UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

INSTITUTO UNIVERSITARIO DE SISTEMAS INTELIGENTES Y APLICACIONES NUMÉRICAS EN INGENIERÍA

TRABAJO FIN DE MÁSTER:

# Mejora del Compilador/Generador de Esqueletos C++ para Componentes CoolBOT

MÁSTER OFICIAL EN SISTEMAS INTELIGENTES Y APLICACIONES NUMÉRICAS EN INGENIERÍA

***FRANCISCO JESÚS SANTANA JORGE***
***Las Palmas de Gran Canaria, Octubre de 2009***

Trabajo Fin de Máster (TFM) del Máster Oficial en Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería de la Universidad de Las Palmas de Gran Canaria, presentado por el alumno:


## *FRANCISCO JESÚS SANTANA JORGE*


**Título del Proyecto**: Mejora del Compilador/Generador de Esqueletos C++ para Componentes CoolBOT.

**Tutor**: Dr. D. Antonio Carlos Domínguez Brito.

**CoTutor**: Dr. D. Jorge Cabrera Gámez.

**CoTutor**: Dr. D. Daniel Hernández Sosa.

# Contenido.

# Índice de Figuras.

# Índice de Tablas.

# Índice de Fragmentos de Código.

# Capítulo 1

# Introducción.

En Diciembre del 2007, se presentó como Proyecto de Fin de Carrera el compilador *coolbotc-1.2.0* [1] [Santana-Jorge, 2007]. Este compilador tenía como objetivo automatizar la generación de esqueletos C/C++ a los desarrolladores que emplean la plataforma CoolBOT [Domínguez-Brito et al., 2003] para desarrollar componentes.

Con este compilador, el desarrollador evitaba tener que realizar un proceso repetitivo y tedioso (con tendencia a cometer errores), centrándose únicamente en la implementación de la funcionalidad del componente.

Con el paso del tiempo, tras utilizar el compilador, se observó que si inicialmente no se definía y se diseñaba bien el componente, volvíamos a un proceso costoso, repetitivo y tedioso porque el programador se veía obligado a realizar una re-compilación[2]. Esto significaba, que tras realizar los cambios oportunos en el fichero descriptivo y volver a compilarlo, el programador tendría que volver a implementar la funcionalidad del componente o recuperarla de la copia de seguridad que generaba el compilador.

En el trabajo presentado en este documento, vamos a estudiar el problema que plantea la re-compilación y propondremos una solución que evite al programador este proceso repetitivo y tedioso.

Por tanto, en este primer capítulo, empezaremos estudiando cómo se desarrolla un componente CoolBOT con el compilador *coolbotc-1.2.0* donde plantearemos con más claridad el problema de la re-compilación. Luego, pasaremos a ver los objetivos que se han establecido en este trabajo para terminar con una descripción de lo que veremos en el resto de capítulos de este documento.

[1] Para distinguir el compilador obtenido en el Proyecto de Fin de Carrera del conseguido en este trabajo, al nombre del compilador le añadiremos un guión con la versión del compilador.

[2] En este documento, se denomina re-compilación cuando se vuelve a compilar un componente que previamente ha sido compilado.

Por último, antes de empezar a ver el primer punto de este capítulo, hemos de mencionar que aparte del compilador *coolbotc-1.2.0*, existe otro denominado *xmlcoolbotc*. Este compilador fue desarrollado como anexo a otro Proyecto de Fin de Carrera y, en el presente trabajo, no se tendrá en cuenta.

## *1.1. Estado actual del tema.*

La plataforma o framework CoolBOT permite desarrollar sistemas robóticos mediante el ensamblaje e integración de componentes a modo de puzzle software. Un componente en CoolBOT no es más que una clase en C/C++ que contiene la infraestructura necesaria para implementar un componente software y, para llevar a cabo su desarrollo, se sigue una determinada metodología que se encuentra dividida en fases.

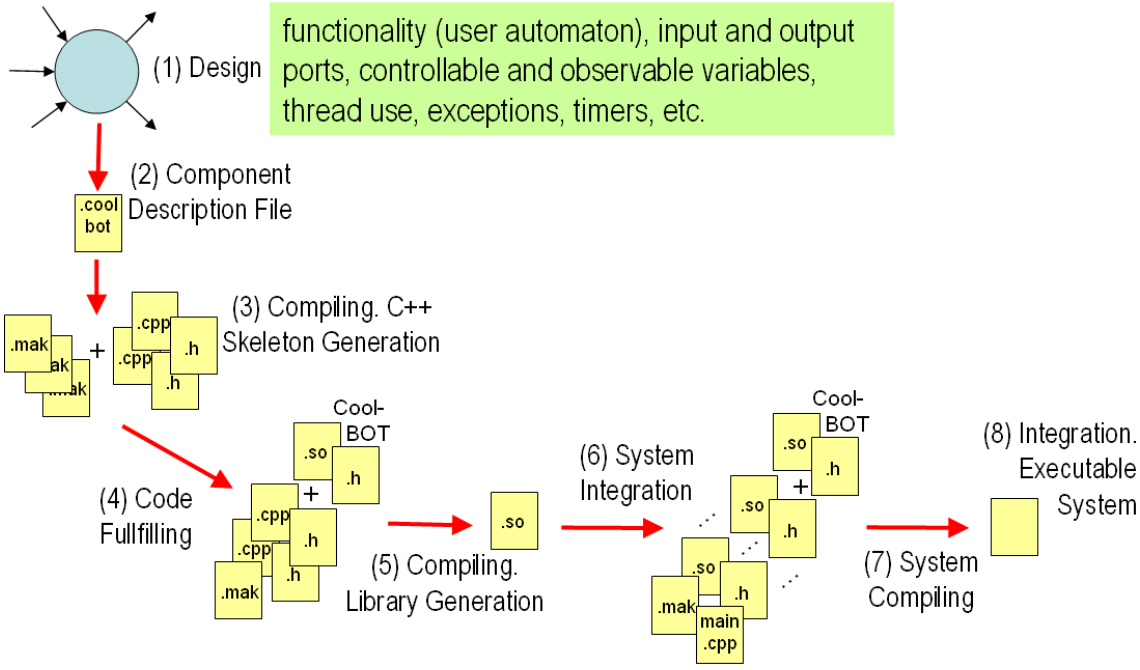


**Figura 1**: Metodología de desarrollo de un componente en CoolBOT.

En la Figura 1, podemos observar la metodología que actualmente se utiliza para desarrollar un componente en CoolBOT con sus respectivas fases que pasamos a comentar brevemente a continuación.

La primera fase es la de *diseño y definición del componente* (fase 1). En esta fase, el programador, en base a un problema real que pretende resolver, definirá un boceto en donde describirá los distintos elementos que necesita para resolver el problema planteado.

Una vez diseñado y definido el componente que queremos implementar, pasamos a redactar el fichero descriptivo (fase 2). El fichero descriptivo no es más que un fichero de texto plano con la extensión "*.coolbot*" donde se define los requisitos planteados en la fase anterior. Los requisitos se definen utilizando el lenguaje *Description Language* (abreviadamente DL) [Santana-Jorge, 2007], objeto de estudio en el capítulo 2.

Para facilitar el trabajo al desarrollador, el compilador *coolbotc-1.2.0* proporciona la opción "*-c*" o "*--create*" que nos genera la estructura de directorios y de ficheros y las variables de entorno necesarias para realizar la compilación con el compilador de C/C++ en la fase 5 que luego veremos. En la Figura 2, podemos contemplar gráficamente lo que realiza esta opción.

Con la opción anterior, se crea un fichero con extensión "*.coolbot*" que contiene la definición mínima de un componente en lenguaje *Description Language*. En el Fragmento de Código 1, podemos ver el contenido en lenguaje DL que genera el compilador *coolbotc-1.2.0* para el componente *First*.

```
/*
 * File: first.coolbot
 * Description: description file for First component.
 * Date: 23 September 2009
 * Generated by coolbot 1.2.0
 */

component First
{

    /*
     * State's definition.
     */

    entry state Main
    {
        //State's body.
    }

};
```

**Fragmento de Código 1:** Definición en lenguaje DL del componente First generado por el compilador.

A continuación, cuando disponemos del fichero descriptivo, pasamos a generar los correspondientes esqueletos C/C++ con el compilador *coolbotc-1.2.0* (fase 3). Estos esqueletos contienen la infraestructura necesaria para poder llevar a cabo la funcionalidad establecida por el programador y son generados automáticamente por el compilador a partir del fichero descriptivo.

Como el programador se puede equivocar escribiendo código en lenguaje DL, el compilador, aparte de informar de las advertencias y errores que se detectan durante la compilación, ofrece la opción "*-v*" o "*--verbose*" y la opción "*-vf*" o "*--verfosefile*" para realizar una traza del componente que estamos compilando.

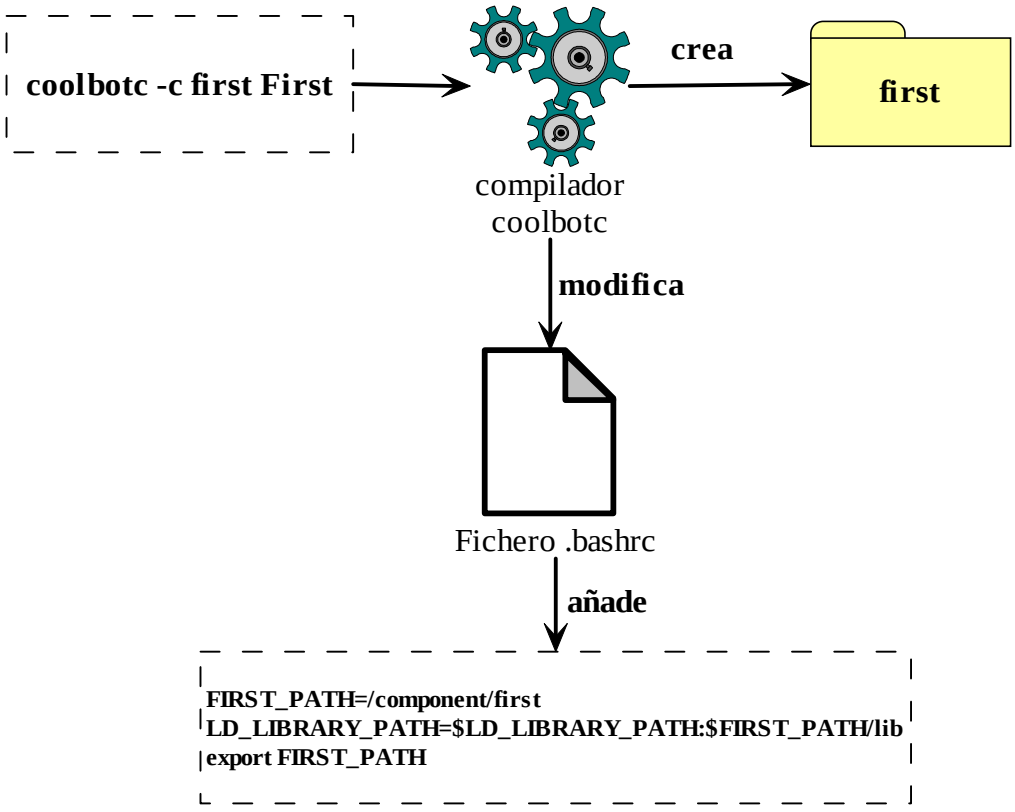


**Figura 2**: Ejecución del compilador "*coolbotc-1.2.0*" con la opción "*-c*" o "*--create*".

Disponiendo de los esqueletos en C/C++, el programador pasa a implementar la funcionalidad del componente CoolBOT (fase 4), completando las distintas partes del esqueleto para obtener la funcionalidad deseada.

Una vez implementado el componente, éste se compila con el compilador de C/C++ para obtener una biblioteca del mismo (fase 5). La biblioteca que se obtiene es la que luego se integra en un sistema robótico (fase 6) y que, tras compilarlo (fase 7) se genera el ejecutable (fase 8).

Para obtener la biblioteca se requiere realizar unas determinadas compilaciones y conocer dónde se encuentra instalada la plataforma CoolBOT. Luego, para poder utilizar la biblioteca obtenida, se debe definir una nueva variable de entorno y modificar otra (variable *LD_LIBRARY_PATH*). La variable de entorno que se ha de crear especifica la ruta donde se encuentra el nuevo componente y el contenido de esta nueva variable se añade a la variable de entorno *LD_LIBRARY_PATH* que es empleada por el

enlazador dinámico para encontrar las rutas alternativas de búsqueda de bibliotecas dinámicas de funciones del sistema. En la Figura 2, podemos ver cómo se ha creado una variable de entorno llamada *FIRST_PATH* y cómo se añade su contenido a la variable de entorno *LD_LIBRARY_PATH*.

El compilador *coolbotc-1.2.0* facilita las fases 5 y 6 al programador. Con la opción "*-c*" o "*--create*" se generan automáticamente los ficheros *makefiles*[3] para realizar las compilaciones necesarias con el compilador de C/C++, y se crean y se modifican las variables de entorno necesarias. En la Figura 3, vemos la estructura de directorios y de ficheros que crea el compilador con la opción "*-c*" o "*--create*", donde se aprecian tres ficheros *makefiles*.

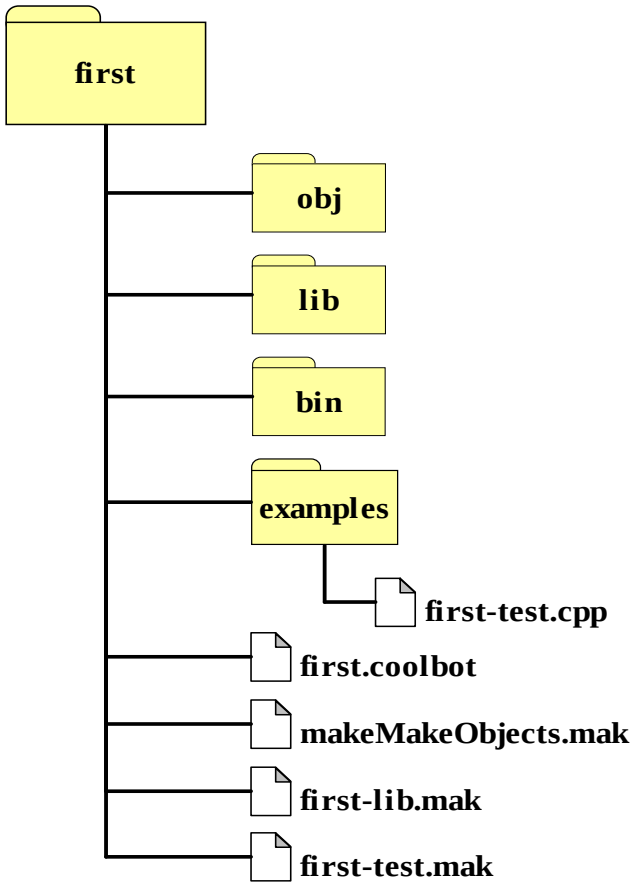


**Figura 3**: Estructura de directorios y de ficheros creados con la opción "*-c*" o "*--create*".

El fichero *makeMakeObjects.mak* se encarga de generar toda la lógica necesaria para poder realizar la compilación con el compilador de C/C++. El fichero terminado con el sufijo "*-lib.mak*" se encarga de compilar el componente definido en formato librería y el fichero con el sufijo "*-test.mak*" genera todos los binarios correspondientes

[3] Visitar para más información: http://www.gnu.org/software/make/manual/make.html

a los programas de prueba implementados en el subdirectorio *examples*. Por defecto, en el directorio *examples* se encuentra un fichero terminado con el sufijo "*-test.cpp*" que contiene un pequeño ejemplo con el cual se puede probar el componente implementado.

Todo componente CoolBOT que se desarrolle debe seguir la metodología que acabamos de ver. De las fases mencionadas, las cuatro primeras son las que presentan mayor dificultad y en las que el desarrollador debe prestar mucha atención.

Si un programador se encuentra en la fase 4 y se da cuenta de que debe añadir nuevos requisitos porque ha definido mal su componente o simplemente debe modificar su diseño, tiene que regresar a la fase 1 y desde ahí volver a repetir la fase 2, la fase 3 (se produce una re-compilación) y la fase 4. Esto no sería un problema si no fuese porque el compilador siempre genera nuevos esqueletos C/C++, los cuales se encuentran vacíos (el código introducido por el programador en la fase 4 no se conserva al volver a compilar el fichero descriptivo). Por tanto, tras una compilación, el programador tiene que volver a programar la misma funcionalidad que ya había implementado con anterioridad a la modificación de requisitos, lo que se traduce en un proceso repetitivo y tedioso con tendencia a provocar errores.

Este problema no lo soluciona el compilador *coolbotc-1.2.0* ya que cuando detecta que se ha producido una re-compilación, simplemente realiza una copia de seguridad que conserva los esqueletos de la compilación anterior. Luego, el programador debería recorrer cada uno de los esqueletos y copiar el código que él ha introducido en los esqueletos resultantes de la nueva compilación. En la Figura 4, podemos ver un esquema visual sobre la compilación y re-compilación que realiza actualmente el compilador *coolbotc-1.2.0*.

En base a esto y teniendo en cuenta las experiencias obtenidas con el compilador *desc* para componentes DES [Domínguez-Brito et al., 2000b], se observo que utilizando un sistema de etiquetas parecido al usado por el lenguaje XML o el lenguaje HTML se podían realizar re-compilaciones sin pérdida de información, de forma que el programador, en caso de interrumpir su tarea, conserva lo que ha implementado en los esqueletos y puede continuar el trabajo por donde lo dejó. En el capítulo 3, podremos ver con más claridad cómo se ha aplicado esta solución.

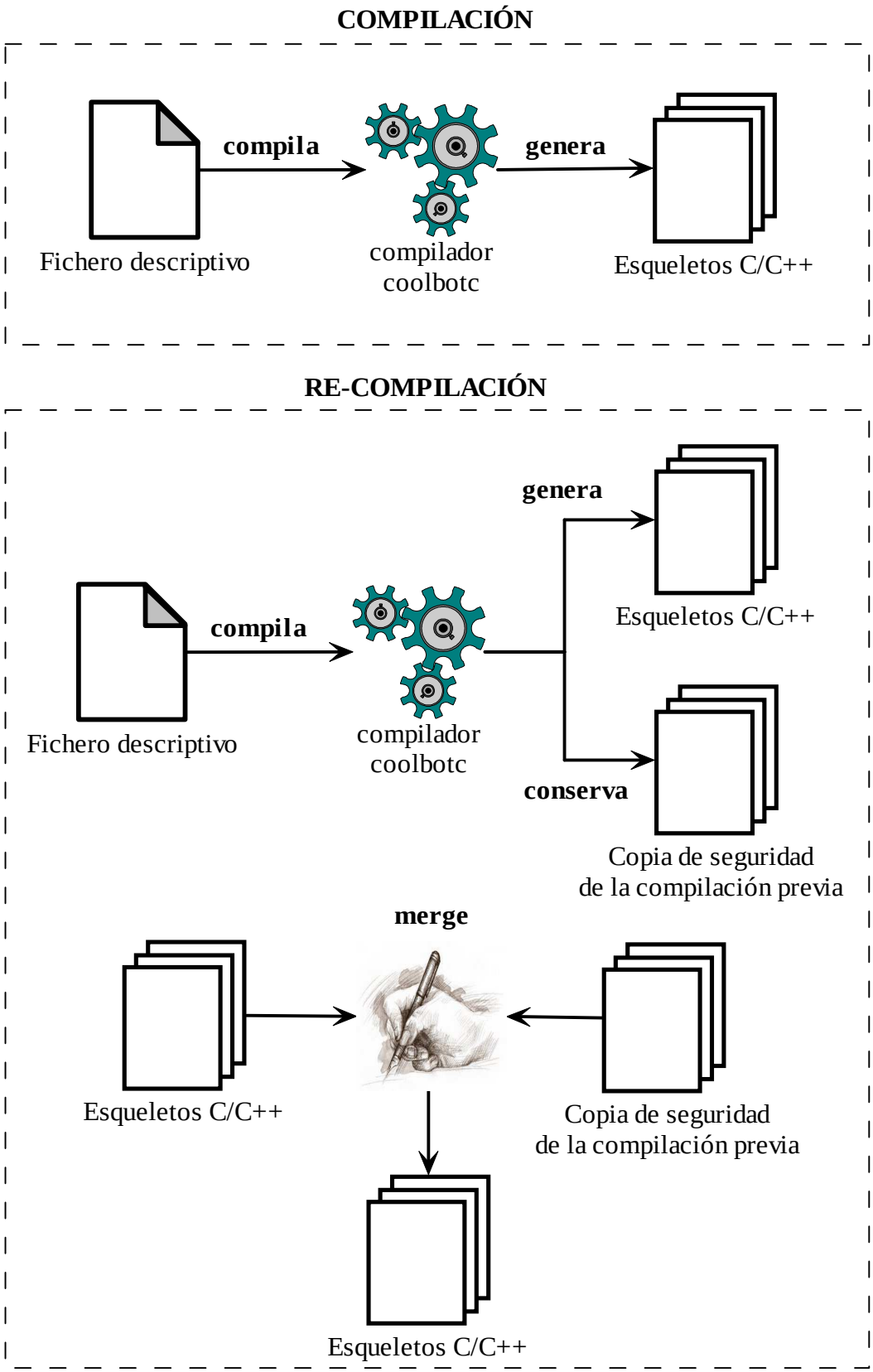


**Figura 4**: Proceso de compilación y re-compilación que realiza el compilador "*coolbotc-1.2.0*".

## *1.2. Objetivos.*

El objetivo de este Trabajo de Fin de Máster es dar una solución al problema que actualmente sufren los programadores que desarrollan componentes CoolBOT cuando deben realizar una re-compilación del fichero descriptivo.

Para ello se va a proceder a realizar una mejora del compilador *coolbotc-1.2.0* para añadir un sistema que permita realizar una re-compilación del fichero descriptivo sin pérdida de información, preservando los cambios introducidos por el programador en los esqueletos.

Este nuevo compilador debe cumplir con los siguientes requisitos que pasamos a comentar:

1. Debe mantener los mismos requisitos del compilador *coolbotc-1.2.0*. Estos requisitos son los siguientes:
    a. Soportar en su totalidad el lenguaje *Description Language* que contempla todas las características de un componente CoolBOT.
    b. Debe proporcionar al programador información sobre la compilación que se está realizando con fines de depuración, informando sobre los errores y advertencias que se produzcan durante el proceso.
    c. Si en la plataforma CoolBOT se incorporan nuevas funcionalidades que impliquen las modificación de los esqueletos C/C++ en un cierto grado, se deben proporcionar medios para que de un modo fácil y sencillo se puedan aplicar estas nuevas funcionalidades sin tener que modificar el compilador.
    d. Deberá proporcionar los medios necesarios para poder generar automáticamente toda la estructura necesaria de ficheros y directorios que exige la plataforma CoolBOT y deberá poder crear aquellas variables de entorno que sea necesario, de forma que el programador, una vez rellenado el fichero descriptivo y compilado, pueda compilar el componente con el compilador C/C++.
    e. Si el programador define nuevos paquetes de puertos (ver el apartado 2.5.2.1 de este documento), aparte de los esqueletos C/C++ del componente se deberán generar los correspondientes a los nuevos paquetes de puertos que se hayan definido.

2. Deberá permitir a los programadores realizar una re-compilación de un componente sin pérdida de información. De esta forma, un componente se puede compilar cuantas veces sea necesario sin perder la información introducida por el usuario en cada re-compilación del fichero descriptivo.
3. Deberá proporcionar un sistema de vuelta atrás en caso de que se produzca algún error durante el proceso de la re-compilación. Este sistema debe dejar los esqueletos C/C++ como estaban antes de realizar la re-compilación.

## *1.3. Contenido de este documento.*

Este documento ha sido organizado en los siguientes 5 capítulos:

- **Capítulo 1: Introducción**. Se trata de este capítulo de introducción.
- **Capítulo 2: El lenguaje Description Language**. En este capítulo estudiaremos el lenguaje descriptivo DL y veremos ejemplos de componentes CoolBOT definidos en dicho lenguaje.
- **Capítulo 3: Desarrollo del trabajo.** Es un capítulo donde comentaremos qué solución hemos aplicado para resolver el problema planteado, así como de que forma se ha diseñado e implementado para obtener una versión operativa del nuevo compilador.
- **Capítulo 4: Prueba y resultados**. Disponiendo de una versión operativa del software, pasamos a evaluar la solución aplicada y a comparar resultados.
- **Capítulo 5: Conclusiones y trabajo futuro**. En este último capítulo se indicarán las conclusiones obtenidas durante el desarrollo de este proyecto. Asimismo se describirán otras posibles mejoras futuras que se pueden plantear.

# Capítulo 2

# El lenguaje Description Language.

El lenguaje *Description Language* (también conocido como lenguaje DL o lenguaje descriptivo) fue definido en [Santana-Jorge, 2007] y se trata de un lenguaje que simplemente describe los elementos que proporciona la plataforma CoolBOT para desarrollar componentes CoolBOT.

En este capítulo, estudiaremos en profundidad los distintos elementos que proporciona el lenguaje, para terminar viendo la definición de un componente completo. Para obtener mayor información sobre los elementos en que se basa el lenguaje DL y la plataforma CoolBOT se recomienda leer [Domínguez-Brito et al., 2003].

## *2.1. Introducción.*

El lenguaje DL se diseñó para reflejar las principales características que proporciona la plataforma CoolBOT para desarrollar componentes CoolBOT. A continuación, antes de empezar a estudiar el lenguaje DL, vamos a comentar brevemente las características principales de la plataforma CoolBOT.

La plataforma CoolBOT se concibe como un marco de programación *orientado a componentes* con el fin de proporcionar modularidad y facilitar la integración entre componentes software. De esta forma, se puede definir funcionalmente un sistema robótico completo mediante la integración de componentes. Además, todos los componentes se definen como un *autómata de puertos* [Steenstrup et al., 1983] [Stewart et al., 1997] [Domínguez-Brito et al., 2000a], lo que proporciona una interfaz uniforme y establece una clara distinción entre la funcionalidad interna de una entidad activa (el autómata) y su interfaz externa (los puertos de entrada y de salida). Por último, todo componente debe ser observable y controlable en cualquier instante tanto desde el exterior como desde el interior del componente para proporcionar robustez y controlabilidad.

Partiendo de lo comentado en el párrafo anterior, en la plataforma CoolBOT se distinguen los siguientes elementos:

- **Puertos de entrada y puertos de salida**: Es un mecanismo de intercambio de información que posee la plataforma. Una pareja formada por un puerto de salida y un puerto de entrada constituye una *conexión de puertos*, donde los datos se trasmiten a través de él en unidades discretas llamadas *paquetes de puertos*, que son enviadas desde los *puertos de salida* y recibidas desde los *puertos de entrada*. Estos paquetes de puertos se clasifican según un *tipo de puerto* y cada puerto de entrada o de salida puede aceptar solamente un conjunto específico de tipos de paquetes de puertos. Para establecer una conexión de puertos, ambos puertos de entrada y de salida deben ser compatibles, es decir, ambos puertos deben aceptar el mismo tipo de paquetes de puertos.
- **Variables observables y variables controlables**: Para proporcionar robustez y controlabilidad a los componentes, se introdujeron los conceptos de variables observables y variables controlables que a continuación pasamos a definir:
    - ***Variable Observable***: Deberían ser declaradas como variables observables aquellos aspectos del componente que se desea exportar externamente. Cualquier cambio en un variable observable dentro de un componente es publicada para su observación externa a través de un puerto de monitorización que proporciona CoolBOT.
    - ***Variable Controlable***: Es aquella que dentro de un componente representa un aspecto del mismo que puede ser controlado, es decir, modificada o actualizada desde el exterior del componente. Las variables controlables son accedidas a través del puerto de control que proporciona CoolBOT.
- **Excepciones**: Para proporcionar mayor robustez, CoolBOT incorpora este mecanismo para tratar de recuperarse de cualquier error que se produzca en el componente o para dejar el componente en un estado controlable.
- **Estados**: Como hemos comentado anteriormente, todo componente es un autómata de puertos. Para ello, todo componente es modelado usando dos autómatas:
    - **Autómata por Defecto**: Autómata que contiene todos los posibles caminos de control de un componente.

- **Autómata de Usuario**: Se encuentra insertado dentro del autómata por defecto y proporciona la funcionalidad específica del componente.

- **Hilos de puertos**: Todos los componentes CoolBOT cuando se ejecutan se encuentran mapeados en hilos y su ejecución se encuentra guiada por los paquetes de puertos que reciben a través de un conjunto de puertos de entrada sobre los cuales son responsables. En CoolBOT, existe un hilo especial presente en todo componente que se denomina hilo **main**, y que controla la ejecución del componente.

## *2.2. Nociones básicas del lenguaje.*

El lenguaje DL es un lenguaje sensible a mayúsculas y minúscula pero con una peculiaridad, los identificadores empleados están exentos de esta regla por lo que no es posible definir el identificador "*First*" y el identificador "*first*" porque para el lenguaje son el mismo identificador.

La unidad principal del lenguaje es el componente y todo gira en torno a él. En cada fichero descriptivo sólo se puede definir un único componente. Como mínimo se debe especificar dentro de la definición de un componente, un estado de usuario.

### 2.2.1. Estructura de un componente.

En todo componente expresado en el lenguaje DL se pueden distinguir dos partes: la parte de declaración y la parte de definición y esta distinción se mantiene en todas las sentencias[1] que define el lenguaje DL.

La parte de declaración de un componente comienza con la palabra reservada *component* seguido de un identificador y la parte de definición se encuentra encerrada entre llaves (símbolos "*{*" y "*}*") y finaliza con un punto y coma (símbolo "*;*"). Entre las llaves se pueden declarar todas las posibles sentencias que permite el lenguaje DL. En la Figura 5 podemos apreciar gráficamente las partes que conforman la definición de un componente.

[1] Una sentencia es la unidad lógica completa más simple que tiene sentido computacional en sí misma, que proporciona sentido computacional completo y que está presente en todo lenguaje de programación.

Cuando definimos más de una sentencia, al igual que pasa con el lenguaje C/C++, debemos emplear el símbolo "*;*" como separador entre una sentencia y otra. Si sólo se especifica una sentencia, el uso del símbolo "*;*" es opcional.

No existe un orden de definición de sentencias, y cada sentencia (excepto la sentencia de documentación cabecera) pueden definirse más de una vez. Como mínimo debe aparecer una sentencia de definición de estados. El resto de sentencias son opcionales, dejando en manos del programador la responsabilidad de definirlos o no. En el Fragmento de Código 1 del capítulo anterior, podemos apreciar la definición mínima de un componente en lenguaje DL.

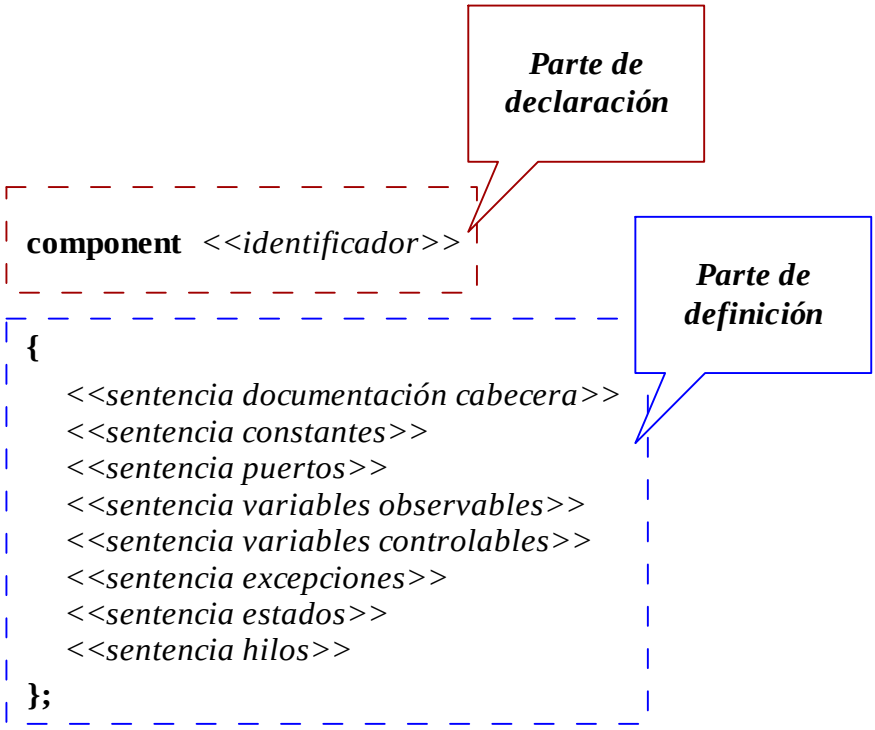


**Figura 5**: Estructura de un componente en lenguaje DL.

### 2.2.2. Definición de comentarios.

Para documentar el código, DL dispone de la posibilidad de incluir comentarios que se definen de la misma forma que en el lenguaje C/C++ por lo que podemos distinguir dos tipos de comentarios:

- *Los comentarios de una línea*. Comienzan con el par de símbolos "//" y se dan por terminados al final de la línea.
- *Los comentarios de múltiples líneas*. Todos los comentarios empiezan con el par de caracteres "/*" y terminan con el par "*/". No pueden haber espacios entre el

asterisco y la barra. Este tipo de comentario no se pueden anidar, es decir, un comentario de múltiples líneas no puede contener otro comentario del mismo estilo.

Se pueden colocar comentarios en cualquier lugar de la definición del componente, siempre y cuando no aparezcan en mitad de una palabra reservada o de un identificador ya que produciría un error. Tampoco se recomienda que se ponga en medio de una expresión porque enturbia su significado. Por lo general, se deben incluir comentarios siempre que sea necesario explicar el funcionamiento del código.

En el Fragmento de Código 2, podemos ver dos ejemplos de un componente llamado *First*. En un ejemplo se hace buen uso de los comentarios y en el otro no, al introducirse errores en los mismos.

```
/*
 * Definición correcta.
 * Comentario de múltiples líneas.
 */
component First
{
    /*
     * Comentario de múltiples
     * líneas.
     */

    entry state Main
    {
        //Comentario de una línea.
    }
};
```

```
/*
 * Definición correcta.
 * Comentario de múltiples
 * líneas.
 */
component First
{
    /*
     Este comentario de
     múltiples líneas

       /*
        Es incorrecto al estar
        Anidado dentro de otro
        comentario
        */

     */
    entry state Main
    {
        /*
           Otro comentario
           Incorrecto al no
           colocar el cierre de
           comentario.
    }

};
```

**Fragmento de Código 2**: Ejemplo de comentarios escritos en lenguaje DL.

### 2.2.3. Definición de identificadores.

Un identificador es una secuencia de caracteres cuya longitud puede variar entre uno y varios caracteres pero sólo los 65 primeros son significativos. Todo identificador se especifica a través de un nombre, el cual debe cumplir las siguientes reglas:

- Debe comenzar con una letra.
- Después del primer carácter puede venir una combinación de letras, dígitos y el carácter subrayado o *undescore* (el símbolo "_"). No puede contener espacios en blanco ni emplear caracteres acentuados ni símbolos especiales.
- No se puede usar como nombre de un identificador palabras reservadas del lenguaje DL así como de la plataforma CoolBOT y del lenguaje C/C++.
- Los identificadores que se definen son únicos, es decir, una vez que se definen no se pueden volver a utilizar para definir otro elemento.
- Los identificadores no son sensibles a mayúsculas y a minúsculas por tanto, en DL, el identificador "*First*" es igual al identificador "*first*".

Para comprender mejor estas reglas, veamos a continuación una serie de definiciones de identificadores correctos e incorrectos. Empezamos con las definiciones de identificadores correctos:

- *MyPionner*.
- *My_Player_Robot*.
- *EsteEsUnComponenteCuyoNombreEsMuyLargoEnTotal47*.
- *Componente_*.
- *simple_component*.
- *A_____1*.

En cuanto a las definiciones de identificadores incorrectos podemos ver los siguientes ejemplos:

- *_Prueba*.
- *My-Player-Robot*.
- *$Variable*.
- *component*.
- *class*.
- *empty_transition*.
- *while*.

- *1_dos_tres*.
- *Camión*.

Se recomienda, como regla de estilo, que el primer carácter del nombre del identificador sea una letra en mayúsculas y el resto en minúsculas. Si el nombre del identificador está formado por varias palabras, se pondrán todas las palabras seguidas unas de otras poniendo la primera letra de cada palabra en mayúscula y el resto en minúscula. Para entenderlo mejor veamos los siguientes ejemplos:

- PlayerRobot.
- PublicInPortOdometry.
- First.

### 2.2.4. Definición de valores literales.

Un literal es toda notación de representación de un valor de código fuente dentro de un lenguaje de programación. En DL podemos distinguir dos tipos de literales:

- *Literales cadena*. Se definen como una secuencia de caracteres encerrados entre dobles comillas donde los primeros 255 caracteres son significativos. Es decir, la longitud máxima permita es de 255 caracteres y a partir de esa cantidad se trunca la cadena. Los literales cadena no se pueden anidar, es decir, dentro de un literal cadena no podemos definir otro.
- *Literales numéricos*. Se definen como un número entero y existen dos variantes: los positivos y los negativos. El uso de números negativos está restringido a la definición de constantes.

Para comprender mejor cómo se define un valor literal, veremos a continuación una serie de ejemplos de definiciones de valores literales tanto correctos como incorrectos:

- *"Esto es un literal cadena"* (definición correcta de un literal cadena).
- *900* (definición correcta de un literal numérico).
- *-1* (definición correcta de un literal numérico).
- *0* (definición correcta de un literal numérico).
- *"Esto también*
  *lo permite si no*
  *me paso de 255*
  *caracteres"* (definición correcta de un literal cadena).

- *"Esto no es correcto "no se puede" anidar literales cadena"* (definición incorrecta de un literal cadena).
- "*Se me olvido el cierre* (definición incorrecta de un literal cadena).

### 2.2.5. Palabras reservadas.

En la Tabla 1, podemos ver todas las palabras reservadas que posee el lenguaje DL.

| ***Palabras reservadas*** | | | |
|---|---|---|---|
| active | answer | attempts | controllable |
| box | component | constants | exception |
| description | each | entry | handler |
| failure | fifo | generic | input |
| has | header | in | length |
| institution | last | lazymultipacket | output |
| multipacket | observable | on | polling |
| packet | packets | period | priority |
| port | poster | priorities | range |
| private | public | pull | state |
| recovery | request | with | timeouts |
| success | thread | tick | variables |
| transition | type | ufifo | |
| version | watchdog | author | |

**Tabla 1**: Palabras reservadas del lenguaje DL.

Además de las mostradas en la Tabla 1, se consideran también palabras reservadas todas aquellas palabras reservadas del lenguaje C/C++ y de la plataforma CoolBOT.

## *2.3. Documentación de cabecera (header).*

DL proporciona otro mecanismo de documentación aparte de los comentarios. Este mecanismo denominado documentación de cabecera, que de aquí en adelante lo llamaremos *header*, nos permite añadir una determinada información al componente. Esta información es utilizada posteriormente por el compilador, el cual la añade dentro de un comentarios en los esqueletos C/C++ resultantes.

La definición del *header* es opcional y si se define sólo puede existir una definición por componente. En caso de haber más de una definición de *header*, sólo se tiene en cuenta la primera definición que aparezca en el fichero descriptivo. El resto son ignoradas.

### 2.3.1. Sintaxis.

En la Figura 6, podemos ver la sintaxis de definición del *header*. La declaración comienza con la palabra reservada *header* seguida de la definición encerrada entre llaves, donde se especifican una serie de atributos. Aunque todos los atributos que se pueden usar en la definición del *header* son opcionales, es obligatoria la definición de uno como mínimo. En la definición no puede haber atributos duplicados y en el caso de haberlos, sólo se tiene en cuenta la primera definición del atributo, el resto de definiciones se omite.

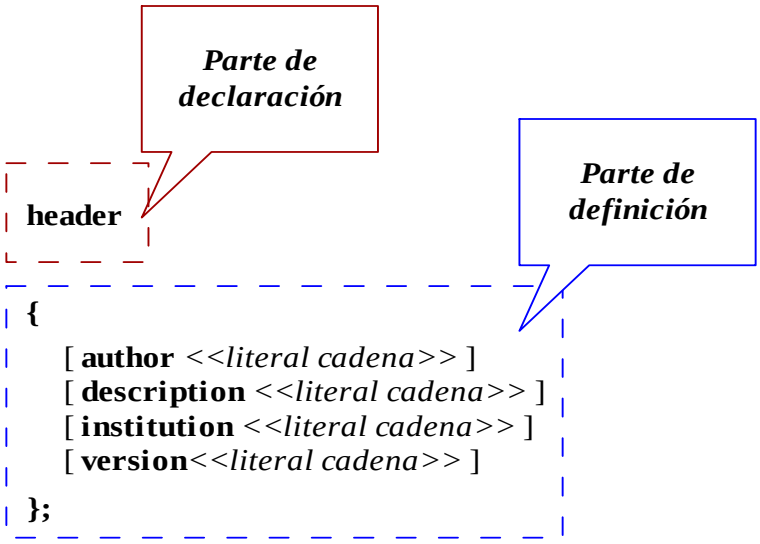


**Figura 6**: Sintaxis de definición de *header*.

Los corchetes que aparecen en la Figura 6, significa que el atributo es opcional.

### 2.3.2. Atributos del header.

Cada atributo empleado en la definición del *header* almacena un tipo concreto de información y los atributos que podemos definir son:

- atributo *author*: permite especificar información sobre el autor o autores del componente. Se define utilizando la palabra reservada *author* seguido de un literal cadena que contiene la información.
- atributo *description*: permite especificar una breve descripción sobre el componente. Se define utilizando la palabra reservada *description* seguida de un literal cadena que contiene la información.
- atributo *institution*: permite especificar la institución o instituciones para las cuales se desarrolla el componente. Se define utilizando la palabra reservada *institution* seguida de un literal cadena que contiene la información.
- atributo *version*: permite especificar la versión del componente que se está definiendo. Se define utilizando la palabra reservada *version* seguida de un literal cadena que contiene la información

### 2.3.3. Ejemplo.

A continuación, veremos una serie de ejemplos donde se define el *header* de forma correcta e incorrecta. En el Fragmento de Código 3, podemos ver dos ejemplos donde se define el *header* de forma correcta y en el Fragmento de Código 4, podemos ver otros dos ejemplos donde se define el *header* de forma incorrecta.

```
component First
{
  header
  {
    author "Francisco J. S. J.
            Antonio C. D. B.";
    description "Prueba.";
    institution "U.L.P.G.C.";
    version "1.1.0"
  };

  entry state Main
  {

  }
};
```

```
component First
{
  header
  {
    author "Francisco J. S. J.";

    //La siguiente definición del
    //atributo author se omite.

    author "Antonio C. D. B.";
    description "Prueba.";
    institution "U.L.P.G.C.";
    version "1.1.0"
  };

  entry state Main
  {

  }
};
```

**Fragmento de Código 3**: Definiciones correctas del *header*.

```
component First
{
  header
  {
    author "Francisco J. S. J.
            Antonio C. D. B.";
    description "Prueba.";
    institution "U.L.P.G.C.";
    version "1.1.0"
  };

  /*
   * Definición errónea:
   * se ha duplicada el header.
   */
  header
  {
    author "Francisco J. S. J.
            Antonio C. D. B.";
    description "Prueba.";
    institution "U.L.P.G.C.";
    version "1.1.0"
  };

  entry state Main
  {

  }
};
```

```
component First
{
  header
  {

  /*
   * Definición errónea: no se ha
   * definido ningún atributo.
   */

  };

  entry state Main
  {

  }
};
```

**Fragmento de Código 4**: Definiciones incorrectas del *header*.

## *2.4. Constantes.*

Una constante es un dato cuyo valor se establece en el momento de la compilación y que permanece inalterado durante la ejecución de un componente.

Las constantes se han introducido en el lenguaje DL ya que resulta muy útil para definir longitudes de paquetes de puertos, números de intentos de recuperación en las excepciones, etc., de modo que podemos modificar dichos valores sin tener que ir a cada línea donde se definen dichos campos.

Su definición es opcional y, siempre y cuando no estén duplicadas, en la parte de definición, podemos definir cuantas constantes deseemos, distinguiéndose dos tipos de constantes: las constantes públicas y las constantes privadas.

### 2.4.1. Sintaxis.

En la Figura 7, podemos ver la sintaxis que se emplea para definir constantes. La declaración comienza con la palabra reservada *constants* seguido de la definición encerrada entre llaves, donde se especifican una o más constantes.

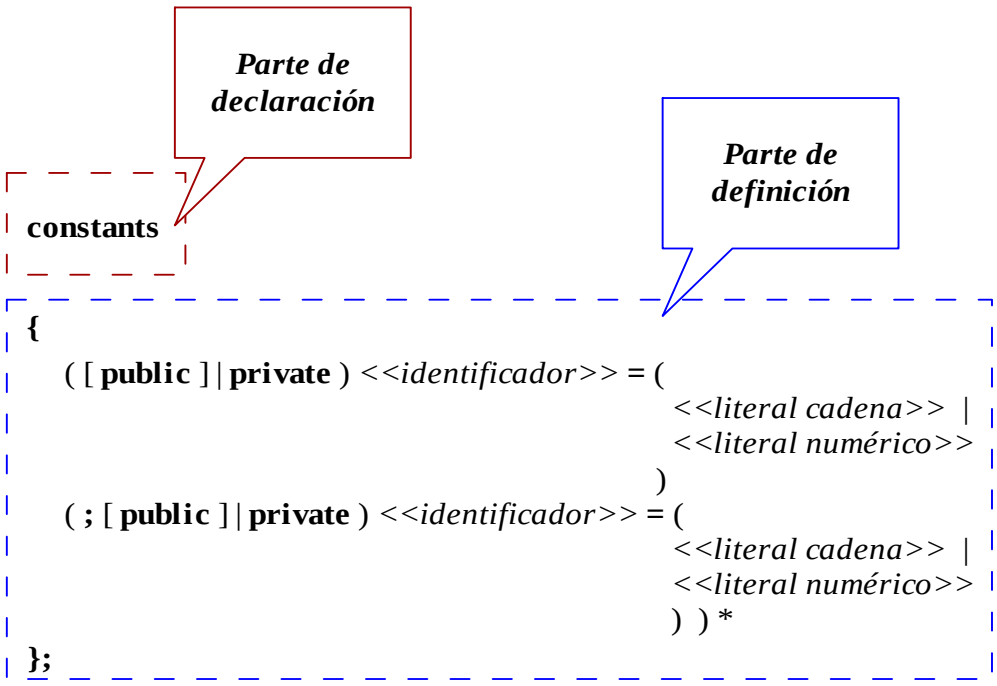


**Figura 7**: Sintaxis de definición de constantes.

En la Figura 7, los paréntesis que aparece significan asociación de elementos, la barra vertical significa o uno u otro y el asterisco significa cero o muchos.

La definición de constante comienza especificando el tipo de constante que vamos a definir. Si la constante que se va a definir es privada se antepone la palabra reservada *private* y si es pública o no se especifica nada o se antepone la palabra reservada *public*. A continuación, se define el identificador de la constante y que se construye siguiendo las reglas descritas en el apartado 2.2.3 del presente capítulo. Al identificador le sigue el símbolo "=" y del valor de la constante. El valor de la constante se define con un literal cadena o con literal numérico.

### 2.4.2. Tipos de constantes.

Como se ha dicho, en el lenguaje DL podemos distinguir dos tipos de constantes: las constantes públicas y las constantes privadas. Las constantes declaradas como públicas simplemente indican que su contenido se especificará en el área pública del esqueleto C++ que genera el compilador de DL. Lo mismo ocurre con las constantes privadas, cuyo contenido se especifica en el área privada del esqueleto C++. El valor que se le asigna a la constante es un valor literal cuyo formato hemos visto anteriormente.

### 2.4.3. Ejemplo.

A continuación, en el Fragmento de Código 5 podemos ver ejemplos donde se definen constantes de forma correcta e incorrecta.

```
/*
 * Definición correcta.
 */
component First
{
  constants
  {
    FRECUENCE = "A::Frecuence";
    private RANGE = -90;
  };

  entry state Main
  {

  }

  constants
  {
    public LENGHT = 100;
  };

};
```

```
/*
 * Definiciones incorrectas.
 */
component First
{
  constants
  {
    /*
     * Debe definirse al menos una
     * constante.
     */
  };

  constants
  {
    public LENGHT = 100;

    /*
     * Definición duplicada de
     * constante.
     */

    public LENGHT = 100;
  };

  entry state Main
  {

  }

  /*
   * Definición duplicada de
   * constante.
   */

  constants
  {
    public LENGHT = 100;
  };

  constants
  {
    RANGE_LASER = 100;
    //Error constante duplicada.
    RANGE_LASER = 16;
  };
};
```

**Fragmento de Código 5**: Ejemplos de definiciones de constantes.

## *2.5. Puertos de entrada y puertos de salida.*

Los puertos de entrada y de salida constituyen el mecanismo que permite a los componentes transmitir datos tanto internamente como externamente. A través de ellos se crean las *conexiones de puertos* por donde se transmiten datos agrupados en unidades de información llamadas *paquetes de puertos* que pueden tener distintos tipos de datos asociados.

### 2.5.1. Sintaxis.

En la Figura 8, podemos ver la sintaxis que se emplea para definir puertos tanto de entrada como de salida.

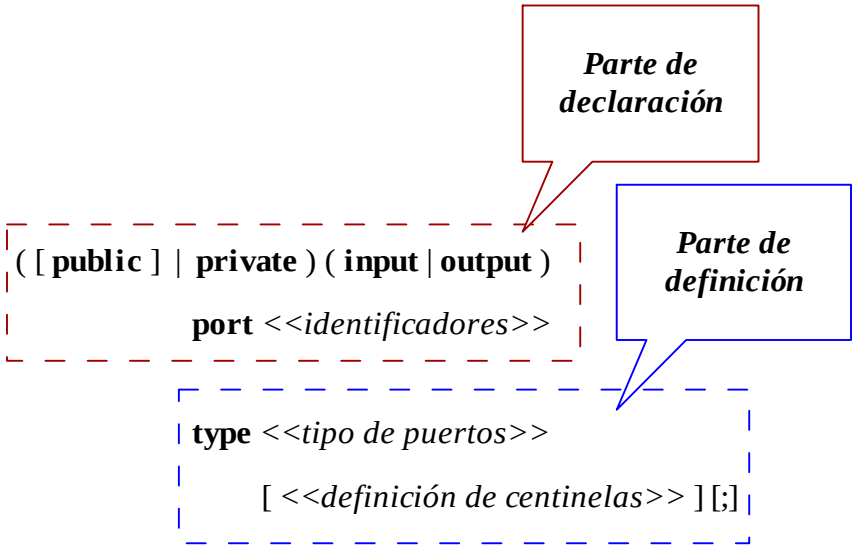


**Figura 8**: Sintaxis de definición de puertos de entrada y de puertos de salida.

La parte de declaración de un puerto comienza especificando si el puerto que se pretende definir es público o privado. Los puertos públicos son accesibles desde fuera del componente de modo que otros componentes pueden usar dichos puertos para establecer conexiones de puertos. En cambio, los puertos privados son solamente accesibles desde el interior del componente donde se define y se utilizan normalmente para intercomunicar los hilos internos de un componente. Para definir un puerto como privado se antepone la palabra reservada *private* y para definir un puerto como público o no se pone nada o se antepone la palabra reservada *public*.

A continuación se especifica si estamos definiendo un puerto de entrada o un puerto de salida. Los puertos de entrada son puertos que sólo pueden recibir datos

procedentes desde uno o varios puertos de salida y los puertos de salida son aquellos que sólo pueden mandar datos a uno o varios puertos de entrada. Para definir un puerto de entrada se utiliza la construcción *input port*, y para definir un puerto de salida se usa *output port.*

La parte de declaración concluye especificando el identificador del puerto que se está definiendo. El identificador se construye siguiendo las reglas del apartado 2.2.3 de este capítulo. Podemos incorporar varios identificadores en una misma declaración, separando cada uno de ellos por el símbolo “,”.

La parte de definición comienza con la palabra reservada *type* seguida de la especificación del tipo de puertos y opcionalmente la definición de un centinela. Ambas definiciones las podremos estudiar en los siguientes apartados.

### 2.5.2. Tipos de puertos.

Todo puerto tiene asociado un tipo de puerto que puede tener cero o más paquetes de puertos asociados. Recordemos que el paquete de puerto es la unidad discreta de información que puede enviarse a través de una conexión de puerto, y es lo que proporciona significado al puerto ya que especifica qué mecanismo y política se empleará para manejar los datos que se envían o se reciben.

En la Tabla 2, podemos ver los tipos de paquetes que podemos definir en DL según la clase de puertos que vayamos a emplear.

| Clase de puerto | Tipo de paquetes asociados |
|---|---|
| Puertos de entrada | *last, fifo, ufifo y priorities.* |
| Puertos tanto de entrada como de salida | *tick, poster, multipacket y pull.* |
| Puertos de salida | *generic, priority y lazymultipacket.* |

**Tabla 2**: Tipos de puertos que podemos definir en el lenguaje DL.

Tal como se puede apreciar en la Tabla 2, sólo los tipos *last*, *fifo*, *ufifo* y *priorities* se pueden definir como puertos de entrada y los tipos *generic*, *priority* y *lazymultipacket* como puertos de salida. El resto (*tick*, *poster*, *multipacket* y *pull*) se pueden definir tanto para puertos de entrada como de salida.

### 2.5.2.1. Paquetes de puertos.

Como hemos dicho, a cada tipo se le pueden asociar cero o más paquetes de puertos. En DL, en función de cómo hagamos referencia a estos paquetes de puertos, podemos distinguir cuatro clases:

- *Paquetes de puertos predefinidos en la plataforma CoolBOT*. Se referencia a través de un identificador, y se trata de una serie de paquetes de puertos definidos y suministrador por la plataforma CoolBOT que podemos ver en la Tabla 3. A este tipo de paquete de puertos, lo llamaremos de aquí en adelante *paquetes de puertos por defecto*.

| PAQUETES DE PUERTOS POR DEFECTO | |
|---|---|
| **Paquete** | **Descripción** |
| PacketUChar | Proporciona valores de caracteres sin signo. |
| PacketInt | Proporciona valores de enteros. |
| PacketLong | Proporciona valores de enteros largo. |
| PacketDouble | Proporciona valores reales. |
| PacketTime | Proporciona valores de tiempo. |
| PacketCoordinates2D | Proporciona una representación de puntos en 2 dimensiones. |
| PacketFrame2D | Proporciona un sistema de referencias en 2 dimensiones. |
| PacketCoordinates3D | Proporciona una representación de puntos en 3 dimensiones. |
| PacketFrame3D | Proporciona un sistema de referencias en 3 dimensiones. |

**Tabla 3**: Paquetes de puertos predefinidos de la plataforma CoolBOT.

Todo identificador que denote un paquete de puertos que coincida con alguno de los mencionados en la Tabla 3, se considera de tipo predefinido en la plataforma CoolBOT.

- *Paquetes de puertos definidos por el usuario*. Si se especifica un identificador que no coincide con ninguno de los anteriores, estamos definiendo un nuevo tipo de paquete de puertos cuya implementación es responsabilidad del programador o usuario. El compilador genera un esqueleto C++ para ellos.

- *Paquetes de puertos definidos en otros componentes*. Se referencia anteponiendo al identificador que especifica el tipo de paquetes de puertos, el nombre del componente seguido del símbolo "*:*" repetido dos veces. Si el nombre del componente coincide con el nombre del componente que estamos definiendo, entonces dicho paquete de puertos corresponderá a uno de los casos anteriormente mencionados.
- *Paquetes de puertos definidos usando un literal cadena*. Otro modo que podemos utilizar para referenciar a un paquete de puerto definido en otro componente es usando un ***literal cadena***. El contenido de ese literal o especifica la implementación o indica donde está definido el paquetes de puertos y el contenido del mismo es responsabilidad del usuario.

El tipo de puerto *tick* es el único que no posee asociado ningún paquete de puertos. Los tipos *last*, *fifo*, *ufifo*, *poster*, *generic*, *priority* y *priorities* admiten una única definición de paquetes de puertos mientras que los tipos *multipacket* y *lazymultipacket* admiten más de una definición de paquetes de puertos. El tipo *pull* admite únicamente dos paquetes de puertos.

Para comprender mejor los paquetes de puertos, en el Fragmento de Código 6, podemos ver un ejemplo donde se define un componente llamado *First* que define una serie de puertos de tipo *fifo* con sus respectivos paquetes de puertos.

```
component First
{
  // Empleo de paquetes de puertos por defecto.
  input port Public_In_A type fifo port packet PacketInt;

  // Empleo de paquetes de puertos definidos por el usuario.
  private input port Private_In_B type fifo port packet MyPacket;

  // Empleo de paquetes de puertos definidos en otros componentes.
  input port Public_In_C type fifo port packet Second::LaserPacket;

  // Empleo de paquetes de puertos definidos usando un literal.
  input port Public_In_D type fifo port packet "Second::LaserPacket";

  entry state Main
  {
    transition on Public_In_A, Private_In_B, Public_In_C;
    transition on Public_In_D;
  }
};
```

**Fragmento de Código 6**: Ejemplos de definiciones de paquetes de puertos.

### 2.5.2.2. Tipo de puerto tick.

El tipo de puerto *tick* es el único que no posee asociado ningún paquete de puertos, y se puede definir tanto con puertos de entrada como con puertos de salida.

Se define empleando la palabra reservada *tick* y la sintaxis completa de definición de un puerto de tipo *tick*, la podemos ver en la Figura 9.

( [ **public** ] | **private** ) ( **input** | **output** )

**port** <<*identificadores*>> **type tick** [;]

**Figura 9**: Sintaxis de definición del puerto *tick*.

Para comprender mejor la sintaxis mostrada en la Figura 9, veamos el siguiente ejemplo mostrado en el Fragmento de Código 7.

```
component First
{
  // Puertos de entrada de tipo tick.
  input port Public_In_A type tick;
  public input port Public_In_B type tick;
  private input port Private_In_C type tick;

  // Puertos de salida de tipo tick.
  output port Public_Out_A type tick;
  public output port Public_Out_B type tick;
  private output port Private_Out_C, Private_Out_D type tick;

  entry state Main
  {
    transition on Public_In_A, Public_In_B, Private_In_C;
  }
};
```

**Fragmento de Código 7**: Ejemplo de definiciones de puertos *tick*.

### 2.5.2.3. Tipo de puerto fifo y ufifo.

El tipo de puerto *fifo* y *ufifo* únicamente se pueden definir como puertos de entrada. Se definen empleando las palabras reservadas *fifo* y *ufifo* y sólo admite un paquete de puertos en su definición. Opcionalmente, permite definir la longitud del paquete de puertos mediante la palabra reservada *length* seguido de un literal numérico o de una constante que indica el valor de longitud. Si no se especifica la palabra reservada *length*, por defecto se toma el valor 1.

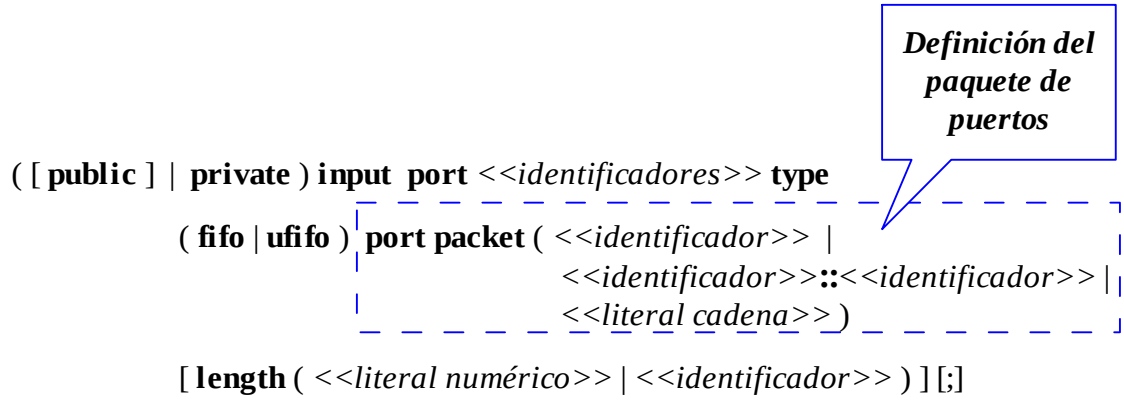


**Figura 10**: Sintaxis de definición de los tipos de puertos fifo y ufifo.

Como podemos apreciar en la Figura 10, el paquete de puertos asociado al puerto se define utilizando primero las palabras reservadas *port* y *packet* y luego especificando el paquete. Este último se define según lo visto en el apartado 2.5.2.1 de este capítulo. Veamos, en el Fragmento de Código 8 y en el Fragmento de Código 9, dos ejemplos de componente donde se definen varios puertos de tipo *fifo* y *ufifo*.

```
component First
{
  constants
  {
    LENGTH = "A::LengthPort";
  };

  input port Public_In_A type fifo port packet PacketInt length 10;
  input port Public_In_B type fifo port packet MPT length LENGTH;
  input port Public_In_C type fifo port packet Second::PacketInt;
  private input port Private_In_D type fifo port packet "MyPacket";

  entry state Main
  {
    transition on Public_In_A, Public_In_B, Private_In_D;
    transition on Public_In_C;
  }
};
```

**Fragmento de Código 8**: Ejemplos de puertos de tipo *fifo*.

```
component First
{
  constants
  {
    LENGTH = "A::LengthPort";
  };

  input port Public_In_A type ufifo port packet PacketInt length 10;
  input port Public_In_B type ufifo port packet MPT length LENGTH;
  input port Public_In_C type ufifo port packet Second::PacketInt;
  private input port Private_In_D type ufifo port packet "MyPacket";

  entry state Main
  {
    transition on Public_In_A, Public_In_B, Private_In_D;
    transition on Public_In_C;
  }
};
```

**Fragmento de Código 9**: Ejemplos de puertos de tipo *ufifo*.

### 2.5.2.4. Tipo de puerto poster.

El tipo de puerto *poster* es un puerto que se puede definir tanto con puertos de entrada como con puertos de salida. Al igual que el tipo de puerto *fifo* y *ufifo* admite un solo paquete de puertos en su definición. En la Figura 11 podemos ver la sintaxis de definición de un tipo de puerto *poster*.

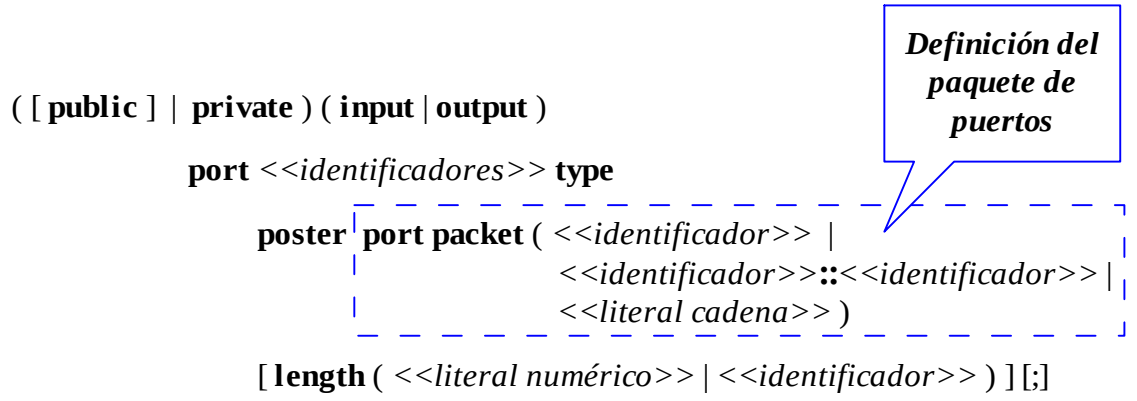


**Figura 11**: Sintaxis de definición del tipo de puerto *poster*.

Como vemos en la Figura 11, un puerto de tipo *poster* se define empleando la palabra reservada *poster* seguida de la definición de paquete de puertos. Opcionalmente, permite definir la longitud del paquete de puertos empleando la palabra reservada *length* seguido de un literal numérico o de una constante que indica el valor de longitud. Si no se especifica la palabra reservada *length*, por defecto se toma el valor 1. En el Fragmento de Código 10, podemos ver un ejemplo de un componente que define varios puertos tanto de entrada como de salida de tipo *poster*.

```
component First
{
  constants
  {
    LENGTH = "A::LengthPort";
  };

  input port A type poster port packet PacketInt length 10;
  public input port B type poster port packet PacketLong;
  private input port C type poster port packet Packet length LENGTH;
  output port D type poster port packet Second::PacketInt;
  private output port E type poster port packet PacketLong;
  public output port F, G type poster port packet PacketLong;

  entry state Main
  {
    transition on A, B, C;
  }
};
```

**Fragmento de Código 10**: Ejemplos de puertos de tipo *poster*.

### 2.5.2.5. Tipo de puerto priority.

El tipo de puerto *priority* es un puerto exclusivo de salida, es decir, únicamente se puede asociar a puertos de salida. Admite un solo paquete de puertos en su definición y en la Figura 12 podemos ver su sintaxis de definición.

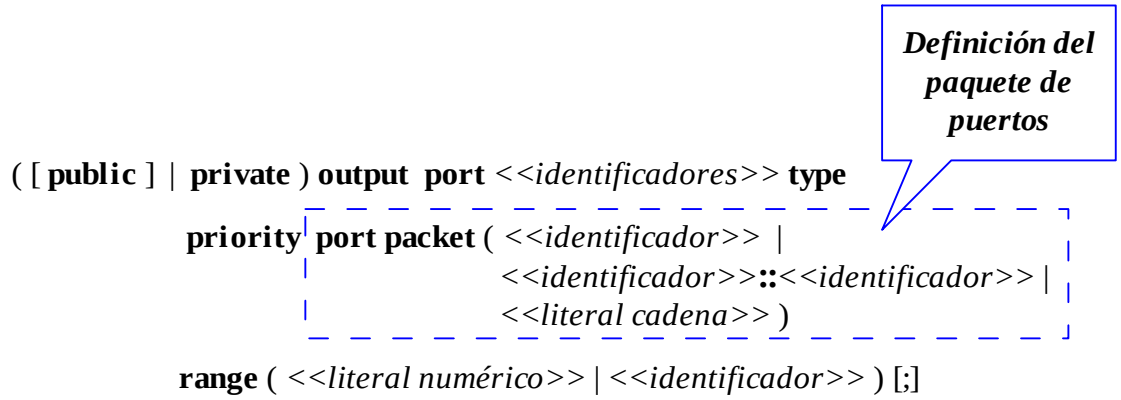


**Figura 12**: Sintaxis del tipo de puerto *priority*.

Como hemos observado en la Figura 12, para definir un tipo de puerto *priority* se emplea la palabra reservada *priority* seguido del paquete de puertos y terminado con la palabra reservada *range*. A continuación, se define un literal numérico o una constante que indica el rango de prioridades sobre las cuales puede trabajar ese puerto. En el Fragmento de Código 11 podemos ver un ejemplo de este tipo de puerto.

```
component First
{
  constants
  {
    RANGE = 10;
  };

  output port A type priority port packet PacketInt range 1;
  public output port B type priority port packet PacketLong;
  private output port C type priority port packet PacketInt
                              range RANGE;

  entry state Main
  {

  }
};
```

**Fragmento de Código 11**: Ejemplo de un componente que define puertos de tipo *priority*.

### 2.5.2.6. Tipo de puerto priorities.

El tipo de puerto *priorities* es un puerto exclusivo de entrada, es decir, únicamente se puede definir con puertos de entrada. Admite un solo paquete de puertos en su definición y en la Figura 13 podemos ver su sintaxis de definición.

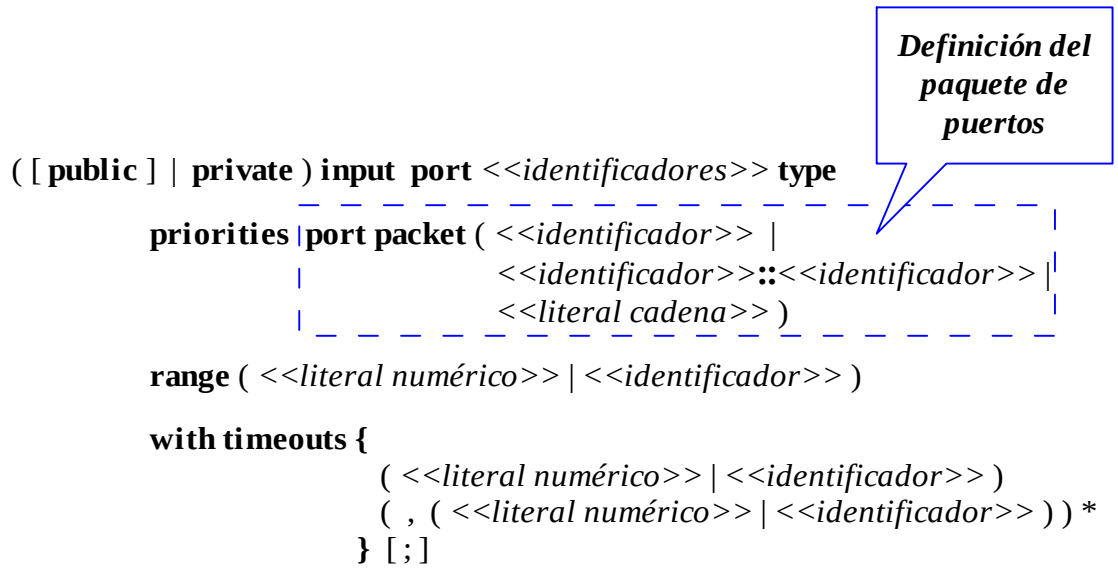


**Figura 13**: Sintaxis de definición del tipo de puerto *priorities*.

Como también vemos en la Figura 13, el tipo de puerto *priorities* se define empleando la palabra reservada *priorities*, seguido del paquete de puertos y del valor que indica el rango de prioridades y terminado con las palabras reservadas *with timeouts* que contiene uno o más valores de tiempo de espera en milisegundos.

Estos valores son definidos entre de un abre y cierra llaves (símbolo "*{*" y símbolo "*}*") y los valores que se pueden definir son literales numéricos y constantes. Si se definen más de un valor se emplea el símbolo "," como separador. El número de valores que se definan deben de concordar con el valor expresado después de la palabra reservada *range*.

En el Fragmento de Código 12, podemos ver un ejemplo de un componente que define varios tipos de puertos *priorities*.

```
component First
{
  constants
  {
    TIME_1 = 1000;
    RANGE = 3;
  };

  input port A type priorities port packet PacketInt
                    range 1 with timeouts { TIME_1 };
  public input port B type priorities port packet PacketLong
                           range 1 with timeouts { 11 };
  /*
   En el puerto C es correcto si el valor de la constante RANGE
   coincide con el número de timeouts expresados.
   */
  private input port C type priorities port packet PacketInt
                             range RANGE
                             timeouts { 84, TIME_1, 100 };

  entry state Main
  {
    transition on A, B, C;
  }
};
```

**Fragmento de Código 12**: Ejemplo de definiciones de puertos de tipo *priorities*.

### 2.5.2.7. Tipo de puerto last.

El tipo de puerto *last* únicamente se puede emplear con puertos de entrada. Se definen empleando las palabras reservadas *last* y sólo admite un paquete de puertos en su definición. En la Figura 14 podemos ver la sintaxis de definición de este tipo de puerto.

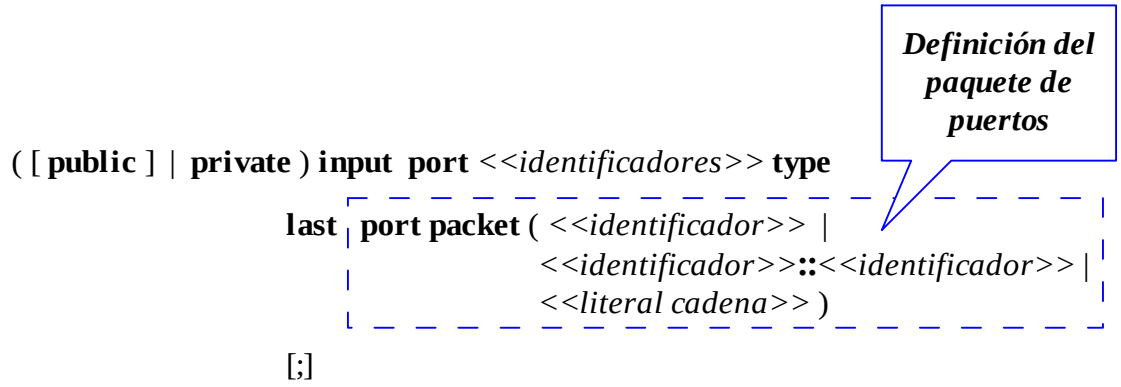


**Figura 14**: Sintaxis de definición del tipo de puerto *last*.

En el Fragmento de Código 13, podemos ver un ejemplo de un componente que define varios puertos de tipo *last*.

```
component First
{
  input port A type last port packet PacketInt;
  public input port B type last port packet PacketLong;
  private input port C type last port packet PacketInt;

  entry state Main
  {
    transition on A, B, C;
  }
};
```

**Fragmento de Código 13**: Ejemplos de definiciones de puertos de tipo *last*.

### 2.5.2.8. Tipo de puerto generic.

El tipo de puerto *generic* únicamente se puede asociar con puertos de salida. Se definen empleando las palabras reservadas *generic* y sólo admite un paquete de puertos en su definición. En la Figura 15 podemos ver la sintaxis de definición de este tipo de puerto.

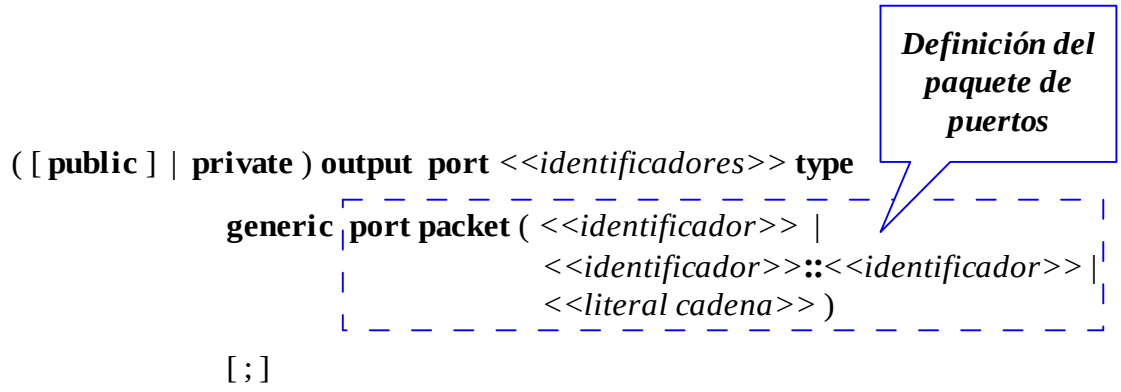


**Figura 15**: Sintaxis de definición del tipo de puerto *generic*.

En el Fragmento de Código 14, podemos ver un ejemplo de un componente que define varios puertos de tipo *generic*.

```
component First
{
  output port A type generic port packet PacketInt;
  public output port B type generic port packet PacketLong;
  private output port C type generic port packet PacketInt;

  entry state Main
  {

  }
};
```

**Fragmento de Código 14**: Ejemplo de definición de varios puertos de tipo *generic*.

### 2.5.2.9. Tipo de puerto multipacket.

El tipo de puerto *multipacket* se puede utilizar para definir tanto puertos de entrada como puertos de salida. Este tipo de puerto admite uno o más paquetes de puertos como podemos ver en la Figura 16 donde se muestra su sintaxis de definición.

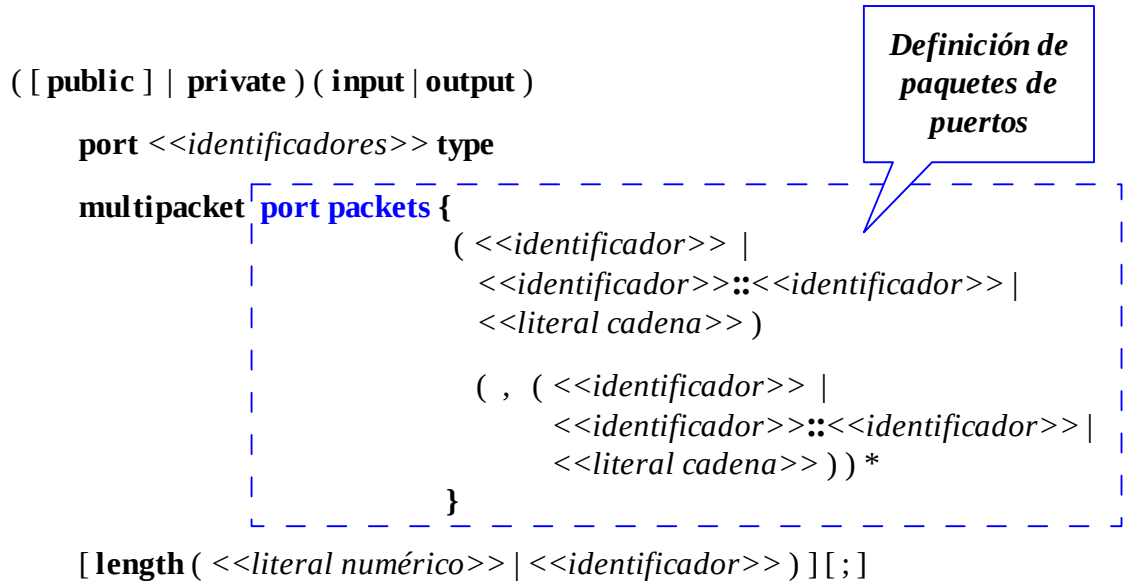


**Figura 16**: Sintaxis de definición del tipo de puerto multipacket.

Como vemos en la Figura 16, para definir el tipo de puerto *multipacket* se emplea la palabra reservada *multipacket*. Después se definen los paquetes de puertos asociados este tipo de puerto. Como vemos, la definición de los paquetes varía un poco con respecto a lo visto en los tipos de puertos anteriores. Ahora, la definición de los paquetes de puertos se encuentra encerrada entre llaves (símbolo "*{*" y símbolo "*}*") donde podemos definir uno o más paquetes de puertos y se emplea las palabras reservadas *port packets* para iniciar la definición.

Opcionalmente, se permite indicar el número de tipos de paquetes de puertos que pueden transmitir el tipo de puerto. Para ello, se emplea la palabra reservada *length* seguido de un literal numérico o de una constante que indica el número de tipos de paquetes de puertos. Si se omite, toma como valor el número de paquetes de puertos definidos.

En el Fragmento de Código 15, podemos ver un ejemplo de un componente donde se definen varios puertos de tipo *multipacket*.

```
component First
{
  constants
  {
    LENGTH = 2;
  };

  input port A type multipacket port packets { PacketInt };
  public input port B type multipacket port packet {
                                                     PacketLong,
                                                     PacketInt
                                                   }
                              length LENGTH;
  private input port C type multipacket port packet {
                                                      PacketInt,
                                                      PacketInt,
                                                      PacketInt
                                                    }
                              length 3;

  output port D type multipacket port packets { PacketInt };
  public output port E type multipacket port packet {
                                                      PacketLong,
                                                      PacketInt
                                                    }
                              length LENGTH;
  private output port F type multipacket port packet {
                                                       PacketInt,
                                                       PacketInt,
                                                       PacketInt
                                                     }
                              length 3;

  entry state Main
  {
    transition on A, B, C;
  }
};
```

**Fragmento de Código 15**: Ejemplos de definiciones de puertos de tipo *multipacket*.

### 2.5.2.10. Tipo de puerto lazymultipacket.

Este tipo de puerto es muy parecido al tipo de puerto *multipacket* con la excepción de que sólo se puede emplear con puertos de salida. Su sintaxis de definición la podemos apreciar en la Figura 17.

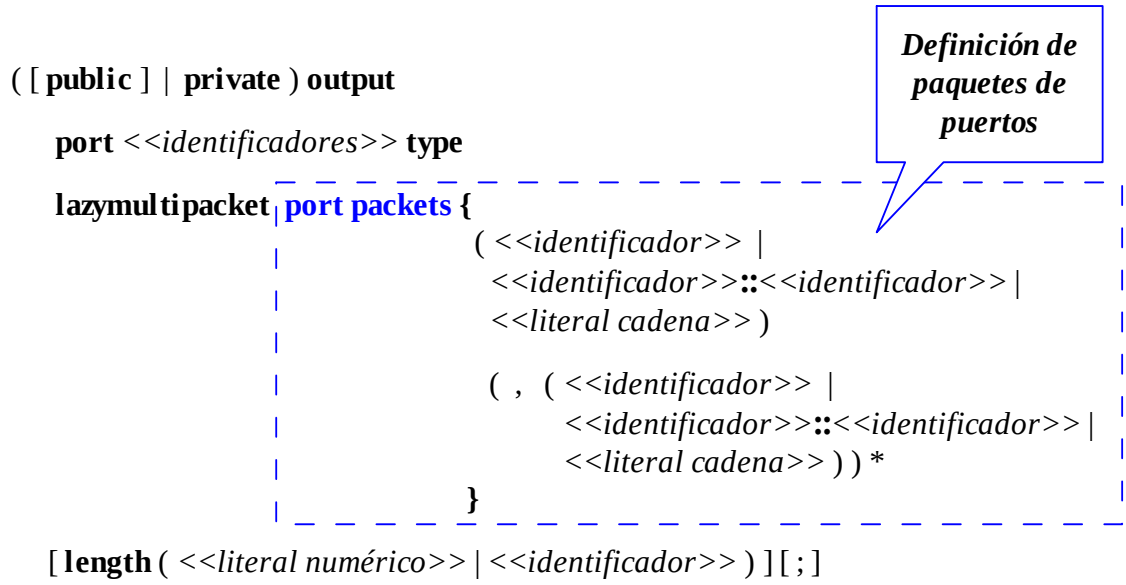


**Figura 17**: Sintaxis de definición del tipo de puerto *multipacket*.

Como vemos en la Figura 17, para definir el tipo de puerto *lazymultipacket* se emplea la palabra reservada *lazymultipacket*. Después se definen los paquetes de puertos asociados este tipo de puerto. Entre llaves (símbolo "*{*" y símbolo "*}*") se puede definir uno o más paquetes de puertos, empleándose las palabras reservadas *port packets* para iniciar la definición. Opcionalmente, se permite indicar el número de tipos de paquetes de puertos que pueden transmitir el tipo de puerto. Para ello, se emplea la palabra reservada *length* seguido de un literal numérico o de una constante que indica el número de tipos de paquetes de puertos. Si se omite, toma como valor el número de paquetes de puertos definidos.

En el Fragmento de Código 16, podemos ver un ejemplo de un componente donde se definen varios puertos de tipo *lazymultipacket*.

```
component First
{
  constants
  {
    LENGTH = 2;
  };

  output port D type lazymultipacket port packets { PacketInt };
  public output port E type lazymultipacket port packet {
                                                          PacketLong,
                                                          PacketInt
                                                         }
                              length LENGTH;
  private output port F type lazymultipacket port packet {
                                                           PacketInt,
                                                           PacketInt,
                                                           PacketInt
                                                          }
                               length 3;

  entry state Main
  {

  }
};
```

**Fragmento de Código 16**: Ejemplo de definiciones de puertos de tipo *lazymultipacket*.

### 2.5.2.11. Tipo de puerto pull.

El tipo de puerto *pull* es un tipo especial porque cuando se emplea estamos definiendo simultáneamente un puerto de entrada y uno de salida. En verdad, lo que se define es un puerto para petición y un puerto para respuesta que serán de entrada o de salida en función de cómo se defina el puerto de tipo *pull*. Si se define de entrada, tendremos un puerto de respuesta como entrada y un puerto de petición como salida y si se define como salida, tendremos un puerto de petición como entrada y un puerto de respuesta como salida. En la Figura 18 podemos apreciar la sintaxis con la que se define este tipo de puerto, usando la palabra reservada *pull* seguida de la definición de paquetes de puertos.

Los paquetes de puertos se definen entre llaves (símbolo "*{*" y símbolo "*}*") y siempre serán dos, uno para petición y otro para respuesta. Al paquete de puerto asociado a la petición se le antepone la palabra reservada *request* y al paquete de puerto asociado a la respuesta se le antepone la palabra reservada *answer*.

Opcionalmente, se permite indicar el número de tipos de paquetes de puertos que pueden transmitir este tipo de puerto que, como ya hemos visto, siempre será 2 que es el valor que toma por defecto en caso de que se omita.

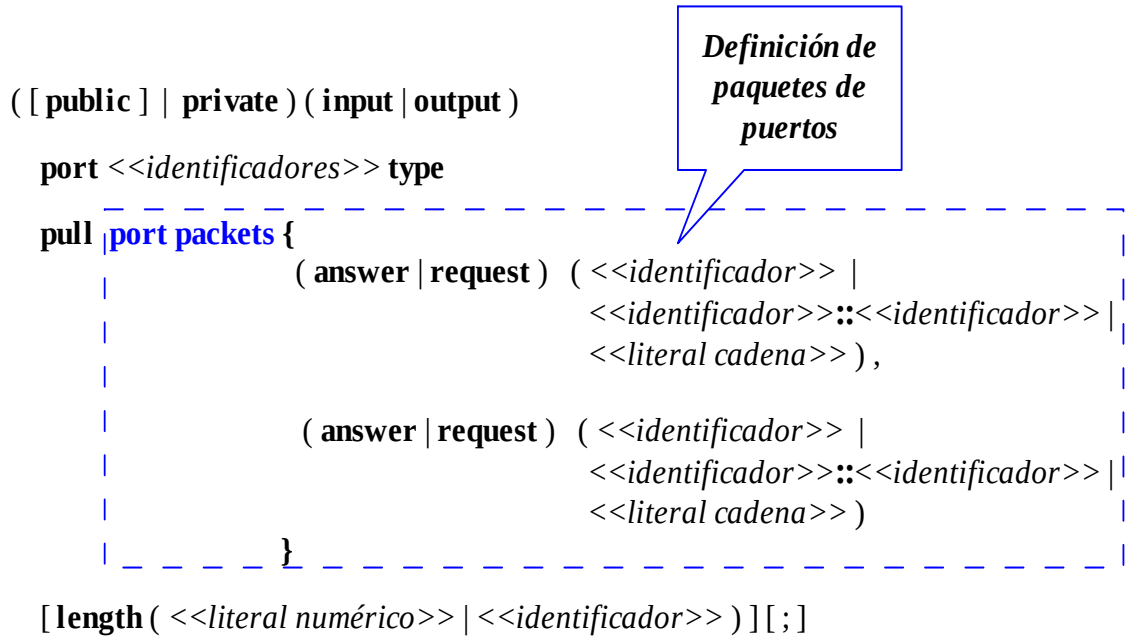


**Figura 18**: Sintaxis de definición del tipo de puerto *pull*.

En el Fragmento de Código 17 podemos ver un ejemplo donde se definen varios puertos de tipo *pull*.

```
component First
{
  constants
  {
    LENGTH = 2;
  };

  input port A type pull port packets {
                                        answer PacketInt,
                                        request PacketInt
                                      };
  public input port B type pull port packet {
                                              request PacketLong,
                                              answer PacketInt
                                            }
                              length LENGTH;
  private input port C type pull port packet {
                                               answer PacketInt,
                                               request PacketInt
                                             }
                               length 2;

  output port D type pull port packets {
                                         answer PacketInt,
                                         request PacketInt
                                       };
  public output port E type pull port packet {
                                               request PacketLong,
                                               answer PacketInt
                                             }
                               length LENGTH;
  private output port F type pull port packet {
                                                request PacketLong,
                                                answer PacketInt
                                              }
                                length 2;

  entry state Main
  {
    transition on A, B, C, D, E, F;
  }
};
```

**Fragmento de Código 17**: Ejemplo de definiciones de puertos de tipo *pull*.

### 2.5.3. Definición de centinelas.

Opcionalmente podemos asociar a un puerto de entrada un centinela que monitorice dicho puerto para que controle situaciones anormales en las cuales no se reciben paquetes de puertos. Los centinelas son una opción exclusiva de los puertos de entrada. Únicamente se puede definir un centinela a un puerto de salida cuando su tipo sea *pull* en cuyo caso se asocia al puerto de petición que se define como entrada. El formato que se emplea para definir un centinela lo podemos ver en la Figura 19.

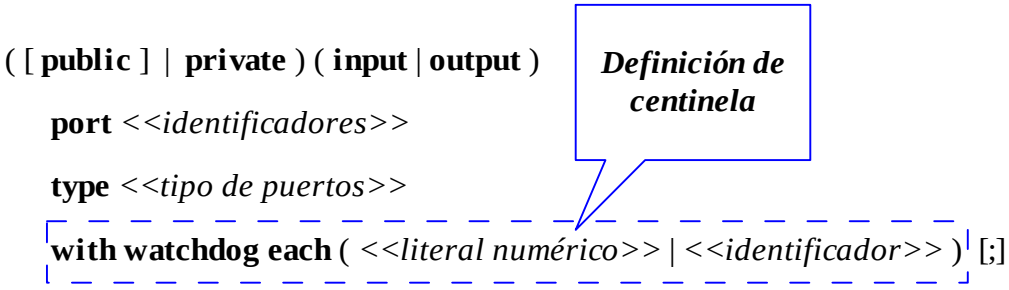


**Figura 19**: Sintaxis para la definición de centinelas.

Como vemos en la Figura 19, un centinela se especifica al final de la definición de un puerto, empleando las palabras reservadas *with watchdog each*. A continuación de estas palabras reservadas se indica el periodo de actuación en milisegundos del centinela que se expresa con literal numérico o con una constante.

La definición de un centinela implica la definición implícita de una excepción denominada "*badWatchDogTransition*". Esta excepción define tres manejadores: un manejador de éxito, un manejador de fracaso y un manejador de recuperación que realiza un máximo de intentos periódicamente como veremos en el apartado 2.7 de este capítulo.

En el Fragmento de Código 18, podemos ver un ejemplo donde a un puerto se le define un centinela.

```
component First
{
  constants
  {
    FRECUENCE = 200;
  };

  input port Public_In_A type tick with watchdog each 100;
  input port Public_In_B type tick with watchdog each FRECUENCE;

  entry state Main
  {
    transition on Public_In_A, Public_In_B;
  }
};
```

**Fragmento de Código 18**: Ejemplo de un puerto que define un centinela.

## *2.6. Variables observables y variables controlables.*

En CoolBOT, para controlar y monitorizar las operaciones que realiza un componente tenemos dos clases de variables: las variables observables y las variables controlables [Domínguez-Brito, 2003]. Ambas variables permiten desde el exterior acceder a los aspectos internos de un componente. Las variables observables se usan para realizar tareas de monitorización y las variables controlables para realizar tareas de control en base a la información aportada por las variables observables.

### 2.6.1. Sintaxis.

En la Figura 20, podemos ver la sintaxis de definición de las variables observables y de las variables controlables.

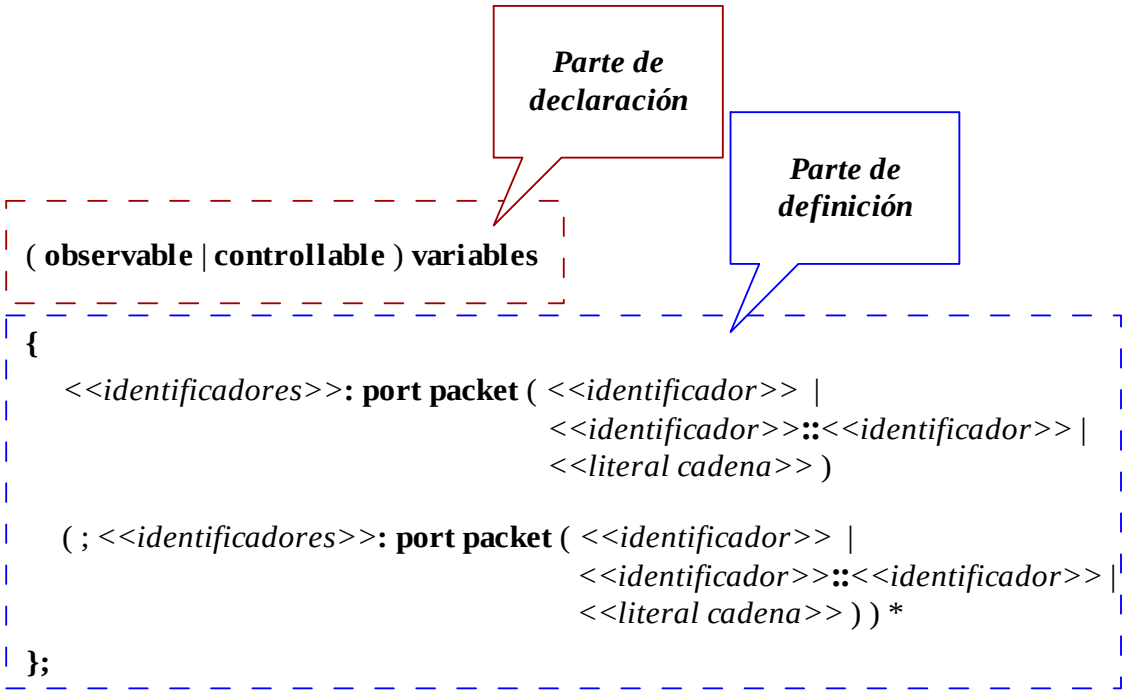


**Figura 20**: Sintaxis de definición de las variables observables y de las variables controlables.

Como vemos en la Figura 20, la parte de declaración comienza con la definición del tipo de variable o variables que se van a definir. Si la variable es observable se utiliza la palabra reservada *observable* y si es controlable se emplea la palabra reservada *controllable*. La parte de declaración termina con la palabra reservada *variables*.

La parte de definición se encuentra encerrada entre llaves (símbolo "*{*" y símbolo "*}*") y debemos de definir como mínimo una variable. Estas se definen

especificando uno o más identificadores (si se especifica más de uno se emplea el símbolo "," como separador de identificadores) seguido del símbolo ":" y la definición de un paquete de puertos (ver el anterior apartado 2.5.2.1).

Cada variable se identifica a través de un nombre de identificador único y no pueden existir dos variables con el mismo identificador aunque sean de tipos diferentes.

### 2.6.2. Ejemplo.

En el Fragmento de Código 19, podemos ver un ejemplo con un componente que define variables observables y variables controlables.

```
component First
{
  observable variables
  {
    Obs_Var_A, Obs_Var_B: port packet PacketInt;
    Obs_Var_C: port packet SimpleComponent::PacketLaser;
  };

  controllable variables
  {
    Cont_Var_A, Cont _Var_B: port packet MyPacketInt;
    Cont_Var_C: port packet PacketLong;
  };

  entry state Main
  {
    transition on Cont_Var_A;
  }
};
```

**Fragmento de Código 19**: Ejemplo de definición de variables controlables y observables.

## *2.7. Excepciones.*

En todo componente pueden darse situaciones excepcionales, anómalas o erróneas que pueden llevar al componente a un mal funcionamiento, por lo que resulta interesante disponer de algún mecanismo que permita actuar ante estas situaciones. CoolBOT emplea como mecanismo, para resolver estas situaciones, las excepciones.

### 2.7.1. Sintaxis.

Las excepciones en DL se definen tal y como se muestra en la siguiente Figura 21.

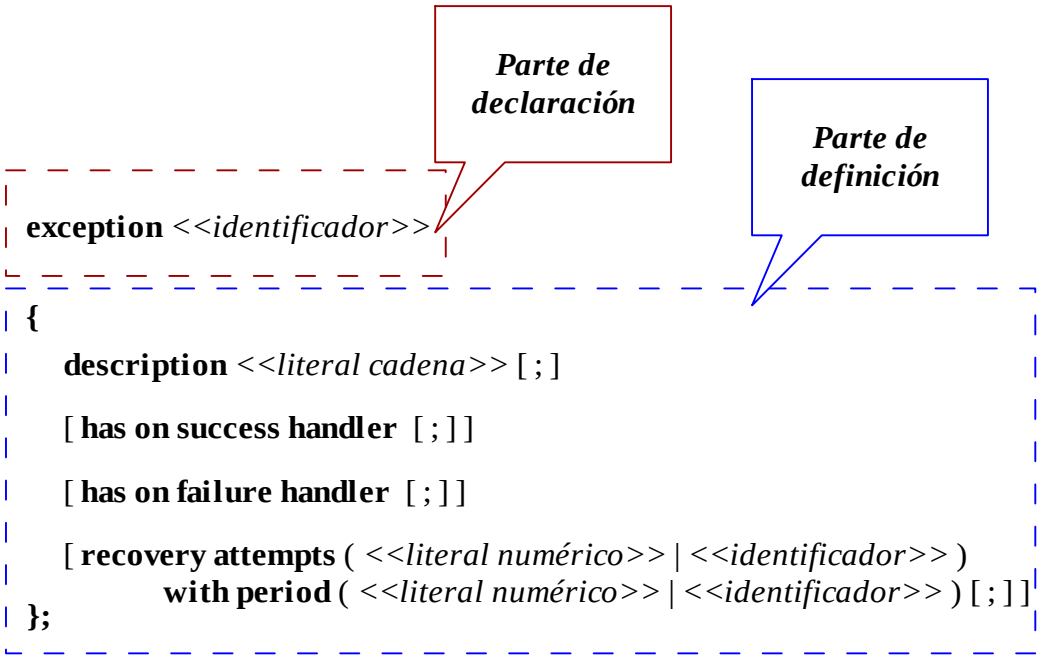


**Figura 21**: Sintaxis de definición de excepciones.

Toda excepción que queramos especificar comienza con la palabra reservada *exception* seguida de un nombre de identificador único y encerrado entre llaves la definición de la excepción. En DL no se puede declarar una excepción con el identificador "*badWatchDogTransition*" debido a que es una excepción especial de la plataforma CoolBOT que se emplea en el caso de que se definan centinelas en la definición un puerto de entrada.

Encerrado entre llaves se definen dos atributos que dan significado a la excepción. El primer atributo proporciona una descripción de la excepción, y es obligatorio. Para ello se utiliza la palabra reservada *description* seguida de un literal

cadena que contiene la descripción. El segundo atributo es opcional y a su vez se divide en tres atributos que se emplean para especificar manejadores de excepciones.

### 2.7.2. Tipos de manejadores.

Un manejador de excepción es un conjunto de instrucciones destinadas a ejecutarse en respuesta a la ocurrencia de una determinada excepción en el sistema. En DL se distinguen tres tipos de manejadores de excepciones:

- *Manejador de recuperación*. La misión de este manejador es hacer que el componente se intente recuperar de la excepción que se ha producido. Para ello realiza cada cierto tiempo un intento de recuperación. Trascurridos un determinado número de intentos, si los intentos de recuperación de la excepción han fracasado se ejecuta el manejador de fracaso si está definido.

  Para definir un manejador de recuperación se especifica la palabra reservada *recovery* seguido del número de intentos de recuperación especificado con la palabra reservada *attempts* y un valor numérico o una constante. Opcionalmente podemos especificar un periodo en milisegundos con la que se realizarán estos intentos. Para ello utilizamos la siguiente construcción formada por dos palabras reservadas: *with period* seguido de un valor numérico o una constante para indicar este periodo.
- *Manejador de éxito*. Se trata del manejador que se ejecuta en caso de que el componente se recupere de la excepción producida. En DL se expresa con la siguiente construcción formada por 4 palabras reservadas. *has on success handler*.
- *Manejador de fracaso*. Se trata del manejador que se ejecuta en caso de que el componente no se recupere de la excepción producida. En DL se expresa con la siguiente construcción formada por 4 palabras reservadas. *has on failure handler*.

En una excepción, los atributos antes mencionados sólo se pueden definir una vez. Si se define un atributo más de una vez, sólo la primera definición se tiene en cuenta, el resto se omite.

### 2.7.3. Ejemplo.

En el Fragmento de Código 20 podemos ver un ejemplo de definición de una excepción llamada *MiExcepcion* que define los tres manejadores de excepción vistos en este apartado.

```
component First
{
  constants
  {
    PERIOD = 200;
    ATTEMPTS = 10;
  };

  exception MiExcepcion
  {
    description "Mi excepción";
    has on success handler;
    has on failure handler;
    recovery attempts ATTEMPTS with period PERIOD;
  };

  entry state Main
  {

  }
};
```

**Fragmento de Código 20**: Ejemplo donde se define un excepción.

## *2.8. Estados.*

La funcionalidad interna de un componente se define como un autómata de estados y todo autómata está formado por uno o más estados donde las transiciones entre estados son determinadas por paquetes de puertos recibidos a través de un puerto de entrada o por cualquier condición interna. Uno de los estados del autómata denominado de entrada se encarga de iniciar el autómata de estados,

En base a lo comentado, todo componente obligatoriamente debe definir un autómata de estados y ese autómata tiene que estar formado por uno o más estados siendo uno de ellos el de entrada. En CoolBOT, el autómata de estados que determina la funcionalidad de un componente se denomina Autómata de Usuario, que se encuentra insertado en otro autómata llamado Autómata por Defecto [Domínguez-Brito, 2003]. En la Figura 22, podemos ver un diagrama donde se muestra el Autómata por Defecto de la plataforma CoolBOT. Dentro del estado *running* se encuentra el Autómata de Usuario que el programador define.

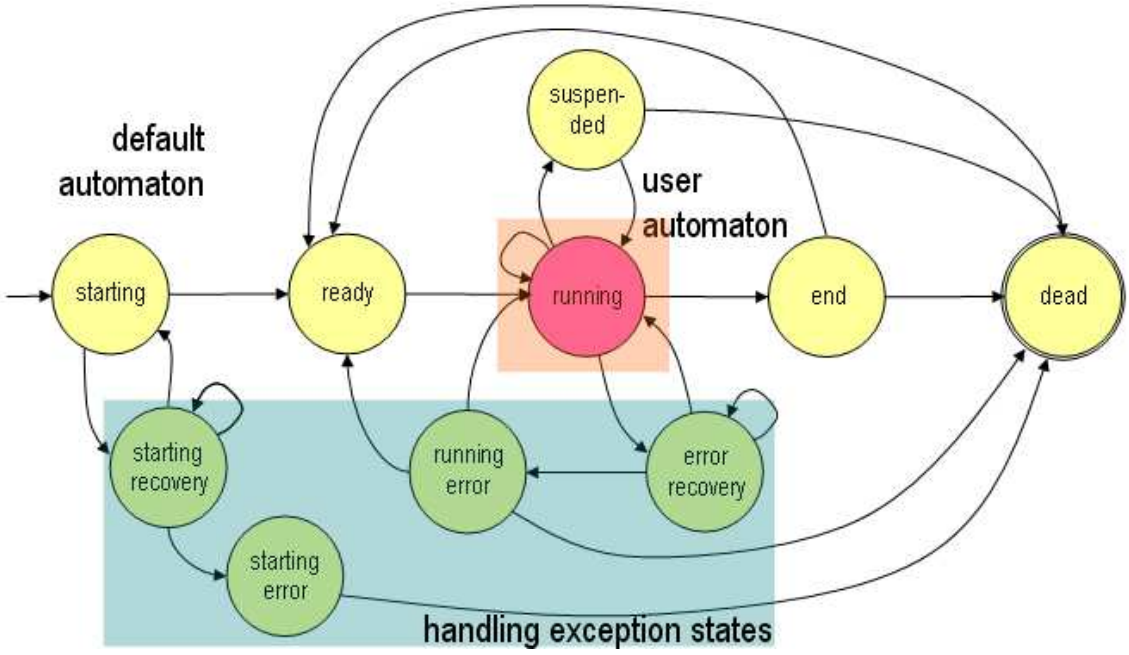


**Figura 22**: Autómata por Defecto de la plataforma CoolBOT.

### 2.8.1. Sintaxis.

En DL, un estado se define como se muestra en la Figura 23. La parte de declaración comienza especificando si se está definiendo un estado de entrada o no. Un estado de entrada es el encargado de iniciar el proceso de ejecución de un autómata. A partir de las condiciones que se dan en ese estado se transita a otros estados. Solamente puede haber un estado de entrada y se define empleando la palabra reservada *entry* antes de la palabra reservada *state*. Si no se especifica, nada el estado no es de entrada.

En la parte de definición, encerrado entre llaves se definen los puertos de entrada y variables controlables con los cuales ese estado va a realizar transiciones. Para ello se emplean las palabras reservadas *transition on* seguida de uno o más identificadores (en este caso, separados por el símbolo ",") que definen puertos de entrada o variables controlables. En el siguiente apartado podemos conocer un poco más sobre las transiciones de estados.

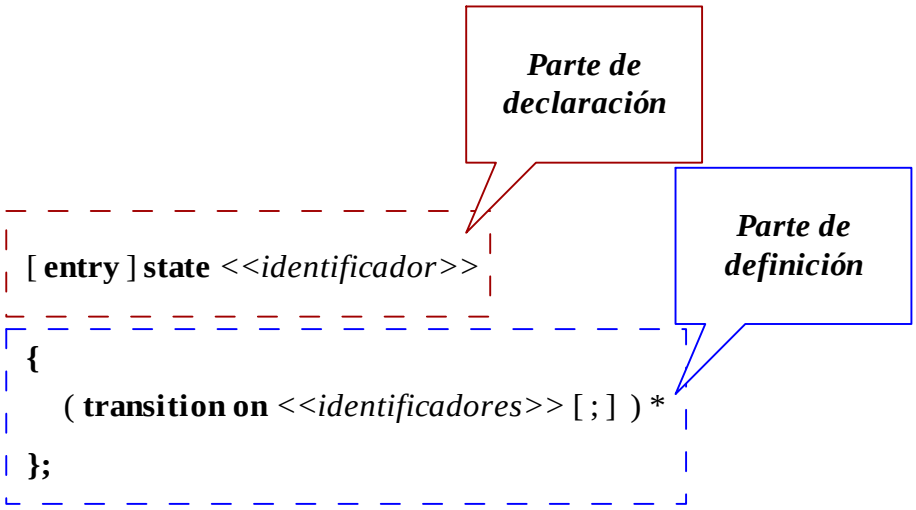


**Figura 23**:_Sintaxis de definición del lenguaje DL.

### 2.8.2. Transiciones de estados.

Como hemos dicho anteriormente, un estado puede realizar transiciones a otros estados sobre puertos de entrada o variables controlables. Todo puerto de entrada y/o variable controlable que se especifique tiene que estar definido previamente, y en un estado no puede haber más de una transición sobre ese puerto de entrada o esa variable controlable.

La incorporación de transiciones es opcional por lo que pueden definirse estados que no realicen transición alguna. También se pueden definir transiciones sobre dos puertos privados predefinidos en la plataforma CoolBOT: "***empty_transition***" y "***timer***" y que podemos utilizar respectivamente para realizar transiciones vacías en el componente o provocadas por un temporizador interno (*timer*) a cada componente.

No se permite realizar transiciones sobre puertos de salida, exceptuando si el puerto se define utilizando el tipo *Pull* ya que este conceptualmente se traduce en dos puertos; uno de entrada y otro de salida. El puerto de entrada sería sobre el cual se haría la transición. Tampoco se pueden utilizar variables observables para realizar transiciones en un estado, solamente se pueden utilizar variables controlables.

### 2.8.3. Ejemplo.

En el Fragmento de Código 21 podemos ver dos ejemplos de definiciones de estados en donde se realizan transiciones sobre los puertos *empty_transition* y *timer*

```
component First
{

  controllable variables
  {
    New_Period: port packet PacketInt;
  };

  input port Period type last port packet PacketInt;
  public output port B type pull port packet {
                                                request PacketLong,
                                                answer PacketInt
                                              };
  entry state Main
  {
    transition on New_Period;
  };

  state FirstState
  {
    transition on Period;
  };

  state SecondState
  {
    transition on B;
    transition on empty_transition, timer;
  };

};
```

**Fragmento de Código 21**: Ejemplo de definición de estados en lenguaje DL.

## *2.9. Hilos de puertos.*

Por si es preciso garantizar los tiempos de respuesta en los componentes se ha introducido en CoolBOT el concepto de hilos de puertos (port threads) que permiten organizar la ejecución interna de un componente utilizando diferentes hilos que son responsables de diferentes conjuntos de puertos de entrada, tal y como se muestra en la Figura 24.

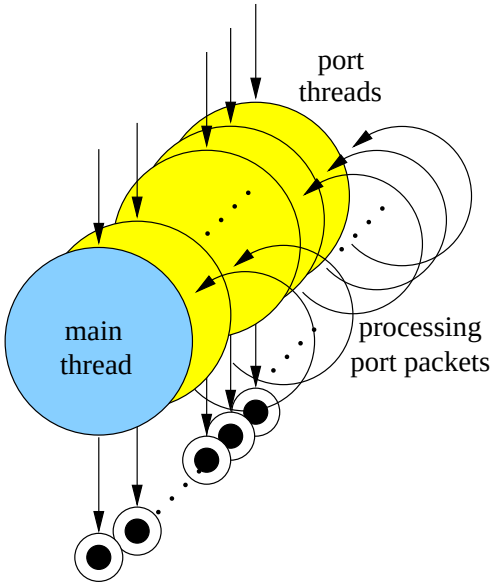


**Figura 24**: Múltiples hilos ejecutándose.

### 2.9.1. Sintaxis.

En DL, los hilos se definen empleando la sintaxis que se muestra en la Figura 25. La parte de declaración empieza especificando si el hilo tiene asociado la política de espera activa (*polling*), seguido de la palabra reservada *thread* y finalizado con un identificador unívoco.

Se denomina espera activa a una técnica donde un proceso repetidamente verifica una condición. Si deseamos que un hilo siga esta política antes de la palabra reservada *thread* se especifica la palabra reservada *polling*. Si no se especifica nada, ese hilo no posee la política de espera activa y realizará un espera pasiva o bloqueante sobre sus puertos de entrada.

En cuanto al identificador que se puede emplear, en todo componente CoolBOT, por defecto, hay un hilo principal que se referencia utilizando el nombre de identificador

*main* (recordemos que los identificadores no son sensibles a mayúsculas y minúsculas por lo que "*Main*", "*MAIN*",… denotan el mismo identificador *main*). Este hilo es responsable de iniciar todos los hilos de puertos del componente así como de monitorizarlos y controlarlos. En DL, podemos declarar un hilo empleando el identificador *main* con la intención de especificar un nuevo comportamiento al hilo *main* de la plataforma CoolBOT. En el caso de que se defina, sólo se puede declarar una vez en el componente y no admite la política de espera activa.

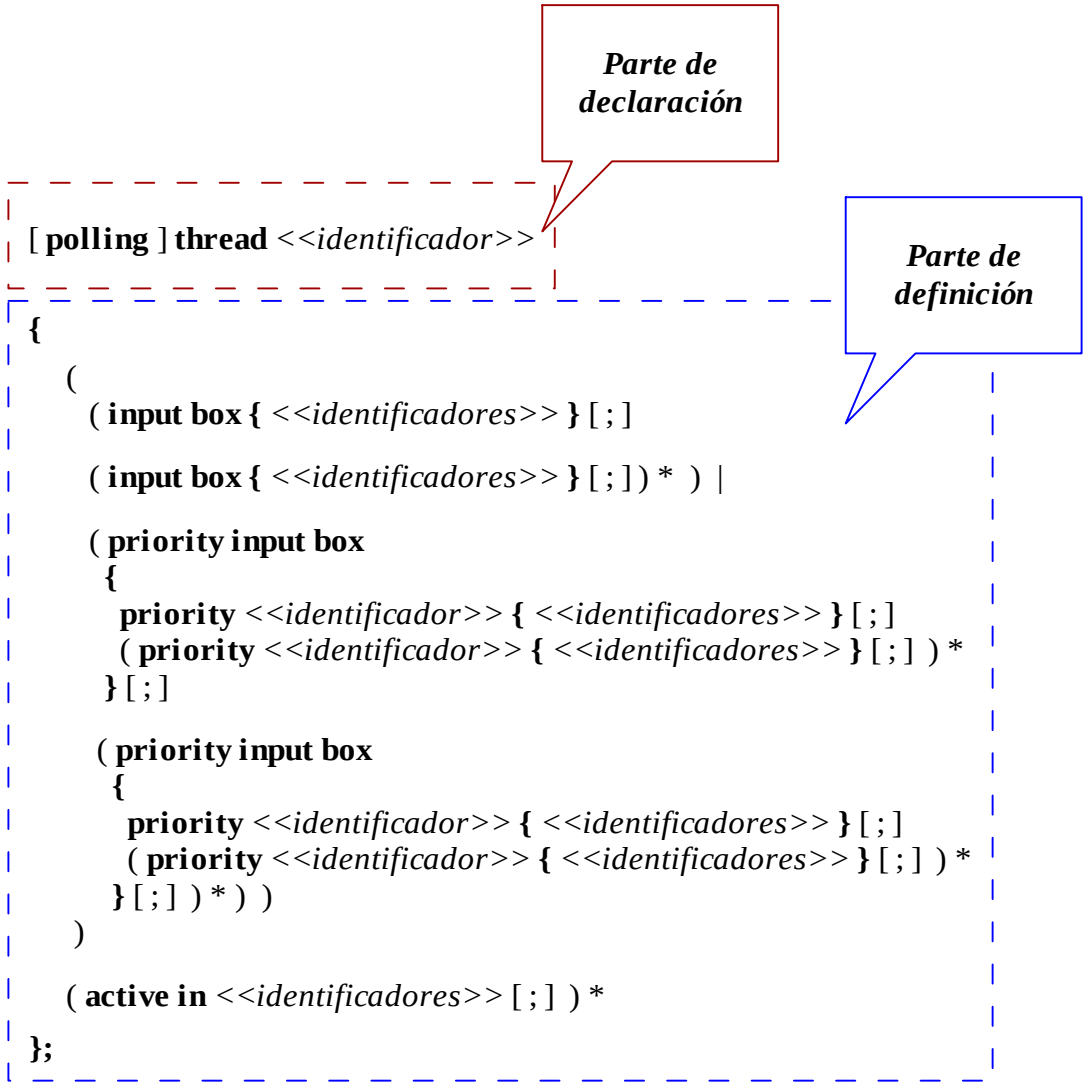


**Figura 25**: Sintaxis de definición de hilos.

En la parte de definición declaramos los puertos de entrada sobre los que actúa el hilo y, opcionalmente, podemos definir los estados donde ese hilo está activo. En función de cómo se definan los puertos de entrada en el hilo, podemos distinguir dos clases de hilos: los hilos con prioridades entre sus puertos de entrada (*priority input box*) y los hilos sin prioridades (*input box*) entre los mismos.

### 2.9.2. Puerto de entrada.

Todo hilo tiene que tener asociado como mínimo a un puerto de entrada sobre el cual actuar. En función de si a los puertos de entrada se le especifican prioridades o no podemos distinguir dos tipos de hilos diferentes. Si se definen prioridades, se dice que el hilo es de clase *priority input box* y si no se definen, se dice que el hilo es de clase *input box*. Un hilo sólo puede ser de una clase.

Los hilos de clase *priority input box* se definen siguiendo la sintaxis que se muestra en la Figura 26.

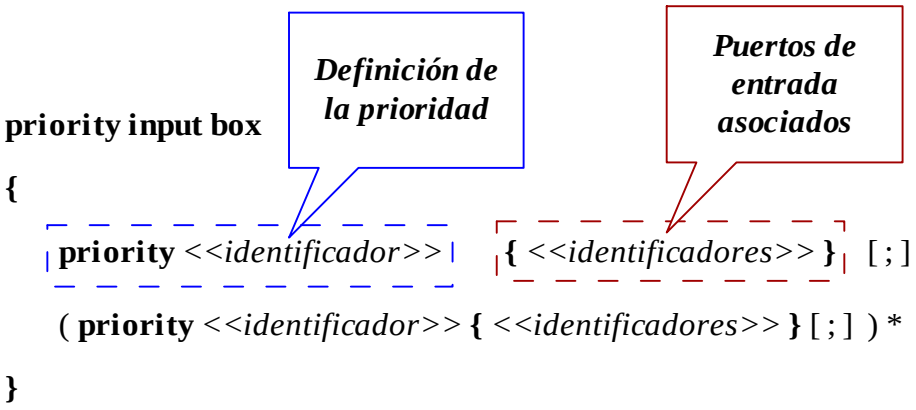


**Figura 26**: Sintaxis de definición de los hilos *priority input box*.

Comos vemos en la figura anterior, primero se inicia la declaración con las palabras reservadas *priority input box* y luego entre llaves se definen las prioridades y los puertos de entrada asociados a dicha prioridad. La definición de prioridad comienza con la palabra reservada *priority* seguida de un identificador que debe ser único en el componente, es decir, no se puede definir el mismo identificador de prioridad dos veces en el mismo hilo o utilizar el mismo identificador de prioridad en otro hilo.

A una prioridad podemos asociarle uno o más puertos de entrada. Estos se definen encerrados entre llaves (símbolo "*{*" y "*}*") y si especificamos más de un puerto se utiliza el símbolo "," como separador. Una vez asociado una prioridad a un puerto de entrada, no podemos volver a utilizar ese puerto en la definición de otra prioridad ni en la definición de otro hilo. Si definimos más de una prioridad hemos de tener en cuenta su orden de definición. Las prioridades se definen de mayor a menor prioridad, por tanto la primera prioridad que definamos será la que tenga mayor prioridad en ese hilo.

Con los hilos de clase *input box* ocurre lo mismo que en los hilos de clase *priority input box*, al hilo podemos asociarle uno o más puertos de entrada empleando el

símbolo "," como separador y una vez asociado no podemos volver asociarlo en ningún otro hilo o en el mismo. En la Figura 27, podemos ver la sintaxis de definición de esta clase de hilo.

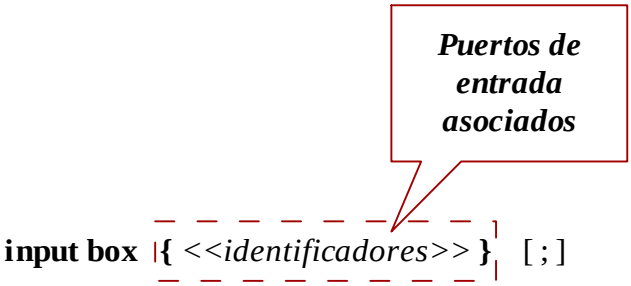


**Figura 27**: Sintaxis de definición de los hilos de clase *input box*.

Por último, es obligatorio que todos los puertos de entrada que se definan en el componente, tengan que estar asignados a algún hilo. Si no se especifica ningún hilo se asocian automática e implícitamente al hilo *main.*

### 2.9.3. Estados activos.

Opcionalmente podemos especificar sobre qué estados del autómata de usuario está activo cada hilo. Los estados especificados deben estar definidos en el componente. Sólo el hilo *main* puede estar activo en los estados del Autómata por Defecto de la plataforma CoolBOT.

La forma de declarar los estados activos en DL es a través de la construcción formada por dos palabras reservadas *active in* seguido de uno o más estados como podemos ver en la Figura 28. Si especificamos más de un estado se emplea el símbolo "," como separador.

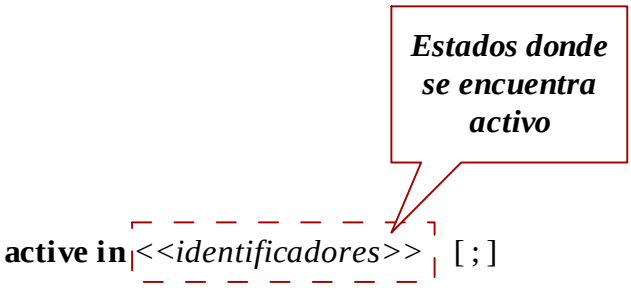


**Figura 28**: Sintaxis de definición de los estados donde el hilo esta activo.

### 2.9.4. Ejemplo.

En el Fragmento de Código 22 podemos un ejemplo de un componente que define el hilo *main* y otro hilo denominado *First_Thread*. En el hilo *main* se han definido dos prioridades *basic* y *high* y el hilo *First_Thread* tiene asociada la política de espera activa.

```
component First
{

  controllable variables
  {
    New_Period: port packet PacketInt;
  };

  input port MyTimer type tick;
  input port Period type last port packet PacketInt;
  public output port B type pull port packet {
                                                request PacketLong,
                                                answer PacketInt
                                              };
  entry state MainState
  {
    transition on New_Period;
  };

  thread main
  {
    Priority input box
    {
      priority high { Period };
      priority basic { B };
    };
  };

  polling thread First_Thread
  {
    input box { MyTimer };
    active in MainState;
  };

};
```

**Fragmento de Código 22**: Ejemplo de definición de hilos en lenguaje DL.

## *2.10. Un ejemplo de una definición de componente completo.*

A continuación, vamos a definir en *Description Language* un componente que actualmente forma parte de un sistema de navegación segura implementado por el grupo de investigación GIAS[2] (Grupo de Inteligencia Artificial y Sistemas). Este sistema de navegación lo podemos apreciar en la Figura 29.

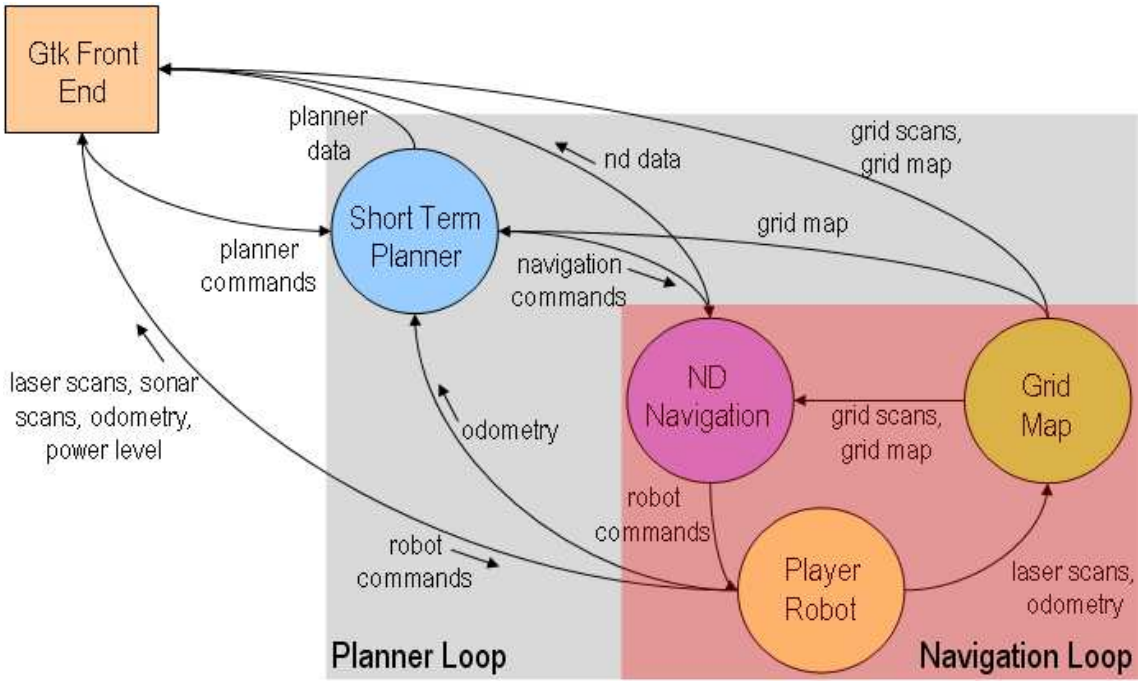


**Figura 29**: Diagrama de un sistema de navegación segura.

El componente que vamos a definir en lenguaje DL se corresponde con el componente *Short Term Planner*. Este componente recibe y manda información en paquetes de puertos a los otros tres componentes, tal y como podemos ver en la Figura 29, y se encarga de implementar una modificación del algoritmo de planificación NF2 de Latombe [Latombe et al., 1991] a modo de planificador a corto plazo durante la navegación y utilizando un mapa de rejilla proporcionado por el componente *Grid Map*.

### 2.10.1. Requisitos del componente Short Term Planner.

Este componente se implementó empleando como base la plataforma CoolBOT, por tanto se trata de un componente CoolBOT escrito directamente en C/C++ y antes de contemplar el código escrito en DL de este componente, vamos a ver los puntos

[2] http://mozart.dis.ulpgc.es/home.html

importantes del mismo para sacar la información necesaria que luego vamos a plasmar en lenguaje DL que lo define.

El componente *Short Term Planner* para realizar su labor requiere que se proporcionen una determinada información que, una vez procesada, envía a otros lugares donde esta información es procesada. Por tanto, este componente requiere que se definan una serie de puertos de entrada y de salida tal y como se muestra en la Tabla 4.

| **Puertos de entrada y puertos de salida** | | | |
|---|---|---|---|
| **Identificador** | **Clase de puerto** | **Tipo de puerto** | **Paquete de Puerto** |
| RobotConfig | entrada | last | PlayerRobot::ConfigPacket |
| Odometry | entrada | last | PlayerRobot::OdometryPacket |
| GridConfig | entrada | last | GridMap::ConfigPacket |
| GridMap | entrada | poster | GridMap::GridMapPacket |
| Commands | entrada | last | CommandPacket |
| PlannerMap | salida | poster | PlannerMapPacket |
| PlannerPath | salida | generic | PlannerPathPacket |
| NavigationCommands | salida | generic | NDNavigation::CommandPacket |
| MatchingRegions | salida | generic | PlannerMatchingPacket |

**Tabla 4**: Puertos de entrada y de salida del componente *Short Term Planner*.

En cuanto al autómata de usuario que define su funcionalidad, el componente *Short Term Planner* define un autómata de usuario formado por cuatro estados, los cuales se pueden ver en la Tabla 5.

| Estados del autómata de usuario | | |
|---|---|---|
| **Identificador** | **Estado de entrada** | **Transiciones** |
| Idle | Si | RobotConfig, GridConfig, GridMap |
| Wait | No | Odometry, GridMap, Commands |
| MainState | No | Odometry, GridMap, Commands, timer |
| NoPlannedMain | No | Odometry, GridMap, Commands, timer |

**Tabla 5**: Estados que conforman el autómata de usuario del componente *Short Term Planner*.

Por último el componente *Short Term Planner* emplea una serie de constantes para uso tanto interno como externo del componente. Estas constantes las podemos ver en la Tabla 6.

| Constantes | | |
|---|---|---|
| **Identificador** | **Tipo de constante** | **Valor** |
| DEFAULT_REPLANNING_CHECK_PERIOD | pública | 5000 |
| ALLOCATION_BLOCK | privada | 25 |
| MAX_PATH_SEGMENT | privada | 8 |
| REPLANNING_SEGMENT | privada | 4 |
| REPLANNING_THRESHOLD | privada | 1000 |
| MIN_REPLANNING_CHECK_TIME | privada | 3000 |

**Tabla 6**: Constantes que se emplean en el componente *Short Term Planner*.

### 2.10.2. Componente Short Term Planner en lenguaje DL.

En el Fragmento de Código 23 podemos ver la implementación del componente *Short Term Planner* en lenguaje *Description Language*.

```
component ShortTermPlanner
{
  header
  {
    author "Antonio C. Domínguez Brito";
    description "ShortTermPlanner component";
    institution "IUSIANI-Universidad de Las Palmas de Gran Canaria";
    version "0.1"
  };

  input port RobotConfig type
             last port packet PlayerRobot::ConfigPacket;
  input port Odometry type
             last port packet PlayerRobot::OdometryPacket;
  input port GridConfig type last port packet GridMap::ConfigPacket;
  input port GridMap type
             poster port packet GridMap::GridMapPacket;
  input port Commands type last port packet CommandPacket;

  constants
  {
    DEFAULT_REPLANNING_CHECK_PERIOD = 5000; // milliseconds
    private ALLOCATION_BLOCK = 25;
    private MAX_PATH_SEGMENT = 8; // in robot diameters
    private REPLANNING_SEGMENT = 4; // in robot diameters
    private REPLANNING_THRESHOLD = 1000; // it is necessary to
                                         // calibrate this value
    private MIN_REPLANNING_CHECK_TIME = 3000; // milliseconds
  };

  output port PlannerMap type poster port packet PlannerMapPacket;
  output port PlannerPath type
              generic port packet PlannerPathPacket;
  output port NavigationCommands type
              generic port packet PlannerMapPacket;
  output port PlannerPath type
              generic port packet NDNavigation::CommandPacket;
  output port MatchingRegions type
              generic port packet PlannerMatchingPacket;

  entry state Idle
  {
    transition on RobotConfig, GridConfig, GridMap;
  };

  state Wait
  {
    transition on Odometry, GridMap, Commands;
  };

  state MainState
  {
    transition on Odometry, GridMap, Commands, timer;
  };

  state NoPlannedMain
  {
      transition on Odometry, GridMap, Commands, timer;
  };
};
```

**Fragmento de Código 23**: Componente *Short Term Planner* en lenguaje DL.

# Capítulo 3
# Desarrollo del trabajo.

El objetivo principal del trabajo presentado en este documento es solventar el problema que vimos en el primer capítulo con la re-compilación de un fichero descriptivo. En este capítulo vamos a hablar de las decisiones que hemos tomado para resolver dicho problema.

Empezaremos realizando un análisis técnico del compilador *coolbotc-1.2.0* donde estudiaremos cuáles fueron las principales líneas de diseño, cómo se estructuró y cómo se implementó. Dado que vamos a realizar una mejora sobre este compilador, el objetivo del análisis será saber qué elementos de la anterior implementación se pueden aprovechar en la nueva implementación y cuáles son los posibles problemas que pueden surgir.

Una vez realizado el análisis anterior, estudiaremos cuáles son los requisitos que debemos tener en cuenta para poder realizar la mejora, planteando la solución que vamos a aplicar.

La siguiente etapa consistirá en el diseño de la mejora que vamos a realizar. Nos apoyaremos en diagramas en notación UML [Larman, 1999] para plasmar la nueva arquitectura del compilador y explicaremos las mejoras que se han incorporado a la nueva versión del compilador.

Por último, una vez que dispongamos del diseño del nuevo compilador, veremos qué recursos necesitamos para llevar a cabo la implementación y cuáles han sido las principales líneas de desarrollo seguidas en la implementación.

## *3.1. Análisis.*

Cuando se planteó el compilador *coolbotc-1.2.0*, se detectó el problema que surgía cuando se realizaba una re-compilación pero no se evalúo su impacto de forma

correcta porque el objetivo principal era disponer de un compilador para el que dado un fichero descriptivo en DL obtuviese sus respectivos esqueletos C/C++.

Para remediar el problema de la re-compilación, en ese momento, se decidió aplicar un sistema de copias de seguridad incremental. Esta solución resolvía el problema pero dificultaba, como posteriormente se comprobó, el desarrollo y mantenimiento de los componentes que se desarrollaban utilizando el compilador.

A continuación vamos a estudiar lo que nos ofrece el actual compilador y partiendo de esta información, vamos a plantear una solución que aplicaremos en este trabajo.

### 3.1.1. El compilador coolbotc-1.2.0.

En el apartado 1.2 del primer capítulo de este documento, comentamos los requisitos que se plantearon que debía seguir el compilador *coolbotc-1.2.0*.

El primer requisito era que debía soportar en su totalidad el lenguaje *Description Language*. Para ello, utilizando las herramientas Flex y Bison[1] se desarrolló un analizador léxico y sintáctico que reconocía toda la sintaxis y gramática del lenguaje DL. Luego para hacer cumplir las reglas semánticas del lenguaje, se implementó en lenguaje C/C++ un analizador semántico que integramos con el analizador sintáctico obtenido con Bison, consiguiendo de esta forma que se soportase en su totalidad el lenguaje *Description Language*.

El segundo requisito consistía en proporcionar al programador información sobre la compilación que se estaba realizando. El compilador por defecto informaba de los errores y advertencias que se producían durante el proceso y para facilitar la depuración de los componentes implementados en DL se amplió esta funcionalidad para facilitar una información extra de depuración. Esto, se tradujo en la incorporación de dos nuevas opciones al compilador: la opción "*-v*" o "*--verbose*" y la opción "*-vf*" o "*--verbosefile*". La opción "*-v*" o "*--verbose*" mostraba la información en pantalla mientras que la opción "*-vf*" o "*--verbosefile*" generaba un fichero con la extensión "*.output*" con dicha información. Ambas opciones mostraban, primero información sobre lo que el analizador léxico y sintáctico había procesado y luego se realizaba un volcado del contenido semántico almacenado por el analizador semántico. En la Figura 30 podemos ver la

[1] Para más información visitar: http://flex.sourceforge.net/ y http://www.gnu.org/software/bison/

información que se muestra con las dos opciones anteriores para el componente *First* visto en el Fragmento de Código 1 en el capítulo 1.

```
Line 1 Column -1: COMMENT - /*
Line 6 Column 1: COMMENT - */
Line 8 Column 0: RESERVED WORD - component
Line 8 Column 11: IDENTIFIER - First
Line 9 Column 0: SYMBOL - {
Line 11 Column 5: COMMENT - /*
Line 13 Column 5: COMMENT - */
Line 15 Column 5: RESERVED WORD - entry
Line 15 Column 12: RESERVED WORD - state
Line 15 Column 19: IDENTIFIER - Main
Line 16 Column 5: SYMBOL - {
Line 17 Column 9: COMMENT - //State's body.
Line 18 Column 5: SYMBOL - }
Line 18 Column 7: Syntactic rule detected: definition of user
automaton state.
Line 20 Column 0: SYMBOL - }
Line 20 Column 1: SYMBOL - ;
Line 20 Column 3: Syntactic rule detected: definition of component.
**************************************
Semantic content:
  Component definition:
        Component Name: First
        --------------------------------
        State definition:
        State Name: Main
        State Type: ENTRY STATE
        State Transitions: Not Defined.
        --------------------------------
**************************************
```

**Figura 30**: Información que se muestra con las opciones: "*-v*" o "*--verbose*" y "*-vf*" o "*--verbosefile*".

El tercer requisito implicaba que el compilador debía ser flexible antes los posibles cambios en la plataforma CoolBOT. En otras palabras, si en la plataforma se incorporaba nuevos elementos y estos debían de aparecer en los esqueletos, el compilador sin sufrir modificación alguna debía de generar los esqueletos C/C++ con estos nuevos elementos. Lógicamente, este requisito estaba limitado a la incorporación de nuevos elementos que no implicaban cambios conceptuales en el compilador.

Para cumplir con este requisito se aplicó una técnica que consistía en utilizar ficheros molde como guías en el proceso de compilación. Tras estudiar los esqueletos C/C++ de un componente CoolBOT, se observo que una parte del esqueleto era común para cualquier componente que se definiera. Por tanto podríamos extraer esa parte y almacenarlo en un fichero de texto que luego utilizaríamos como un molde que todo componente debe seguir. El compilador procesaría este fichero y se encargaría de generar el resto del esqueleto C/C++.

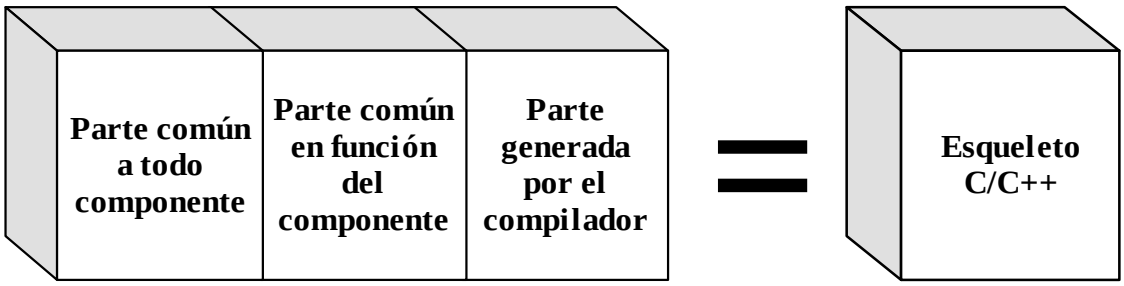


**Figura 31**: Estudio de los esqueletos C/C++.

Estudiando los esqueletos C/C++ se observo que además de esta parte común, se podía distinguir otra parte que varía en función de los requisitos del componente (tal y como podemos ver en la Figura 31). Si en un componente se definían hilos o centinelas, en los esqueletos aparecían trozos de código que sólo aparecían en esos componentes. Partiendo de esta singularidad, observamos que los componentes CoolBOT se podían clasificar en los siguientes tipos:

- *Componentes básicos*. Componentes donde encontramos trozos de código para la implementación de puertos de entrada, puertos de salida, variables controlables, variables observables, excepciones y estados. También se les puede denominar componentes monohilo pues incorporan trozos de código para gestionar el único hilo del que disponen, el hilo *main*.
- *Componentes con hilos o multihilo*. Se trata de componentes básicos que además incluyen trozos de código para la implementación de múltiples hilos (1 ó más hilos sin contar el hilo *main*).
- *Componentes con centinelas*. Se trata de componentes que definen puertos de entrada con centinelas (*watchdogs*) por lo que se incluyen trozos de código especificados para su implementación. En este tipo de componente podemos distinguir dos variantes:
    - *Componentes básicos con centinelas.* Se trata de un componente básico que define puertos de entrada con centinelas.
    - *Componentes con hilos con centinelas*. Se trata de componentes con múltiples hilos que define puertos de entrada asociados con centinelas.

Como resultado de esta clasificación, el compilador *coolbotc-1.2.0* va a disponer de cuatro ficheros molde, uno para cada tipo de componente, de forma que cada molde estaría especializado en la generación de un determinado esqueleto. En verdad, como el compilador debe generar un esqueleto C/C++ para el fichero cabecera (.h) y otro para el fichero fuente (.cpp) de la clase C++ que implementará el componente, necesitaremos 8

ficheros molde más otros 2 ficheros molde para el caso de que se definan nuevos paquetes de puerto [Santana-Jorge, 2007]. A partir del fichero descriptivo que define a cada componente, el compilador internamente se encargará automáticamente de identificar el tipo de componente que se ha definido en el fichero descriptivo y de emplear el fichero molde adecuado.

Como hemos comentado anteriormente, el compilador utilizaba estos ficheros molde como guía en la compilación. Para ello, se añadían en los ficheros molde unas determinadas etiquetas con un formato específico. El compilador cuando procesaba el fichero molde y localizaba una etiqueta, lo primero que hacía era identificar esa etiqueta y luego generaba el código correspondiente a esa etiqueta sin que la etiqueta apareciera en el esqueleto final. Todo lo que no fuera etiquetas, el compilador lo copia directamente del fichero molde al fichero de salida correspondiente. En la Figura 32 podemos ver un ejemplo de esto último, donde se muestra cómo es procesado el fichero molde para el fichero .h de un componente básico con objeto de generar el esqueleto C/C++ correspondiente al fichero cabecera.

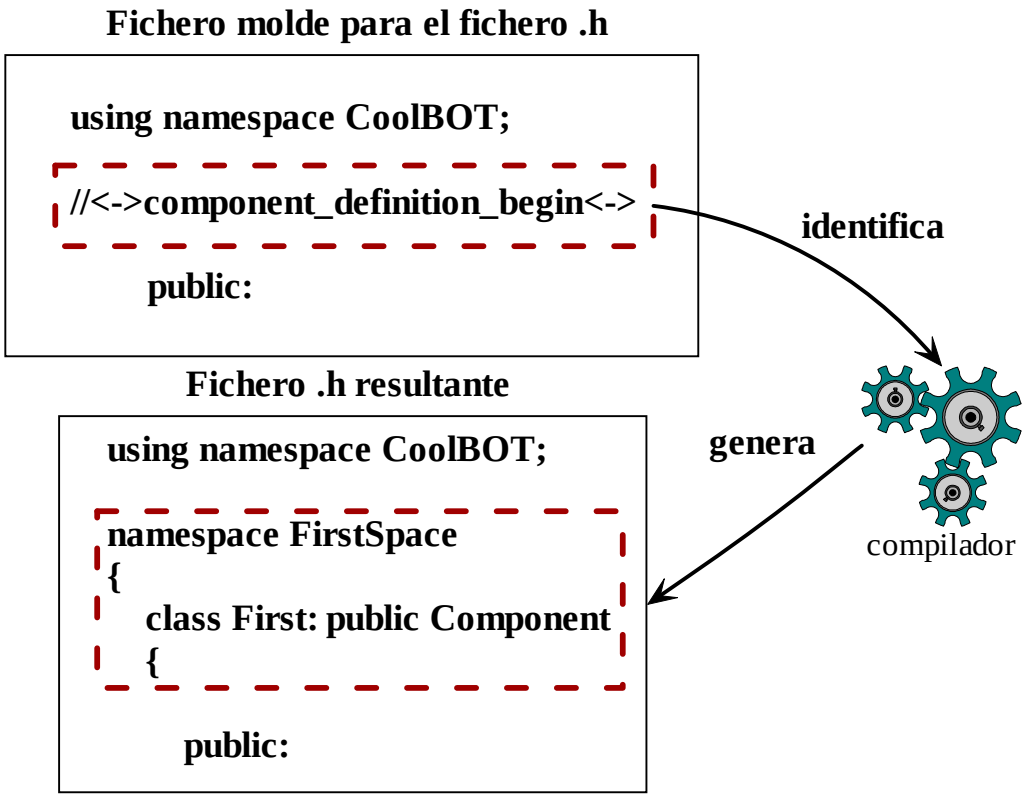


**Figura 32**: Procesamiento de etiquetas por parte del compilador.

Una etiqueta se construye tal y como se muestra en la Figura 33. Primero se define una cabecera formada por los siguientes caracteres: "//<->". Luego se identifica la etiqueta

que se esta definiendo con un identificador y por último, se concluye definiendo un cierre formado por los caracteres "<->".

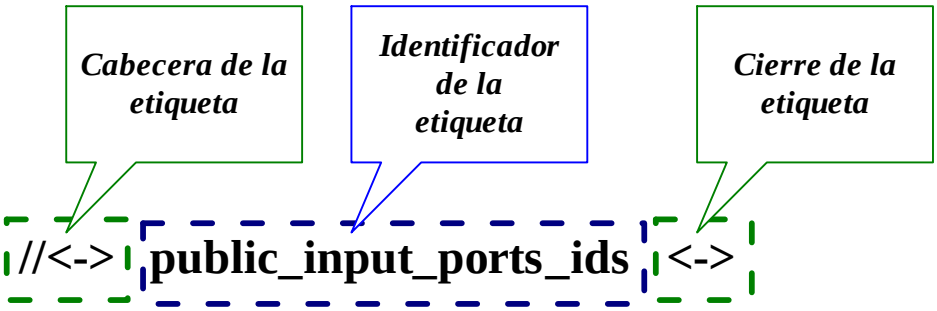


**Figura 33**: Formato de construcción de una etiqueta.

El cuarto requisito exigía que el compilador debía proporcionar los medios necesarios para poder generar automáticamente toda la estructura de ficheros y directorios que exige la plataforma CoolBOT y para poder crear y modificar aquellas variables de entorno que sea necesario.

Para cumplir con este cuarto requisito, el compilador *coolbotc-1.2.0* proporciona la opción "*-c*" o "*--create*" que se encarga de generar toda la estructura de directorios y de ficheros y las variables de entorno necesarias. En la Figura 34, podemos contemplar gráficamente cómo funciona esta opción.

El quinto requisito consistía en generar unos esqueletos C/C++ en caso de que en el fichero descriptivo se definieran nuevos paquetes de puertos específicos del componente. En este caso, el compilador por defecto genera dos esqueletos, uno para el fichero cabecera (.h) y otro para el fichero de implementación (.cpp) que contienen el código C/C++ correspondiente para la implementación de los nuevos paquetes de puertos.

Para la generación de estos esqueletos se dispone de dos ficheros molde, uno para el fichero .h y otro para el fichero .cpp, que contienen los trozos de código y las etiquetas necesarias para que el compilador defina los esqueletos para los nuevos paquetes de puertos.

Por último, el compilador debía implementar un sistema que evitara la pérdida de información en una re-compilación de un fichero descriptivo. De este modo, si por equivocación compilamos un componente, que había sido compilado y posteriormente completado por el programador, no se perderá la información de ese componente.

El sistema que se implantó consistía en un sistema de copias de seguridad incremental. Este sistema tras detectar que se está realizando una re-compilación, renombraba los ficheros .h y .cpp añadiéndoles una extensión formada por la fecha y

hora del sistema más la extensión "*.old*". De esta forma por cada re-compilación que se realizaba se mantenía una copia de seguridad. En la Figura 35 podemos ver un ejemplo.

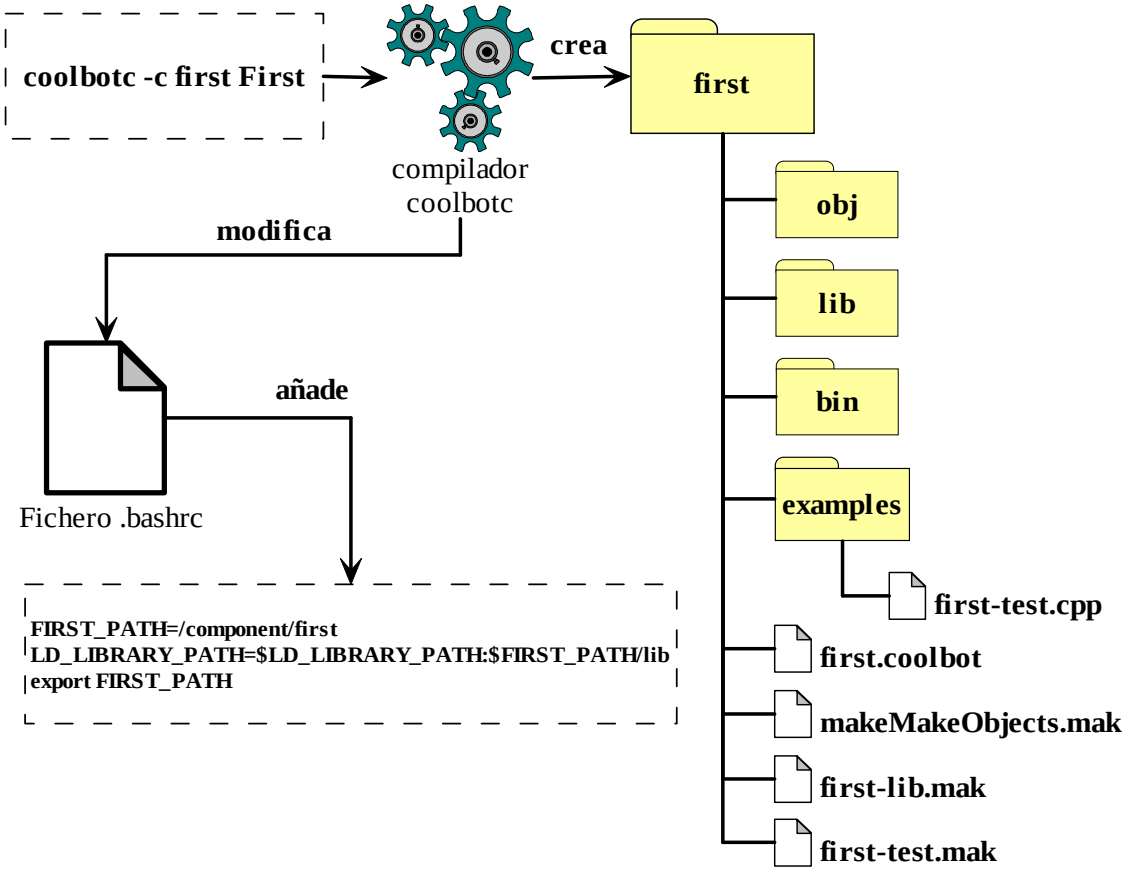


**Figura 34**: Opción "*-c*" o "*--create*" del compildor *coolbotc-1.2.0*.

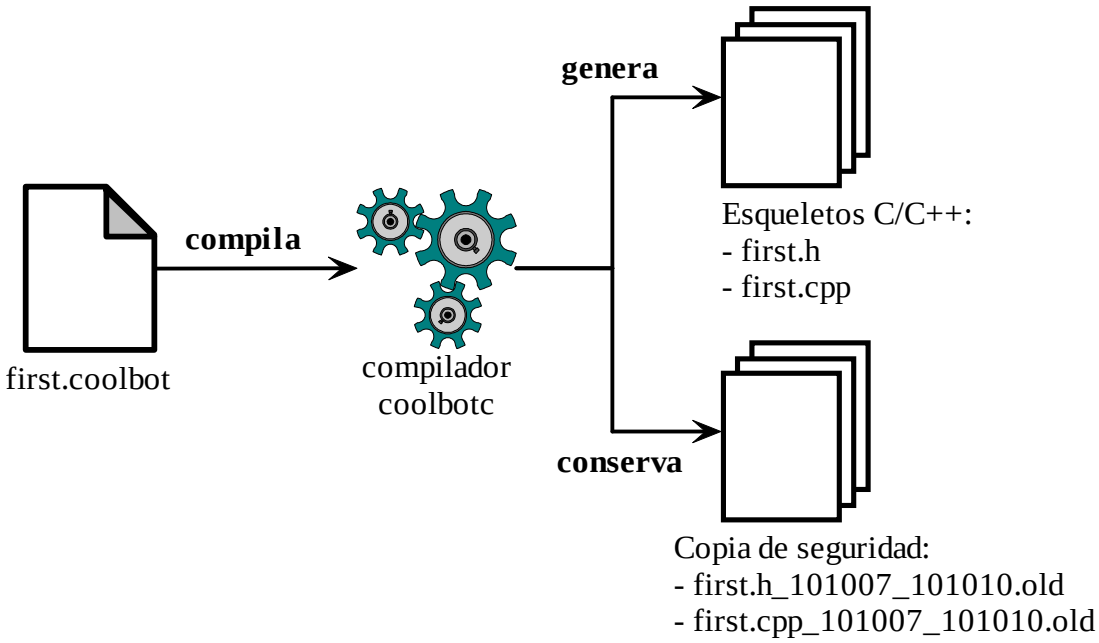


**Figura 35**: Resultado de realizar una re-compilación al fichero descriptivo "*first.coolbot*".

### 3.1.2. Requisitos del nuevo compilador.

En el apartado anterior hemos estudiado la situación actual del compilador *coolbotc-1.2.0*. Sobre el que vamos a realizar una mejora para permitir la re-compilación sin perdida de información, de forma que el programador pueda continuar la implementación del componente por donde lo había dejado antes de volver a realizar la compilación.

La idea es sencilla. Cuando se realiza una re-compilación de un fichero descriptivo, el esqueleto C/C++ resultante debe de conservar todo el código que ha introducido el programador. Para ello, vamos a ampliar la funcionalidad del sistema de etiquetas empleado por el compilador *coolbotc-1.2.0*.

Actualmente, los ficheros molde se encuentra marcados con etiquetas que indican al compilador que es lo que tiene que generar. Estas etiquetas no a parecen en los esqueletos resultantes. Ahora, los ficheros molde seguirán estando etiquetados pero las etiquetas se mantienen en los esqueletos que se generan. De forma que cuando se produzca una re-compilación, el compilador tomará como guía o molde los esqueletos de la última compilación realizada utilizando las etiquetas para saber qué es lo que debe generar y cual es lo qué debe conservar de la compilación previa.

Otro cambio que debe producirse es que en vez de utilizar una sóla etiqueta, se debe utilizar una etiqueta de inicio y otra de fin. Todo lo que está en medio de las etiquetas de inicio y fin será despreciado y generado por el nuevo compilador, conservándose todo lo que no esté encerrado entre etiquetas. En la Figura 36, podemos apreciar el concepto de la etiqueta de inicio y de fin.

La etiqueta de inicio se construye de la misma forma que una etiqueta en el compilador *coolbotc-1.2.0*. Tal y como vimos en la Figura 33, posee una cabecera formada por los siguientes caracteres: “//<->”, un identificador de etiqueta y por último, un cierre que cierra la definición de etiqueta y que está formado por los caracteres “<->”. La etiqueta de fin se construye de la misma manera pero cambiando la cabecera por otra formada por los siguientes caracteres: “//<->/”. El resto mantiene el mismo formato que una etiqueta de inicio.

Complementariamente, a las etiquetas de inicio y fin se les puede añadir más información. De este modo, se pueden introducir uno o dos identificadores que identifiquen con más precisión esa etiqueta. En la Figura 37, se puede apreciar el formato de construcción tanto de la etiqueta de inicio como la de fin con mayor claridad.

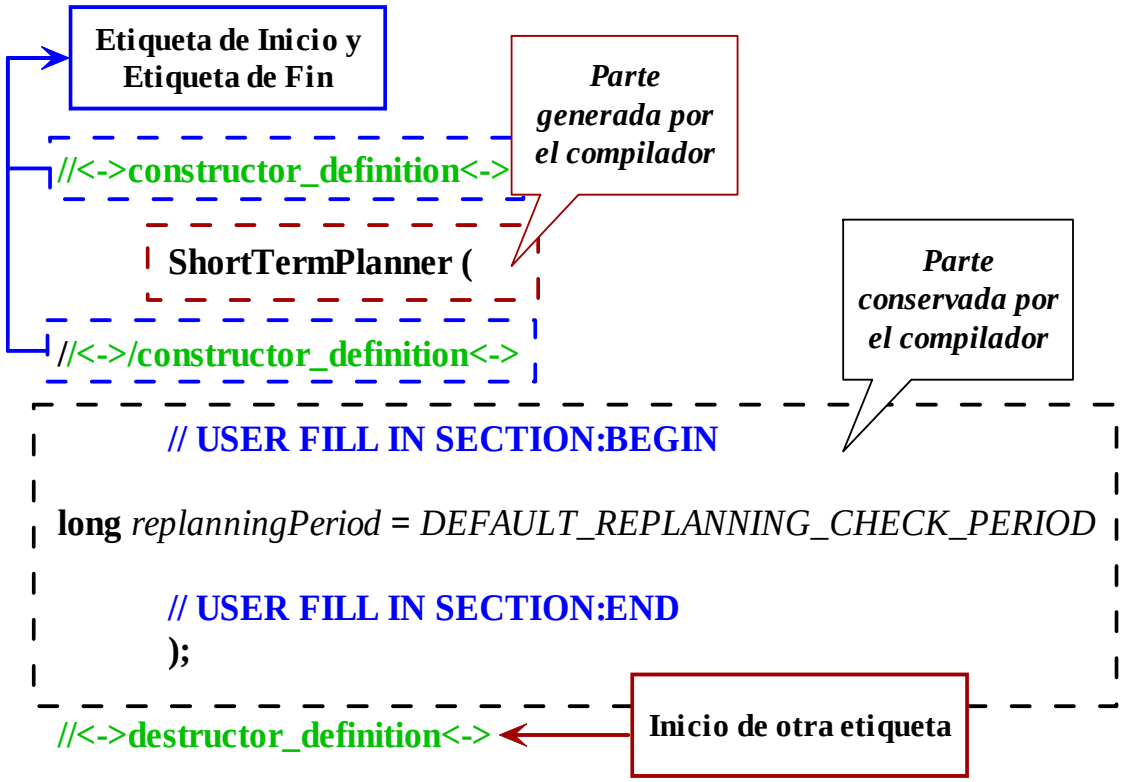


**Figura 36**: Nuevo sistema de etiquetas: la etiqueta de inicio y la etiqueta de fin.

En los esqueletos C/C++, estas etiquetas aparecerán como si fueran comentarios C/C++ de una línea. Las etiquetas comenzarán desde la primera columna y el programador no deberá ni alterar su formato ni cambiarlas de lugar. Para ello, el nuevo compilador dispondrá de medios para detectar:

- Ausencia de una etiqueta. Por ejemplo, se ha definido una etiqueta de inicio pero no se encuentra su etiqueta de fin correspondiente, el compilador deberá alertar al programador sobre la ausencia de la etiqueta de fin.
- Alteración de una etiqueta. Por ejemplo, si el formato de una etiqueta de inicio no coincide con una etiqueta de fin, el compilador deberá alertar al programador sobre la alteración detectada.
- Eliminación de una etiqueta. Por ejemplo, si el programador por error elimina una etiqueta de inicio y de fin, el compilador deberá alertar al programador sobre la ausencia de dicha etiqueta.

Por último, en la aplicación de este sistema de etiquetas debemos de tener muy en cuenta un caso especial: la re-compilación que produce un cambio de tipo de componente. Para entender mejor este caso especial, imaginemos que un programador define un componente con dos hilos denominados *FirstThread* y *SecondThread*.

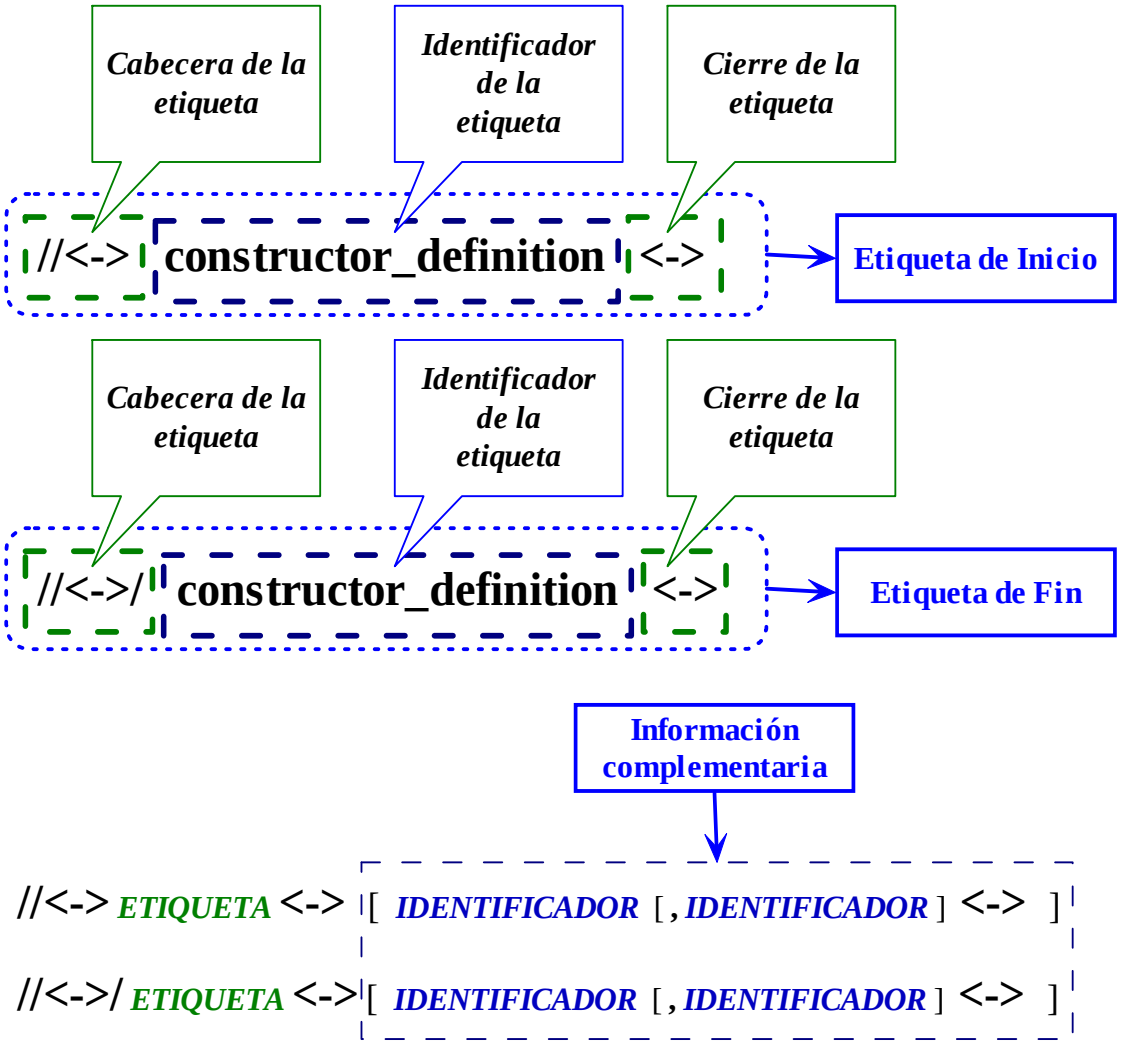


**Figura 37**: Formato de construcción de las etiquetas de inicio y fin.

Cuando se compila el fichero descriptivo, el compilador genera código para un componente de tipo multihilo porque se han definido dos hilos. Más adelante, cuando ha completado la implementación o se encuentra a la mitad, el programador se da cuenta de que no le hacen falta esos dos hilos y como no le hacen falta, los elimina del fichero descriptivo y vuelve a compilar. En este caso, el esqueleto C/C++ resultante deberá conservar todo el código que ha escrito el programador y debe estar libre de todo aquel código que esté relacionado con el soporte y gestión de los componentes multihilo. Es decir, el código resultante debe estar libre de código espúreo.

Con el actual sistema de ficheros de molde, al aplicar este caso especial obtendríamos unos esqueletos con código espúreo. Esto es debido a que los ficheros molde del compilador *coolbotc-1.2.0* están especializado en función del tipo de componente. Por tanto, el nuevo compilador debe desacoplar los ficheros molde para que sean independientes del tipo de componente que representan.

Por consiguiente, el nuevo compilador dispondrá de dos ficheros molde genéricos (uno para el fichero .h y otro para el fichero .cpp) en vez de los 8 del anterior compilador. Estos ficheros estarán etiquetados de tal forma que se puedan utilizar como guía durante la compilación, independientemente de la tipología del componente, estando las partes específicas en un fichero molde aparte. De esta forma, si se da un caso especial como el anteriormente mencionado, los esqueletos C/C++ estarán libres de código espúreo. En la Figura 38 podemos apreciar el sistema de ficheros moldes del compilador *coolbotc-1.2.0* y la nueva modificación.

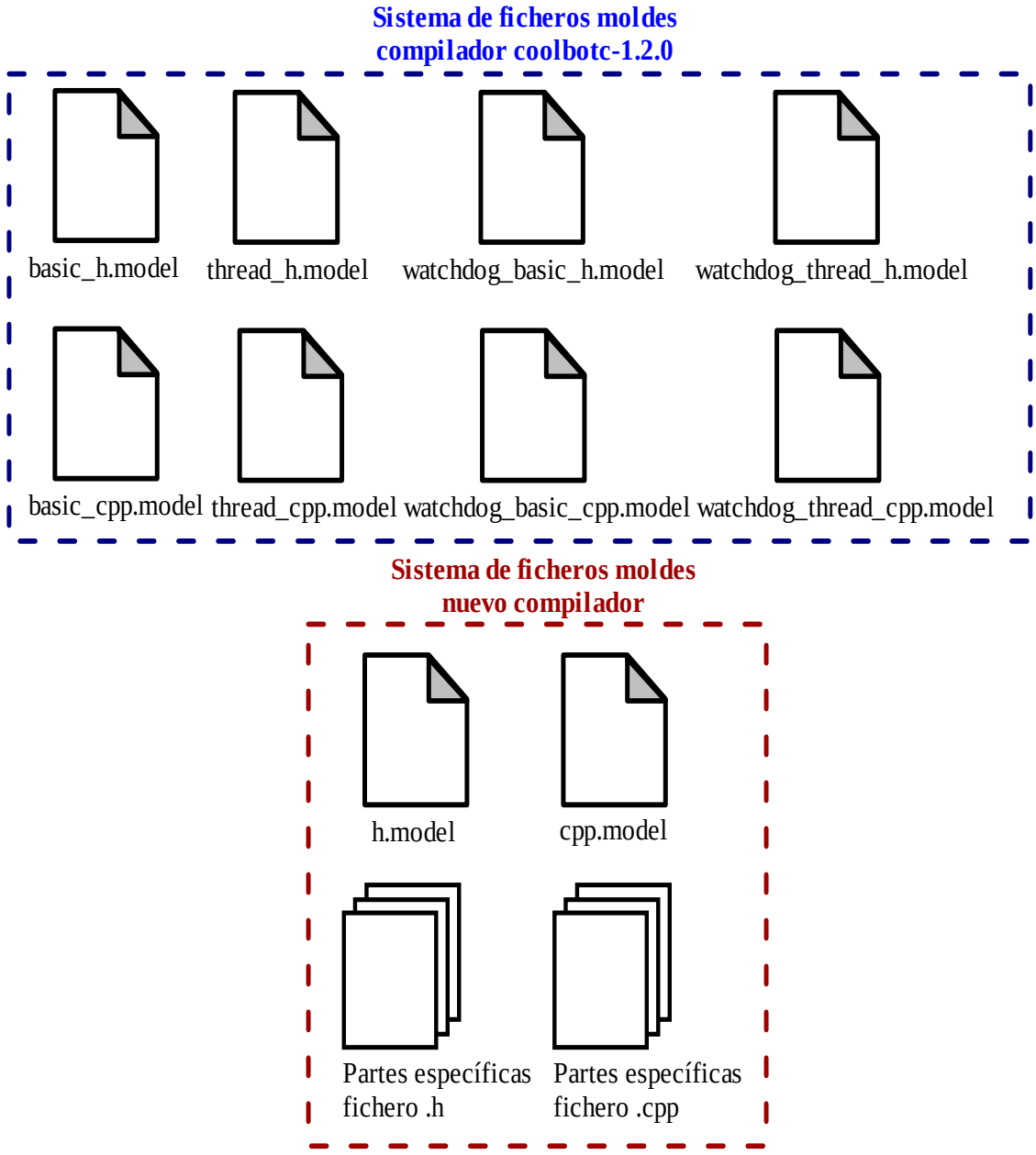


**Figura 38**: Sistema de ficheros moldes del compilador *coolbotc-1.2.0* y la nueva modificación.

## *3.2. Diseño e implementación.*

En la Figura 39, podemos ver, utilizando diagramas UML [Larman, 1999], la arquitectura del compilador *coolbotc-1.2.0*. Esta arquitectura estaba enfocada para el lenguaje C/C++ y está formada por los siguientes módulos:

- *Compiler*. Este módulo contiene el analizador léxico, el analizador sintáctico y el programa principal. El analizador léxico es proporcionado por la herramienta Flex y el analizador sintáctico por la herramienta Bison. Dentro del analizador sintáctico se encuentra incorporado el programa principal.
- *Common*. Contiene declaraciones y clases empleadas por los módulos *Skeleton* y *Semantic*.
- *Errors*. Contiene clases para el tratamiento de errores y de advertencias.
- *Util*. Contiene clases que proporcionan utilidades como el tratamiento de cadenas de caracteres, chequeo de identificadores y acceso al shell del sistema. Se emplea por los módulos *Skeleton* y *Semantic*.
- *Semantic*. Este módulo contiene declaraciones y clases para el tratamiento semántico del lenguaje DL. En otras palabras, contiene el analizador semántico.
- *Skeletons*. Contiene declaraciones y clases para la generación de código en C/C++.

Los módulos *Common*, *Errors*, *Util*, *Semantic* y *Skeletons* son accesibles en el módulo *Compiler* mediante una clase que contiene la implementación del patrón *Facade*, el cual me permite desacoplar el software reduciendo la complejidad y minimizando las dependencias entre los subsistemas que lo conforman. De esta forma, todos los módulos anteriores quedan encapsulados en otro módulo denominado CoolBOT.

Este diseño, en el nuevo compilador no es aplicable en su totalidad. Siguiendo el criterio de que el compilador debe ser fácil de mantener, debemos de realizar una pequeña reordenación y estructuración, siendo el cambio más importante la incorporación de la herramienta Flex en la fase de generación de código. Con esta incorporación, se ha logrado más simplicidad y sencillez en la fase de generación de código mejorando de esta forma el mantenimiento. Aparte de esto, se ha optimizado parte del código existente y corregido pequeños fallos que no se habían detectado en la versión anterior.

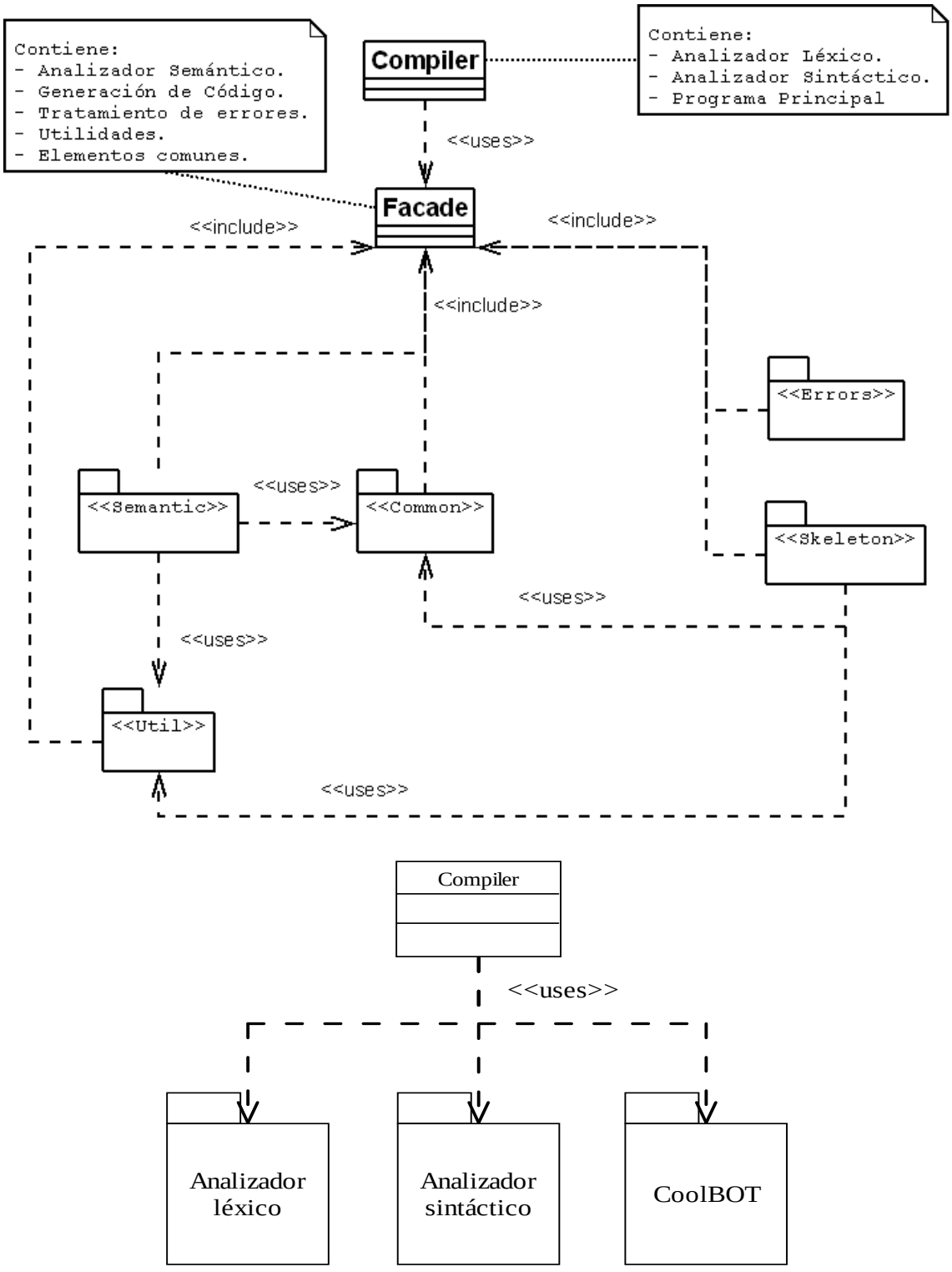


**Figura 39**: Arquitectura del compilador *coolbotc-1.2.0*.

### 3.2.1. Arquitectura del software.

En la Figura 40, podemos ver la arquitectura del nuevo compilador. Aparentemente es muy parecido a la arquitectura del compilador *coolbotc-1.2.0* pero, como veremos a continuación, ha habido pequeñas variaciones.

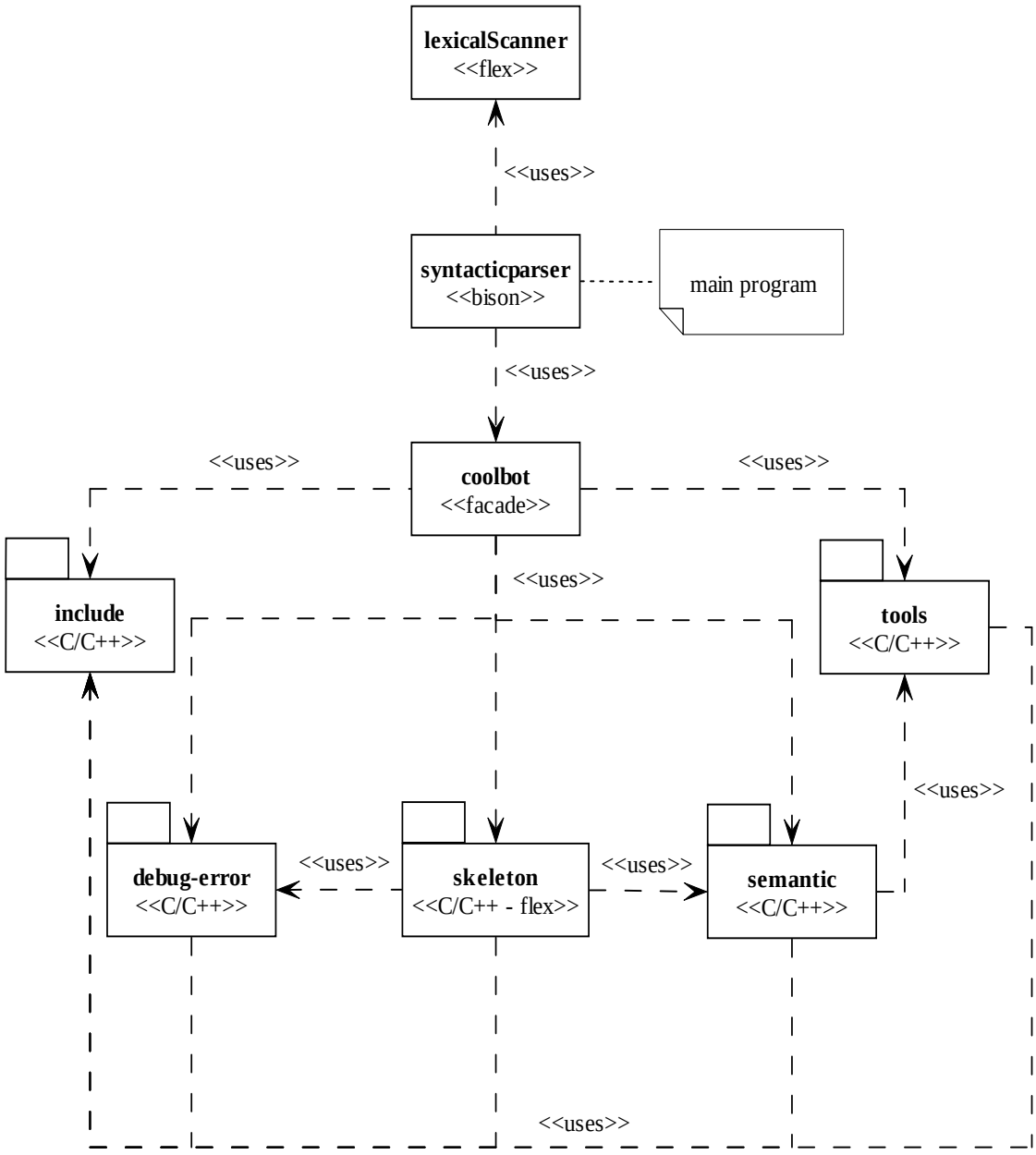


**Figura 40**: Arquitectura del nuevo compilador.

El módulo *include* contiene un fichero con todas las constantes que se emplean en el nuevo compilador (fichero *const.h*) y un fichero que contiene las macros para el tratamiento de los mensajes de error y de advertencia (fichero *locale.h*). Este módulo es usado por todos los módulos integrados en el módulo *coolbot*.

El módulo *debug-error* es un módulo empleado para la depuración y para el tratamiento de errores y advertencias y está formado a su vez, como se aprecia en la Figura 41, por tres módulos: *error*, *warning* y *debugger*. El primero se encarga del tratamiento de errores, el segundo del tratamiento de advertencias y el último se encarga mostrar la información en las depuraciones. Este último módulo esta formado a su vez por 8 módulos, cada uno de los cuales proporciona una funcionalidad para la depuración de un elemento del lenguaje DL.

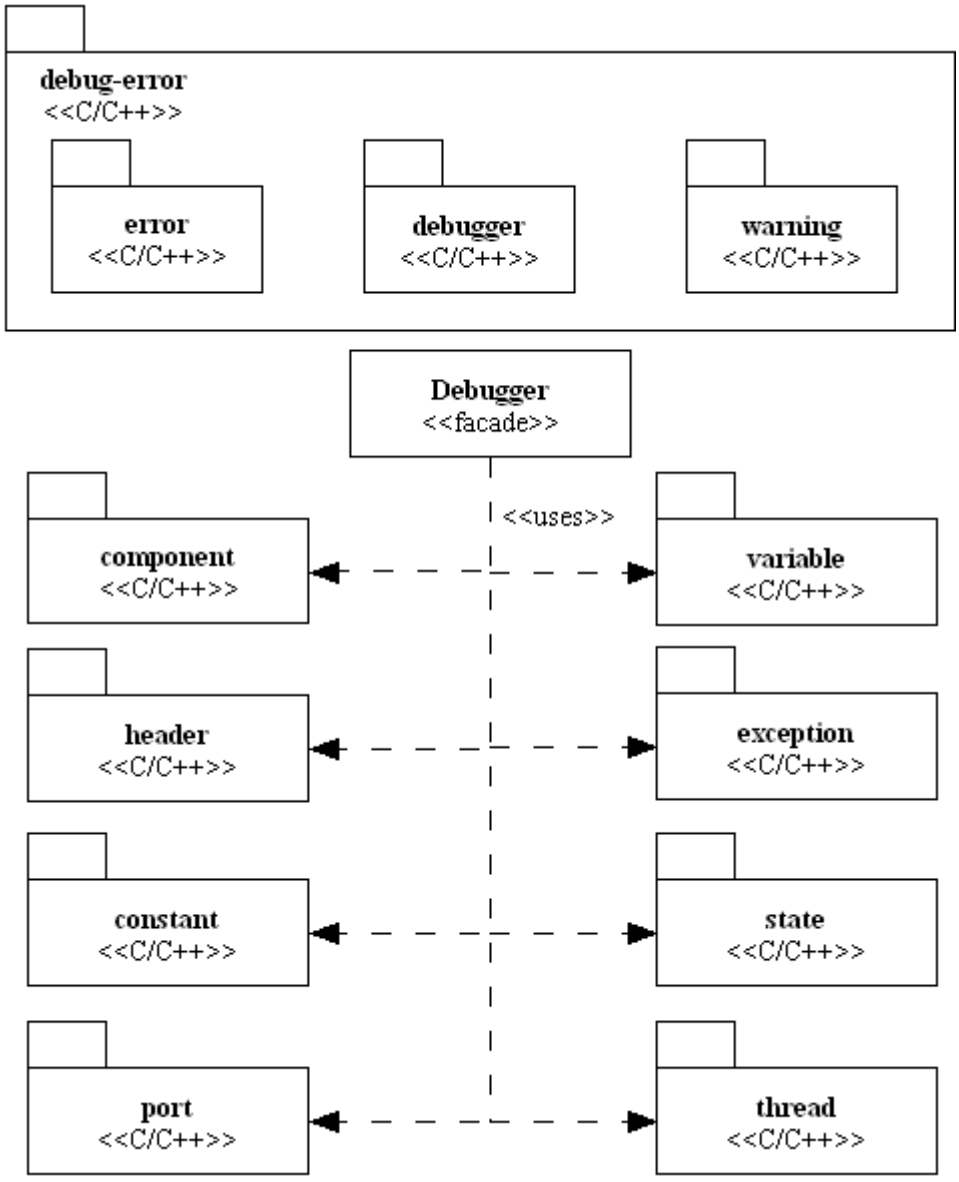


**Figura 41**: Módulo *debug-error*.

El módulo *tools* es el módulo que contiene utilidades y herramientas que emplean los módulos *semantic* y *skeleton*. Está formado por los siguientes tres módulos:

- Módulo *identifier*. Contiene la clase *Identifier* y se trata de un contenedor de identificadores.

- Módulo *util*. Este módulo proporciona funcionalidad para el tratamiento de cadenas (clase *StringLibrary*), acceso al shell del sistema (clase *Shell*) y para generar la salida del compilador (clase *File*).
- Módulo *value*: Contiene la clase *Value* y se trata de un contenedor de valores literales numéricos y de literales cadena.

El módulo *semantic* es el módulo que implementa el analizador semántico del lenguaje DL. Como vemos en la Figura 42, este módulo está formado por otros módulos que implementan clases contenedoras que almacena información semántica y que proporcionan métodos, aparte de los métodos propios de acceso y modificación (métodos Get, Set,…), para realizar chequeos y comprobaciones.

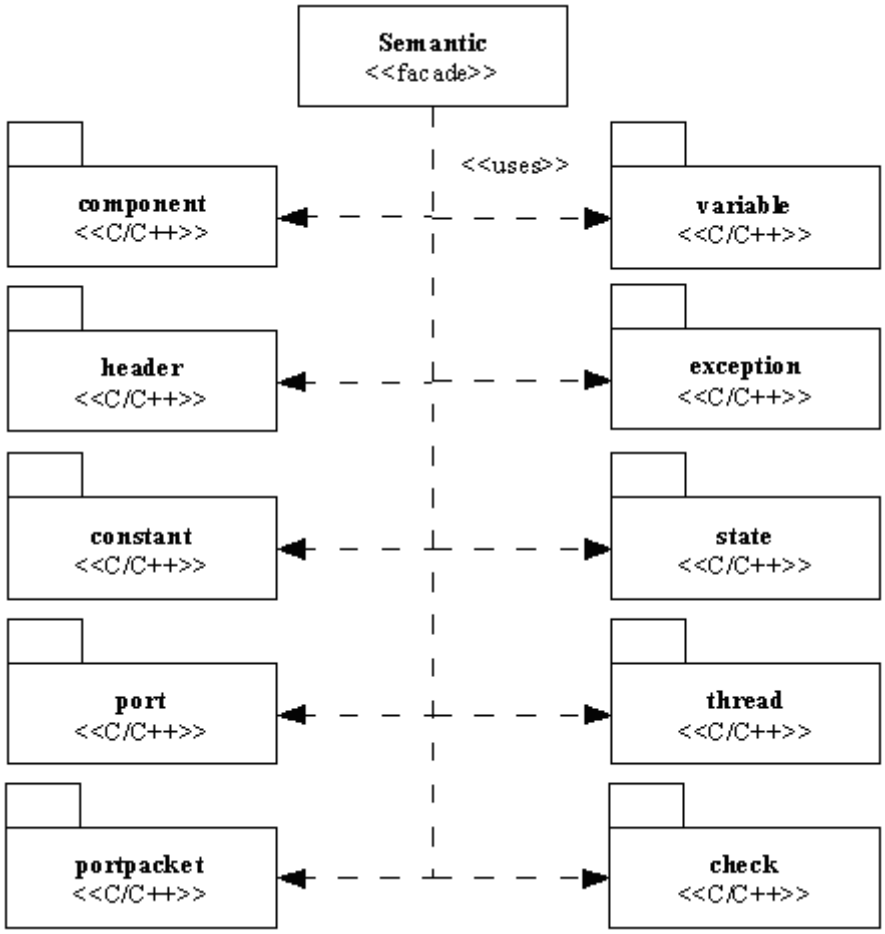


**Figura 42**: Módulo *semantic*.

El módulo *skeleton* se encarga de realizar la generación de esqueletos C/C++. Como vemos en la Figura 43, este módulo posee un módulo denominado *scanner*. Este módulo es el encargado de realizar la generación de código y está implementado con la herramienta Flex. Exactamente lo que realiza es detectar etiquetas y generar el código

asociado a dichas etiquetas. Dentro de este módulo distinguimos los siguientes reconocedores:

- *scanFileH.l* y *scanFileCPP.l.* Se encarga de la generación de esqueleto C/C++ para el fichero cabecera y el fichero de implementación de un componente (fichero .h y .cpp).
- *scanPacketFileH.l* y *scanPacketFileCPP.l.* Se encarga de la generación de esqueleto C/C++ para la definición de nuevos paquetes de puertos a especificar por el usuario (ficheros ***–packets.h y ***-packets.cpp).

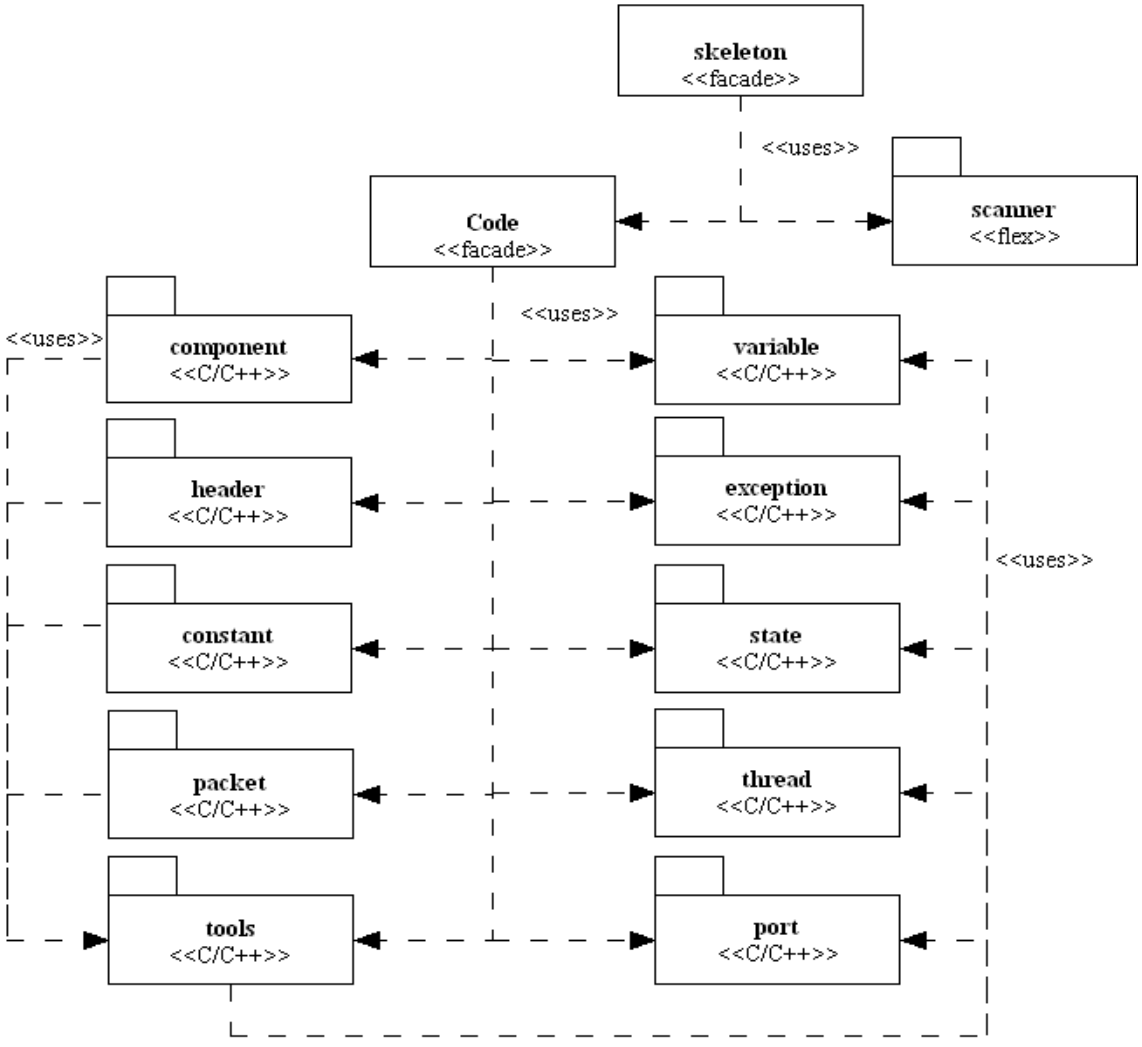


**Figura 43**: Módulo *skeleton*.

La clase *Code* implementa el patrón *Facade* y contiene todos los métodos necesarios para llevar a cabo la generación de los esqueletos. El módulo *scanner* hace uso de esta clase invocando a los métodos de la misma cuando detecta una determinada etiqueta. La clase *Code* tiene acceso a una serie de módulos como vemos en la Figura 43. Cada módulo implementa una clase que se encarga de proporcionar los métodos necesarios

para generar el código C/C++ de un determinado elemento del lenguaje DL. De estos módulos, el módulo *tools* proporciona unas clases que se emplean mucho en la generación de código. Estas clases son las siguientes:

- Clase *Mapping*. Clase empleada para mapear puertos, estados, variables e hilos a un formato específico que se emplea en la generación de esqueletos en la parte de mapeo de puertos, hilos de puertos y máscaras de estados.
- Clase *Tag*. Clase utilizada para generar etiquetas, tanto de inicio como de fin.
- Clase *Register*. Clase que se emplea para registrar etiquetas y para determinar la ausencia de etiquetas.

### 3.2.2. Implementación, recursos necesarios.

Una vez planteado el diseño, pasamos a implementar la solución y para ello, requerimos de una serie de recursos tanto hardware como software. A continuación, realizaremos una pequeña descripción de cada uno de los recursos empleados:

- *Recursos hardware*. Ordenador PC Pentium IV con al menos 512 MB, USB 2.0 y con conexión a Internet proporcionado por el grupo de investigación GIAS, situado en el laboratorio del Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (IUSIANI).
- *Recursos software*. El software empleado es software libre, empleándose las siguientes aplicaciones:
    - *Sistema operativo*. GNU/Linux Ubuntu.
    - *Bison* (GNU parser generador). Generador sintáctico compatible con YACC empleado para implementar el analizador sintáctico.
    - *Flex* (The Fast Lexical Analyzer). Generador léxico empleado para implementar el analizador léxico.
    - *GCC* (the GNU Compiler Collection). Compilador GNU del lenguaje C/C++.
    - *cgdb*. Depurador GNU para depurar código C/C++.
    - *Make*. Heramienta GNU de generación o automatización de código empleado para generar el ejecutable de nuestro compilador.
    - *KDevelop*: IDE utilizado empleado para desarrollar el compilador.
    - *Meld* (Diff and merge tool). Herramienta GNU empleado para comparar ficheros y localizar diferencias entre ellos.

# Capítulo 4

# Prueba y Resultados.

La prueba [Pressman, 2006] es una parte muy importante en este proyecto, no sólo por su importancia en el logro de resultados correctos sino por el tiempo y recursos requeridos en la realización de las mismas. Su objetivo es asegurar que el nuevo compilador (al que denominadores *coolbotc-1.3.0*) cumple con los objetivos marcados en este trabajo y lo haga con unos mínimos de calidad.

En este capítulo, comenzaremos hablando de los objetivos a cumplir en la prueba. Hablaremos sobre el plan seguido en la fase de pruebas y los casos de prueba realizados.

Por último, hablaremos sobre los resultados obtenidos al utilizar el nuevo compilador con respecto al antiguo compilador *coolbotc-1.2.0*.

## *4.1. La prueba.*

Los objetivos a cumplir en la fase de prueba son los siguientes:

1. Verificar que el compilador *coolbotc-1.3.0* soporta todas las características léxicas, sintácticas y semánticas del lenguaje *Description Language*.
2. Verificar que el compilador *coolbotc-1.3.0* soporta correctamente todas las opciones extras (opciones "*-v*", "*-vf*",…) que ofrecía el compilador *coolbotc-1.2.0*.
3. Verificar que el compilador *coolbotc-1.3.0* genera correctamente los esqueletos C/C++ de un componente como lo haría el compilador *coolbotc-1.2.0*. Se verificarán también la generación de esqueletos para la definición de nuevos paquetes de usuarios definidos por el usuario.

4. Verificar que el compilador *coolbotc-1.3.0* tras realizar una re-compilación obtiene unos esqueletos C/C++ sin pérdida de información y preservando lo que el programador haya añadido en los esqueletos C/C++.
5. Verificar que el compilador *coolbotc-1.3.0*, tras realizar una re-compilación, ha aplicado una copia de seguridad de forma exitosa.
6. Verificar que el compilador *coolbotc-1.3.0* tras realizar una re-compilación fallida, el sistema de vuelta atrás haya funcionado correctamente.
7. Verificar que el compilador *coolbotc-1.3.0* soluciona el caso especial visto en el apartado 3.1.2 del anterior capítulo (la re-compilación produce un cambio de tipo de componente).

Para verificar el cumplimiento de estos objetivos, se elaboró un plan de prueba. En este plan, la prueba la dividimos en dos partes: la prueba para el *front-end* y la prueba para el *back-end* del nuevo compilador.

La prueba para el *front-end* se encarga de probar el correcto funcionamiento del analizador léxico, sintáctico y semántico del nuevo compilador. Para ello, el compilador *coolbotc-1.3.0* incluye una nueva opción (la opción "*-nc*" o "*--notcompile*") que permite compilar un fichero descriptivo sin generar los respectivos esqueletos C/C++.

Las pruebas específicas que se realizaron para el *front-end* se plantearon en función del tipo de analizador. A continuación, podemos ver los objetivo de las pruebas que se realizaron para cada tipo de analizador:

- *Analizador Léxico*. Las pruebas que se realizaron se centraron en:
    - Verificar que los identificadores empleados en un fichero descriptivo cumplen con las reglas de construcción del lenguaje DL.
    - Verificar que los comentarios empleados en un fichero descriptivo se ajustan a las reglas definidas en el lenguaje DL.
    - Verificar que los literales cadenas definidos en un fichero descriptivo cumplen con las reglas definidas en el lenguaje DL.
- *Analizador Sintáctico*. Las pruebas que se realizaron se centraron en verificar que las reglas sintácticas empleadas en el lenguaje DL se reconocen perfectamente, detectándose correctamente los errores de sintaxis.
- *Analizador Semántico*. Las pruebas que se realizaron se centraron en:

- Verificar que los identificadores empleados en un fichero descriptivo no se encuentren duplicados ni sean palabras reservadas ni coincidan con el identificador del componente que se define en el fichero descriptivo.
- Verificar el correcto cumplimiento de las distintas reglas semánticas de los distintos elementos que podemos definir en el lenguaje DL.
- Verificar el correcto volcado del contenido semántico correspondiente a un fichero descriptivo analizado.

En la realización de las distintas pruebas planteadas en el párrafo anterior, nuestra atención se centrará en los siguientes puntos:

1. El compilador *coolbotc-1.3.0* procesa correctamente el fichero descriptivo sin detectarse problemas de violación de segmento o *core dump*[1].
2. Se cumple con el objetivo marcado en esa prueba.
3. El compilador *coolbotc-1.3.0* detecta y muestra correctamente los errores y advertencias producidos durante el proceso.

La prueba para el *back-end* se encarga de probar el correcto funcionamiento de la generación de los esqueletos C/C++ tanto para un proceso de compilación como para un proceso de re-compilación. Para ello, se plantearon pruebas específicas que consistían en lo siguiente:

- Verificar que el esqueleto C/C++ generado es correcto y refleja correctamente todo lo definido en el fichero descriptivo.
- Verificar que tras realizar una re-compilación se obtiene esqueletos C/C++ sin pérdida de información.
- Verificar que tras realizar una re-compilación se ha generado también una copia de seguridad de forma exitosa.
- Verificar que tras realizar una re-compilación fallida, el sistema de vuelta atrás haya funcionado correctamente.
- Verificar que si en la re-compilación se produce un cambio de tipo de componente, el esqueleto C/C++ resultante esta libre de código espúreo.

En la realización de las distintas pruebas de la parte del *back-end*, nos vamos a centrar en los siguientes puntos:

[1] Para más información, visitar: http://es.wikipedia.org/wiki/Violación_de_acceso

1. Éxito en la compilación. El compilador *coolbotc-1.3.0* procesa correctamente el fichero descriptivo y detecta y muestra correctamente los errores y advertencias producidos durante el proceso.
2. Esqueletos C/C++ correctos. Comprobamos que el esqueleto generado se corresponde a lo definido en el fichero descriptivo, que sea correcto y que no haya pérdida de información en caso de una re-compilación. En este punto, dado el elevado número de líneas que posee un esqueleto C/C++, nos apoyaremos en la herramienta *meld* que nos permite comparar dos ficheros de texto de forma gráfica mostrando las diferencias habidas entre los dos ficheros.
3. Compilación con el compilador C/C++ y con la plataforma CoolBOT. Para evitar que algo se nos pase por alto durante la comprobación del esqueleto, compilamos los esqueletos con el compilador C/C++ y con la plataforma CoolBOT. Si la compilación es exitosa, el esqueleto C/C++ fue generado correctamente.
4. Ejecución del componente. Con la compilación con el compilador C/C++ y con la plataforma CoolBOT, hemos comprobado que la sintaxis del esqueleto C/C++ es correcta pero nos falta por verificar que lo generado funcione correctamente. Para ello, ejecutamos un programa de prueba (asumimos que el componente se ha definido usando la opción "*-c*" o "*--create*") que consiste en lanzar el estado *running* (estado que contiene el Autómata de Usuario definido en el fichero descriptivo), pararlo durante 1000 milisegundos para luego suspenderlo y por último matarlo (transita al estado *dead* del Autómata por Defecto). Si el programa de prueba se ejecuta correctamente, el esqueleto C/C++ generado es correcto.

## *4.2. Resultados.*

Tras realizar la fase de prueba y corregir los fallos detectados, comprobamos que el nuevo compilador *coolbotc-1.3.0* procesa correctamente un fichero descriptivo y genera correctamente su respectivo esqueleto C/C++. Por tanto, disponemos de una versión operativa que nos proporciona una herramienta que cumple con el objetivo principal marcado en este trabajo. El compilador *coolbotc-1.3.0* nos permite realizar una re-compilación todas las veces que queramos sin que se produzca pérdida de información, cosa que no permite el anterior compilador *coolbotc-1.2.0*.

Si realizamos una comparación entre los dos compiladores con respecto a su diseño e implementación, observamos las siguientes mejoras que proporciona el compilador *coolbotc-1.3.0*:

- Diseño e implementación del analizador léxico con la herramienta Flex. Aunque mantiene las mismas reglas que el compilador *coolbotc-1.2.0*, se han optimizado siendo más clara y simple la nueva implementación.
- Diseño e implementación del analizador sintáctico con la herramienta Bison. Mantiene las mismas reglas que el compilador *coolbotc-1.2.0* pero se ha optimizado y se ha reescrito el programa principal facilitando aún más el mantenimiento.
- Diseño e implementación del analizador semántico. Se ha reescrito y reestructurado el analizador semántico. Aparte de esto, el código se ha optimizado y mejorado, consiguiendo que sea más fácil de mantener, de aprender y de seguir.
- Diseño e implementación de la generación de código. Se ha reescrito y reestructurado la generación de los esqueletos. Ahora, la generación de los esqueletos C/C++ se implementa a través de la herramienta Flex y se provee soporte para la re-compilación.
- Seguridad y robustez. Se han mejorado y ampliado las comprobaciones para facilitar aún más la tarea al programador, mejorando el sistema de copias de seguridad e incorporando el sistema de vuelta atrás en caso de fallo.

En resumen, se ha logrado obtener una herramienta mucho mejor que la anterior y que cumpla el objetivo principal de este trabajo.

# Capítulo 5

# Conclusiones y trabajo futuro.

A lo largo del presente trabajo, hemos visto como ha sido desarrollado el compilador *coolbotc-1.3.0*. Hemos comprobado que el nuevo compilador ha sido rediseñado con objeto de mejorarlo y hacerlo más eficiente en cuanto a código y a los esqueletos C/C++ que genera. Además, cumple el objetivo principal de este trabajo: se permite re-compilar un fichero descriptivo tantas veces como queramos, conservando en cada compilación el código introducido por el programador en los esqueletos C/C++.

Ahora, en este capítulo, vamos a presentar las conclusiones principales de este trabajo así como las futuras líneas de desarrollo.

## *5.1. Conclusiones.*

La compilador *coolbotc-1.3.0* resuelve eficientemente el problema que el compilador *coolbotc-1.2.0* no solucionaba. Si un programador a mitad del desarrollo de un componente debe redefinir el fichero descriptivo para que su componente refleje nuevas características, puede volver a compilar el fichero descriptivo sin temer a perder los datos que ha introducido en los esqueletos C/C++. Como se produce una re-compilación sin pérdida de información, el programador puede continuar el desarrollo de su componente CoolBOT por donde lo dejó. Ésta es una gran mejora que evita al programador un proceso tedioso y repetitivo, propenso a error que debería llevarse a cabo cada vez que se modifica el fichero descriptivo y se vuelve a compilar.

Además, se ha mejorado la implementación con respecto al compilador anterior. El nuevo compilador se ha implementado de forma que sea más simple de entender, mantener y de ampliar. Mantiene las mismas características que el compilador *coolbotc-1.2.0*, corrigiendo pequeños fallos no contemplados anteriormente. Se ha flexibilizado aún más la tarea al programador mejorando las comprobaciones semánticas y

proporcionando información adicional sobre los errores y advertencias detectados durante la compilación.

Además, proporciona un sistema de vuelta atrás en la re-compilación. De esta forma, si durante una re-compilación se produce algún fallo, el compilador retrocede al estado anterior a la compilación dejando todo como estaba antes de volver a compilar el fichero descriptivo.

En resumen, en el presente trabajo, hemos logrado una nueva versión de una herramienta que es más segura, robusta y eficiente y que facilita aun más el desarrollo de componentes CoolBOT. Con esta herramienta, el programador sólo tiene que plantear qué es lo que quiere y luego implementar directamente la funcionalidad necesaria para conseguir su objetivo.

## *5.2. Trabajo futuro.*

Como cualquier cosa, todo es mejorable y por tanto, las posibles líneas de desarrollo son muy amplias. En el trabajo que originó el compilador *coolbotc-1.2.0* ya se plantearon algunas líneas, siendo una de ellas la que inspiró este trabajo. Algunas ideas que se plantearon en el anterior trabajo aún están vigentes.

A continuación, vamos a comentar las futuras líneas de desarrollo que se pueden seguir a partir de los resultados obtenidos en este trabajo. Para ello, hemos agrupado las ideas en los siguientes grupos:

- Lenguaje descriptivo *Description Language*. Aunque el actual lenguaje contiene todas las principales características que ofrece la plataforma CoolBOT, sería interesante ampliar el lenguaje para poder soportar nuevos conceptos como la definición de componentes compuestos (el compilador *coolbotc-1.3.0* sólo soporta componentes atómicos, ver [Domínguez-Brito, 2003] para más información) o para permitir especificar funcionalidad en el fichero descriptivo.
- Sobre re-compilación. El sistema actual de etiquetas es efectivo pero sensible a la malicia del programador. Aunque el compilador *coolbotc-1.3.0* ofrece mecanismos para detectar la ausencia, alteración o eliminación de etiquetas, se podría desarrollar otras soluciones no triviales que solucionaran el problema sin usar etiquetas. En ese trabajo se plantearon las siguientes ideas:
    - Realizar un *parser* del lenguaje C/C++. De esta forma, en una re-compilación, se procesaría el esqueleto generado en la anterior

compilación sin tener que emplear etiquetas. El problema que plantea esta posible solución es que hay que diseñar y construir un *front-end* (analizador léxico, sintáctico y semántico) para el lenguaje C/C++, lo cual en 200 horas (tiempo máximo estimado para la realización de este trabajo) no sería posible por su complejidad.

  - Aplicar el concepto del comando GNU/Linux *merge*. El comando GNU/Linux *merge* permite fusionar tres archivos en uno solo y la idea sería generar de nuevo los esqueletos en una re-compilación y luego fusionar los nuevos esqueletos con la copia de seguridad que se realiza. El problema que plantea es que el código resultante debe evitar tener código espúreo y se necesitaría mucho tiempo para establecer unos criterios que permitan el fusionado sin código espúreo ya que no es una tarea trivial.

- Desarrollo de un entorno integrado de desarrollo (IDE). Disponer de un IDE especializado en la plataforma CoolBOT sería muy interesante pues se lograría mejorar el desarrollo de los componentes y se aumentaría la productividad. La idea sería disponer un entorno que tome por un lado la plataforma CoolBOT, el compilador *coolbotc-1.3.0*, por otro lado el compilador y depurador GNU del lenguaje C/C++ y lo juntase todo con sus respectiva documentaciones en una única herramienta que disponga de un editor de texto. De esta forma, el programador dispondría de todos elementos necesarios parar desarrollar un componente CoolBOT de forma simple y sencilla.
- Divulgación del trabajo. Lógicamente, la herramienta se ha desarrollado para ser usada ya que es el único medio de corregir fallos y de mejorarla. La decisión de usarla o no está en manos del programador y para dar más peso a su decisión sería interesante desarrollar un sitio Web para divulgar tanto este trabajo como la plataforma CoolBOT donde un programador pudiese encontrar información de soporte, ejemplos, manuales, tutoriales, etc. De esta forma se puede establecer una comunidad de usuarios que a través de sus opiniones y críticas mejoraría tanto el compilador como la plataforma.

# Referencias y Bibliografía.

**[bison]** Página oficial del proyecto *Bison*. http://www.gnu.org/software/bison/

**[Booch et al., 1999]** Grady Booch, James Rumbaugh, Ivar Jacobson (1999). "El Lenguaje Unificado de Modelado". ). Ed. Addison-Wesley.

**[Cabrera et al., 2000]** Cabrera, J., Hernández, D., Domínguez, A. C., Castrillón, M., Lorenzo, J., Isern, J., Guerra, C., Pérez, I., Falcón, A., Hernández, M., and Méndez, J. (2000). Experiences with a museum robot. Workshop on Edutainment Robots 2000, Institute for Autonomous Intelligent Systems, German National Research Center for Information Technology, Bonn, 27-28 September, Germany.

**[Cabrera-Gámez et al., 2000]** Cabrera-Gámez, J., Domínguez-Brito, A. C., and Hernández-Sosa, D. (2000). Coolbot: A component-oriented programming framework for robotics. Dagstuhl Seminar 00421, Modelling of Sensor-Based Intelligent Robot Systems, To appear in Springer Lecture Notes in Computer Science in summer 2001. Centro de Tecnología de los Sistemas y de la Inteligencia Artificial (CeTSIA), University of Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, Campus Universitario de Tafira, 35017 Las Palmas, SPAIN.

**[cgdb]** Página oficial del proyecto *cgdb*. http://cgdb.sourceforge.net/

**[Domínguez-Brito et al., 2000a]** Domínguez-Brito, A. C., Andersson, M., and Christensen, H. I. (2000a). A Software Architecture for Programming Robotic Systems based on the Discrete Event System Paradigm. Technical Report CVAP 244, Centre for Autonomous Systems, KTH - Royal Institute of Technology), S-100 44 Stockholm, Sweden.

**[Domínguez-Brito et al., 2002]** Domínguez-Brito, A. C., Hernández-Sosa, D., and Cabrera-Gámez, J. (2002). Programming with Components in Robotics. Waf 2002 - III Workshop Hispano-Luso en Agentes Físicos, Murcia.

**[Domínguez-Brito et al., 2000b]** Domíınguez-Brito, A. C., Hernández-Tejera, F. M., and Cabrera-Gámez, J. (2000b). A Control Architecture for Active Vision Systems. Frontiers in Artificial Intelligence and Applications: Pattern Recognition and Applications, M.I. Torres and A. Sanfeliu (eds.), pp. 144-153, IOS Press, Amsterdam.

**[Domínguez-Brito, 2003]** Domíınguez-Brito, A. C. (2003). CoolBOT: a Component-Oriented Programming Framework for Robotics. Tesis Doctoral. Universidad de Las Palmas de Gran Canaria.

**[flex]** Página oficial del proyecto Flex. http://flex.sourceforge.net/

**[Gamma et al., 1995]** Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). Patrones de Diseño.

**[gcc]** Página oficial del proyecto GCC. http://gcc.gnu.org/

**[kdevelop]** Página oficial del proyecto KDevelop. http://www.kdevelop.org/

**[Larman, 1999]** Larman Craig (1999). UML y patrones: introducción al análisis y diseño orientado a objetos. Editorial Prentice Hall.

**[Levine, 1995]** John Levine (1995). Lex & yacc. Editorial O'Reilly and Associates (2nd. ed.).

**[make]** Página oficial del proyecto Make. http://www.gnu.org/software/make/

**[meld]** Página oficial del proyecto Meld. http://meld.sourceforge.net/

**[Pérez-Aguiar 1998]** Miguel Ángel Pérez Aguiar (1998). Traductores e Intérpretes. Escuela Universitaria de Informática. Universidad de Las Palmas de Gran Canaria.

**[Pressman, 2006]** Roger S. Pressman (2006). Ingeniería del software: un enfoque práctico. Editorial McGraw-Hill.

**[Santana-Jorge, 2007]** Francisco J. Santana Jorge (2007). Compilador/Generador de Esqueletos C++ para componentes CoolBOT. Página oficial del proyecto http://gias720.dis.ulpgc.es/Gias/pfcs/coolbot-compiler/

**[Steenstrup et al., 1983]** Steenstrup, M., Arbib, M. A., and Manes, E. G. (1983). Port automata and the algebra of concurrent processes. Journal of Computer and System Sciences, 27:29–50.

**[Stroustrup, 2000]** Stroustrup, B. (2000). The C++ Programming Language. Addison Wesley, Special Edition edition.

**[wikipedia]** Enciclopedia de contenido libre. http://es.wikipedia.org/wiki/Portada