACTION=="add", SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_interface",
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="a6d0", DRIVER=="",
RUN+="/sbin/modprobe -b ftdi_sio"

ACTION=="add", SUBSYSTEM=="drivers", ENV{DEVPATH}=="/bus/usb-
serial/drivers/ftdi_sio", ATTR{new_id}="0403 a6d0"

ACTION=="add", KERNEL=="ttyUSB*",
ATTRS{interface}=="BeagleBone", ATTRS{bInterfaceNumber}=="00",
SYMLINK+="beaglebone-jtag"

ACTION=="add", KERNEL=="ttyUSB*",
ATTRS{interface}=="BeagleBone", ATTRS{bInterfaceNumber}=="01",
SYMLINK+="beaglebone-serial"
```

Para que el sistema cargue el fichero, se debe ejecutar la siguiente orden:

```
udevadm control --reload-rules
```

Ahora, al introducir la orden `ifconfig`, aparecerá un nuevo interfaz de red:

```
eth1  Link encap:Ethernet  direcciónHW 6c:ec:eb:5c:44:54
      Direc. net:192.168.7.1  Difus.:192.168.7.3  Másc:255.255.255.252
      Dirección inet6: fe80::6eec:ebff:fe5c:4454/64 Alcance:Enlace
      ACTIVO DIFUSIÓN FUNCIONANDO MULTICAST  MTU:1500  Métrica:1
      Paquetes RX:38 errores:0 perdidos:0 overruns:0 frame:0
      Paquetes TX:41 errores:0 perdidos:0 overruns:0 carrier:0
      colisiones:0 long.colaTX:1000
      Bytes RX:7741 (7.7 KB)  TX bytes:9577 (9.5 KB)
```

La *BeagleBone* tiene activado por defecto el servicio *ssh*, por lo que se puede conectar con ella mediante la orden `ssh 192.168.7.2`. Una vez estemos conectados, estableceremos una contraseña para el usuario *root* mediante la orden `passwd`. Si se desea aumentar la seguridad, se cambiará el puerto de escucha del servicio *ssh* editando el fichero `/etc/ssh/sshd_config`.

Para posibilitar el acceso a través de la interfaz Ethernet, es necesario habilitar la interfaz de red etho del dispositivo. Para ello, se ha editado el fichero `/etc/network/interfaces`, eliminando el comentario de la línea `iface eth0 inet dhcp`.

5.13.2 Instalación de *Tomcat* versión 7

Se ha escogido como servidor WEB el paquete *Tomcat* fundamentalmente por su baja necesidad de recursos. Si bien no se trata de un servidor Java EE completo, con las funcionalidades de servicio *http* y contenedor de *servlets* que proporciona es más que suficiente para las necesidades del presente trabajo.

En primer lugar, para evitar que la instalación de *Tomcat* de error, se debe editar el guion de inicio `/etc/init.d/led_aging.sh` y añadirle la información relativa a las dependencias de ejecución:

```
#!/bin/sh
# tary, 16:46 2013422
### BEGIN INIT INFO
# Provides:              led_aging.sh
# RequiredStart:         $local_fs
# RequiredStop:          $local_fs
# DefaultStart:          2 3 4 5
# DefaultStop:           0 1 6
# ShortDescription:      Start LED aging
# Description:           Starts LED aging (whatever that is)
### END INIT INFO

x=$(/bin/ps -ef | /bin/grep "[l]ed_acc")
if [ ! -n "$x" -a -x /usr/bin/led_acc ]; then
   /usr/bin/led_acc &
fi
```

A continuación se describen los pasos a seguir para instalar el paquete *Tomcat*:

1. Instalar el paquete tomcat7 mediante las siguientes órdenes:

```
apt-get update
apt-get install tomcat7
```

2. Inhabilitar el servicio apache2 que viene instalado por defecto ejecutando:

```
update-rc.d apache2 disable
```

3. Habilitar el servicio tomcat7:

```
service tomcat7 enable
```

4. Reiniciar el sistema y confirmar el estado de los servicios mediante la orden:

```
service --status-all
```

5. Finalmente abrir un navegador WEB e introducir como URL *http://ip_beagle:8080* para verificar que el servicio de *Tomcat* funciona.

Si para mayor seguridad se quiere que *Tomcat* proporcione acceso a través de SSL, se deberán realizar los siguientes pasos:

- En primer lugar habrá que disponer de un certificado. En nuestro caso será suficiente con un certificado auto firmado que se creará mediante la siguiente orden:

```
keytool -genkey -alias tomcat
        -keyalg RSA -keystore /etc/tomcat7/keystore
```

  Se nos pedirá una primera contraseña que corresponde al almacenamiento de claves, una serie de datos relativos a nuestra identidad y finalmente una segunda clave asociada al certificado. Si se quiere que esta segunda clave coincida con la del almacenamiento de claves, se deberá pulsar la tecla ENTRAR.

- A continuación se debe abrir el fichero `/etc/tomcat7/server.xml`, eliminar los comentarios de la entrada asociada al conector SSL y añadir la información relativa a la ubicación del certificado y su contraseña:

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->

<Connector port="8443" protocol="HTTP/1.1"
           SSLEnabled="true" maxThreads="150"
           scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS"
           keystoreFile="/etc/tomcat7/keystore"
           keystorePass=".rulesNetKNX2015"/>
```

- Finalmente se deberá reiniciar el servicio tomcat7 para que los cambios tengan efecto.

Para verificar que todo funciona, bastará con abrir un navegador WEB e introducir la siguiente URL: *https://ip_beagle:8443*.

5.13.3 Instalación de jdk versión 1.8.33

En primer lugar, hay que descargar el fichero jdk-8u33-linux-arm-vfp-hflt.tar.gz de Oracle desde http://www.oracle.com/technetwork/Java/Javase/downloads/jdk8-arm-downloads-2187472.html y copiarlo a la *BeagleBone* mediante la siguiente orden:

```
scp jdk-8u33-linux-arm-vfp-hflt.tar.gz root@ip_beagle:/usr/lib/jvm/
```

A continuación, se debe descomprimir el fichero en `/usr/lib/jvm` mediante la siguiente orden:

```
tar zxvpf jdk8u33linuxarmvfphflt.tar.gz
```

El siguiente paso es cambiar el enlace al que apunta Java:

```
ln -sf /usr/lib/jvm jdk1.8.0_33/jre/bin/Java /etc/alternatives/Java
```

Seguidamente, deberemos comprobar que se ha configurado correctamente la versión de Java mediante la orden `Java -version`.

Finalmente, se debe editar el fichero `/etc/default/tomcat7` para que utilice la versión correcta de Java incluyendo la siguiente línea:

```
JAVA_HOME=/usr/lib/jvm/jdk1.8.0_33
```

5.13.4 Despliegue de la aplicación

Para simplificar la subida de aplicaciones al servidor durante el proceso de desarrollo, se ha instalado el *App Manager* de *Tomcat* mediante la siguiente orden:

```
apt-get install tomcat7admin
```

Una vez instalado el paquete, se deben definir un usuario y los roles *managergui* y *admingui* para poder acceder a esta funcionalidad desde un navegador WEB. Para ello, se ha editado el fichero `/etc/tomcat7/tomcatusers`.

```
xml:
<role rolename="managergui"/>
<role rolename="admingui"/>
<user username="admin" password=".shcMANger" roles="managergui,
admingui"/>
```

Para terminar, se debe reiniciar el servicio de tomcat7:

```
service tomcat7 restart
```

Para acceder al *APP Manager*, basta con introducir en un navegador la URL *http://ip_beagle:8080/manager*. Para poder acceder se nos requerirá el nombre de usuario y la contraseña definidos en el fichero `/etc/tomcat7/tomcatusers`.

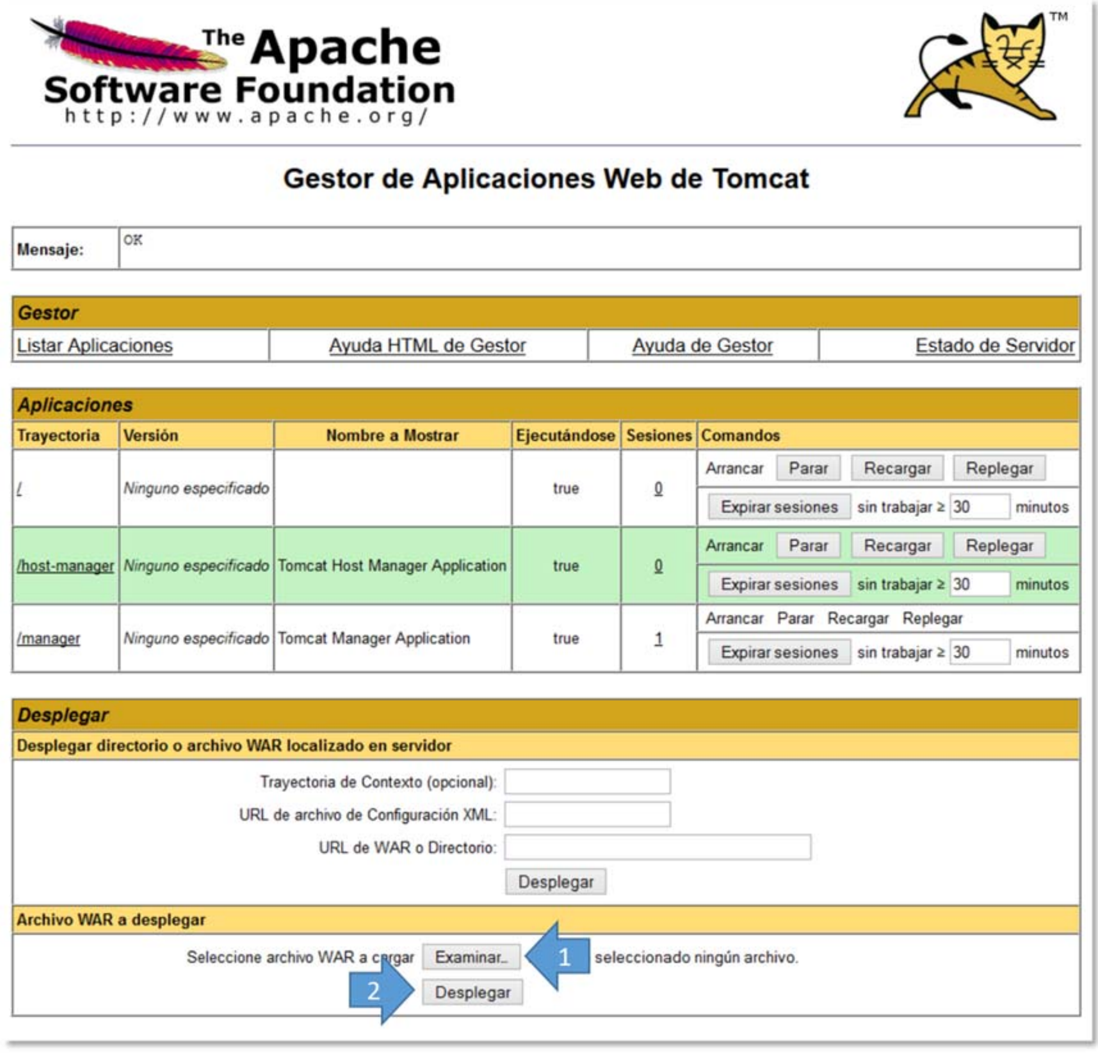


Figura 31: Gestor de aplicaciones WEB de Tomcat

Para cargar la aplicación deberemos hacer clic en el botón *Examinar*, que aparece en la parte inferior de la página, y seleccionar el archivo *.war* que contiene la aplicación. A continuación se deberá pulsar sobre el botón *Desplegar* para que comience la subida y posterior instalación de la aplicación en el servidor.

6 Pruebas de uso

6.1 Introducción

A continuación, se va a mostrar el funcionamiento del dispositivo implementado actuando sobre la instalación KNX de la que se dispone, la cual cuenta con los siguientes elementos:

- Una fuente de alimentación *ZPS160M* marca *Zennio*.
- Dos módulos *ACTinBOX Classic* marca *Zennio*.
- Una pasarela KNX IP modelo 730 de la marca *Weinzierl*.
- Una pantalla táctil modelo *InZennio Z38* de la marca *Zennio*.

Mediante estos elementos se da servicio a una vivienda compuesta por diversas estancias, cada una de las cuales dispone de una serie de elementos manejados por KNX:

- *Recibidor*: sensor de movimiento y sensor de puerta.
- *Habitación*: luz.
- *Comedor*: luz, persiana, sensor de movimiento y sensor de puerta.
- *Cocina*: luz.

Además, la vivienda cuenta como elementos generales con una alarma, un enchufe controlado por relé y un sensor que mide la temperatura en el interior.

6.2 Identificación KNX

Según lo expuesto en el apartado 2.2.5, para poder obtener información de los dispositivos o poder mandarles órdenes, es preciso conocer cuáles son sus objetos de comunicación y sus correspondientes direcciones de grupo. Así pues, el primer problema a resolver es obtener dicha información.

Esta información debería obtenerse utilizando un módulo de identificación KNX que forma parte de otro TFG. Desafortunadamente dicho TFG no ha sido completado, por lo que ha sido necesario buscar una forma alternativa de obtener la información necesaria. La solución escogida ha consistido en escribir un programa, haciendo uso de la librería *Calimero*, que muestre el contenido de los telegramas que se generan por cada evento. Para generar eventos, se ha ido actuando sobre los distintos elementos de la instalación. Este programa se ha estructurado en tres clases:

- **HandlerKNX**: Que contiene los métodos para establecer la conexión con KNX. El contenido de esta clase se detalla en el apartado 5.9.
- **EventListenerKNX**: Que implementa el interfaz `NetworkLinkListener` de *Calimero*. Esta clase se encargará de capturar los telegramas y mostrar su contenido, para lo cual se ha sobrescrito el método `indication` del interfaz.

```
public void indication(FrameEvent fe) {
  TDPU = ((CEMILData) fe.getFrame()).getPayload();
  GroupAdress = ((CEMILData)  fe.getFrame()).getDestination().
                                                      toString();
  System.out.println(GroupAdress);
  System.out.println(Arrays.toString(TDPU));
}
```

- **SnifferKNX**: Que contiene el método principal

```
package snifferknx;

  import java.io.IOException;

  public class SnifferKNX {
    static HandlerKNX handlerKNX;
    public static void main(String[] args) throws IOException{
    handlerKNX = new HandlerKNX();
    handlerKNX.openConection();
    if (handlerKNX.getConnected()) {
      EventListenerKNX eventListener;
      eventListener = new EventListenerKNX();
      handlerKNX.setEventListener(eventListener);
      System.out.println("Pulse \'intro\' para finalizar");
      System.in.read();
      handlerKNX.closeConnection();
    }
  }
}
```

Veamos por ejemplo el resultado de encender y apagar una de las luces, en concreto la de la habitación:

```
1/6/1
[0, -127]
1/6/1
[0, -127]
1/6/1
[0, -128]
1/6/1
[0, -128]
```

Se observa que la dirección de grupo asociada es la 1/6/1 y que por cada acción se generan dos mensajes con idéntico valor. El primero de ellos va a estar asociado al objeto de comunicación con el que se le dice al actuador que conecte su salida para encender la luz. El segundo mensaje corresponderá al objeto de comunicación que devuelve el estado de la luz. Al actuar sobre el pulsador que apaga la luz se generan

otros dos telegramas, aunque esta vez el valor del segundo byte es diferente. Así pues, se ha obtenido la dirección de grupo de los objetos de comunicación asociados al encendido y apagado de la luz, y al estado de la misma.

Probemos ahora a actuar sobre los pulsadores para la subida y bajada de la persiana del comedor mediante una pulsación larga:

```
5/1/1
[0, -128]
5/1/3
[0, -128, -8]
5/1/3
[0, -128, -25]
5/1/3
[0, -128, -42]
5/1/3
[0, -128, -59]
      .
      .
```

En este caso aparece un primer telegrama similar al de la luz, que será el asociado al objeto de comunicación para dar a la persiana la orden de subir. A continuación aparecen una serie de telegramas de 3 bytes, asociados a una dirección de grupo diferente y en cuyo último byte va modificándose el valor; dicho valor va a representar la posición en que se encuentra la persiana. Si ahora probamos con el pulsador de bajada, obtenemos algo similar:

```
5/1/1
[0, -127]
5/1/3
[0, -128, 3]
5/1/3
[0, -128, 20]
5/1/3
[0, -128, 37]
5/1/3
[0, -128, 54]
      .
      .
```

Para que esta información sea más legible, habría que buscar el *DPTXlator* adecuado a cada objeto de comunicación. Como primera aproximación, para el caso de la luz y demás elementos todo/nada, se podría utilizar el *DPTXlator* asociado al *datapoint* DPT_BOOL. El código de dicho *datapoint* según el estándar [1] es 1.002 y toma como valores *true* o *false*. A continuación, se muestra el resultado obtenido para la luz de la habitación:

```
1/6/1
[false, true]
1/6/1
[false, true]
1/6/1
[false, false]
```

```
1/6/1
[false, false]
```

Por tanto, para encender la luz hay que escribir *true* en la dirección de grupo y *false* para apagarla. Del mismo modo, si se lee *true* en su dirección de grupo, indicará que está encendida y si se lee *false*, indicará que está apagada.

En el caso de la persiana, el más adecuado sería el DPT_SCALING, que toma valores entre 0% y 100% y cuyo código es 5.001. A continuación, se muestra el resultado obtenido al actuar sobre el pulsador de subida de la persiana:

```
5/1/1
[false, false]
5/1/3
[0 %, 50 %, 96 %]
5/1/3
[0 %, 50 %, 90 %]
      .
      .
5/1/3
[0 %, 50 %, 4 %]
5/1/3
[0 %, 50 %, 0 %]
```

Como vemos, para subir la persiana se deberá escribir *false* en la dirección de grupo 5/1/1. Para hacer bajar la persiana, habrá por tanto que escribir *true*. En la dirección de grupo 5/1/3, asociada a la posición de la persiana, obtendremos 100% cuando esté completamente bajada y 0% cuando esté completamente subida.

Para detener el movimiento de la persiana, se realiza una pulsación corta sobre cualquiera de los dos botones (subir/bajar). Veamos a que objeto de comunicación está asociado dicha pulsación corta:

```
5/1/1
[false, true]
5/1/3
[0 %, 50 %, 2 %]
5/1/3
[0 %, 50 %, 9 %]
      .
      .
5/1/3
[0 %, 50 %, 29 %]
5/1/2
[false, false]
```

Se observa que se trata de la dirección de grupo 5/1/2. Si se escribe *true* o *false* en esta dirección, la persiana se detendrá.

El sensor de temperatura se ha localizado en la dirección de grupo 3/1/1, ofreciendo un telegrama compuesto por cuatro bytes:

```
3/1/1
[0, 64, 13, 55]
```

Para descodificar el valor de temperatura se debe utilizar un *DPTXlator* asociado al *datapoint* DPT_TEMPERATURE, que toma valores entre -273 to +670760 °C y cuyo código es 9.001. Vemos que en el segundo elemento del vector devuelto por el *DPTXLator* se obtiene el valor de temperatura medido por el sensor

```
3/1/1
[0, 64, 13, 55]
[0.64 °C, 26.7 °C]
```

En la siguiente tabla se detallan los elementos con los que se va a contar una vez obtenida la información relativa a las direcciones de grupo mediante el análisis del tráfico KNX:

Topoelemento	Domoelemento	Tipo	Datapoint	Dir. grupo
HABITACIÓN	LUZ	LIGHT_ON_OFF	CONTROL	1/6/1
			STATUS	1/6/1
COCINA	LUZ	LIGHT_ON_OFF	CONTROL	1/3/1
			STATUS	1/3/2
COMEDOR	LUZ	LIGHT_ON_OFF	CONTROL	1/2/1
			STATUS	1/2/1
	PUERTA	DOOR_SENSOR	STATUS	4/2/2
	SENSOR	MOTION_SENSOR	STATUS	4/2/1
	PERSIANA	BLIND	UP_DOWN	5/1/1
			STOP	5/1/2
			POSITION	5/1/3
RECIBIDOR	PUERTA	DOOR_SENSOR	STATUS	4/1/2
	SENSOR	MOTION_SENSOR	STATUS	4/1/1
MALETA	ENCHUFE	POWER_SOCKET	CONTROL	2/1/1
			STATUS	2/1/2
	ALARMA	ALARM	CONTROL	6/0/1
			STATUS	6/0/2
	TEMPERATURA	TEMP	VALUE	3/1/1

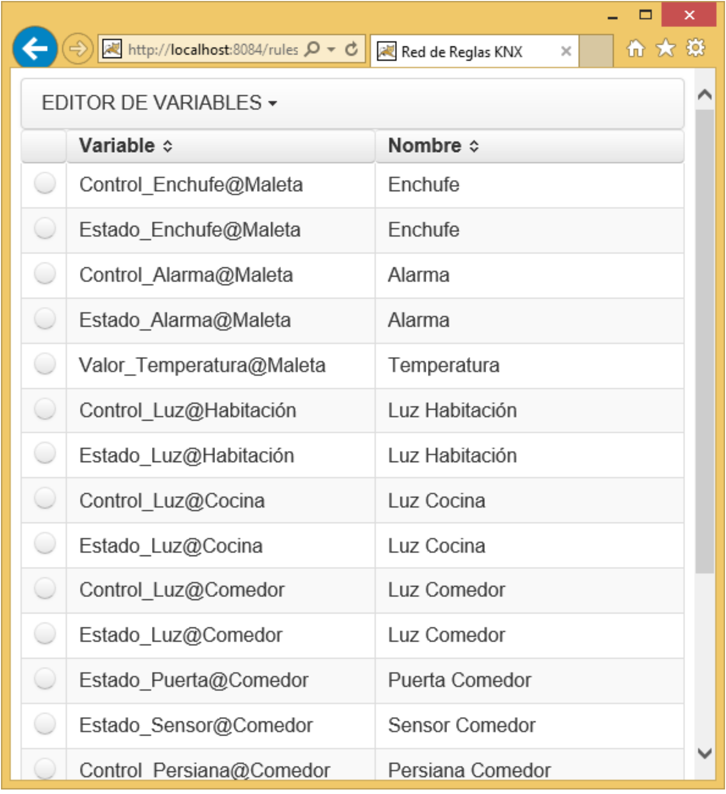
Tabla 5: Elementos de la instalación KNX para pruebas

6.3 Representación de la instalación en formato XML

Como entrada de datos del dispositivo se debe confeccionar un fichero XML que describa la instalación KNX. Según lo expuesto en el punto 4.1, la instalación de la que se dispone se representaría tal y como se muestra a continuación:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<topoElement name="Maleta">
  <domoElement name="Enchufe" type="POWER_SOCKET">
    <dataPoint type="CONTROL" groupAddress="2/1/1"/>
    <dataPoint type="STATUS" groupAddress="2/1/2"/>
  </domoElement>
  <domoElement name="Alarma" type="ALARM">
    <dataPoint type="CONTROL" groupAddress="6/0/1"/>
    <dataPoint type="STATUS" groupAddress="6/0/2"/>
  </domoElement>
  <domoElement name="Temperatura" type="TEMP">
    <dataPoint type="VALUE" groupAddress="3/1/1"/>
  </domoElement>
  <topoElement name="Habitación">
    <domoElement name="Luz" type="LIGHT_ON_OFF">
      <dataPoint type="CONTROL" groupAddress="1/6/1"/>
      <dataPoint type="STATUS" groupAddress="1/6/1"/>
    </domoElement>
  </topoElement>
  <topoElement name="Cocina">
    <domoElement name="Luz" type="LIGHT_ON_OFF">
      <dataPoint type="CONTROL" groupAddress="1/3/1"/>
      <dataPoint type="STATUS" groupAddress="1/3/2"/>
    </domoElement>
  </topoElement>
  <topoElement name="Recibidor">
    <domoElement name="Puerta" type="DOOR_SENSOR">
      <dataPoint type="STATUS" groupAddress="4/1/2"/>
    </domoElement>
    <domoElement name="Sensor" type="MOTION_SENSOR">
      <dataPoint type="STATUS" groupAddress="4/1/1"/>
    </domoElement>
  </topoElement>
  <topoElement name="Comedor">
    <domoElement name="Luz" type="LIGHT_ON_OFF">
      <dataPoint type="CONTROL" groupAddress="1/2/1"/>
      <dataPoint type="STATUS" groupAddress="1/2/1"/>
    </domoElement>
    <domoElement name="Puerta" type="DOOR_SENSOR">
      <dataPoint type="STATUS" groupAddress="4/2/2"/>
    </domoElement>
    <domoElement name="Sensor" type="MOTION_SENSOR">
      <dataPoint type="STATUS" groupAddress="4/2/1"/>
    </domoElement>
    <domoElement name="Persiana" type="BLIND">
      <dataPoint type="UP_DOWN" groupAddress="5/1/1"/>
      <dataPoint type="STOP" groupAddress="5/1/2"/>
      <dataPoint type="POSITION" groupAddress="5/1/3"/>
    </domoElement>
  </topoElement>
</topoElement>
```

A continuación, se muestra una captura de pantalla del editor de variables, en la que puede verse la lista de las variables generadas por el dispositivo a partir del fichero de instalación. En la columna de la derecha, aparecen los nombres de dichas variables tras ser modificados por el usuario.



6.4 Prueba de reglas

Seguidamente, se introducirán distintos juegos de reglas en el dispositivo y se probará su ejecución sobre la instalación KNX. La información sobre el proceso de ejecución se ha tomado del fichero de *log* de *Tomcat* mediante la siguiente orden:

```
tail -f /var/log/tomcat7/catalina.out | egrep -w 'INFO|SEVERE|WARNING'
```

6.4.1 Control de iluminación.

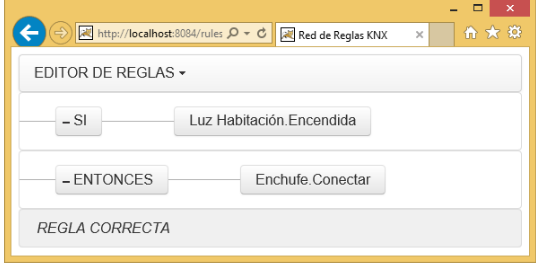
Se pretende que cuando la persiana esté al 75% de su recorrido (casi bajada del todo), se detenga y se enciendan las luces del comedor. Por otra parte, cuando esté al 25% de su recorrido (casi subida del todo), deberá detenerse y las luces del comedor se apagarán. Este comportamiento se va a definir mediante las siguientes reglas:

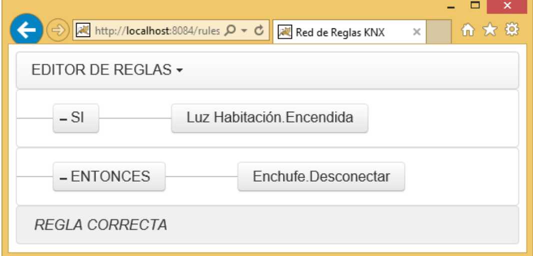
En la siguiente captura se muestra el *log* de ejecución para el juego de reglas propuesto:

```
1. root@158.42.53.155
INFO: ESTABLECIDA CONEXION KNX
INFO: ESTADO ACTUAL DE LA INSTALACION MALETA
INFO: Enchufe.Desconectado - true
INFO: Enchufe.Conectado - false
INFO: Alarma.Reposo - true
INFO: Alarma.Disparada - false
INFO: Temperatura.Frío - false
INFO: Temperatura.Confort - true
INFO: Temperatura.Calor - false
INFO: Luz Habitación.Apagada - true
INFO: Luz Habitación.Encendida - false
INFO: Luz Cocina.Apagada - true
INFO: Luz Cocina.Encendida - false
INFO: Luz Comedor.Apagada - true
INFO: Luz Comedor.Encendida - false
INFO: Puerta Comedor.Abierta - true
INFO: Puerta Comedor.Cerrada - false
INFO: Sensor Comedor.Inactivo - true
INFO: Sensor Comedor.Activo - false
INFO: Persiana Comedor.0% - false
INFO: Persiana Comedor.25% - false
INFO: Persiana Comedor.50% - false
INFO: Persiana Comedor.75% - false
INFO: Persiana Comedor.100% - false
INFO: Puerta Recibidor.Abierta - false
INFO: Puerta Recibidor.Cerrada - true
INFO: Sensor Recibidor.Inactivo - true
INFO: Sensor Recibidor.Activo - false
INFO: COMENZANDO EJECUCIÓN DE REGLAS
INFO: Temperatura - 3/1/1 - 26.5 ºC
INFO: Persiana Comedor - 5/1/1 - down (true)
INFO: Persiana Comedor - 5/1/3 - 44 %
INFO: Persiana Comedor - 5/1/3 - 50 %
INFO: Persiana Comedor - 5/1/3 - 56 %
INFO: Persiana Comedor - 5/1/3 - 63 %
INFO: Persiana Comedor - 5/1/3 - 70 %
INFO: Persiana Comedor - 5/1/3 - 76 %
INFO: Se ha disparado la regla SI (Persiana Comedor.75%) ENTONCES (Luz Comedor.Encender Y Persiana Comedor.DetenerBajar)
INFO: Se ha afirmado el hecho Luz Comedor.Encender
INFO: new CommandKNX (1/2/1,true,1)
INFO: Se ha afirmado el hecho Persiana Comedor.DetenerBajar
INFO: 1/2/1 - true - BOOLEAN
INFO: Luz Comedor - 1/2/1 - on (true)
INFO: new CommandKNX (5/1/2,true,1)
INFO: 5/1/2 - true - BOOLEAN
INFO: Persiana Comedor - 5/1/3 - 77 %
INFO: Persiana Comedor - 5/1/1 - up (false)
INFO: Persiana Comedor - 5/1/3 - 72 %
INFO: Persiana Comedor - 5/1/3 - 65 %
INFO: Persiana Comedor - 5/1/3 - 59 %
INFO: Persiana Comedor - 5/1/3 - 52 %
INFO: Persiana Comedor - 5/1/3 - 45 %
INFO: Persiana Comedor - 5/1/3 - 39 %
INFO: Persiana Comedor - 5/1/3 - 32 %
INFO: Se ha disparado la regla SI (Persiana Comedor.25%) ENTONCES (Luz Comedor.Apagar Y Persiana Comedor.DetenerSubir)
INFO: Persiana Comedor - 5/1/3 - 25 %
INFO: Se ha afirmado el hecho Luz Comedor.Apagar
INFO: new CommandKNX (1/2/1,false,1)
INFO: Se ha afirmado el hecho Persiana Comedor.DetenerSubir
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: 1/2/1 - false - BOOLEAN
INFO: new CommandKNX (5/1/2,false,1)
INFO: 5/1/2 - false - BOOLEAN
INFO: Persiana Comedor - 5/1/3 - 25 %
```

6.4.2 Oscilaciones

El objetivo del siguiente juego de reglas es mostrar el funcionamiento del mecanismo encargado de detectar y neutralizar las posibles oscilaciones generadas por la red de reglas. Para ello, se van a utilizar las siguientes reglas:





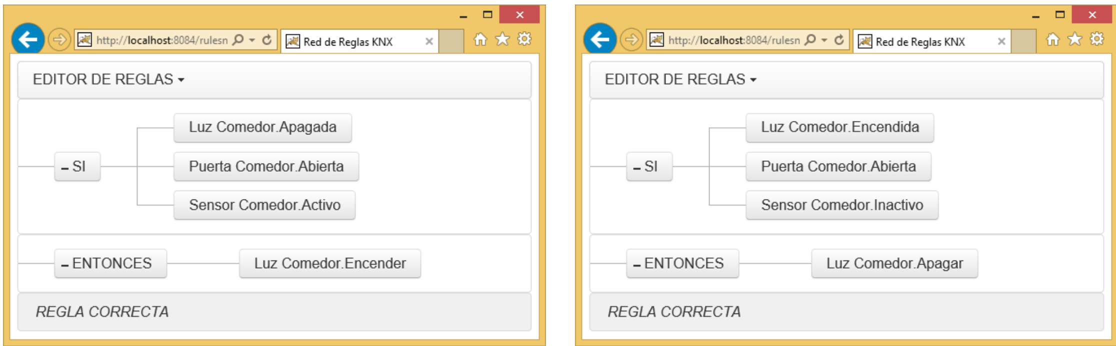
En la siguiente captura se muestra el *log* de ejecución para el juego de reglas propuesto:

```
1. root@158.42.53.155
INFO: ESTABLECIDA CONEXION KNX
INFO: ESTADO ACTUAL DE LA INSTALACION MALETA
INFO: Enchufe.Desconectado - false
INFO: Enchufe.Conectado - true
INFO: Alarma.Reposo - true
INFO: Alarma.Disparada - false
INFO: Temperatura.Frío - false
INFO: Temperatura.Confort - true
INFO: Temperatura.Calor - false
INFO: Luz Habitación.Apagada - true
INFO: Luz Habitación.Encendida - false
INFO: Luz Cocina.Apagada - true
INFO: Luz Cocina.Encendida - false
INFO: Luz Comedor.Apagada - true
INFO: Luz Comedor.Encendida - false
INFO: Puerta Comedor.Abierta - true
INFO: Puerta Comedor.Cerrada - false
INFO: Sensor Comedor.Inactivo - true
INFO: Sensor Comedor.Activo - false
INFO: Persiana Comedor.0% - false
INFO: Persiana Comedor.25% - true
INFO: Persiana Comedor.50% - false
INFO: Persiana Comedor.75% - false
INFO: Persiana Comedor.100% - false
INFO: Puerta Recibidor.Abierta - false
INFO: Puerta Recibidor.Cerrada - true
INFO: Sensor Recibidor.Inactivo - true
INFO: Sensor Recibidor.Activo - false
INFO: COMENZANDO EJECUCIÓN DE REGLAS
INFO: Temperatura - 3/1/1 - 26.5 ºC
INFO: Se ha disparado la regla SI (Luz Habitación.Encendida) ENTONCES (Enchufe.Desconectar)
INFO: Luz Habitación - 1/6/1 - on (true)
INFO: Se ha afirmado el hecho Enchufe.Desconectar
INFO: Luz Habitación - 1/6/1 - on (true)
INFO: new CommandKNX (2/1/1,false,1)
INFO: 2/1/1 - false - BOOLEAN
INFO: Se ha disparado la regla SI (Luz Habitación.Encendida) ENTONCES (Enchufe.Conectar)
INFO: Se ha afirmado el hecho Enchufe.Conectar
INFO: new CommandKNX (2/1/1,true,1)
INFO: 2/1/1 - true - BOOLEAN
INFO: Se ha disparado la regla SI (Luz Habitación.Encendida) ENTONCES (Enchufe.Desconectar)
INFO: Enchufe - 2/1/2 - off (false)
INFO: Se ha afirmado el hecho Enchufe.Desconectar
INFO: new CommandKNX (2/1/1,false,1)
SEVERE: La red de reglas genera oscilaciones
INFO: 2/1/1 - false - BOOLEAN
WARNING: La red de reglas ha sido desactivada
INFO: Enchufe - 2/1/2 - on (true)
INFO: Enchufe - 2/1/2 - off (false)
INFO: Luz Habitación - 1/6/1 - off (false)
INFO: Luz Habitación - 1/6/1 - off (false)
INFO: Luz Habitación - 1/6/1 - on (true)
INFO: Luz Habitación - 1/6/1 - on (true)
INFO: Luz Habitación - 1/6/1 - off (false)
INFO: Luz Habitación - 1/6/1 - off (false)
```

6.4.3 Múltiples antecedentes

Con el siguiente juego de reglas se pretende mostrar el funcionamiento con más de un antecedente. En este caso se va a establecer que la luz del comedor se encienda cuanto la puerta del mismo esté abierta y se active el sensor de movimiento. La luz se apagará cuando la puerta esté abierta y el sensor de movimiento esté inactivo. Además, solo se dará la orden de encender o apagar la luz si esta está apagada o encendida respectivamente.

A continuación se muestran las reglas definidas para establecer el comportamiento antes descrito:



En la siguiente captura se muestra el *log* de ejecución para el juego de reglas propuesto:

```
1. root@158.42.53.155
INFO: ESTABLECIDA CONEXION KNX
INFO: ESTADO ACTUAL DE LA INSTALACION MALETA
INFO: Enchufe.Desconectado - true
INFO: Enchufe.Conectado - false
INFO: Alarma.Reposo - true
INFO: Alarma.Disparada - false
INFO: Temperatura.Frío - false
INFO: Temperatura.Confort - true
INFO: Temperatura.Calor - false
INFO: Luz Habitación.Apagada - true
INFO: Luz Habitación.Encendida - false
INFO: Luz Cocina.Apagada - true
INFO: Luz Cocina.Encendida - false
INFO: Luz Comedor.Apagada - true
INFO: Luz Comedor.Encendida - false
INFO: Puerta Comedor.Abierta - false
INFO: Puerta Comedor.Cerrada - true
INFO: Sensor Comedor.Inactivo - true
INFO: Sensor Comedor.Activo - false
INFO: Persiana Comedor.0% - false
INFO: Persiana Comedor.25% - true
INFO: Persiana Comedor.50% - false
INFO: Persiana Comedor.75% - false
INFO: Persiana Comedor.100% - false
INFO: Puerta Recibidor.Abierta - false
INFO: Puerta Recibidor.Cerrada - true
INFO: Sensor Recibidor.Inactivo - true
INFO: Sensor Recibidor.Activo - false
INFO: COMENZANDO EJECUCIÓN DE REGLAS
INFO: Temperatura - 3/1/1 - 26.8 ºC
INFO: Sensor Comedor - 4/2/1 - occupied (true)
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: Sensor Comedor - 4/2/1 - not occupied (false)
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: Puerta Comedor - 4/2/2 - open (false)
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: Sensor Comedor - 4/2/1 - occupied (true)
INFO: Se ha disparado la regla SI (Luz Comedor.Apagada Y Puerta Comedor.Abierta Y Sensor Comedor.Activo) ENTONCES (Luz Comedor.Encender)
INFO: Se ha afirmado el hecho Luz Comedor.Encender
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: new CommandKNX (1/2/1,true,1)
INFO: Se ha disparado la regla SI (Luz Comedor.Apagada Y Puerta Comedor.Abierta Y Sensor Comedor.Activo) ENTONCES (Luz Comedor.Encender)
INFO: 1/2/1 - true - BOOLEAN
INFO: Luz Comedor - 1/2/1 - on (true)
INFO: Se ha afirmado el hecho Luz Comedor.Encender
INFO: new CommandKNX (1/2/1,true,1)
INFO: 1/2/1 - true - BOOLEAN
INFO: Se ha disparado la regla SI (Luz Comedor.Encendida Y Puerta Comedor.Abierta Y Sensor Comedor.Inactivo) ENTONCES (Luz Comedor.Apagar)
INFO: Sensor Comedor - 4/2/1 - not occupied (false)
INFO: Se ha afirmado el hecho Luz Comedor.Apagar
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: Luz Comedor - 1/2/1 - off (false)
INFO: 1/2/1 - false - BOOLEAN
INFO: new CommandKNX (1/2/1,false,1)
```

6.5 Representación de la red de reglas en XML

A continuación, se muestra un ejemplo del fichero de salida generado por el dispositivo de acuerdo a lo expuesto en el apartado 0:


```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <RulesNet installation="Maleta">
  - <variables>
    - <variable type="OUTPUT" translatorID="1.001" tag="Control_Enchufe@Maleta" name="Enchufe" groupAddress="2/1/1">
        <fact name="Desconectar" varTag="Control_Enchufe@Maleta" varName="Enchufe"/>
        <fact name="Conectar" varTag="Control_Enchufe@Maleta" varName="Enchufe"/>
      </variable>
    - <variable type="INPUT" translatorID="1.001" tag="Estado_Enchufe@Maleta" name="Enchufe" groupAddress="2/1/2">
        <fact name="Desconectado" varTag="Estado_Enchufe@Maleta" varName="Enchufe"/>
        <fact name="Conectado" varTag="Estado_Enchufe@Maleta" varName="Enchufe"/>
      </variable>
    - <variable type="OUTPUT" translatorID="1.003" tag="Control_Alarma@Maleta" name="Alarma" groupAddress="6/0/1">
        <fact name="Desarmar" varTag="Control_Alarma@Maleta" varName="Alarma"/>
        <fact name="Armar" varTag="Control_Alarma@Maleta" varName="Alarma"/>
      </variable>
    - <variable type="INPUT" translatorID="1.005" tag="Estado_Alarma@Maleta" name="Alarma" groupAddress="6/0/2">
        <fact name="Reposo" varTag="Estado_Alarma@Maleta" varName="Alarma"/>
        <fact name="Disparada" varTag="Estado_Alarma@Maleta" varName="Alarma"/>
      </variable>
    - <variable type="INPUT" translatorID="9.001" tag="Valor_Temperatura@Maleta" name="Temperatura" groupAddress="3/1/1">
        <fact name="Frío" varTag="Valor_Temperatura@Maleta" varName="Temperatura"/>
        <fact name="Confort" varTag="Valor_Temperatura@Maleta" varName="Temperatura"/>
        <fact name="Calor" varTag="Valor_Temperatura@Maleta" varName="Temperatura"/>
      </variable>
    - <variable type="OUTPUT" translatorID="1.001" tag="Control_Luz@Habitación" name="Luz Habitación" groupAddress="1/6/1">
        <fact name="Apagar" varTag="Control_Luz@Habitación" varName="Luz Habitación"/>
        <fact name="Encender" varTag="Control_Luz@Habitación" varName="Luz Habitación"/>
      </variable>
    - <variable type="INPUT" translatorID="1.001" tag="Estado_Luz@Habitación" name="Luz Habitación" groupAddress="1/6/1">
        <fact name="Apagada" varTag="Estado_Luz@Habitación" varName="Luz Habitación"/>
        <fact name="Encendida" varTag="Estado_Luz@Habitación" varName="Luz Habitación"/>
      </variable>
    + <variable type="OUTPUT" translatorID="1.001" tag="Control_Luz@Cocina" name="Luz Cocina" groupAddress="1/3/1">
    + <variable type="INPUT" translatorID="1.001" tag="Estado_Luz@Cocina" name="Luz Cocina" groupAddress="1/3/2">
    + <variable type="OUTPUT" translatorID="1.001" tag="Control_Luz@Comedor" name="Luz Comedor" groupAddress="1/2/1">
    + <variable type="INPUT" translatorID="1.001" tag="Estado_Luz@Comedor" name="Luz Comedor" groupAddress="1/2/1">
    - <variable type="INPUT" translatorID="1.009" tag="Estado_Puerta@Comedor" name="Puerta Comedor" groupAddress="4/2/2">
        <fact name="Abierta" varTag="Estado_Puerta@Comedor" varName="Puerta Comedor"/>
        <fact name="Cerrada" varTag="Estado_Puerta@Comedor" varName="Puerta Comedor"/>
      </variable>
    - <variable type="INPUT" translatorID="1.018" tag="Estado_Sensor@Comedor" name="Sensor Comedor" groupAddress="4/2/1">
        <fact name="Inactivo" varTag="Estado_Sensor@Comedor" varName="Sensor Comedor"/>
        <fact name="Activo" varTag="Estado_Sensor@Comedor" varName="Sensor Comedor"/>
      </variable>
    - <variable type="OUTPUT" translatorID="1.008" tag="Control_Persiana@Comedor" name="Persiana Comedor" groupAddress="5/1/1">
        <fact name="Subir" varTag="Control_Persiana@Comedor" varName="Persiana Comedor"/>
        <fact name="Bajar" varTag="Control_Persiana@Comedor" varName="Persiana Comedor"/>
      </variable>
    - <variable type="OUTPUT" translatorID="1.017" tag="Detener_Persiana@Comedor" name="Persiana Comedor" groupAddress="5/1/2">
        <fact name="DetenerSubir" varTag="Detener_Persiana@Comedor" varName="Persiana Comedor"/>
        <fact name="DetenerBajar" varTag="Detener_Persiana@Comedor" varName="Persiana Comedor"/>
      </variable>
    - <variable type="INPUT" translatorID="5.001" tag="Posicion_Persiana@Comedor" name="Persiana Comedor" groupAddress="5/1/3">
        <fact name="0%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
        <fact name="25%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
        <fact name="50%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
        <fact name="75%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
        <fact name="100%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
      </variable>
    + <variable type="INPUT" translatorID="1.009" tag="Estado_Puerta@Recibidor" name="Puerta Recibidor" groupAddress="4/1/2">
    + <variable type="INPUT" translatorID="1.018" tag="Estado_Sensor@Recibidor" name="Sensor Recibidor" groupAddress="4/1/1">
    </variables>
  - <rules>
    - <rule name="R0001" desc="SI (Persiana Comedor.75%) ENTONCES (Luz Comedor.Encender Y Persiana Comedor.DetenerBajar)">
        <antecedent name="75%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
        <consequent name="Encender" varTag="Control_Luz@Comedor" varName="Luz Comedor"/>
        <consequent name="DetenerBajar" varTag="Detener_Persiana@Comedor" varName="Persiana Comedor"/>
      </rule>
    - <rule name="R0002" desc="SI (Persiana Comedor.25%) ENTONCES (Luz Comedor.Apagar Y Persiana Comedor.DetenerSubir)">
        <antecedent name="25%" varTag="Posicion_Persiana@Comedor" varName="Persiana Comedor"/>
        <consequent name="Apagar" varTag="Control_Luz@Comedor" varName="Luz Comedor"/>
        <consequent name="DetenerSubir" varTag="Detener_Persiana@Comedor" varName="Persiana Comedor"/>
      </rule>
    </rules>
</RulesNet>
```

7 Conclusiones y futuras ampliaciones

En el presente trabajo se ha implementado un dispositivo que permite, al usuario de una instalación domótica basada en KNX, definir un comportamiento automático para su hogar mediante el uso de reglas del estilo *SI condición ENTONCES acción*. Dicho comportamiento automático complementa (no invalida) el funcionamiento programado en KNX.

La programación del comportamiento deseado se realiza mediante Redes de Reglas, ofreciéndose una aplicación web que permite al usuario establecer reglas de comportamiento entre elementos de la red KNX, expresados en forma de variables y hechos.

Para llevar a cabo la ejecución de las reglas, el dispositivo interactúa con la instalación KNX, monitorizando los mensajes que generan los elementos domóticos para conocer el estado (entradas de la Red de Reglas) y transmitiendo las órdenes necesarias (salidas de la Red de Reglas). La ejecución de la Red de Reglas puede realizarse bien de forma autónoma o bien con la supervisión del usuario.

Además de la aplicación, en este documento se incluyen guías paso a paso para la implantación del sistema mediante dispositivos *hardware* comerciales de bajo coste (*BeagleBone Black*).

Como futura ampliación se contempla la incorporación de un mecanismo para definir temporizadores, de manera que pudiera incorporarse el tiempo – periodos de actividad y sueño, hora del día, cuenta atrás, etc. – en las reglas.

Utilizar como soporte hardware el ordenador monoplaca *BeagleBone Black* se ha mostrado como una buena opción en cuanto a rendimiento, siendo capaz de ejecutar de manera fluida la aplicación WEB desarrollada. Además, tanto el consumo como el coste de este dispositivo son reducidos. Como único inconveniente, resulta necesario en esta versión una pasarela KNX/IP adicional. En futuras versiones se estudiará aprovechar las posibilidades de expansión de la *BeagleBone* para implementar con ella la pasarela KNX/IP directamente.

La utilización de la librería de componentes *Primefaces* de JSF, ha posibilitado la implementación de una interfaz de usuario que permite la creación de reglas de manera sencilla. La concepción como aplicación WEB, proporciona el acceso desde cualquier dispositivo que cuente con un navegador, no habiéndose detectado problemas de compatibilidad con ningún navegador. La mayor desventaja del uso de *PrimeFaces* ha sido la curva de aprendizaje y sobre todo la detección de errores durante la etapa de

desarrollo, para lo cual ha resultado de gran ayuda disponer del complemento *Firebug* para el navegador Firefox.

El editor de reglas desarrollado funciona de forma adecuada, si bien en futuras versiones se incorporará el análisis de las propiedades de la RdR más allá de la detección de reglas defectuosas.

Aunque en el diseño de la interfaz se ha buscado la sencillez, sería interesante explorar en el futuro los componentes para dispositivos móviles que ofrece la librería *PrimeFaces*, proporcionando al usuario una interfaz en función del tipo de dispositivo que utilice para conectarse.

En el trabajo se han sentado las bases para la descripción en formato XML de cualquier instalación KNX, habiéndose definido los dispositivos básicos más utilizados. Quedaría como futuro trabajo la definición de una biblioteca más amplia de dispositivos.

Todas estas futuras ampliaciones serán más sencillas gracias a la inclusión de una guía del programador detallada, disponible en el Anexo I. Finalmente, los usuarios del sistema disponen en el Anexo II de una guía del usuario.

Bibliografía

[1] KNX Association, "Vol. 3, Part 7, Ch. 2, System Specifications: Interworking: Datapoint Types," in *KNX Handbook*, KNX Association.

[2] «Calimero 2.0 API documentation,» [En línea]. Disponible en: http://calimero.sourceforge.net/docs204/calimero-2.0.3_doc/index.html. [Último acceso: Junio 2015].

[3] The World Wide Web Consortium (W3C), «Extensible Markup Language (XML) 1.0 (Third Edition),» Febrero 2004. [En línea]. Disponible en: http://www.w3.org/TR/2004/REC-xml-20040204/. [Último acceso: Julio 2015].

[4] B. Malinowsky, G. Neugschwandtner y W. Kastner, «Calimero: Next Generation,» 2007. [En línea]. Disponible en: http://calimero.sourceforge.net/calimero-ng.pdf. [Último acceso: Junio 2015].

[5] A. B. Pina, "Diseño de Automatismos basados en redes de reglas", Trabajo fin de carrera, Universidad Politécnica de Valencia, Valencia, 1992.

[6] A. Tomar, «System Reference Manual for element14 BeagleBone Black Revision C Development Platform,» 24 Abril 2013. [En línea]. Disponible en: http://www.element14.com/community/docs/DOC-54165/l/system-reference-manual-for-element14-beaglebone-black-revision-c-development-platform?forceNoRedirect=true. [Último acceso: Julio 2015].

[7] «JAXB Release Documentation,» 14 Octubre 2014. [En línea]. Disponible en: https://jaxb.Java.net/nonav/2.2.11/docs/release-documentation.pdf. [Último acceso: Julio 2015].

[8] Ç. Çivici, «Primefaces User Guide 5.2,» 2015. [En línea]. Disponible en: http://www.primefaces.org/docs/guide/primefaces_user_guide_5_2.pdf. [Último acceso: Junio 2015].

[9] O. Varaksin y M. Çalişkan, PrimeFaces Cookbook, Birmingham: Packt Publishing Ltd., 2013.

[10] Apache Software Foundation, «Apache Tomcat 7 Documentation,» 7 Mayo 2015. [En línea]. Disponible en: https://tomcat.apache.org/tomcat-7.0-doc/. [Último acceso: Julio 2015].

ANEXO I: Manual del programador

Introducción

En el presente anexo se van a tratar temas relacionados con la implementación de la aplicación WEB que no se desarrollaron en el capítulo 5.

Ficheros de configuración

En este apartado se va a hablar acerca de los ficheros de configuración `context.xml` y `web.xml`. En el primero de ellos se fija la ruta en la que se ofrecerá la aplicación en el servidor WEB.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/rulesNetKNX"/>
```

En cuanto al fichero `web.xml`, contiene diversos parámetros de contexto, algunos de los cuales ya se han visto en el apartado 5.3. Cabe destacar el primero de los parámetros, denominado `Javax.faces.PROJECT_STAGE`. Durante la etapa de desarrollo debe fijarse el valor de este parámetro en `Development`, de este modo se obtiene una información de errores más completa, aunque a costa de una merma en el rendimiento. Una vez se tenga la aplicación en producción, se podrá cambiar el valor de este parámetro a `Production`, desactivándose ciertos mensajes de error y mejorándose el rendimiento.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <context-param>
    <param-name>primefaces.FONT_AWESOME</param-name>
    <param-value>true</param-value>
  </context-param>
  <context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>bootstrap</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
      .
```

```
      .
    <servlet-mapping>
     <servlet-name>Faces Servlet</servlet-name>
     <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>
</web-app>
```

Implementación de la interfaz de usuario

En este punto se detallan los controles de la librería *PrimeFaces* utilizados para la implementación de la interfaz de usuario, así como el contenido de los distintos *managed bean* empleados.

Mensajes emergentes

Para mostrar mensajes emergentes al usuario, se ha utilizado el control `<p:growl>`. En su declaración se ha especificado el tiempo de vida del mensaje en milisegundos, que se muestre el campo detalle del mensaje y que se actualice automáticamente su contenido. De este modo, cada vez que se crea un mensaje, el contenido del control se actualiza y se muestra por pantalla.

```
<p:growl id="growl"
         life="3500"
         showDetail="true"
         autoUpdate="true"/>
```

A continuación puede verse el código que debe incluirse en los *managed bean* para crear un mensaje:

```
FacesContext context = FacesContext.getCurrentInstance();
context.addMessage(null,
                   new FacesMessage(FacesMessage.SEVERITY_WARN,
                   "AVISO",
                   "Si continua,
                    se perderá el contenido de la red de reglas"));
```

El primer campo del constructor del mensaje indica el tipo de mensaje, el cual puede ser de *error*, *advertencia* o *información*. A continuación, se proporciona una cadena de texto con el resumen del mensaje y finalmente otra cadena de texto con el detalle del mensaje.

Pantalla principal

Los elementos que componen la pantalla principal, incluyendo los cuadros de diálogo, se han agrupado mediante la etiqueta `<h:panelGroup>` con el identificador `id="rnknxMain"`. El ***managed bean*** que contiene los atributos y manejadores de eventos para esta sección es `MainBean.Java`.

```
<h:panelGroup id="rnknxMain">
  <p:menubar  id="mainMenu"
              rendered="#{mainBean.showMain}"
              toggleEvent="click">
      .
      .
  </p:menubar>
  <p:dataTable id="rulesList"
             value="#{mainBean.rulesNet.rules}"
             var="rule"
             rowKey="#{rule.name}"
             selection="#{mainBean.selectedRule}"
             emptyMessage="No existen reglas definidas"
             rendered="#{mainBean.showMain}">
      .
      .
  </p:dataTable>
  <p:dialog header="Cargar instalación"
            widgetVar="installationLoaderWidget"
            id="installationLoader"
            modal="true"
            resizable="false">
      .
      .
  </p:dialog>
  <p:dialog header="Cargar reglas"
            widgetVar="ruleLoaderWidget"
            id="ruleLoader"
            modal="true"
            resizable="false">
      .
      .
  </p:dialog>
  <p:dialog header="Ejecutar reglas"
            widgetVar="runRNWidget"
            id="runRN"
            modal="true"
            resizable="false"
      .
      .
  </p:dialog>
</h:panelGroup>
```

La barra de menú se ha implementado mediante el control `<p:menubar>`. Esta barra solo será mostrada cuando el atributo `showMain` del ***managed bean*** sea cierto. Se ha establecido que su contenido se despliegue al hacer clic sobre ella mediante el atributo `toggleEvent`.

```
<p:menubar  id="mainMenu"
            rendered="#{mainBean.showMain}"
            toggleEvent="click">
  <p:submenu label="RED DE REGLAS" style="font-weight: bold">
    <p:menuitem value="Cargar Instalación"
                onclick="PF('installationLoaderWidget').show()"
                actionListener="#{mainBean.overWriteWarning()}"
                icon="fa fa-home"/>
    <p:separator />
    <p:menuitem value="Editar variables"
                actionListener="#{mainBean.editVars()}"
                update="varList,rnknxMain"
                icon="fa fa-pencil"
                disabled="#{!mainBean.fileLoaded}"/>
    <p:separator />
    <p:menuitem value="Nueva regla"
                actionListener="#{mainBean.callNewRule}"
                update="ruleEditor,rnknxMain"
                icon="fa fa-plus"
                disabled="#{!mainBean.fileLoaded}"/>
    <p:menuitem value="Cargar reglas"
                onclick="PF('ruleLoaderWidget').show()"
                actionListener="#{mainBean.overWriteWarning()}"
                icon="fa fa-upload"/>
    <p:menuitem value="Ejecutar reglas"
                onclick="PF('runRNWidget').show()"
                icon="fa fa-play"
                disabled="#{mainBean.rulesNet.rules.isEmpty()}"/>
    <p:menuitem value="Guardar reglas"
                ajax="false"
                icon="fa fa-download"
                disabled="#{mainBean.rulesNet.rules.isEmpty()}">
      <p:fileDownload value="#{mainBean.fileDownload}"/>
    </p:menuitem>
  </p:submenu>
</p:menubar>
```

La barra de menú incluye un único submenú definido por la etiqueta `<p:submenú>` llamado *RED DE REGLAS*. Las distintas opciones contenidas en dicho submenú se implementan mediante etiquetas `<p:menuitem>`. El nombre de cada una de las opciones se determina mediante el atributo `value`.

Las opciones *Nueva regla*, *Editar variables* y *Guardar reglas* están inicialmente inhabilitadas hasta que el usuario carga un fichero de instalación o de reglas. La opción *Ejecutar reglas* estará inhabilitada mientras no existan reglas definidas.

Cuando se produce el evento `onclick`, las opciones *Cargar instalación*, *Cargar reglas* y *Ejecutar reglas*, llaman al método `show` de los correspondientes cuadros de diálogo. Los cuadros de diálogo son referenciados con el valor de su atributo `widgetVar`, el cual representa su identificador en la parte del cliente. La llamada se puede hacer de dos formas:

- Mediante el acceso directo PF :

```
PF('nombreWidget').show()
```

- Mediante el espacio de nombres *widgets* de *PrimeFaces*:

```
Primefaces.widgets['nombreWidget'].show()
```

La opción ***Nueva regla*** llama al método `callNewRule` del ***managed bean*** que hace cierto el atributo `showEditor` y falso el atributo `showMain`. Una vez completado el método, se actualizan los controles indicados en el atributo `update` con lo que se mostrará la pantalla del editor y se ocultará la principal. Finalmente, la opción encargada de guardar las reglas contiene a su vez al control `<p:fileDownload>` por lo que se ha desactivado el *AJAX* para dicha opción.

Para la lista de reglas se ha empleado el control `<p:dataTable>`. Este control se rellena con los elementos del vector de reglas contenido en el objeto `rulesNet` que hay definido en el ***managed bean***. Se ha establecido como campo clave el atributo `name` de las reglas. El usuario podrá ordenar la lista tanto por el atributo del nombre como por el de la descripción.

```
<p:dataTable id="rulesList"
             value="#{mainBean.rulesNet.rules}"
             var="rule"
             rowKey="#{rule.name}"
             selection="#{mainBean.selectedRule}"
             emptyMessage="No existen reglas definidas"
             rendered="#{mainBean.showMain}">
  <p:ajax event="rowSelectRadio"
          listener="#{mainBean.onRowSelectRadio}"
          update="editRule,delRule"/>
  <p:column selectionMode="single"
            style="width:16px;text-align:center"/>
  <p:column headerText="Nombre"
            style="width: 150px;text-align: left"
            sortBy="#{rule.name}">
    <h:outputText value="#{rule.name}"/>
  </p:column>
  <p:column headerText="Descripción"
            style="text-align: left"
            sortBy="#{rule.desc}">
    <h:outputText value="#{rule.desc}"/>
  </p:column>
      .
```

```
      .
  <f:facet name="footer">
    <h:panelGroup style="display:block; text-align:center">
      <p:commandButton actionListener="#{mainBean.callEditRule()}"
                       id="editRule" title="Editar regla"
                       update=":rnknxForm:ruleEditor,
                               :rnknxForm:rnknxMain"
                       icon="fa fa-edit"
                       disabled="#{mainBean.noneSelected}"/>
      <p:commandButton actionListener="#{mainBean.deleteRule()}"
                       id="delRule" title="Borrar regla"
                       update=":rnknxForm:rulesList,
                               :rnknxForm:mainMenu"
                       icon="fa fa-trash"
                       disabled="#{mainBean.noneSelected}"/>
    </h:panelGroup>
  </f:facet>
</p:dataTable>
```

Al igual que en el caso del menú, este control solo será mostrado cuando el atributo `showMain` del *managed bean* sea cierto. Cuando la lista no contiene elementos, muestra un mensaje genérico, por lo que se ha definido uno más específico para la aplicación mediante el atributo `emptyMessage`.

La lista consta de tres columnas definidas mediante la etiqueta `<p:column>`. La primera de ellas contiene un botón de opción por cada fila para que el usuario seleccione la regla que desee eliminar o editar. Se ha fijado que solo pueda seleccionarse una fila cada vez mediante el atributo `selectionMode`. La regla que seleccione el usuario en cada momento será almacenada en el objeto `selectedRule` del *managed bean*.

Las otras dos columnas son las que muestran el nombre y la descripción de cada una de las reglas. El título de la cabecera se establece mediante el atributo `headerText` y el contenido de cada fila mediante el atributo `value` de la etiqueta `<h:outputText>`. Como puede verse, lo que se hace es acceder al campo `name` y `desc` de la variable `rule` definida en el atributo `var` de la etiqueta `<p:dataTable>`.

Los botones para editar y borrar una regla de la lista se han creado mediante el control `<p:commandButton>`. Ambos botones muestran únicamente un icono indicativo de su función, pero si el usuario pasa el puntero por encima de ellos, aparece un mensaje de texto definido mediante el atributo `title`. Dentro del control `<p:dataTable>` se ha definido el evento AJAX `rowSelectRadio` para que los botones de editar y borrar solo estén activos si hay una regla seleccionada en la lista.

En el caso del botón para editar, este llama al método `callEditRule` del *managed bean* que muestra la pantalla del editor cargando los datos de la regla seleccionada. Tal como se indicó previamente, dicha regla es almacenada en `selectedRule` en el momento en que el usuario selecciona una fila de la lista.

El botón para borrar invoca al método `deleteRule` del ***managed bean***, que elimina la regla seleccionada del vector de reglas contenido en el objeto `rulesNet`.

Los cuadros de diálogo para la carga de un fichero de instalación o de reglas se han implementado haciendo uso del control `<p:dialog>`.

```
<p:dialog header="Cargar instalación"
          widgetVar="installationLoaderWidget"
          id="installationLoader"
          modal="true"
          resizable="false">
  <p:panelGrid columns="1"
               styleClass="ui-noborder"
               style="display:block; text-align:center">
    <p:fileUpload styleClass="ui-noborder"
                  allowTypes="/(\.|\/)(xml)$/"
                  invalidFileMessage="Tipo de archivo no válido"
                  auto="false"
                  update="rnknxMain,ruleEditor"
                  label="Examinar"
                  uploadLabel="Cargar"
                  cancelLabel="Cancelar"
                  fileUploadListener="#{mainBean.uploadInstallation}"
                  oncomplete="PF('installationLoaderWidget').hide()"/>
    <p:commandButton type="button"
                     value="Cerrar"
                     icon="fa fa-close"
                     onclick="PF('installationLoaderWidget').hide()" />
  </p:panelGrid>
</p:dialog>
```

```
<p:dialog header="Cargar reglas"
          widgetVar="ruleLoaderWidget"
          id="ruleLoader"
          modal="true"
          resizable="false">
  <p:panelGrid columns="1"
               styleClass="ui-noborder"
               style="display:block; text-align:center">
    <p:fileUpload styleClass="ui-noborder"
                  allowTypes="/(\.|\/)(xml)$/"
                  invalidFileMessage="Tipo de archivo no válido"
                  auto="false"
                  update="ruleEditor,ruleList"
                  label="Examinar"
                  uploadLabel="Cargar"
                  cancelLabel="Cancelar"
                  fileUploadListener="#{rulesEditorBean.uploadRules}"
                  oncomplete="PF('ruleLoaderWidget').hide()"/>
    <p:commandButton type="button"
                     value="Cerrar"
                     icon="fa fa-close"
                     onclick="PF('ruleLoaderWidget').hide()"/>
  </p:panelGrid>
</p:dialog>
```

Los diálogos se han definido como modales y se ha inhabilitado la posibilidad de redimensionarlos. Ambos diálogos contienen el control `<p:fileUpload>` que ofrece ya de manera nativa botones para la selección y carga de un fichero. Las etiquetas de dichos botones aparecen por defecto en inglés, por lo que se han utilizado los atributos correspondientes para que el texto aparezca en castellano.

Se ha limitado la carga de archivos para que solo se acepten aquellos con extensión *.xml* y también se ha definido el mensaje de error a mostrar. En este caso el método del ***managed bean*** encargado de gestionar este control se establece mediante el atributo especial `fileUploadListener`; recordar que en el resto de controles el atributo se denomina `actionListener`.

Una vez completada la acción, se invoca al método `hide()` para ocultar el diálogo y se actualizan los controles indicados. Al igual que ocurría con el método para mostrar el diálogo, el control es referenciado mediante su nombre en la parte del cliente. Finalmente, se ha incluido un botón *Cerrar* para que el usuario pueda cancelar la operación si lo desea; este botón simplemente oculta el diálogo.

En cuanto al cuadro de diálogo dedicado a la conexión con KNX y la ejecución de reglas, utiliza su propio ***managed bean*** `RulesPlayer.Java`.

```
<p:dialog header="Ejecutar reglas"
          widgetVar="runRNWidget"
          id="runRN"
          modal="true"
          resizable="false"
          focus="gatewayIP">
  <h:panelGrid columns="2"
               columnClasses="ui-panelgrid-right,ui-panelgrid-left">
    <h:outputLabel for="beagleIP"
                   value="IP Beaglebone black: "
                   style="font-weight: bold"/>
    <p:outputLabel id="beagleIP"
                   value="#{rulesPlayer.handlerKNX.beagleIP}"/>
    <h:outputLabel for="gatewayIP" value="IP pasarela KNX/IP: "
                   style="font-weight: bold"/>
    <p:outputLabel id="gatewayIP"
                   value="#{rulesPlayer.handlerKNX.gatewayIP}" />
    <h:outputLabel for="portKNX" value="Puerto KNX: "
                   style="font-weight: bold;"/>
    <p:spinner id="portKNX"
               value="#{rulesPlayer.handlerKNX.portKNX}"
               min="0000" max="9999" maxlength="4" size="2"
               disabled="#{rulesPlayer.handlerKNX.connected}"/>
    <p:separator style="visibility: hidden"/>
      .
```

Cuenta con una serie de etiquetas en las que se muestra la IP de la *BeagleBoard* y de la pasarela KNX/IP. El puerto de comunicación se establece mediante un control `<p:spinner>` para el que se ha fijado el rango de valores y su longitud máxima.

En la parte inferior hay una serie de botones <p:commandButton> para la conexión y desconexión con KNX, la puesta en marcha y paro de la ejecución de reglas y el cierre del diálogo.

```
      .
<f:facet name="footer">
      <h:panelGroup style="display:block; text-align:center">
        <p:commandButton id="buttonConnect"
                         iconPos="center"
                         icon="fa fa-thumbs-up"
                         title="Conectar"
                         actionListener="#{rulesPlayer.connect2KNX(
                                     mainBean.rulesNet.installation,
                                     mainBean.rulesNet.variables)}"
                         update="gatewayIP, buttonConnect,
                                 buttonDisconnect, buttonClose,
                                 buttonPlay"
                                 disabled="#{rulesPlayer.
                                             handlerKNX.connected}"
                         styleClass="ui-button-dialog-small"/>
        <p:commandButton id="buttonDisconnect"
                         icon="fa fa-thumbs-down"
                         title="Desconectar"
                         actionListener="#{rulesPlayer.disconnectKNX
                                  (mainBean.rulesNet.installation)}"
                         update="gatewayIP, buttonConnect,
                         buttonDisconnect, buttonClose,
                         buttonPlay, buttonStop"
                         disabled="#{!rulesPlayer.
                                      handlerKNX.connected}"
                         styleClass="ui-button-dialog-small"/>
        <p:commandButton id="buttonClose"
                         icon="fa fa-close"
                         title="Cerrar"
                         onclick="PF('runRNWidget').hide()"
                         disabled="#{rulesPlayer.
                         handlerKNX.connected}"
                         styleClass="ui-button-dialog-small"/>
        <p:commandButton id="buttonPlay"
                         icon="fa fa-play"
                         title="Ejecutar reglas"
                         onclick="PF('editVarWidget').hide()"
                         actionListener="#{rulesPlayer.runRules
                                           (mainBean.rulesNet,)}"
                         update="buttonPlay,buttonStop"
                         disabled="#{!rulesPlayer.playEnabled}"
                         styleClass="ui-button-dialog-small"/>
        <p:commandButton id="buttonStop" title="Detener reglas"
                         icon="fa fa-stop"
                         onclick="PF('editVarWidget').hide()"
                         actionListener="#{rulesPlayer.stopRules()}"
                         update="buttonPlay,buttonStop"
                         disabled="#{!rulesPlayer.runningRules}"
                         styleClass="ui-button-dialog-small"/>
      </h:panelGroup>
    </f:facet>
  </h:panelGrid>
</p:dialog>
```

A continuación, se detallan los métodos del ***managed bean*** asociados a cada uno de estos botones:

Botón *Conectar* → Método `connect2KNX`

```
public void connect2KNX(String installationName,
                        ArrayList<VariableXML> variables ) {
  handlerKNX.openConection();
  if (handlerKNX.getConnected()) {
    FacesContext context = FacesContext.getCurrentInstance();
    context.addMessage(null,
                       new FacesMessage(FacesMessage.SEVERITY_INFO,
                       "Conexión KNX",
                       "Se ha conectado a " + installationName));
    playEnabled = true;
    checkInstallation(installationName,variables);
  }
}
```

A este método se le pasa como parámetros el nombre de la instalación y las variables de la red de reglas. Se encarga de establecer la conexión con KNX y de leer el estado actual de las variables de entrada mediante el método `checkInstallation`.

```
public void checkInstallation(String installationName,
                              ArrayList<VariableXML> variables) {
  Boolean currentValue;
  Logger.getLogger(RulesPlayerBean.class.getName()).
                log(Level.INFO,
                "ESTADO ACTUAL DE LA INSTALACION " +
                installationName.toUpperCase());
  for (VariableXML var : variables) {
    if (var.getType().equals("INPUT")) {
      switch (var.getVarXlatorMainNumber()) {
        case 1:
          currentValue = HandlerKNX.readBoolSensor(var.getGroupAddress());
          var.getFacts().get(0).setFactState(!currentValue);
          var.getFacts().get(1).setFactState(currentValue);
          Logger.getLogger(RulesPlayerBean.class.getName()).
            log(Level.INFO, var.getFacts().get(0).getFullName() +
            " - " + var.getFacts().get(0).getFactState());
          Logger.getLogger(RulesPlayerBean.class.getName()).
            log(Level.INFO, var.getFacts().get(1).getFullName() +
            " - " + var.getFacts().get(1).getFactState());
          break;
            .
            .
        default:
          break;
      }
    }
  }
}
```

Botón Desconectar → Método `disconnectKNX`

```
public void disconnectKNX(String installationName) {
        if (runningRules) {
            stopRules();
        }
        handlerKNX.closeConnection();
        FacesContext context = FacesContext.getCurrentInstance();
        context.addMessage(null, new
FacesMessage(FacesMessage.SEVERITY_INFO, "Conexión KNX", "Se ha
desconectado de " + installationName));
        playEnabled = false;
    }
```

A este método se le pasa como parámetro el nombre de la instalación. En caso de que las reglas estén ejecutándose, llama al método `stopRules` para detenerlas. Si no hay reglas en ejecución, cierra la conexión con KNX llamando al método `closeConnection` de la clase `HandlerKNX`, la cual se describió en el apartado 5.9.

Botón *Ejecutar reglas* → Método `runRules`

```
public void runRules(RulesNetXML rulesNet) {
  eventQueue = new ArrayBlockingQueue<>(1024);
  commandQueue = new ArrayBlockingQueue<>(1024);
  EventListenerKNX eventListener = new EventListenerKNX (rulesNet,
                                                    eventQueue);
  execRN = new RulesNetExec(rulesNet, eventQueue, commandQueue);
  execRN.setName("Ejecutor de Reglas");
  execKNX = new CommandExecKNX(commandQueue, handlerKNX);
  execKNX.setName("Ejecutor de órdenes KNX");
  Logger.getLogger(RulesPlayerBean.class.getName()).
    log(Level.INFO, "COMENZANDO EJECUCIÓN DE REGLAS");
  handlerKNX.setEventListener(eventListener);
  execRN.start();
  execRN.applyReasoning();
  execKNX.start();
  runningRules = true;
  playEnabled = false;
}
```

A este método se le pasa como parámetros el objeto que contiene la red de reglas. El método realiza las siguientes tareas:

- Crea las colas de eventos y órdenes KNX.
- Pone en marcha los hilos de ejecución necesarios para la detección de eventos, la ejecución de reglas y la ejecución de órdenes KNX
- Lanza un primer razonamiento para detectar si el estado inicial sensibiliza alguna de las reglas.

Botón *Detener reglas* → Método `stopRules`

```
public void stopRules() {
  handlerKNX.removeEventListener();
  while (!eventQueue.isEmpty());
    while (!commandQueue.isEmpty());
      execRN.ceaseRunning();
      execKNX.ceaseRunning();
      playEnabled = true;
      runningRules = false;
}
```

En primer lugar, detiene el detector de eventos, desvinculando el `EventListener` del enlace KNX. Para ello, hace uso del el método `removeEventListener` de la clase `HandlerKNX`. A continuación, espera a que las colas de eventos y órdenes se vacíen. Finalmente, detiene los hilos encargados de la ejecución de reglas y de la ejecución de órdenes KNX.

Editor de variables

Los elementos que componen la pantalla del editor de variables se han agrupado mediante la etiqueta `<h:panelGroup>` con el identificador `id="varList"`. Por motivos de funcionamiento, el diálogo para editar una variable se ha colocado fuera del formulario principal `rnknxForm`, incluyendo dentro de dicho diálogo su propio formulario con el identificador `editVarForm`.

```
      .
      .
<h:form id="rnknxForm" enctype="multipart/form-data" >
      .
      .
<h:panelGroup id="varList">
  <p:menubar  rendered="#{MainBean.showVarsList}"
              toggleEvent="click">
      .
      .
  </p:menubar>
  <p:dataTable id="varsList"
      .
      .

  </p:dataTable>
</h:panelGroup>
      .
      .
</h:form>
<p:dialog header="Editar Variable"
          widgetVar="editVarWidget"
      .
      .
  <h:form id="editVarForm">
      .
      .
  </h:form>
</p:dialog>
```

En esta pantalla se dispone de una barra de menú que contiene una única opción para regresar a la pantalla principal.

```
<p:menubar  rendered="#{mainBean.showVarsList}" toggleEvent="click">
  <p:submenu label="EDITOR DE VARIABLES" style="font-weight: bold">
    <p:menuitem value="Volver"
                actionListener="#{mainBean.endEditVars()}"
                update=":rnknxForm:rnknxMain,:rnknxForm:varList"
                icon="fa fa-reply" />
    </p:submenu>
</p:menubar>
```

También cuenta con un control *DataTable* en el que se muestra la lista de variables a editar.

```
<p:dataTable id="varsList"
             value="#{mainBean.rulesNet.variables}"
             var="var" rowKey="#{var.tag}"
             selection="#{mainBean.selectedVar}"
             emptyMessage="No existen variables definidas"
             rendered="#{mainBean.showVarsList}">
  <p:ajax event="rowSelectRadio"
          listener="#{mainBean.onRowSelectRadio}"
          update="editVarButton"/>
  <p:column selectionMode="single"
            style="width:16px;text-align:center"/>
  <p:column headerText="Variable"
            style="text-align: justify"
            sortBy="#{var.tag}">
    <h:outputText value="#{var.tag}"/>
  </p:column>
  <p:column headerText="Nombre"
            style="text-align: justify"
            sortBy="#{var.name}">
    <h:outputText value="#{var.name}"/>
  </p:column>
  <f:facet name="footer">
    <h:panelGroup style="display:block; text-align:center">
      <p:commandButton id="editVarButton" title="Editar variable"
                       icon="fa fa-edit"
                       disabled="#{mainBean.noneSelected}"
                       onclick="PF('editVarWidget').show();"
                       update=":editVarForm">
      </p:commandButton>
    </h:panelGroup>
  </f:facet>
</p:dataTable>
```

En la parte inferior de la lista, se dispone de un botón de edición que se habilita al seleccionar una de las filas de la tabla. Al pulsar el botón, se muestra el diálogo para editar el nombre de la variable seleccionada.

Este diálogo muestra, mediante una etiqueta `<p:outputLabel>`, el ***tag*** de la variable cuyo nombre se quiere modificar. El nombre de la variable se muestra en una caja de texto `<p:inputText>`. Se ha definido el contenido de esta caja de texto como obligatorio mediante el atributo `required="true"`, de modo que si se deja en blanco, no se modificará el valor del atributo en el ***managed bean***, quedando como estaba. En la parte inferior del diálogo se dispone de los habituales botones *Aceptar* y *Cancelar*.

```
<p:dialog header="Editar Variable"
          widgetVar="editVarWidget"
          id="editVar"
          modal="true"
          resizable="false">
  <h:form id="editVarForm">
    <h:panelGrid columns="2"
                 columnClasses="ui-panelgrid-right,
                                ui-panelgrid-simple">
      <h:outputLabel for="varTag"
                     value="Variable: "
                     style="font-weight: bold"/>
      <p:outputLabel id="varTag"
                     value="#{mainBean.selectedVar.tag}"/>
      <h:outputLabel for="varName"
                     value="Nombre: "
                     style="font-weight: bold"/>
      <p:inputText id="varName"
                   value="#{mainBean.selectedVar.name}" size="40"
                   required="true"/>
      <p:separator style="visibility: hidden"/>
      <f:facet name="footer">
        <h:panelGroup style="display:block; text-align:center">
          <p:commandButton value="Aceptar"
                           update=":rnknxForm:varList"
                           oncomplete="PF('editVarWidget').hide()"
                           actionListener="#{mainBean.
                                             callUpdateRulesNet()}"
                           styleClass="ui-button-dialog"/>
          <p:commandButton value="Cancelar"
                           type="button"
                           onclick="PF('editVarWidget').hide()"
                           styleClass="ui-button-dialog"/>
        </h:panelGroup>
      </f:facet>
    </h:panelGrid>
  </h:form>
</p:dialog>
```

Como peculiaridad, este diálogo cuenta con su propio formulario. Debido a problemas con la implementación del control, si el ***DataTable*** se encuentra en el mismo formulario que el diálogo, el contenido del objeto `selectedVar` no se transmite correctamente entre ambos.

Editor de reglas

Los elementos que componen la pantalla del editor de reglas se han agrupado mediante la etiqueta `<h:panelGroup>` con el identificador `id="ruleEditor"`. El ***managed bean*** que contiene los atributos y manejadores de eventos para esta sección es `RuleEditorBean.Java`.

```
<h:panelGroup id="ruleEditor">
  <p:menubar  rendered="#{MainBean.showEditor}"
              toggleEvent="click">
       .
       .
  </p:menubar>
  <p:tree id="treeIF"
       .
       .
  </p:tree>
  <p:tree id="treeTHEN"
       .
       .
  </p:tree>
  <p:panel id="statusBar"
       .
       .
  </p:panel>
  <p:dialog header="Establecer Antecedentes"
            widgetVar="setAntsWidget"
            modal="true"
            resizable="false">
       .
       .
  </p:dialog>
  <p:dialog header="Establecer Consecuentes"
            widgetVar="setConsWidget"
            modal="true"
            resizable="false">
       .
       .
  </p:dialog>
</h:panelGroup>
```

Del mismo modo que en el caso de la pantalla principal, la barra de menú se ha implementado mediante el control `<p:menubar>`. Esta barra de menú solo será mostrada cuando el atributo `showEditor` del ***managed bean*** sea cierto.

```
<p:menubar  rendered="#{mainBean.showEditor}"
            toggleEvent="click">
  <p:submenu label="EDITOR DE REGLAS"
             style="font-weight: bold">
    <p:menuitem value="#{ruleEditor.IF}"
                onclick="PF('setAntsWidget').show();"
                icon="ui-icon-bullet" />
    <p:menuitem value="#{ruleEditor.THEN}"
                  onclick="PF('setConsWidget').show();"
                  icon="ui-icon-play" />
            .
```

```
              .
     <p:separator />
     <p:menuitem value="Guardar"
                 actionListener="#{mainBean.callSaveRule}"
                 update="rnknxMain,ruleEditor"
                 icon="fa fa-save"
                 disabled="#{ruleEditor.badRule}"/>
     <p:menuitem value="Cancelar"
                 actionListener="#{mainBean.cancelRule}"
                 update="rnknxMain, ruleEditor"
                 icon="ui-icon-circle-close" />
   </p:submenu>
 </p:menubar>
```

La barra de menú incluye un único submenú definido por la etiqueta `<p:submenu>` llamado EDITOR DE REGLAS. Las distintas opciones contenidas en dicho submenú se implementan mediante etiquetas `<p:menuitem>`. Las opciones se han dividido en grupos mediante la etiqueta `<p:separator/>`. En el primer grupo se encuentran las opciones "SI" y "ENTONCES" que llaman respectivamente a los diálogos para establecer los antecedentes y consecuentes de la regla. En el segundo grupo están las opciones para guardar la regla creada / editada o cancelar su creación / edición.

Para la representación en pantalla de la regla se ha utilizado el control *HorizontalTree* de *PrimeFaces*. Este control representa un árbol horizontal con controles para expandir o contraer las ramas de los nodos.

```
<p:tree id="treeIF"
        value="#{ruleEditor.rootIF}"
        var="nodeIF"
        orientation="horizontal"
        rendered="#{mainBean.showEditor}"
        onNodeClick="PF('setAntsWidget').show();">
  <p:treeNode>
    <h:outputText value="#{nodeIF}"/>
  </p:treeNode>
</p:tree>
<p:tree id="treeTHEN"
        value="#{ruleEditor.rootTHEN}"
        var="nodeTHEN"
        orientation="horizontal"
        rendered="#{mainBean.showEditor}"
        onNodeClick="PF('setConsWidget').show();">
  <p:treeNode>
    <h:outputText value="#{nodeTHEN}" />
  </p:treeNode>
</p:tree>
```

El control se crea mediante la etiqueta `<p:tree>`, indicando en su atributo `orientation` el valor `horizontal`. Se han utilizado dos de estos controles:

- Uno cuyo nodo raíz es SI y representa los antecedentes de la regla.
- Otro cuyo nodo raíz es ENTONCES y representa los consecuentes de la regla.

Ambos controles solo se muestran cuando el atributo `showEditor` del ***managed bean*** es cierto. Los árboles son almacenados en los objetos `rootIF` y `rootTHEN` del ***managed bean*** y se rellenan mediante los métodos `fillIFTree` y `fillTHENTree`.

```
public void fillIFTree() {
  rootIF = new DefaultTreeNode(IF, null);
  rootIF.setExpanded(true);
  if (!pickInputFacts.getTarget().isEmpty()) {
    Iterator<FactXML> iter = pickInputFacts.getTarget().iterator();
    while (iter.hasNext()) {
      TreeNode node = new DefaultTreeNode(iter.next().getFullName(),
                                          rootIF);
    }
  }
}

public void fillTHENTree() {
  rootTHEN = new DefaultTreeNode(THEN, null);
  rootTHEN.setExpanded(true);
  if (!pickOutputFacts.getTarget().isEmpty()) {
    Iterator<FactXML> iter = pickOutputFacts.getTarget().iterator();
    while (iter.hasNext()) {
      TreeNode node = new DefaultTreeNode(iter.next().getFullName(),
                                          rootTHEN);
    }
  }
}
```

Los objetos `rootIF` y `rootTHEN` son instancias de la clase `DefaultTreeNode`, propia de ***PrimeFaces***. En su constructor hay que indicar tanto el texto que aparece en el nodo, como quien es su nodo antecesor. En el caso del nodo raíz se indica `null` como antecesor, los distintos nodos que representan los antecedentes o los consecuentes de la regla tendrán como antecesor al nodo raíz.

Mediante el atributo `onNodeClick`, se ha determinado que cada vez que se haga clic sobre un nodo del árbol, se muestre el diálogo para establecer los antecedentes o los consecuentes de la regla según corresponda.

La barra de estado se ha creado haciendo uso del control ***Panel*** de ***PrimeFaces*** mediante la etiqueta `<p:panel>`. Este control solo será mostrado cuando el atributo `showEditor` del ***managed bean*** sea cierto. Contiene una etiqueta `<h:outputText>` en la que se indicará la operación que se esté realizando en el editor, ya sea la creación de una nueva regla o la edición de una existente. También se advertirá de si la regla presenta alguna anomalía en su formulación.

```
<p:panel id="statusBar"
         rendered="#{mainBean.showEditor}"
         styleClass="ui-status-bar">
  <h:outputText value="#{ruleEditor.status}"/>
</p:panel>
```

Los diálogos para establecer los antecedentes y consecuentes se han implementado haciendo uso del control `<p:dialog>`. Los diálogos se han definido como modales y se ha inhabilitado la posibilidad de redimensionarlos.

```
<p:dialog header="Establecer Antecedentes"
          widgetVar="setAntsWidget"
          modal="true"
          resizable="false">
  <p:pickList value="#{ruleEditor.pickInputFacts}"
              var="fact"
              itemLabel="#{fact.fullName}"
              itemValue="#{fact}"
              converter="factConverter"
              effect="none"
              addLabel="Añadir"
              addAllLabel="Añadir todos"
              removeLabel="Quitar"
              removeAllLabel="Quitar todos">
    <f:facet name="sourceCaption">Hechos</f:facet>
    <f:facet name="targetCaption">Antecedentes</f:facet>
  </p:pickList>
  <f:facet name="footer">
    <h:panelGroup style="display:block; text-align:center">
      <p:commandButton value="Aceptar"
                       update="treeIF,statusBar,editorMenu"
                       oncomplete="PF('setAntsWidget').hide()"
                       actionListener=
                             "#{ruleEditor.updateAntecedents}"
                       styleClass="ui-button-dialog"/>
      <p:commandButton value="Cancelar"
                       type="button"
                       onclick="PF('setAntsWidget').hide()"
                       styleClass="ui-button-dialog"/>
    </h:panelGroup>
  </f:facet>
</p:dialog>
```

Cada uno de estos diálogos incluyen un control *PickList* implementado mediante la etiqueta `<p:pickList>` que contiene dos listas llamadas fuente y destino. El usuario puede desplazar los elementos de una a otra lista mediante una serie de botones o bien arrastrándolos con el ratón. Las listas de este control forman parte de un objeto que es una instancia de la clase `DualListModel` de *PrimeFaces*. Al constructor de este objeto se le pasan como parámetros dos `ArrayList<FactXML>` de hechos que representan las listas fuente y destino. Dichas listas podrán ser accedidas posteriormente mediante los métodos `getSource()` y `getTarget()`.

```
<p:dialog header="Establecer Consecuentes"
          widgetVar="setConsWidget"
          modal="true"
          resizable="false">
  <p:pickList value="#{ruleEditor.pickOutputFacts}"
              var="fact"
              itemLabel="#{fact.fullName}"
              itemValue="#{fact}"
              converter="factConverter"
              effect="none"
              addLabel="Añadir"
              addAllLabel="Añadir todos"
              removeLabel="Quitar"
              removeAllLabel="Quitar todos">
     <f:facet name="sourceCaption">Hechos</f:facet>
     <f:facet name="targetCaption">Consecuentes</f:facet>
  </p:pickList>
  <f:facet name="footer">
    <h:panelGroup style="display:block; text-align:center">
      <p:commandButton value="Aceptar"
                       update="treeTHEN,statusBar,editorMenu"
                       oncomplete="PF('setConsWidget').hide()"
                       actionListener=
                             "#{ruleEditor.updateConsequents}"
                       styleClass="ui-button-dialog"/>
      <p:commandButton value="Cancelar"
                       type="button"
                       onclick="PF('setConsWidget').hide()"
                       styleClass="ui-button-dialog"/>
    </h:panelGroup>
  </f:facet>
</p:dialog>
```

Si se está creando una nueva regla, las listas fuente de los objetos `pickInputFacts` y `pickOutputFacts` se rellenan con los hechos de las variables de entrada y salida respectivamente. Las listas destino de ambos objetos estarán vacías y se irán llenando por parte del usuario.

En el caso de que se esté editando una regla, las listas destino de los objetos `pickInputFacts` y `pickOutputFacts` se rellenarán con los antecedentes y consecuentes de la regla, mientras que en las listas fuente se incluirán aquellos hechos de entrada y salida que no estén en las respectivas listas destino.

```
public void fillInputFactsPL(RulesNetXML rulesNet) {
  pickInputFacts.getSource().clear();
  pickInputFacts.getTarget().clear();
  for (VariableXML var : rulesNet.getVariables()) {
    if (var.getType().equals("INPUT")) {
      for (FactXML fact : var.getFacts()) {
        pickInputFacts.getSource().add(fact);
      }
    }
  }
      .
```

```
      .
  if (editedRule!=null && !editedRule.getAntecedents().isEmpty()) {
    for (FactXML fact : editedRule.getAntecedents()) {
      VariableXML var = rulesNet.getVarByTag(fact.getVarTag());
      pickInputFacts.getTarget().add(var.getRuleFact(fact));
    }
    for (FactXML fact : pickInputFacts.getTarget()) {
      if (pickInputFacts.getSource().contains(fact)) {
          pickInputFacts.getSource().remove(fact);
      }
    }
  }
}
```

Los atributos `addLabel`, `addAllLabel`, `removeLabel` y `removeAllLabel` indican el mensaje de texto que aparecerá al pasar el ratón por encima de los botones para mover elementos entre las listas fuente y destino.

El atributo `itemLabel` indica la representación en pantalla en forma de cadena de texto de cada elemento de la lista, mientras que el atributo `itemValue` indica el objeto Java que representa cada elemento de la lista. De manera nativa el control *PickList* opera con objetos de tipo cadena, para que pueda manejar otros tipos de objeto se precisa un conversor de cadena a objeto y viceversa. Dicho conversor se ha creado mediante la clase `EntityConverter`.

```
import java.util.Map;
import java.util.Map.Entry;
import java.util.UUID;
import java.util.WeakHashMap;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

@FacesConverter(value = "factConverter")
public class EntityConverter implements Converter {

  private static Map<Object, String> entities =
                                new WeakHashMap<Object, String>();
  @Override
  public String getAsString(FacesContext context,
                            UIComponent component,
                            Object entity) {
    synchronized (entities) {
      if (!entities.containsKey(entity)) {
        String uuid = UUID.randomUUID().toString();
        entities.put(entity, uuid);
        return uuid;
      } else {
        return entities.get(entity);
      }
    }
  }
      .
```

```
      .
  @Override
  public Object getAsObject(FacesContext context,
                            UIComponent component,
                            String uuid) {
    for (Entry<Object, String> entry : entities.entrySet()) {
      if (entry.getValue().equals(uuid)) {
        return entry.getKey();
      }
    }
    return null;
  }
}
```

Esta clase implementa el interfaz `Javax.faces.convert.Converter` y en ella se han sobrescrito los métodos `getAsString` y `getAsObject`. La anotación `@FacesConverter` identifica a la clase como un conversor y con su atributo `value` se especifica el nombre por el que es referenciado, en este caso `factConverter`.

Finalmente, al pie de cada diálogo se han colocado los habituales botones *Aceptar y Cancelar*. El botón *Aceptar* oculta el cuadro de diálogo e invoca a al método correspondiente para actualizar el árbol que representa los antecedentes o los consecuentes de la regla.

```
public void updateAntecedents() {
  fillIFTree();
  RuleChecker ruleChecker = new RuleChecker();
  status = ruleChecker.checkRule(
           (ArrayList<FactXML>) pickInputFacts.getTarget(),
           (ArrayList<FactXML>) pickOutputFacts.getTarget());
  badRule = !status.equals("REGLA CORRECTA");
}

public void updateConsequents() {
  fillTHENTree();
  RuleChecker ruleChecker = new RuleChecker();
  status = ruleChecker.checkRule(
           (ArrayList<FactXML>) pickInputFacts.getTarget(),
           (ArrayList<FactXML>) pickOutputFacts.getTarget());
  badRule = !status.equals("REGLA CORRECTA");
}
```

Estos métodos, además de actualizar el árbol, comprueban que la regla sea correcta antes de permitir que está se pueda guardar. Para verificar las reglas se ha creado la clase `RuleChecker` que contiene los métodos para comprobar si la regla es pura.

```
public Boolean isIncomplete(RuleXML rule) {
  return (rule.getAntecedents().isEmpty() |
          rule.getConsequents().isEmpty());
}
```

```
public Boolean isImposible(RuleXML rule) {
  for (FactXML fact : rule.getAntecedents()) {
    String varTag = fact.getVarTag();
    String factName = fact.getName();
    for (FactXML cFact : rule.getAntecedents()) {
      if (cFact.getVarTag().equals(varTag) &&
         !cFact.getName().equals(factName)) {
        return true;
      }
    }
  }
  return false;
}
```

```
public Boolean isSelfContradictory(RuleXML rule) {
  for (FactXML fact : rule.getConsequents()) {
    String varTag = fact.getVarTag();
    String factName = fact.getName();
    for (FactXML cFact : rule.getConsequents()) {
      if (cFact.getVarTag().equals(varTag) &&
         !cFact.getName().equals(factName)) {
        return true;
      }
    }
  }
  return false;
}
```

```
public Boolean isFedBack(RuleXML rule) {
  for (FactXML fact : rule.getAntecedents()) {
    String factName = fact.getFullName();
      for (FactXML cFact : rule.getConsequents()) {
        if (cFact.getFullName().equals(factName)) {
          return true;
        }
      }
    }
  return false;
}
```

El método `checkRule` de la clase `RuleChecker`, llama a estos cuatro métodos y devuelve una cadena indicando si la regla es correcta o si presenta alguna anomalía.

ANEXO II: Manual de usuario

Introducción

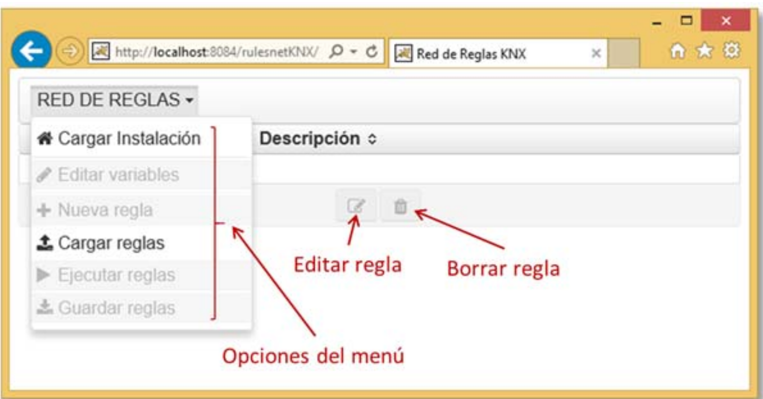
En el presente anexo se describe el manejo de las distintas funcionalidades que ofrece la aplicación WEB implementada. Para acceder a la aplicación, se deberá disponer de un navegador WEB en el que introduciremos la siguiente URL:

http://IP_BeagleBone:Puerto_Tomcat/rulesnetKNX/

Pantalla principal

La pantalla principal muestra una barra de menú y la lista de las reglas definidas por el usuario. En la lista de reglas se indica para cada regla su nombre y su descripción. En principio la lista de reglas estará vacía.

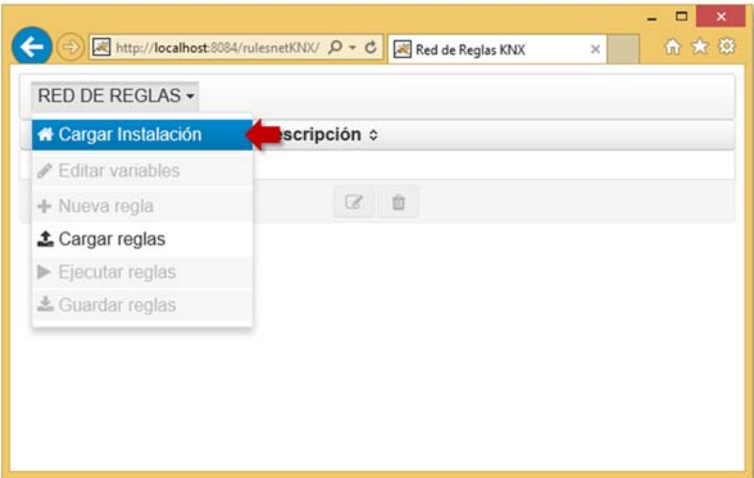
En la parte inferior de la lista de reglas existen dos botones para editar y borrar una regla previamente seleccionada. En caso de no tener ninguna regla seleccionada, estos botones aparecen inhabilitados.



En cuanto al menú, inicialmente solo están habilitadas las opciones que permiten la carga de un fichero XML, conteniendo bien una instalación KNX o bien un conjunto de reglas.

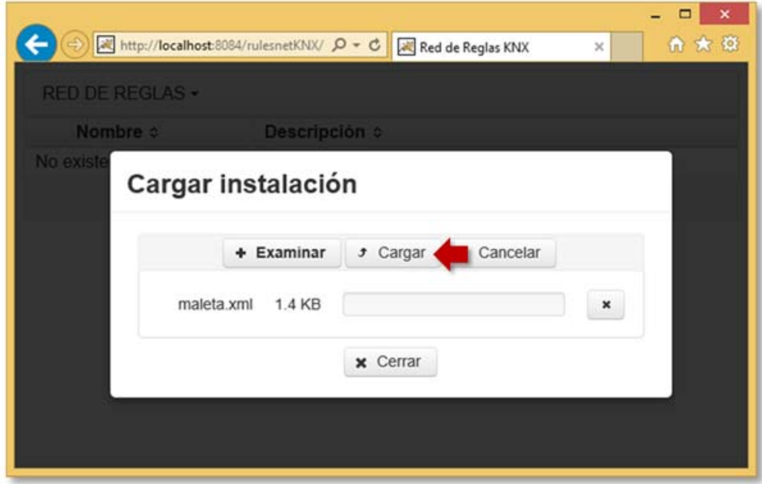
Cargar un fichero de instalación

Para cargar un fichero XML representando la instalación KNX, hay que hacer clic sobre la opción correspondiente del menú.



A continuación, aparecerá un cuadro de diálogo para seleccionar y cargar el archivo correspondiente.

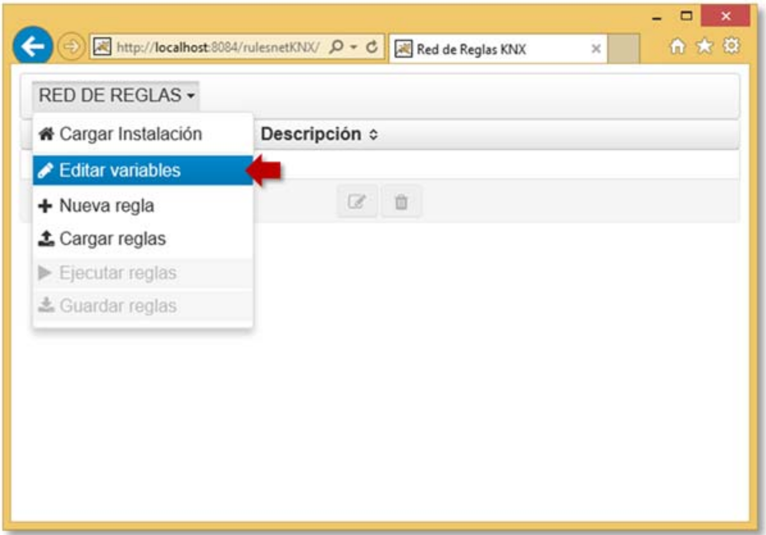
Pulsando sobre el botón *Examinar* se nos mostrará el explorador de archivos para poder seleccionar el fichero de instalación que se quiere cargar. Únicamente se permite la carga de ficheros con extensión *.xml*, si se intenta cargar otro tipo de archivo se informará al usuario mediante un mensaje de error.



Una vez seleccionado el fichero, se deberá pulsar sobre el botón *Cargar*. Si se produce algún error durante el procesado del archivo, se indicará al usuario mediante un mensaje emergente. Si todo funciona correctamente, aparecerán habilitadas todas las opciones del menú.

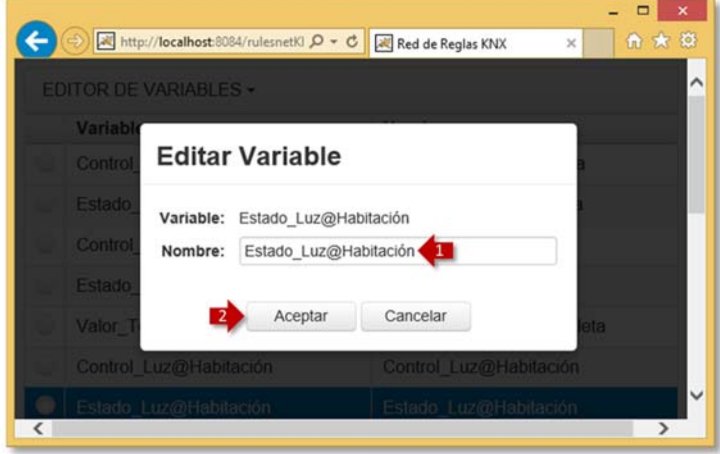
Editar el nombre de las variables

Tras cargar el fichero de la instalación, la aplicación genera automáticamente las variables con sus hechos, dando a estas unos nombres por defecto. Si el usuario lo desea, puede dar a las variables un nombre que le resulte más adecuado; el nombre de los hechos no se puede modificar. Para acceder al editor de variables, se deberá seleccionar la opción correspondiente en el menú. Esta opción solo estará habilitada si se ha cargado un fichero de instalación o de reglas.



Una vez en el editor, para modificar el nombre de una variable, hay que seleccionarla haciendo clic en el botón de opción situado en la primera columna de la lista, esto hará que se habilite el botón *Editar* situado en la parte inferior de la pantalla. A continuación, haremos clic sobre dicho botón.

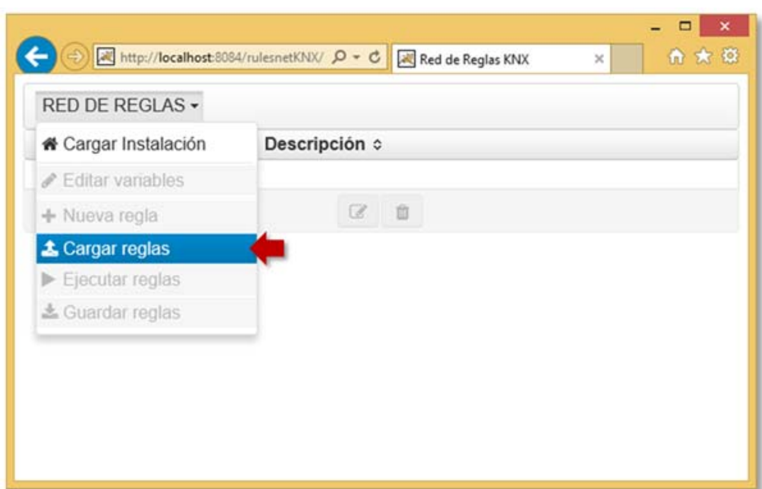
Se nos mostrará un cuadro de diálogo en el que podremos modificar el nombre de la variable. Para guardar los cambios pulsaremos sobre el botón *Aceptar*. En caso de dejar vacío el campo del nombre, no se modificará el nombre de la variable al pulsar sobre *Aceptar*.



Una vez finalicemos, podremos volver a la pantalla principal pulsando sobre la opción *Volver* del menú. Si se modifica el nombre de una variable cuyos hechos ya formen parte de los antecedentes o los consecuentes de alguna regla, la descripción de dicha regla se modificará automáticamente.

Cargar un fichero de reglas

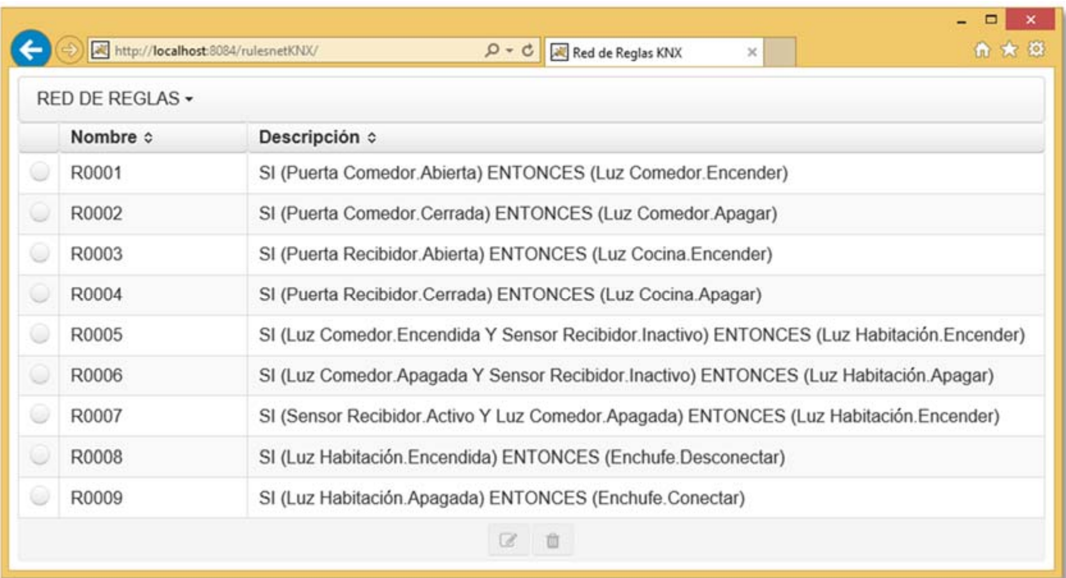
Para cargar un fichero conteniendo reglas que hayamos definido en una sesión de trabajo anterior, en primer lugar deberemos seleccionar la opción correspondiente del menú.



Seguidamente, nos aparecerá un cuadro de diálogo en el que pulsaremos sobre el botón *Examinar* para seleccionar el fichero que queramos cargar. Solamente se permite la carga de ficheros con extensión *.xml*, si se intenta cargar otro tipo de fichero, se nos mostrará un mensaje de error.

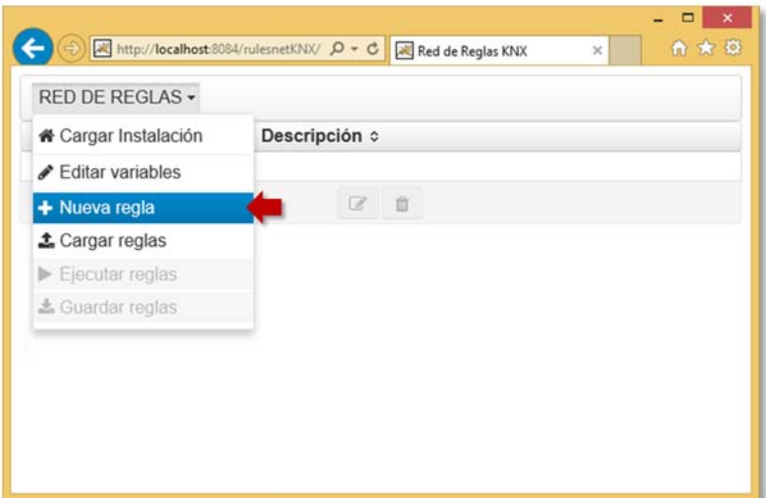
Una vez seleccionado el fichero, pulsaremos sobre el botón Cargar para subirlo a la aplicación.

Si se produce algún error durante el procesado del fichero, se nos indicará mediante un mensaje emergente. Si la carga se realiza correctamente, nos aparecerán en la lista las reglas que contenía el fichero y se habilitarán todas las opciones del menú.



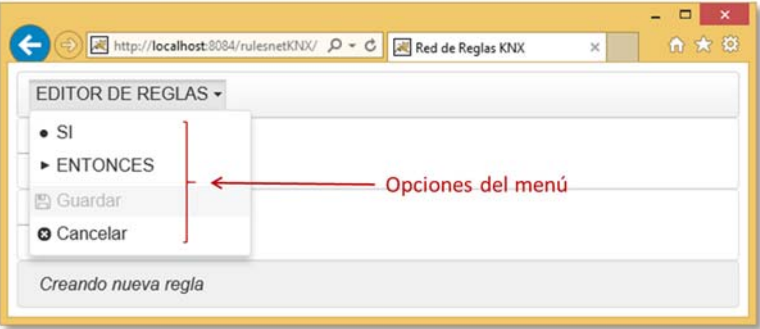
Crear una nueva regla

Para definir una nueva regla es necesario tener cargado en la aplicación un fichero de instalación o uno de reglas. En primer lugar, escogeremos la opción correspondiente del menú.

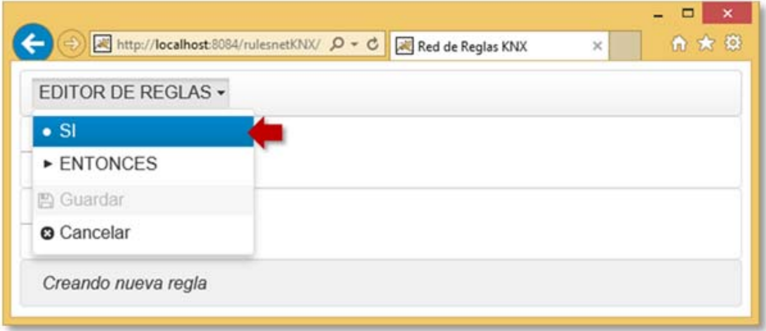


A continuación, se nos mostrará la pantalla del editor de reglas. En la parte superior tenemos una barra de menú, en la parte central nos aparece una representación de la regla que está siendo editada y en la parte inferior tenemos una barra de estado que nos irá informando de incidencias durante el proceso de edición.

El menú cuenta con las opciones para el establecimiento de los antecedentes y consecuentes de la regla, el guardado de la regla o la cancelación de la operación, lo cual nos devuelve a la pantalla principal sin realizar ningún cambio.



Para establecer los antecedentes de la regla, se seleccionará la opción *SI* en el menú o se hará clic sobre el área de antecedentes de la representación de la regla.



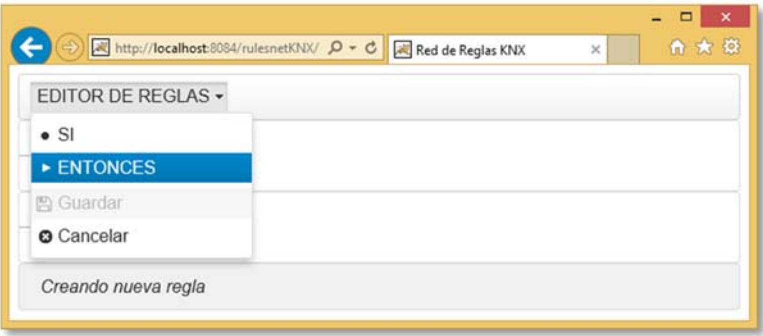
Aparecerá un cuadro con dos listas y una serie de botones de operación. La lista de la izquierda contiene los hechos de todas la varieables de entrada que hay disponibles. La lista de la derecha estará inicialmente vacía y en ella deberemos colocar aquellos hechos que queramos definir como antecedentes de la regla.

Para mover los hechos de una lista a otra podemos realizar tres operaciones:

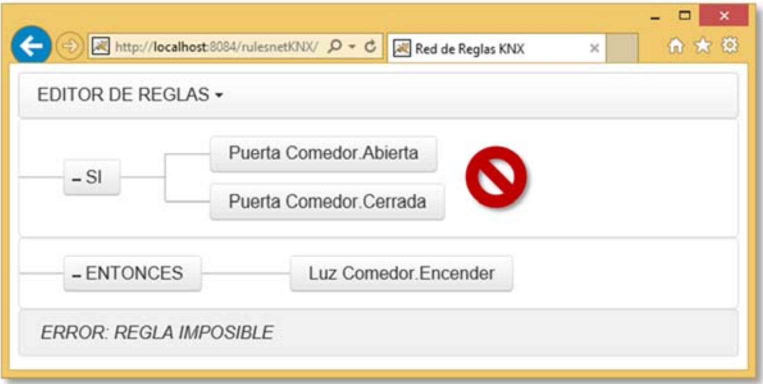
- Hacer doble clic sobre el hecho.
- Pinchar y arrastrar el hecho de una lista a la otra.
- Seleccionar uno o más hechos y hacer clic sobre alguno los botones situados entre ambas listas en función de la operación que queramos realizar.

Una vez hayamos establecido los antecedentes de la regla, pulsaremos sobre el botón *Aceptar* para guardar los cambios. Volveremos a la pantalla del editor y nos aparecerá en la barra de estado el mensaje de que la regla está incompleta, ya que carece de consecuentes. Si accedemos al menú veremos que no se nos permite el guardado de la regla en estas condiciones.

Para establecer los consecuentes de la regla, se seleccionará la opción *ENTONCES* en el menú o se hará clic sobre el área de consecuentes de la representación de la regla. Aparecerá un cuadro similar al visto para el establecimiento de antecedentes, con la diferencia de que en la lista de la izquierda aparecerán los hechos de las variables de salida.

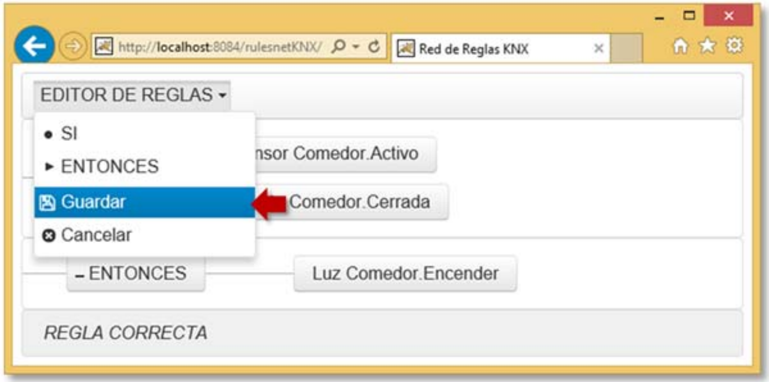


Si durante la creación de la regla seleccionamos como antecedente más de un hecho de una misma variable, se nos mostrará en la barra de estado un mensaje indicando que la regla es imposible y no se nos permitirá guardarla.



Si lo que hacemos es seleccionar como consecuente de una regla más de un hecho de una misma variable, aparecerá en la barra un mensaje indicando que la regla es autocontradictoria y tampoco se nos permitirá guardarla.

En caso de que la regla no presente ninguna anomalía, se nos permitirá su guardado y regresaremos a la pantalla principal.



En la lista aparecerá la nueva regla con su identificador y su descripción.

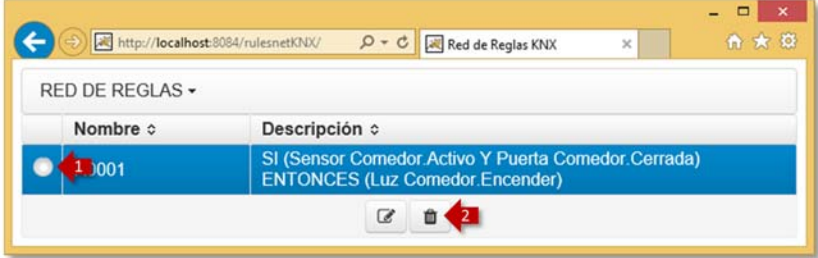
Editar una regla existente

Para editar una regla existente, se deberá seleccionar en la lista haciendo clic sobre el botón de opción que parece en la primera columna.

Seguidamente, se pulsará sobre el botón *Editar* que abrirá la pantalla del editor de reglas mostrando los antecedentes y consecuentes de la regla seleccionada.

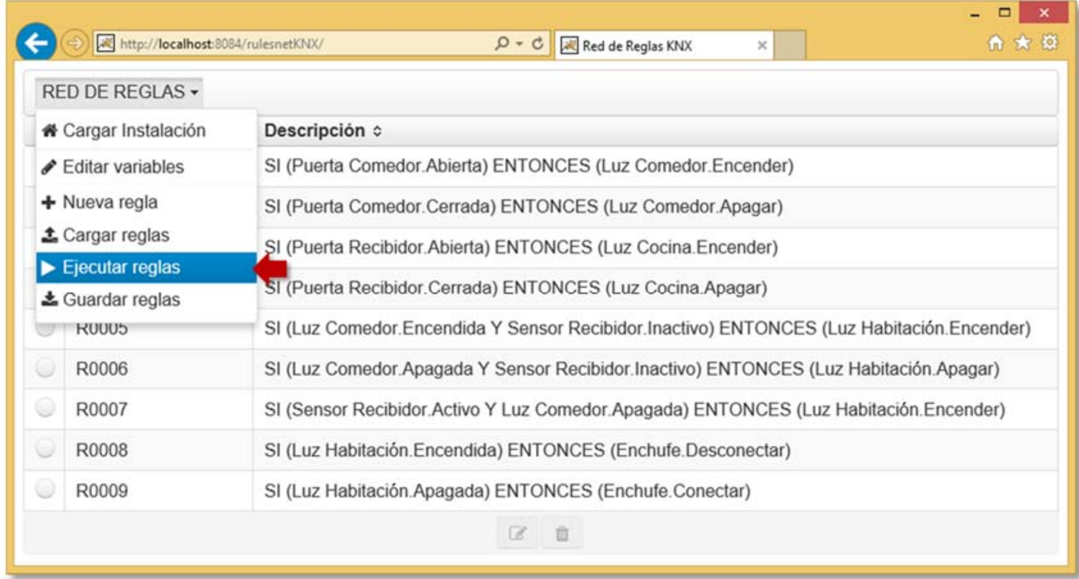
Eliminar una regla existente

Para eliminar una regla existente, se deberá seleccionar en la lista haciendo clic sobre el botón de opción que parece en la primera columna. A continuación, se pulsará sobre el botón *Borrar* para eliminar la regla.

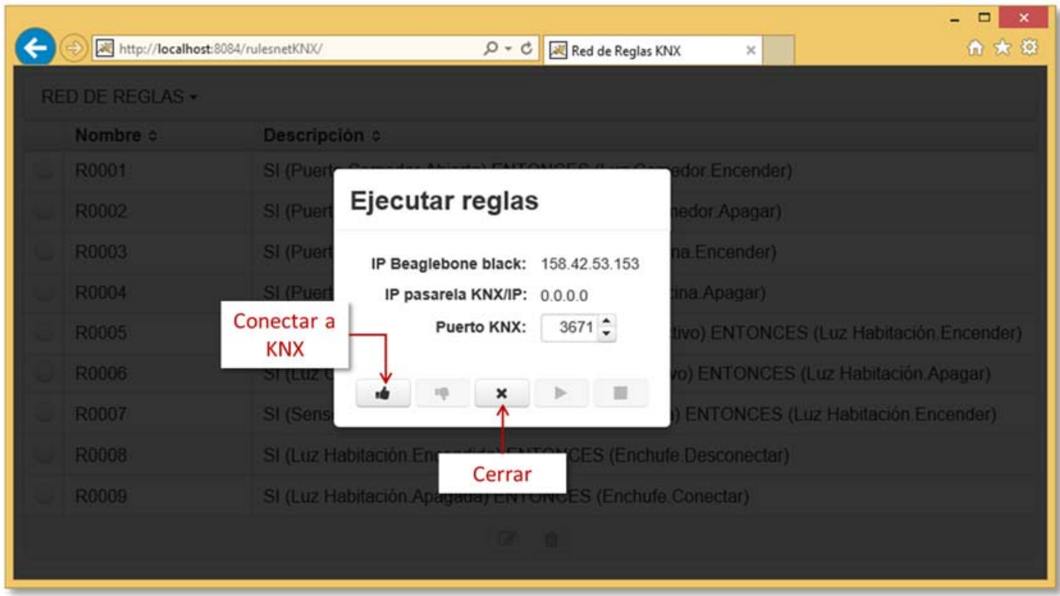


Ejecutar reglas

Para ejecutar las reglas que se hayan definido sobre la instalación KNX, se debe hacer clic sobre la opción correspondiente del menú. Está opción solo estará habilitada si existe al menos una regla definida.



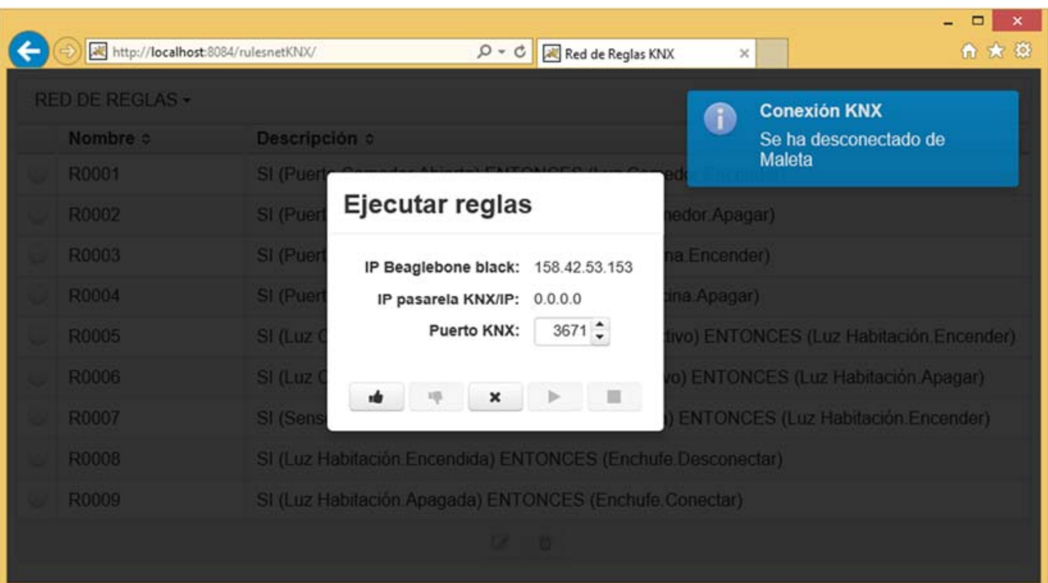
Aparecerá un cuadro de diálogo en el que se muestra la dirección IP de la *BeagleBone*, la dirección IP de la pasarela KNX/IP, el puerto empleado para la comunicación y una serie de botones de operación.



Hasta que no se establezca la conexión, el campo de la IP de la pasarela KNX/IP aparecerá como 0.0.0.0 y solo estarán habilitados los botones para conectar con KNX y cerrar el cuadro de diálogo. El usuario podrá determinar el puerto con el que realizar la conexión, el cual deberá coincidir con el empleado por la pasarela KNX/IP.

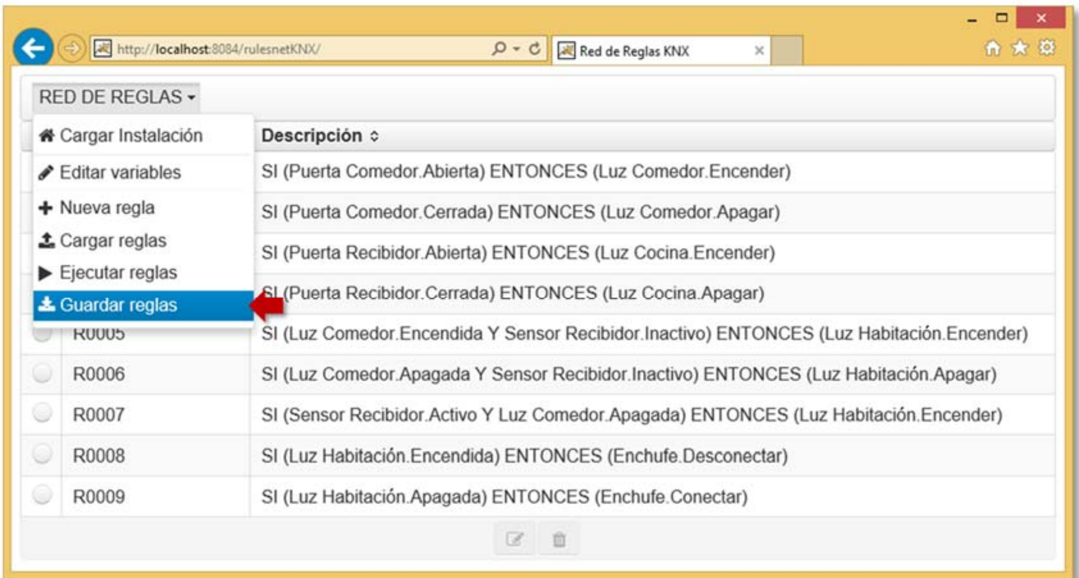
Una vez se haya establecido la conexión, se habilitará el botón para poder iniciar la ejecución de las reglas. Tras pulsar dicho botón, comenzará la ejecución de las reglas y se habilitará el botón para detener su ejecución.

Para finalizar la ejecución de las reglas, se podrá pulsar tanto el botón *Detener* como el botón *Desconectar*. Para poder cerrar el diálogo, será necesario pulsar previamente sobre el botón *Desconectar* para finalizar la conexión con la instalación KNX.



Guardar la red de reglas en un fichero

La aplicación permite guardar la red de reglas, compuesta por las variables con sus hechos y las reglas definidas por el usuario, en un fichero con formato XML. Para ello, habrá que seleccionar la opción correspondiente del menú.



A continuación, nos aparecerá un cuadro de diálogo del sistema para seleccionar el nombre del archivo y la ubicación donde queremos que sea guardado.