Programa Oficial de Postgrado en Ciencias, Tecnología y Computación
Máster en Computación
Facultad de Ciencias - Universidad de Cantabria

Implementación de la tecnología de componentes CCM en lenguaje C/C++ y sobre la plataforma Linux

Helder Ricardo de Oliveira e Silva Castro
oliveirahr@unican.es

Directores:
Julio Medina Pasaje
Grupo de Computadores y Tiempo Real
Departamento de Electrónica y Computadores.

Santander, Septiembre de 2008
Curso 2007/2008

Agradecimientos:

A José María Drake por toda su ayuda y paciencia durante todo el tiempo que llevo trabajando con él.

A Laura Barros por su gran ayuda y dedicación en ayudarme en este proyecto.

A Ángela del Barrio por su gran disponibilidad.

A mi director Julio Medina Pasaje por su orientación en la elaboración de este trabajo.

A Sole, por su inestimable paciencia y ayuda al transformar cualquier día estresante en soleado.

A mis padres, que aún estando lejos me apoyan en todo.

A todo el Grupo de Computadores y Tiempo Real porque son mucho más que compañeros, en especial a Pablo, que no sólo me lo he pasado muy bien con él, sino que también me ha ayudado mucho todo este tiempo.

Índice

Índice de figuras

1. Ámbito y objetivos

Este trabajo se encuadra dentro de la línea de investigación que tiene como objetivo el desarrollo de la tecnología de componentes ICE-CCM. En el se abordan dos aspectos: El desarrollo de esta tecnología haciendo uso del lenguaje de programación C/C++ sobre una plataforma LINUX y el desarrollo de un componente de entrada/salida que requiere el acceso al hardware del sistema a través de drivers.

1.1 Tecnología ICE-CCM (Container Component Model)

Se ha utilizado la tecnología *Container Component Model* CCM [1, 2, 3] utilizando como mecanismo de interacción entre componentes el *middleware* ICE de la empresa ZeroC [4].

El objetivo esencial de la metodología de componentes es poder construir aplicaciones ensamblando módulos software reutilizables, que han sido previamente desarrollados sin conocer las aplicaciones en las que se van a emplear. Para el diseñador de una aplicación basada en componentes, estos son elementos opacos, esto es, no tiene que modificar, y ni siquiera conocer, su código interno. A tal fin, los componentes proporcionan un modelo de comportamiento que es definido por la tecnología, y que proporciona como información reflectiva (*metadata*) todo lo que el diseñador necesita para utilizarlo e instalarlo.

Los componentes pueden haber sido desarrollados en diferentes lenguajes, estar siendo ejecutados en diferentes procesadores con diferentes sistemas operativos, y desplegados en una plataforma de ejecución distribuida. A fin de reducir la diversidad de versiones que se necesitaría para utilizar un componente con esta diversidad de condiciones, la metodología de componentes CCM hace independiente el diseño y el código de la lógica de negocio del componente de las características de la plataforma en que se ejecuta. Para ello, el código de las instancias de los componentes se descompone en dos partes:

- El código de negocio: Es la sección del componente que implementa la funcionalidad de negocio del componente. Es desarrollado por un experto del dominio de aplicación al que corresponda el componente y su diseño se realiza sin tener en consideración las características de la plataforma. A fin de poder gestionar posteriormente este código de forma opaca, el diseñador debe presentarlo en una forma normalizada que define la tecnología y ofrecer ciertas interfaces de gestión.
- El contenedor (*wrapper*): Es la sección del componente que adapta el código de negocio a la plataforma específica en la que se va a instalar y ejecutar. Es un código que es generado de forma automática mediante herramientas una vez que en el plan de despliegue de de la aplicación se hayan fijado las condiciones de instanciación y ejecución del componente.

Toda tecnología de componentes que vaya a ser utilizada para construir aplicaciones distribuidas requiere un middleware de comunicación que facilite la interacción entre los componentes instanciados en diferentes nudos de la plataforma, y así mismo lo haga independiente de los mecanismos de comunicación entre procesadores que se utilice. En este trabajo, reutiliza el middleware ICE que permite trabajar con plataformas Linux, Window y JVM (Maquina Virtual Java) y diferentes lenguajes de programación Java, C/C++, VisualBasic, Pitón, etc.

Las principales características del middleware ICE, son:

- Proporcionar una plataforma de middleware adecuada al desarrollo de aplicaciones basada en objetos en entornos distribuidos heterogéneos.
- Proporcionar un conjunto completo de características que apoyan el desarrollo de aplicaciones distribuidas para una amplia variedad de dominios.
- Evitar la innecesaria complejidad, tornando la plataforma fácil de aprender y de utilizar.

- Proporcionar una implementación que es eficiente en ancho de banda de red, el uso de memoria y lo CPU *overhead*
- Proporcionar una aplicación que tenga incorporada la seguridad, por lo que es idónea para su uso en redes públicas más inseguras.

ICE para C++ suministra la potencia/funcionalidad requerida para el rendimiento de los sistemas distribuidos, sin obligar a los usuarios a tratar con un API de middleware difícil de conocer. ICE para C++ permite a los programadores ser rápidamente productivos y reduce el desarrollo y los costos de mantenimiento al eliminar la necesidad de la gestión de memoria. La productividad es aún mayor mediante el uso de la Biblioteca de plantillas estándar para tipos de datos, eliminación de fugas de memoria con la recolección de basura, y la disponibilidad de una amplia variedad de herramientas útiles y servicios, incluida la documentación, seguridad, el despliegue, integración y una base de datos.

1.2 Operaciones de entrada y salida

El dominio de aplicación del componente que se desarrolla en este proyecto es la gestión de tarjetas de adquisición de datos que permiten leer y establecer señales de entrada y salida de tipo analógico y digital. La implementación del código se ha realizado para una tarjeta comercial PCI-9111, y así mismo se han utilizado los drivers genéricos de software libre Comedi. Los drivers Comedi son explicados en una próxima sección de esta memoria.

Las tarjetas suportadas por Comedi tienen uno o más de las siguientes señales: entrada analógica, salida analógica, entrada digital, salida digital, contador entrada, contador de salida, impulso de entrada, impulso de salida:

- Señales **Digitales** son conceptualmente simples, y no necesitan grandes configuraciones: el número de canales, su organización, y su dirección de entrada y salida.
- Señales **Analógicas** son algo más complicadas. Típicamente, una adquisición de un canal analógico puede ser programada para generar o leer tensiones entre un rango de valores mínimos y máximos (por ejemplo -10v y 10v), la tarjeta puede programarse para que automáticamente muestre un conjunto de canales en una orden predefinida o una rutina de interrupción para descargar los datos en una área de memoria predefinida.
- Señales de gestión de **Pulsos** (contadores, temporizadores, encoders, etc.) son conceptualmente un poco más complejos que entradas e salidas digitales, en realidad es añadida algunos requisitos temporales a la señal. Comedi tiene únicamente un número limitado de drivers para este tipo de señales.

1.3 Drivers en Linux

Un driver (dispositivo) es un módulo *software* que hace de interfaz de un recurso *hardware*: una impresora, una tarjeta de sonido, etc. Ofrece un API que proporciona operaciones para configurar el hardware y para acceder a su funcionalidad.

Los controladores de los dispositivos (*device drivers*) son a menudo escritos por programadores, que sólo tienen su aplicación específica en mente, especialmente en aplicaciones de tiempo real. Por ejemplo, alguien puede escribir un *driver* para el puerto paralelo, a fin de usarlo para generar pulsos que controlen un motor paso a paso. Este enfoque conduce a menudo a controladores de *drivers* que dependen demasiado de una aplicación específica, y no son lo suficientemente generales como para ser reutilizados para otras aplicaciones. Una regla de oro para el programador de *drivers* es independizar el mecanismo y la forma de uso.

- **Mecanismo**: El mecanismo de la interfaz del dispositivo, es una fiel representación de la funcionalidad del dispositivo, independientemente de qué parte de la funcionalidad de una aplicación se va a utilizar.

- **Forma de uso**: Una vez que un controlador ofrece la interfaz de software para el mecanismo del driver, un programador puede utilizar este mecanismo de interfaz para usar el dispositivo de una forma particular. Es decir, algunas de las estructuras de datos ofrecidas por el mecanismo se interpretan en unidades físicas específicas, o algunas de ellas son tomadas en conjunto porque esta composición es relevante para la aplicación. Por ejemplo, una tarjeta de salida analógica puede utilizarse para generar tensiones que son las entradas para los dispositivos electrónicos de los motores de un robot; estas tensiones pueden interpretarse como parámetros para la velocidad deseada de estos motores, y seis de ellos son considerados en conjunto para dirigir un robot con seis grados de libertad. Algunos de los otros productos del mismo dispositivo físico puede ser usado por otro programa de aplicación, por ejemplo para generar una onda sinusoidal que impulsa una vibración.

Desde el punto de vista de los desarrolladores de aplicaciones, hay muchas razones para dar la bienvenida a la normalización de la API y la estructura arquitectónica de software de las tarjetas de adquisición de datos (DAQ):

- **API**: dispositivos que ofrecen funcionalidades similares, deben tener la misma interfaz de software, y sus diferencias deben ser enfrentadas con la parametrización de las interfaces, no por cambiar la interfaz para cada nuevo producto en la familia. Sin embargo, los fabricantes de DAQ nunca han podido llegar a ese esfuerzo de normalización.

- **Estructura arquitectónica**: muchas interfaces electrónicas tienen más de una capa de funcionalidad entre el hardware y el sistema operativo y el controlador de dispositivo de código debe reflejar este hecho. Por ejemplo, muchas tarjetas de interfaz utilizan el mismo controlador PCI (Interconexión de Componentes Periféricos), o usar el puerto paralelo como producto de intermedio para conectar con el dispositivo de hardware. Por lo tanto, a nivel inferior los controladores de dispositivos para estos chips PCI y puertos permiten una mayor modularidad. Encontrar semejanzas genéricas y la estructura entre las diferentes tarjetas ayuda en el desarrollo de controladores de dispositivo más rápido y con mejor documentación.

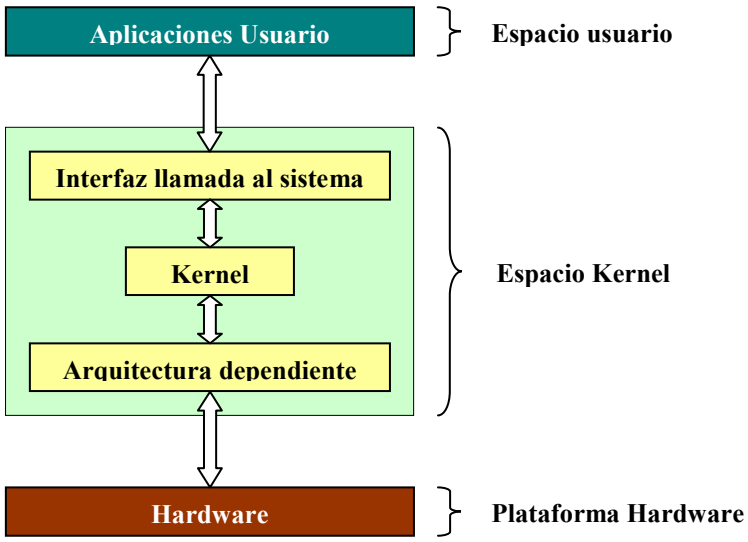

Figura 1-1: Espacio usuario y Espacio Kernel

Como se demuestra en la figura anterior en el caso del sistema operativo Linux, los programadores de controladores de dispositivos tienen que tener en mente los siguientes puntos:

- **Espacio Kernel vs espacio de usuario**: El sistema operativo Linux tiene dos niveles que requieren básicamente diferentes enfoques de programación. Sólo procesos privilegiados pueden correr en el núcleo, donde tienen acceso a todo el hardware y para todas las estructuras de datos del núcleo. Programas de aplicación normales pueden ejecutar sus procesos sólo en el espacio de usuario, donde estos procesos son protegidos unos de otros, y del acceso directo al hardware y datos críticos del sistema operativo, estos programas de espacio de usuario ejecutan una grande parte de su funcionamiento de sistema a través de llamadas al sistema.
- **Ficheros de dispositivo o sistema de ficheros de dispositivo:** Los usuarios que escriben una aplicación para un dispositivo determinado, deben vincular su aplicación a ese dispositivo del controlador de dispositivo. Parte de este controlador de dispositivo, no obstante, se ejecuta en el espacio del *Kernel*, y el software de aplicación en el espacio de usuario. Así, el sistema operativo proporciona una interfaz entre ambos. En Linux o Unix, estas interfaces son en forma de "archivos" en el directorio /dev (núcleos 2.2.x o anteriores) o el directorio /devfs (iguales o superiores a *kernels* 2.4.x). Cada dispositivo suportado en el *Kernel* tiene un fichero de dispositivo representante el espacio de usuario, y su funcionalidad se puede acceder con operaciones de: abrir, cerrar, leer, escribir, y ioctl.
- **Interface /proc**: Linux (y algunos otros sistemas operativos UNIX) ofrecen una archivo de interfaz para los dispositivos conectados (y otros sistemas operativos relacionados con la información) a través de los directorios /proc. Estos archivos no existen realmente, sino que ofrece una interfaz familiar para los usuarios, con las que puede inspeccionar el estado actual de cada dispositivo
- Tiempo real frente a no tiempo real. Si el dispositivo se va a utilizar en aplicaciones en RTLinux / GPL o RTAI, hay algunos requisitos adicionales, ya que no todas las llamadas al sistema están disponibles en el *Kernel* en tiempo real de los sistemas operativos RTLinux / GPL o RTAI. Las API´s de RTAI y RTLinux difieren de distintas maneras, por lo que los desarrolladores tengan que gastar una gran cantidad de esfuerzos para hacer genéricos las envolturas para las primitivas RTOS: temporizadores, asignación de memoria, el registro de manipuladores de interrupción, etc.

1.4 Objetivos del trabajo

El objetivo concreto de este trabajo ha sido el desarrollo del componente **da_IOCard** dentro de la tecnología ICE-CCM, cuya funcionalidad es la gestión de una tarjeta de adquisición de datos con capacidad de leer y establecer señales hardware externas analógicas y digitales. Características relevantes de este componente son:

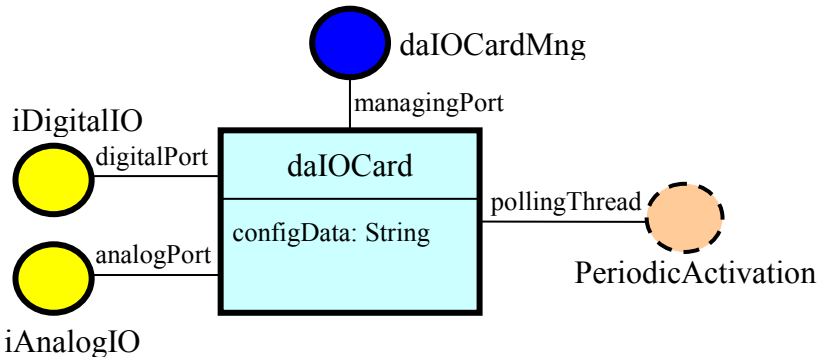

Figura 1-2: Componente daIOCard

- Es un componente formulado independientemente de la tarjeta DAQ que se utilice.

- Ofrece su funcionalidad a través de dos servicios caracterizados por las interfaces iDigitalIO para la entrada y salida de señales digitales, e iAnalogIO para la entrada y salida de señales analógicas.
- Permite quedar a la espera de eventos externos: Bien que una señal digital de entrada tome un determinado estado, o que una señal analógica traspase un determinado umbral de tensión.
- Permite la generación de pulsos singulares y señales periódicas (*blinking*) en las salidas digitales.

El componente es activo, esto es, dispone de un thread que ejecuta una tarea de pollling periódica, que se utiliza para detectar los eventos y generar las señales de salida. La frecuencia de esta tarea periódica es configurable.

1. De este componente se ha diseñado la implementación correspondiente a la tarjeta PCI- 9111. Esta tarjeta es de muy bajo coste y dispone de líneas de entrada y salida digital, entradas analógicas basadas en un multiplexor y un único conversor A/D de 12 bits, y un único canal analógico de salida basado en un conversor D/A de 12 bits de resolución y otros recursos como timer, memoria interna, etc. que no se han utilizado en este trabajo.

Con la realización de este componente se han cubierto tres aspectos que se requerían estudiar en varios proyectos de investigación:

1.1. Implementar la tecnología ICE-CCM para componentes desarrollados en el lenguaje C/C++. El uso del lenguaje C tiene especial interés en todos aquellos dominios en los que se requiere el acceso en bajo nivel al hardare del sistema, o a los recursos del sistema operativo en plataformas UNIX. La implementación de la tecnología requiere definir la estructura del framework CCM, y establecer los patrones en lenguaje C/C++ con los que se realiza el código de sus elementos. En el futuro, el código del contenedor del componente debe ser generado mediante herramientas automáticas, pero un paso previo al desarrollo de las herramientas, es haberlo formulado a mano y verificado de forma completa. Esta es la función que cubrimos en este trabajo.

1.2. Probar los *drivers* de software abierto Comedi. Una dificultad que se ha encontrado cuando se utilizan tarjetas hardwares comerciales, es conseguir *drivers* adecuados para la plataforma en la que se instalan. Obviamente cuando se plantea la necesidad de una de estas tarjetas, se considera como requisito la disponibilidad de *drivers* adecuados a la plataforma en uso. El problema que se plantea es que cuando la plataforma evoluciona, aunque sólo sea en la versión, el *driver* ya no sirve, y además el *driver* que se necesita no esta disponible hasta que pasa un largo tiempo. En este proyecto se han probado los *drivers* de la familia **Comedi [5]**, que pertenecen un proyecto de software libre y abierto y que facilita su evolución con la plataforma.

1.3. Disponer de un componente de entrada/salida para señales digitales y analógicas. Los componente de entrada y salida ya sean de señales digitales/analógicas es un componente esencial en el diseño de aplicaciones de tiempo real, ya que constituyen el elemento básico de conexión con el entorno.

2. Tarjeta PCI-9111 y drivers Comedi

2.1 Tarjeta PCI-9111 Arquitectura y funcionalidad

La tarjeta de adquisición utilizada fue la tarjeta PCI9111 de la empresa ADLink Technology Inc.

Es una avanzada tarjeta de adquisición de datos basada en la arquitectura de 32-bit PCI Bus. Alto rendimiento hace con que esta tarjeta sea ideal para el registro de datos y analice de señales en aplicaciones médicas, control de procesos, controlador de seguridad, comunicaciones, teste de productos, generación de pulsos y de ondas.

Esta tarjeta tiene las siguientes características:

- 16 líneas digitales de entrada.
- 16 líneas digitales de salida.
- 8 líneas digitales programables como entradas o como salida.
- 16 líneas analógicas de entrada leíbles mediante un conversor A/D de 12 bits de resolución.
- Sistema de muestreo interno con buffer de 2Kbyte, frecuencia de muestreo de hasta 10K muestras/s y diferentes mecanismos de disparo.
- Una línea de salida analógica con conversor D/A de 12 bits de resolución.

La tarjeta PCI-9111 presenta funcionalidades distintas para operar con los canales analógicos e digitales.

La funcionalidad para adquirir señales analógicas/digitales se presenta a seguir:

- Adquisición de señales de entrada analógicos:
  Como fue descrito en las características de la tarjeta el canal analógico presenta 16 canales de entrada. La señal analógica puede ser convertida para valores digitales pelo conversor A/D. Para evitar bucles de tierra y una más precisa medición de los valores A/D, es necesario comprender el tipo de la fuente de señal, para esto hay que tener en atención que una o más fuente de señal son relativas a una única conexión de tierra.
- Establecer señal de salida analógica
  La tarjeta dispone de una única salida analógica. El rango de la señal puede ser unipolar o bipolar.
- Adquisición/Establecer señales digitales
  Disponemos de 16 entradas digitales e 16 salidas digitales disponibles por el conector. La señal digital I/O es completamente compatible con TTL/DTL.

Para su gestión se ha desarrollado un componente concreto, con una capacidad de operar sobre el middleware de soporte ICE:

Componente **daIOCard**: Componente operando sobre la librería C del driver Comedi para operar en el entorno LINUX y utilizando el middleware ICE de ZeroC.

2.2 Drivers Comedi

Comedi es un proyecto de software libre para desarrollar drivers, herramientas, e librerías para varios tipos de adquisición de datos: lectura y escritura de canales analógicos, leer y escribir señales digitales, contajes de pulsos y frecuencia, generar frecuencias, leer encoders, etc. La fuente de código es distribuida en dos paquetes, Comedi y Comedilib, y proporciona varios módulos Kernel de Linux y una libraría de espacio de usuario:

- **Comedi** es una colección de drivers para una variedad común de adquisición de datos de *plug-in cards* (los cuales son denominados por "dispositivos" en terminología Comedi). Los dispositivos son implementados como una combinación

de un modulo de núcleo único do Kernel (denominado "Comedi") ofertando funcionalidades comunes, y individuales para cada dispositivo.

- **Comedilib** es un paquete distribuido separadamente conteniendo una libraría de espacio de usuario que oferta una interface amigable para los dispositivos Comedi.
- **Kcomedilib** es un modulo de Linux Kernel que oferta la misma interface que comedilib en el espacio del Kernel, y adecuado para tareas de tiempo real. E efectivamente una libraria de Kernel para utilizar tareas de tiempo real.

Comedi trabaja con Kernels standard de Linux, y también con sus librarías de extensión para tiempo real RTAI y RTLinux/GPL.

A continuación explicamos como se instala un *driver* Comedi en LINUX.

Para compilar los módulos de Comedi, necesitará tener correctamente configurado un *kernel* de Linux. La mejor manera de obtener una es descargar un archivo tar de kernel.org y compilar su propio *kernel*. Comedi deberá trabajar con la mayoría de versiones de *kernel* LINUX 2.2, 2.4, y 2.6. Versiones de *kernels* 2.6.x superiores a 2.6.6 no son admitidas. Para versiones de 2.0.3x ya no es mantenido activamente.

También se puede compilar un *kernel* que coincida con lo que está actualmente trabajando, si usted tiene el archivo de configuración (en la distribución Debian los archivos de configuración de los paquetes kernel-image se instalan en el directorio /boot). Se necesitan permisos de súper utilizador para escribir en el kernel de LINUX.

1. Obtenga una copia de las fuentes del kernel que coincidan con el núcleo que esta trabajando. Desempaquetar (/usr/src/linux) y copiar el archivo de configuración del kernel que esta utilizando (/boot/config-<<kernel_version>>) a (/usr/src/linux/.config).
2. Es posible que tenga que editar el archivo "Makefile". En la parte superior del Makefile, la variable EXTRAVERSION esta definida. Si es necesario, cambie para que coincida con su kernel (por ejemplo, si el comando 'uname-r "produce" 2.4.16-386 "y luego su EXTRAVERSION debe ser colocada como "EXTRAVERSION =- 386 '.
3. Ejecutar "make oldconfig" en el directorio de sus fuentes para importar las configuración del kernel actual.
4. Ejecutar "make dep" (para núcleos 2.6, hacer un "make modules_prepare" en lugar de un completo "make") en el directorio de fuentes del kernel.

Con el *kernel* configurado, podemos partir para la instalación de Comedi.

Pasos a seguir para la instalación:

1. **Configurar**: configurar con "./Configure". Si el *script* de configuración no existe, puede ser generado mediante la ejecución "./ autogen.sh" . El autoconf, automake, autoheader, etc son herramientas necesarias para generar el configure script (recomendado automake versión> = 1.7).
2. **Compilar**: comando "make".
3. **Instalar**: En modo de súper utilizador instalar usando "make install". Esto instala los archivos:
    a. /lib/modules/<<kernel version>>/comedi/comedi.o
    b. /lib/modules/<<kernel version>>/comedi/kcomedilib.o
    c. /lib/modules/<<kernel version>>/comedi/<<driver files>>.o

Se tiene la necesidad de crear ficheros de dispositivo para acceder al hardware desde un proceso de utilizador. Estos pueden ser creados con el comando "make dev". Los siguientes archivos se crearán:

- /dev/comedi0 ... /dev/comedi15

Para utilizar Comedi, el *driver* y el módulo Comedi deben ser cargados en el *kernel*. Esto se hace invocando el comando:

- /sbin/modprobe <<driver>>

En nuestro caso *driver* debe ser substituido por pci9111_hr.

Con el fin de configurar un módulo del *driver* para utilizar un determinado archivo (/dev/comediN) y un dispositivo en particular, es necesario utilizar el comando "comedi_config", que es parte de la distribución comedilib. Comedi_config se invoca utilizando:

- comedi_config /dev/comedi0 <device name> < lista opciones>

Para nuestro caso en concreto:

- device_name = pci9111_hr
- La lista de opciones no és necesaria una vez que trabajamos con una sola tarjeta. La lista de opciones es necesaria en caso de tener más de una tarjeta y ser necesario indicar su enderezo I/O, canales DMA, etc.

Con en kernel de LINUX configurado, instalado el *driver* de Comedi, se puede operar sobre la tarjeta PCI.

2.3 Funcionalidad y modelo de uso de las tarjetas IO

Comedi tiene funciones para la adquisición de datos para trabajar con los conjuntos de canales.

Para entender la funcionalidad y modelo de uso haremos una introducción de cómo Comedi puede organizar todo el hardware según la jerarquía genérica:

- **Canal:** el componente de hardware más bajo, representa las propiedades de un único canal de dados, por ejemplo, una entrada analógica, o una salida digital. Cada canal tiene varios parámetros, tales como, el rango de voltios, la polaridad del canal (unipolar, bipolar), un factor de conversión entre voltios e unidades físicas, el valor binario "0" y "1".
- **Sub-dispositivo:** conjunto de canales con funcionalidades idénticas que están físicamente implementados en una misma tarjeta. Por ejemplo, un conjunto de 16 salidas analógicas idénticas. Cada sub-dispositivo tiene parámetros para identificar el número del canal y el tipo de canales.
- **Dispositivo:** conjunto de sub-dispositivos que están físicamente implementados en una misma tarjeta. Como ejemplo nuestra tarjeta es un sub-dispositivo de 16 canales analógicos, otro sub-dispositivo con 16 canales digitales de salida y otro con 16 canales digitales de entrada. Cada dispositivo tiene parámetros para: identificar el fabricante, identificar el sistema operativo (sirve para discriminar entre múltiples tarjetas del mismo tipo), el numero de sub-dispositivos.

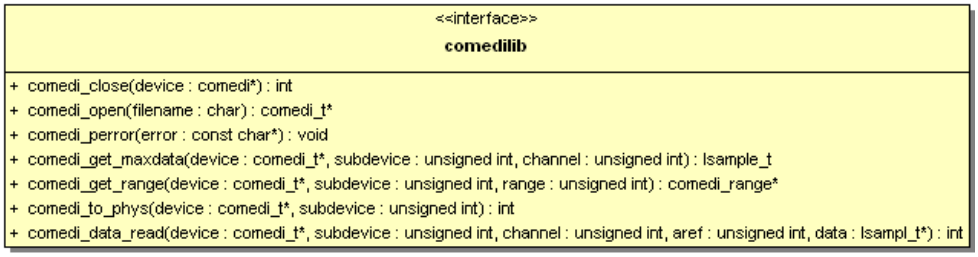

Figura 2-1: Interface comedilib

Estos son algunos de los métodos ofrecidos por la librería comedilib para accder al driver de comedi.

Comedi para hacer la adquisición de datos simple tiene llamadas a funciones para realizar sincrónicamente una sola adquisición de datos en un determinado canal: *comedi_data_read(), comedi_data_write(), comedi_dio_read(), comedi_dio_write()*. Sincrónicamente significa que el proceso de llamada bloquea hasta que la adquisición de datos este terminada.

A continuación se explica como se puede hacer la adquisición de datos simples de un canal analógico:

1. **comedi_open**: Abrir el dispositivo
2. **comedi_get_maxdata**: devuelve o valor máximo de datos permitidos para ese canal.
    a. Ejemplo: canal de 16 bits 2^16=65536
3. **comedi_get_range**: funcion que devuvelve un puntero para una estrutura que contiene informacion que ira ser utilizada para convertir valores de o para unidades fisicas (volt)
4. **comedi_data_read**: ler un dato del canal.
5. **comedi_to_phys:** converte los datos obtenidos en unidades fisicas
    a. **Ejemplo:** 65536 corresponde a 10v.

```
#include <stdio.h>
#include <comedilib.h>

int subdev = 0;
 int chan = 0;
int range = 0;
 int aref = AREF_GROUND;

const char filename[] = "/dev/comedi0";

int main(int argc, char *argv[])
{
        comedi_t *cf;
        lsampl_t data;
        int maxdata;
        double volts;
        comedi_range *cr;
        int retval;

        cf = comedi_open("/dev/comedi0");
        if(cf == NULL)
        {
                comedi_perror(filename);
                return 1;
        }
        maxdata = comedi_get_maxdata(cf, subdev, chan);
        if(maxdata == 0)
        {
                comedi_perror(filename);
                return 1;
        }
        cr = comedi_get_range(cf, subdev, chan, range);
        if(cr == NULL)
        {
                comedi_perror(filename);
                return 1;
        }
        retval = comedi_data_read(cf, subdev, chan, range, aref, &data);
        if(retval < 0)
        {
                comedi_perror(filename);
                return 1;
        }
        volts = comedi_to_phys(data, cr, maxdata);
        printf("Data %d volts: %g\n", data, volts);

        return 0;
}
```

Se pueden saber cuantos sub-dispositivos pose la tarjeta, cuantos canales están asociados a cada sub-dispositivo, el nombre de la tarjeta, etc. Para saber como proceder para obtener estos datos consultar o código en Anexo A y la pagina de referencia Comedi [5].

2.4 Capacidades y limitaciones de adquisición y configuración Comedi

El driver Comedi és un software libre en constante evolución. Siempre y cuando existe una nueva versión de Linux se puede instalar en contra de la espera de los drivers comerciales se adapten a la nueva versión.

- **Adquisición simple**: obtención de una única muestra de o de la tarjeta. Pueden ser digitales o analógicas.
- **Adquisiciones múltiples**: Por medio de uso de instrucciones, pueden ser ejecutadas múltiples adquisiciones idénticas en el mismo canal.
- **Disparo interno (*internal triggering*)**: Este evento puede por ejemplo causar que el driver inicie una conversión o pare una adquisición de datos
- **Comandos para adquisición de *streaming***: Un comando especifica una particular secuencia de adquisición de datos, en la cual consiste en un número de *scans*, e cada *scan* se compone de una serie de conversiones, que normalmente corresponden una simple adquisición A/D o D/A.

Comedi no solamente ofrece la API para acceder a las funcionalidades de la tarjeta, pero también para preguntar las capacidades de los *drivers* instalados. Es decir, un proceso de usuario puede saber en **tiempo de ejecución que canales están disponibles, y cuales son sus parámetros** (rango, dirección de entrada/salida).

Como limitaciones el driver Comedi no permite operar sobre los canales extendidos digitales de entrada/salida.
Tiene entre otras limitaciones:

- **Estados múltiples de entrada digital**: No son permitidos cambios por transacciones de flanco de una o varias entradas para modificar una salida digital o un acumulador.
- **Configuración analógica de filtrado**: Algunos dispositivos tienen la capacidad para añadieren ruido a la medida. Este ruido adicional puede ser eliminado para obtener una más precisa medición.
- **Generación analógica de una onda:** no permite la generación de ondas.

Capacidades	Limitaciones
Saber en tiempo de ejecución los canales disponibles	Operar sobre los canales extendidos digitales de entrada/salida
Open Source	Estados múltiples de entrada digital
Adquisición simple	Configuración analógica de filtrado
Adquisiciones múltiples	Generación analógica de una onda
Disparo interno	
Comandos para adquisición de *streaming*	

Tabla: Comparación entre capacidades e limitaciones del driver

3. Especificación del componente daIOCard

Por especificación de un componente entendemos la información funcional y no funcional que describe el comportamiento de un componente visto desde fuera, esto es, desde el punto de vista del diseñador de la aplicación que lo usa. A partir de ella, el diseñador tiene que poder:

1. Decidir si el componente es útil para ser utilizado en la aplicación que está construyendo por su funcionalidad y por el comportamiento no lineal que ofrece.
2. Saber como tiene que ensamblar sus instancias en la aplicación configurándolas, y conectándolas con las instancias de otros componentes que necesita.
3. Instalar la instancia en el nudo procesador en que se vaya a ejecutar y lanzar su ejecución.

La especificación de los componentes, es establecida por el agente *Especificador* que es un experto del dominio de aplicación al que pertenece el componente. Para ello, el *especificador* analiza aplicaciones que previamente se han desarrollado dentro de un determinado dominio y decide que el disponer un módulo con una determinada funcionalidad y características no funcionales puede ser de interés para el diseño de futuras aplicaciones. Desde un punto de vista estricto, la especificación de un componente por el *especificador* no tiene que estar motivada por una aplicación concreta, sino por la experiencia que se tiene del dominio de aplicación.

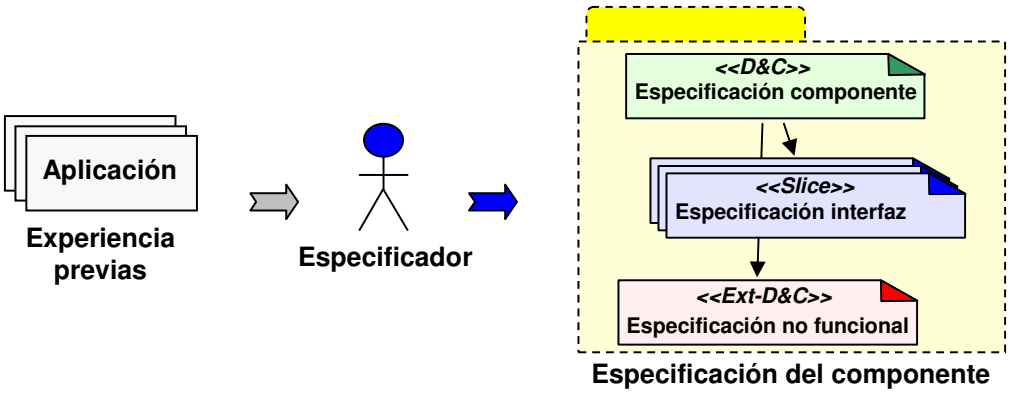

Figura 3-1: Origen de la especificación de un componente

La interfaz de un componente contiene:

- La descripción de la funcionalidad del componente, definida a través de la descripción de las interfaces de los puertos que el componente ofrece (*facets*), de las interfaces de los puertos que el componente requiere (*receptacles*). Las interfaces se describen utilizando un lenguaje específico para ello como es Idl de CORBA , o Slice de ICE. Estos lenguajes son independiente de los lenguajes de programación (C, C++, Java, Ada, etc.) y su aspecto más importante es que ofrece transductores bien definidos a cualquiera de ellos.
- La descripción de los elementos que ofrece y requiere para que el modelo de no funcional del componente pueda complementarse con el modelo no funcional de otros componentes.
- La descripción de la especificación del componente propuesta por el estándar de D&C de OMG que reúne en un documento XML formalizado por el W3C Schema ComponentDataModel.xsd. En él se incluye toda la información que puede requerir una herramienta que procese la especificación del componente. Incluye los puertos (facetas y receptáculos del componente, la referencia de las especificaciones funcionales de las

interfaces que implementan cada uno de estos puertos, los parámetros de configuración del componente y la referencia al modelo de comportamiento no funcional del componente.

3.1 Interfaces iDigitalIO y iAnalogIO

La funcionalidad de los puertos que ofrece (facetas) a los clientes y de los puertos de otros componentes servidores que requiere para operar, se describe a través de las interfaces. En nuestro caso, cada interfaz se describe en el lenguaje Slice. En la descripción Slice se incluyen las operaciones que son implementadas, incluyendo para cada una de ellas el valor que retorna, los parámetros que tiene definidos, las excepciones que puede lanzar y los cualificadores que caracterizan la forma de ejecución de su invocación.

Todas las implementaciones que existan del componente deben implementar las operaciones descritas en sus interfaces.

A continuación se describen las funcionalidades de las interfaces definidas para el componente daIOAnalog que se ha desarrollado en este trabajo:

3.1.1 Definición de la interfaz iDigitalIO

La interfaz iDigitalIO ofrece servicios para la gestión de líneas digitales de entradas y salida. En particular ofrece funciones para:

- Leer el estado de líneas digitales de entrada.
- Establecer el estado de líneas digitales de salida.
- Permanecer a la espera de que una línea digital de entrada alcance un estado o realice una determinada transición.
- Generar señales con forma de pulsos o cuadradas periódicas en líneas digitales de salida.

Como se muestra en el diagrama de clases de la figura 3-2, la interfaz iDigitalIO declara las siguientes operaciones:

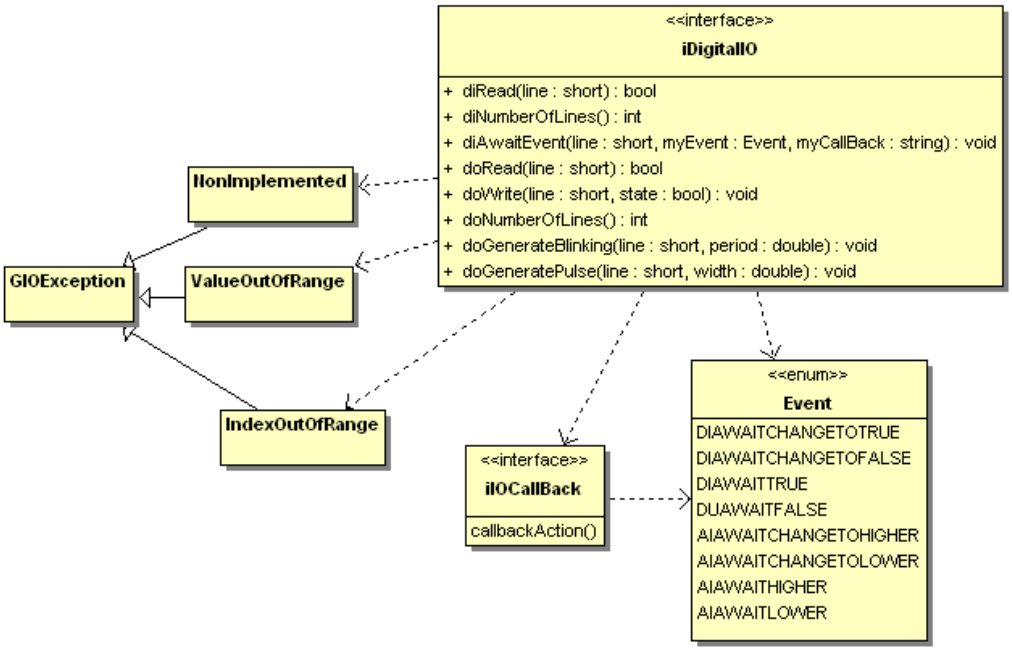

Figura 3-2: Diagrama de clases UML de la interface iDigitalIO

- ***diNumberOfLines():int*** => Retorna el número de líneas de entrada de que dispone la tarjeta.
- ***diRead(line:short):bool*** **=>** Retorna el estado de la línea digital de entrada *line*.
- ***doNumberOfLines():int*** **=>** Retorna el número de líneas de salida de que dispone la tarjeta.
- ***doWrite(line: short, state: boolean):void*** **=>** Establece el estado de la línea digital de salida con el valor *state*.
- ***doRead(line: short):bool*** **=>** Retorna el estado actual de la línea.
- ***diAwaitEvent(line: short, myEvent: Event, myCallback: string):void*** **=>** Invoca la operación callBackAction del objeto que se ha pasado como callback, cuando en la línea digital de entrada *line* se ha alcanzado el estado que corresponde al evento especificado en *event*.
- ***doGeneratePulse(line: short, width: double, state:Boolean):void*** **=>** Genera en la línea digital de salida *line*, un pulso de duración en segundos especificada en width y de polaridad *state*.
- ***doGenerateBlinking(line: short, period: double):void*** => Genera en la línea digital de salida especificada, una señal cuadrada con el periodo en segundos establecido en period.

Estas operaciones pueden elevar las excepciones:

- ***NonImplemented***: Si en el correspondiente componente concreto, no se ha implementado la operación.
- ***IndexOutOfRange***: Si la línea que se referencia en la operación no existe en la tarjeta.
- ***ValueOutOfRange***: Si el valor de tiempo que se establece en las operaciones de generación está fuera del rango admitido por la implementación.

El tipo enumerado ***Event***: Representa los tipos de eventos que pueden ser designados en la operación. Los valores enumerados que se definen son:

- DIAWAITCHANGETOTRUE: El evento que se pretende producir es un flanco de transición de False a True.
- DIAWAITCHANGETOFALSE: El evento que se pretende producir es el flanco de transición de True a False.
- DIAWAITTOTRUE: El evento que se pretende producir es que el valor de la línea sea TRUE.
- DIAWAITTOFALSE: El evento que se pretende producir es que el valor de la línea sea FALSE.

3.1.2 Definición de la interfaz iAnalogIO

La interfaz iAnalogIO proporciona recursos para gestionar y controlar una tarjeta de adquisición analógica. En particular permite:

- Leer la tensión de líneas analógicas de entrada a través de un conversor A/D.
- Establecer la tensión de una línea analógica de salida a través de un conversor D/A.
- Permanecer a la espera de que una línea analógica de entrada alcance un estado.

En el diagrama de clases de la figura 3-3, se muestran los elementos que constituyen la de aclaración de la interfaz iAnalogIO. Las operaciones que declara son:

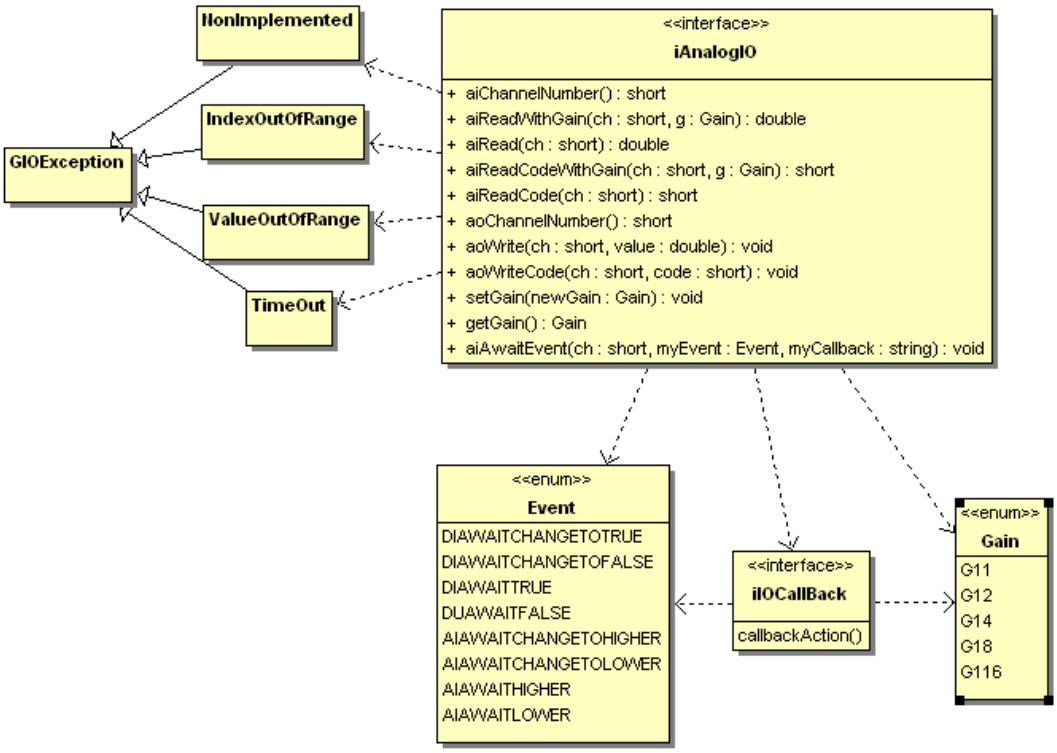

Figura 3-3: Diagrama de clases UML de la interface iAnalogIO

- ***aiChannelNumber():int*** => Retorna el número de canales analógicos de entrada que son gestionadas por la tarjeta
- ***aiReadWithGain(ch: short, g: Gain):double*** => Retorna el valor del canal en unidades físicas (volts) con la ganancia especificada.
- ***aiRead(ch: short):double*** => Retorna el valor del canal en unidades físicas.
- ***aiReadCodeWithGain(ch: short, g: gain):short*** => Retorna el valor del canal en unidades binarias con la ganancia especificada.
- ***aiReadCode(ch: short): short*** => Retorna el valor del canal en unidades binarias.
- ***aoChannelNumber( ): short*** =>Retorna el número de canales analógicos de salida que son gestionadas por la tarjeta.
- ***aoWrite(ch: short, value: double):void*** => Establece en el canal de salida el valor *value* en unidades físicas.
- ***aoWriteCode(ch: short, code: short):void*** => Establece en el canal de salida el valor *value* en unidades binarias.
- ***setGain(gain:Gain):void*** => Establece la ganancia que se aplica a la adquisición en el futuro para leer los canales de entrada. Al establecer la ganancia, se define el rango de la señal de entrada que se lee, y el error de cuantización que se introduce.
- ***getGain(): Gain*** => Retorna la ganancia efectiva que se encuentra establecida en la tarjeta. No tiene que coincidir exactamente la que fue establecida, ya que las ganancias de las tarjetas suelen estar cuantificadas a valores múltiplos de 2 (1,2,4,8,16, 32, …).
- ***aiAwaitEvent(ch: short, myEvent: Event, myCallback: string)*** => Invoca la operación callBackAction del objeto que se ha pasado como callback, cuando en el canal analógico de entrada *ch* se ha geneado el valor al que corresponde el evento especificado en *event*.

Estas operaciones pueden elevar las excepciones:

- Excepción ***NonImplemented***: Si en el componente concreto que implementa la interfaz, no se ha implementado esta operación.

- Excepción ***IndexOutOfRange***: Si la línea que se referencia en la operación no existe en la tarjeta.
- Excepción ***ValueOutOfRange***: Si el valor o el código que se da para establecer en la salida está fuera del rango del conversor D/A.
- Excepción ***TimeOut***: Si transcurre un determinado tiempo definido en cada implementación, en la invocación de conversión al conversor A/D.

Tipo enumerado ***Event***: Representa los tipos de eventos que pueden ser designados en la operación. Los valores enumerados que se definen son:

- AIAWAITCHANGETOHIGHER: El evento que se pretende producir es una transacción de un valor inferior al umbral especificado para un valor superior al mismo umbral (threshold).
- AIAWAITCHANGETOLOWER: El evento que se pretende producir es una transacción de un valor superior al umbral especificado para un valor inferior al mismo umbral.
- AIWAITTOLOWER: El evento que se pretende producir es que valor superior en la línea sea inferior superior al valor de umbral
- AIWAITTOHIGHER: El evento que se pretende producir es que valor superior en la línea sea superior al valor de umbral

3.2 Declaración formal Slice de las interfaces iDigitalIO y iAnalogIO

El fichero io.ice que se muestra en la tabla siguiente, contiene la declaración formal en lenguaje Slice de las diferentes interfaces que se han definido en el dominio io. Características de esta descripción son:

- El módulo Slice se identifica con el dominio que en este caso es io.
- Se declaran las excepciones que se definen en las interfaces del dominio. Se ha definido una excepción de alto nivel para el dominio GIOException, y de ella se han derivado las excepciones concretas que lanzan las operaciones.
- Se han declarado los dos tipos enumerados Events y Gain definidos en el dominio.
- Se han declarados las tres interfaces iDigitalIO, iAnalogIO y iIOCallback. Cada una de ellas incluye las operaciones que tienen definidas.

```
/*****************************************************************************
* \Description: The io model has 4 interfaces : iDigitalIO, iAnalogIO,  iConfiguration & iIOCallback
* \Autores:  Helder Castro, J.M. Drake
* \Version 2/9/2008
*****************************************************************************/
module io{
/**
* GIOException Abstract Class wich is the base of all the excepctions defined in the module
*/
        exception GIOException{string mssg;};
   /**
    * Derived class from GIOException
    */
        exception IndexOutOfRange extends GIOException{string objectName; string operName; string indexName;};
        exception NonImplemented extends GIOException{string objectName; string operName;};
        exception ValueOutOfRange extends GIOException { string objectName;  string operName; string valueName;};
        exception UnKnownCard extends GIOException{int numCard;};
        exception TimeOut extends GIOException{string objectName;  string operName;};
   /**
    * Enumeration of the Events an Gains
    */
    /**
        enum Event {
                DIAWAITCHANGETOTRUE,
                DIAWAITCHANGETOFALSE,
                DIAWAITOTRUE,
                DIAWAITOFALSE,
```

```
                    AIAWAITCHANGETOHIGHER,
                    AIAWAITCHANGETOLOWER,
                    AIAWAITHIGHER,
                    AIAWAITLOWER,
            };
            enum Gain {G11, G12, G14, G18, G116};

    interface iIOCallback{
            void callbackAction (string caller, Event myEvent,short line);
    };

    /**
     * Interfaces iDigitalIO y iAnalogIO
     */
     interface iDigitalIO {
            bool diRead(short line) throws IndexOutOfRange;
            short diNumberOfLines();
            short doNumberOfLines();
            void doWrite(short line, bool state) throws IndexOutOfRange;
            bool doRead(short line)throws IndexOutOfRange;
            void diAwaitEvent( short line, Event myEvent, iIOCallback* myCallback)
                    throws IndexOutOfRange,ValueOutOfRange;
            void doGeneratePulse(short line, double width, bool state)
                    throws IndexOutOfRange,ValueOutOfRange;
            void doGenerateBlinking(short line,double period)
                    throws IndexOutOfRange,ValueOutOfRange;
    };

    interface iAnalogIO {
            short aiChannelNumber() throws NonImplemented;
            double aiReadWithGain( short ch,  Gain g) throws ValueOutOfRange;
            double aiRead( short ch)throws ValueOutOfRange;
            short aiReadCodeWithGain( short ch, Gain g) throws ValueOutOfRange;
            short aiReadCode(short ch) throws ValueOutOfRange;
            short aoChannelNumber();
            void aoWrite(short ch, double value) throws IndexOutOfRange,ValueOutOfRange;
            void aoWriteCode(short ch, short codigo) throws IndexOutOfRange,ValueOutOfRange;
            void setGain(Gain newGain);
            Gain getGain();
            void aiAwaitEvent( short line, Event myEvent,double threshold, iIOCallback* myCallback)
                    throws IndexOutOfRange,ValueOutOfRange;
    };
};
```

Fichero: io.ice

3.3 Especificación del componente daIOCard

A continuación describimos la especificación del componente daIOCard. Este es un componente que gestiona una tarjeta de adquisición de entrada y salida con capacidad de leer y establecer un conjunto de líneas de entrada y salida digitales, y así mismo leer y establecer un conjunto de canales de entrada y salida analógicas, a través de los correspondientes conversores A/D y D/A. .

La especificación de un componente describe su funcionalidad externa que es común a todas las implementaciones del componente. un componente se define mediante su interfaz externa. Esta incluye:

- Tipos del dominio: En nuestro componente daIOCard utilizamos los tipos enumerados Event y Gain. Event define para el exterior que tipos de eventos pueden ser invocados. Gain define las ganancias que serán aplicadas.
- Interfaz de gestión (da_ioCard_mng): Constituye la interfaz a través del que el contenedor gestiona el ciclo de vida del componente. Representa la funcionalidad de gestión que se requiere en cualquier componente C++-CCM.
    - Funciones de acceso a las facetas del componente:
        - get_analogPort: Implementa la interfaz io::iAnalogIO y ofrece servicios para leer la tensión de las líneas analógicas de entrada y establecer la tensión de las líneas analógicas de salida

- get_digitalPort: Implementa la interfaz io::iDigitalIO y ofrece servicios para leer el estado de las lineas digitales de entrada y establecer el estado en las líneas digitales de salida

- Operaciones de configuración: set_confData.
- Operaciones de gestión del ciclo de vida del componente: activate(), passivate() y remove(): se utilizan para controlar el ciclo de vida del componente.

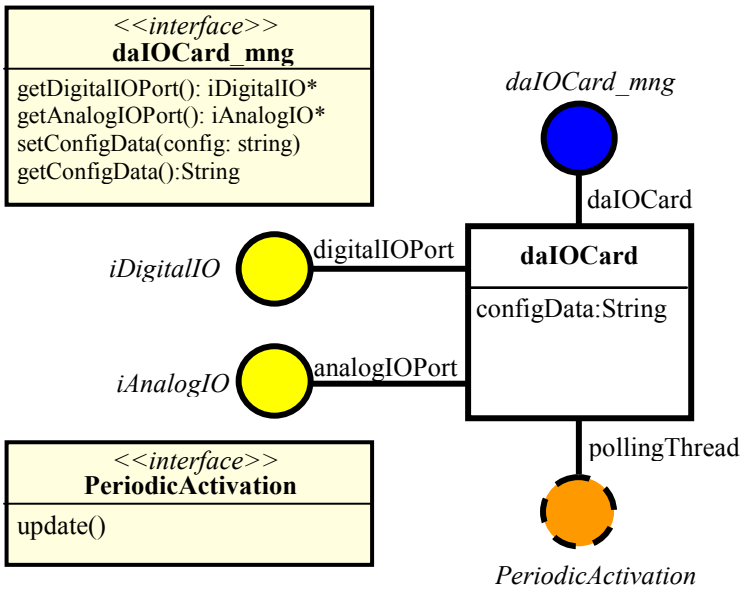

Figura 3-4: Elementos que definen la interfaz del componente daIOCard

Las interface iDigitalIO y iAnalogIO estan definidas en los diagramas de clases de las figuras 3.2 y 3.3 respectivamente.

3.4 Especificación formal de la interfaz del componente daIOCard

La descripción del componente, se realiza mediante el fichero que describe la interfaz del componente (*IoCard.ccd.xml*) y su contenido y formato son conformes a la especificación D&C de OMG, que en nuestro caso se define a través del W3C Schema ComponentDataModel.xsd. La interfaz del componente constituye la guia que utilizan las herramientas para acceder a toda la información que describe el componente.

La información del componente que contiene la descripción estándar de su interfaz es:

- Su identificador (*URI*) es *http://ctr.unican.es/hola/Iocard.*
- Ofrece el puerto (faceta) *digitalPort* que implementa la interfaz *iDigitalIO* y el puerto *analogPort* que implementa la interfaz *iAnalogIO* la cual se describe en el módulo *io* del fichero *interfaces/io/io.ice*
- Define las propiedades de configuración *confData* de la tarjeta y *pollingThreadPeriod* que es el periodo de ejecución con el que se realiza la tarea periódica.
- Asocia a muchos elementos los comentarios (*label)* que definen los elementos.
- Formula un conjunto de propiedades de información como el nombre del autor (*author*), etc.
- Facetas:
  - digitalIOPort: ofrece los servicios de negocio del componente que puede invocar un cliente para operaciones digitales, por ejemplo diread, diNumberOfLines. Implementa la interfaz que implementa iDigitalIO.
  - analogIOPort: ofrece los servicios de negocio del componente que puede invocar un cliente para operaciones digitales, por ejemplo aiChannelNumber, aiReadWithGain. Implementa la interfaz que implementa iAnalog.

- Puertos de activación: implementan una de las dos interfaces *PeriodicActivation* o *OneShotActivation* definidas en la tecnología C++-CCM. Cuando el componente es instanciado en una aplicación, el contenedor reconoce estos puertos, y por cada uno de ellos, crea un thread y haciendo uso de él ejecuta el procedimiento definido en la interfaz. Si el puerto implementa la interfaz PeriodicActivation, el thread invocará periódicamente el método *update()*, y si la interfaz implementada por el puerto es OneShotActivation, invocará una sola vez el procedimiento *run()*. Por cada puerto PeriodicActivation que implemente el componente, este debe definir una propiedad *<nombre del puerto>Period* con la que se define en configuración el periodo de invocación del método update().

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<DnCcdm:componentInterfaceDescription xmlns:xmi="http://www.omg.org/XMI"
        xmlns:DnCcdm="http://ctr.unican.es/cbsdnc/DnCComponentDataModel"
        xmlns:DnCct="http://ctr.unican.es/cbsdnc/DnCCommonTypes"
        xmlns:rtDnC="http://ctr.unican.es/cbsdnc/rtDnC"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:schemaLocation="http://ctr.unican.es/cbsdnc/DnCComponentDataModel
            ..\..\..\schemas\dnc\DnCComponentDataModel.xsd ">
    <description label="Componente que permite la gestión de entradas y salida analógicas y digitales a través de
                        una tarjeta de adquisición estándar"
            UUID="http://ctr.unican.es/hola/io/daIOCard"
            specificType="components/io/daIOCard.ccd.xml"
            supportedType="components/io/daIOCard.ccd.xml">
        <!-- Bussines port descrition -->
        <port name="digitalIOPort"
                    specificType ="interfaces/io/io.ice::io::iDigitalO"
                    supportedType ="interfaces/io/io.ice::io::iDigitalO"
                    provider ="true"
                    kind ="FACET"/>
        <port name="analogIOPort"
                    specificType ="interfaces/io/io.ice::io::iAnalogIO"
                    supportedType ="interfaces/io/io.ice::io::iAnalogIO"
                    provider ="true"
                    kind ="FACET"/>
        <port name="pollingThread"
                    specificType ="PERIODICACTIVATION"
                    supportedType ="PERIODICACTIVATION"
                    provider ="true"
                    kind ="PERIODICACTIVATION"/>
        <!-- Configuration properties -->
        <property name="confData"
                    type ="STRING"
                    label ="Identificador de la tarjeta dentro del procesador. Viene definido por la posición
                        de la tarjeta en el bus PCI"/>
        <property type="FLOAT" name="pollingThreadPeriod"/>
        <!-- General data of the component -->
        <infoProperty name="author">
            <value>Helder Oliveira y Laura Barros</value>
        </infoProperty>
        <infoProperty name="version">
            <value>02/09/08</value>
        </infoProperty>
        <infoProperty name="repository">
            <value>http://ctr.unican.es/cbsdnc</value>
        </infoProperty>
    </description>
</DnCcdm:componentInterfaceDescription>
```

Fichero: IoCard.ccd.xml

4. Código de negocio de la implementación PCI9111IOCard del componente daIOCard

Por código de negocio de un componente entendemos el código de una implementación del componente que proporciona una solución a la funcionalidad del componente, que ha sido formulada en su especificación. El código de negocio es una implementación desarrollada en un determinado lenguaje de programación, en un entorno monoprocesador. La estructura interna del código de negocio puede ser elegida de forma libre por el diseñador que lo desarrolla. Sin embargo, se le requiere que implemente el conjunto de interfaces que requiere la tecnología a fin de poder ser gestionado y adaptado a la plataforma de ejecución mediante herramientas automáticas, y en el contexto del plan de despliegue de una aplicación.

El código de negocio es diseñado e implementado por el *Diseñador* que es un experto en del dominio al que pertenece el componente, y que tiene conocimiento para generar el código que implemente su funcionalidad. Así mismo, el diseñador que conoce los detalles del diseño, es también el responsable de formular los modelos no funcionales del componente. Sin embargo, al diseñador no se le requiere que conozca los detalles de la plataforma de ejecución, esto es, no necesita ser experto de la adaptación del sistema operativo del procesador en el que se ejecuta el componente, ni tampoco los detalles del middleware de comunicaciones, a través del que son invocados sus servicios, o del que invoca los servicios que necesita para implementar su funcionalidad.

En la siguiente figura se muestran los elementos que constituyen el código de negocio PCI9111IOCard que se ha implementado del componente daIOCard.

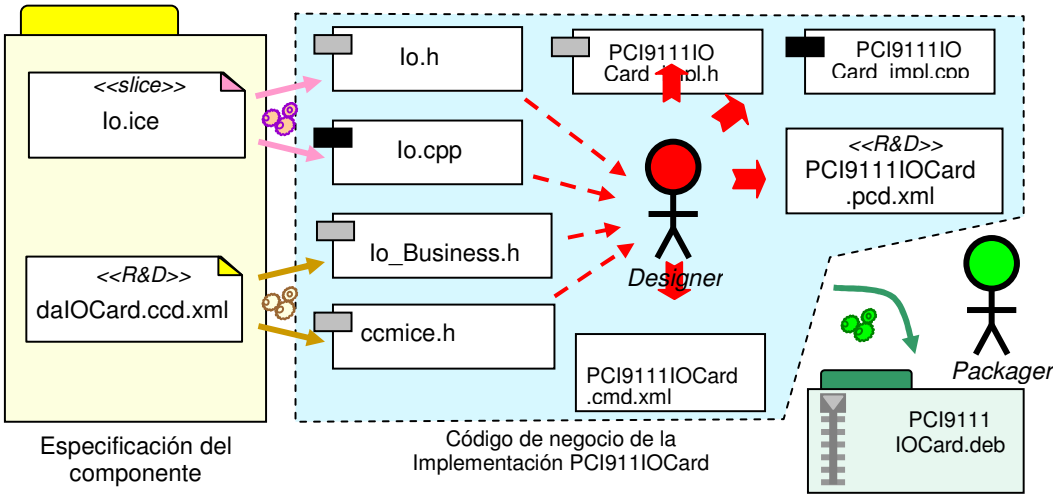

Figura 4-1: Elementos del código de negocio de un componente

El diseño del código de negocio parte de la especificación del componente daIOCard, esto es de la descripción completa de la interfaz del componente que describe los puertos, los parámetros de configuración, los requerimientos de threads, y de las descripciones de las interfaces que implementan los puertos y que describen la funcionalidad de negocio.

La especificación se procesa mediante herramientas automáticas para generar las interfaces que deben ser implementadas por el código de negocio:

- Utilizando la herramienta ***slice2cpp*** que proporciona ZeroC junto con el middleware ICE, se compilan las especificaciones de las interfaces descritas en lenguaje Slice, y se obtiene los ficheros io.h e io.cpp. que contiene las interfaces en lenguaje C++, los tipos asociados a ellas, y el código para su gestión.
- Utilizando la herramienta ***CppBIMGenerator*** se genera el código C++ de la interfaz daIOCard_mng compuesta por los ficheros io_Business.h y ccmice.cpp. La entrada a la

herramienta es el fichero que describe la interfaz del componente, y en el que se encuentra la información sobre el los puertos del componente y sobre sus parámetros de configuración.

El diseñador debe elaborar el código de negocio:

- Debe escribir el código de los ficheros PCI9111IOCard_impl.h y PCI9111IOCard_impl.cpp que implementa la funcionalidad de negocio del componente. La única restricción que le impone la tecnología es que debe implementar las interfaces de negocio iDigitalIO e iAnalogIO cuyo código de describe en el fichero io.h, implementar la interfaz daIOCard_mng cuyo código se formula en el fichero daIOCard_mng.h previamente generados, e implementar la interfaz PeriodicActivation definida en la tecnología.
- Debe formular el modelo de comportamiento no funcional que corresponde a la implementación que diseña. Esta deberá formularse en el lenguaje de modelado que se utilice en el entorno de desarrollo para cada aspecto de comportamiento no funcional. Por ejemplo, si el aspecto no funcional es el comportamiento de tiempo real, se formularía utilizando la metodología de modelado CBS-Mast.
- Debe escribir todos los elementos que constituyen la implementación utilizando el estándar D&C de OMG. En este caso debe escribir el fichero XML PCI9111IOCard.pcd .xml con una formación que corresponda a lo definido en el W3C Schema ComponentDataModel.xsd. En el se describe todos los elementos que constituyen el código de negocio, se referencia la descripción de la interfaz que se implementa, y se describe como debe ser utilizado el código de negocio en el contexto de una aplicación.
- A fin de su distribución como un elemento software independiente, el Empaquetador puede generar un fichero tipo deb que incluya todos los elementos, y permita su futura instalación en un entorno de desarrollo de aplicaciones.

4.1 Especificación D&C de la implementación

En esta descripción se describe la implementación del componente *daIOCard* dentro del modelo ejecutado por el desarrollador, que se realiza mediante un fichero (*daIoCard.pcd.xml*) cuyo contenido y formato son conformes a la especificación D&C de OMG (siguiendo la plantilla *DnCComponentDataModel.xsd)*.

Algunas de las características de este componente son:

- Su identificador (*URI*) es *http://ctr.unican.es/io/daIoCard*
- Su interfaz se describe en el fichero externo*components/io/daIOCard.ccd.xml*
- La implementación se denomina: *PCI9111IOCard.* Una implementación es una versión del componente que ha sido desarrollada utilizando una estrategia de diseño, un lenguaje y un entorno de programación específicos. Incluye todas aquellas características de comportamiento o de instalación que no quedan incluidas en el código de las implementaciones y que deben ser introducidas por el implementador directamente:
    1. Informa que es un componente de negocio reutilizable con diferentes *middlewares*.
    2. Es una implementación monolítica cuya especificación no depende de otras implementaciones.
    3. Contiene múltiples ficheros con códigos y datos (recursos y capacidades que se requieren del entorno para que pueda ser instanciado el componente):
- El elemento *PCI9111IOCard_impl* contiene el programa principal. Es un fichero, y su dirección una vez instalado será *components/io/.IoCard/PCI9111IOCard_impl.cpp*. Para ser ejecutado requiere que el recurso "IO_Card" sea del tipo PCI9111, del proveedor ADLINK.

- El elemento *io_business.h, PCI9111IOCard_impl.h, y otros más* son ficheros de datos y su dirección una vez instalado en el repositorio será components/io/.IoCard/Io_Bussiness.h y components/io/.IoCard/PCI9111IOCard_impl.h.

En la siguiente tabla, se muestra la descripción de la implementación *PCI9111IOCard* que se describe de forma completa en el Anexo C.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<DnCcdm:packageConfiguration xmlns:DnCcdm="http://ctr.unican.es/cbsdnc/DnCComponentDataModel"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ctr.unican.es/cbsdnc/DnCComponentDataModel
..\..\..\schemas\dnc\DnCComponentDataModel.xsd" label="Implementaciones disponibles en el proyecto ICE-CCM del
componente ioCard" UUID="http://ctr.unican.es/io/daIoCard">
<!-- Component Package Description -->
<basePackage>
 <realizes>
    <!-- ComponentInterfaceDescription-->
    <ref>components/io/daIOCard.ccd.xml</ref>
 </realizes>
 <!-- ComponentImplementationDescription-->
 <!-- Implementation -->
 <implementation name="PCI9111IOCard">
   <referencedImplementation>
      <description label="Implementación que permite la gestion de la tarjetaPCI9111-DG de la  empresa ADLink Company
Inc. con driver Comedi abierto para Linux.Permite el control de las 16 primeras lineas digitales de entrada y de las 16 primeras
de salida">
          <implements>
              <ref>components/io/daIoCard.pcd.xml</ref>
          </implements>
          <!--MonolithicImplementationDescription-->
          <monolithicImpl>
            <primaryArtifact name=" PCI9111IOCard">
              <referencedArtifact>
                <description location="components/io/.IoCard/PCI9111IOCard_impl.cpp">
                    <infoProperty name="ArtifactType">
                        <value>MAINCplusplusCODE</value>
                    </infoProperty>
                    <deployRequirement resourceType="IO_Card" name="IO_Card_Requirement">
                        <property name="type">
                            <value>PCI9111</value>
                        </property>
                        <property name="provider">
                            <value>ADLINK</value>
                        </property>
                    </deployRequirement>
                    <deployRequirement resourceType="os" name="os_Requirement">
                        <property name="type">
                            <value>Ubuntu</value>
                        </property>
                        <property name="version">
                            <value>6.06</value>
                        </property>
                        <property name="libs">
                            <value>comedilib-0.7.22 comedi-0.7.73</value>
                        </property>
                    </deployRequirement>
                </description>
              </referencedArtifact>
            </primaryArtifact>
            <primaryArtifact name="PCI9111IOCard_impl.h">
              <referencedArtifact>
                <description location="components/io/.IoCard/PCI9111IOCard_impl.h">
                  <infoProperty name="ArtifactType">
                      <value>HEAD</value>
                  </infoProperty>
                </description>
              </referencedArtifact>
            </primaryArtifact>
            <primaryArtifact name="Io_Business.h">
              <referencedArtifact>
                <description location="components/io/.IoCard/Io_Bussiness.h">
```

```
                    <infoProperty name="ArtifactType">
                        <value>HEAD</value>
                    </infoProperty>
                </description>
            </referencedArtifact>
…
</description>
</referencedImplementation>
</implementation>
</basePackage>
</DnCcdm:packageConfiguration>
```

Fichero: daIoCard.pcd.xml

4.2 Implementación PCI9111IOCard en lenguaje C/C++ del componente daIOCard.

Para cada una de las implementaciones del componente que se realicen, el diseñador debe formular el código de negocio que implementa la funcionalidad que tiene definida el componente en su especificación. Éste puede ser legado o desarrollado de forma específica para él.

A fin de que pueda ser procesado por la herramienta de integración, se requiere que el código de negocio satisfaga el siguiente conjunto de características:

1. Debe ofrecer un constructor que al ser invocado genere una instancia local de todos los objetos que constituyen el código de negocio. El modo de invocación del constructor, así como los parámetros que le deben ser pasados, se detallan en la descripción de la implementación D&C (en la descripción *IoCard.pcd.xml*).
2. Debe disponer de una clase (*pci9111ioCard_impl*) que implemente la interfaz de gestión del componente (*daIOCard_mng*). El constructor retorna el acceso a la clase principal referenciada como un objeto que implementa esta interfaz.
3. La interfaz de gestión del componente se genera automáticamente a partir de su especificación D&C (*IoCard.ccd.xml*).
4. La implementación de negocio debe ofrecer las facetas que están descritas en su especificación. Para acceder a cada faceta que pose el componente se accede por las funciones (*get_analogPort()*) y (*get_digitalPort())* que a tal fin tienen la interfaz de gestión. Las facetas deben implementar las interfaces que se especifican en el modelo, cuya descripción exacta (*io.h*) y (*io.cpp*) se genera de forma automática a través de la descripción Slice de la interfaz (*io.ice*). Además, como tiene varias facetas implementadas, la interfaz deberá crear un objeto por cada una de ellas.
5. Lo mismo ocurre con los puertos de activación del componente, por cada uno de ellos, el código de negocio deberá instanciar un objeto que implemente la interfaz correspondiente, y en consecuencia, la funcionalidad asociada a dicho puerto.

La generación de una implementación de negocio de un componente se efectúa siguiendo los siguientes pasos:

Utilizando la herramienta CppBIMGenerator se genera la interfaz de gestión del componente:

```
class da_ioCard_mng : ccmice::BusinessLifeControl {
        public:

        // Method for obtaining the analogPort facet
        virtual iAnalogIO *get_analogPort () = 0;

        // Method for obtaining the digitalPort facet
```

```
            virtual iDigitalIO *get_digitalPort () = 0;

            // Method for obtaining the pollingThread periodicActivation port
             virtual ccmice::PeriodicActivation *getpollingThread()=0;

            // Mutattor and accesor methods for confData
            virtual void set_confData (::std::string attr) = 0;

            virtual ::std::string get_confData () = 0;
            ~da_ioCard_mng(){};

};
```

Fichero: Interfaz daIOCard incluida en el fichero io_business.h

1. Se procesan las interfaces que se encuentran descritas en lenguaje Slice (*io.ice*) utilizando el precompilador (*Slice2cpp*) proporcionado por ZeroC. El resultado es la interfaz funcional que debe implementar el código de negocio (*io.h) y (io.cpp)*. También se generan otros ficheros auxiliares relativos a los tipos referenciados en la interfaz.
2. El diseñador debe formular el código de negocio del componente. La única limitación es que ofrezca el constructor, la interfaz de gestión y las interfaces de negocio previamente generadas. Éstas pueden ser generadas en una o en varias clases (es necesario cuando, por ejemplo, un componente tiene varias facetas que implementan la misma interfaz).

```
/*********************************************************************
 *                  PROYECTO HESPERIA
 *            Grupo Computadores y Tiempo Real (CTR)
 *               UNIVERSIDAD DE CANTABRIA
 *
 * Description: Equivalent interface of the home.
 *
 * @Autor  Jose M Drake
 * @Version 12/03/08
 *********************************************************************/
#include <string>
#include <comedilib.h>
#include <iostream>
#include <sstream>

#include <sys/types.h>
#include <sys/time.h>

#include <queue>
#include <vector>

using namespace std;

#include "io_business.h"

            class Action;
            class diAction;
            class doAction;
            class aiAction;

class pci9111ioCard_impl : bsn_io::da_ioCard_mng {
            public:
                        pci9111ioCard_impl();
                        ~pci9111ioCard_impl();

            class IDigitalIO_facet : public bsn_io::iDigitalIO  {...};
            class IAnalogIO_facet : public bsn_io::iAnalogIO {...};

     /*************************************************************************/
     /*          Attributes pci9111ioCard_impl                                */
     /*************************************************************************/
     private:
            IDigitalIO_facet* digitalIOPort;
            IAnalogIO_facet* analogIOPort;
            std::string configData;
```

```
    /*************************************************************************/
    /*      Methods pci9111ioCard_impl                                       */
    /*************************************************************************/
    private:
            inline double convertToDouble( const std::string& s );
            void pollingHandler();

    /*************************************************************************/
    /*      from BusinessLifeControl interface                               */
    /*************************************************************************/
    public:
            void activate();
            void passivate();
            void remove();
    /*************************************************************************/
    /*              from da_ioCard_mng interface                             */
    /*************************************************************************/

            // Method for obtaining the analogPort facet
            bsn_io::iAnalogIO* get_analogPort();
            // Method for obtaining the digitalPort facet
            bsn_io::iDigitalIO* get_digitalPort();
            // Method for obtaining the pollingThread periodicActivation port
             virtual ccmice::PeriodicActivation* getpollingThread()=0;

            // Mutattor and accesor methods for confData
            void set_confData (::std::string attr);
            ::std::string get_confData ();
};

class Action{…};

class aiAction : Action{…};
class doAction : Action{…};
class diAction : Action{…};
```

Fichero: PCI9111IOCard.h

4.3 Estructura de la implementación del componente

En este apartado se explica la estructura software de la implementación del componente.

En el diagrama de clases de la figura 4-2 y figura 4-3, se muestran las clases que constituyen el código de negocio de la implementación PCI9111IOCard_impl. El código de negocio debe implementar los métodos definidos en la interfaz de gestión. Estos métodos sirven para gestionar la vida del componente.

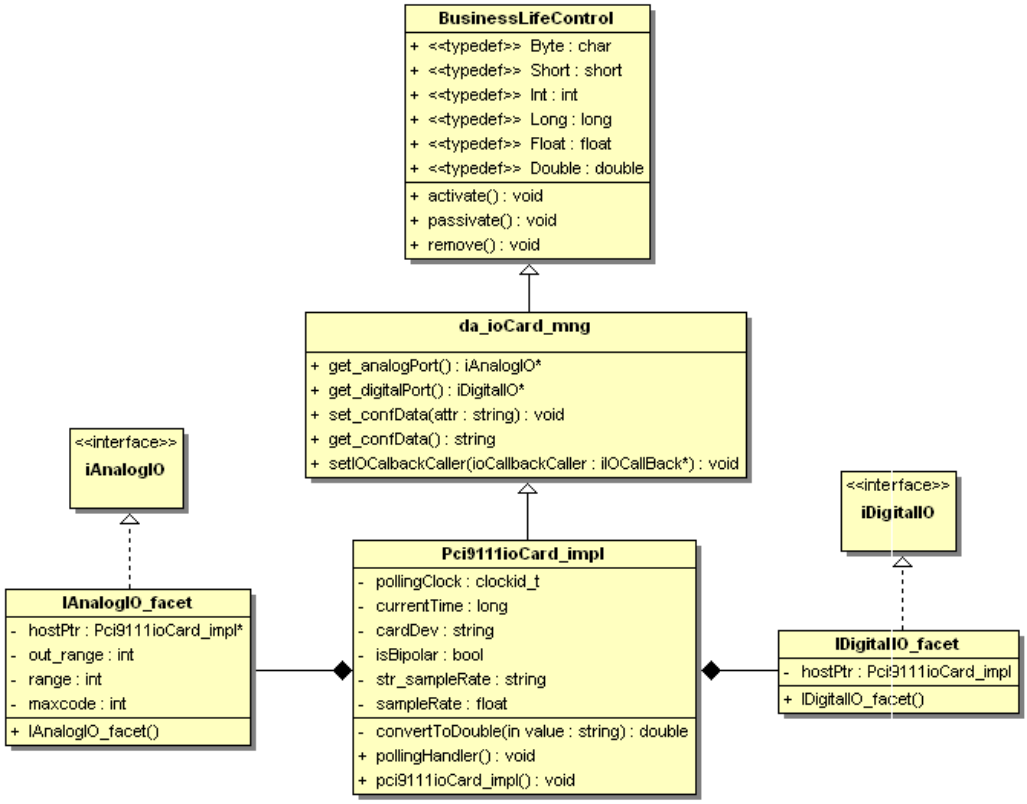

Figura 4-2: Estructura de la implementación PCI9111IOCard

Definir clases que implementen todas las interfaces definidas para las operaciones con la tarjeta. En nuestro componente disponemos de dos interfaces, una para operar el canal digital y otra para operar el canal analógico.

La implementación debe ofrecer las facetas por medio de sus puertos. El método get_analogPort() y get_digitalPort() ofrecen las interfaces (facetas) iDigitalIO y iAnalogIO respectivamente.

Para la gestión de los eventos se crea una cola de prioridades. El orden de la cola está basado en los plazos (deadLines) de cada evento:

```
priority_queue<Action*, vector<Action*>, greater<vector<Action*>::value_type> > awaitingAction;
```

Los deadLines para el canal digital pueden ser de dos tipos:

- period: periodo de blinking para un determinado evento. Este evento genera una onda cuadrada.
- with: tiempo en el cual una línea se queda con un estado, este tiempo sirve para generar un pulso.

Para los canales analógicos los deadLines para un nuevo evento se definen inicialmente igual al tiempo actual. La espera de que el evento se cumpla es activa, esto significa que el estado de las líneas tiene que ser verificado a cada instante de tiempo, y los siguientes deadLines se incrementan en una unidad de tiempo (tick).

Se crean tres arrays:

```
diAction *diActions[];
doAction *doActions[];
aiAction *aiActions[];
```

Estos arrays sirven para verificar el estado actual de las acciones y tienen como función principal alterar el tipo de la acción en la cola.

El array contiene las referencias de las acciones en la cola, que permiten operar sobre ella.

Son necesarios, porque una vez que las colas de prioridad de C++ no permiten recoger sus elementos insertados.

A continuación se visualiza un diagrama de clases del código de negocio con la las clases que implementan las distintas acciones:

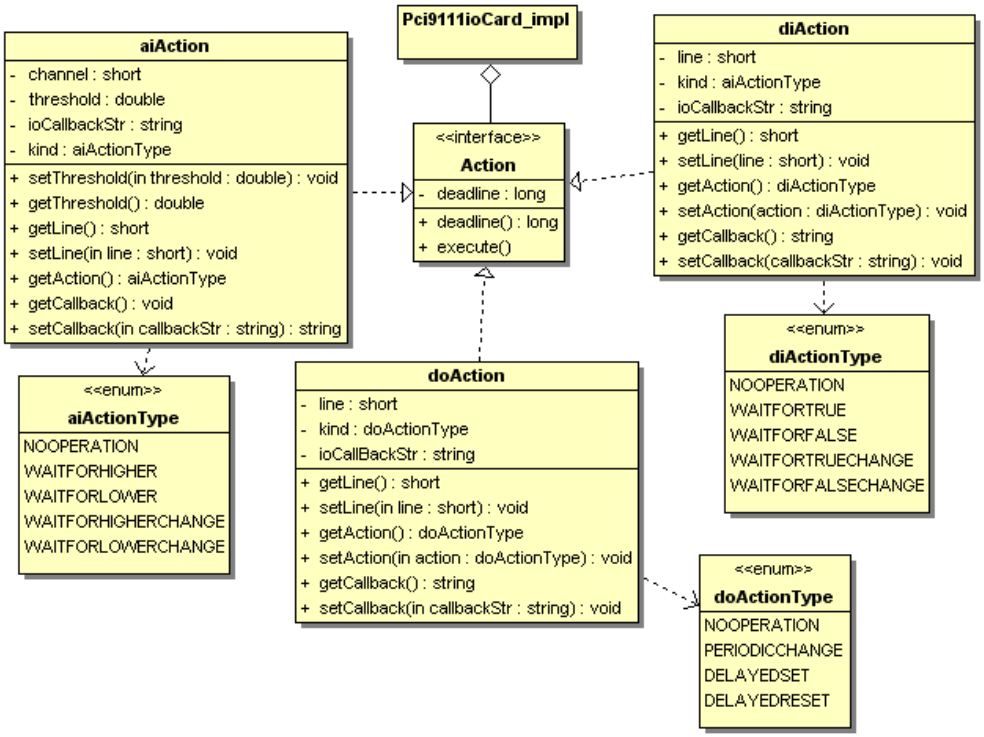

Figura 4-3: Diagrama de clases UML que implementan las distintas acciones

Las acciones necesarias para gestión de los eventos son:

- aiAction: necesaria cuando el componente recibe una petición para generar acciones en el canal analógico, como por ejemplo esperar que una línea tenga un valor superior a un umbral definido.
- doAction: para genera acciones en los canales digitales de salida, como por ejemplo generar ondas cuadradas.
- diAction: acciones creadas para hacer espera activa en una línea. Por ejemplo verificar si el estado de una línea es *TRUE*.

El método *execute()* es invocado siempre que el tiempo de la acción encolada, sea igual o inferior al tiempo actual.

La tarea de verificación de los deadLines en la cola, se resuelve con un thread periódico definido en la clase pollingThread.

El thread implementa su periodicidad con un reloj monótono de POSIX [6]:

- clock_gettime(CLOCK_MONOTONIC, &next_period);

Con la cola de prioridad y las acciones para operar en los distintos canales es posible crear y gestionar los distintos eventos que pueden ser solicitados por clientes.

Como ejemplo suponemos la llegada se un evento analógico.

Se recuerda que la interface iAnalogIO tiene un método para definir la ganancia. Este valor es muy útil para mejorar la resolución de los valores leídos de la tarjeta.

Evento del tipo AIAWAITHIGHER:
Verificar si en la cola existe alguna acción para la línea especificada.

1. Si existe una acción para la línea, colocar la variable *kind* como NOOPERATION en el array aiActions. Así que la acción que llegue al principio de la cola será descartada, este proceso tiene como función eliminar una acción que no sea la primera de la cola.
2. Se crea una nueva acción y con sus parámetros correctamente definidos, tales como, line, deadLine, ioCallbackStr. Este último será retornado al thread que espera por el evento. La espera de eventos basada en callback será explicada en el apartado 4-5.

Si no existe ninguna acción para la línea se crea una nueva acción y se coloca en la cola.

```
        void pci9111ioCard_impl::IAnalogIO_facet::aiAwaitEvent(::ccmice::Short _line, ::bsn_io::Event _eventype,
::ccmice::Double threshold, const ::std::string callbackProxyStr )
        {
                hostPtr -> aiActions[_line] -> setCallbackStr( callbackProxyStr );
                hostPtr -> aiActions[_line] -> setDeadline( hostPtr -> getCurrentTime() + 1 );
                switch( _eventype ){
                        case ::bsn_io::AIAWAITHIGHER:
                                hostPtr -> aiActions[_line] -> setKind( ::aiAction::WAITFORLOWER );
                        break;
                        case ::bsn_io::AIWAITCHANGETOLOWER:
                                hostPtr -> aiActions[_line] -> setKind( ::aiAction::WAITFORLOWERCHANGE );
                        break;
                        case ::bsn_io::AIWAITCHANGETOHIGHER:
                                hostPtr -> aiActions[_line] -> setKind( ::aiAction::WAITFORLOWERCHANGE );
                        break;
                        case ::bsn_io::AIAWAITLOWER:
                                hostPtr -> aiActions[_line] -> setKind( ::aiAction::WAITFORLOWER );
                        break;
                        default:
                        break;
                }
                 hostPtr -> awaitingAction.push(hostPtr -> aiActions[_line]);
        };
```

Con este procedimiento se pueden crear y gestionar las acciones para los eventos invocados.
A continuación se explica el funcionamiento de la tarea de polling.

4.4 Tarea de polling

En el código de implementación no esta permitido la creación de threads. Siempre que es requerido un thread ese es implementado en el contenedor.

El thread periódico en nuestro componente tiene la tarea de verificar si la cola tiene acciones pendientes. En caso de tener acciones en la cola verifica sus deadLine, si son menores que el tiempo actual, invoca al método *execute()*.

El thread es implementado en el contenedor y tiene un periodo definido pela variable period del fichero *confiValues,* este fichero se comenta el apartado 5-6. Se le pasa como parámetros una estructura con los parámetros de periodo y la referencia para el código de negocio. Esta referencia sirve para posibilitar invocar el método de *pollingHandler()* de la implementación.

El periodo es asegurado por un clock de alta resolución de POSIX, reloj monótono. Este reloj es lo adecuado para aplicaciones de tiempo real.
A cada periodo se invoca el método *pollingHandler()*. La función de este método es verificar si existen acciones pendientes en la cola.
Con este procedimiento conseguimos tener una invocación periódica del método por parte del contenedor y conseguimos verificar periódicamente nuestra lista de acciones pendientes en la cola.
El evento descrito en los dos apartados anteriores es basado en espera de eventos con callback.
El funcionamiento es explicado a continuación.

4.5 Espera de eventos basada en callback

El patrón de callback, aplicado a la tecnología cliente/servidor, se basa en permitir que el servidor se conecte con el cliente cuando finaliza su petición correspondiente, en vez de mantenerlos continuamente conectados:

- El cliente envía una petición para recuperar un proyecto del servidor, ofreciendo la información necesaria para la retrollamada junto con la petición.
- El cliente se desconecta del servidor y permite que este emplee el tiempo necesario para recuperar el proyecto. El cliente puede y debe continuar trabajando hasta que la información esté disponible. Esto puede lograrse simplemente haciendo que el cliente sea multithread.
- Cuando el servidor completa la tarea, conecta con el cliente y envía información del proyecto solicitado.

Dicho proceso, se observa en la figura 4-4.

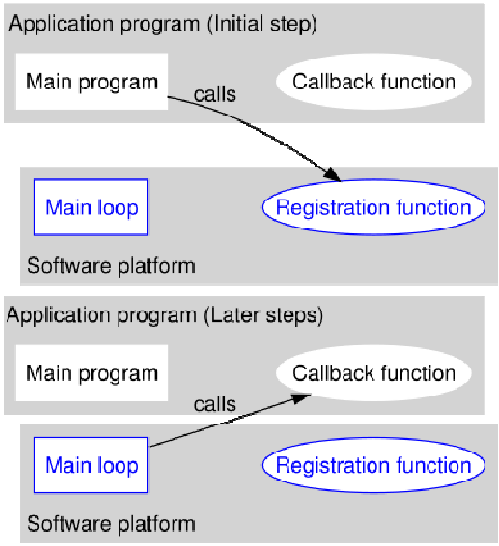

Figura 4-4: Registro del callback y posterior llamada en una aplicación

El propósito consiste en permitir que un cliente se registre en el componente PCI9111IOCard para ciertas operaciones. De esta forma, el componente puede notificar al cliente cuando la operación haya finalizado.
Las operaciones en las que el cliente se registra, mediante el paso del proxy de tipo *iIOCallback** que posteriormente el componente utilizará para llamarlo cuando se produzca el evento que el cliente espera que le notifiquen, son:

- diAwaitEvent(line: short, myEvent: Event, myCallback: iIOCallback*): void, espera que ocurra en la línea digital, el evento especificado en el parámetro de tipo Event.
- aiAwaitEvent(line: short, myEvent: Event, threshold: double, myCallback: iIOCallback*): void, espera que ocurra en la línea analógica, el evento especificado en el parámetro de tipo Event.

Al compilar los métodos en lenguaje *Slice* a C++, el proxy *iIOCallback** se traduce a un objeto de la clase *iIOCallbackPrx*. Como el código de negocio no debe conocer nada de la tecnología Ice, debemos resolverlo mediante el siguiente protocolo de actuaciones:

- En el código de negocio:
  1. Definimos los siguientes métodos dentro de la interfaz de gestión del componente y los implementamos en el código de negocio:

setIoCallbackCaller(ioCallbackCaller: iIOCallback*): void, establece el valor del proxy al que debemos notificar el evento, cuyo valor es pasado por el parámetro ioCallbackCaller.
geIoCallbackCaller():iIOCallback*, devuelve el valor del proxy al que debemos notificar el evento.

2. Los proxies (*string*) de los clientes que esperan cambios de eventos en cada una de las líneas son guardados en una variable de tipo string en cada una de la acciones correspondientes
3. Al definir los métodos de las interfaces, pasamos como objeto que espera el evento, un *string* que posteriormente será convertido a Proxy.
   diAwaitEvent(line: Short, myEvent: Event, callbackProxyStr: string):void
   aiAwaitEvent(line: Short, myEvent: Event, threshold: Float, callbackProxyStr: string) :void
4. En el método *execute()* de la clase *Action* que ha sido explicado en el apartado anterior y es utilizado para ejecutar la acción dependiendo del evento que se produzca, debemos ejecutar el método *callbackAction()* sobre el Proxy correspondiente (el que esté esperando que se produzca el evento en la línea).

- En el Wrapper:
  Creamos la clase *IoCallbackWrapper:*
  callbackAction(caller: string, myEvent: Event, line: short, proxyStr: string): void, convierte el parámetro proxyStr a proxie e invoca el método *callbackAction()* sobre el mismo. Este método sirve para informar el cliente de que se ha producido el evento esperado en la línea que es enviada como parámetro.
  En el método *configurationComplete()* del *Wrapper*, ejecutaremos el método *setIoCallbackCaller()* de la implementación.
  En los métodos *diAwaitEvent()* y *aiAwaitEvent()* de las facetas *iDigitalIOFacet* y *iAnalogIOFacet* respectivamente, se invocan los métodos del mismo nombre de la implementación, pero pasando como parámetro un string (convirtiendo el Proxy a string mediante el método proporcionado por Ice: ice_toString())

5. Estructura e implementación del contenedor

5.1 Estructura del contenedor

La arquitectura interna de la implementación de un componente se ha diseñado de modo que se incorpore el código de negocio sin necesidad de ser modificado, y sea más sencilla la generación automática del código de los elementos del contenedor, dejando al desarrollador del componente únicamente la responsabilidad del código de negocio.

Dicho código puede ser elaborado, además, como si el entorno de ejecución fuese siempre monolenguaje y monoprocesador.

En la figura 5-1 se muestran los elementos que forman parte de la estructura interna del componente PCI9111IOCard:

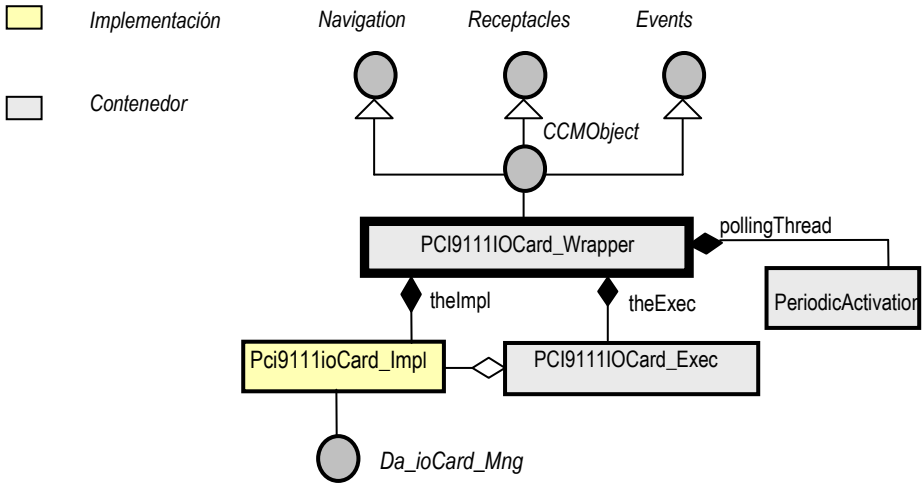

Figura 5-1: Estructura de la implementación del componente daIOCard

La tecnología requiere que el código de negocio del componente ofrezca un puerto para gestionar de forma opaca y uniforme su configuración, conectividad y ciclo de vida. Este puerto implementa la interfaz <nombre componente>Mng, que no tiene dependencias con la tecnología subyacente. Para la gestión del componente esta interfaz define los siguientes métodos:

- El contenedor en sí (*Wrapper*), que ofrece la interfaz equivalente del componente. Dicha interfaz es la que LwCCM establece como la única interfaz a través de la que los clientes o la herramienta de despliegue acceden al componente. Esta interfaz está estandarizada, y extiende a la interfaz CCMObject, que entre otras, ofrece métodos para acceder a las facetas del componente, conectar componentes a sus receptáculos, etc.
- El ejecutor (*executor*), que implementa los servants o los adaptadores, a través de los que se invocan las operaciones de las facetas del código de negocio. Es este elemento quien aporta todos los mecanismos ICE que se requieren para llevar a cabo la ejecución de las invocaciones.
- El gestor de threads incluye todos aquellos elementos necesarios para gestionar los puertos de activación que se hayan declarado en el componente. En la tecnología CCM, por cada puerto de activación especificado en la especificación D&C, se deberá crear un thread con sus correspondientes parámetros (el procedimiento a ejecutar, el parámetro de planificación correspondiente, el periodo en el caso de una activación periódica) y gestionarlo a lo largo de todo su ciclo de vida. En el caso C++, creamos un thread de POSIX y para suya ejecución periódica, utiliza un reloj monótono. Este reloj es el más indicado para usar aplicaciones de tiempo real, por ejemplo su hora no puede ser cambiada.

El código de negocio del componente sólo debe cumplir el requisito de implementar la interfaz de gestión (da_ioCard_mng), que es generada de forma automática y que no presenta

dependencias con la tecnología subyacente. Esta interfaz se ha definido como parte de la tecnología y engloba todos los métodos que permiten al contenedor manejar el código de negocio durante todo el ciclo de vida del componente. Para la gestión del componente esta interfaz define los siguientes métodos:

- Métodos de navegación, *get<FacetName>*(), que permiten obtener cada una de las implementaciones de las facetas que ofrece el componente.
- Métodos de asignación, *set<AttributeName>*(), de los valores de los atributos de configuración del componente.
- Métodos de navegación, *get<ActivationPortName>*(), que permiten obtener las implementaciones de los puertos de activación definidos para el componente.
- Métodos para la gestión del ciclo de vida del componente.

A fin de diseñar componentes cuyo código tenga un comportamiento temporal predecible, no se permite crear threads en su código de negocio. En su lugar, cuando el componente necesita un thread para implementar su funcionalidad, declara un tipo especial de puerto, denominado puerto de activación, que implementa una de las dos interfaces *PeriodicActivation* o *OneShotActivation*. Cuando el componente es instanciado en una aplicación, el contenedor reconoce estos puertos, y por cada uno de ellos, le otorga al componente un thread, a través de la ejecución del correspondiente procedimiento implementado por el puerto con un thread creado por él. En el caso de un puerto de tipo PeriodicActivation, el thread invocará periódicamente el método *update()*, mientras que en el caso de un OneShotActivation, invocará una sola vez el procedimiento *run()*.

En nuestro componente tenemos una interface *PeriodicActivation,* el thread invoca periódicamente el método threadPolling().

A continuación se explica la las implementaciones de cada uno de los elementos.

5.2 Implementación del Contenedor

Todos los componentes que siguen la especificación CCM deben ofrecer la interfaz CCMObject (que a su vez hereda de otras interfaces propias de la tecnología). Para tener acceso a una interfaz desde el middleware ICE se necesita disponer de su especificación en lenguaje SLICE. Para ello, compilamos a lenguaje C++ (mediante la herramienta *slice2cpp* de Ice) la traducción de las interfaces definidas por OMG en el modelo LwCCM (expresadas en lenguaje IDL) al lenguaje SLICE [7]. Dichas interfaces se muestran en seguida en el fichero *ccmcomponens.ice* cuyo código completo puede verse en el Anexo C.

```
/*****************************************************************
 *                PROYECTO C++-CCM
 *            Grupo Computadores y Tiempo Real (CTR)
 *               UNIVERSIDAD DE CANTABRIA
 *
 * Description: CCM model
 *
 * @Autor  J.M Drake, Patricia López,Laura Barros, Helder Castro
 * @Version 18/12/2007
 *****************************************************************/
module ccm{
/* Navigation Interface */
            …
          //CCM: interface Navigation {
           //                    Object provide_facet (in FeatureName name)raises (InvalidName);
           //                    FacetDescriptions get_all_facets();
           //                    FacetDescriptions get_named_facets (in NameList names)raises (InvalidName);
           //                    boolean same_component (in Object object_ref);
           //};
           interface Navigation {
                                Object* provideFacet (string name) throws InvalidName;
                                FacetDescriptions getAllFacets();
                                 FacetDescriptions getNamedFacets (NameList names) throws InvalidName;
                                 bool sameComponent (Object* objectRef);
           };
```

```
/* Events Interface  */
        …
/*Receptacles Interface */
        …
/*CCM object Interface */
          …
          //CCM: interface CCMObject : Navigation, Receptacles, Events {
          //                    CCMHome get_ccm_home( );
          //                    …
          //};
        interface CCMObject extends Navigation, Receptacles, Events {
                        Proxy getComponentDef();
                                                …
        };
};
```

Fichero: ccmcomponent.ice

La versión de negocio de la implementación del componente se transforma en una implementación instanciable e invocable en una plataforma con la tecnología C++-CCM a través de dos objetos de las clases ejecutor y contenedor (*PCI9111IOCardExec, PCI9111IOCardWrapper*). En la figura siguiente se muestra el diagrama de clases en el que se representan las relaciones entre estas clases:

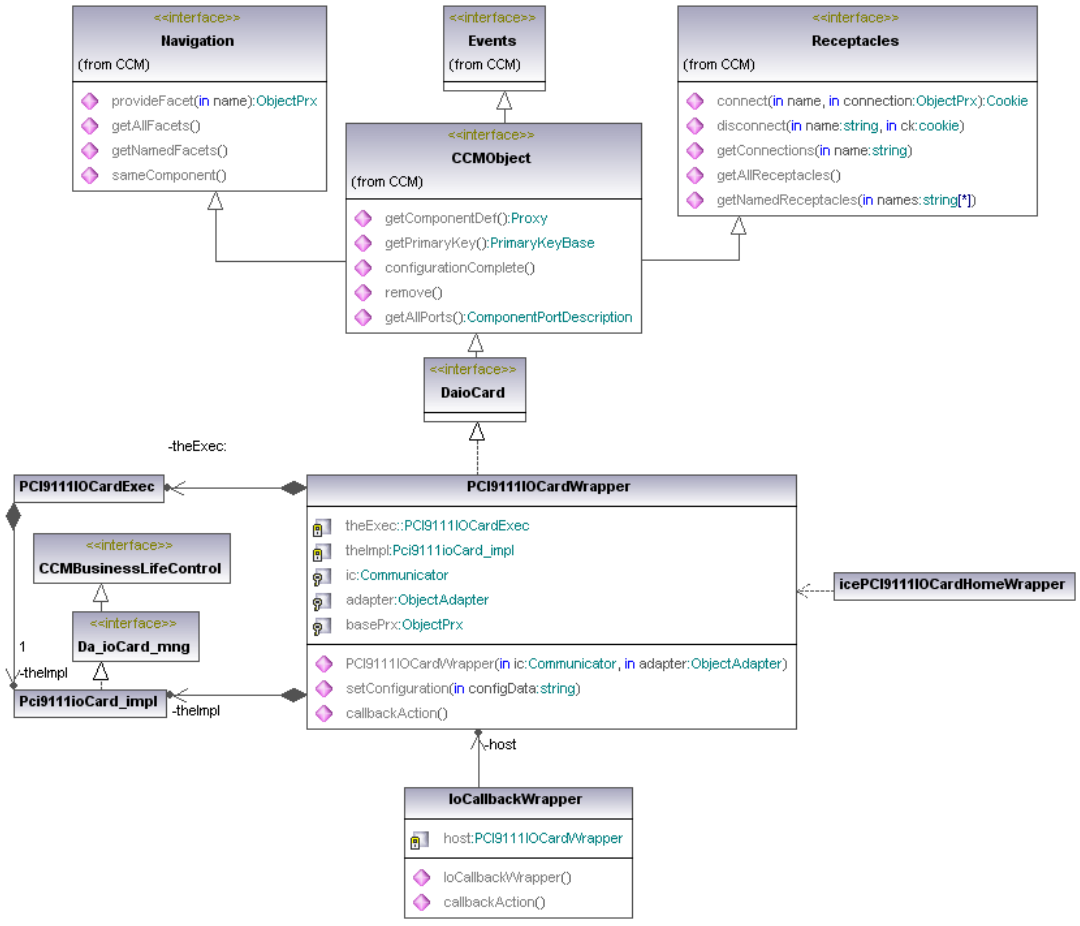

Figura 5-2: Diagrama de clases de la implementación C++-CCM de un componente

Código de negocio del componente (*PCI9111IoCard_impl*): Representa la clase principal del código de negocio, que ha sido desarrollado de acuerdo con las reglas definidas en el apartado previo. El código debe importar las interfaces que implementan sus facetas (el paquete *io*). Estos paquetes se obtienen del precompilado de las interfaces en lenguaje *Slice.*

Interfaz de referencia (*daioCardPkg*): interfaz especificada por CCM, a través de la que los clientes o la herramienta de despliegue acceden al componente ya que extiende a la interfaz *CCMObject*, que hereda de otras interfaces definidas en la tecnología: una que ofrece métodos para acceder a las facetas del componente (*Navigation)*, otra que ofrece métodos para la gestión de eventos (*Event*) y por último (*Receptacles*) que sirve para conectar componentes a sus receptáculos.

Las cabeceras e implementaciones de las clases Wrapper, Executor y Home, se encuentran en los ficheros *icePCI9111IOCardWrapper.h* y *icePCI9111IOCardWrapper.cpp* respectivamente que se pueden consultar en el Anexo B.

5.3 Implementación del Wrapper.

Contenedor (*icePCI9111IOCardWrapper*): Representa el contenedor que es creado sobre el nudo de ejecución, a fin de ejecutar el código de negocio y los mecanismos de comunicación proporcionados por el *middleware*. Ofrece también la interfaz (*daioCardPkg*) para gestionar el ciclo de vida del componente, su configuración y conexión/desconexión con otros componentes. Esta interfaz está estandarizada y corresponde a una extensión directa de la interfaz *CCMObject* definida en el paquete de la tecnología *ccm*, y por tanto es independiente de la funcionalidad del componente.
El diagrama de clases de la figura 5-2, muestra los elementos que componen la clase Wrapper:

- El Wrapper contiene la referencia al Executor y a la referencia a la Implementación (*theExec,theImpl*).
    - Delega los métodos de navegación de facetas (*provideFacet(),getAllFacets(),*etc) en los del mismo nombre, ofrecidos por la clase Ejecutor.
    - Los métodos de conexión y navegación de receptáculos (*connect(), getAllReceptacles(),*etc) no serán implementados porque el componente DaioCard no requiere servicios de otros componentes, por lo que no presenta receptáculos.
    - Ofrece un método para activar la implementación y establecer la conexión del componente a través de sus receptáculos: *configurationComplete()*.
    - Ofrece un método para obtener todos los puertos funcionales del componente: *getAllPorts()*.
    - Dispone de un método para borrar el acceso a la instancia del componente: *remove()*.
    - Dispone de un método para establecer los valores de los atributos de configuración de la instancia del componente: *setConfiguration()*.

Las cabeceras e implementaciones de la clase Wrapper se encuentra en los ficheros *icePCI9111IOCardWrapper.h* y *icePCI9111IOCardWrapper.cpp* respectivamente que se pueden consultar en el Anexo B.

5.4 Implementación del Executor

Ejecutor (*PCI9111IoCardExec*): Representa los mecanismos por los que otros componentes invocan las operaciones ofrecidas por las facetas. Básicamente se utiliza para ejecutar localmente las operaciones invocadas.

El diagrama de clases de la figura 5-3, muestra los elementos que componen la clase Ejecutor:

- El ejecutor contiene una referencia a la implementación (*theImpl)* para invocar las operaciones ofrecidas por el código de negocio a través de sus facetas.

- Contiene un array (*theFacets*) formado por elementos *FacetDescription* que es una estructura de dos campos, el nombre de cada una de las facetas y el proxie a las mismas.
- Ofrece métodos para:
  - Creación de las facetas*: createFacetExecutors().*
  - Obtener una faceta determinada por su nombre: *providefacet().*
  - Obtener todas las facetas del componente: *getAllFacets().*
  - Obtener un conjunto de facetas especificadas por sus nombres: *getNamedFacets().*

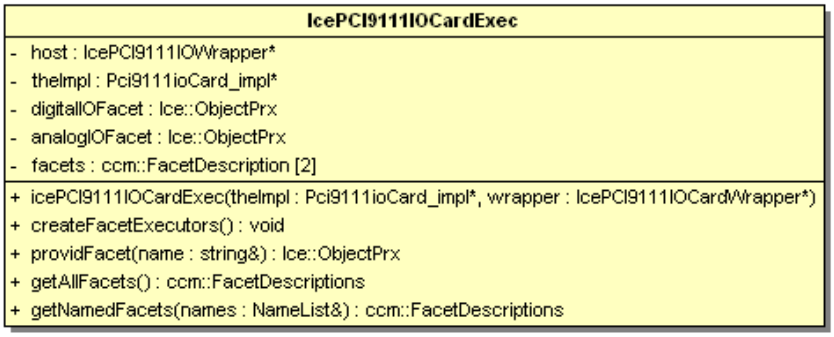

Figura 5-3: Diagrama de clases de la clase Executor

Las cabeceras e implementaciones de la clase Executor se encuentra en los ficheros *icePCI9111IOCardWrapper.h* y *icePCI9111IOCardWrapper.cpp* respectivamente que se pueden consultar en el Anexo B.

5.5 Implementación del Home

Gestor del componente (*icePCI9111IoCardHomeWraper)*: Representa el mecanismo a través del que se pueden instanciar y configurar las implementaciones de los componentes en un determinado procesador de la plataforma. Presenta un método estático, a través del cual se puede crear el *Home* en un nudo. A partir de él se pueden establecer los parámetros de configuración, invocando la operación *setConfigurationValues()* y luego invocando la función *create()* se pueden crear un número arbitrario de instancias del componente con unos ciertos valores de configuración de los que hablaremos posteriormente.

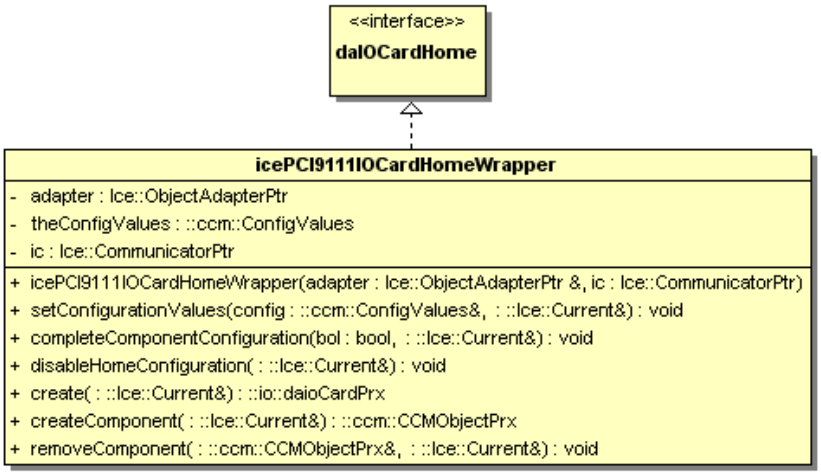

Figura 5-4: Diagrama de clases UML de la clase Home

La clase Home deberá implementar la interfaz *daioCardHome* que a su vez extiende de las interfaces definidas en la tecnología que definen los métodos de gestión del *home* del componente.

Los elementos que componen la clase Home:

- Mecanismos para almacenar los valores de configuración del componente(*theConfigValues*).
- Ofrece métodos para:
  - La creación de una o varias instancias del componente con los valores de configuración establecidos en el plan de despliegue: *create() y createComponent().*
  - Completar la configuración del componente y desactivar el componente: *completeComponentConfiguration() y disableHomeConfiguration().*
  - Borrar un componente y evitar futuros accesos al mismo: *removeComponent().*
- Un método *main()* que crea el *servant* del *home* a través del cual, la instancia del componente, espera peticiones del entorno.

Las cabeceras e implementaciones de la clase Home se encuentra en los ficheros *icePCI9111IOCardWrapper.h* y *icePCI9111IOCardWrapper.cpp* respectivamente que se pueden consultar en el Anexo B. El método *main()*, referenciado anteriormente, se encuentra en el fichero *cardHomeWrapper.cpp.*

5.6 Configuración del componente basada en propiedades ICE

En la figura 5-5 se muestra el conjunto de los ficheros de código que constituyen el componente daiOCard de la tecnología C++-CCM. Las descripciones Slice de las interfaces de los puertos de negocio del componente y de sus las interfaces de gestión así como de su constructor (*Home*) se procesan utilizando el pre-compilador Slice2cpp de la empresa ZeroC. Como resultado se genera un conjunto de ficheros .h y .cpp que son requeridos por las clases principales para implementar sus capacidades de comunicación con otros elementos:

- ccm: contiene los ficheros resultantes de la compilación del fichero *ccmcomponentes.ice* que contiene las interfaces SLICE que define la tecnología CCM.
- io: contiene los ficheros resultantes de la compilación del fichero *io.ice.*

Todos los ficheros presentes en la siguiente figura, pueden consultarse en el Anexo B.

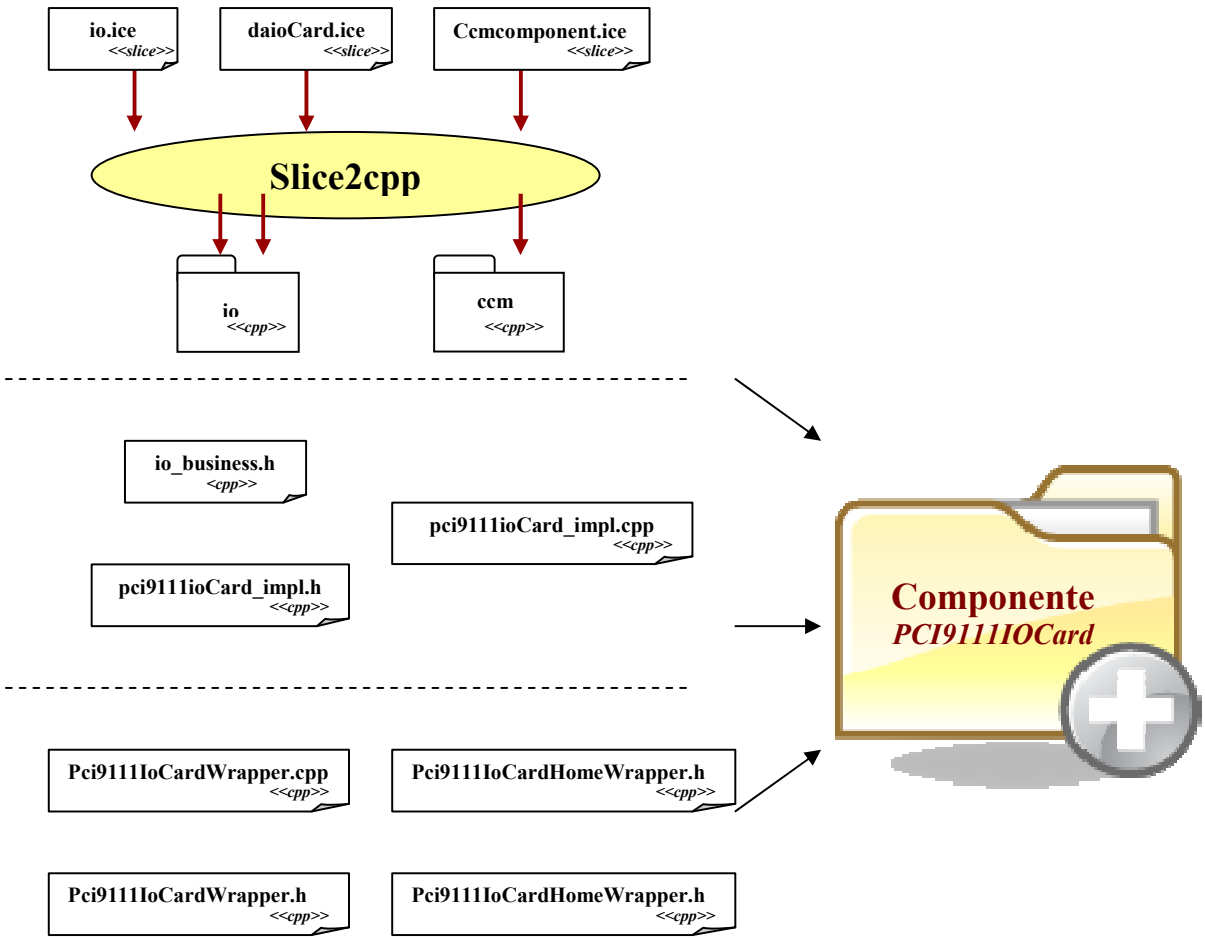

Figura 5-5: Ficheros del componente C++-CCM

En un fichero de texto, el usuario puede definir cualquier propiedad que requiera su aplicación, simplemente estableciendo un par de elementos constituido por un nombre y un valor:

Este fichero (configValues), constituye un parámetro de ejecución para la aplicación que efectúa el despliegue cuyo código se encuentra en el Anexo B. Por ejemplo, para ejecutarlo se ejecutaría la siguiente instrucción:

$./deploymentTool --Ice.Config=home/.../config.Components

```
adioCard.objectName = ioCard
adioCard.protocolData = default -p 8000
adioCard.confData = /dev/comedi0 TRUE 10*10^-3
adioCard.period = 10000000
```

Fichero: configValues

6. Empaquetamiento de componentes

6.1 Empaquetamiento de módulos en Linux

El propósito de este punto es describir la construcción general de un paquete Debian [8]

Los programas que pueden ser necesarios para el desarrollo:

- **dpkg-dev:** Este paquete contiene herramientas necesarias para desempaquetar, construir y cargar los archivos fuente de Debian.
- dh_make: herramienta que convierte los archivos fuente en paquetes de Debian
- **file:** Este práctico programa puede determinar el tipo de un archivo.
- **gcc:** El compilador de C de GNU, necesario si el programa, al igual que muchos otros, ha sido escrito en C. Este paquete también se instalará varios otros paquetes, como *binutils* que incluye el software utilizados para elaborar y construir los archivos objeto y CPP, el preprocesador de C.
- **g++:** El GNU C++, necesario si el programa ha sido escrito en C++.
- **libc6-dev:** las bibliotecas para C y archivos de cabecera que gcc necesita para crear archivos objeto
- **make:** generalmente la creación de un programa se realiza en varias etapas, y por lo tanto, es mejor usar este programa para automatizar los pasos para la creación de un *Makefile* en lugar de correr siempre los comandos

Como nota hay que distinguir entre dos tipos de paquetes que se pueden crear: fuente y binarios. Un paquete fuente contiene un código fuente que puede compilar. Un paquete binario contiene sólo el programa listo. No se debe confundir los términos, como el código fuente del programa y la fuente del paquete de programas.

A continuación se describen los pasos para compilar.

Para crear un paquete deb es necesario crear un directorio llamado debían en la raíz del código fuente donde se situaran una serie de archivos que serán usados para la construcción de los paquetes y para su posterior clasificación en los repositorios

La forma más sencilla es copiar nuestro código fuente a un directorio nuevo con el seguirte formato:

testpack-0.0.1

Los dígitos añadidos representan la versión de nuestro paquete.

Muy importante:

- No usar mayúsculas
- No usar caracteres especiales
- Usar una nomenclatura de versiones lo más estándar posible (con 3 números)
- Separar el nombre del proyecto de la versión con un guión medio.

Entrar en el directorio de las fuentes y generamos el directorio debían con la ayuda de:

dh_make --creatorig

dpkg-buildpackage –rfakeroot

La compilación debe ser hecha en modo usuario.
Algunos de los archivos creados en la compilación son explicados a continuación.

Los más importantes de ellos son *control*, *copyright, changelog* y *rules*, que son necesarios para todos los paquetes:

- control: Este fichero contiene varios valores que *dpkg*, *dselect* y otras herramientas de control de paquetes se utilizan para administrar el paquete.

Como ejemplo se presenta un fichero de control que dh_make crea:

```
1 Source: package
2 Section: unknown
3 Priority: optional
4 Maintainer: Helder Castro <oliveirahr@unican.es>
5 Build-Depends: debhelper (>= 4.0.0), libc6-dev
6 Standards-Version: 3.6.2
7 Package: package
8 Architecture: i386 source
9 Depends:  ${shlibs:Depends}, ${misc:Depends}
10 Description: <insert up to 60 chars description>
11 <insert long description, indented with spaces>
```

Las líneas de 1-6 son para información para el control del paquete.
La línea 1 es el nombre del paquete
La línea 2 es la sección de distribución en la que el paquete se incluirá.
La línea 3 describe cómo es de importante este paquete.
La línea 4 es el nombre y la dirección de correo electrónico del mantenedor.
La línea 5 contiene una lista de paquetes necesarios para construir el paquete. Algunos paquetes como gcc y make están implícitos. Si cualquier compilador o herramienta no estándar es necesario para construir el paquete, deberá añadirse a la línea «Build-Depends." Las entradas múltiples se separan por comas.
La línea 6 es la versión de este paquete.
La línea 7 es el nombre del paquete binario. Este suele ser el mismo que el nombre del paquete fuente, pero no necesita ser de esta manera.
La línea 8 describe la arquitectura del CPU del paquete a fin de que el paquete se pueda compilar. Dejamos esto como «any» porque dpkg-gencontrol rellenar este campo con el valor apropiado para cualquier máquina en la que este paquete se ha compilado.
La línea 9 muestra una de las más poderosas características del sistema de empaquetamiento de Debian. Los paquetes se pueden relacionar unos con otros de diversas maneras. Además de la *Depends*, los demás campos relacionales son: *Recommends*, *Suggests*, *Pre-Depends*, *Conflicts*, *Provides*, y *Replaces*.

Lo que significan las dependencias:

- Depends: El paquete no se instalará a menos que los paquetes que dependen estén instalados.
- Recommends: El Interfaces como dselect o aptitude pregunta si desea instalar los paquetes recomendados junto con el paquete, dselect vai incluso insistir.
- Suggests: Cuando un usuario instala el programa, todas las interfaces le preguntaran si quiere instalar los paquetes sugeridos. El dpkg y apt-get no.
- Pre-Depends Esto es más importante que *Depends*. El paquete no se instalará a menos que los paquetes que "pre-dependen" se han instalado y configurado correctamente.
- Conflicts: El paquete no será instalado hasta que todos los paquetes con los que tiene conflictos se eliminen.
- Provides: para algunos tipos de paquetes hay muchos nombres virtuales, que pueden definirse. Se pueden obtener la lista completa en el archivo /usr, /share, /doc, /debian-policy, /virtual-package-name-list.txt.gz. Utilizar si el programa ofrece una función existente en un paquete virtual.
- Replaces: Utilizar si el programa reemplaza ficheros de otro paquete, o sustituir completamente otro paquete (generalmente se usa junto con *Conflicts*). Archivos de otros paquetes listados serán sobreescritos con los archivos del paquete.

Todos estos campos tienen una sintaxis uniforme. Se trata de una lista de nombres de paquetes separados por comas. Estos nombres de paquetes también pueden ser listas de nombres de paquetes alternativos, separados por *pipes*.

Los campos pueden limitar la aplicación a versiones determinadas de cada paquete de la lista. Estas versiones se enumeran entre paréntesis después de cada nombre de paquete individual, y deberán llevar una relación de la siguiente lista seguido por el número de versión. Las relaciones se permite: <<, <=, =,> = y >> anteriormente, antes o igual, exactamente igual, igual o más jóvenes, respectivamente.

Por último, la última característica es $ (shlibs: Depends). Una vez que el paquete se ha compilado e instalado en el directorio temporario, dh_shlibdeps buscará los binarios y bibliotecas de ese directorio, determina las dependencias de bibliotecas compartidas y detecta paquetes en los que están, como libc6 o xlib6g. Él pasará la lista a dh_gencontrol que va a llenar el campo correctamente.

La línea 10 es una breve descripción.

La línea 11 es donde la descripción detallada entra. Este debe ser un apartado que proporciona más información sobre el paquete. No puede haber líneas en blanco. Por otra parte, no puede haber más de una línea en blanco después de la descripción detallada.

- Copyright: Este archivo contiene información sobre los recursos superiores del paquete, los detalles del derecho de autor y licencia.

Lo importante que se añade a este archivo es el lugar de donde se obtuve el paquete, los detalles del derecho de autor y de licencias del paquete. Se debe incluir la licencia completa, a menos que sea una licencia de software libre común, como GNU GPL.

- Changelog: Este archivo tiene un formato que usan dpkg y otros programas para obtener el número de versión, revisión, distribución y urgencia de su paquete.

El dh_make crea el archivo siguiente:

```
1 package (0.1-1) unstable; urgency=low
2 * Initial release Closes: #nnnn  (nnnn is the bug number of your ITP)
3 -- Helder Castro <oliveirahr@unican.es> Sun, 10 August 2008 10:37:42 +0200
```

La línea 1 es el nombre del paquete, versión, distribución y urgencia. El nombre debe ser el nombre del paquete fuente, la distribución puede ser o inestable (o incluso «experimental»), y la urgencia no debería ser cambiada para nada más que «bajo».

La línea 3 es la entrada de registro, donde debe documentar las modificaciones introducidas en esta revisión del paquete.

- Rules: Tenemos que ver las reglas exactas que dpkg-buildpackage tendrá que seguir para crear el paquete. Esto es, en realidad, otro Makefile, pero que no sea proporcionado por el autor del código fuente. A diferencia de los otros ficheros en debian, el archivo debe ser ejecutable.

Cada fichero «rules», al igual que muchos otros Makefiles, se compone de varias reglas que especifican cómo tratar con el código fuente. Cada regla son los objetivos, nombres de archivos o nombres de acciones que deben seguirse (por ejemplo, «build» o «install»). Las reglas que quieren ser invocados como argumentos de las líneas de comando (por ejemplo, ./debian/rules/build o make-f rules install). Después se pueden enumerar las dependencias, programas o archivos que depende la regla.

Esto es un ejemplo de un archivo debian/rules generado por dh_make:

```
1  #!/usr/bin/make -f
2  # -*- makefile -*-
```

```
 3  # Sample debian/rules that uses debhelper.
 4  # This file was originally written by Joey Hess and Craig Small.
 5  # As a special exception, when this file is copied by dh-make into a
 6  # dh-make output file, you may use that output file without restriction.
 7  # This special exception was added by Craig Small in version 0.37 of dh-make.
 8  # Uncomment this to turn on verbose mode.
 9  #export DH_VERBOSE=1
10  configure: configure-stamp
11  configure-stamp:
12          dh_testdir
13          # Add here commands to configure the package.
14          touch configure-stamp
15  build: build-stamp
16  build-stamp: configure-stamp
17          dh_testdir
18          # Add here commands to compile the package.
19          $(MAKE)
20          #docbook-to-man debian/testpack.sgml > testpack.1
21          touch $@
22  clean:
23          dh_testdir
24          dh_testroot
25          rm -f build-stamp configure-stamp
26          # Add here commands to clean up after the build process.
27          $(MAKE) clean
28          dh_clean
29  install: build
30          dh_testdir
31          dh_testroot
32          dh_clean -k
33          dh_installdirs
34          # Add here commands to install the package into debian/testpack.
35          $(MAKE) DESTDIR=$(CURDIR)/debian/testpack install
36  # Build architecture-independent files here.
37  binary-indep: build install
38  # We have nothing to do by default.
39  # Build architecture-dependent files here.
40  binary-arch: build install
41          dh_testdir
42          dh_testroot
43          dh_installchangelogs
44          dh_installdocs
45          dh_installexamples
46  #       dh_install
47  #       dh_installmenu
48  #       dh_installdebconf
49  #       dh_installlogrotate
50  #       dh_installemacsen
51  #       dh_installpam
52  #       dh_installmime
53  #       dh_python
54  #       dh_installinit
55  #       dh_installcron
56  #       dh_installinfo
57          dh_installman
58          dh_link
59          dh_strip
60          dh_compress
61          dh_fixperms
62  #       dh_perl
63  #       dh_makeshlibs
64          dh_installdeb
65          dh_shlibdeps
66          dh_gencontrol
67          dh_md5sums
68          dh_builddeb
69  binary: binary-indep binary-arch
70  .PHONY: build clean binary-indep binary-arch binary install configure
```

Algunos comentarios al archivo:

Las líneas 11-16 son la columna vertebral de apoyo para los parámetros DEB_BUILD_OPTIONS, controlan si los binarios deben construirse con la tabla de símbolos, o

si deben ser removidos de las instalaciones.
Líneas 18-26 describen el estado *build*, que ejecuta el Makefile para hacerse con el programa para compilarlo.
La norma *clean*, tal como se especifica en las líneas 28-36, limpia cualquier binario que no es necesario.
El proceso de instalación, la regla *install*, comienza en la fila 38.

En el final obtenemos los archivos:

- testpack _0.0.1-1.diff.gz
- testpack _0.0.1-1.dsc
- testpack _0.0.1-1.changes
- testpack _0.0.1.orig.tar.gz
- testpack _0.0.1-1_i386.deb

La descripción más detallada de los archivos, generación y el proceso detallado del empaquetamiento del componente PCI9111IOCard se describirá a continuación.

6.2 Empaquetamiento del componente daIOCard

A continuación se explica como se genera el empaquetamiento del componente daIOCard.
Como fue explicado en el apartado anterior, el primer paso para crear un paquete deb es crear un directorio llamado debian en la raiz del código fuente.

Copiar nuestro código fuente a un directorio nuevo con el siguiente nombre:

```
cppiceccm-1.0.0
```

Ademas de los programas mencionados en el apartado 6.1:

- libc6-dev
- dpkg-dev
- dh_make

Necesitamos tener los programas:

- build-essential: lista de información de los paquetes
- debhelper: programa de ayuda para debian/rules
- fakeroot: Fornece un ambiente con una ruta falsa
- libcomedi-0.7.22: librería para Comedi
- libgcc1 (>= 1:4.2.1): librería de suporte GCC
- libiceutil32: Ice for C++ misc utility library
- libzeroc-ice32: Ice for C++ runtime library

La librería Comedi es nuestro driver para trabajar con la tarjeta PCI9111IOCard. Las librerías ICE son las librerías de desarrollo y de suporte para el middleware utilizado en nuestro componente.

Entramos al directorio de las fuentes y generamos el directorio debian con ayuda de dh-make:

```
dh_make --createorig
```

Esto creará un tar.gz de las fuentes y añadirá el directorio debian con todas las herramientas (*control*, *copyright, changelog* y *rules)* comentadas en el apartado anterior.

Para generar el paquete deb ejecutamos la instrucción:

dpkg-buildpackage -rfakeroot

La compilación de paquetes hay que hacerla como usuario (como root podemos cargarnos algo si tenemos un Makefile mal diseñado)

Este comando llama secuencialmente a lo siguiente:

- make clean
- dpkg-source (genera el tar.gz de las fuentes y el diff.gz de debian)
- configure
- make
- make install DESTDIR=$(CURDIR)/usr/bin/cppiceccm
- "builddeb"

Es decir, se limpian las fuentes, se crea un archivo comprimido con nuestro código fuente y los parches que aplica debian.

Después el proceso es, configurar, compilar y para instalar se usa la variable DESTDIR. Instalara nuestro programa en el directorio de Linux de binarios.

Luego se copian la documentación, el changelog, se generan las dependencias de ejecución mirando el linkado dinámico de nuestros binarios (ldd nuestro_programa) y se coloca todo en uno archivo deb.

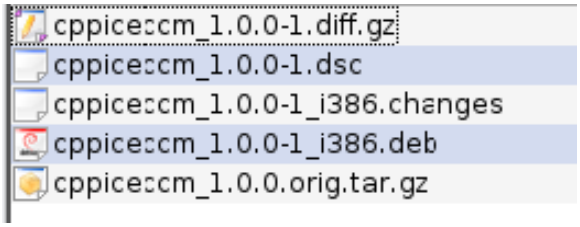

Figura 6-1: Ficheros generados por el empaquetamiento

Los archivos pueden ser consultados en el Anexo D.
El archivo dsc contiene la información del paquete fuente y los hash (md5, sha1) de los archivos diff.gz y orig.tar.gz)

El archivo changes es una copia de lo que vemos en consola cuando se compila o también llamado log de construcción.

Por último el paquete deb, se puede instalar con un simple:

dpkg -i cppiceccm_0.0.1-1_i386.deb

Este paquete será instalado en el directorio indicado por DESTDIR (/usr/bin/cppiceccm).

Será compilado y instalado en otras maquinas distintas desde que los programas mencionados en este apartado estén instalados en esa maquina.

7. Verificación del componente daIOCard

En este capítulo se describen las pruebas que se han realizado para verificar la funcionalidad del componente desarrollado. Por motivo del tiempo disponible son pruebas muy simples orientadas a comprobar que el componente funciona, y satisface los requisitos establecidos.

7.1 Arquitectura del sistema de prueba.

El componente ha sido desarrollado para ser accedido como un elemento remoto dentro de una plataforma distribuida, por ello las pruebas se han planteado con una arquitectura distribuida.

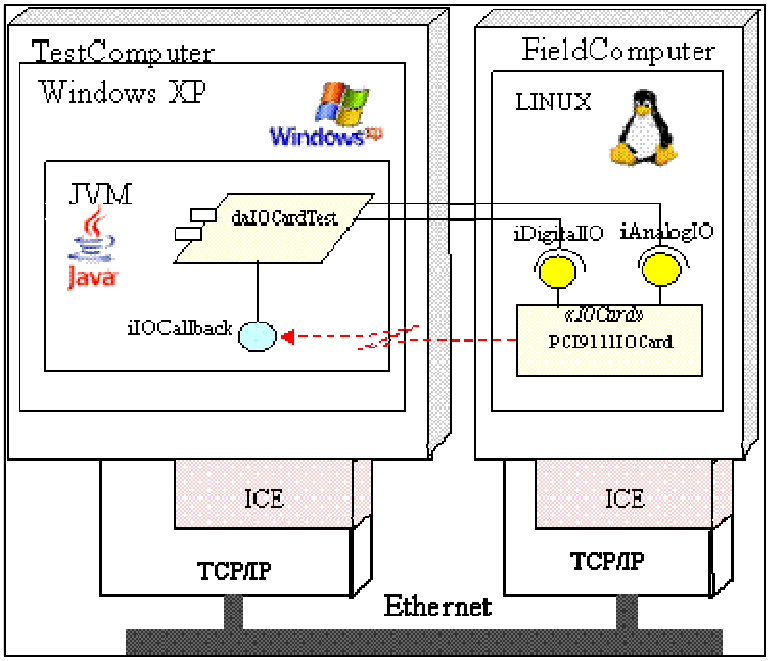

Figura 7-1: Arquitectura

Se ha desarrollado la aplicación Java daIOCardTextJava basada en una interfaz de usuario muy simple que se ejecuta en un procesador Windows. En la siguiente figura se muestra la interfaz.

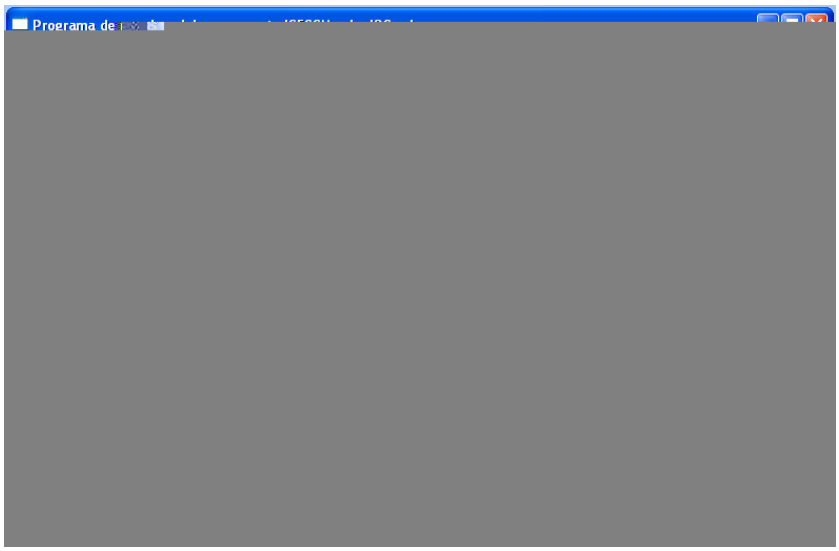

Figura 7-2: GUI pruebas

- Ofrece los campos de texto para introducir el IP, Puerto e Identificador del componente que deben ser inicializados por el operador. Cuando estos estén establecido, pulsando el botón Connect se lleva a cabo el protocolo de conexión, y se realiza una primera prueba.
- Ofrece un conjunto de ocho botones con los que se puede iniciar cada uno de ocho test de prueba.
- Dispone de un campo de texto en el que se muestran los resultados que se generan en el último test realizado.

El componente daIOCard se instala en una plataforma Linux UBUNTU 7.10 Este procesador tiene instalado la tarjeta PCI-9111HR.

En la siguiente figura se muestra el sistema experimental que se está utilizando. Los dos ordenadores que ejecutan el programa de prueba y el componente desarrollado están interconectados mediante Ethernet y ambos están dotados con el middleware ICE de la empresa ZeroC. El segundo computador tiene instalada la tarjeta PCI-9111HR, y a sus entradas y salidas se accede a través de una caja de clemas. Las señales de entrada se controlan mediante un generador de funciones, y las señales de salida se visualizan a través de un osciloscopio.

El procedimiento de prueba es el siguiente:

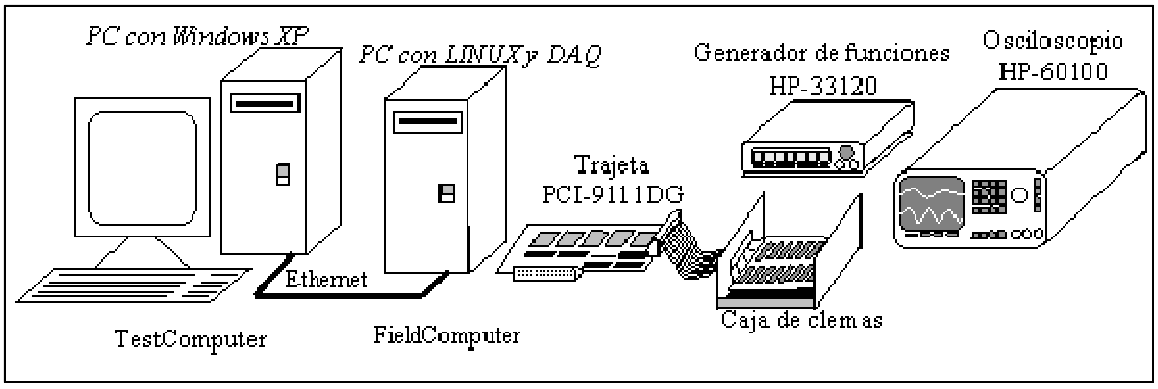

Figura 7-3: Esquema del montaje de pruebas

1. Se instala el componente icePCI9111ioCard en el FieldComputer.
2. Se arranca en el Test Computer la aplicación de prueba daIOCardTest.
3. Se establece en la interfaz de usuario el IP de FieldComputer, y el nombre y el puerto del componente icePCI9111ioCard.
4. Se pulsa el botón Connect, esto hace que el programa de prueba conecte con el componente y verifica la accesibilidad a sus puertos (Test0).
5. Se conectan las líneas de entrada a los equipos externos de prueba. La forma de conexión puede ser muy variada, en función de las pruebas que se deseen hacer. La conexión por defecto que se propone en este protocolo es :
   - Se conectan la tierra analógica y la tierra digital de la tarjeta PCI9111DG a la tierra del generador y a la tierra del osciloscopio.
   - Se conecta la sonda 1 del osciloscopio a la salida del generador de funciones.
   - Las dos primeras y las dos últimas líneas de entrada analógicas (canal0, canal1, canal14 y canal15) y digitales (linea0, linea1, linea14 y línea 15) se conectan a la salida del generador de funciones.
   - El generador de funciones se programará y la segunda sonda del osciloscopio se conecta a una de las salidas digitales o a la salida analógica de la tarjeta, de acuerdo con la prueba que se realiza.
6. Se ejecutan las diferentes pruebas. Para cada una de ellas, se programa el generador y se conecta la segunda sonda del osciloscopio con la configuración que corresponda a la prueba. Se pulsa el botón de la correspondiente prueba, y se analizan los resultados que se generan en la GUI y se observan las formas de ondas que se visualizan en el osciloscopio.

7.2 Descripción de las pruebas realizadas

A continuación se describen las pruebas que se han realizado.

Text 0: Conexión y acceso a funciones básicas:
Esta prueba se ejecuta cuando el programa cliente establece conexión con el componente daIOCard. Comprueba que tiene acceso a la interfaz de referencia del componente, y a sus dos puertos de negocio. Sobre estos realiza alguna operación básica que no requiere señales en los puertos de la tarjeta.
Actividad que ejecuta:

- Genera el Proxy correspondiente a la interfaz de referencia del componente.
- Invoca las funciones de provideFacet() con las claves adecuadas para obtener el Proxy correspondiente al digitalPort y al analogPort.
- Requiere del digitalPort el número de líneas digitales de entrada diNumberOf Lines() y el número de líneas digitales de salida doNumberOfLine()
- Requiere del analogPort el número de canales analógico de entrada aiChannel Number() y el número de canales analógicos de salida aoChannelNumber()
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: No se requiere ninguna conexión.
Formato de los resultados de salida con operación correcta:

```
Componente daIOCard conectado
Número de líneas digitales de entrada: 16
Número de líneas digitales de salida: 16
Número de canales analógicos de entrada: 16
Número de canales analógicos de salida: 1
```

Text 1: Lectura de entradas digitales
Lee las líneas de entrada digital {linea0, linea1, linea14 y línea 15}. Muestra en la consola los valores leídos y el tiempo de lectura.
Actividad que ejecuta:
Para cada una de las cuatro líneas:

- Lee el tiempo actual.
- Lee la entrada digital invocando la operación diRead()
- Lee el tiempo actual
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: No se requiere conexión ninguna.
Formato de los resultados de salida con operación correcta:

```
Lectura de líneas digitales de entrada:
Línea0: false   Tiempo ms: 0
Línea1: true   Tiempo ms: 0
Línea14: true   Tiempo ms: 0
Línea15: false   Tiempo ms: 0
```

Test 2: Establecimiento de las líneas digitales de salida
Establece el estado de las líneas de salida digital linea0, y línea 15 a estado false, y las líneas linea1, linea14 a estado true. Verifica el estado de las líneas de salida que muestra en la consola y evalúa el tiempo de acceso.
Actividad que ejecuta:
Para cada una de las cuatro líneas:

- Lee el tiempo actual.
- Establece la línea digital de salida al estado especificado invocando la operación doWrite()

- Lee el tiempo actual
- Lee el valor establecido en línea digital de salida utilizando la operación doRead().
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: No se requiere ninguna conexión.
Formato de los resultados de salida con operación correcta:

```
Se establecen líneas digitales de salida:
Línea0 se establece a false   Tiempo ms: 0  Valor leído: false
Línea1: se establece a true   Tiempo ms: 0  Valor leído: true
Línea14: se establece a true   Tiempo ms: 0  Valor leído: true
Línea15: se establece a false   Tiempo ms: 0  Valor leído: false
```

Haciendo uso del osciloscopio se debe verificar que las líneas digitales de salida línea0, y línea 15 se encuentran en estado false, y las líneas línea1, línea14 se encuentran en estado true.

Test 3: Lectura de los canales de entrada analógica.
Se ejecuta cuando se pulsa el botón Test3. Lee el valor de las entradas analógicas Channel0, Channel1, Channel14 y Channel15 Muestra en la consola los valores leídos y el tiempo de lectura.

Actividad que ejecuta:

Para cada una de los cuatro canales:

- Lee el tiempo actual.
- Lee el valor analógico del canal analógico de entrada que corresponde invocando la operación aiReadWithGain()
- Lee el tiempo actual
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: Se conectan los canales analógicos de entrada Channel0 y Channel15 a tierra, y los canales Channel1 y Channel14 a una tensión de +6V.
Formato de los resultados de salida con operación correcta:

```
Se leen los canales de entrada analógicos:
Channel0: 0.05    Tiempo ms: 0
Channel1: 6.02    Tiempo ms: 0
Channel14: 5.98   Tiempo ms: 0
Channel15: -0.02  Tiempo ms: 0
```

Test 4: Lectura de los canales de entrada analógica
Se establece un valor en el canal analógico de salida. Verifica el valor de salida registrado y evalúa el tiempo de acceso.

Actividad que ejecuta:

- Lee el tiempo actual.
- Establece el canal analógico de salida a una tensión de 6 v. invocando la operación aoWrite()
- Lee el tiempo actual
- Lee el valor establecido en el canal analógico de salida utilizando la operación aoRead().
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: No se requiere ninguna conexión.
Formato de los resultados de salida con operación correcta:

```
Se establecen el canal analógico de salida:
Channel0: se establece a + 6 V   Tiempo ms: 0  Valor leído: 5.87
```

Haciendo uso del osciloscopio se debe verificar que el canal analógico de salida toma una tensión de aproximadamente 6 voltios.

Test 5: Parpadeo de las salidas digitales.
Se ejecuta cuando se pulsa el botón Test5. Programa las líneas de salida digital para que parpadeen con diferentes frecuencias: la Línea0 con periodo 500 ms, la Línea1 con periodo de 1000 ms, la Línea14 con periodo 1500 ms, la Línea15 con periodo de 2000 ms.

Actividad que ejecuta:

- Para cada una de las cuatro líneas:
  - Se establece el modo parpadeo con la frecuencia especificada, invocando la función doGenerateBlinking().
- Espera 20 ms y estableblece las cuatro líneas a 0
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: No se requiere ninguna conexión.
Formato de los resultados de salida con operación correcta:

```
Se establecen la líneas digitales de salida a modo parpadeo:
Línea0 parpadeo con 500 ms de periodo
Línea1 parpadeo con 1000 ms de periodo
Línea2 parpadeo con 1500 ms de periodo
Línea3 parpadeo con 2000 ms de periodo
```

Haciendo uso del ociloscopio se debe verificar que las líneas digitales de salida línea0, línea1, línea14 y línea15 parpadean a la frecuencia especificada.

Test 6: Generación de pulsos
Se ejecuta cuando se pulsa el botón Test6. En cada línea de salida digital Linea0 se genera un pulso positivo de 1000 ms.

Actividad que ejecuta:

- Establece la línea digital Linea0 a estado false.
- Se espera 2000ms
- Se genera en la línea digital de salida Linea0 un pulso positivo de 1000 ms de duración utilizando la operación doGeneratePulse().
- Imprime los resultados o información sobre las excepciones lanzadas.

Conexión de las líneas físicas: No se requiere ninguna conexión.
Formato de los resultados de salida con operación correcta:

```
Generación de pulso en la linea digital de salida Linea0:
Se establece la línea a cero.
Se inicia el pulso.
```

Haciendo uso del ociloscopio se debe verificar que las líneas digitales de salida línea0, se genera el pulso especificado.

Test 7: Espera del estado de líneas digitales
Se ejecuta cuando se pulsa el botón Test7. Programa cuatro de las líneas digitales de entrada a cada una de la condiciones de espera definidas: Linea0 => AWAIT_TRUE, Linea1 => AWAIT_FALSE, Linea14 => AWAIT_CHANGE_TO_TRUE y Línea15 => AWAIT_CHANGE_TO_FALSE.

Actividad que ejecuta:

- Para cada una de las cuatro líneas:
  - Programa la línea para espera invocando la operación diAwaitEvent()
- Imprime comentario para que el operador suba la tensión.
- Imprime las respuesta mediante la función callback

Conexión de las líneas físicas: Las líneas Linea0, linea1, Linea 14 y Linea 15 se conectan a la salida del generador. Este inicialmente esta a 0 V. y se sube hasta + 5 y posteriormente se vuelve a bajar a 0 V .
Formato de los resultados de salida con operación correcta:

```
Espera a estado de  líneas digitales de entrada:
Línea1 AWAIT_FALSE encontrado.
Subir la señala 5 V y luego bajar a 0V:
Línea0 AWAIT_TRUE encontrado.
Línea14 AWAIT_CHANGE_TO_TRUE encontrado.
Línea15: AWAIT_CHANGE_TO_FALSE encontrado.
```

Haciendo uso del ociloscopio se debe verificar que las líneas digitales de salida línea0, y línea 15 se encuentran en estado false, y las líneas línea1, línea14 se encuentran en estado true.

Test 8: Espera de estados de valores en los canales de entrada analógicos:
Se ejecuta cuando se pulsa el botón Test8. Programa cuatro de los canales de entrada analógica a cada una de las condiciones de espera definidas: Channel0 => AWAIT_LOWER, Channel1=>AWAIT_HIGHER, Channel14=>AWAIT_CHANGE_TO_LOWER, Channel15=> AWAIT_CHANGE_TO_HIGHER.

Actividad que ejecuta:

- Para cada una de los cuatro canales:
  - Programa la tarjeta para que el canal espere la situación especificada invocando la operación aiAwaitValue(), estableciendo como umbral 2.5 V.
- Imprime comentario para que el operador suba la tensión.
- Imprime las respuesta mediante la función callback

Conexión de las líneas físicas: Los canales de entrada analógicos Channel0, Channel1, Channel14 y Channel15 se conectan a la salida del generador. Este inicialmente esta a 0 V. y se sube hasta + 5 y posteriormente se vuelve a bajar a 0 V.

Formato de los resultados de salida con operación correcta:

```
Espera a estado de canal analógico de entrada:
Canal0 AWAIT_LOWER encontrado.
Subir la señala 5 V y luego bajar a 0V:
Canal1 AWAIT_HIGHER encontrado.
Canal15 AWAIT_CHANGE_TO_HIGHER encontrado.
Canal14: AWAIT_CHANGE_TO_FALSE encontrado.
```

8. Conclusiones

En este trabajo se han abordados dos aspectos: el primero relativo a la implementación de la tecnología CCM, y el segundo relativo a la estandarización de los drivers en plataformas LINUX.

Con relación a la tecnología de componentes se ha realizado el proceso completo de implementación de un componente de la tecnología CCM (Container Component Model) en lenguaje de programación C++, en plataforma LINUX y utilizando ICE de la empresa ZeroC, como middleware de comunicaciones:

- Se ha establecido el procedimiento que debe seguirse para deducir a partir de la especificación de un componente, la interfaces que debe implementar el código de negocio de sus implementaciones, a fin de que posteriormente el código pueda ser integrado en cualquier plataforma de ejecución sin ser modificado.
- Se ha propuesto la estructura que debe tener el contenedor que adapta el código de negocio para ser ejecutado en una plataforma LINUX.
- Se ha propuesto la estructura del elemento Executor que es el que proporciona los mecanismos para que los servicios del componente puedan ser accedidos desde componentes clientes que se ejecutan en otros procesadores de la plataforma distribuidos, haciendo uso del middleware ICE.

Para estos tres aspectos se ha seguido las recomendaciones que propone la organización OMG en la especificación LightWeightCCM específica para sistemas embebidos con recursos limitados. El proceso se ha completado y se ha verificado el funcionamiento correcto del diseño, aunque en este momento está aun pendiente su automatización a través de herramientas.

El componente que se ha desarrollado tiene como objeto abstraer las operaciones de entrada/salida analógicas y digitales que se realizan haciendo uso de una tarjeta de entrada/salida comercial. Con el componente desarrollado se garantiza tanto la reutilización del código en cualquier aplicación distribuida que requiera operaciones de entrada/salida, como la independencia de las aplicaciones que lo usan de la tarjeta hardware que en cada caso se utiliza.

El segundo aspecto que se ha abordado, ha sido experimentar con los *drivers* de la familia Comedi. Esta familia ha sido desarrollada dentro de un proyecto de software libre que tiene un doble objeto: conseguir que los *drivers* de los elementos hardware en la plataforma Linux tengan una funcionalidad y un *framework* estandarizado, y facilitar que los propios desarrolladores de software puedan actualizar los *drivers* cuando las versiones de LINUX avanzan. Su uso ha demostrado ser eficaz en estos aspectos, aunque con él, se ha tenido que renunciar a gran parte de la funcionalidad ofrecida por el hardware. En el caso de las tarjetas PCI9111 de entrada/salida se ha comprobado que el uso del *driver* Comedi la funcionalidad que se ha obtenido es sólo una parte de la que se obtenía utilizando los drivers cerrados que proporciona el fabricante del hardware. Es posible que esto sea sólo consecuencia de la inmadurez de la tecnología, y en versiones futuras, los *drivers* tengan capacidad de acceder a la totalidad de los recursos ofrecidos por el hardware.

Referencias

[1] ITEA project MERCED (Market-Enabler for Retargetable COTS components in Embedded Domain). http://www.itea-merced.org.
[2] IST projects: "COMPARE (Component-based approach for real-time and embedded systems)". http://www.ist-compare.org.
[3] IST project: "FRESCOR (Framework for Real-time Embedded Systems based on Contracts)". http://www.frescor.org
[4] M. Henning y M. Spruiell: "Distributed Programming with ICE". http://www.zeroc.com.
[5] The Control and Measurement Device Interface handbook. http://www.comedi.org/
[6] The Open Group. http://www.unix.org/online.html
[7] L.Barros, P.López, J.M. Drake :"Tecnología de componentes CCM basada en conectores" XVI Jornadas de Concurrencia y Sistemas Distribuidos, Albacete 2008.
[8] Guía del nuevo desarrollador de Debian. http://www.us.debian.org/doc/maint-guide/