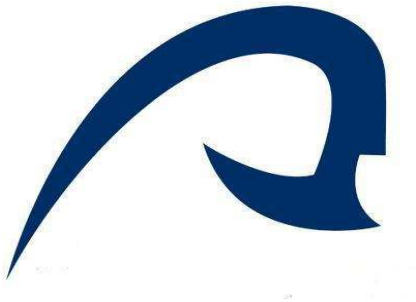
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

FACULTAD DE INFORMÁTICA



PROYECTO DE FIN DE CARRERA

COMPILADOR/GENERADOR DE ESQUELETOS C++ PARA COMPONENTES CoolBOT

FRANCISCO JESÚS SANTANA JORGE
Las Palmas de Gran Canaria, Noviembre de 2007

Proyecto fin de carrera de la Facultad de Informática de la Universidad de Las Palmas de Gran Canaria presentado por el alumno:

FRANCISCO JESÚS SANTANA JORGE

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

Tutor: Antonio Carlos Domínguez Brito.

CoTutor: Jorge Cabrera Gámez.

A todos aquellos que han creído en mí.

AGRADECIMIENTOS.

Este Proyecto de Fin de Carrera no se habría podido llevar a cabo sin la generosa colaboración de varias personas a quienes expreso mi agradecimiento.

Deseo expresar mi agradecimiento a mi tutor principal D. Antonio Carlos Domínguez Brito, por su predisposición permanente e incondicional en aclarar mis dudas y por sus substanciales sugerencias durante la elaboración del presente Proyecto.

También deseo expresar mi agradecimiento a mis compañeros de la Facultad de Informática de la Universidad de Las Palmas de Gran Canaria y a todas aquellas personas que han participado directa e indirectamente en este proyecto por sus consejos y por su colaboración.

Contenido

Índice de Figuras

Índice de Tablas

Índice de Fragmentos de Código

Capítulo 1

Introducción.

Se requiere de bastante tiempo para generar la estructura básica de un componente CoolBOT. Un programador (usuario que desarrolla el componente) debe concentrar toda su atención en un proceso repetitivo y tedioso con tendencia a cometer errores y en el que se tarda, partiendo de un componente previamente desarrollado, aproximadamente más de dos horas.

En el trabajo presentado en este documento, estudiaremos las dificultades que se presentan en la fase de generación de esqueletos C++ de un componente CoolBOT y platearemos una solución que ayude a reducir el esfuerzo requerido en esta fase.

En este primer capítulo, vamos a ver como se desarrolla en el momento actual un componente CoolBOT. Estudiaremos los inconvenientes que presenta la fase de generación de esqueletos C++ y que medidas se han tomado por el momento para solucionarlos. Una vez planteado el problema, 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.

1.1. Estado actual del tema.

La plataforma CoolBOT [Domínguez-Brito, 2003], que estudiaremos en el capítulo 2, proporciona una serie de recursos para el desarrollo de componentes. En la actualidad para desarrollar un componente se sigue la metodología que se describe en la Figura 1. Esta metodología se encuentra dividía en una serie de fases que a continuación pasamos a describir:

- **Fase 1: Diseño y definición del componente**. El programador, en base a un problema real que pretende resolver, definirá un boceto en donde describirá los distintos elementos que requiere el componente para realizar una tarea.
- **Fase 2: Generación del esqueleto C++**. Una vez diseñado y definido el componente, se genera un esqueleto C++ para nuestro componente.

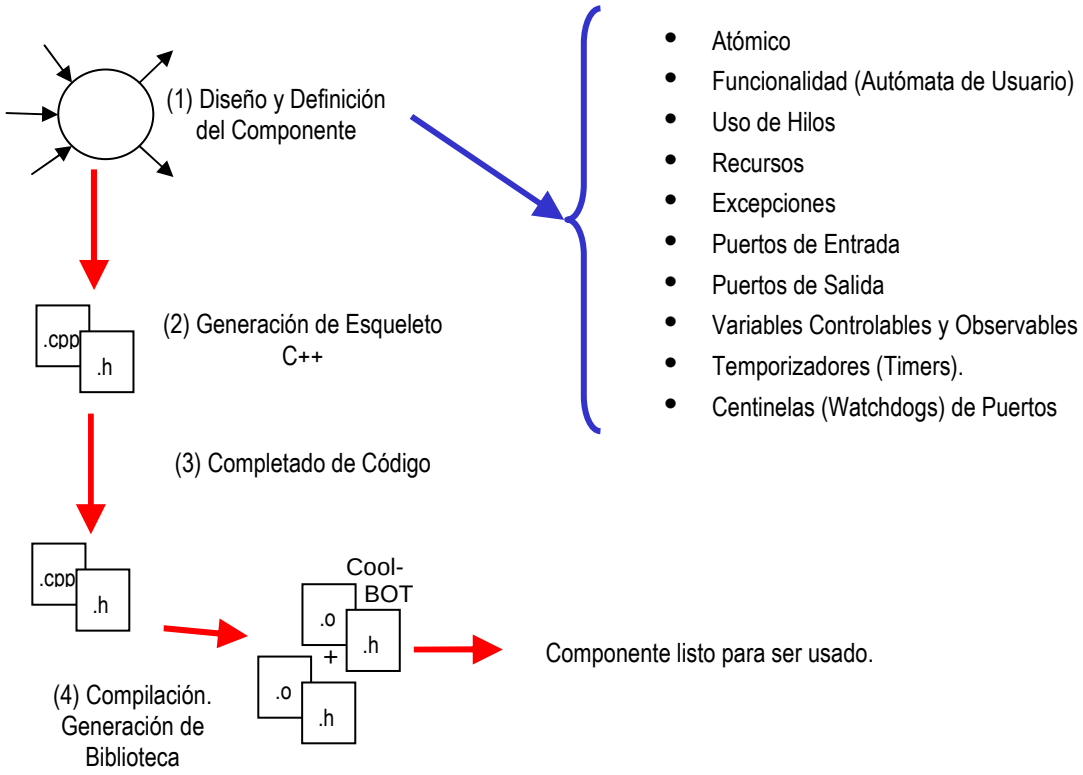


Figura 1: Metodología de desarrollo de un componente CoolBot.

En la elaboración del esqueleto C++, si no tenemos suficiente experiencia en la plataforma CoolBOT, el proceso de desarrollo es lento y tedioso ya que se ha de tener en cuenta una serie de detalles de implementación (ver apartado 2.8).

Para facilitar la tarea al programador se ha empleado la técnica de ***usar moldes***. Esta técnica consiste en coger un componente (que ya se encuentra desarrollado y se acerque en lo posible al diseño final que tendrá el componente) y utilizarlo como punto de partida en la elaboración del esqueleto C++. De este modo, disponemos al instante un esqueleto C++ operativo que solo tendremos que adaptar a las necesidades de nuestro componente.

Aunque se trata de una solución práctica, el proceso de adaptación no es fácil. Un esqueleto puede tener fácilmente más de 1900 líneas de código, que hemos de recorrer línea a línea para eliminar, modificar o añadir elementos que requiere el componente que deseamos desarrollar. A parte de esto, se exige de la atención del programador poder adaptar el *molde* sin que se produzcan errores ni aparezca código *espúreo* (líneas de código que no son necesarias en el componente).

Hasta el desarrollo del presente trabajo, este es el procedimiento que se emplea para generar los esqueletos.

- **Fase 3: Completado del código**. En esta fase se parte del esqueleto del componente proveniente de la fase anterior. En esta fase es donde se completa la codificación del componente mediante el "rellenado" del esqueleto, con ello nos referimos, al proceso por el cual el programador va completando las distintas partes del esqueleto que definirán la funcionalidad del componente final.
- **Fase 4: Compilación y generación de Biblioteca**. Una vez completado el componente, éste se compila para obtener una biblioteca del mismo. CoolBOT exige que todo componente sea empaquetado en una determinada estructura de directorios y ficheros, para que este pueda ser compilado. Esta estructura se muestra a continuación en la Figura 2.

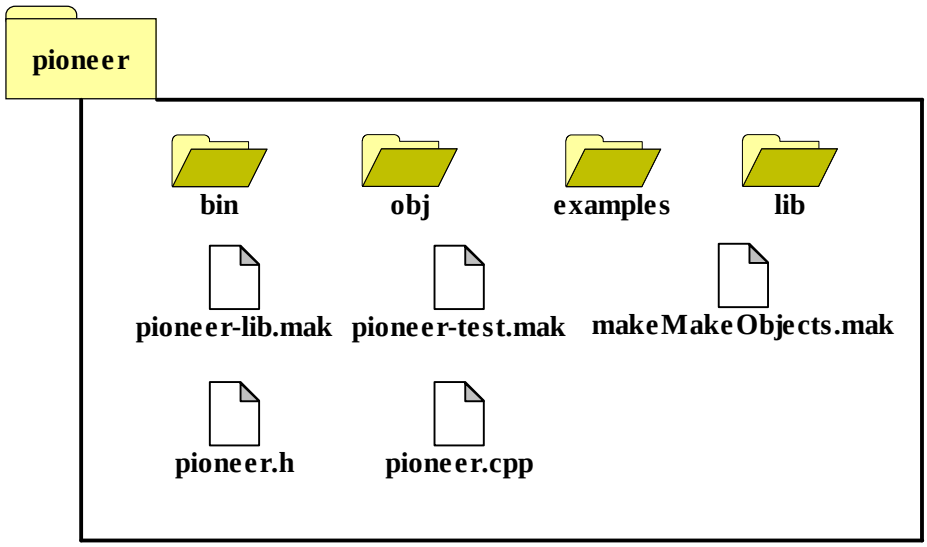


Figura 2: Estructura de directorios y ficheros de un componente denominado *Pioneer*.

Tal y como se aprecia en la Figura 2, la estructura de directorios parte de un directorio raíz en donde se encuentra almacenado el fichero de cabecera y el fichero fuente con la implementación del componente que vamos a compilar. De este directorio cuelgan los siguientes subdirectorios:

1. **Subdirectorio *"bin"***. Contiene los binarios correspondientes a la compilación de los ficheros de prueba situados en el subdirectorio "***examples***". A través de estos binarios podemos comprobar que el componente se ha implementado correctamente y cumple todos los objetivos.
2. **Sudirectorio *"examples"***. Contiene ficheros fuente C++ que testean la implementación del componente.

3. **Subdirectorio *"lib"***. El resultado de la compilación del componente es una librería la cual se almacena en este directorio. Esta librería se emplea para testear y para integrar el componente. Cuando queremos integrar el componente en cualquier aplicación, esta es la librería que es preciso enlazar.
4. **Subdirectorio *"obj"***. Este directorio contiene los ficheros objetos necesarios para generar la compilación del componente y de los ficheros de prueba.

También vemos en la Figura 2, una serie de ficheros que pasamos a describir a continuación:

1. **Ficheros *MakeFile***. Se trata de tres ficheros que se encargan de una determinada actividad en la compilación del componente. Tendremos:
   - Un fichero que se encargará de generar toda la lógica necesaria para realizar la compilación.
   - Un fichero que se encargará de compilar el componente en formato librería.
   - Un fichero que generará todos los binarios correspondientes a los programas de prueba implementados en el subdirectorio "***examples***".
2. **Archivos de cabecera y archivos fuente**. Archivos con las extensiones .h y .cpp que contienen la implementación del componente.

Al emplearse la técnica de ***usar moldes*** (ver fase 2 de este apartado), automáticamente tendremos esta estructura de directorios y ficheros. El programador se limitará a renombrar y a modificar los ficheros que contiene los directorios.

Aparte de esta estructura de directorios y ficheros, se necesita además crear una variable de entorno y modificar otra para integrar el componente en la plataforma CoolBOT. 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.

Como hemos visto en la fase de generación de esqueletos, la solución de emplear componentes previamente desarrollados como *moldes* no es la ideal porque, aunque ganamos en tiempo de desarrollo, sigue siendo un proceso tedioso y propenso al error.

Estudiando en profundidad los esqueletos C++ y partiendo de la experiencia conseguida con el compilador ***desc*** para componentes Des [Domínguez-Brito et al., 2000b], se observo que esta fase se puede automatizar completamente mediante la construcción de un compilador. Así pues se desarrolló un lenguaje descriptivo a través del cual se modela el componente y a partir de él, usando un compilador que procesaba dicho lenguaje, se obtenía uno esqueletos Java para cada componente Des.

En base a esto, se puede automatizar la generación de esqueletos C++ de componentes CoolBOT, desarrollando un compilador que a partir de un lenguaje descriptivo genere automáticamente los esqueletos C++. De este modo, lo que antes requería de horas, ahora se realiza "instantáneamente" y así el programador pueda empezar directamente con la fase de completado de código.

1.2. Objetivos.

El objetivo de todo Proyecto de Fin de Carrera es que el alumno ponga en práctica el conocimiento adquirido en las distintas disciplinas impartidas en la titulación, desarrollando por su propia mano todas las fases de un proyecto para que se exponga por primera vez a los problemas de un proyecto real. Por esta razón, podemos distinguir pues dos objetivos en este proyecto: un objetivo de carácter académico y otro de carácter práctico.

En cuanto al objetivo de carácter académico, se pretende hacer un uso práctico de los conocimientos adquiridos en las diversas áreas impartidas a lo largo de la titulación. En particular, en este proyecto se hace uso de las siguientes disciplinas:

- Teoría de autómatas y lenguajes formales.
- Metodología y tecnología de la programación.
- Estructura de datos y de la información.
- Ingeniería del software.
- Inglés técnico.
- Procesadores de lenguaje.

Respecto al objetivo de carácter práctico, se pretende dar solución a un problema real. En este caso se pretende eliminar las dificultades que hemos visto en el anterior apartado con respecto a la generación del esqueleto C++ con el fin de disminuir el tiempo de desarrollo de un componente CoolBOT. Tal y como hemos mencionado en el anterior apartado, este problema se va solucionar con el desarrollo de un compilador que a través de un simple lenguaje descriptivo obtenga automáticamente el esqueleto C++ correspondiente.

Este compilador, a su vez, debe cumplir con una serie de requisitos que pasamos a comentar:

- El lenguaje descriptivo a utilizar debe contemplar todas las características de un componente CoolBOT así como ser simple de usar e intuitivo.
- El compilador soportará comandos que suministren información al usuario sobre la compilación que se esta realizando con fines de depuración.
- El compilador deberá de ser capaz de generar automáticamente toda la estructura necesaria de ficheros y directorios que exige la plataforma CoolBOT de forma que el usuario, una vez rellenado el fichero descriptivo y compilado, pueda compilar el componente sobre la plataforma.
- El compilador, aparte de los esqueletos C++, generará esqueletos XML y a partir de estos esqueletos XML se podrán generar los correspondientes esqueletos C++.

Este último requisito es para usos futuros ya que existe otro Proyecto de Fin de Carrera que se encuentra en desarrollo que haría uso de este requisito.

1.3. Contenido de este documento.

Este documento ha sido organizado en 8 capítulos y en 4 apéndices:

- **Capítulo 1: Introducción**. Se trata de este capítulo de introducción.
- **Capítulo 2: Estudio de la plataforma CoolBOT**. En este capítulo estudiaremos los puntos más revelantes de la plataforma CoolBOT así como los esqueletos C++ que poseen sus componentes.
- **Capítulo 3: Metodología, recursos y plan de trabajo.** Se trata de un capítulo donde comentaremos que metodologías y recursos hemos empleado en la elaboración de este Proyecto. También veremos el plan de trabajo que se siguió.

- **Capítulo 4: Análisis**. En este capítulo vamos a analizar el problema planteado y a capturar los requisitos necesarios para poder diseñar la solución que cumplirá los objetivos marcados en la sección 1.2.
- **Capítulo 5: Diseño**. En base a los requisitos planteados en el anterior capítulo, plantearemos la estructura software que emplearemos en este Proyecto.
- **Capítulo 6: Implementación**. Se trata de un capítulo, donde se comentará los aspectos más revelantes de la etapa de implementación.
- **Capítulo 7: Prueba y resultados**. Disponiendo de una versión operativa del software, pasamos a evaluar la solución obtenida y a comparar resultados.
- **Capítulo 8: Conclusiones y trabajo futuro**. En este último capítulo se indicarán las conclusiones obtenidas durante el desarrollo de este proyecto. Asimismo se hablarán de que otras posibles mejoras futuras se puedan plantear.
- **Apéndice A: Manual de usuario**. En este apéndice se explicará al usuario común como usar el producto resultante de este proyecto.
- **Apéndice B: Diagramas sintácticos**. Observaremos los diagramas sintácticos del lenguaje descriptivo que se ha planteado en este proyecto.
- **Apéndice C: Notación EBNF**. Observaremos en notación EBNF el lenguaje descriptivo que se ha planteado en este proyecto.
- **Apéndice D: Componente CoolBOT completo**. En este último apéndice veremos un ejemplo completo del esqueleto C++ de un componente CoolBOT generado por el compilador.

Capítulo 2
Estudio de la plataforma CoolBOT.

A continuación pasamos a realizar un estudio del marco de programación CoolBOT. Sólo nos centraremos en las partes de dicha plataforma que son necesarias para el presente trabajo. Para un conocimiento más profundo de esta plataforma, se recomienda leer [Domínguez-Brito et al., 2003].

2.1. Orígenes de CoolBOT.

CoolBOT fue principalmente originado debido a consideraciones muy prácticas. A principios del año 2000, se observó la falta de una metodología sistemática de desarrollo de sistemas robóticos [Hernández et al., 1999] [Cabrera Gámez et al., 2000]. Tampoco se había definido una arquitectura software válida para estos sistemas.

Durante el desarrollo de varios sistemas robóticos por parte del grupo GIAS[1] (Grupo de Inteligencia Artificial y Sistemas) se llegó a la conclusión que era necesario diseñar e implementar algún tipo de infraestructura software común que disminuyera los costes de desarrollo e integración. Esta infraestructura tenía que ser lo suficientemente genérica como para soportar cualquier arquitectura o esquema de control ideado para los proyectos en los que el grupo se encontraba involucrado en aquel momento. Al mismo tiempo, debía permitir integrar software de manera fácil.

Partiendo de estos requisitos, se diseño e implementó un marco software basado en componentes denominado CAV [Domínguez-Brito et al., 2000a]. CAV fue una herramienta que permitió modelar software de control como redes de agentes software interconectados, y proporcionaba mecanismos de intercomunicación entre agentes, ya fueran locales o remotos. CAV carecía de muchos recursos y primitivas que se consideraron también necesarias, por ejemplo, un conjunto más rico de mecanismos de intercomunicación, y un suporte para programación multihilo (multithreading) más

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

sistemática y menos propensa a errores. Pero sobre todo carecía de mecanismos que facilitaran la integración de software, que continuaba siendo un importante problema a resolver en los diferentes proyectos.

Trabajos y experiencias posteriores utilizando CAV llevaron al diseño y desarrollo de CoolBOT [Domínguez-Brito, 2003], pasando por diferentes fase que pueden seguirse en [Domínguez-Brito et al., 2000b], [Cabrera-Gámez et al., 2001] y [Domínguez-Brito et al., 2002]. CoolBOT [Domínguez-Brito et al., 2004a] es un marco de programación C++ orientado a componentes donde el software que controla un sistema se ve como una red dinámica de unidades de ejecución interconectadas por medio de caminos de datos. Cada una de estas unidades de ejecución es un componente software, modelado como un autómata de puertos [Steenstrup et al., 1983] [Domínguez-Brito et al., 2000b] [Stewart et al., 1997], que proporciona una funcionalidad dada, oculta tras un interfaz externo de puertos de entrada y salida que especifica claramente los datos que el componente consume, y cuáles produce. Cualquier componente, una vez es definido, construido y probado, puede instanciarse e integrarse tantas veces como se necesite en otros sistemas. CoolBOT proporciona la infraestructura necesaria para soportar este concepto de componente software, así como para que se intercomuniquen entre ellos mediante conexiones de puertos puedan establecerse y desestablecerse dinámicamente.

2.2. Principios básicos de diseño de la plataforma CoolBOT.

La plataforma o framework CoolBOT plantea una infraestructura software que permita programar sistemas robóticos mediante ensamblaje e integración de componentes a modo de puzzle software. Provee un entorno potente donde es posible sintetizar diferentes arquitecturas usando el mismo lenguaje de especificación.

A continuación se enumeran los principios básicos de la plataforma:

- *Orientado a Componentes*. CoolBOT se concibe como una programación orientada a componentes, que se vale de un lenguaje de especificación de manipulación componentes como bloques de construcción con el fin de definir funcionalmente un sistema robótico completo mediante la integración de componentes. Esta aproximación no sólo fuerza la modularidad y la disciplina sino también la integridad.

- *Uniformidad entre Componentes.* Una aproximación basada en componentes claramente demanda cierto nivel de uniformidad entre componentes. Dentro de CoolBOT esta uniformidad se manifiesto en dos importantes aspectos:
    1. *Autómata de Puertos.* Se define un interfaz uniforme para todos los componentes basado en el concepto de Autómatas de Puertos [Steenstrup et al., 1983] [Stewart et al., 1997] [Domínguez-Brito et al., 2000a] que establece una clara distinción entre la funcionalidad interna de una entidad activa (el autómata) y su interfaz externo, los puertos de entrada y salida.
    2. *Puertos por defecto*. Todo componente debe ser observable y controlable en cualquier instante desde el exterior del propio componente. Para ello se establece una interfaz y estructura de control uniforme a todos los componentes para que todo componente facilita su observabilidad y su controlabilidad. Esta interfaz y estructura de control son los denominados puertos por defecto que se encuentra presentes en todos los componentes:
        - *El puerto de control c*. A través de él un componente puede ser externamente controlado utilizando las llamadas *variables controlables*.
        - *El puerto de monitorización m*. Las variables exportadas por el componente, llamadas *variables observables*, son monitorizadas mediante este puerto.

    La Figura 3 muestra el aspecto externo de un componente donde el componente mismo es representado por un círculo, los puertos de entrada i_i por las flechas orientadas hacía el círculo, y los puertos de salida o_i por flechas orientadas hacía afuera del círculo. Como se muestra en la figura, el interfaz externo mantiene ocultas las interioridades del componente. En la Figura 5 se observa un ejemplo del aspecto interno de un componente, concretamente el autómata que lo modela, donde los círculos son estados del autómata, y las flechas, las transiciones entre estados. Estas transiciones son activadas por eventos (e_i), causados, bien por los datos que entran a través de algún puerto de entrada, bien por alguna condición interna al componente, o bien por una combinación de datos entrantes y/o condiciones internas. Los círculos dobles representan estados finales del autómata. Al modelar la funcionalidad interna de un componente como un autómata se provee de un medio para hacer los componentes observables y controlables. La Figura 4

muestra parte de este interfaz externo uniforme, los puertos por defecto, presentes en todo componente.

- *Robustez y Controlabilidad*. Un sistema robótico orientado a componentes será robusto y controlable si sus componentes son también robustos y controlables. Un componente se considerará robusto cuando:
    1. Sea capaz de observar su propio rendimiento, adaptándose a condiciones de operación cambiantes, e implementando sus propios mecanismos de adaptación y recuperación para tratar todo error que pueda ser detectado internamente (dentro el componente, robustez local).
    2. Cualquier error detectado por un componente que no pueda ser tratado y/o resuelto por sus propios medios, debería ser notificado utilizando algún mecanismo estándar a través de su interfaz externo (robustez externa), llevando al componente a un estado de inactividad total (idle) en el que se espera por una intervención externa, que consistirá, o bien, en reiniciar el componente, o en abortarlo. Las comunicaciones enviadas y recibidas por un componente cuando esta tratando/resolviendo excepciones deberían ser comunes a todos los componentes.

    Adicionalmente, se considerará a un componente controlable cuando el pueda ser llevado bajo supervisión externa a través de su interfaz - por medio de un supervisor o un controlador - a lo largo de una trayectoria de control establecida. En orden a conseguir dicha controlabilidad externa, los componentes serán modelados como autómatas cuyos estados serán forzados por un supervisor externo, y compartiendo todos ellos la misma estructura en su autómata de control.
- *Modularidad y Jeraquía*. La arquitectura de un sistema robótico se definirá en CoolBOT utilizando componentes como unidades funcionales elementales. Como en casi cualquier marco basado en componentes, habrá unidades atómicas y compuestas. Un componente atómico será indivisible, es decir, uno que está formado/compuesto por otros componentes. Un componente compuesto será un componente que incluye en su definición a otros componentes, atómicos o no, y provee un supervisor para su observación y control. Con esta visión, un sistema completo no es nada más que un componente compuesto único, que a su vez incluye otros componentes, que análogamente incluyen otros, y así hasta que esta cadena de descomposiciones finaliza en algún componente atómico. Así pues, un

sistema completo puede verse como una jerarquía de componentes desde un punto de vista de coordinación y control.

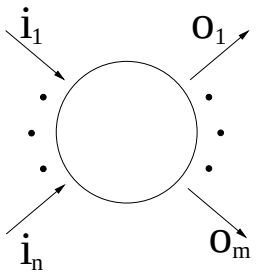


Figura 3: Aspecto o "vista" externa de un componente.

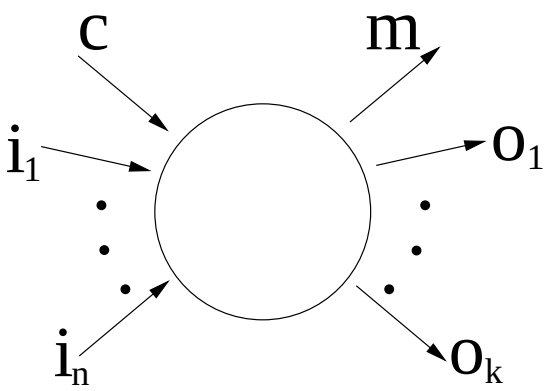


Figura 4: Los puertos por defecto. El puerto de control c y el puerto de monitorización m.

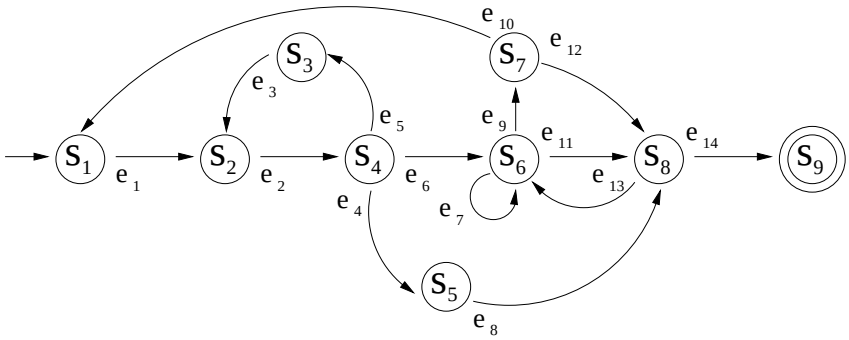


Figura 5: Aspecto o "vista" interna de un componente.

- *Distribuido*. La distribución de componentes sobre un entorno de computación distribuida es una necesidad fundamental en muchos sistemas de control. CoolBOT debería gestionar las comunicaciones entre componentes situados en la misma y/o en diferentes máquinas de forma que para un usuario de CoolBOT parecería que se realizaran exactamente de la misma forma.
- *Reutilización*. Los componentes son unidades que mantienen sus interioridades ocultas detrás de un interfaz uniforme. Una vez ellos han sido definidos, implementados y probados, podrían utilizarse como componentes integrantes de otros componentes o sistemas mayores. Los modernos sistemas robóticos están llegando a convertirse en sistemas realmente complejos, y muy pocos grupos de investigación tienen los recursos humanos necesarios para construir sistemas desde cero. Los diseño orientado a componentes representan una forma apropiada de aliviar esta situación. En nuestra opinión, la investigación en robótica podría beneficiarse enormemente de la posibilidad de intercambiar componentes entre

laboratorios como un medio de validación cruzada de los resultados de investigación.

- *Completitud y Expresividad*. El modelo de computación subyacente en CoolBOT debería ser válido para construir arquitecturas muy diferentes para sistemas robóticos y ser lo suficientemente expresivo como para tratar la concurrencia, el paralelismo, la distribución y compartición de recursos, la respuesta en tiempo real, la existencia de múltiples y simultáneos bucles de control y de múltiples objetivos a satisfacer, de una forma estable y sistemática.

2.3. El Autómata por Defecto.

Todos los componentes internamente son modelados usando un mismo autómata de estados que denominaremos **Autómata por Defecto** y que contiene todos los posibles caminos de control para un componente. En la Figura 6 se muestra este autómata.

Se dice que el Autómata por Defecto es "controlable" porque el puede ser llevado externamente mediante su puerto de control *c* a cualquiera de los estados controlables del autómata: *ready*, *running*, *suspended* y *dead*, en un tiempo finito. Los restantes estados son alcanzados sólo internamente, y desde ellos puede siempre forzarse una transición a uno de los estados controlables. El estado *running*, el estado marcado con un círculo discontinuo en la Figura 6, representa el estado o conjunto de estados que estructura la funcionalidad específica del componente. Este autómata, denominado **Autómata de Usuario**, varía en cada componente y debe ser definido por el desarrollador/usuario cuando el componente es desarrollado.

Cuando un componente es instanciado, es llevado al estado *starting*, donde el componente se provee de los recursos necesarios para su funcionamiento, y realiza su inicialización. Cualquier error en la provisión de recursos provoca un nuevo intento de solicitud de los mismos, hasta que un número máximo de intentos se ha realizado. En este punto, se considera fracasada la inicialización del componente y se transita al estado *starting error*, donde se debe esperar por intervención externa a través del puerto de control, antes de saltar al estado *dead* donde el componente es destruido.

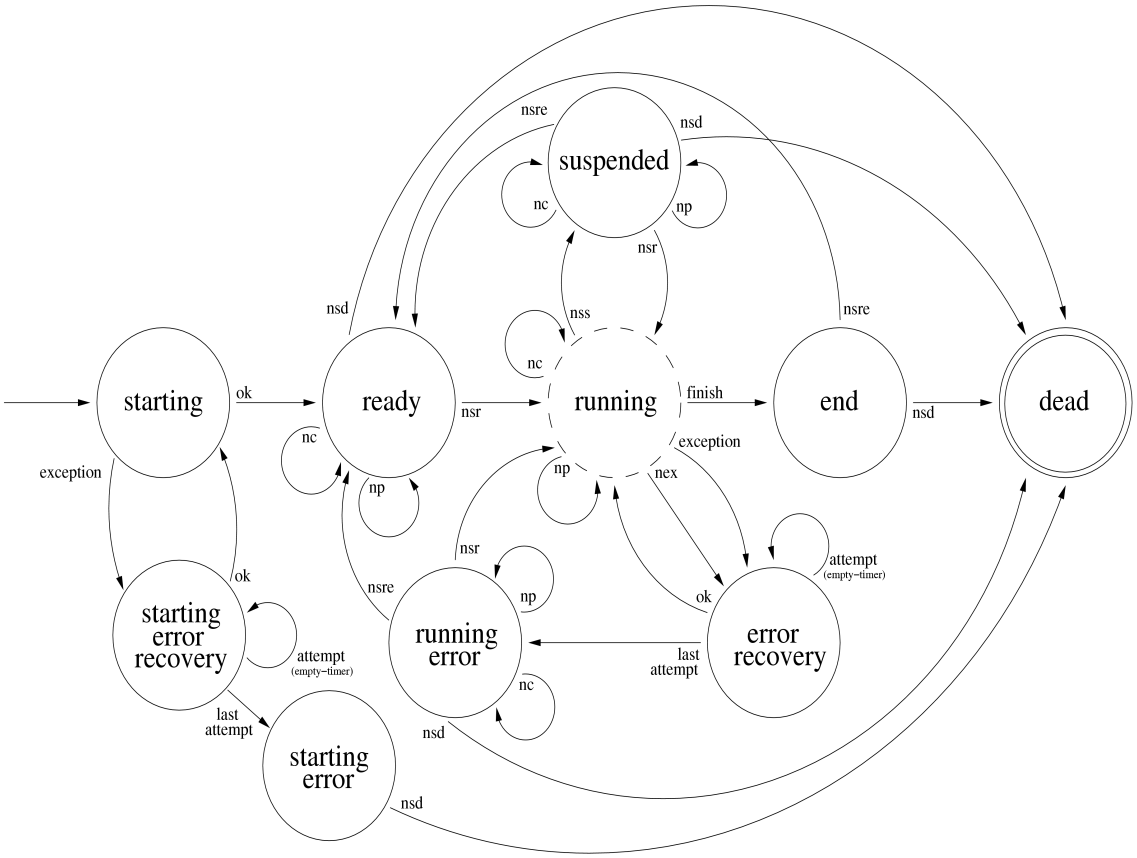


Figura 6: Autómata por Defecto.

En caso de que la inicialización se realice adecuadamente, el componente es llevado al estado *ready*, ahí el componente espera inactivo (idle), por su primera ejecución de tarea entrando en el Autómata de Usuario a través del estado *running*, o bien por su destrucción si es conducido al estado *dead*. El estado *running* es la parte del autómata (el Autómata de Usuario) definido por el desarrollador/usuario que desarrolla el componente y le dota con su funcionalidad específica. Este autómata tiene sus propios estados y transiciones, pero el autómata por defecto le impone algunos requerimientos:

1. Desde cualquiera de los estados del Autómata de Usuario el componente puede ser conducido al estado *suspended* del Autómata por Defecto. Esto implica que el componente debería ser externamente interrumpible en cualquier estado del Autómata de Usuario, con una latencia que dependerá de su diseño interno. El componente debe salvaguardar su estado interno en caso de continuar la ejecución de la tarea una vez interrumpido. De *suspended* el componente puede también ser reinicializado (reset), es decir, conducido al estado *ready*, o bien, destruido llevándolo al estado *dead.*

2. Algunos estados, proporcionados por el Autómata por Defecto para indicar estados generales en todo componente, deben ser accesibles desde los estados del Autómata de Usuario:
   - *error recovery***.** A este estado se llega cuando se detecta un error durante la ejecución de la tarea, y se ha concebido como un estado para la recuperación de un error sin cancelar la ejecución de la tarea actual. En este estado, un procedimiento de recuperación de error puede ser intentado varias veces, hasta que se han realizado un máximo número de intentos sin éxito. En este caso, el autómata pasa al estado *running error*. Si el error es resuelto en cualquiera de los intentos de recuperación, el autómata continua allá donde la ejecución de la tarea fue interrumpida previamente. Ello implica que el estado interno del componente debe de salvaguardarse también durante la recuperación de errores.
   - *running error***.** Un componente transita a este estado cuando un componente no ha podido recuperarse de un error por si mismo. Sólo la intervención externa a través del puerto de control puede llevar de nuevo al componente al estado *ready* para comenzar una nueva ejecución de tarea, o bien al estado *dead* y a la destrucción de esta instancia del componente.
   - *end***.** Si un componente ha finalizado su tarea, entonces el pasa directamente al estado *end*. Desde este estado el componente puede ser destruido, o puede comenzar una nueva ejecución de tarea si es llevado de nuevo al estado *ready*.

2.4. 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 deberían exportarse externamente. Cualquier cambio en un variable observable dentro de un componente es publicada para su observación externa a través del puerto de monitorización del componente. CoolBOT dota a todo componente de varias variables observables por defecto que podemos ver en la Tabla 1.

- *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 del componente y las que dota CoolBOT por defecto a todo componente que las podemos ver en la Tabla 1.

VARIABLES OBSERVABLES POR DEFECTO	
Nombre	**Breve descripción**
state	Indica el estado del autómata por defecto donde el componente se encuentra en cada momento.
priority	Esta variable es usada para publicar la prioridad con la que el componente se esta ejecutando.
config	Pide un cambio supervisado de configuración, o confirma órdenes de configuración. Actualmente, es usado exclusivamente en los componentes compuestos.
result	Cuando una tarea ha sido finalizada por un componente, está variable indica los resultados conseguidos.
error description	Cuando el componente entra en los estados *starting error* o *running error*, se proporciona mediante esta variable información sobre la situación de error

VARIABLES CONTROLABLES POR DEFECTO	
Nombre	**Breve descripción**
new state	Se utiliza para llevar al componente a uno de los estados controlables del autómata por defecto, y se modifica a través del puerto de control del componente.
new priority	Esta variable es usada para especificar la prioridad de ejecución que deseamos que tenga el componente.
new config	A través de esta variable la configuración del componente puede ser modificada y actualizada durante la ejecución del componente.
new exception	Esta variable permite inducir excepciones producidas desde el exterior.

Tabla 1: Variables observables y variables controlables por defecto en CoolBOT**.**

2.5. Puertos de Entrada y de Salida.

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 las conexiones de puertos 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 paquete* y cada puerto de entrada o de salida puede aceptar solamente un conjunto especifico 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. En la Figura 7 podemos apreciar todas las posibles conexiones de puertos que se pueden realizar, así como la cardinalidad que posee ese tipo de conexión.

Adicionalmente, los puertos de entrada y de salida pueden ser públicos o privados:

- **Públicos**. Los puertos de entrada y de salida son accesibles desde el exterior del componente. Otros componentes pueden usar libremente estos puertos para establecer nuevas conexiones de puertos.
- **Privados**. Los puertos de entrada y de salida no son accesibles desde el exterior del componente. Estos puertos se encuentran ocultados en la interfaz interior del componente y tiene tres usos principales:
    1. Para soportar centinelas (watchdogs) y temporizadores (timers). Los centinelas los emplea CoolBOT para monitorizar determinados puertos con el fin de controlar determinadas situaciones anormales en las cuales no se reciben paquetes de puertos. Los temporizadores se emplean para establecer periodos de tiempo en donde se pueda observar o controlar una determina situación.
    2. Para controlar, monitorizar e intercomunicar el interior de un componente dentro de un componente compuesto.
    3. Para controlar, monitorizar e intercomunicar puertos asociados a hilos de puertos en el interior de un componente.
    4. Para realizar transiciones vacías entre los estados tanto del autómata.

A continuación en la Tabla 2 podemos ver los *tipos de puertos* que podemos definir en función de la clase de puerto y en la Tabla 3 los paquetes de puertos que proporciona la plataforma por defecto.

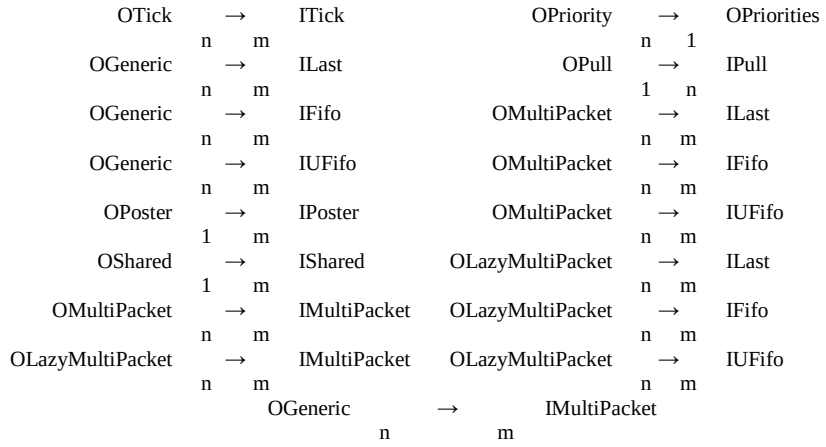


Figura 7: Conexiones de puertos (n,m Є N; n,m ≥ 1).

TIPOS DE PUERTOS	
Clase de puerto	**Tipo**
Entrada	***ITick***, ***IPoster***, ***IMultipacket***, ***IPull***, ***ILast***, ***IFifo***, ***IShared***, ***IUFifo*** y ***IPriorities***.
Salida	***OTick***, ***OPoster***, ***OMultipacket***, ***OPull***, ***OGeneric***, ***OShared***, ***OPriority*** y ***OLazymultipacket.***

Tabla 2: Tipos de puertos permitidos en CoolBOT.

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 por defecto.

2.6. Manejo de Excepciones.

Los mecanismos de manejo de excepciones de CoolBOT explotan dos ideas básicas. Primero, todos los componentes, independientemente de que sean atómicos o compuestos, comparten los mismos esquemas de manejo de excepciones, comunicaciones y control. Segundo, CoolBOT se apoya en la idea de construir un sistema robusto a partir de componentes robustos. Estas ideas se plasman en las siguientes guías de diseño:

- Un componente debe incorporar la capacidad de medir su propio rendimiento. Por ejemplo, en el caso de tratarse de una tarea periódica, debe ser capaz de verificar que está respetando su frecuencia de operación. Adicionalmente, pueden asociarse temporizadores a los puertos de entrada, de forma que pueda identificarse fácilmente si un componente externo que debería estar suministrando datos de entrada no está cumpliendo con la frecuencia esperada o simplemente no está funcionando.

  La definición del componente incluye una lista de las excepciones que ese componente es capaz de detectar, junto con los *planes de contingencia* específicos, si es que existen. Actualmente en CoolBOT las excepciones se declaran utilizando el siguiente patrón simple que podemos ver en la Figura 12:

  On Error**:** ***<error_id>***
  <description>
  <handler> [<retries> <lap>]
  <on_success_handler>
  <on_failure_handler>

  Figura 8: Formato de definición de excepciones en CoolBOT.

  donde se declaran primero el número de error y su descripción; a continuación se indica el manejador asociado, que es normalmente una función interna al componente, y que opcionalmente puede ir acompañada del número de veces y la frecuencia con que debe activarse. Los dos últimos campos contienen, respectivamente, los manejadores que deberían dispararse si el mecanismo de recuperación del error ha tenido éxito o no.
- La evolución del estado del componente cuando ha ocurrido una excepción es la misma en todos los casos. Las transiciones que pueden tener lugar son las que están recogidas en el grafico del *Autómata por Defecto* (ver Figura 6)

Cuando un componente detecta un error que no puede solventar, bien porque no existe mecanismo de recuperación en este nivel, o bien porque el plan de recuperación previsto ha fracasado, lo comunica a su *supervisor* y pasa a un estado de error de ejecución (*running error* en el Autómata por Defecto) donde espera por una intervención externa para continuar o finalizar su ejecución. Un *supervisor* es el autómata que coordina y controla la funcionalidad de un componente compuesto y, como en el caso de los componentes atómicos, sigue el esquema de control definido por el Autómata por Defecto.

Los errores que llegan a un supervisor desde un componente local deben ser primero tratados en este nivel. El manejador puede ignorarlos, propagarlos a un nivel superior o tratados como se ha expuesto. Sin embargo, cuando se tratan excepciones dentro de un componente compuesto son posibles algunos mecanismos estándar de recuperación, además de la reinicialización del componente que ha fallado. Supongamos, por ejemplo, que se dispone de diversos componentes que constituyen alternativas equivalentes para el desarrollo de la misma tarea, posiblemente haciendo uso de recursos diferentes, pero ofreciendo la misma interfaz externa. Dichos componentes podrían ser usados de forma alternativa en el desempeño de la tarea con lo que una estrategia general de tratamiento de errores sería la simple sustitución de un componente por otro que proporcione la misma interfaz y funcionalidad. Una estrategia complementaria puede ser útil para evitar suspender un componente compuesto cuando un miembro de la composición pasa al estado de error de ejecución. Componentes equivalentes pueden ser declarados como redundantes y ejecutados concurrentemente o en paralelo (i.e. si los componentes redundantes se ejecutan en procesadores diferentes), de manera que si uno de ellos falla, los restantes pueden mantener al componente compuesto en ejecución.

Cuando una instancia de un componente local pasa al estado de error de ejecución, si existe un sustituto, el supervisor crea una instancia del mismo para realizar sustitución y mantener al componente compuesto corriendo, colocando a la instancia que falló en una cola de instancias pendientes de recuperación. Estas últimas son reactivadas periódicamente para comprobar si el error persiste durante un tiempo máximo, tras el cual la instancia habrá sido destruida (no ha conseguido recuperarse) o bien se ha restablecido la situación previa a la sustitución (la instancia se ha recuperado del error).

Si una instancia local en error de ejecución no puede ser sustituida, se añadirá a la cola de recuperación previamente comentada. Si el tiempo máximo de intento de recuperación es alcanzado, la instancia es destruida, provocando o no que todo el componente compuesto pase al estado de error de ejecución o no dependiendo de su funcionamiento.

Un error que precisa un tratamiento especial es aquél que se produce cuando un componente se bloquea durante la ejecución. En esta situación, no podrá atender sus puertos de monitorización y control, por lo que la única solución posible es la destrucción del componente mediante una llamada al sistema tipo *kill(),* y obviamente no es añadido a la cola de recuperación.

A continuación en la Tabla 5 podemos ver las excepciones definidas por defecto en CoolBOT:

EXCEPCIONES POR DEFECTO	
Nombre	**Breve descripción**
No memory	Indica una situación de memoria no disponible durante la creación de memoria dinámica.
Inconsistency	Indica una inconsistencia en el autómata del componente.
File Not Found	Indica una situación en donde no se encuentra un fichero que debería encontrarse.
Bad Watchdog Transition	Indica una situación en la que un centinela detecta una situación anómala en la que un puerto de entrada no esta recibiendo paquetes de puertos correctamente.

Tabla 4: Excepciones por defecto en CoolBOT**.**

2.7. Hilos de puertos.

Todos los componentes CoolBOT cuando se ejecutan se encuentran mapeados en hilos y frecuentemente cada componente posee más de un hilo. En este caso se tendría varios flujos de ejecución concurrentemente y/o en paralelo en donde cada hilo ejecutaría un bucle que estaría constantemente procesando paquetes de puertos. En la Figura 7 podemos ver esta idea.

En CoolBOT, los hilos usados en el interior del componente para su ejecución se denominan ***hilos de puertos*** (port threads), de aquí en adelante **hilos**, cuya ejecución es

guiada por los paquetes de puertos que ellos reciben a través de un conjunto de puertos de entrada sobre los cuales es responsable. De ahí, la ejecución constante de bucle de procesamiento de paquetes de puertos. Aparte de los puertos de entrada, los hilos también son responsables de un conjunto de puertos de salida y tanto el conjunto de puertos de entrada y de salida que emplean cada hilo tienen que ser disjuntos. Por último, para el control y observabilidad externa, los hilos poseen un puerto de entrada de control y un puerto de salida para la monitorización.

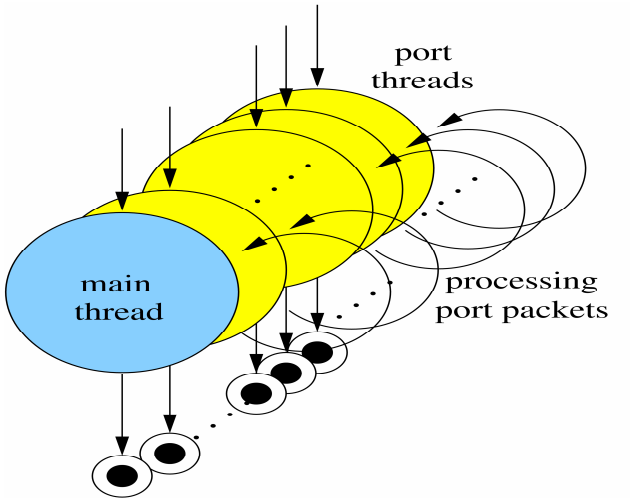


Figura 9: Múltiples hilos ejecutándose.

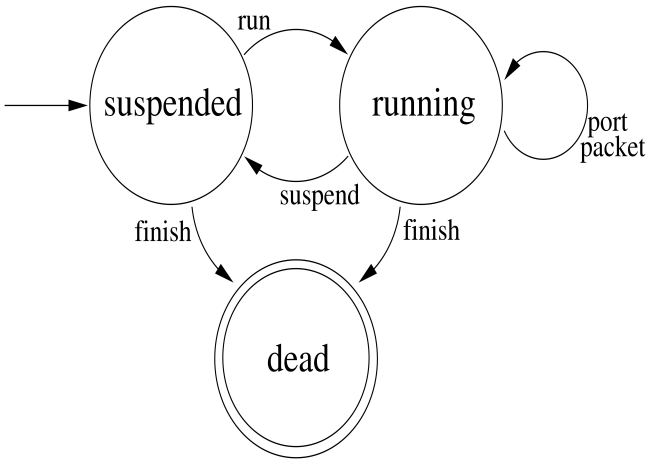


Figura 10: Autómata de ejecución de un hilo.

La ejecución de un hilo sigue el autómata mostrado en la Figura 8, el cual esta formado por tres estados:

- *supended*. Una vez que un hilo ha sido lanzado, se pone en el estado *suspended*, donde permanece ocioso hasta que se le ordene externamente o a través de su puerto de control que vaya al estado *running* o *dead.*
- *running*. En este estado, el hilo ejecuta continuamente transiciones sobre el Autómata de Usuario del componente en función de los paquetes de puertos que reciben los puertos de entrada asociados al hilo. De este estado pueden ser enviados a los estados *suspended* o *dead* mediante el puerto de entrada de control.
- *dead*. En este estado se finaliza la ejecución del hilo.

Como hemos dicho un componente CoolBOT para su ejecución puede ejecutar múltiples hilos, pero todo componente necesita de un hilo especial que se encarga de la ejecución del Autómata por Defecto, de la observabilidad y control del componente a través del puerto de entrada de control y del puerto de salida de monitorización y de mantener la consistencia de las estructuras de datos internas que conforman el estado interno del componente entero. Este hilo especial se denomina hilo ***main***.

El hilo ***main*** controla la ejecución del componente, ejecutando las transiciones de puertos, transitando de estado a estado, lanzando, suspendiendo, ejecutando y matando o destruyendo el resto de hilos que emplea el componente.

Durante el diseño e implementación de componentes multihilo es necesario indicar para cada estado que hilos se encargarán de su ejecución en el Autómata de Usuario. Durante la ejecución, el hilo ***main*** controlará y/o suspenderá cada hilo dependiendo del estado del autómata en que se encuentre en ese momento la ejecución.

2.8. Estudio del esqueleto C++ de un componente CoolBOT.

La plataforma CoolBOT emplea como lenguaje de implementación de componentes el lenguaje C++, por lo que todo componente CoolBOT se implementa en C++ en dos archivos: uno de cabecera (fichero .h) donde se realizan las definiciones del componente y otro archivo de implementación (fichero .cpp) donde se realiza la implementación de las definiciones que se hayan realizado. En la plataforma se emplea como paradigma de programación[2] la programación orientada a objetos. Luego todo componente se corresponderá a un objeto que se encuentra implementado en una clase C++.

Como ya hemos comentado, en CoolBOT se distinguen dos tipos de componentes: los atómicos y los compuestos. De aquí en adelante, nos vamos a centrar solo y exclusivamente en los **componentes atómicos** ya que es el tipo de componentes CoolBOT para los que se genera esqueleto C++ en este proyecto.

Para ver con claridad estos ficheros, vamos a dividir tanto el fichero cabecera como el fichero de implementación en una serie de partes en las cuales se comentará muy brevemente los elementos más revelantes con el fin de entender como se implementa un componente CoolBOT. En el Apéndice D podemos ver el esqueleto completo de un componente ejemplo.

Por último, hemos de comentar que junto a estos dos ficheros pueden aparecer otros que sirven de complemento a la implementación del componente (por ejemplo, los ficheros que contienen la definición de nuevos paquetes de puertos). Al tratarse de casos especiales que no siempre se dan, no lo trataremos en este apartado pero si lo tendremos en cuenta en la generación del esqueleto.

2.8.1. Esqueleto C++ correspondiente al fichero cabecera.

En el Fragmento de Código 1 podemos ver la estructura que posee este fichero. Como hemos comentado anteriormente, el fichero cabecera contiene las definiciones necesarias para implementar el componente y se encuentra dividido, de principio a fin en orden secuencial, en una serie de partes.

[2] Un paradigma de programación [Pressman, 2006] representa un enfoque particular o filosofía para la construcción de software.

```
/*
 * File: simple-component.h
 * Date: 03 November 2007
*/
#ifndef SIMPLE_COMPONENT_H
#define SIMPLE_COMPONENT_H

#include "coolbot.h"
#include "component/coolbot_packet_instances.h"

// INCLUDE USER SECTION: BEGIN

// INCLUDE USER SECTION: END

using namespace CoolBOT;

namespace SimpleComponentSpace
{
    class SimpleComponent: public Component
    {
        public:
               // ZONA DE DEFINICIÓN PÚBLICA.
               // ZONA COMÚN DE DEFINICIÓN PÚBLICA.
        private:
               // ZONA DE DEFINICIÓN PRIVADA.
               // ZONA DE DEFINICIONES ESTÁTICAS.
               // ZONA DE DEFINICIÓN DE MANEJADORES DE EXCEPCIONES Y DE CENTINELAS.
               // ZONA DE DECLARACIONES DE ATRIBUTOS Y MÉTODOS.
               // ZONA DE DEFINICIÓN COMÚN.
    };
}
namespace
{
    CoolBOT::StaticInit<SimpleComponentSpace::SimpleComponent>
                              JOIN2(staticInit,SimpleComponent);
}
#endif // SIMPLECOMPONENT_H
```

Fragmento de Código 1: Esqueleto C++ del fichero cabecera de un componente denominado *SimpleComponent*.

A continuación, pasamos a comentar cada una de las partes que se muestran en el Fragmento de Código 1:

- *Cabecera del componente*. Por normal general se inicia la definición de un componente con un comentario en donde se especifica información sobre el componente que se esta implementando. Normalmente, por convención se pone la siguiente información:
    1. Nombre del fichero (que contiene la implementación).
    2. Descripción del componente.
    3. Información sobre el autor o autores de dicho componente, incluyendo la institución donde se desarrolla el componente.
    4. Versión del componente desarrollado.
    5. Fecha de última modificación.

  No es obligatorio poner el comentario y si se pone tampoco es obligatorio poner toda esa información. Se recomienda que como mínimo se especifique el nombre del fichero y la fecha de última modificación.

A continuación aparecen las directivas *#ifndef* y *#define* para evitar que la definición de un componente se realice más de una vez.

Luego comienza la zona de *include* donde se especifica que otros ficheros se requieren para que el componente se pueda implementar. CoolBOT requiere que como mínimo se especifiquen dos ficheros a incluir. El primero nos da acceso a todos los elementos de la plataforma y el segundo nos da acceso a los paquetes de puertos predefinidos en la plataforma CoolBOT.

A esta zona le sigue una zona de declaración de espacios de nombres que emplearemos en la definición del componente. Siempre debe aparecer definido el espacio de nombres de la plataforma.

En el Fragmento de Código 2, podemos ver la definición de una cabecera de componente tal y como acabamos de mencionar.

```
/*
 * File: simple-component.h
 * Description: Componente de prueba.
 * Date: 15 October 2007
 * Author: Antonio C. Dominguez-Brito
 *         Dpto. Informatica y Sistemas
 *         Universidad de Las Palmas de Gran Canaria.
 */

#ifndef SIMPLE_COMPONENT_H
#define SIMPLE_COMPONENT_H

#include "coolbot.h"

#include "component/coolbot_packet_instances.h"

// INCLUDE USER SECTION: BEGIN

// INCLUDE USER SECTION: END

using namespace CoolBOT;
```

Fragmento de Código 2: Cabecera del componente *SimpleComponent*.

- *Zona definición de componente*. La definición del componente comienza con la declaración de un espacio de nombres propio seguido de la declaración de la clase que contendrá la definición. Tanto el espacio de nombres como la clase se construyen con el nombre del componente y la clase siempre hereda de la clase *Component*.
- *Zona de definición pública*. En esta zona se declaran puertos de entrada, puertos de salida, variables controlables, variables observables, excepciones, estados y constantes que son públicos en el componente. Todo esto, exceptuando las constantes cuya definición es opcional, siempre se tiene que definir aunque en el componente no se empleen dichas definiciones.

En el Fragmento de Código 3 podemos apreciar la zona de definición pública del componente *SimpleComponent*. En este componente sólo se ha definido un estado (el estado *Main*) del Autómata de Usuario:

```
public:
    //USER AUTOMATON STATES
    enum States
    {
      MAIN=DEFAULT_STATES,
      STATES,
      RUNNING_ENTRY_STATE=MAIN
    };
    //PUBLIC INPUT PORTS
    enum InputPorts
    {
      INPUT_PORTS=DEFAULT_INPUT_PORTS
    };
    //PUBLIC OUTPUT PORTS
    enum OutputPorts
    {
      OUTPUT_PORTS=DEFAULT_OUTPUT_PORTS
    };
    //CONTROLLABLE VARIABLES
    enum ControllableVariables
    {
      CONTROLLABLE_VARIABLES=DEFAULT_CONTROLLABLE_VARIABLES
    };
    //OBSERVABLE VARIABLES
    enum ObservableVariables
    {
      OBSERVABLE_VARIABLES=DEFAULT_OBSERVABLE_VARIABLES
    };
    //EXCEPTIONS
    enum Exceptions
    {
      EXCEPTIONS=DEFAULT_EXCEPTIONS
    };
```

Fragmento de Código 3: Zona de definición pública del componente *SimpleComponent*.

Como podemos apreciar en este fragmento de código, cada definición se expresa a través de un enumerado (que en C++ se representa con la estructura *enum*) y cada enumerado posee un nombre y una serie de elementos que poseen un determinado valor entero. Tanto la nomenclatura como algunos de los elementos y valores de los enumerados son proporcionados por CoolBOT.

A continuación vamos a explicar brevemente cada uno de los enumerados:

1. *Definición de estados*. Los estados se definen en el enumerado *States* y en el se definen cada estado que queremos que tenga el componente. Siempre se comienza con la definición del estado de entrada al cual se le asocia el valor *DEFAULT_STATES*. Seguido a este se definen el resto de estados si los hay. La definición se cierra especificando el elemento *STATES* y especificando el estado de entrada al Autómata de Usuario (*RUNNING_ENTRY_STATE*).

2. *Definición de puertos de entrada y de salida públicos*. Los puertos de entrada se definen en el enumerado *InputPorts* y los puertos de salida en el enumerado *OutputPorts*. En ambos casos, al primer puerto que se defina se le asigna un valor por defecto (*DEFAULT_INPUT_PORT*S para los puertos de entrada y *DEFAULT_OUTPUT_PORTS* para los puertos de salida) y después de la última definición de puerto se define un determinado elemento que cierra la definición (*INPUT_PORTS* en los puertos de entrada y *OUTPUT_PORTS* en los de salida). Si no se definen ningún puerto a este último elemento se le asigna el valor por defecto, tal y como podemos apreciar en el Fragmento de Código 3.
3. *Definición de variables controlables y variables observables*. Las variables observables se definen en el enumerado *ObservableVariables* y las variables controlables en el enumerado *ControllableVariables*. Se aplica aquí el mismo esquema de definición de los puertos de entrada y de salida, cambiándose los elementos y valores por los respectivos a las variables observables y variables controlables.
4. *Definición de excepciones*. Las excepciones se definen en el enumerado Exceptions y se define de la misma forma que los anteriores pero aplicando los elementos y valores correspondientes a las excepciones.
5. *Definición de constantes*. El concepto y uso de constantes no está definido en la plataforma CoolBOT sino que es una consecuencia del lenguaje de implementación que emplea la plataforma. Por esta razón, las constantes se definen en un enumerado sin nombre y sin valores y elementos especiales.

En el Fragmento de Código 4 podemos ver un ejemplo en donde se define un puerto de entrada público llamado *A*, un puerto de salida público llamado *B*, una variable controlable llamada *C*, una variable observable llamada *D*, una excepción llamada *E*, dos estados llamados *Main* y *First* y dos constantes públicas llamadas *F* y *G*:

```
public:
    //USER AUTOMATON STATES
    enum States
    {
      MAIN=DEFAULT_STATES,
      FIRST,
      STATES,
      RUNNING_ENTRY_STATE=MAIN
    };
    //PUBLIC INPUT PORTS.
    enum InputPorts
    {
      A=DEFAULT_INPUT_PORTS,
      INPUT_PORTS
    };
    //PUBLIC OUTPUT PORTS.
    enum OutputPorts
    {
      B=DEFAULT_OUTPUT_PORTS,
      OUTPUT_PORTS
    };
    //CONTROLLABLE VARIABLES
    enum ControllableVariables
    {
      C=DEFAULT_CONTROLLABLE_VARIABLES,
      CONTROLLABLE_VARIABLES
    };
    //OBSERVABLE VARIABLES
    enum ObservableVariables
    {
      D=DEFAULT_OBSERVABLE_VARIABLES,
      OBSERVABLE_VARIABLES
    };
    //EXCEPTIONS
    enum Exceptions
    {
      E=DEFAULT_EXCEPTIONS,
      EXCEPTIONS
    };
    //PUBLIC CONSTANTS
    enum
    {
      F = 10,
      G = 0xff
     };
```

Fragmento de Código 4: Ejemplo zona de definición pública.

- *Zona común de definición pública*. Justo después de la zona de definición pública comienza una parte que es común en todo componente CoolBOT. Se trata de una zona donde se encuentra la definición del constructor y destructor del componente así como una serie de métodos estándar que la plataforma exige que tenga todo componente. La definición del constructor se encuentra en el fichero de implementación mientras que la definición del destructor se implementa en el fichero de cabecera. Aparte de las definiciones indicadas, el programador tiene reservada un área para definir sus propias definiciones. En el Fragmento de Código 5, podemos ver un ejemplo de esta zona para el componente *SimpleComponent*:

```
SimpleComponent();
~SimpleComponent() { _release_(); };

int inputPorts() { return INPUT_PORTS; }
int outputPorts() { return OUTPUT_PORTS; }
int controllableVariables() { return CONTROLLABLE_VARIABLES; }
int observableVariables() { return OBSERVABLE_VARIABLES; }
OBox* getOBox(int port)
{
  return (
           ( (port<0) ||
             (port>=OUTPUT_PORTS) )? NULL:
                                    _pOBox_
         );
}
...
int exceptions() { return EXCEPTIONS; }
const char* exceptionName(int exception)
{
  return (
           ( (exception<0) ||
             (exception>=EXCEPTIONS) )?
                   NULL:
                   _ppExceptionsStrings_[exception]
            );
}

// PUBLIC USER SECTION: BEGIN

// PUBLIC USER SECTION: END
```

Fragmento de Código 5: Zona común de definición pública del componente *SimpleComponent*.

- *Zona de definición privada*. Lo que se define en esta parte, varía en función de los requisitos del componente que queremos implementar. Siempre se define en esta zona las declaraciones de los puertos de entrada, puertos de salida y los hilos que son privados en el componente. Al igual que en la zona de definición pública, también se pueden definir constantes privadas.

  Aparte de estos elementos puede aparecer definiciones de centinelas (ver apartado 2.5) y de prioridades de puertos (prioridades que poseen puertos que están asociados a determinados hilos). Esta última definición, solamente aparece cuando se definen componentes con hilos con puertos con prioridades.

  Los elementos se definen de la misma forma que en la zona pública. Es decir, cada elemento se define en un determinado enumerado. La única diferencia esta en la forma de nombrar a los enumerados y a sus respectivos elementos. En CoolBOT a todo elemento que se define como privado se le añade un guión bajo o *undescore* (símbolo "_") antes y despúes del nombre. Si el elemento se define como protegido, se le añade un guión bajo antes del nombre. Los elementos de un enumerado de la zona privada siempre se definen privados por lo que siempre aparecerá el guión bajo antes y después del nombre.

En el Fragmento de Código 6 podemos ver la zona de definición privada del componente *SimpleComponente*:

```
private:
    //PRIVATES INPUT PORTS
    enum _InputPorts_
    {
      _INPUT_PORTS_=_DEFAULT_INPUT_PORTS
    };
    //PRIVATES OUTPUT PORTS
    enum _OutputPorts_
    {
      _OUTPUT_PORTS_=_DEFAULT_OUTPUT_PORTS
    };
    //THREADS
    enum _Threads_
    {
      _THREADS_=_DEFAULT_THREADS
    };
```

Fragmento de Código 6: Zona de definición privada del componente *SimpleComponent*.

A continuación pasamos a comentar brevemente cada uno de estos enumerados:

1. *Definición de puertos de entrada privados*. Los puertos de entrada privados en el enumerado se definen siguiendo un determinado orden. En la Figura 11 podemos apreciar un diagrama de flujo donde se indica el orden en que se definen los puertos de entrada privados en función si el componente es multihilo[3] o no o si se han definido o no puertos de entrada privados con centinelas.

   Por si no se entiende la Figura 11, en el Fragmento de Código 7, vamos a ver cuatro ejemplos de definiciones de puertos de entrada privados. El primero (caso A) refleja la definición de un puerto de entrada privado llamado A definido en un componente monohilo[4]. El segundo (caso B) refleja el mismo puerto pero esta vez le hemos asociado un centinela. El tercer y cuarto caso (caso C y D) son iguales a los dos anteriores pero en este caso el componente es multihilo (se define un hilo denominado HILO).

[3] Un componente CoolBOT es multihilo cuando en dicho componente se definen uno o más hilos (sin contabilizar el hilo *main*). En este caso también puede aparecer la definición del hilo *main* pero siempre acompañado de la definición de uno o más hilos.

[4] Un componente CoolBOT es monohilo cuando en dicho componente no se define ningún hilo a parte del hilo hilo *main*.

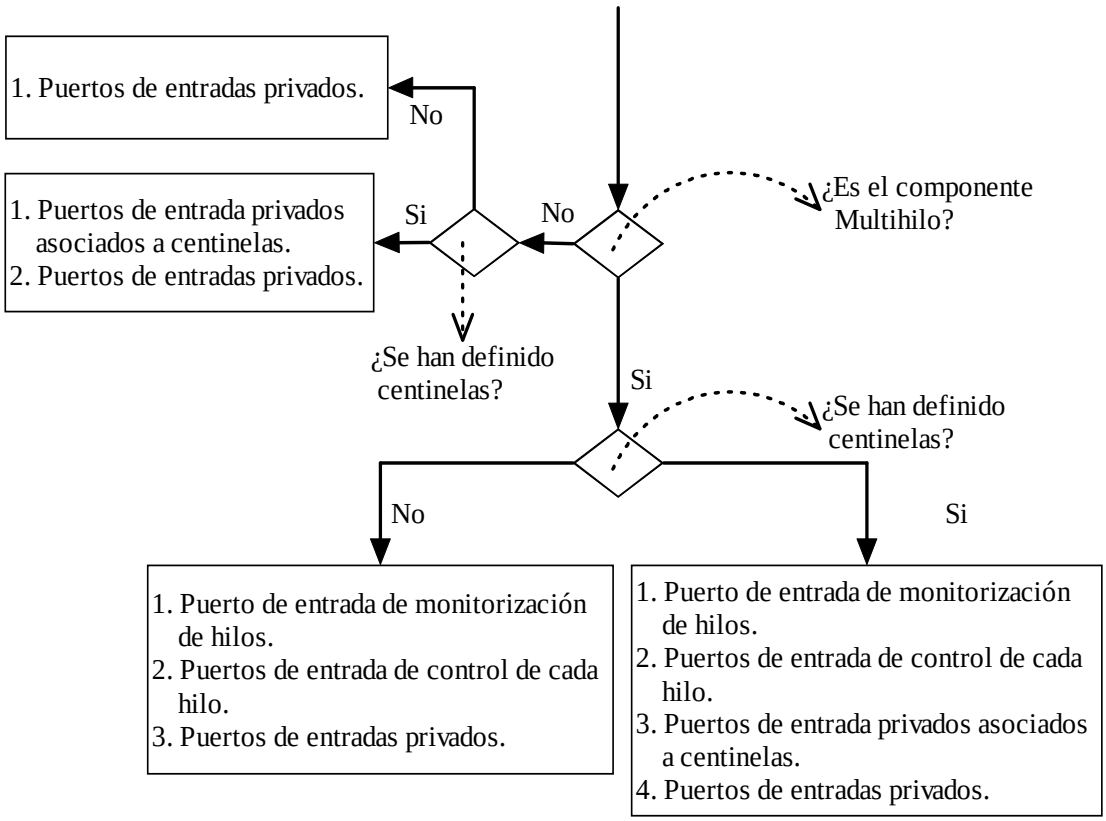


Figura 11: Orden de definición de los puertos de entrada privados.

```
// Caso A.
enum _InputPorts_
{
  _A_=_DEFAULT_INPUT_PORTS,
  _INPUT_PORTS_
};
// Caso B.
enum _InputPorts_
{
  _A_WATCHDOG_PORT_=_DEFAULT_INPUT_PORTS,
  _A_,
  _INPUT_PORTS_
};
// Caso C.
enum _InputPorts_
{
  _PORT_THREAD_MONITORING_=_DEFAULT_INPUT_PORTS,
  _HILO_CONTROL_,
  _A_,
  _INPUT_PORTS_
};
// Caso D.
enum _InputPorts_
{
  _PORT_THREAD_MONITORING_=_DEFAULT_INPUT_PORTS,
  _HILO_CONTROL_,
  _A_WATCHDOG_PORT_,
  _A_,
  _INPUT_PORTS_
};
```

Fragmento de Código 7: Ejemplos de definiciones de puertos de entrada privados.

2. *Definición de puertos de salida privados*. En los puertos de salida privados también deben definirse siguiendo un determinado orden dentro del enumerado en función si el componente es monohilo o multihilo. Si el componente es multihilo, siempre se comienza definiendo el puerto de salida de control de hilos seguido de los puertos de salida privados definidos en el componente. En el Fragmento de Código 8 podemos ver la definición de un puerto de salida privado llamado B en un componente monohilo (caso A) y la misma definición de puerto en un componente multihilo (caso B).

```
// Caso A.
enum _OutputPorts_
{
  _B_=_DEFAULT_OUTPUT_PORTS,
  _OUTPUT_PORTS_
};
// Caso B.
enum _OutputPorts_
{
  _PORT_THREAD_CONTROL_=_DEFAULT_OUTPUT_PORTS,
  _B_,
  _OUTPUT_PORTS_
};
```

Fragmento de Código 8: Ejemplos de definiciones de puertos de salida privados.

3. *Definición de hilos*. Tanto si definimos hilos como si no, hemos de declarar un enumerado que contengan los hilos definidos en el componente. El hilo *main* nunca se contempla en este enumerado. Si no se definen hilos, el enumerado se define tal y como se puede apreciar en el Fragmento de Código 6. En el Fragmento de Código 9, vemos un ejemplo donde se definen dos hilos: el hilo *FIRST_THREAD* y el hilo *SECOND_THREAD*.

```
//THREADS.
enum _Threads_
{
  _FIRST_THREAD_=_DEFAULT_THREADS,
  _SECOND_THREAD_,
  _THREADS_
};
```

Fragmento de Código 9: Definición de dos hilos en un componente multihilo.

Además de estos enumerados y del enumerado de las constantes privadas, pueden aparecer otros dos enumerados, un enumerado para el caso de hilos con puertos de entrada con prioridades y otro enumerado para los centinelas asociados tanto a puertos de entrad públicos como privados. En el Fragmento de Código 10 podemos ver un ejemplo donde aparecen estos dos enumerados.

```
//INPUT PORTS PRIORITIES
enum _HILOInputPortPriorities_
{
  _LOW_=_DEFAULT_INPUT_PORT_PRIORITIES,
  _MEDIUM_,
  _HILO_INPUT_PORT_PRIORITIES_
};
//WATCHDOGS
enum _WatchDogs_
{
  _WATCHDOG_A_=0,
  _WATCHDOGS_
};
```

Fragmento de Código 10: Resto de enumerados.

- *Zona de definiciones estáticas*. Después de las declaraciones de los elementos públicos y privados del componente, comienza una serie de definiciones y métodos especificados con el atributo *static* de C++ que son comunes en todos los componentes. En esta parte del fichero se distinguen tres áreas que varían en función de determinadas circunstancias. A continuación citamos dichas áreas:

    1. *Definiciones de distintos vectores para contener información de los elementos especificados en las zonas de definición pública y privada*. En estas definiciones hay partes que aparecen en todo componente y otras que varían en función de los requisitos del componente, como el que se hayan especificado prioridades o se hayan especificado variables controlables u observables. En el Fragmento de Código 11 podemos ver las definiciones para el caso de un componente monohilo.

```
static const char* _pName_;
static char* _ppStatesStrings_[STATES];
static char* _ppInputPortsStrings_[INPUT_PORTS+_INPUT_PORTS_];
static char* _ppOutputPortsStrings_[OUTPUT_PORTS+_OUTPUT_PORTS_];
static char* _ppControllableVariablesStrings_[CONTROLLABLE_VARIABLES];
static char* _ppObservableVariablesStrings_[OBSERVABLE_VARIABLES];
static char* _ppExceptionsStrings_[EXCEPTIONS];
static char* _ppExceptionsDescriptions_[EXCEPTIONS];
static char* _ppThreadNames_[_THREADS_];
static int _pPrivateInputPorts_[_INPUT_PORTS_];
static int _pPrivateOutputPorts_[_OUTPUT_PORTS_];
static State** _ppStates_;
```

Fragmento de Código 11: Definición de vectores.

    2. *Definición de estructuras y métodos correspondientes a los estados del Autómata por Defecto*. Esta parte es obligatoria y esta presente en todo componente. Varía en función si el componente es monohilo (no se definen hilos) o multihilo. Para cada estado se definen unas determinadas máscaras que modelan el comportamiento de dicho estado y los métodos necesarios para implementar las transiciones del autómata. En el Fragmento de Código 12 se define las estructuras y métodos para el estado *starting* del Autómata por Defecto para un componente monohilo.

```
// common definitions for all states: begin
static int _newPriority_(int port, int index, Component* pComp);
// common definitions for all states: end

// DEFAULT STATES: BEGIN

// state starting: begin
static const bool _pStartingPortsMask_[INPUT_PORTS+_INPUT_PORTS_];
static const bool _pStartingControlMask_[CONTROLLABLE_VARIABLES];

static int _startingEntry_(Component* pComp);
static void _startingExit_(Component* pComp);
// state starting: end
```

Fragmento de Código 12: Estructuras y método para acceder a los estados del Autómata por Defecto de un componente monohilo.

3. *Definición de estructuras y métodos correspondientes a los estados del Autómata del Usuario*. Igual que la anterior área salvo que además de lo especificado, se añaden determinados métodos correspondientes a las transiciones que implementan el Autómata de Usuario del componente. En el Fragmento de Código 13 podemos ver lo comentado en este punto.

```
// USER STATES: BEGIN

// common definitions for all running states: begin
static int _runningNewState_(int port, int index, Component* pComp);
static int _runningNewException_(int port, int index,
                                 Component* pComp);
// common definitions for all running states: end

// state MAIN: begin
static const bool _pMainPortsMask_[INPUT_PORTS+_INPUT_PORTS_];
static const bool _pMainControlMask_[CONTROLLABLE_VARIABLES];

static int _mainEntry_(Component* pComp);
static void _mainExit_(Component* pComp);
// state MAIN: end

// USER STATES: END
```

Fragmento de Código 13: Estructuras y método para acceder a los estados del Autómata de Usuario de un componente monohilo.

- *Zona de definición de manejadores de excepciones y de centinelas*. CoolBOT exige, tanto si se definen como si no, que se creen las estructuras necesarias para manipular excepciones. Si se definen excepciones, además de definir las estructuras necesarias, se especifican los métodos que definen los manipuladores que tratan dichas excepciones.

  La definición de centinelas es opcional en los componentes CoolBOT por lo que solamente definimos las estructuras necesarias para tratarlos cuando se definen en el componente. Aparte de definir las estructuras necesarias, es posible definir por cada centinela un método que trate dicho centinela.

En el Fragmento de Código 14 podemos apreciar definiciones de manejadores de excepciones y definiciones de centinelas.

```
static Exception** _ppExceptions_;

// EXCEPTION HANDLERS: BEGIN

// Exception E begin

static bool _eHandler_(Component* pComp);
static void _eOnSuccessHandler_(Component* pComp);
static void _eOnFailureHandler_(Component* pComp);

// Exception E: end

// EXCEPTION HANDLERS: END
// WATCH DOG DECLARATIONS: BEGIN

static WatchDogStruct _pWatchDogStructs_[_WATCHDOGS_];
static int _pPortToWatchDog_[INPUT_PORTS+_INPUT_PORTS_];
void _launchStopWatchDogs_();

/*
 * WARNING: _restartWatchDog_() SHOULD BE ONLY CALLED INSIDE A
 * WATCHDOG FUNCTION, NEVER NEVER OUTSIDE!.
 */
void _restartWatchDog_(int port)
{ _ppWatchDogs_[ _pPortToWatchDog_[port] ]->start(); }

void _setWatchDogTimeout(int watchDog, Time& timeout);

static int _watchDogTransition_(int port, Component* pComp);

// Watch dog functions: begin
static int _AWatchDog_(int port, Component* pComp);
// Watch dog functions: end

// WATCH DOG DECLARATIONS: END
```

Fragmento de Código 14: Definición de manejadores de excepciones y definición de estructuras y métodos para los centinelas**.**

- *Zona de declaraciones de atributos y métodos*. En esta parte se declaran una serie de estructuras que exige CoolBOT que todo componente disponga. Una parte de estas estructuras es común a todo componente y otra parte varía en función de las necesidades. Destaca en esta zona, en el caso de componente multihilo con espera activa, la definición de un manejador de puertos y la de un método para gestionar la espera activa. En el Fragmento de Código 15 podemos ver un ejemplo donde se da el caso mencionado anteriormente.
- *Zona de definición común*. Esta parte varía en función de si el componente es monohilo o multihilo. Si el componente es multihilo se añade fragmentos de código para garantizar la exclusión mutua de los hilos. El contenido para cada caso es común a todos los componentes de esa clase.
- *Pie del componente*. Lo último que se declara en el fichero son una serie de instancias donde se inicializa distintos elementos estáticos del componente antes que estos se utilicen.

```
// NOTE: multithreaded components need an array of threads
 Thread** _ppThreads_;
 static PollingThreadBody _pThreadBodies_[_THREADS_];
 static void _thread_(void* pParameter);

 // NOTE: this function member only is necessary in multithreaded
 //components
 static bool _portHandler_(int port,void* pComp)
 {
   Prueba* pThis=static_cast<Prueba*>(pComp);
   TransitionFunction pfTransition =
         pThis->_ppStates_[
                pThis->_currentState_]->getPortTransition(port);
   if(pfTransition) pfTransition(port,pThis);
   return true;
 }

 // POLLING THREAD: BEGIN
 static bool _hiloBody_(void* pComp);
 // POLLING THREAD: END
```

Fragmento de Código 15: Definición de manejador de puertos y espera activa.

2.8.2. Esqueleto C++ correspondiente al fichero de implementación.

En la Fragmento de Código 16 podemos ver la estructura que posee este fichero. En este fichero se implementan todas las definiciones realizadas en el fichero cabecera y al igual que se hizo en ese fichero, hemos dividido el fichero en varias partes.

```
/*
 * File: simple-component.cpp
 * Date: 03 November 2007
*/
#include "coolbot_compiler_adjustments.h"
#define SIMPLE_COMPONENT_CPP
#include "simple-component.h"
// INCLUDE USER SECTION: BEGIN

// INCLUDE USER SECTION: END
namespace SimpleComponentSpace
{
    const char* SimpleComponent::_pName_="SimpleComponent";
    char* SimpleComponent::_ppStatesStrings_[SimpleComponent::STATES];
    char* SimpleComponent::_ppInputPortsStrings_
                    [SimpleComponent::INPUT_PORTS+SimpleComponent::_INPUT_PORTS_];
    char* SimpleComponent::_ppOutputPortsStrings_
                    [SimpleComponent::OUTPUT_PORTS+SimpleComponent::_OUTPUT_PORTS_];
    char* SimpleComponent::_ppControllableVariablesStrings_
                    [SimpleComponent::CONTROLLABLE_VARIABLES];
    char* SimpleComponent::_ppObservableVariablesStrings_
                    [SimpleComponent::OBSERVABLE_VARIABLES];
    char* SimpleComponent::_ppExceptionsStrings_[SimpleComponent::EXCEPTIONS];
    char* SimpleComponent::_ppExceptionsDescriptions_[SimpleComponent::EXCEPTIONS];
    char* SimpleComponent::_ppThreadNames_[_THREADS_];

    // ZONA DE MAPEO DE PUERTOS PRIVADOS.
    // ZONA DE MAPEO DE PRIORIDADES DE PUERTOS.
    // ZONA DE MÁSCARAS DE LOS ESTADOS DEL AUTÓMATA POR DEFECTO.
    // ZONA DE MÁSCARAS DE LOS ESTADOS DEL AUTÓMATA DE USUARIO.
    // ZONA DE MAPEO DE HILOS Y MAPEO DE CENTINELAS.
    // ZONA DE DEFINICIÓN DE MÉTODOS.

    // USER DEFINITIONS SECTION: BEGIN
    // USER DEFINITIONS SECTION: BEGIN
}
```

Fragmento de Código 16: Estructura del fichero de implementación de un componente denominado *SimpleComponent*.

A continuación, pasamos a comentar cada una de las partes que se muestran en el Fragmento de Código 16:

- *Cabecera del componente*. Esta parte es casi identifica a la del fichero de cabecera. Tal y como vemos en el Fragmento de Código 17, al principio del todo tenemos un comentario con la misma información que hemos especificado en el fichero de cabecera. Le sigue una inclusión, obligada por la plataforma CoolBOT, para que el componente se ajuste a los requisitos exigidos por el compilador de C++. Luego se emplea la cláusula *#define* para evitar que el compilador defina dos veces este fichero y por último, se inicia una serie de sentencias de *#include*, encabezado por el "*include*" que referencia al fichero de cabecera del componente, y a continuación comienza el espacio de nombres donde se realiza la implementación de lo definido en el fichero cabecera del componente.

```
/*
 * File: simple-component.cpp
 * Description: Componente de prueba.
 * Date: 15 October 2007
 * Author: Antonio C. Dominguez-Brito
 *         Dpto. Informatica y Sistemas
 *         Universidad de Las Palmas de Gran Canaria.
 */

#include "coolbot_compiler_adjustments.h"
#define SIMPLE_COMPONENT_CPP
#include "simple-component.h"

// INCLUDE USER SECTION: BEGIN

// INCLUDE USER SECTION: END

namespace SimpleComponentSpace
{
  const char* SimpleComponent::_pName_="SimpleComponent";
  . . .
```

Fragmento de Código 17: Cabecera del componente denominado *SimpleComponent*.

- *Zona de mapeo de puertos privados*. Tanto los puertos de entrada como de salida se almacena en vectores independientes que hemos definido como *static* en el fichero de cabecera. A los puertos públicos, independientemente de que sean de entrada o de salida, se accede directamente empleando como índice el nombre del puerto especificado en los enumerados correspondientes.

  A los puertos privados no es posible acceder directamente ya que si empleásemos su nombre como índice estaríamos accediendo a algún puerto público. Por esta razón, se definen dos vectores para los puertos privados, uno para los puertos de entrada y otros para los puertos de salida donde se mapean

las direcciones de forma que se pueda acceder a través del valor de sus enumerados definidos en el fichero cabecera.

En el Fragmento de código 18, vemos como se mapean los puertos de entrada privados en el componente *SimpleComponent*. Como podemos observar en cada posición del vector se calcula la dirección en donde se encuentra el puerto de entrada privado. Los puertos *_EMPTY_TRANSITION* y *_TIMER* son puertos de entrada privados por defecto de la plataforma CoolBOT (ver apartado 2.5).

```
int SimpleComponent::_pPrivateInputPorts_[SimpleComponent::_INPUT_PORTS_]=
{
    INPUT_PORTS + _EMPTY_TRANSITION,
    INPUT_PORTS + _TIMER
};
```

Fragmento de Código 18: Mapeo de puertos privados.

- *Zona de mapeo de prioridades de puertos*. Si hemos definido hilos con prioridad, a cada puerto de entrada asociado al hilo le habremos indicado una prioridad. Para que CoolBOT sepa que prioridad posee cada puerto de entrada, se puede definir un vector de prioridad por hilo. En cada vector se especifican, para cada puerto definido en el componente más los proporcionados por la plataforma, la prioridad que posee cada puerto. Primero se especifican los puertos de entrada definidos como públicos y luego los definidos como privados. Si en ese hilo hay un puerto que no se encuentra asociado se le asigna la prioridad más alta que hemos definido en el enumerado correspondiente del fichero de cabecera. En el Fragmento de código 19, podemos ver un ejemplo donde se define un hilo denominado *HILO* con dos prioridades, *LOW* y *MEDIUM*, a las cuales se le asocian respectivamente los puertos de entrada *A* (público) y *B* (privado). Este último tiene asociado un centinela.

```
int SimpleComponent::_pHILOInputPortPriorities_[INPUT_PORTS+_INPUT_PORTS_]=
{
  _MEDIUM_, // CONTROL
  _LOW_,    // A
  _MEDIUM_, // _EMPTY_TRANSITION
  _MEDIUM_, // _TIMER
  _MEDIUM_, // _PORT_THREAD_MONITORING_
  _MEDIUM_, // _LISTENING_CONTROL_
  _MEDIUM_, // _B_WATCHDOG_PORT_
  _MEDIUM_  // _B_
};
```

Fragmento de Código 19: Mapeo de prioridades de puertos.

- *Zona de máscaras de los estados del Autómata por Defecto*. En cada estado del Autómata por Defecto se definen unas determinadas máscaras que de determinan el comportamiento de ese estado, concretamente si en un estado determinado un puerto de entrada o una variable controlable está habilitada

(desenmascarada) o deshabilitada (enmascarada). Para ello se emplean vectores de tipo lógico donde se especifica si el valor que caracteriza esa máscara esta enmascarado o desenmascarado. Si esta enmascarado se pone el valor *true* y si esta desenmascarado se pone el valor *false*. En el Fragmento de Código 20, observamos las máscaras del estado *starting* del componente monohilo *SimpleComponent*.

```
// state starting: begin

const bool SimpleComponent::_pStartingPortsMask_[INPUT_PORTS+_INPUT_PORTS_]=
{
    true,  // CONTROL
    true,  // _EMPTY_TRANSITION
    true   // _TIMER
};
const bool SimpleComponent::_pStartingControlMask_[CONTROLLABLE_VARIABLES]=
{
    true,  // NEW_STATE
    true,  // NEW_PRIORITY
    true   // NEW_EXCEPTION
};

// state starting: end
```

Fragmento de Código 20: Máscaras de los estados del Autómata por Defecto.

- *Zona de máscaras de los estados del Autómata de Usuario*. Igual que la parte anterior salvo que se aplica para los estados del Autómata de Usuario.

```
// USER STATES MASK DEFINITIONS: BEGIN

// state MAIN: begin

const bool SimpleComponent::_pMainPortsMask_[INPUT_PORTS+_INPUT_PORTS_]=
{
  false, // CONTROL
  true,   // _EMPTY_TRANSITION
  true   // _TIMER
};
const bool SimpleComponent::_pMainControlMask_[CONTROLLABLE_VARIABLES]=
{
   false, // NEW_STATE
   false, // NEW_PRIORITY
   false  // NEW_EXCEPTION
};

// state end: MAIN

// USER STATES MASK DEFINITIONS: END
```

Fragmento de Código 21: Máscaras de los estados del Autómata de Usuario.

- *Zona de mapeo de hilos y mapeo de centinelas*. Obligatoriamente, tanto si el componente es monohilo o multihilo, hemos de definir un vector donde se asocia a cada hilo el puerto de entrada que le hemos asociado. Los puertos proporcionados por la plataforma siempre se asocian al hilo *main*, al cual se le asignarán todos los puertos de entrada en el caso de que el componente sea monohilo.

En el caso de que hayamos definido hilos con la política de espera activa, dispondremos de otro vector donde se mapean las funciones que gestionan la espera activa para cada hilo. Si un hilo no posee la política de espera activa se coloca el valor *NULL*.

Por último si hemos definido centinelas en el componente deberemos, también definir una serie de estructuras donde se mapean los centinelas con los puertos asociados a dichos centinelas.

En el Fragmento de Código 22 podemos apreciar un ejemplo de un componente multihilo que pose un hilo con espera activa denominado HILO y un puerto de entrada privado llamado B al que se le ha asociado un centinela (watchdog).

```
// THREAD STRUCTURES: BEGIN

int SimpleComponent::_pPortToThread_[INPUT_PORTS+_INPUT_PORTS_]=
{
   _MAIN, // CONTROL
   _MAIN, // _EMPTY_TRANSITION
   _MAIN, // _TIMER
   _MAIN, // _PORT_THREAD_MONITORING_
   _HILO_,  // _HILO_CONTROL_
   _MAIN,  // _B_WATCHDOG_PORT_
   _HILO_  // _B_
};

// THREAD STRUCTURES: END

// WATCH DOG STATIC STRUCTURES: BEGIN

Component::WatchDogStruct Prueba::_pWatchDogStructs_[_WATCHDOGS_]=
{
  { // _B_WATCHDOG_PORT_
    _B_, // timedPort
    _pPrivateInputPorts_[_B_WATCHDOG_PORT_], // watchDogPort
    _bWatchDog_ // pfWatchDogFunction
  }
};
int SimpleComponent::_pPortToWatchDog_[INPUT_PORTS+_INPUT_PORTS_]=
{
   -1, // CONTROL,
   -1, // EMPTY_TRANSITION,
   -1, // TIMER,
   -1, // _PORT_THREAD_MONITORING_,
   -1,  // _HILO_CONTROL_
   _WATCHDOG_B_  // _B_WATCHDOG_PORT_
   -1  // _B_
};

// WATCH DOG STATIC STRUCTURES: END

Component::PollingThreadBody  Prueba::_pThreadBodies_[_THREADS_]=
{
  NULL, // _MAIN for the main thread should be always NULL)
  _hiloBody_  // HILO
};
```

Fragmento de Código 22: Máscaras de hilos y centinelas.

- *Zona de definición de métodos*. En esta parte procedemos a declarar los cuerpos “vacíos” de cada método especificado en el fichero de cabecera, en los cuales el programador completará la lógica necesaria para implementar la funcionalidad

del componente. Esta parte es la más dura y costosa de realizar ya que debemos de tener en cuenta diversos factores que caracterizan al componente.

Cada método posee una determina estructura y las partes donde el programador escribe la lógica del método se encuentra delimitado por unos comentarios que empiezan por: *USER FILL IN SECTION: BEGIN* y terminan con *USER FILL IN SECTION: END*.

Capítulo 3

Metodología, recursos y plan de trabajo.

En este tercer capítulo, vamos a comentar que metodologías hemos empleado en el desarrollo del presente proyecto. Además veremos los recursos que se necesitaron y plan de trabajo que se siguió en este proyecto.

3.1. Metodología.

Tal y como hemos visto en el apartado 1.2, nuestro objetivo es desarrollar un compilador que a partir de un determinado lenguaje, que nos permita modelar componentes CoolBOT, nos genere unos esqueletos en C++ (recordemos que también se generará esqueletos XML y que a partir de estos se podrán obtener los esqueletos C++).

La disciplina de Procesadores de Lenguaje nos proporciona la metodología adecuada para el diseño y desarrollo de un compilador. El proceso de desarrollo se divide en una serie de fases[5] en donde cada fase se encarga de una tarea del proceso de compilación. Cada fase se construye sobre la fase anterior, es decir, la salida resultante de una fase, se utiliza como entrada en la siguiente fase.

Esta metodología resulta ideal para el desarrollo del compilador ya que nos proporciona un guión de cómo hemos de desarrollarlo. No obstante, aunque esta metodología nos dice en todo momento que es lo que hay que hacer, no nos dice como hacerlo. Es decir, no nos dice que paradigma de programación debemos emplear para implementar cada fase. Por esta razón, a esta metodología que nos proporciona el área de Procesadores de Lenguaje, le vamos a añadir otros criterios proporcionados por la disciplina de la Ingeniería del Software.

---

[5] Se puede definir una *fase* como un módulo en el que podemos descomponer el trabajo del compilador [Pérez-Aguiar 1998]. Un compilador se diseña en varias fases, que no tienen por qué ser secuenciales y que se engloban a su vez en dos fases:

- Fases de análisis: contiene las fases del análisis léxico, sintáctico y semántico
- Fases de síntesis: contiene la fase de generación de código.

La Ingeniería del Software nos proporciona metodologías para estructurar y organizar el software. Siguiendo estas metodologías nos aseguramos que el producto resultante cumpla con los requisitos marcados al principio del proyecto y que tenga garantizado unos mínimos de calidad.

Dadas las características y fases de diseño de un compilador, podemos estructurar el software aplicando el modelo de capas o de máquina abstracta [Pressman, 2006]. Este modelo se caracteriza en que organiza un sistema software en una serie de capas, donde en cada capa se suministran un conjunto de servicios y se define una máquina abstracta (en la capa se define un tipo de abstracción) cuyo lenguaje máquina (los servicios suministrados por la capa) se utilizan para implementar el siguiente nivel de la máquina abstracta. De este modo, dispondremos de una capa para el análisis léxico, otra capa para el análisis sintáctico que se construye sobre los servicios que proporciona la capa del análisis léxico y así sucesivamente.

Este enfoque permite aplicar como paradigma de desarrollo el modelo incremental [Pressman, 2006] de modo que a medida que vamos desarrollando vamos disponiendo de partes funcionales del compilador (incrementos) sin tener que esperar hasta el final del proceso de desarrollo. De este modo por cada fase de diseño del compilador tendremos una parte funcional que podremos usar y probar. En la siguiente Figura 10 podemos apreciar lo anteriormente mencionado.

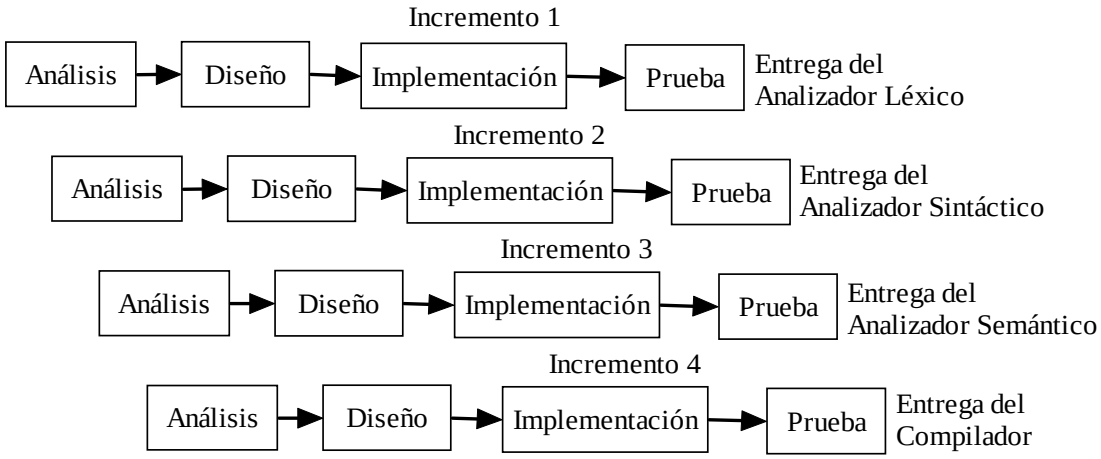


Figura 12: Paradigma de desarrollo: Modelo Incremental.

Cada incremento esta constituido por una serie fases que constituyen el proceso de desarrollo del paradigma:

- *Análisis*: en esta fase estudiamos el problema que se pretende resolver en ese incremento, capturando los requisitos funcionales y estableciendo los objetivos

del incremento. Por ejemplo, en el análisis léxico nos centramos en que símbolos reconocerá el analizador léxico.

- *Diseño*: en base a los resultados obtenidos en la fase de análisis, se determina como se va a desarrollar el incremento.
- *Implementación*: una vez diseñado el incremento, simplemente se pasa a codificarlo empleando un lenguaje de alto nivel.
- *Prueba*: una vez implementado y compilado el código, al incremento se le ejecutan una batería test con el fin de depurarlo y de asegurar que el incremento cumple los objetivos planteados en la fase de análisis.

3.2. Recursos necesarios.

Para la elaboración del presente trabajo requerimos de una serie de recursos tanto hardware como software. En la Figura 11, podemos ver un resumen de los recursos empleados en este trabajo:

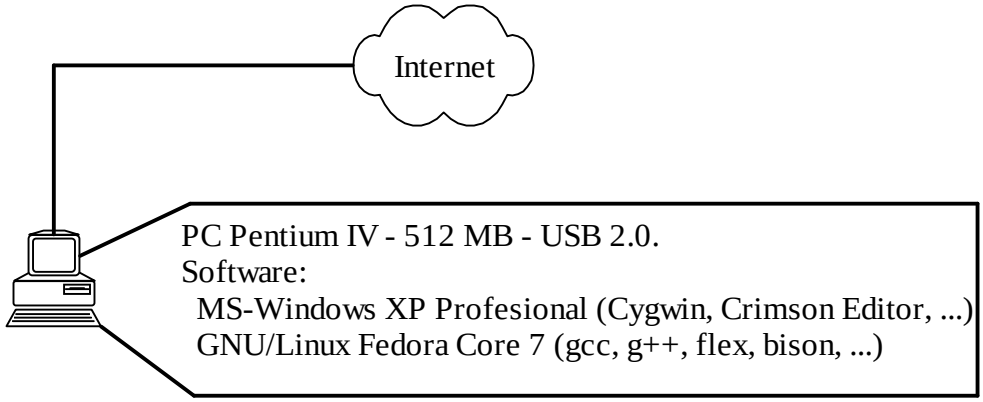


Figura 13: Recursos necesarios.

En este apartado, mostraremos una pequeña descripción de cada uno de los recursos empleados:

3.2.1. Recursos hardware.

Requerimos de los siguientes elementos:

- *Ordenador personal*. PC Pentium IV con al menos 512 MB y USB 2.0 proporcionado por el grupo de investigación GIAS, situado en el laboratorio del Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (IUSIANI).

- *Conexión a Internet*. conexión por banda ancha suministrado por la Universidad de Las Palmas de Gran Canaria.
- *Pendrive USB*. unidad de almacenamiento de 1GB empleado para realizar copias de seguridad.

3.2.2. Recursos software.

El trabajo presentado en este documento se ha desarrollado usando tanto la plataforma GNU/Linux como la plataforma MS Windows por lo que se ha empleado tanto software libre como propietario, predominando siempre el software libre sobre el de pago. Ambas plataformas se encontraban preinstaladas en el ordenador personal suministrado. A continuación, pasamos a comentar el uso que se le dio a cada plataforma y el software que se empleado en cada una:

- *Plataforma MS Windows*: esta plataforma se usó en la fase de desarrollo (análisis, diseño e implementación) y gran parte del software que se utilizó se encontraba preinstalado en el ordenador. A continuación podemos ver el software empleado:
    1. *Sistema operativo*: Microsoft Windows XP Professional Edition (preinstalado).
    2. *Software Cygwin* [cygwin]: se trata de un conjunto de librerías dinámicas (DLL) las cuales proveen una capa de emulación de la funcionalidad de GNU/Linux o Unix, de modo que es posible compilar y ejecutar programas para GNU/Linux en Windows.

       El software instalado sobre *Cygwin* para la elaboración del proyecto es el siguiente:
        - *bison* [bison]: generador sintáctico compatible con YACC empleado para implementar el analizador sintáctico.
        - *flex* [flex]: generador léxico empleado para implementar el analizador léxico.
        - *cvs*: aplicación que implementa un sistema de control de versiones para la gestión de los ficheros del proyecto.
        - *gcc* [gcc]: compilador GNU lenguaje C.
        - *g++*: compilador GNU lenguaje C++.
        - *gdb y cgdb*: depuradores GNU para depurar código C/C++.

    - *make*: herramienta GNU de generación o automatización de código empleado para generar el ejecutable de nuestro compilador.
    - *nano*: editor de texto en modo texto.
    - *X Window System*: sistema de ventanas que dota de interfaz gráfica a los sistemas operativos GNU/Linux.
  3. *Software SSH*: aplicación preinstalada para acceder y controlar de forma segura a máquinas remotas a través de una red. Se ha empleado para poder acceder al servidor SIAS situado en el laboratorio del IUSIANI donde se encuentra el grupo GIAS.
  4. *Software Crimson Editor*: editor profesional de código fuente, empleado para realizar la implementación del compilador.
  5. *Software 7zip*: Compresor con un alto grado de compresión que soporta, entre otros, los formatos ZIP y RAR.
  6. *Software Microsoft Office Professional Edition*: software ofimático (preinstalado) empleado para realizar la documentación.

- *Plataforma GNU/Linux*: se ha empleado como sistema operativo la distribución GNU/Linux Fedora Core 7 (preinstalado), en el cual se ha instalado el mismo software que se ha empleado en la plataforma MS Windows cambiando el software propietario empleado en MS Windows por uno equivalente de amplio uso en la comunidad de Software libre. Esta plataforma se ha empleado en la fase de prueba y depuración.

Como podemos observar, la mayoría de las herramientas software utilizadas son gratuitas con licencia GNU y por tanto Software libre. El software propietario utilizado en este proyecto fue proporcionado por el laboratorio del Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (IUSIANI).

3.3. Plan de trabajo y temporización.

El plan de trabajo se ha desglosado en varias etapas, alguna de las cuales está dividida a su vez en partes para especificar más adecuadamente dicha etapa. En cada una de las diferentes etapas se va a especificar el tiempo, en horas, que se tardó en la realización de dicha etapa, así como una breve descripción de cada etapa. En la Figura 7 podemos apreciar gráficamente un resumen del porcentaje de tiempo dedicado a cada etapa.

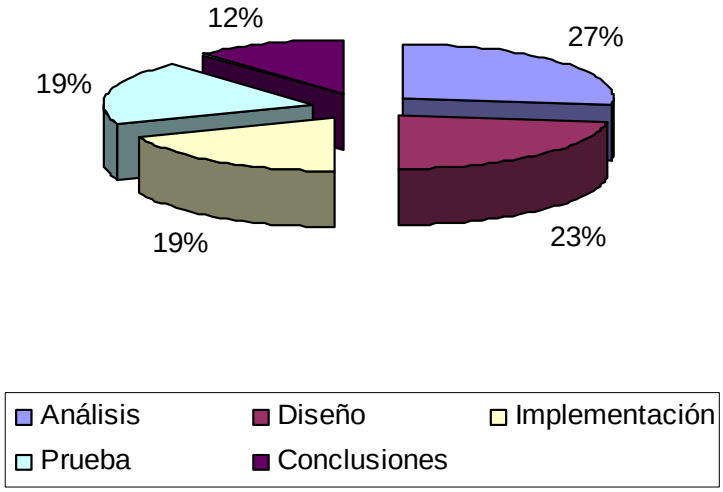


Figura 14: Porcentajes de tiempo dedicados a cada etapa.

3.3.1. Etapa 1: Análisis.

Esta etapa vamos a analizar el problema planteado y a capturar los requisitos necesarios para poder diseñar la solución que cumplirá los objetivos marcados en la sección 1.2. Esta etapa se subdivide a su vez en las siguientes partes:

- *Estudio de la plataforma CoolBOT*. Se estudiará el marco de programación CoolBOT en general, y de la estructura de los esqueletos C++, tomando como punto de partida componentes CoolBOT ya desarrollados y operativos. Duración: 40 horas.
- *Elaboración del lenguaje descriptivo*. En base al estudio resultante de la etapa anterior, se definirá un lenguaje descriptivo que defina y describa el esqueleto C++ de un componente CoolBOT. Al mismo tiempo que elaboramos el lenguaje descriptivo, se especificará la sintaxis XML del mismo. Duración: 15 horas.
- *Requisitos del software*. Una vez concluidas las etapas anteriores, procedemos a especificar que requerimientos necesitamos para obtener la funcionalidad que deseamos que tenga el software para que cumpla los objetivos marcados. Duración: 15 horas.

  Tiempo total dedicado a esta etapa: 70 horas.

3.3.2. Etapa 2: Diseño.

En esta etapa, partiendo de los requisitos obtenidos en la etapa de análisis, elaboramos la estructura del software que vamos a desarrollar en este trabajo. En concreto se describen los elementos del diseño arquitectónico [Pressman, 2006], el lenguaje de programación a utilizar y la tecnología necesaria.

Tiempo total dedicado a esta etapa: 60 horas.

3.3.3. Etapa 3: Implementación.

En la etapa de implementación, simplemente pasamos a un proceso de codificación donde se tiene en cuenta el diseño planteado en la etapa anterior. Esta etapa se subdivide en dos partes:

- *Estudio de las herramientas necesarias*. Estudiaremos que herramientas necesitaremos para llevar a cabo la implementación. Duración: 10 horas.
- *Implementación de la solución*. Se implementa la solución planteada en este trabajo. Duración: 40 horas.

Tiempo total dedicado a esta etapa: 50 horas.

3.3.4. Etapa 4: Prueba y resultados.

Una vez implementado y compilado la solución, pasamos a ejecutar una batería de test para asegurarnos de que dicha solución cumple con los objetivos marcados en la sección 1.2.

Tiempo total dedicado a esta etapa: 50 horas.

3.3.5. Etapa 5: Evaluación y conclusiones finales.

En esta etapa hacemos un análisis comparativo en cuanto al tiempo de desarrollo ganado utilizando la solución planteada frente al desarrollo actual. A partir de este análisis obtendremos las conclusiones experimentales finales de este trabajo.

Tiempo total dedicado a esta etapa: 30 horas.

3.3.6. Etapa 6: Documentación.

Confección del presente documento, así como la preparación de la defensa de este trabajo.

Tiempo total dedicado a esta etapa: 40 horas.

3.3.7. Tiempo total.

El tiempo total dedicado a este proyecto es de 300 horas.

Capítulo 4
Análisis.

En el capítulo anterior estudiamos la plataforma CoolBOT [Domínguez-Brito et al., 2003] y la estructura que posee un esqueleto C++ de un componente CoolBOT. En este capítulo, en base a ese estudio, elaboraremos un lenguaje descriptivo, a través del cual podremos definir componentes CoolBOT, y la sintaxis XML correspondiente a dicho lenguaje. Una vez definido el lenguaje descriptivo pasaremos a ver que otros requisitos necesitamos para desarrollar la solución planteada en el apartado 1.2.

4.1. Lenguaje Descriptivo.

Antes de empezar a ver el lenguaje descriptivo debemos introducir una serie de conceptos con respecto a la sintaxis XML. Todos los documentos escritos en XML deben estar *bien formados*, y este es el requisito mínimo que deben cumplir los documentos. Un documento esta bien formado si cumple con todas las definiciones básicas de formato y puede, por lo tanto, ser analizado correctamente por cualquier analizador sintáctico (parser) que cumpla con la norma. Las definiciones básicas de formato que ha de cumplir el documento son las siguientes:

- Los documentos han de seguir una estructura estrictamente jerárquica con lo que respecta a las etiquetas que delimitan sus elementos. Una etiqueta debe estar correctamente incluida en otra, es decir, las etiquetas deben estar correctamente anidadas. Los elementos con contenido deben estar correctamente cerrados.
- Los documentos XML sólo permiten un elemento raíz del que todos los demás sean parte, es decir, solo pueden tener un elemento inicial.
- Los valores atributos en XML siempre deben estar encerrados entre comillas simples o dobles.
- El XML es sensible a mayúsculas y minúsculas.

- Es necesario asignar nombres a las estructuras, tipos de elementos, entidades, elementos particulares, etc. En XML los nombres tienen alguna característica en común.
- Las construcciones como etiquetas, referencias de entidad y declaraciones se denominan marcas; son partes del documento que el procesador XML espera entender y dan estructura al documento. El resto del documento entre marcas conforma el contenido del documento.

Por último, aunque no forma parte de las definiciones básicas de formato, se recomienda que los documentos XML empiecen con una sentencia que declara al documento como un documento XML.

Con estas decisiones nos aseguramos que un documento escrito en XML este bien construido pero no tenemos conocimiento de si el contenido que almacena es correcto o no. Para ello, el documento se ha de validar.

La validación es la parte más importante dentro de este análisis, ya que determina si un documento se ciñe a las restricciones descritas en el esquema utilizado para su construcción. Cuando creamos documentos XML válidos aumentamos su funcionalidad y utilidad.

En este trabajo, los documentos XML no se validarán. Únicamente nos encargaremos de generar y procesar documentos XML bien formados. No obstante, durante la definición de la sintaxis XML, se aportará la suficiente información para que posteriormente, en un uso futuro, se pueda implementar la validación.

4.1.1. Definición del lenguaje descriptivo.

El lenguaje descriptivo que vamos a definir se llama **Description Language**, de aquí en adelante **DL**, y refleja todas las características de un componente CoolBOT. Las características principales de este lenguaje son las siguientes:

- *Se trata de un lenguaje sensible a mayúsculas y minúsculas*. Es decir, existe una distinción entre los caracteres en minúsculas y mayúsculas por lo que los identificadores “First” y “first” no son iguales y denotarían dos elementos distintos.
- *Orientado a componentes*. La unidad principal del lenguaje es el componente y todo gira en torno a él. Los atributos que se pueden utilizar para definir un componente son:
    1. Puertos, tanto de entrada como de salida.

2. Variables observables y variables controlables.
3. Excepciones.
4. Estados del autómata de usuario.
5. Hilos (Threads).
6. Centinelas de puertos (WatchDogs).

En el apéndice B podemos ver los diagramas sintácticos correspondientes a este lenguaje y en el apéndice C la notación EBNF (Extended Backus-Naur Form) del mismo. Para una mayor compresión del lenguaje DL, se recomienda seguir tanto los diagramas sintácticos como la notación EBNF de dichos apéndices mientras se ven los siguientes apartados.

4.1.2. 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. Todo lo que hay hasta el final de línea se interpreta como un comentario.
- *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.

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.

Los comentarios de múltiples líneas no se pueden anidar. Es decir, un comentario no puede contener otro comentario. En el Fragmento de Código 23, podemos ver una serie de ejemplos de comentarios en DL tanto válidos como inválidos.

Los comentarios en XML se especifican empleando el mismo formato que en lenguaje HTML. Por tanto, tal y como podemos apreciar en el Fragmento de Código 24, todo comentario en XML se inicia con la secuencia "<*!*--" y se termina con la secuencia "-->". Todo lo que esta en medio de esas dos secuencias se considera un comentario.

```
// Esto es un comentario de una línea.
//y este también.
/*
    Sin embargo, este es un comentario
    de múltiples líneas.
*/
/*
    Este comentario no es válido.
    /*
         No es válido porque posee otro comentario anidado,
    */
*/
/*  Comentario inválido al no finalizarlo correctamente.
```

Fragmento de Código 23: Ejemplos de comentarios escritos en DL.

```
<!-- Esto es un comentario -->
<!-- y este es
       otro comentario -->
```

Fragmento de Código 24: Ejemplos de comentarios escritos en XML.

4.1.3. Nombre 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++.

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 esta 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. En la Tabla 5 y 6 podemos ver ejemplos de nombres de identificadores válidos e inválidos.

Nombres de identificadores válidos	
Pioneer	*EsteEsUnComponenteCuyoNombreEsMuyLargoEnTotal47*
FirstComponent	simple_component
Mi_player_robot	Componente_40

Tabla 5: Ejemplos de nombres de identificadores válidos.

Nombres de identificadores no válidos	
_Prueba	*Mi-player-robot*
$Variable	*class*

Tabla 6: Ejemplos de nombre de identificadores inválidos.

4.1.4. Valores literales.

Un literal es toda notación de representación de un valor dentro de un lenguaje de programación de código fuente. 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. Solamente en la definición de constantes se pueden definir literales numéricos negativos.

  En la Tabla 7 podemos ver ejemplos de valores literales válidos e inválidos.

Literales cadena válidos		
“Esto es un literal cadena”	*“Esto también es válido”*	
Literales cadena no válidos		
“Esto no es válido “porque no se pueden anidar” literales cadena”		
Literales numéricos válidos		
900	0	-1

Tabla 7: Ejemplos de valores literales.

4.1.5. Estructura de un componente.

En cada fichero fuente, sólo se puede definir un componente y la estructura que sigue esta formada por dos partes: una parte de declaración donde especificamos que vamos a definir un componente y otra parte de definición donde se define el componente. A continuación, podemos ver dicha estructura:

```
/* Parte de declaración del componente. */
component <<identificador>>
{
    /* Parte de definición del componente. */
    <<sentencia documentación cabecera>>
    <<sentencia constantes>>
    <<sentencia puertos>>
    <<sentencia variables observables>>
    <<sentencia variables controlables>>
    <<sentencia excepciones>>
    <<sentencia estados>>
    <<sentencia hilos>>
};
```

La parte de declaración comienza con la palabra reservada *component* seguido de un nombre de identificador, el cual no se podrá volver a utilizar más como identificador en la parte de definición el componente.

La parte de definición se encuentra encerrada entre llaves (símbolos “*{*“ y “*}*”) y contiene todas las posibles sentencias que permite definir el lenguaje DL en la definición de un componente. La definición del componente se cierra con un punto y coma (símbolo “*;*”)

Una sentencia es la unidad lógica completa más simple que tiene sentido computacional en sí misa, que proporciona sentido computacional completo y que esta 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 documentación cabecera) pueden definirse más de una vez. Como mínimo debe aparecer en la definición la sentencia estados donde el programador especifica la definición del autómata de usuario y las transiciones que se realizan en cada estado. El resto de sentencias son opcionales, dejando en manos del programador la responsabilidad de definirlos o no.

En el Fragmento de Código 25 podemos ver la correcta definición de un componente llamado *SimpleComponent* y la definición incorrecta de otro componente denominado *ComponentNotValid*.

```
//Definición correcta componente en DL.
component SimpleComponent
{
    entry state Main
    {
        transition on empty_transition;
    }
};
```

```
//Definición incorrecta componente en DL.
component ComponentNotValid
{
    /*
      Definición de componente no válido.
      Como mínimo hay que definir un
      estado del autómata de usuario.
    */
};
```

Fragmento de Código 25: Definición correcta e incorrecta de un componente CoolBOT en DL.

En un documento DLXML[6], un componente se declara con las etiquetas *<component>* y *</component>*. En medio de esas dos etiquetas se declaran una serie de atributos que definen el componente. El primer atributo que se define es el que especifica el nombre que tendrá el componente. Este atributo se define especificando un identificador

[6] Denominaremos documento *DLXML* (Descripción Language XML) a un documento que contiene la definición en DL de un componente CoolBOT en sintaxis en XML.

encerrado entre las etiquetas <*name*> y </*name*>. El resto de atributos se definen en un determinado orden y son los siguientes:

- Documentación de cabecera (Header).
- Definición de constantes.
- Definición de puertos.
- Definición de variables.
- Definición de excepciones.
- Definición de estados.
- Definición de hilos.

En el Fragmento de Código 26 podemos ver la definición en XML del componente *SimpleComponent* correspondiente a la definición vista en el Fragmento de Código 25.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Definición de mi primer componente en DL. -->
<component>
          <name>SimpleComponent</name>
          <state>
               <class>entry</class>
               <name>Main</name>
               <transition>
                    <name>empty_transition</name>
                    <source>Main</source>
                    <target>Main</target>
               </transition>
          </state>
</component>
```

Fragmento de Código 26: Definición del componente *SimpleComponent* en XML.

4.1.6. Documentación de cabecera (header).

DL proporciona otro mecanismo de documentación a parte de los comentarios. Este mecanismo denominado documentación de cabecera, de aquí en adelante *header*, nos permite añadir una determinada información al componente.

La definición del *header* es opcional y si se define sólo puede existir una definición por componente. El formato de definición es el siguiente:

```
/* Parte de declaración del Header. */
header
{
    /* Parte de definición del Header. */
    <<atributo autor>>
    <<atributo descripción>>
    <<atributo institución>>
    <<atributo versión>>
}
```

La declaración comienza con la palabra reservada *header* seguido 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.

4.1.6.1. Atributos del header.

Cada atributo almacena un tipo concreto de información y los atributos que podemos definir son:

- *atributo autor*: 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 esta definiendo. Se define utilizando la palabra reservada *version* seguida de un literal cadena que contiene la información

4.1.6.2. Sintaxis XML del header.

En un documento DLXML, el header se define empleando la siguiente sintaxis:

```
<!-- Parte de declaración del Header. -->
<header>
        <!-- Parte de definición del Header. -->
        <<atributo autor>>
        <<atributo descripción>>
        <<atributo institución>>
        <<atributo versión>>
</header>
```

donde:

- *atributo autor*: la definición comienza con la etiqueta **"<*author*>"** y termina con la etiqueta "**</*author*>**". En medio de esas etiquetas sólo se puede especificar un valor literal cadena.
- *atributo descripción*: la definición comienza con la etiqueta **"<*description*>"** y termina con la etiqueta "**</*description*>**". En medio de esas etiquetas sólo se puede especificar un valor literal cadena.
- *atributo institución*: la definición comienza con la etiqueta **"<*institution*>"** y termina con la etiqueta "**</*institution*>**". En medio de esas etiquetas sólo se puede especificar un valor literal cadena.
- *atributo versión*: la definición comienza con la etiqueta **"<*version*>"** y termina con la etiqueta "**</*version*>**". En medio de esas etiquetas sólo se puede especificar un valor literal cadena

En el Fragmento de Código 27 podemos ver ejemplos de definiciones de header tanto correctos como incorrectos. Las definiciones correctas del header vienen acompañadas de su correspondiente traducción a sintaxis XML.

```
//Definición válidas del header en DL con su correspondiente sintaxis en XML.
header                                  <header>
{                                           <author>“Francisco J. S. J.
    author “Francisco J. S. J.                      Antonio C. D. B.”</author>
            Antonio C. D. B.”;              <description>“Prueba.”</description>
    description “Prueba.”;                  <institution>"U.L.P.G.C."</institution>
    institution “U.L.P.G.C.”;               <version>"1.1.0"</version>
    version “1.1.0”                     </header>
}
//Definiciones no válidas del header.
header                                      header
{                                           {
    /*                                          author “Francisco J. S. J.”
      Definición de Header no válido.           author “Antonio C. D. B.”
      Como mínimo hay que definir un            /*
      atributo.                                    Atributo duplicado.
     */                                            El segundo atributo author se omite
}                                                  y sólo se tiene en cuenta el primer
                                                   atributo.
                                                */
                                            }
//Definición no válida del header dentro de la definición de un componente en DL.
component SimpleComponent
{
        header
        {
            author “Francisco J. Santana Jorge”;
            version “1.1.0”
        };
        header
        {
            version “1.2.0”
            /*  Definición duplicada del Header.  Se produce un error.   */
        };
        entry state Main
        {
            transition on empty_transition;
        }
};
```

Fragmento de Código 27: Ejemplos correctos e incorrectos de header.

4.1.7. Definición de 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. Como veremos más adelante, esta propiedad nos resultará muy útil ya que nos permitirá definir longitudes de paquetes de puertos, números de intentos de recuperación en las excepciones,… de modo que podemos modificar dichos valores sin tener que ir a cada línea donde se definen dichos campos. El formato de definición de constantes es el siguiente:

```
/* Parte de declaración de constantes. */
constants
{
        /* Parte de definición de constantes */
        <<constante>>
}
```

Esta construcción es opcional y se puede definir cuantas veces se requieran pero siempre que se defina tendremos que especificar al menos una constante en cada sentencia *constants*.

Siempre y cuando no estén duplicadas, en la parte de definición, podemos definir cuantas constantes deseemos y se distinguen dos tipos de constantes: las constantes públicas y las constantes privadas.

Las constantes especificadas como públicas simplemente especifican 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.

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*.

Después de especificar el tipo se especifica el nombre del identificador de la constante, que no debe estar repetido, seguido del símbolo “=” y del valor de la constante que como ya hemos dicho es un valor literal numérico o cadena.

En el Fragmento de Código 28 podemos observar un ejemplo donde se definen tres constantes correctamente y otro dos ejemplos donde se definen constantes de forma incorrecta.

```
/* Definiciones correctas de constantes. */
constants
{
       FRECUENCE = “A::Frecuence”; // Definición de constante pública.
       private LENGHT_PORT = -80; // Definición de constante privada.
       public RANGE_PRIORITIES = 3; // Definición de constante pública.
}
/* Definiciones incorrectas de constantes. */
constants
{
    /* Definición no válida.
       Como mínimo hay que definir
       una constante.  */
}
```

```
constants
{
     ATTEMPTS  = 20;
};
constants
{
    ATTEMPTS  = 20; //Constante duplicada.
};
```

Fragmento de Código 28: Ejemplos correctos e incorrectos de constantes en DL.

4.1.7.1. Sintaxis XML de las constantes.

En un documento DLXML, las constantes se definen empleándose la siguiente sintaxis:

<!-- Parte de declaración de constantes. -->

<constant>

<!-- Parte de definición de constantes -->

<<atributos de la constante>>

</constant>

donde los atributos se especifican de la siguiente manera:

- *atributo de acceso*: indica si la constante se define como pública o como privada. Se define empezando con la etiqueta *<access>* y terminando con la etiqueta *</access>*. En medio de esas dos etiquetas sólo pueden existir dos valores: el valor *public* o el valor *private*.
- *atributo nombre*: especifica el nombre de la constante y su definición empieza con la etiqueta *<name>* y termina con la etiqueta </name>. En medio de esas dos etiquetas sólo se puede definir un identificador.
- *atributo valor*: especifica el valor que se asignará a la constante. La definición comienza con la etiqueta *<value>* y termina con la etiqueta *</value>*. En medio

de estas dos etiqueta sólo se puede especificar o un valor literal cadena o un valor literal numérico.

En el Fragmento de Código 29 podemos ver un ejemplo de definición de constantes en DL con su correspondiente traducción a sintaxis XML.

```
// Definiciones correctas de constantes con su correspondiente traducción a XML.
constants                                          <constant>
{                                                      <access>public</access>
    FRECUENCE = “A::Frecuence”;                        <name>FRECUENCE</name>
    private LENGHT_PORT = -80;                         <value>"A::Frecuence"</value>
    public RANGE_PRIORITIES = 3;                   </constant>
}                                                  <constant>
                                                       <access>public</access>
                                                       <name>RANGE_PRIORITIES</name>
                                                       <value>3</value>
                                                   </constant>
                                                   <constant>
                                                       <access>private</access>
                                                       <name>LENGHT_PORT</name>
                                                       <value>-80</value>
                                                   </constant>
```

Fragmento de Código 29: Definiciones correctas de constantes en DL y en sintaxis XML.

4.1.8. Definición de puertos.

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 unas conexiones denominadas *conexiones de puertos* por donde se transmiten datos agrupados en unidades de información llamadas *paquetes de puertos* que pueden tener distintos tipos de datos unidos. La definición de un puerto en DL sigue el siguiente formato:

<<clase>> <<identificador>> **type** *<<tipo de puertos>> <<definición centinela>>*

A continuación pasaremos a describir cada uno de los elementos que se emplean en el formato de definición de puertos.

4.1.8.1. Clase de puerto.

Especifica el tipo y la clase de puerto que vamos a definir y se especifica de la siguiente manera: <<*tipo*>> <<*clase*>> donde tipo indica si el puerto se define como público o privado y clase si el puerto es de entrada o de salida.

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*.

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 siguiente construcción formada por dos palabras reservadas: *input port* y para definir un puerto de salida se utiliza esta otra construcción: *output port*.

4.1.8.2. Identificador de puertos.

Especifica el nombre que se le asignará al puerto que estamos definiendo. Este nombre no puede coincidir con ningún nombre de puerto que se haya especificado anteriormente o que coincida con algún puerto definido por defecto en la plataforma CoolBOT. Tampoco puede coincidir con el nombre dado a una constante. En la misma definición, podemos definir varios nombres de puertos separando cada nombre con el símbolo “,”.

4.1.8.3. 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 que mecanismo y política se empleará para manejar los datos que se envían o se reciben.

En la siguiente Tabla 8, podemos ver los tipos de paquetes que podemos definir en DL según la clase de puertos que vayamos a definir:

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 8: Tipos de puertos que podemos definir en DL.

Si comparamos la Tabla 8 con la Tabla 2 del apartado 2.5, observamos que en DL los tipos de puertos se especifican sin el prefijo ***I*** de entrada y sin el prefijo ***O*** de salida. También podemos observar que en DL no hay soporte para el tipo de puerto ***IShared*** y ***OShared*** de la plataforma CoolBOT. La razón es que este tipo de puerto esta en desuso y en un futuro será eliminado de la plataforma.

Tal como se puede apreciar en esta Tabla 8, 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.

Las conexiones de puertos que podemos establecer en DL son las mismas que podemos apreciar en la Figura 7 del apartado 2.5. Para obtener más información sobre los tipos de puertos y las conexiones de puertos que se pueden establecer entre ellos consultar [Domínguez-Brito, 2003].

Como hemos dicho, a cada tipo se le pueden asociar cero o más paquetes de puertos que es la unidad de información que se emplea para transmitir datos a través de los 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 referencian a través de un identificador y se tratan de una serie de paquetes de puertos definidos y suministrador por la plataforma CoolBOT que podemos ver en la Tabla 3 del apartado 2.5. Todo identificador que denote un paquetes 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 especificando 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 paquetes de puertos corresponderá a uno de los casos anteriormente mencionados. Por ejemplo, en el Fragmento de Código se define un puerto de entrada de tipo Last cuyo paquete de puertos esta definido en el componente A.

```
component SimpleComponent
{
   input port miPuerto type last port packet A::MiPaquete;
   entry state Main
   {
     transition on miPuerto;
   }
};
```

Fragmento de Código 30: Ejemplo de paquetes de puertos definidos en otro componentes.

- *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 esta 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 paquetes 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.

Opcionalmente, a los tipos de puertos se le pueden asociar un valor numérico que en los tipos *last, fifo, ufifo, poster* y *generic* indican la longitud que poseen y en los tipos *multipacket, lazymultipacket* y *pull* indican el número de tipos de paquetes de puertos que pueden transmitir. Los tipos *last, generic, priority* y *priorities* no admiten la definición de este valor.

Si no se especifica este valor, en el primer caso se toma como valor por defecto 1 y el último caso se toma como valor el número de paquetes de puertos asociados a dicho tipo. El valor que se especifica debe ser o un literal numérico o una constante.

Los tipos *priority* y *priorities* son dos tipos de puertos especiales que permiten trabajar con prioridades y que se definen conjuntamente ya que a un puerto de tipo *priorities* sólo se le puede asociar un puerto de tipo *priority* y viceversa.

Tanto al tipo *priority* como *priorities* se les asocia un valor que indica el rango de prioridades sobre las cuales pueden trabajar y además, al tipo *priorities* se le asocia, en función del rango, uno o más valores de tiempo de espera en milesegundos. Los valores pueden ser literales numéricos o una constante y si se especifica más de uno se emplea el símbolo "," como separador.

Otro tipo especial es el tipo *pull* 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.

A continuación y en base a lo comentado, pasamos a ver los distintos formatos de definición de los tipos de puertos:

- *Tipo tick*: simplemente, como se aprecia en el Fragmento de Código 31, se especifica la palabra reservada *tick.*

```
/* Puertos de entrada de tipo tick. */
input port A type tick;
private input port B type tick;
/* Puertos de salida de tipo tick. */
output port C type tick;
private output port D type tick;
```

Fragmento de Código 31: Ejemplos de puertos de tipo *tick.*

- *Tipos fifo, ufifo y poster*: el formato varía en función si se especifica o no el valor de longitud:

  <<*tipo puerto*>> **port packet**

  <<*paquetes de puertos*>> [**length** <<*valor*>>]

  donde en función del tipo de puerto, <<*tipo de puerto*>> se cambia por una de las siguientes palabras reservadas: *fifo, ufifo* o *poster*. En el Fragmento de Código 32 podemos ver una serie de ejemplos de estos tipos de puertos.

```
/* Puertos de entrada de tipo fifo. */
input port A type fifo port packet PacketLong;
private input B port type fifo port packet PacketInt;
input port C type fifo port packet PacketDouble length 90;
/* Puertos de entrada de tipo ufifo. */
input port type D port packet PacketLong;
private input port F type ufifo port packet PacketInt;
input port G type ufifo port packet PacketDouble length 90;
/* Puertos de entrada de tipo poster. */
input port H type poster port packet PacketLong;
private input port I type poster port packet PacketInt;
input port J type poster port packet PacketDouble length 90;
/* Puertos de salida de tipo poster. */
output port I type poster port packet PacketLong;
private output port J type poster port packet PacketInt;
output port K type port poster packet PacketDouble length 90;
```

Fragmento de Código 32: Ejemplos de puertos de tipo *fifo*, *ufifo* y *poster*.

- *Tipo priority*: el formato es el siguiente:

  priority port packet <<*paquetes de puertos*>> **range** <<*valor*>>

  donde <<*valor*>> es un valor literal numérico positivo o una constante. En el Fragmento de Código 33 podemos ver ejemplos de definición de este tipo de paquete.

```
/* Puertos de entrada de tipo prioriy. */
output port A type priority port packet PacketLong range 2;
private output B port type priority port packet PacketInt range VALOR;
```

Fragmento de Código 33: Ejemplos de puertos de tipo *priority*.

- *Tipo priorities*: el formato es el siguiente:

  priorities port packet <<*paquetes de puertos*>>

  range <<*valor*>> **with timeouts** { <<*valores*>> }

  A continuación podemos ver en el Fragmento de Código 34 unos ejemplos:

```
/* Puertos de salida de tipo priorities. */
intput port A type priorities port packet PacketLong
                                range 2 with timeouts { 100, VALOR };
private intput B port type priorities port packet PacketInt
                                 range VALOR with timeouts { 300 };
```

Fragmento de Código 34: Ejemplos de puertos de tipo *priorities*.

- *Tipo multipacket y lazymultipacket*: el formato varía en función si se especifica o no el valor de longitud:

  <<*tipo de puerto*>> **port packets** { <<*paquetes de puertos*>> }

  [**length** <<*valor*>>]

donde en función del tipo de puerto, <<*tipo de puerto*>> se cambia o por la palabra reservada *multipacket* o por *lazymultipacket*. En el Fragmento de Código 35 podemos ver ejemplos de estos tipos de puertos.

```
/* Puertos de entrada de tipo multipacket. */
input port A type multipacket port packets  { PacketLong };
private input B port type multipacket port packets  { PacketInt, PacketLong };
input port C type multipacket port packets  { PacketDouble } length 1;
/* Puertos de salida de tipo multipacket. */
output port D type multipacket port packets  { PacketLong };
private output E port type multipacket port packets  { PacketInt, PacketLong };
output port F type multipacket port packets  { PacketDouble } length 1;
/* Puertos de salida de tipo lazymultipacket. */
output port G type lazymultipacket port packets  { PacketLong };
private output H port type lazymultipacket port packets  { PacketInt, PacketLong };
output port I type lazymultipacket port packets  { PacketDouble } length 1;
```

Fragmento de Código 35: Ejemplos de puertos de tipo *multipacket* y *lazymultipacket.*

- *Tipo last y generic*: el formato es el siguiente:

  <<*tipo de puerto*>> **port packet** <<*paquetes de puertos*>>

  donde en función del tipo de puerto, <<*tipo de puerto*>> se cambia o por la palabra reservada *last* o por *generic*. En el Fragmento de Código 36 podemos ver ejemplos de estos tipos de puertos.

```
/* Puertos de entrada de tipo last. */
input port A type last port packet PacketLong;
private input B port type last port packet PacketInt;
/* Puertos de salida de tipo generic. */
output port D type generic port packetsPacketLong;
private output E port type generic port packetPacketInt;
```

Fragmento de Código 36: Ejemplos de puertos de tipo *last* y *generic*.

- *Tipo pull*: el formato varía en función si se especifica o no el valor de longitud:

  pull port packets {

  <<*tipo*>> <<*paquetes de puertos*>>,

  <<*tipo*>> <<*paquetes de puertos*>>

  } [**length** <<*valor*>>]

  donde <<*tipo*>> indica si el tipo de puerto es para petición o para respuesta. Si es para petición se emplea la palabra reservada *answer* y para la respuesta se utiliza *request*. Los dos paquetes de puertos no pueden ser a la vez ni de petición ni de respuesta. Se tiene que definir uno de petición y otro de respuesta. El orden de definición no importa. En el Fragmento de Código 37 podemos ver ejemplos de estos tipos de puertos.

```
/* Puertos de entrada de tipo pull. */
input port A type pull port packets
                      { answer PacketLong, request PacketLong };
private input B port type pull port packets
                             { request PacketInt, answer PacketLong };
input port C type pull port packets
                      { request PacketInt, answer PacketLong } length 2;
/* Puertos de salida de tipo pull. */
output port D type pull port packets
                       { answer PacketLong, request PacketLong };
private output E port type pull port packets
                              { request PacketInt, answer PacketLong };
output port F type pull port packets
                       { request PacketInt, answer PacketLong } length 2;
```

Fragmento de Código 37: Ejemplos de puertos de tipo *pull*.

4.1.8.4. 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 es el siguiente.

with watchdog each <<*valor*>>

donde <<*valor*>> especifica un valor numérico que indica el periodo de actuación en milisegundos del centinela. Este valor puede ser un valor literal numérico o una constante. En el Fragmento de Código 38 podemos ver ejemplos de definición de centinelas.

```
/* Puertos de entrada con centinelas. */
input port A type pull port packets
                      { answer PacketLong, request PacketLong }
                      with watchdog each 1000;
private input B port type tick with watchdog each VALOR;
```

Fragmento de Código 38: Ejemplos de puertos con centinelas.

4.1.8.5. Sintaxis XML de los puertos.

Los puertos se definen en un documento DLXML a través del siguiente formato:

<!-- Parte de declaración de puertos. -->

<port>

<!-- Parte de definición de puertos. -->

<<atributo de acceso de puerto>>

<<atributo de puerto>>

<<atributo identificador de puerto>>

<<atributo tipo de puerto>>

<<atributo centinela de puerto>>

</port>

A continuación veremos como se especifican cada uno de los atributos que definen un puerto:

- *atributo de acceso de puerto*: indica si el puerto se ha definido como público o privado. La definición comienza con la etiqueta *<access>* y termina con la etiqueta *</access>* y en medio de esas etiquetas sólo se pueden especificar los valores de *public* o de *private*.
- *atributo de puerto*: indica si el puerto se define de entrada o de salida. La definición comienza con la etiqueta *<class>* y termina con la etiqueta *</class>*. En medio de esas dos etiquetas sólo pueden aparecer los valores de *output* o de *input*.
- *atributo identificador de puerto*: indica el nombre del puerto. La definición comienza con la etiqueta *<name>* y termina con la etiqueta *</name>*. En medio de estas dos etiquetas sólo pueden aparecer un identificador.
- *atributo tipo de puerto*: indica el tipo de paquete que asociamos al puerto que se esta definiendo. El formato de definición es el siguiente:

  <type>

  <<atributo identificador de tipo de puerto>>

  <<atributo de paquetes de puerto>>

  <<atributo rango de prioridades>>

  <<atributo longitud>>

  </type>

donde:

1. *atributo identificador de tipo de puerto*: especifica el identificador que identifica al tipo de paquete que se esta definiendo. La definición de este atributo comienza con la etiqueta <*name*> y termina con la etiqueta </*name*>. En medio de estas dos etiquetas sólo podemos definir un identificador que debe coincidir con alguno de los especificados en la Tabla 8.
2. *atributo de paquetes de puertos*: especifica, si el tipo o tipos asociados al puerto. El formato que emplea es el siguiente:

   <portpackets>

   <<*atributo identificador de paquetes de puertos*>>

   <<*atributo referencia de paquetes de puertos*>>

   <<*atributo clase de paquetes de puertos*>>

   </portpackets>

   El atributo identificador de paquetes de puertos se define empleando la etiqueta <*name*> seguido de un identificador y terminado con la etiqueta </*name*>. Este atributo especifica el identificador de paquetes de puertos que estamos definiendo.

   El atributo referencia de paquetes de puertos específica el nombre del componente donde se especificó el paquetes de puertos que estamos definiendo. Se define empleando la etiqueta <*reference*> seguido de un identificador y terminado con la etiqueta </*reference*>.

   El atributo clase de paquetes de puertos sólo se especifica cuando vamos a definir un puerto de tipo *Pull* y define si el paquetes de puertos que se especifica es de petición o de respuesta. Se define empleando la etiqueta <*class*> seguido del valor *answer* o del valor *request* y terminado con la etiqueta </*class*>.
3. *atributo rango de prioridades*: Este atributo sólo se define cuando estamos definiendo un puerto de tipo *Priorities* o *Priority*.

   Tanto si el puerto es de tipo *Priorities* o *Priority*, hemos de definir un atributo que especifica un rango de prioridades sobre las que trabaja. Esto se define empleando la etiqueta <range> seguida de una constante o un valor literal numérico y terminado con la etiqueta </range>.

Si el puerto es de tipo *Priorities*, aparte de definir el rango de prioridades, hemos de especificar, además, un listado de valores de periodos en milisegundos con los que se refresca cada nivel de prioridad. Estos valores se especifican siguiendo el siguiente formato:

<priorities>

<<atributo valor>>

</priorities>

donde el atributo valor se define empleando la etiqueta *<value>* seguido de un literal cadena o de un literal numérico y terminado con la etiqueta *</value>*.

4. *atributo longitud*: indica la longitud de los paquetes de puertos que se puede recibir o transmitir. Se define empleando la etiqueta *<length>* seguido de una constante o de un valor numérico y terminado con la etiqueta *</length>*.

- *atributo centinela de puerto*: indica si el puerto tiene o no asociado un centinela. La definición empieza con la etiqueta *<watchdog>* y termina con la etiqueta *</watchdog>*. En medio de esas etiquetas sólo se puede especificar un valor literal numérico o una constante que especifica el periodo en milisegundos con el que se refresca el centinela.

En el Fragmento de Código 39, podemos ver un ejemplo en donde se define el puerto de salida *Value* con su correspondiente sintaxis XML.

```
output port Value type generic      <port>
        port packet A::PacketOng;       <access>public</access>
                                        <class>output</class>
                                        <name>Value</name>
                                        <type>
                                            <name>generic</name>
                                            <portpackets>
                                               <name>PacketOng</name>
                                               <reference>A</reference>
                                            </portpackets>
                                        </type>
                                    </port>
```

Fragmento de Código 39: Definición del puerto de salida *Value* en sintaxis XML.

4.1.9. Definición de variables observables y 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. Para definir dichas variables se emplea la siguiente sentencia:

/ Parte de declaración de variables. */*
<<tipo de variable>>
{
/ Parte de definición de variables. */*
<<definición de variable>>
}

4.1.9.1. Tipo de variable.

Específica que el tipo de variable que se va a definir. Si queremos definir variables observables se emplea la siguiente construcción formada por dos palabras reservadas: *observable variables* y si se trata de variables controlables se utiliza esta otra construcción: *controllable variables.*

4.1.9.2. Definición de variable.

Define una variable ya sea controlable u observable. Como mínimo debe haber una definición de variable en una sentencia de definición de variables y cada variable se define según el siguiente formato:

<<identificador>> **: port packet** *<<paquetes de puertos>>*

Cada variable se identifica a través de un nombre de identificador único. No pueden existir dos variables con el mismo nombre aunque sean de tipos diferentes y el nombre de la variable tampoco puede coincidir con el nombre dado a una constante o puerto. En una misma definición podemos definir varias variables empleando el símbolo “,” como separador.

A parte de un identificador debemos especificar el tipo de paquetes de puertos que tiene dicha variable. Este aspecto es lo que define la variable y los paquetes de puertos se especifican tal y como se vio en el apartado 4.1.8.3.

En el siguiente Fragmento de Código 40 se aprecian varias definiciones de variables tanto observables como controlables.

```
// Definición de variables observables.
observable variables
{
     A, B, C: port packet PacketInt;
     D: port packet MiComponente::Paquete1;
     E: port packet Paquete;
};
// Definición de variables controlables.
controllable variables
{
    AC, BC, CC: port packet PacketInt;
    DC: port packet MiComponente::Paquete1;
    EC: port packet Paquete;
};
```

Fragmento de Código 40: Ejemplo de definiciones de variables.

4.1.9.3. Sintaxis XML para las variables.

En un documento DLXML, las variables se definen siguiendo el formato siguiente:

/ Parte de declaración de variables. */*

<variable>

/ Parte de definición de variables. */*

<<atributo tipo de variable>>

<<atributo identificador de variable>>

<<atributo paquetes de puertos>>

</variable>

Donde:

- *atributo tipo de variable*: específica que tipo de variable vamos a definir. Se define utilizando la etiqueta <*class*> seguido del valor *observable* o *controllable* y terminado con la etiqueta </class>.
- *atributo identificador de variable*: se define empleando la etiqueta <*name*> seguido de un identificador y terminado con la etiqueta *</name>*.

- *atributo paquetes de puertos*: indica el paquete de puertos asociado a la variable y se sigue el mismo formato especificado en el apartado 4.1.8.5. La única diferencia es que el atributo clase de paquetes de puertos nunca es aplicable en este caso.

En el Fragmento de Código 41 podemos apreciar la definición de una variable controlable denominada *NEW_PERIOD* en sintaxis XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component>
          <name>SimpleComponent</name>
          <variable>
                    <class>controllable</class>
                    <name>NEW_PERIOD</name>
                    <portpackets>
                              <name>PacketLong</name>
                              <reference>FirstComponent</reference>
                    <portpackets>
          </variable>
          <state>
               <class>entry</class>
               <name>Main</name>
               <transition>
                    <name>empty_transition</name>
                    <source>Main</source>
                    <target>Main</target>
               </transition>
          </state>
</component>
```

Fragmento de Código 41: Ejemplos de definiciones de variables en sintaxis XML.

4.1.10. Definición de 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 [Domínguez-Brito, 2003] emplea como mecanismo, para resolver estas situaciones, las excepciones y ellas se definen en DL de la siguiente forma:

```
/* Parte de declaración de excepción. */
exception <<identificador>>
{
    /* Parte de definición de excepción. */
    <<atributos de la excepción>>
}
```

Toda excepción que queramos especificar comienza con la palabra reservada *exception* seguida de un nombre de identificador y encerrado entre llaves la definición de la excepción. A través del nombre del identificador se identifica de forma unívoca la excepción. No puede haber dos excepciones con el mismo identificador y éste no puede coincidir ni con el identificador de una constante, puerto o variable ni con el de una excepción predefinida de la plataforma CoolBOT.

Encerrado entre llaves se definen dos atributos que dan significado a la excepción. El primer atributo proporciona una descripción de la excepción. Este atributo es obligatorio y siempre que se defina una excepción debemos especificar una descripción. Para ello se utiliza la palabra reservada *description* seguido 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. 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. En el Fragmento de Código 42 podemos ver un ejemplo de definición de una excepción llamada *MiExcepcion* que define los tres manejadores de excepción vistos en este apartado.

```
// Definición de excepciones.
exception MiExcepcion
{
    description "Prueba";
    has on success handler;
    has on failure handler;
   recovery attempts 10 with period 100;
}
```

Fragmento de Código 42: Definición de la excepción *MiExcepcion* en DL.

4.1.10.1. Sintaxis XML de las excepciones.

El formato de definición de una excepción en un documento DLXML es la siguiente:

<!-- Parte de declaración de excepción. -->
<exception>
<!-- Parte de definición de excepción.-->
<<atributo identificador de la excepción>>
<<atributo descripción de la excepción>>
<<atributo manejadores de la excepción>>
</exception>

donde el *atributo identificador de la excepción* especifica el identificador que identifica la excepción y se construye con la etiqueta *<name>* seguido de un identificador y terminado con la etiqueta *</name>*. El *atributo descripción de la excepción* indica un breve comentario sobre lo que la excepción realiza. Se define comenzando con la etiqueta *<description>* seguida de un literal cadena y terminado con la etiqueta *</description>*. El *atributo manejadores de la excepción* especifica los manejadores que se han definido en la excepción para poder tratarla. Podemos distinguir tres tipos de manejadores:

- *manejador de recuperación*: se emplea para hacer que el componente se intente recuperar de la excepción que se ha producido. Se define siguiendo el siguiente formato:

  <recoveryhandler>
  <<atributo número de intentos de recuperación>>
  <<atributo periodo de recuperación>>
  </recoveryhandler>

  Donde *el atributo número de intentos de recuperación* especifica un número de intentos que el manejador realizará para poder recuperarse y poder continuar. Se define empezando con la etiqueta *<attempts>* seguida de un valor literal numérico o de una constante y terminado con la etiqueta *</attempts>*. El *atributo periodo de recuperación* indica el periodo en milisegundos con la que se inicia un intento de recuperación y se define empezando con la etiqueta

<*period*> seguido de un valor literal numérico o de una constante y terminado con la etiqueta </*period*>.

- *manejador de éxito y de fracaso*: El manejador de éxito se especifica con las etiquetas <*successhandler*> y </*successhandler*> y el manejador de fracaso con las etiquetas <*failurehandler*> y </*failurehandler*>. Entre esas estiquetas sólo pueden aparecer o el valor *true* o el valor *false.*

En el Fragmento de Código 43 podemos ver un ejemplo de un componente llamado *SimpleComponent* donde se define una excepción llamada *NotTransition* que define un manejador de recuperación, un manejador de éxito y un manejador de fracasos.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component>
          <name>SimpleComponent</name>
          <exception>
               <name>NotTransition</name>
               <description>”Se produce al no realizar  transición.”</description>
               <recoveryhandler>
                         <attempts>10</attempts>
                         <period>FRECUENCE</period>
               </recoveryhandler>
               <successhandler>true</successhandler>
               <failurehandler>true</failurehandler>
          </exception>
          <state>
               <class>entry</class>
               <name>Main</name>
               <transition>
                    <name>empty_transition</name>
                    <source>Main</source>
                    <target>Main</target>
               </transition>
          </state>
</component>
```

Fragmento de Código 43: Ejemplo de definición de una excepción en sintaxis XML.

4.1.11. Definición de estados.

La funcionalidad interna de un componente se define como un autómata de estados y todo autómata esta 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 un de ellos de entrada. En CoolBOT, el autómata de estados que determina la funcionalidad de un componente se denomina Autómata de Usuario [Domínguez-Brito, 2003].

En DL, un estado se define de la siguiente forma:

```
/* Parte de declaración de estado. */
<<tipo estado>> state <<identificador>>
{
   /* Parte de definición de estado. */
   <<transiciones del estado>>
}
```

4.1.11.1. Tipos de estados.

Especifica si el estado se define 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.

4.1.11.2. Identificador de estados.

Identifica unívocamente al estado. Es decir, no puede haber dos estados con el mismo nombre de identificador o que el nombre de identificador coincida con el de una constante, puerto, variable o excepción. Tampoco puede coincidir el nombre de

identificador con los nombres de los estados del autómata por defecto de la plataforma CoolBOT [Domínguez-Brito, 2003].

4.1.11.3. Transiciones del estado.

Específica sobre que puertos de entrada o variables controlables se producen transiciones en ese estado. Todo puerto de entrada y/o variable controlable que se especifique tiene que haberse definido previamente y en un estado no puede haber más de una transición sobre ese puerto de entrada o esa variable controlable.

También se puede definir transiciones sobre dos puertos privados predefinidos en la plataforma CoolBOT: "***timer***" y "***empty_transition***" y que podemos utilizar para realizar transiciones vacías en el componente o provocadas por un temporizador interno (*timer*) a cada componente.

No se pueden 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.

Para especificar una transición se utiliza la siguiente construcción formada por dos palabras reservadas: *transition on* seguido de uno o más nombres de identificadores de puertos de entrada o variables controlables. Si especificamos más de un nombre de identificador debemos emplear el símbolo "," como separador. La definición de transiciones es opcional ya que podemos definir estados sin transiciones. En el Fragmento de Código 44 podemos ver dos ejemplos de definiciones de estados en donde se realizan transiciones sobre los puertos *empty_transition* y *timer*.

```
// Definición del  estado de entrada.
entry state Main
{
    transition on empty_transition;
}
state Secundario
{
    transition on empty_transition;
    transition on timer;
}
```

Fragmento de Código 44: Ejemplos de definiciones de estados.

4.1.11.4. Sintaxis XML de los estados.

Los estados se especifican en un documento DLXML a través del siguiente formato:

<!-- Parte de declaración de estado.-->
<**state**>
<!-- Parte de definición de estado.-->
<<atributo clase de estado>>
<<atributo identificador de estado>>
<<atributo transición de estado>>
</state>

donde:

- *atributo clase de estado*: indica si el estado es de entrada o no y se define comenzando con la etiqueta *<class>* seguida del valor *entry* o del valor *noentry* terminando con la etiqueta *</class>*.
- *atributo identificador de estado*: especifica el identificador que identifica el estado. Se define empezando con la etiqueta *<name>* seguido de un identificador terminando con la etiqueta *</name>*.
- *atributo transición de estado*: indica una transición que se originan en ese estado. Se define especificando el puerto o variable sobre la que realiza la transición seguido del estado de partida y del estado al que se transita. El formato de definición es el siguiente:

  <**transition**>
  <<atributo puerto o variable de la transición>>
  <<atributo estado de partida>>
  <<atributo estado de llegada>>
  </transition>

  El atributo puerto o variable de la transición especifica el puerto o variable sobre el cual el estado realiza una transición. La definición comienza con la etiqueta *<name>* seguida de un identificador y terminando con la etiqueta *</name>*.

  El atributo estado de partida especifica el estado desde el que se realiza la transición. Se especifica empezando con la etiqueta *<source>* seguido de un identificador y terminado con la etiqueta *</source>*.

El atributo estado de llegada define el estado al que se llega tras la transición y se especifica empezando con la etiqueta <*target*> seguido de un identificador y terminado con la etiqueta <*/target*>. Este atributo no se encuentra definido en DL y se ha añadido a la sintaxis XML ya que es un atributo que se usara en el futuro por parte de otro Proyecto de Fin de Carrera que se encuentra bajo desarrollo.

En el Fragmento de Código 45 podemos ver un ejemplo de un componente donde se definen dos estados: uno de entrada llamado *Main* y otro llamado *FirstState* que no es de entrada.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component>
          <name>SimpleComponent</name>
          <state>
               <class>entry</class>
               <name>Main</name>
               <transition>
                    <name>empty_transition</name>
                    <source>Main</source>
                    <target>Main</target>
               </transition>
          </state>
          <state>
               <class>noentry</class>
               <name>FirstState</name>
               <transition>
                    <name>empty_transition</name>
                    <source>FirstState</source>
                    <target> FirstState </target>
               </transition>
          </state>
</component>
```

Fragmento de Código 45: Ejemplo de definiciones de estado en sintaxis XML.

4.1.12. Definición de hilos.

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. En DL se definen utilizando el siguiente formato de sentencia:

```
/* Parte de declaración del hilo. */
<<espera activa>> thread <<identificador>>
{
    /* Parte de definición del hilo. */
    <<puertos de entrada>>
    <<estados activos>>
}
```

4.1.12.1. Espera activa.

Se denomina espera activa a una técnica donde un proceso repetidamente verifica una condición. Si deseamos que 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.

4.1.12.2. Identificador de hilo.

Identifica unívocamente al hilo. Es decir, no puede haber dos hilos con el mismo nombre de identificador o que el nombre de identificador coincida con el de una constante, puerto, variable, excepción o estado.

En todo componente CoolBOT, por defecto, hay un hilo principal que se referencia utilizando el nombre de identificador *main* y que es responsable de iniciar todos los hilos de puertos del componente así como de monitorizarlos y controlarlos.

En DL, podemos definir el hilo *main* para especificar un nuevo comportamiento. Sólo podemos definirlo una vez en el componente y no admite la política de espera activa.

4.1.12.3. Puertos de entrada.

Todo hilo tiene que tener asociado como mínimo un puerto de entrada sobre el cual actuar. En función de si se especifican prioridades a los puertos de entrada o no podemos distinguir dos formas de asociación que determinan la clase de hilo que estamos definiendo.

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.

Un hilo de clase *priority input box* se define en DL siguiendo el siguiente formato:

```
/* Parte de declaración del hilo con prioridades. */
priority input box
{
    /* Parte de definición del hilo con prioridades. */
    <<prioridad puertos de entrada>>
}
```

donde <<*prioridad puertos de entrada*>> asocia una prioridad a uno o más puertos de entrada. Para ello se especifica la palabra reservad *priority* seguido de un nombre de identificador que indica el nivel de prioridad y uno o más puertos de entrada encerrados entre llaves. Es decir:

priority <<*identificador prioridad*>> { <<*puertos de entrada*>> }

El nombre *identificador de prioridad* es único en el componente, es decir, no se puede definir una prioridad dos veces en el mismo hilo o utilizar la misma prioridad en otro hilo. Tampoco puede coincidir con el identificador de una constante, puerto, variable, excepción, estado o hilo.

A una prioridad podemos asociarle uno o más puertos de entrada. Si especificamos más de un puerto se utiliza el símbolo “,” comos separador. Una vez asociado una prioridad a un puerto de entrada, no podemos volver a utilizar ese puerto ni en otro hilo ni en el mismo 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.

Un hilo de clase *input box* se define en DL con el siguiente formato:

input box { <<*puertos de entrada*>> }

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 DL existe una regla que obliga 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 automáticamente al hilo *main*.

4.1.12.4. Estados activos.

Opcionalmente podemos especificar sobre que estados del autómata de usuario está activo cada hilo. Los estados especificados deben haberse definido en el componente. Sólo el hilo *main*[7] puede estar activo en los estados del Autómata por Defecto de la plataforma CoolBOT.

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

En el Fragmento de Código 46 podemos ver la definición del hilo *main* y de otro hilo denominado *First_Thread*. En el hilo *main* se han definido dos prioridades *basic* y *high*.

```
// Definición de hilos.
thread Main // Definición del hilo Main.
{
        priority input box
        {
              priority basic {iPublicPull, oPublicPull};
              priority high {iPrivatePull, oPrivatePull};
        };
}
polling thread First_Thread
{
       input box {iPublicMultipacket};
       active in RestComponent;
}
```

Fragmento de Código 46: Ejemplo de definiciones de hilos.

[7] En DL, el identificador de hilo empleado para referirse al hilo *main* no es sensible a mayúsculas y minúsculas por lo que es correcto referirse a dicho hilo con los identificadores Main, MAIN, main,….

4.1.12.5. Sintaxis XML de los hilos.

El formato de definición de los hilos en sintaxis XML es la siguiente:

<!-- Parte de declaración del hilo.-->
<thread>
<!-- Parte de definición del hilo.-->
<<atributo espera activa>>
<<atributo identificador de hilo>>
<<atributo puertos asociados al hilo>>
<<atributo estados asociados al hilo>>
</thread>

El *atributo espera activa* específica si el hilo sigue o no la política de espera activa. Se define empezando con la etiqueta *<polling>* seguida del valor *true* o del valor *false* y terminado con la etiqueta *</polling>*.

El *atributo identificador del hilo* define el identificador que identifica al hilo y se especifica empezando con la etiqueta *<name>* seguida de un identificador y terminado con la etiqueta *</name>*.

El *atributo puertos asociados al hilo* especifica los puertos de entrada que están asociados al hilo. Hay que recordar que hay dos formas de especificar dichos puertos en función si se define el hilo con o sin prioridades. Si definimos el hilo sin prioridades los puertos se definen siguiendo el siguiente formato:

<inputbox>
<<atributo identificador de puerto>>
</inputbox>

donde el *atributo identificador de puerto* especifica el puerto de entrada al que está asociado el hilo. Se define empezando con la etiqueta *<inputport>* seguida de un identificador y terminado con la etiqueta *</inputport>*.

Si definimos el hilo con prioridades los puertos se definen siguiendo el siguiente formato:

<priorityinputbox>
<<atributo prioridad de puerto>>
</priorityinputbox>

El *atributo prioridad de puerto* específica una prioridad a la cual están asociados uno o más puertos de entrada. Por cada prioridad que definíamos debemos emplear el siguiente formato:

<priority>

<<*atributo identificador de la prioridad*>>

<<*atributo identificador de puerto*>>

</priority>

donde el *atributo identificador de la prioridad* especifica un identificador que identifica la prioridad que hemos asociado uno o más puertos. Se define empezando con la etiqueta <*name*> seguida de un identificador y terminado con la etiqueta </*name*>. El *atributo identificador de puerto* se define de la misma forma que en el caso de los hilos sin prioridades.

Por último, *el atributo estados asociados al hilo* especifica en que estados el hilo se encuentra activo. Cada uno se define comenzando con la etiqueta <*stateactive*> seguido de un identificador y terminado con la etiqueta </*stateactive*>.

En el Fragmento de Código 47 podemos ver un ejemplo de definición de hilos en sintaxis XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component>
          <name>SimpleComponent</name>
          <port>
                   <access>public</access>
                   <class>input</class>
                   <name>Period</name>
                   <type>
                         <name>last</name>
                         <portpackets>
                                  <name>PacketLong</name>
                                  <reference>FirstComponent</reference>
                         </portpackets>
                   </type>
          </port>
         <thread>
           <polling>false</polling>
               <name>Main</name>
               <priorityinputbox>
                          <priority>
                                  <name>basic</name>
                                   <inputport>Period</inputport>
                          </priority>
                </priorityinputbox>
          </thread>
          <state>
               <class>entry</class>
               <name>Main</name>
               <transition>
                    <name>Period</name>
                    <source>Main</source>
                    <target>Main</name>
               </transition>
          </state>
</component>
```

Fragmento de Código 47: Ejemplo de definiciones de hilos en sintaxis XML.

4.1.13. Palabras reservadas y etiquetas XML.

En la Tabla 9 podemos ver todas las palabras reservadas que emplea DL y en la Tabla 10 se muestra una tabla resumen con todas las etiquetas que podemos usar para definir un documento DLXML:

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

Tabla 9: Palabras reservadas de DL.

Etiqueta de comienzo	Etiqueta de terminación	Etiqueta de comienzo	Etiqueta de terminación
<component>	</component>	<type>	</type>
<header>	</header>	<portpackets>	</portpackets>
<constant>	</constant>	<reference>	</reference>
<port>	</port>	<range>	</range>
<variable>	</variable>	<priorities>	</priorities>
<exception>	</exception>	<length>	</length>
<state>	</state>	<watchdog>	</watchdog>
<thread>	</thread>	<recoveryhandler>	</recoveryhandler>
<author>	</author>	<attempts>	</attempts>
<description>	</description>	<period>	</period>
<institution>	</institution>	<successhandler>	</successhandler>
<version>	</version>	<failurehandler>	</failurehandler>
<transition>	</transition>	<source>	</source>
<polling>	</polling>	<target>	</target>
<name>	</name>	<inputbox>	</inputbox>
<access>	</access>	<inputport>	</inputport>
<value>	</value>	<priorityinputbox>	</priorityinputbox>
<class>	</class>	<priority>	</priority>
<stateactive>	</stateactive>		

Tabla 10: Etiquetas empleadas en un documento DLXML.

4.1.14. Ejemplo de un componente CoolBOT en DL.

En el Fragmento de Código 48 podemos ver la definición en DL de un componente muy simple al que llamaremos *SimpleComponent* que posee un puerto de entrada de usuario llamado *period*, un puerto de salida llamado *value*, una variable controlable llamada *new_Period* y un sólo estado que se llama *Main*.

El componente simplemente publica un número a través del puerto *value* con la frecuencia que especifica el puerto period cuyo valor de frecuencia podremos modificar a través de la variable controlable new_Period. En el Fragmento de Código 49 y 50 podemos ver misma definición componente pero en sintaxis XML.

```
component SimpleComponent
{
   header
   {
       author “Francisco J. Santana Jorge.”;
       description “Componente de aprendizaje”;
       institution “ULPGC”;
       version “1.1.0”
   };
   output port value type generic port packet PacketInt;
   controllable variables
   {
       new_Period: port packet PacketInt;
   };
   entry state Main
   {
       transition on period;
   };
   input port period type last port packet PacketInt;
};
```

Fragmento de Código 48: Ejemplo de un componente CoolBOT escrito en DL.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component>
    <name>SimpleComponent</name>
    <header>
          <author>"Francisco J. Santana Jorge."</author>
          <description>"Componente de aprendizaje"</description>
          <institution>"ULPGC"</institution>
          <version>"1.1.0"</version>
    </header>
```

Fragmento de Código 49: Ejemplo de un componente DL escrito en sintaxis XML**.**

```
        <port>
            <access>public</access>
            <class>output</class>
            <name>value</name>
            <type>
                <name>generic</name>
                <portpackets>
                    <name>PacketInt</name>
                    <reference>SimpleComponent</reference>
                </portpackets>
            </type>
        </port>
        <port>
            <access>public</access>
            <class>input</class>
            <name>period</name>
            <type>
                <name>last</name>
                <portpackets>
                    <name>PacketInt</name>
                    <reference>SimpleComponent</reference>
                </portpackets>
            </type>
        </port>
        <variable>
            <class>controllable</class>
            <name>new_Period</name>
            <portpackets>
                <name>PacketInt</name>
                <reference>SimpleComponent</reference>
            </portpackets>
        </variable>
        <state>
            <class>entry</class>
            <name>Main</name>
            <transition>
                <name>Period</name>
                <source>Main</source>
                <target>Main</target>
            </transition>
        </state>
    </component>
```

Fragmento de Código 50: Continuación del ejemplo de un componente DL escrito en sintaxis XML.

4.2. Requisitos del software.

El objetivo principal de este proyecto es desarrollar un compilador en línea de comandos que a partir del lenguaje descriptivo DL descrito en el apartado 4.1, genere esqueletos C++ con la estructura CoolBOT descrita en el apartado 2.8. Además de generar esqueletos C++, también debe generar sintaxis XML y a partir de ella obtener esqueletos C++.

Por tanto, en este trabajo se van a desarrollar dos compiladores, uno llamado *coolbotc* que a partir de un fichero fuente escrito en lenguaje DL genere esqueletos C++ y XML y otro llamado *xmlcoolbotc* que a partir de un fichero fuente escrito en sintaxis XML genere esqueletos C++. En la Figura 15 podemos apreciar esta idea.

En este apartado vamos a estudiar las necesidades que requieren cada compilador para su desarrollo.

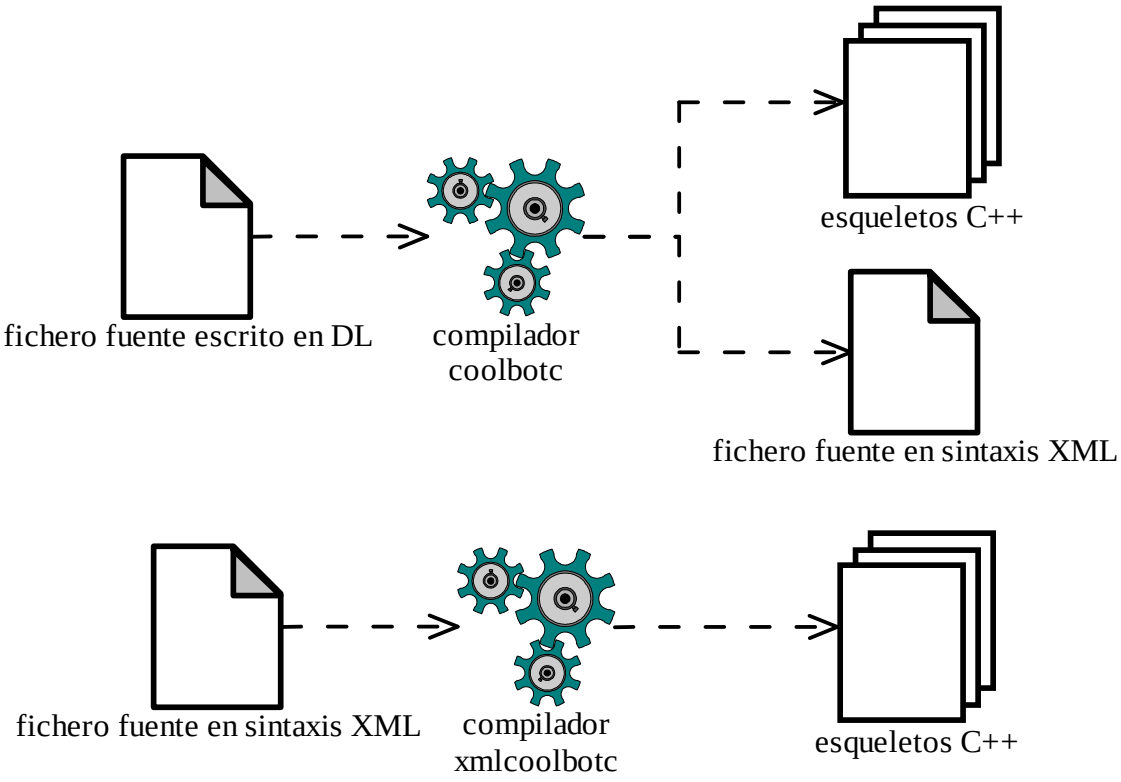


Figura 15: Compiladores a desarrollar en este proyecto.

4.2.1. Requisitos funcionales.

Para estudiar los requisitos funcionales de cada compilador vamos a emplear los *diagramas de casos de uso* [Larman, 1999]. Los diagramas de casos de uso se emplean para visualizar el comportamiento del sistema, una parte de el o de una sola clase, de forma que se pueda conocer como se comporta esa parte del sistema. En un diagrama de casos de uso se emplean los siguientes términos cuya representación gráfica se muestra en la Figura 16:

- *Actor*: se puede definir como el rol o función que asume una persona, sistema o entidad que interactúa con el sistema que estamos construyendo.
- *Caso de uso*: es una técnica para la captura de requisitos de un sistema software. Cada caso de uso proporciona uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para conseguir un objetivo específico.
- *Relaciones entre casos de uso*: Los casos de uso pueden tener relaciones con otros casos de uso mediante las siguientes relaciones:
    1. *Include*: indica que ese caso de uso necesita de otros para resolverlo.
    2. *Extends*: extiende una operación. Una relación de un caso de uso A hacia un caso de uso B indica que el caso de uso B implementa la funcionalidad del caso de uso A.
    3. *Uses*: indica que ese caso de uso puede acceder a otras operaciones. Se recomienda utilizar cuando se tiene un conjunto de características que son similares en más de un caso de uso y no se desea mantener copiada la descripción de la característica.

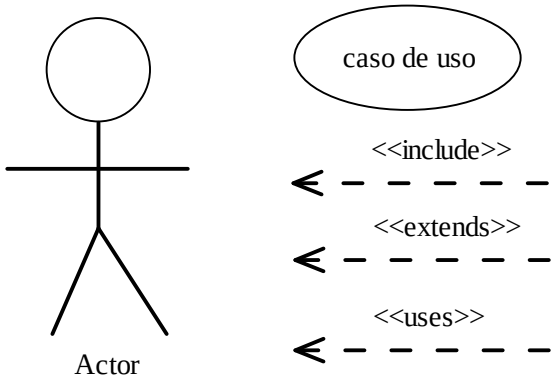


Figura 16: Elementos de un diagrama de casos de uso.

Por tanto, a través de los diagramas de casos de uso veremos las distintas opciones que proporciona el compilador al programador para realizar compilaciones.

Hemos denominado *programador* a la persona que utiliza el compilador, por tanto el programador será nuestro *Actor* en los diagramas de casos de uso. En la Figura 17 podemos ver la representación gráfica del actor programador.

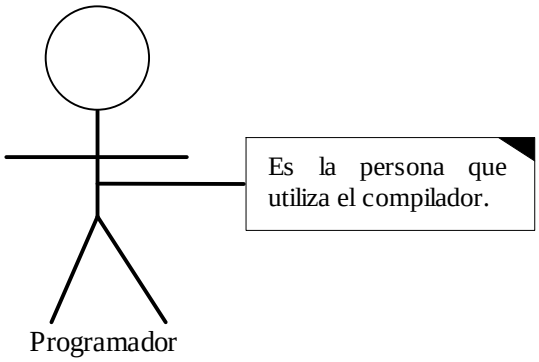


Figura 17: Actor programador.

Una vez identificado el actor del sistema software (compilador) que se va a desarrollar, en la Figura 18 pasamos a ver los respectivos casos de uso:

- *Ver ayuda*. El programador en todo momento puede consultar las opciones con las que se puede invocar el compilador. Si se invoca al compilador de forma incorrecta se muestra esta información.
- *Compilar fichero*. El programador podrá compilar ficheros tanto en lenguaje DL (compilador *coolbotc*) como en lenguaje XML (compilador *xmlcoolbotc*). La funcionalidad de este caso de uso se puede extender con los siguientes casos de uso:
    1. *Activar modo verbose en pantalla*. El programador puede querer activar el modo verbose (mostrar el progreso de la tarea que realiza) durante la compilación del fichero fuente y que se muestre el contenido en la pantalla.
    2. *Activar modo verbose en fichero*. Igual que el anterior caso de uso pero la información no se muestra en pantalla sino que se almacena en un fichero.
- *Asignar ruta compilador*. El programador, para el proceso de compilación, debe poder especificarle al compilador la ruta donde este se encuentra. En el Capítulo 5 de este documento se explica mejor este caso de uso.
- *Estructura de directorios CoolBOT para un componente*. El programador, para poder compilar el componente desarrollado en la plataforma CoolBOT, debe

crear toda la estructura de ficheros y directorios (visto en el apartado 1.1) que exige la plataforma CoolBOT para un componente. Este caso de uso no estará disponible en el compilador *xmlcoolbotc*.

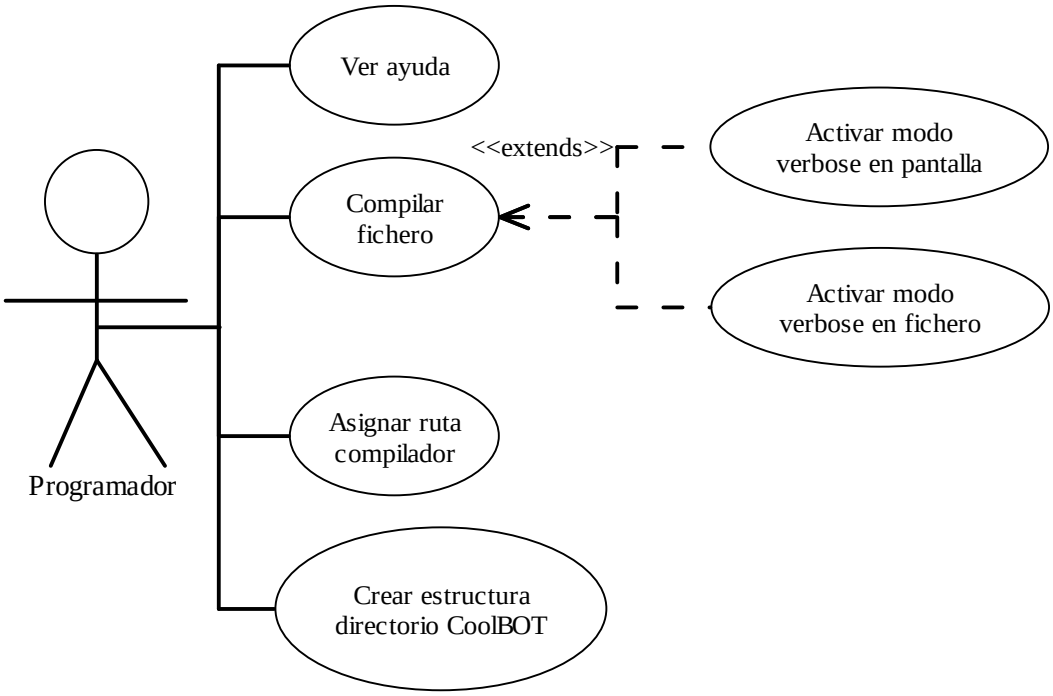


Figura 18: Diagrama de casos de uso.

4.2.2. Requisitos del Análisis Léxico.

La fase de Análisis Léxico [Pérez-Aguiar 1998] es la primera fase en el diseño de un compilador y el analizador léxico es el encargado de llevar a cabo esta fase. El análisis Léxico funciona a nivel de símbolos, es decir, lee una secuencia de caracteres uno a uno y construye símbolos que poseen un significado propio. Estos símbolos constituyen las unidades más simples que tienen significado y se denominan *tokens*. Por tanto, la unidad que recoge el análisis léxico es el carácter y lo que entrega es el *token*.

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un *token* conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. A parte de encargarse de reconocer caracteres y devolver *tokens*, se encarga de realizar estas otras funciones:

- Manejo del fichero de entrada del programa fuente. Se encarga de abrirlo, de leer sus caracteres, de cerrarlo y de gestionar los posibles errores de lectura.
- Eliminar comentarios, espacios en blanco, tabuladores y saltos de línea (caracteres no válidos para formar un token).

- Contabilizar el número de líneas y columnas para emitir mensajes de error.

Se trata de una fase en la que se trabaja sobre gramáticas regulares o expresiones regulares mediante el uso de autómatas finitos.

El analizador léxico del compilador *coolbotc* se va encargar de reconocer las siguientes clases de *tokens*:

- *Palabras reservadas*. Reconocerá como palabra reservadas aquellas que se especificaron en la Tabla 9 y las que correspondan al lenguaje C++.
- *Identificadores*. Reconocerá como un identificador a aquella secuencia de caracteres formada por una letra (dentro de las letras no se incluyen los caracteres acentuados) y por cero o más letras, dígitos (números del 0 al 9) y el carácter "***undescore***" (el símbolo "_"). No puede contener espacios en blanco ni símbolos especiales.
- *Símbolos especiales*. Reconocerá como símbolos especiales los siguientes símbolos: "***{}:,;=***".
- *Literales cadena*. Reconocerá como un literal cadena a aquella secuencia de caracteres que se encuentren encerrados entre comillas dobles.
- *Números enteros positivos y negativos*. Reconocerá como un número entero positivo a aquella secuencia de caracteres formada exclusivamente por dígitos. Los números enteros negativos se reconocen de la misma forma que los positivos con la salvedad que se antepone el símbolo "-" al principio del número.
- *Comentarios*. Reconocerá como un comentario a la secuencia de caracteres que comienza o con los símbolos "/*" y termina "*/" (pudiendo haber saltos de línea en medio) o con los símbolos "//" (no puede haber saltos de línea). En los comentarios, el analizador léxico no devuelve ningún *token*, simplemente los reconoce.

Todo aquello que se reconozca y no coincida con ninguno de los tokens que se han mencionado se considera un error y el analizador léxico deberá mostrar un mensaje de error. A parte de los mensajes de error, el analizador léxico puede mostrar mensajes de advertencia, de aquí en adelante *warnings*, en determinados casos. En la Tabla 11 podemos ver los mensajes de error y warnings que mostrará el analizador léxico.

Para verificar el correcto funcionamiento del analizador léxico se construirá una batería de pruebas que se encargan de chequear que se detecten correctamente los tokens y que, en el caso de existir errores, se muestren los warnings y los mensajes de

error correspondientes al usuario. Las pruebas que se realizarán se clasifican en los siguientes grupos:

- Pruebas para los comentarios.
- Pruebas para los identificadores.
- Pruebas para los literales cadena.
- Pruebas para los números enteros positivos y negativos

Mensajes de error.	
Mensaje	**Descripción**
Bad identifier definition	Se produce cuando un identificador esta mal construido.
Malformed commentary	Se produce cuando no se ha cerrado un comentario.
Bad literal definition	Se produce cuando un literal esta mal construido
Invalid character	Se produce cuando se ha utilizado un carácter inválido.
Mensajes de warnings.	
Mensaje	**Descripción**
String too large, it will be truncated	Se produce cuando se ha definido un literal que supera los 255 caracteres.

Tabla 11: Mensajes de error y warnings del analizador Léxico del compilador *coolbotc*.

El analizador léxico del compilador *xmlcoolbotc* es casi idéntico al analizador léxico del compilador *coolbotc*. Se diferencias en la clases de tokens que reconocen y que a continuación pasamos a comentar:

- *Palabras reservadas*. Reconocerá como palabra reservadas aquellas etiquetas especificas en la Tabla 10.
- *Identificadores*. Reconocerá como un identificador a aquella secuencia de caracteres formada por una letra (dentro de las letras no se incluyen los caracteres acentuados) y por cero o más letras, dígitos (números del 0 al 9) y el carácter "***undescore***" (el símbolo "_"). No puede contener espacios en blanco ni símbolos especiales.
- *Símbolos especiales*. Reconocerá como símbolos especiales los siguientes símbolos: "**=<>?/**".

- *Literales cadena*. Reconocerá como un literal cadena a aquella secuencia de caracteres que se encuentren encerrados entre comillas dobles.
- *Números enteros positivos y negativos*. Reconocerá como un número entero positivo a aquella secuencia de caracteres formada exclusivamente por dígitos. Los números enteros negativos se reconocen de la misma forma que los positivos con la salvedad que se antepone el símbolo "-" al principio del número.
- *Comentarios*. Reconocerá como un comentario a la secuencia de caracteres que comienza con los símbolos "<*!*--" y termina "-->" (pudiendo haber saltos de línea en medio). En los comentarios, el analizador léxico no devuelve ningún *token*, simplemente los reconoce.

Para facilitar la detección de errores léxicos que se puedan producir, tanto para el compilador *coolbotc* como para el compilador *xmlcoolbotc* se dispone de una opción que permite al programador ver en pantalla lo que el analizador léxico ha reconocido (modo verbose). Por cada *token* que reconozca a parece en pantalla el número de línea y de columna y el *token* reconocido. Del mismo modo que se permite al programador volcar esta información en pantalla, se dispone de otra opción que permita volcar ese contenido en un fichero.

4.2.3. Requisitos del Análisis Sintáctico.

El Análisis Sintáctico [Pérez-Aguiar 1998] es la segunda fase en el diseño de un compilador y el analizador sintáctico es el encargado de llevar a cabo esta fase. El análisis sintáctico funciona a nivel de gramática, es decir, partimos de una determina definición del lenguaje (gramática) que nos indica que programas y estructuras podemos escribir. Las funciones que realiza son las siguientes:

- Comprobar si las secuencias de *tokens* proporcionadas por el analizador léxico pueden ser generadas por la gramática que define el lenguaje fuente (gramática independiente del contexto).
- Construir el árbol de análisis sintáctico que define la estructura jerárquica de un programa y obtener la serie de derivaciones para generar la secuencia de *tokens*. El árbol sintáctico se utilizará como representación intermedia en la generación de código.

- Informar de los errores sintácticos de forma precisa y significativa y deberá estar dotado de un mecanismo de recuperación de errores para continuar el análisis.

El análisis sintáctico se puede considerar como una función que toma como entrada la secuencia de *tokens* producidas por el analizador léxico y produce como salida el árbol sintáctico.

El análisis sintáctico trabaja sobre gramáticas independientes del contexto y la gramática se define mediante diagramas sintácticos y la notación EBNF (Extended Backus-Naur Form) [Pérez-Aguiar 1998].

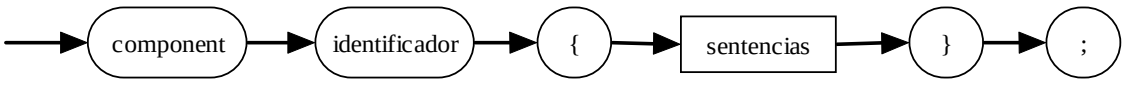
Los diagramas sintácticos son grafos dirigidos que tienen dos componentes esenciales:

- El *rectángulo* que indica que en ese punto hay un elemento del vocabulario que es ***no-terminal***, es decir que tendrá otro diagrama sintáctico para definirle.
- El *círculo* para indicar que en ese punto hay un símbolo ***terminal*** o *token*, es decir que es un elemento perteneciente al vocabulario con significado por si mismo.

La notación EBNF es una extensión de la notación BNF. La notación BNF es una metasintaxis (sintaxis para describir otras sintaxis), que al igual que los diagramas sintácticos, sirve para expresar gramáticas independientes del contexto.

La notación EBNF es una colección de reglas denominadas producciones. Toda regla describe un fragmento específico de sintaxis. Un documento es válido si puede ser reducido, a través de la aplicación repetida de reglas, a una única regla específica. El formato de una regla es el siguiente: <*simbolo*> *::=* <*expresión con símbolos*> donde <*símbolo*> es un *no-terminal*, y la expresión consiste en secuencias de símbolos y/o secuencias separadas por la barra vertical (”|”) indicando una opción. Los símbolos que nunca aparecen en un lado izquierdo son *terminales*.

En la Figura 19 podemos ver la definición de un componente en el lenguaje DL tanto en notación EBNF como a través de diagramas sintácticos.




```
<inicio> ::= <especificacion_componente> <definicion_componente> ";" .
<especificacion_componente> ::= component identifier .
<definicion_componente> ::= "{" <sentencias> "}"
```

Figura 19: Definición de un componente en DL empleando diagramas sintácticos y notación EBNF**.**

Para el analizador sintáctico del compilador *coolbotc* y del compilador *xmlcoolbotc* se va a utilizar como lenguaje el descrito en el apartado 4.1 (lenguaje DL y sintaxis XML para documentos DLXML).

La condición necesaria para que este lenguaje se pueda emplear en un analizador sintáctico es que la gramática de dicho leguaje no sea ambigua. Por tanto antes de empezar a trabajar con el analizador sintáctico debemos asegurarnos que la gramática del lenguaje no sea ambigua.

Para estudiar la ambigüedad de las gramáticas se emplean los conjuntos directores que aseguran que en dicha gramática no se pueden construir dos o más árboles sintácticos que generen el mismo conjunto. Los conjuntos directores se basan en el cálculo de los primeros y de los siguientes y los estudiamos apoyándonos en los diagramas sintácticos.

Los diagramas sintácticos empleados en este trabajo se encuentran descritos en el Apéndice B y después de realizar el estudio de conjuntos directores hemos observado que la gramática de nuestro lenguaje no es ambigua y por tanto podemos construir un analizador sintáctico que acepte el lenguaje del apartado 4.1.

Los errores que se pueden producir en el analizador sintáctico del compilador *coolbotc* son debidos a que se ha escrito incorrectamente alguna palabra reservada o que se ha olvidado de poner algún *token* requerido en alguna construcción para ser correcta (por ejemplo el olvido de poner una coma o un punto y coma,…).

Los errores que se pueden producir en el analizador sintáctico del compilador *xmlcoolbotc* son debidos a que se pueda haber escrito incorrectamente alguna etiqueta o que se haya olvidado una la etiqueta de cierre, o se haya empleado una etiqueta en una construcción donde no se permite.

Para verificar que el analizador sintáctico desempeña su trabajo correctamente, se ha realizado una batería de pruebas que se centrará en los siguientes puntos:

- *Para el compilador coolbotc.* Nos centraremos en:
    1. Detectar palabras reservadas que han sido escritas incorrectamente o que no se pueden usar en una determinada regla sintáctica.
    2. Olvido de *tokens requeridos en* una determinada regla sintáctica como es el olvido de un abre llaves, coma, punto y coma o de una palabra reservada
- *Para el compilador xmlcoolbotc*. Nos centraremos en:

1. Detectar las etiquetas que han sido escritas incorrectamente o que no se pueden emplear en una determinada regla sintáctica.
2. Olvido de colocar las etiquetas de cierre o de colocar un determinado *token.*

4.2.4. Requisitos del Análisis Semántico.

El Análisis Semántico [Pérez-Aguiar 1998] es la tercera fase en el diseño de un compilador y el analizador semántico es el encargado de llevar a cabo esta fase. En el lenguaje fuente, aparte de la información léxica y sintáctica, existe una información adicional que está relacionada con el significado (la semántica) del programa y no con su estructura (la sintaxis). Esta información es dependiente del contexto y es procesado en la fase de análisis semántico.

Las funciones que se realizan en el análisis semántico son las siguientes:

- La construcción de la Tabla de Símbolos para llevar un seguimiento del significado de los identificadores en el programa (variables, funciones, tipos, parámetros, …)
- Realizar la comprobación e inferencia de tipos en expresiones y sentencias (por ejemplo, que ambos lados de una asignación tengan tipos adecuados, que no se declaren variables con el mismo nombre, que los parámetros de llamada a una función tengan tipos adecuados, número de parámetros correctos, …)
- Generación de representación interna.

Aunque el análisis semántico se vea como una fase separada del análisis sintáctico, normalmente suele llevarse a cabo al mismo tiempo que se construye el árbol sintáctico. Si realizamos más de una pasada (se denomina así a la implementación de cada fase en un compilador), el análisis semántico se realiza independientemente.

Tanto el compilador *coolbotc* como el compilador *xmlcoolbotc* emplean el mismo analizador semántico. Este analizador se encargará de realizar una serie de comprobaciones que hemos agrupados en los siguientes grupos:

- Comprobaciones relacionadas con la definición de un componente:
    1. Que el identificador del componente no coincida con ninguna palabra reservada de la plataforma CoolBOT.

2. Que como mínimo exista en el componente la definición de un estado del Autómata de Usuario. Este estado tiene que estar definido como de entrada.

- Comprobaciones relacionadas con la definición del header:
    1. Que en la definición del header tenga definido un atributo como mínimo.
    2. Que la definición del header no aparezca más de una vez en la definición del componente.
    3. Que en la definición del header no aparezca definido más de una vez el atributo *author*.
    4. Que en la definición del header no aparezca definido más de una vez el atributo *description*.
    5. Que en la definición del header no aparezca definido más de una vez el atributo *institution*.
    6. Que en la definición del header no aparezca definido más de una vez el atributo *version*.
- Comprobaciones relacionadas con la definición de constantes:
    1. Que el identificador empleado para definir una constante no coincida con ninguna palabra reservada de la plataforma CoolBOT ni con el identificador del componente que se esta definiendo, ni el identificador de una constante que previamente se ha definido.
    2. Que cuando iniciemos la definición de las constantes se asegure que como mínimo se defina una constante.
    3. Que si las constantes se utilizan en otras definiciones, estas estén correctamente definidas.
- Comprobaciones relacionadas con la definición de puertos:
    1. Que el identificador empleado para definir un puerto no coincida con ninguno de los siguiente casos:
        - Palabra reservada de la plataforma CoolBOT.
        - Identificador del componente que se esta definiendo.
        - Identificador de un puerto que previamente se haya definido.
        - Identificador de una constante que previamente se ha definido.
    2. Que en la definición del puerto se emplee el tipo de puerto adecuado en función si el puerto es de entrada o de salida.

3. Que si hemos definido que el puerto emplee un tipo de puerto *multipacket* o *lazymultipacket* o *pull* y se ha especificado el campo *lenght*, el valor definido en ese campo coincida con el número de paquetes de puertos que hayamos especificado en la definición.
4. Que si definimos centinelas, estos estén exclusivamente asociados a puertos de entrada. No puede haber ningún centinela asociado a un puerto de salida, exceptuando los puertos de tipo *pull*.
5. Que si se emplea constantes en la definición de un puerto, estas estén definidas en el componente.
6. Que si definimos el puerto de tipo *priorities*, el número de tiempos de espera (timeouts) especificados coincida con el valor del rango especificado.
7. Que el identificador empleado para definir un paquetes de puertos no coincida con ninguna palabra reservada de la plataforma CoolBOT ni el identificador empleado para definir un componente.

- Comprobaciones relacionadas con la definición de variables:
    1. Que el identificador empleado para definir una variable no coincida con ninguno de los siguientes:
        - Palabra reservada de la plataforma CoolBOT.
        - Identificador del componente que se esta definiendo.
        - Identificador de una variable que previamente se haya definido.
        - Identificador de una constante que previamente se haya definido.
        - Identificador de un puerto que previamente se haya definido.
    2. Que si iniciamos la definición de variables, se garantice que como mínimo se define una variable.
- Comprobaciones relacionadas con la definición de excepciones:
    1. Que el identificador empleado para definir una excepción no coincida con ninguno de los siguientes:
        - Palabra reservada de la plataforma CoolBOT.
        - Identificador del componente que se esta definiendo.
        - Identificador de una excepción que previamente se haya definido.
        - Identificador de una variable que previamente se haya definido.
        - Identificador de una constante que previamente se haya definido.

    - Identificador de un puerto que previamente se haya definido.
2. Que en la definición de una excepción siempre aparezca definido el atributo *description*.
3. Que en la definición de una excepción no exista más de una definición del atributo *description*.
4. Que en la definición de una excepción no exista más de una definición de manejador de recuperación.
5. Que en la definición de una excepción no exista más de una definición de manejador de éxito.
6. Que en la definición de una excepción no exista más de una definición de manejador de fracaso.

- Comprobaciones relacionadas con la definición de estados:
    1. Que el identificador empleado para definir un estado no coincida con ninguno de los siguientes:
        - Palabra reservada de la plataforma CoolBOT.
        - Identificador del componente que se esta definiendo.
        - Identificador de un estado que previamente se haya definido.
        - Identificador de una excepción que previamente se haya definido.
        - Identificador de una variable que previamente se haya definido.
        - Identificador de una constante que previamente se haya definido.
        - Identificador de un puerto que previamente se haya definido.
    2. Que si especificamos transiciones están se realicen sobre puertos de entrada o sobre variables controlables que hemos definido en el componente.
    3. Que si especificamos transiciones, no exista más de una transición con el mismo puerto de entrada o variable controlable en ese mismo estado.
    4. Que todo puerto de entrada definido en el componente aparezca definido en alguna transición de algún estado.
    5. Que sólo haya definido un estado de entrada en la definición de componentes.
- Comprobaciones relacionadas con la definición de hilos:
    1. Que el identificador empleado para definir un hilo no coincida con ninguno de los siguientes:

- Palabra reservada de la plataforma CoolBOT.
- Identificador del componente que se esta definiendo.
- Identificador de un hilo que previamente se haya definido.
- Identificador de un estado que previamente se haya definido.
- Identificador de una excepción que previamente se haya definido.
- Identificador de una variable que previamente se haya definido.
- Identificador de una constante que previamente se haya definido.

2. Que si definimos un hilo con la política de espera activa, este hilo no sea el hilo definido como *main*.
3. Que en la definición de un hilo siempre aparezca la definición de puertos de entrada a los cuales el hilo esta asociado.
4. Que en la definición de un hilo, los puertos de entrada que se definan hayan sido definidos en el componente.
5. Que un hilo sólo puedan ser o de tipo *input box* o de *priority input box.*
6. Que en los puertos de entrada asociados a un hilo no haya puertos de entrada repetidos. En cada hilo sólo se puede definir una vez un puerto de entrada y este no puede volver aparecer ni en ese hilo ni en ningún otro hilo.
7. Si definimos los puertos de entrada de tipo *priority input box*, que los identificadores que empleemos para designar la prioridad no coincida ni con una palabra reservada de la plataforma CoolBOT ni con el identificador empleado para definir el componente. Tampoco puede coincidir con el identificador de una constante, puerto, variable, excepción, estado o hilo.
8. Si definimos los puertos de entrada de tipo *priority input box*, que no haya dos identificadores para designar prioridad iguales. No puede definirse el mismo identificador de prioridad más de una vez dentro de un hilo y tampoco se puede utilizar en la definición de otro hilo.
9. Si definimos estados donde el hilo esta activo, que los identificadores empleados para designar los estados no coincida ni con el identificador empleado para definir el componente ni con ninguna palabra reservada de la plataforma CoolBOT.

10. Si definimos estados donde el hilo esta activo, que los identificadores empleados para designar los estados no coincida con los estados del Autómata por Defecto de la plataforma CoolBOT.
11. Si definimos estados donde el hilo esta activo, que los identificadores empleados para designar los estados no coincida con los identificadores de constantes, puertos, variables, excepciones, estados e hilos.
12. Si definimos estados donde el hilo esta activo, que no existan dos estados con el mismo identificador. Es decir, que no se puede especificar en un hilo el mismo estado más de una vez.
13. Si definimos estados donde el hilo esta activo, que dichos estados se hayan definido en el componente.
14. Que todos los puertos de entrada definidos en el componente se encuentren asociados a un hilo.

Estas comprobaciones darán lugar a errores y warnings que deberán ser mostrados al programador. En las Tablas 12, 13, 14 y 15 podemos ver un listado de mensajes de error y de advertencia que los respectivos compiladores muestran:

Mensajes de error
Bad component definition: reserverd word used
Bad component definition: no entry state defined, there should be one, it is mandatory
Bad header sentence: no attribute has been defined
Bad header sentence: sentence header definition duplicated
Forbidden constant identifier: component name is unique, it cannot be used
Forbidden constant identifier: reserved word in CoolBOT
Bad constant definition: constant identifier duplicated
Unknown constant: unknown constant identifier
Bad constant definition: it is mandatory to define at least a constant as minimal
Forbidden port identifier: component name is unique, it cannot be used
Forbidden port identifier: reserved word in CoolBOT
Bad port definition: port type not applicable
Bad port definition: length mismatch, length and port packets do not match

Tabla 12: Mensajes de error del Análisis Semántico.

Mensajes de error (continuación)
Bad port definition: it is not possible to define a watchdog for an output port
Forbidden port identifier: port identifier duplicated
Forbidden port identifier: constant identifier used
Port packets have not been defined. The port packets definition is mandatory
Bad Priorities port definition: not defined constant value
Bad port definition: range of priorities mismatch, range and timeouts do not match
Forbidden port packet identifier: component name is unique, it cannot be used
Forbidden port packet identifier: reserved word in CoolBOT
Forbidden variable identifier: component name is unique, it cannot be used
Forbidden variable identifier: reserved word in CoolBOT
Forbidden variable identifier: variable identifier duplicated
Empty variable sentence: there should be at least one variable definition
Forbidden variable identifier: constant or port identifier used
Forbidden exception identifier: component name is unique, it cannot be used
Forbidden exception identifier: reserved word in CoolBOT
Forbidden exception identifier: exception identifier duplicated
Bad exception sentence: description attribute cannot be omitted
Forbidden exception identifier: identifier already used
Forbidden state identifier: component name is unique, it cannot be used
Forbidden state identifier: reserved word in CoolBOT
Bad state transition: transition duplicated on the same input port or controllable variable
Bad state transition: component name is unique, it cannot be used to define a state transition
Bad state transition: reserved word in CoolBOT
Bad state declaration: there should be only one entry state
Forbidden state identifier: state identifier duplicated
Forbidden state identifier: identifier used
Bad state transition: input port or controllable variable not defined
Forbidden thread identifier: component name is unique, it cannot be used

Tabla 13: Continuación de los mensajes de error del Análisis Semántico.

Mensajes de error (continuación)
Forbidden thread identifier: reserved word in CoolBOT
Forbidden priority identifier: component name is unique, it cannot be used
Bad priority identifier: reserved word in CoolBOT
Bad priority identifier: priority name duplicated
Forbidden priority input port identifier: component name is unique, it cannot be used
Forbidden priority input port identifier: reserved word in CoolBOT
Bad priority input port definition: priority input port name duplicated
Bad input box input port identifier: component name is unique, it cannot be used
Bad input box input port identifier: reserved word in CoolBOT
Bad input box input port identifier: name duplicated
Bad thread definition: a thread can only have input box or priority input box
Bad thread state identifier: component name is unique, it cannot be used
Bad thread state identifier: reserved word in CoolBOT
Bad thread state identifier: name duplicated
Bad thread definition: no input ports specified or you have defined the thread main as POLLING
Bad thread definition: thread identifier duplicated
Forbidden thread identifier: identifier used
Forbidden input port identifier: constant identifier used
Forbidden state identifier: constant identifier used
Bad thread definition: input port identifier not defined
Bad thread definition: state identifier not defined
Bad thread definition: input port definition is used in more than one definition
Bad thread definition: this port not this one associated with no thread
Forbidden priority identifier: identifier already used

Tabla 14: Continuación de los mensajes de error del Análisis Semántico.

Mensajes de advertencia (warnings)
Bad header sentence: author attribute definition duplicated: omitting
Bad header sentence: description attribute definition duplicated: omitting
Bad header sentence: institution attribute definition duplicated: omitting
Bad header sentence: version attribute definition duplicated: omitting
Bad exception sentence: description attribute duplicated: omitting
Bad exception sentence: recovery handler attribute duplicated: omitting
Bad exception sentence: success handler attribute duplicated: omitting
Bad exception sentence: failure handler attribute duplicated: omitting
Input port with no state transition associated
It is not necessary to have in the definition of Thread, the definition the ports control, empty_transition and timer

Tabla 15: Mensajes de advertencias.

Para poder realizar las anteriores comprobaciones, se necesita de una estructura de datos donde se almacena la información asociada a los *tokens*. Esta estructura se denomina *tabla de símbolos*.

Nuestro analizador semántico no dispondrá de una sola tabla de símbolos, sino que ésta se encontrará separado en varias tablas. Por cada elemento que se permita definir se dispondrá de una tabla de símbolos para contener, en exclusiva, la información de ese elemento. De este modo existirá una tabla de símbolos para el header, otra para la definición de constantes,… La unión de todas estas tablas constituye la tabla de símbolos que emplea el analizador semántico.

Para facilitar la depuración y seguimiento de los fallos semánticos producidos, se dispone de una opción para realizar un volcado de la tabla de símbolos en pantalla o en un fichero.

Por último para probar el buen funcionamiento del analizador semántico, se realizarán varios pruebas que chequearán a fondo las comprobaciones mencionadas anteriormente.

4.2.5. Requisitos de la Generación de Código.

La Generación de Código [Pérez-Aguiar 1998] es una fase de la fase de síntesis de un compilador y se ejecuta sobre los resultados obtenidos en la fase de análisis (Análisis Léxico, Análisis Sintáctico y Análisis Semántico). Se trata de la fase en donde se realiza la traducción del lenguaje de programación de partida al lenguaje de programación de destino.

En esta fase nos centramos en la generación de esqueletos C++ que es una fase común tanto para el compilador *coolbotc* como para el compilador *xmlcoolbotc*. Esta fase toma como entrada la información almacenada en la tabla de símbolos, por lo que sólo se iniciará cuando no se ha detectado ningún error en la fase de análisis.

En el apartado 2.8 se ha visto como se organizaban los esqueletos C++ de un componente CoolBOT. También vimos que los esqueletos tienen una parte que es común a todo componente y otra que varía en función del diseño especifico del componente. Teniendo esto en cuenta, se puede separar la generación de esqueletos C++ en dos bloques: el que contiene la parte común y el que contiene la parte que varía, de forma que la unión de estos dos bloques permite obtener como resultados los esqueletos C++ completos para los componentes.

El bloque que contiene la parte que varía en función de los requisitos del componente, la podemos separar a su vez en cuatro modelos distintos que especifican los tipos de componentes atómicos que podemos definir en CoolBOT:

- *Componentes básicos*. Componentes que no poseen definiciones ni de centinelas ni de hilos (son componentes monohilos). En estos componentes sólo encontramos definiciones de header, constantes, puertos, variables, excepciones, estados y hilos.
- *Componentes con hilos*. Componentes básicos que no poseen definiciones de centinelas pero si de hilos (componentes multihilo).
- *Componentes básicos con centinelas*. Componentes básicos que poseen definiciones de centinelas (son componentes monohilos).
- *Componentes con hilos con centinelas*. Componentes con hilos que poseen definiciones de centinelas (componentes multihilo).

Lo comentado en los párrafos anteriores, sólo se tendrá en cuenta en la generación de esqueletos C++. Para la generación de esqueletos XML, simplemente se

realizará un proceso de traducción de lenguaje DL a sintaxis XML de un documento DLXML.

Capítulo 5
Diseño.

En este capítulo vamos a explicar como se ha organizado y estructurado el software. Para ello, comenzaremos hablando sobre la arquitectura que poseen los compiladores para luego estudiar los patrones de diseño empleados y ver los diagramas de clases. Tanto el compilador *coolbotc* como *xmlcoolbotc* poseen el mismo diseño arquitectónico, sólo varían el analizador léxico y sintáctico los cuales son proporcionados por las herramientas Flex y Bison. Por tanto, la arquitectura software que se explica en este apartado es valida para los dos compiladores

5.1. Arquitectura del software.

Dado que nuestro objetivo principal es generar esqueletos C++, vamos a emplear el lenguaje C++ en la construcción de los compiladores. En apartado 3.1. se explico que se va a emplear, como modelo estructural, un modelo de capas de forma que cada fase de diseño se corresponderá con una capa y que emplearemos el modelo incremental como paradigma de programación.

Dado que se emplearán las herramientas Flex y Bison (ver apartado 6.2) para la generación de los analizadores léxicos y sintácticos, nos centraremos en el diseño del analizador semántico y en la generación de código. En la Figura 20 podemos apreciar un diagrama UML [Larman, 1999] donde vemos la arquitectura del software. Esta arquitectura esta formado por los siguientes módulos:

- *Common*. Este módulo contiene declaraciones y clases que son empleados por los módulos *Semántic* y *Skeletons*.
- *Util*. Este módulo contiene una serie de clases que proporcionan métodos para el tratamiento de cadenas de caracteres (ristras), chequeos de identificadores y acceso al shell del sistema. Se emplea en los módulos *Semántic* y *Skeletons*.
- *Errors*. Este módulo contiene clases para el tratamiento de errores.

- Semantic. Módulo que contiene el analizador semántico y la implementación de la tabla de símbolos.
- Skeletons. Módulo que contiene la generación de esqueletos C++ y XML.

Todos estos módulos son accesibles desde el exterior a través de una clase que contiene la implementación del patrón Facade (ver apartado 5.2) de forma que todos estos módulos a su vez están agrupados en otro modulo que denominado CoolBOT. En la Figura 21 podemos ver otro diagrama UML donde se aprecia esta última arquitectura.

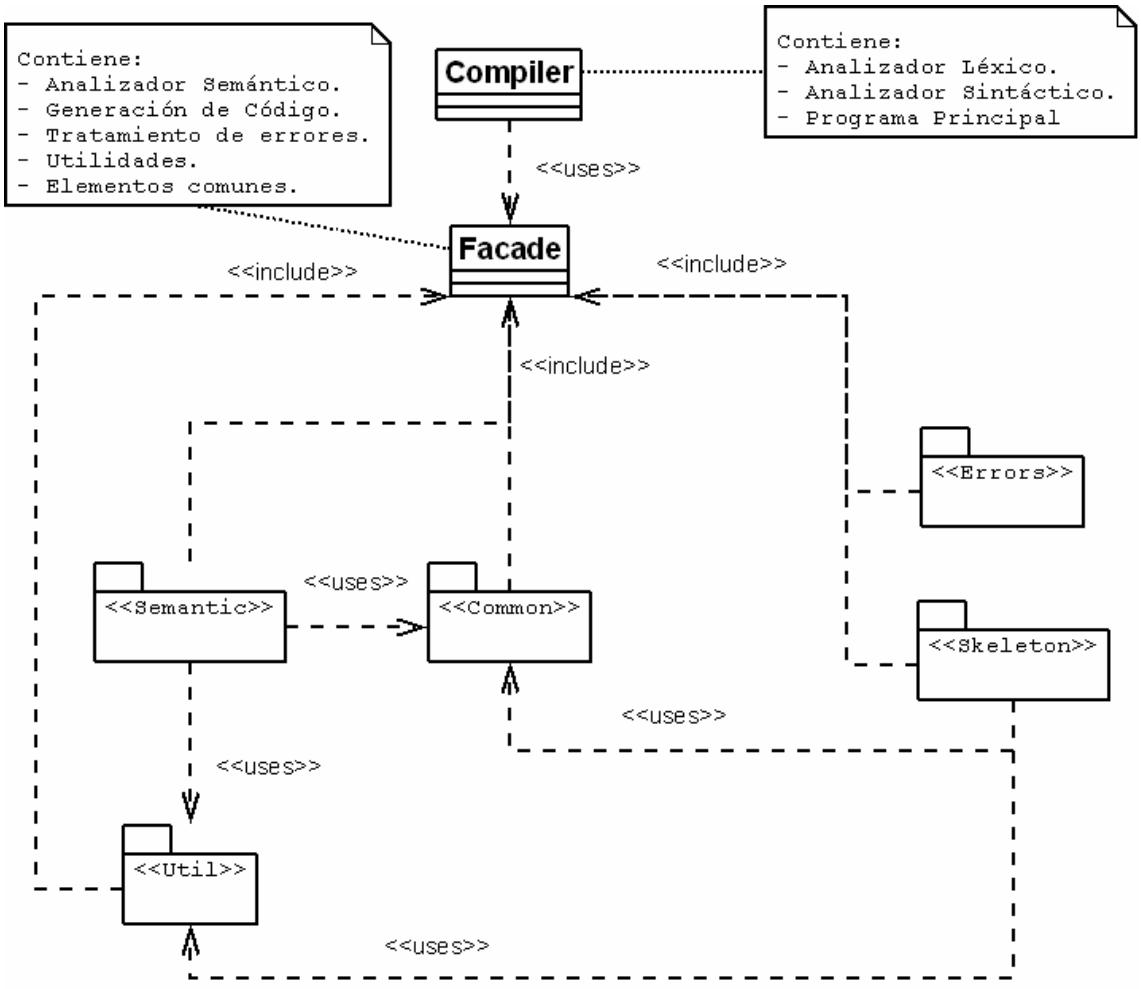


Figura 20: Arquitectura del software.

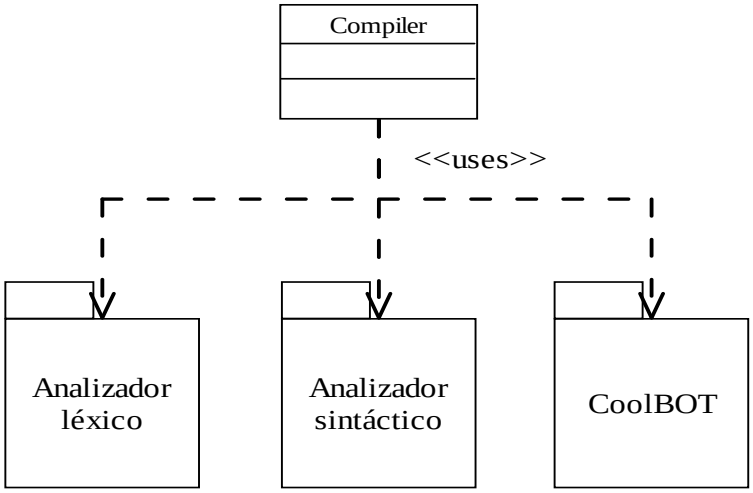


Figura 21: Arquitectura del software.

5.2. Patrones de diseño.

Un patrón de diseño [Gamma et al., 1995] es una solución a un problema de diseño no trivial que es efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias). En la arquitectura mostrada en la Figura 20, vamos a emplear como patrón de diseño el patrón *Facade* o *Fachada*.

El patrón *Facade* me permite proporcionar una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas. De este modo conseguimos desacoplar el software reduciendo la complejidad y minimizando las dependencias entre los subsistemas que conforman el software. En la Figura 22 podemos ver un gráficamente como actúa el patrón *Facade*.

5.3. Diagramas de clases.

Un *diagrama de clase* es la herramienta que me permite modelar la parte estática de la realidad del usuario (requisitos del usuario), empleando la clase como elemento principal. Una clase es una categoría o grupo de cosas que tienen atributos y acciones similares.

En los diagramas de clases se presentan clases y como estas están relacionadas entre sí. En la Figura 23 podemos ver los elementos principales que aparecen en estos diagramas y que a continuación pasamos a comentar:

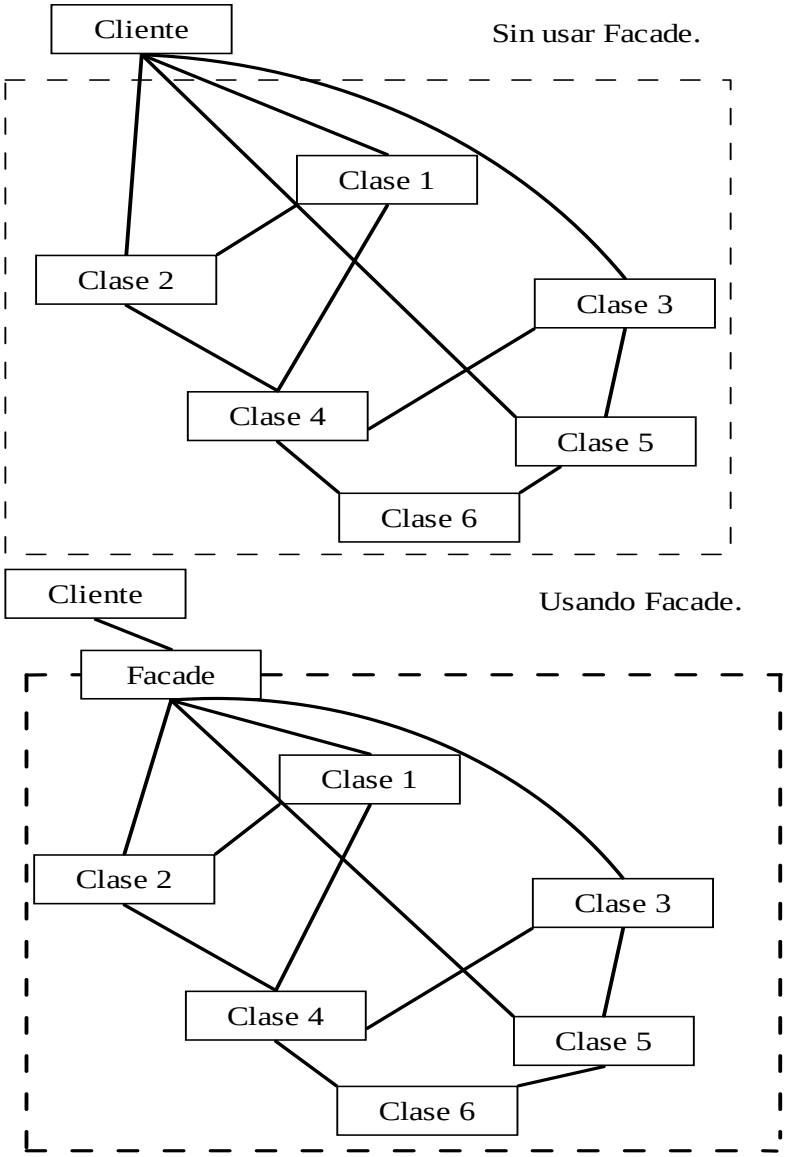


Figura 22: Patrón Facade.

- *Estereotipos*: es un adjetivo de la clase, es decir, nos adjetiva la interpretación que tenemos de la clase. Va entre << y >>.
- *Multiplicidad*: es la cantidad de objetos de un clase que se relacionan con un objeto de la clase asociada. La multiplicidad se representa con un número colocado sobre la línea de asociación, agregación o composición junto a la clase correspondiente.
- *Tipos de relaciones*: las clases se pueden relacionar entre sí, distinguiéndose los siguientes tipos de relaciones:
    1. *Clases relacionadas por Herencia*: se le denomina también generalización y se da cuando una clase hereda de otra.

2. *Clases relacionadas por Asociación*: cuando las clases se conectan entre sí de forma conceptual, esta conexión se conoce como asociación. Cuando una clase se asocia con otra, cada una de ellas juega un papel dentro de tal asociación.
3. *Clases relacionadas por Agregación*: se produce cuando una clase consta de otras clases de forma que su definición tiene sentido aunque falte alguna de las clases de las que consta.
4. *Clases relacionadas por Composición*: se produce cuando una clase consta de otras clases de forma que su definición no tiene sentido cuando falte alguna de las clases de las que consta.

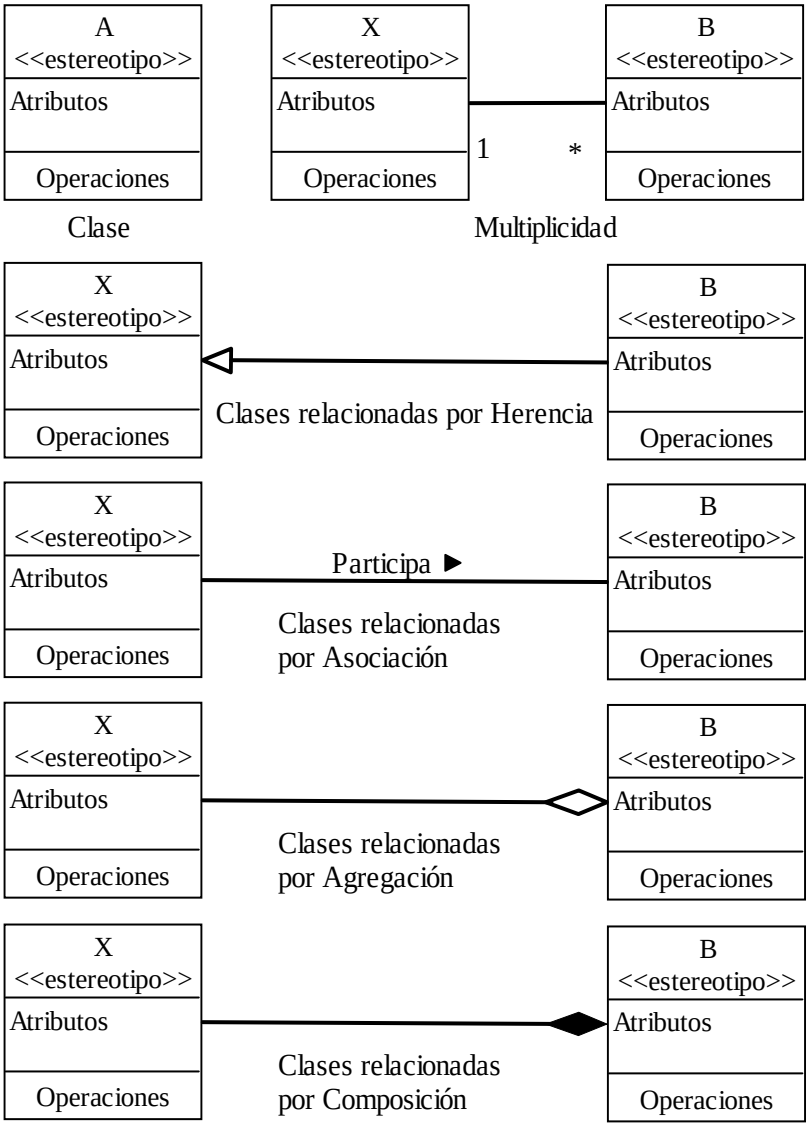


Figura 23: Principales elementos de un diagrama de clase.

A continuación se presentan los diagramas de clases con los cuales se pretende facilitar un futuro mantenimiento de los compiladores.

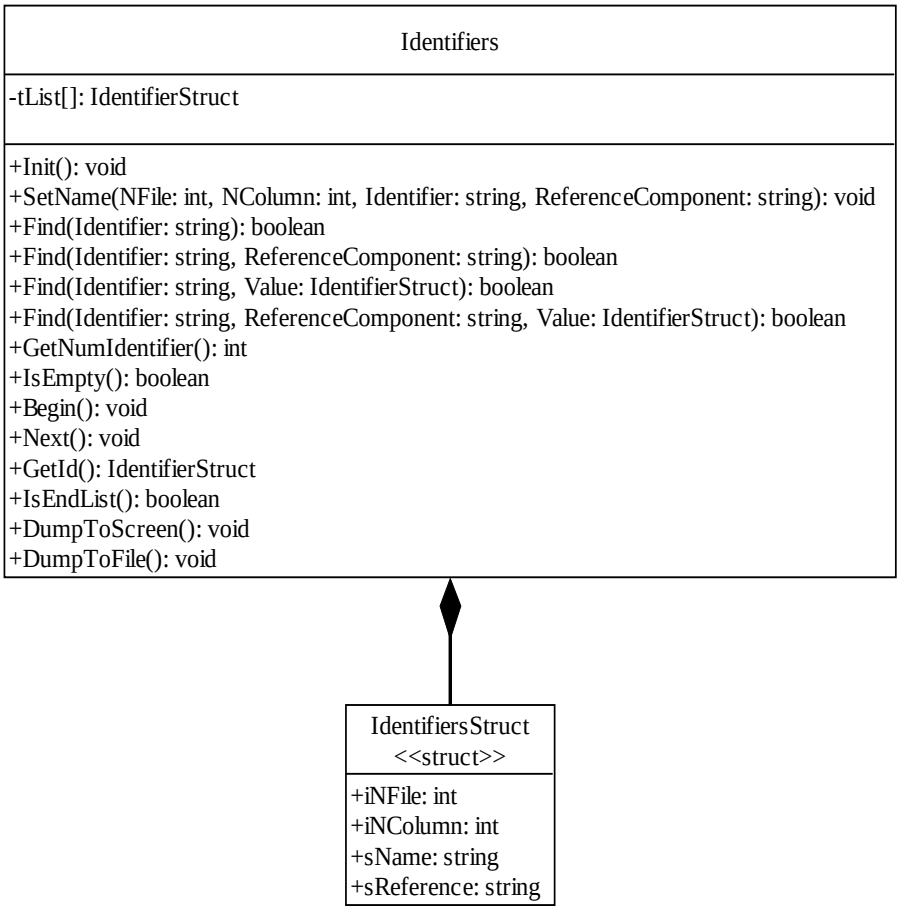


Figura 24: Clase *Identifiers* perteneciente al módulo *Common*.

En la Figura 24 podemos ver la clase *Identifiers* que pertenece al módulo *Common*. Esta clase se emplea principalmente en el analizador semántico y se trata de un contenedor de identificadores.

En la Figura 25 vemos las distintas clases que forman el módulo *Util*. Como se puede observar, este módulo esta formado por las siguientes clases:

- Clase *Shell*: Esta clase contiene métodos para poder acceder y trabajar con el intérprete de comandos o shell del sistema. También posee métodos para generar la estructura de ficheros y directorios descritos en la fase 4 de la metodología de desarrollo de un componente CoolBOT (ver apartado 1.1).
- Clase *Checkup*: Esta clase contiene métodos para validar identificadores y paquetes de puertos predefinidos en la plataforma CoolBOT.
- Clase *StringLibrary*: Esta clase contiene métodos para el tratamiento de cadenas.
- Clase *UtilInterface*: Implementación del patrón Facade.

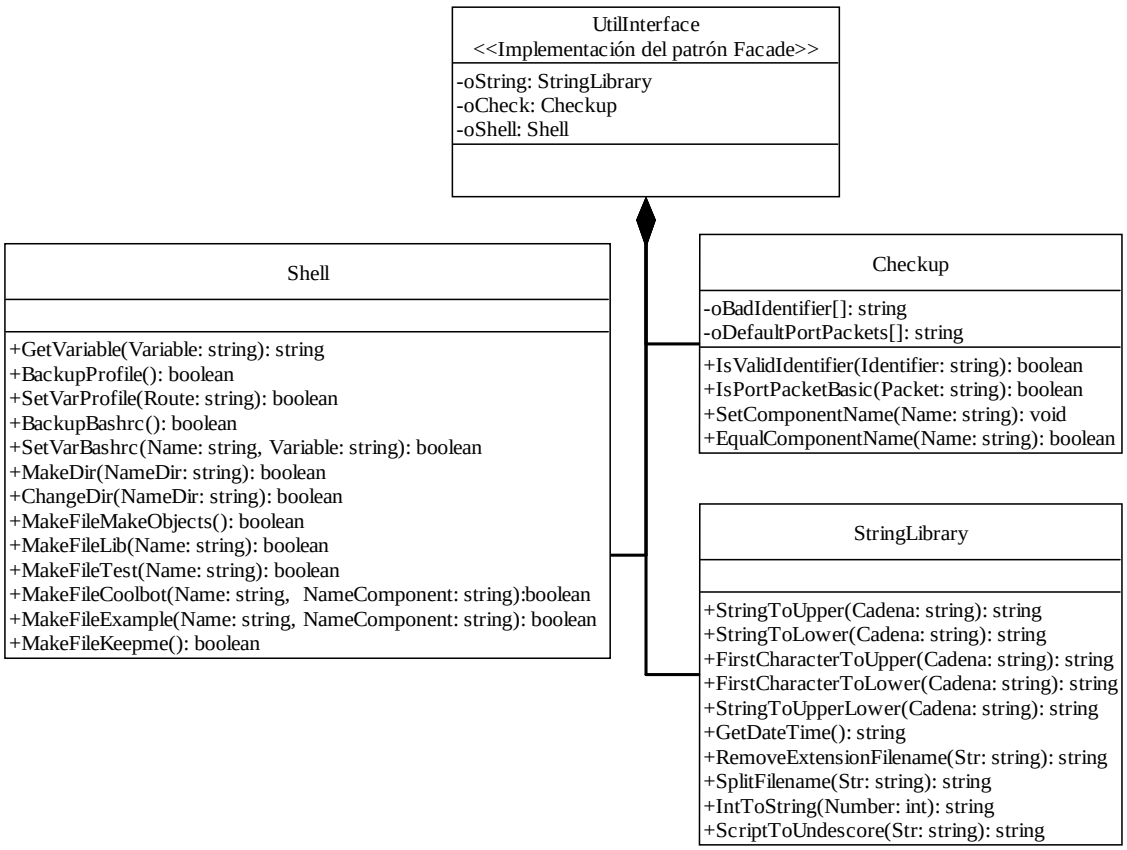


Figura 25: Módulo *Util*.

En la Figura 26 podemos observar las clases que conforma el módulo *Errors*. Este módulo esta formado por las siguientes clases:

- Clase *Errors*: Esta clase contiene métodos para el tratamiento de errores.
- Clase *Verbose*: Esta clase contiene métodos para el tratamiento del modo verbose.
- Clase *ErrorsInterface*: Implementación del patrón Facade.

En la Figura 27 se observa el módulo *Semantic* que contiene las siguientes clases:

- Clase *ComponentInterface* (subsistema *Componente*): Implementación del patrón Facade para el módulo que gestiona la definición del componente y del header.
- Clase *ConstantInterface* (subsistema *Constant*): Implementación del patrón Facade para el módulo que gestiona la definición de constantes.
- Clase *PortInterface* (subsistema *Port*): Implementación del patrón Facade para el módulo que gestiona la definición de puertos.
- Clase *VariableInterface* (subsistema *Variable*): Implementación del patrón Facade para el módulo que gestiona la definición de variables.

- Clase *ExceptionInterface* (subsistema *Exception*): Implementación del patrón Facade para el módulo que gestiona la definición de excepciones.
- Clase *StateInterface* (subsistema *State*): Implementación del patrón Facade para el módulo que gestiona la definición de estados.
- Clase *ThreadInterface* (subsistema *Thread*): Implementación del patrón Facade para el módulo que gestiona la definición de hilos.
- Clase *SemanticInterface* (subsistema *Semantic*): Implementación del patrón Facade que gestiona el análisis semántico.

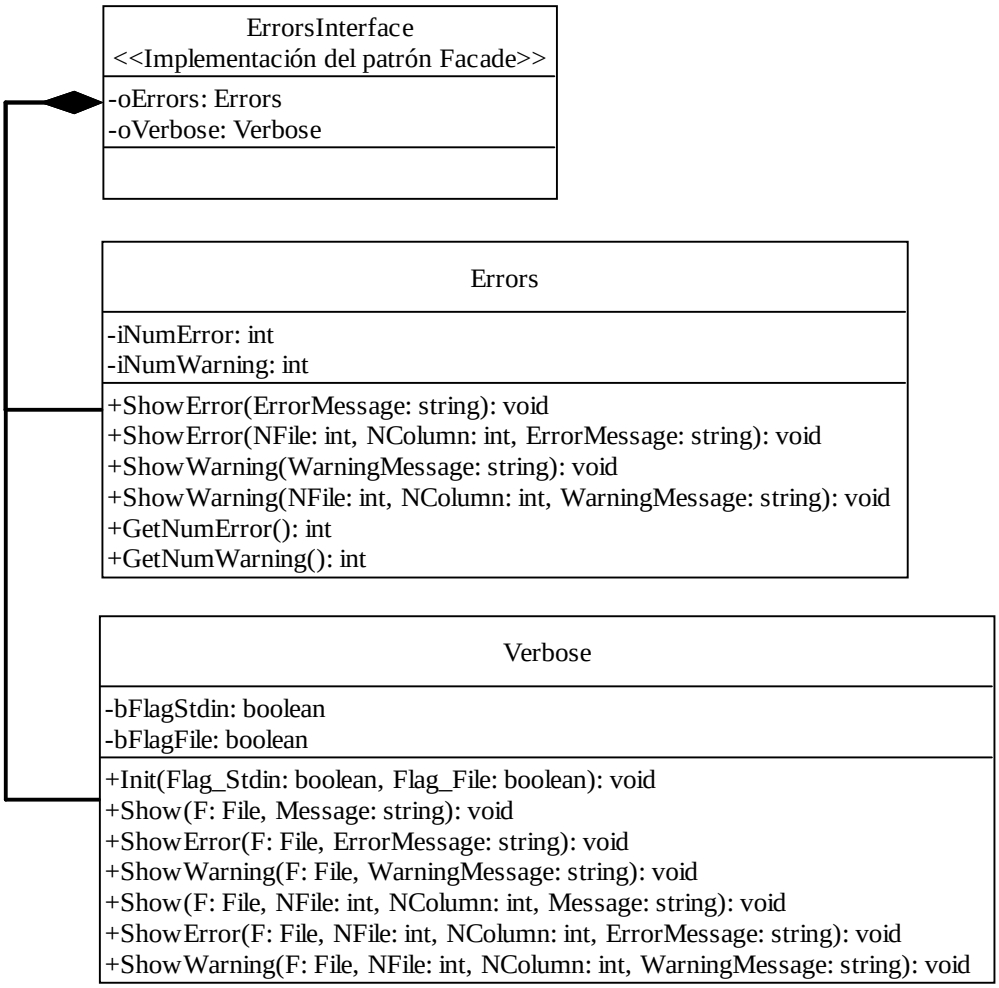


Figura 26: Módulo *Errors*.

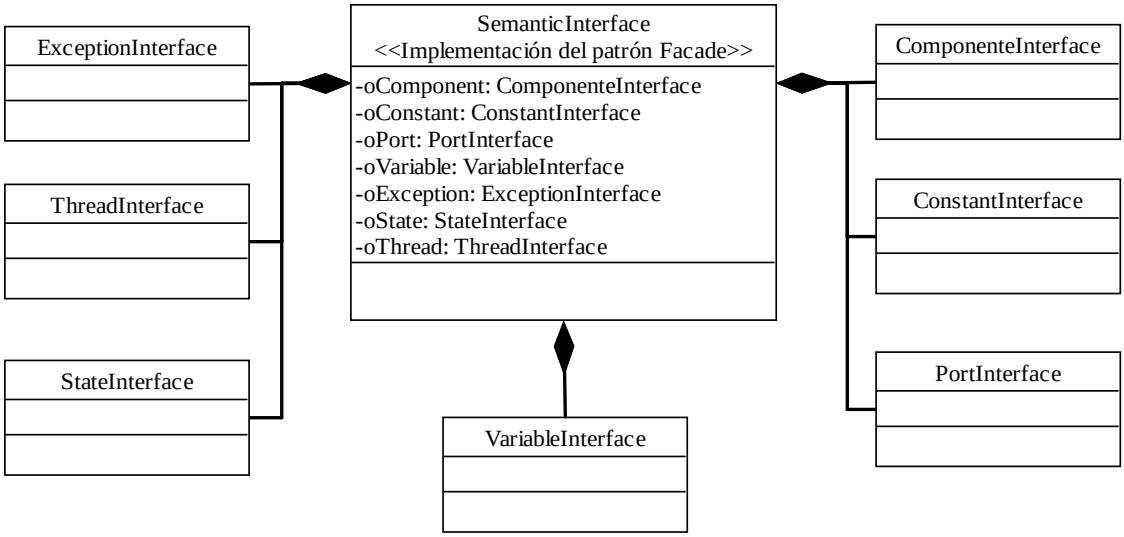


Figura 27: Módulo *Semantic*.

En la Figura 28 podemos ver el subsistema *Componente* perteneciente al módulo *Semantic* que gestiona la definición de componente y la del header. Esta formado por las siguientes clases:

- Clase *Component*: Se trata de una clase contenedora que almacena información sobre la definición del componente.
- Clase *Header*: Se trata de una clase contenedora que almacena información sobre la definición del header.
- Clase ComponenteInterface: Implementación del patrón Facade que gestiona el subsistema *Componente*.

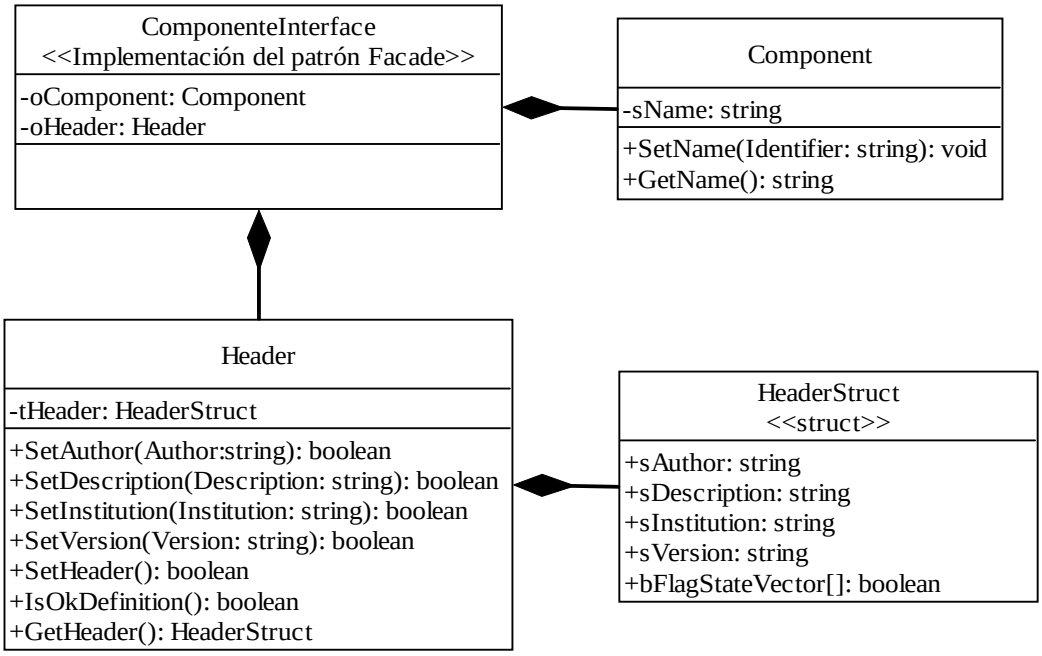


Figura 28: Subsistema *Componente* perteneciente al módulo *Semantic*.

En la Figura 29 se observa el modulo *Contant* que gestiona la definición de constantes y posee las siguientes clases:

- Clase *Constant*: Se trata de una clase contenedora que almacena información sobre la definición de constantes.
- Clase ConstantInterface: Implementación del patrón Facade que gestiona el subsistema *Constant*.

En la Figura 30 podemos ver el módulo *Port* que gestiona la definición de puertos y posee las siguientes clases:

- Clase *Port*: Se trata de una clase contenedora que almacena información sobre la definición de puertos. Esta clase requiere del modulo *Packet*, que podemos ver en la Figura 31, para poder realizar su labor. Este modulo gestiona los tipos de puertos y de paquetes de puertos. Esta formado por las siguientes clases:
    1. Clase *Priorities*: Se trata de una clase contenedora asociado al tipo *priorities* cuya función es almacenar los *timeouts* especificados en este tipo de puerto.
    2. Clase *SimplePacket*: Se trata de una clase contenedora que almacena la información de todos aquellos tipos de puertos que no sean el tipo *multipacket*, *lazymultipacket* o *pull*. En la Figura 32 podemos ver el diagrama de clase correspondiente a la clase *SimplePacket*.
    3. Clase *MultiPacket*: Se trata de una clase contenedora que almacena la información de los tipos de puertos *multipacket* y *lazymultipacket*. En la Figura 32 podemos ver el diagrama de clase correspondiente a la clase *MultiPacket*.
    4. Clase *PullPacket*: Se trata de una clase contenedora que almacena la información del tipo de puerto *pull*. En la Figura 33 podemos ver el diagrama de clase correspondiente a la clase *PullPacket*.
- Clase *PortInterface*: Implementación del patrón Facade que gestiona el subsistema *Port*.

Este modulo hace uso de la clase *Identifiers* que hemos visto anteriormente en la Figura 24.

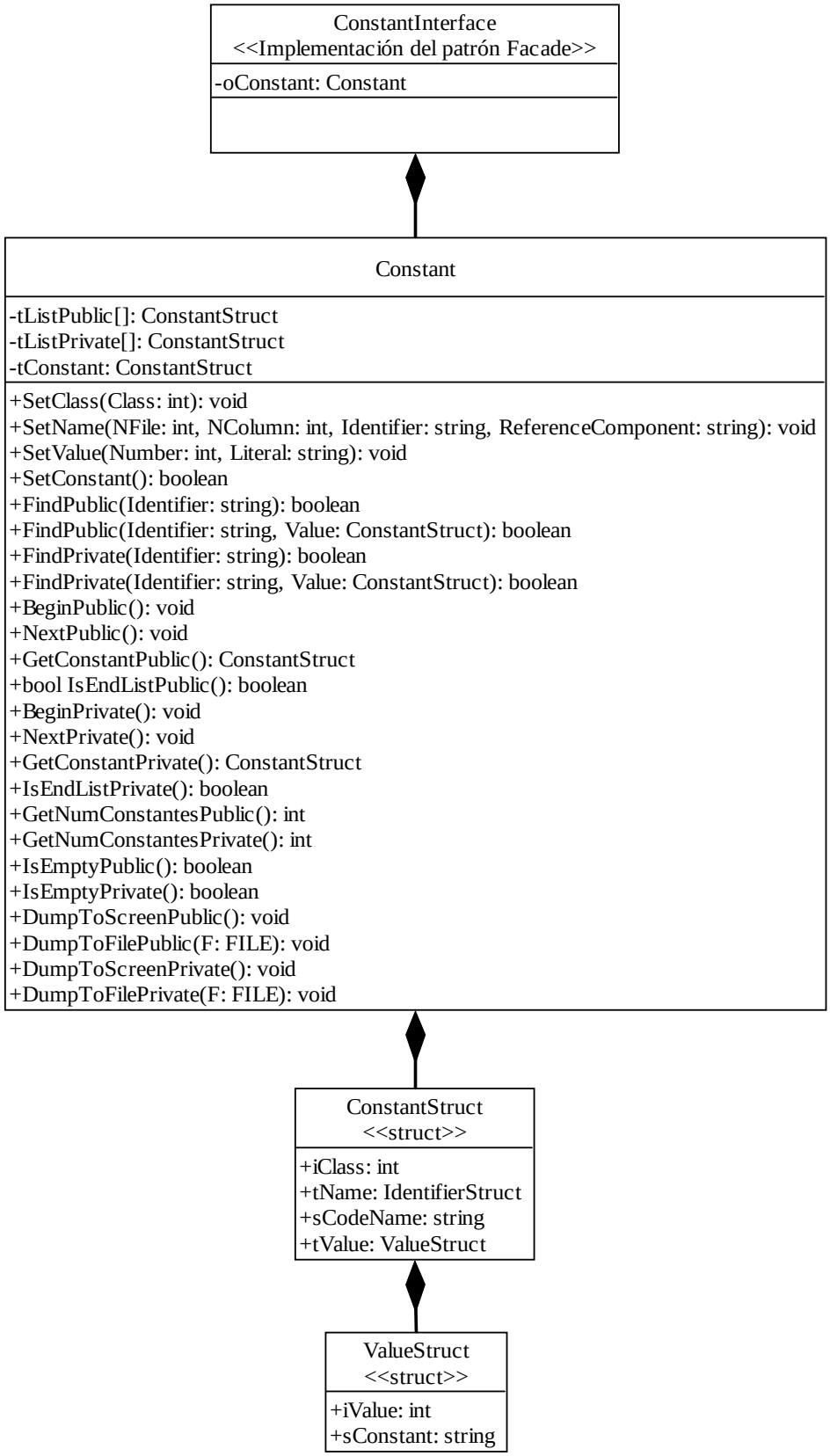


Figura 29: Subsistema *Constant* del módulo *Semantic*.

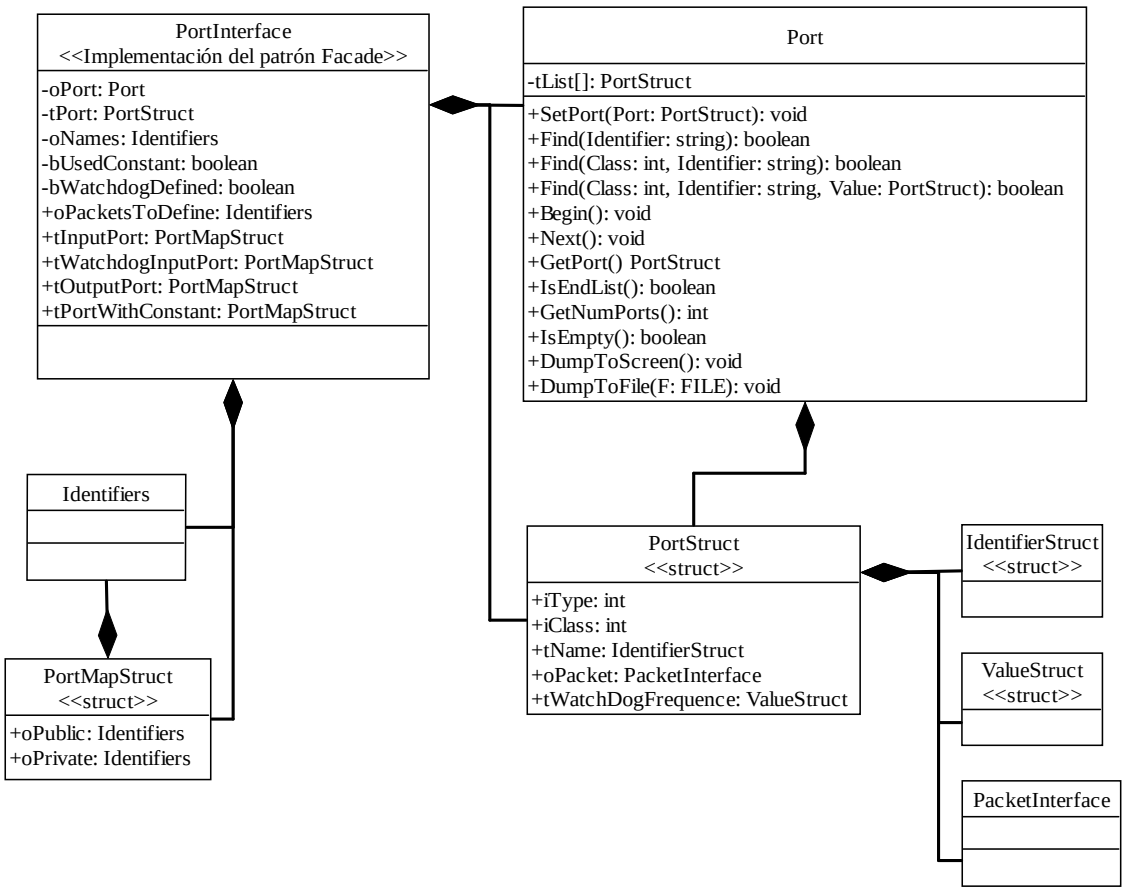


Figura 30: Subsistema *Port* del módulo *Semantic*.

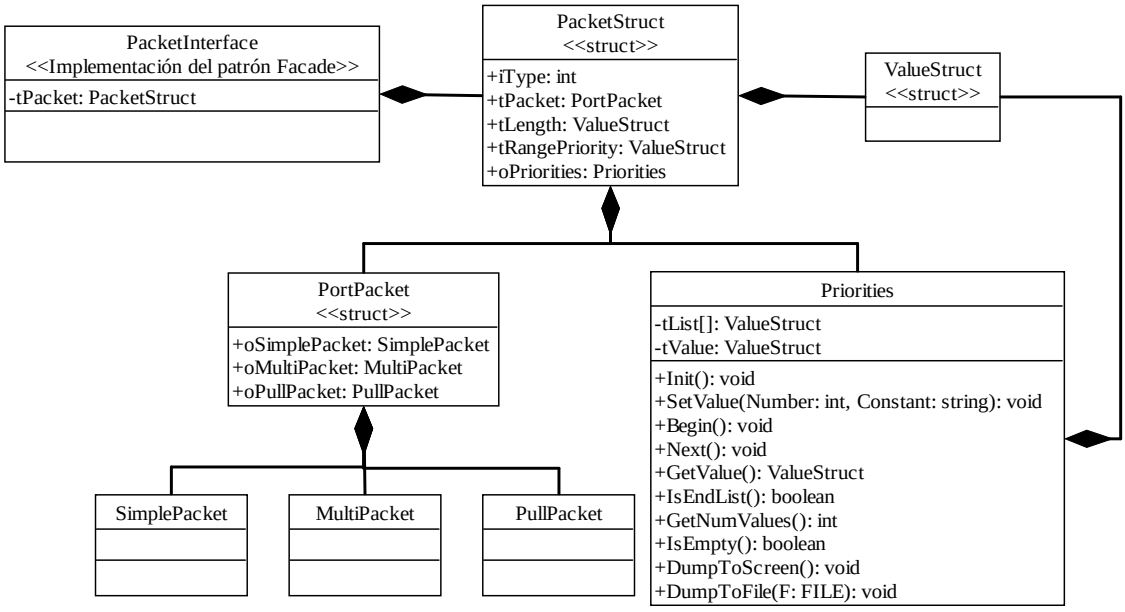


Figura 31: Subsistema *Packet* perteneciente al subsistema *Port*.

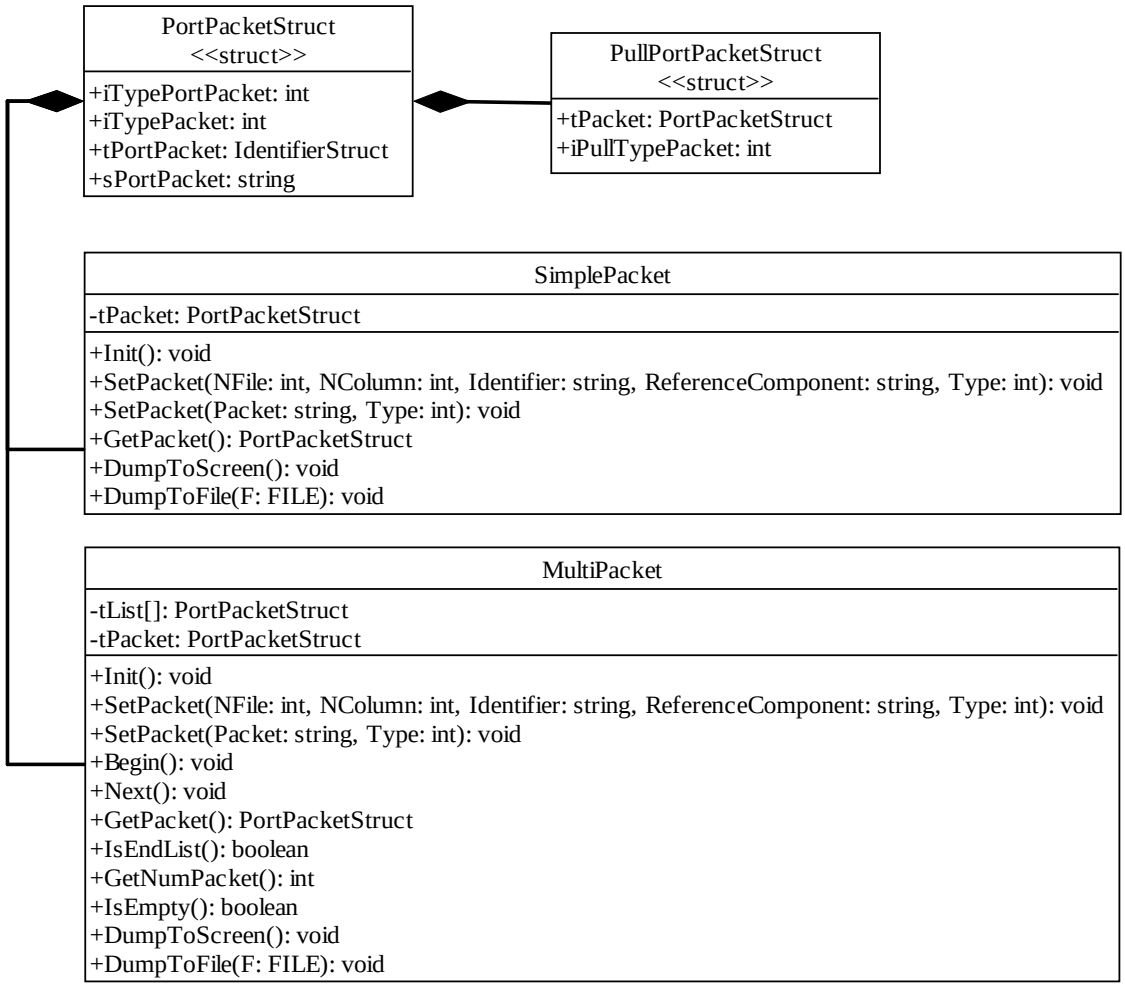


Figura 32: Clases *SimplePacket* y *Multipacket* perteneciente al subsistema *Packet*.

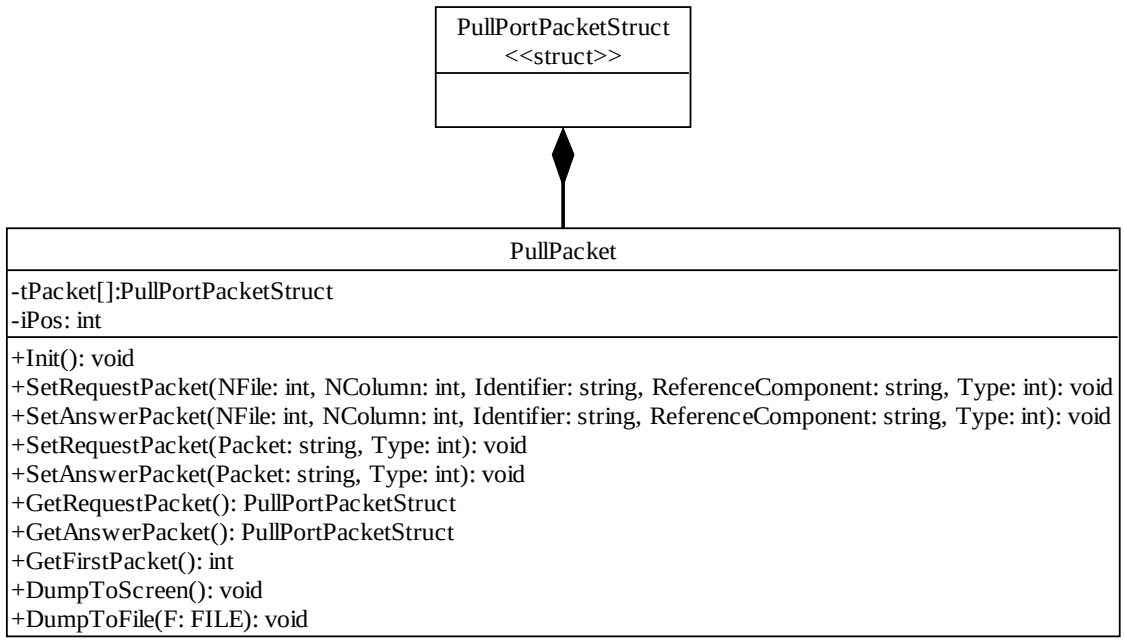


Figura 33: Clase *PullPacket* perteneciente al subsistema *Packet*.

En la Figura 34 podemos ver el diagrama de clases del modulo *Variable* que gestiona la definición de variables. Este modulo esta formado por las siguientes clases:

- Clase *Variable*: Se trata de una clase contenedora que almacena la información de definición de una variable.
- Clase *VariableInterface*: Implementación del patrón Facade que gestiona el subsistema *Variable*.

Este modulo hace uso de la clase *Identifiers* que hemos visto anteriormente en la Figura 24.

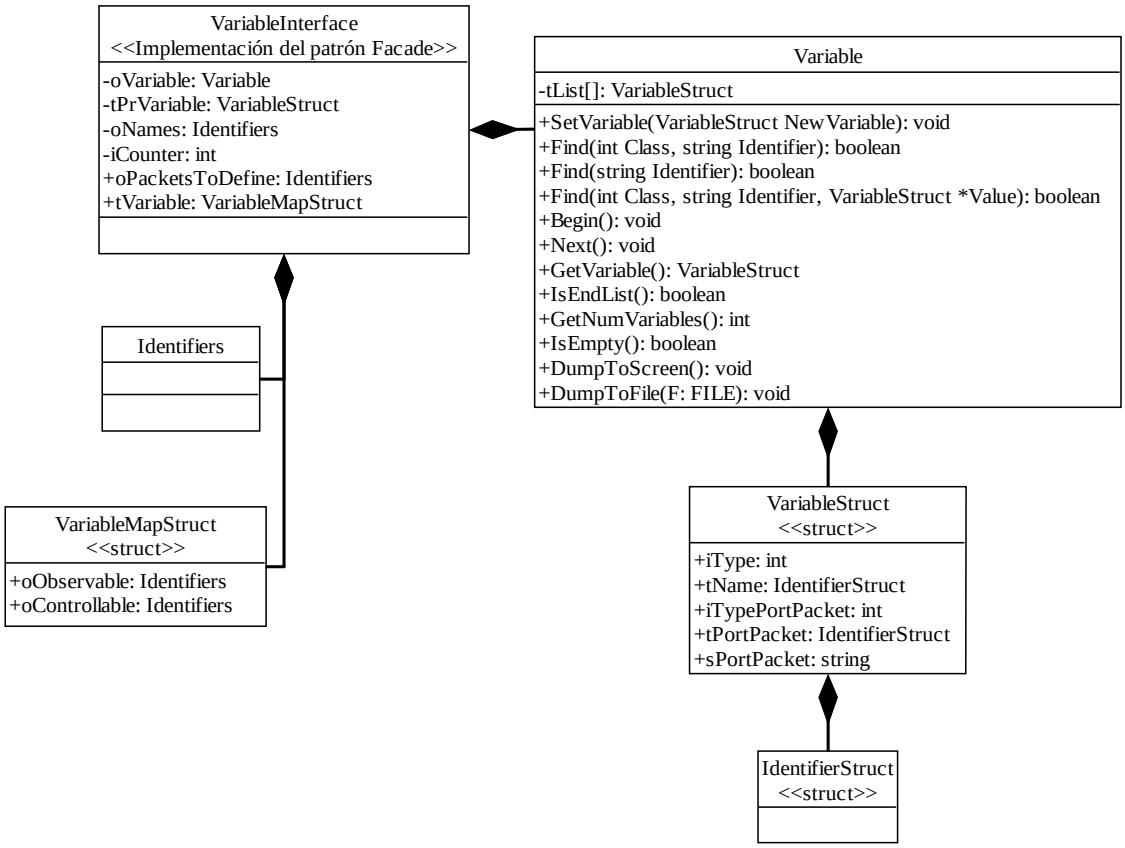


Figura 34: Subsistema *Variable* perteneciente al módulo *Semantic*.

En la Figura 35 se observa el subsistema *Exception* que gestiona la definición de excepciones. Este subsistema esta formado por las siguientes clases:

- Clase *Exception*: Se trata de una clase contenedora que almacena la información de definición de excepciones.
- Clase *ExceptionInterface*: Implementación del patrón Facade que gestiona el subsistema *Exception*.

Este modulo hace uso de la clase *Identifiers* que hemos visto anteriormente en la Figura 24

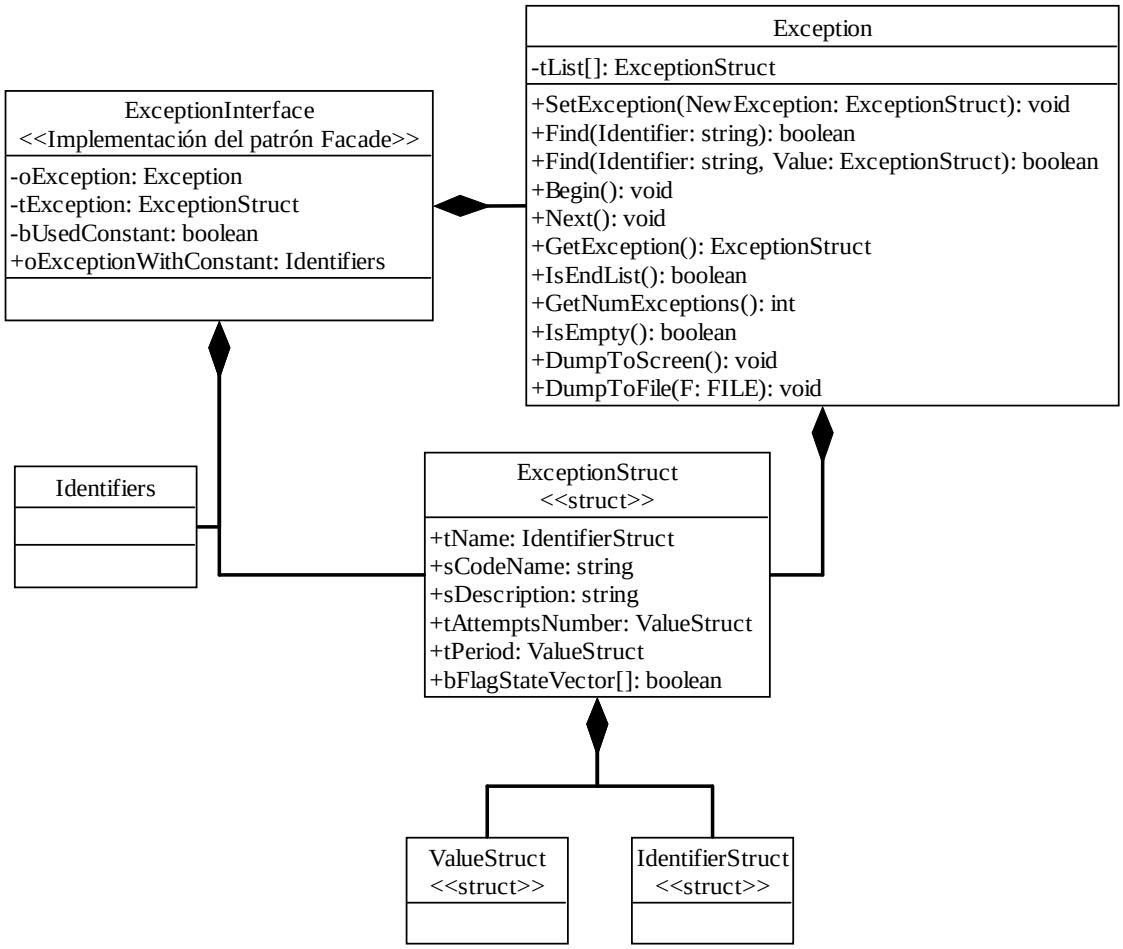


Figura 35: Subsistema *Exception* perteneciente al módulo *Semantic*.

En la Figura 36 se observa el subsistema *State* que gestiona la definición de estados. Este subsistema esta formado por las siguientes clases:

- Clase *State*: Se trata de una clase contenedora que almacena la información de definición de estados.
- Clase *StateInterface*: Implementación del patrón Facade que gestiona el subsistema *State*.

Este modulo hace uso de la clase *Identifiers* que hemos visto anteriormente en la Figura 24

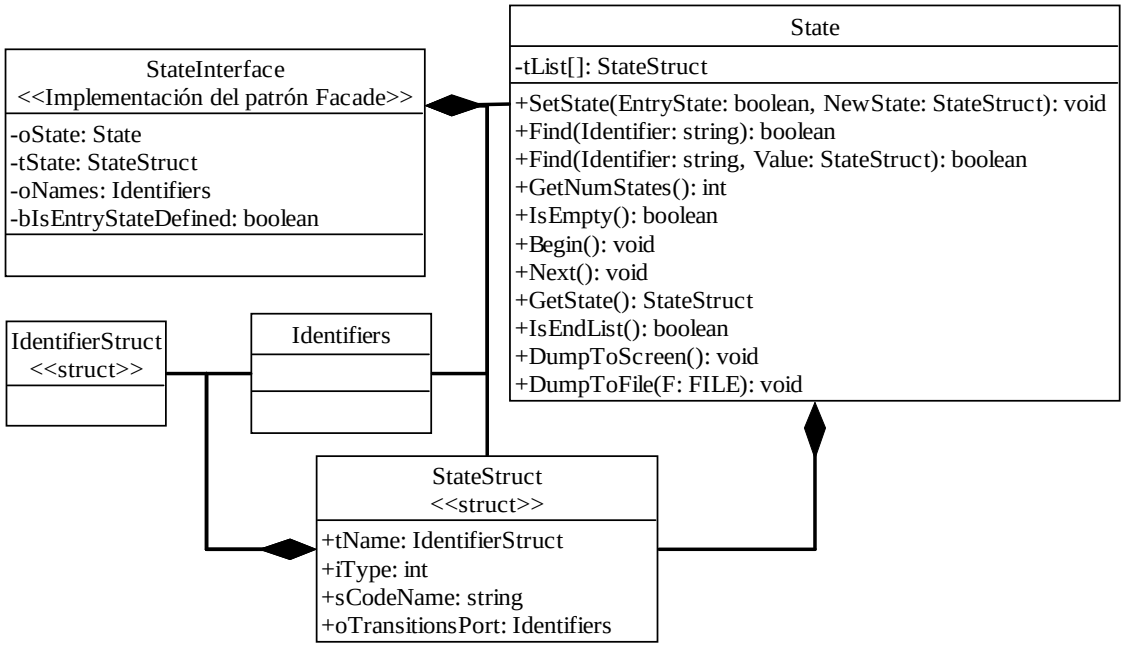


Figura 36: Subsistema *State* perteneciente al módulo *Semantic*.

En la Figura 37 se observa el subsistema *Thread* que gestiona la definición de hilos. Este subsistema esta formado por las siguientes clases:

- Clase *Thread*: Se trata de una clase contenedora que almacena la información de definición de hilos. Esta clase requiere del modulo *PortThread* que gestiona los puertos de entrada asociados a un hilo y que podemos ver en la Figura 38. Este modulo esta formado por las siguientes clases:
    1. Clase *InputBox*: Se trata de una clase contenedora que almacena información sobre los puertos de entrada asociados a hilos de clase *input box* (ver apartado 4.1.13.3). En la Figura 39 podemos ver el diagrama de clase de la clase *InputBox*.
    2. Clase *PriorityInputBox*: Se trata de una clase contenedora que almacena información sobre los puertos de entrada asociados a hilos de clase p*riority input box* (ver apartado 4.1.13.3). En la Figura 39 podemos ver el diagrama de clase de la clase *PriorityInputBox*.
- Clase *ThreadInterface*: Implementación del patrón Facade que gestiona el subsistema *Thread*.

Este modulo hace uso de la clase *Identifiers* que hemos visto anteriormente en la Figura 24.

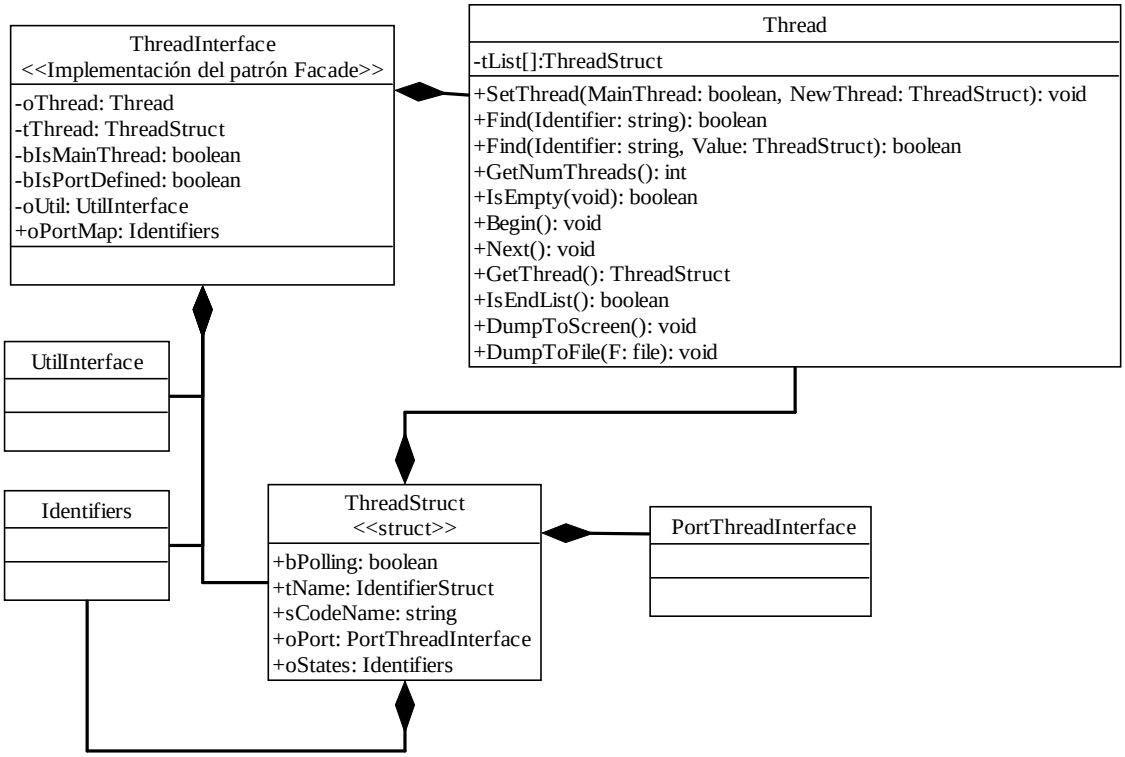


Figura 37: Subsistema *Thread* perteneciente al módulo *Semantic*.

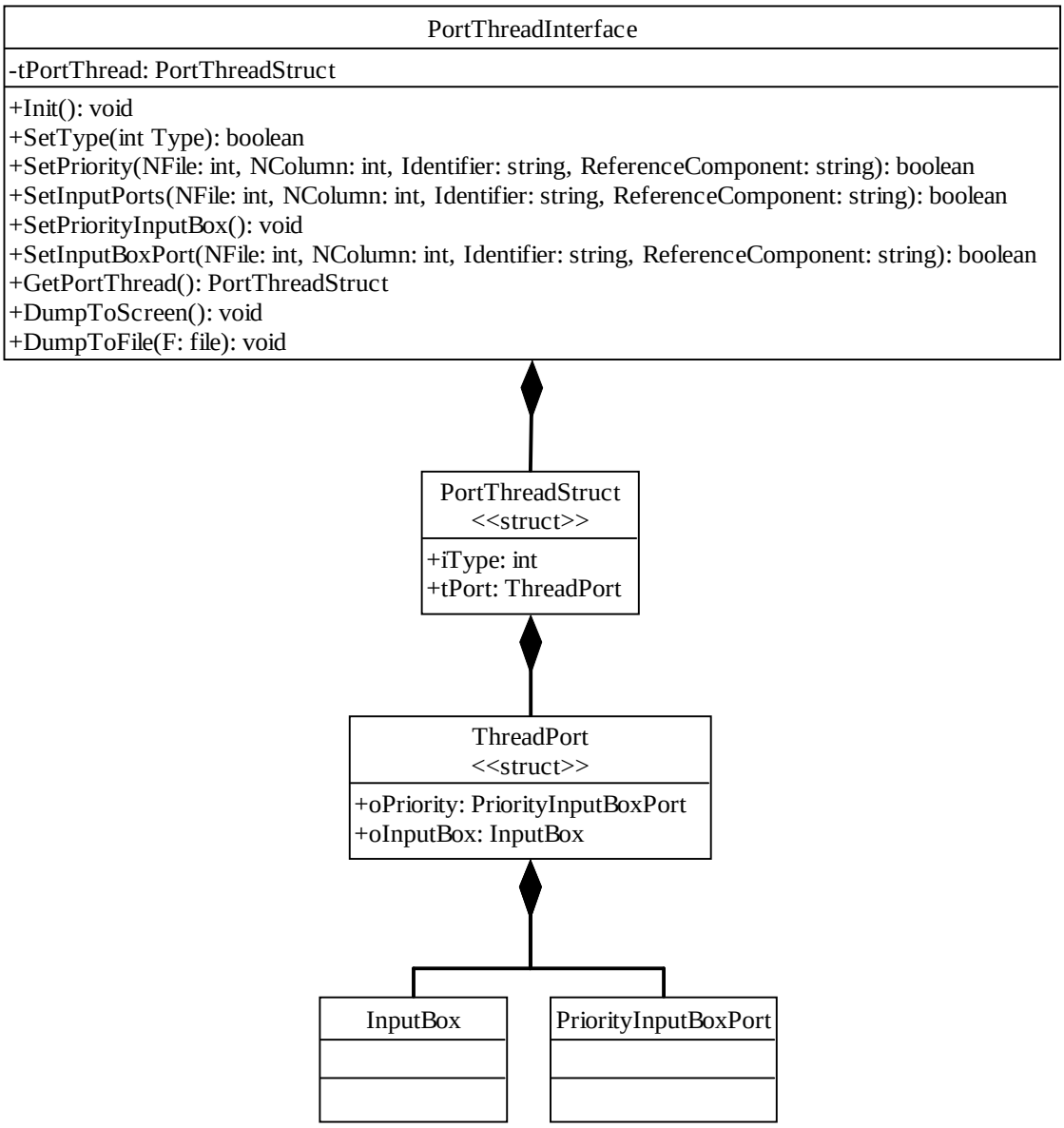


Figura 38: Subsistema *PortThread* perteneciente al subsistema *Thread*.

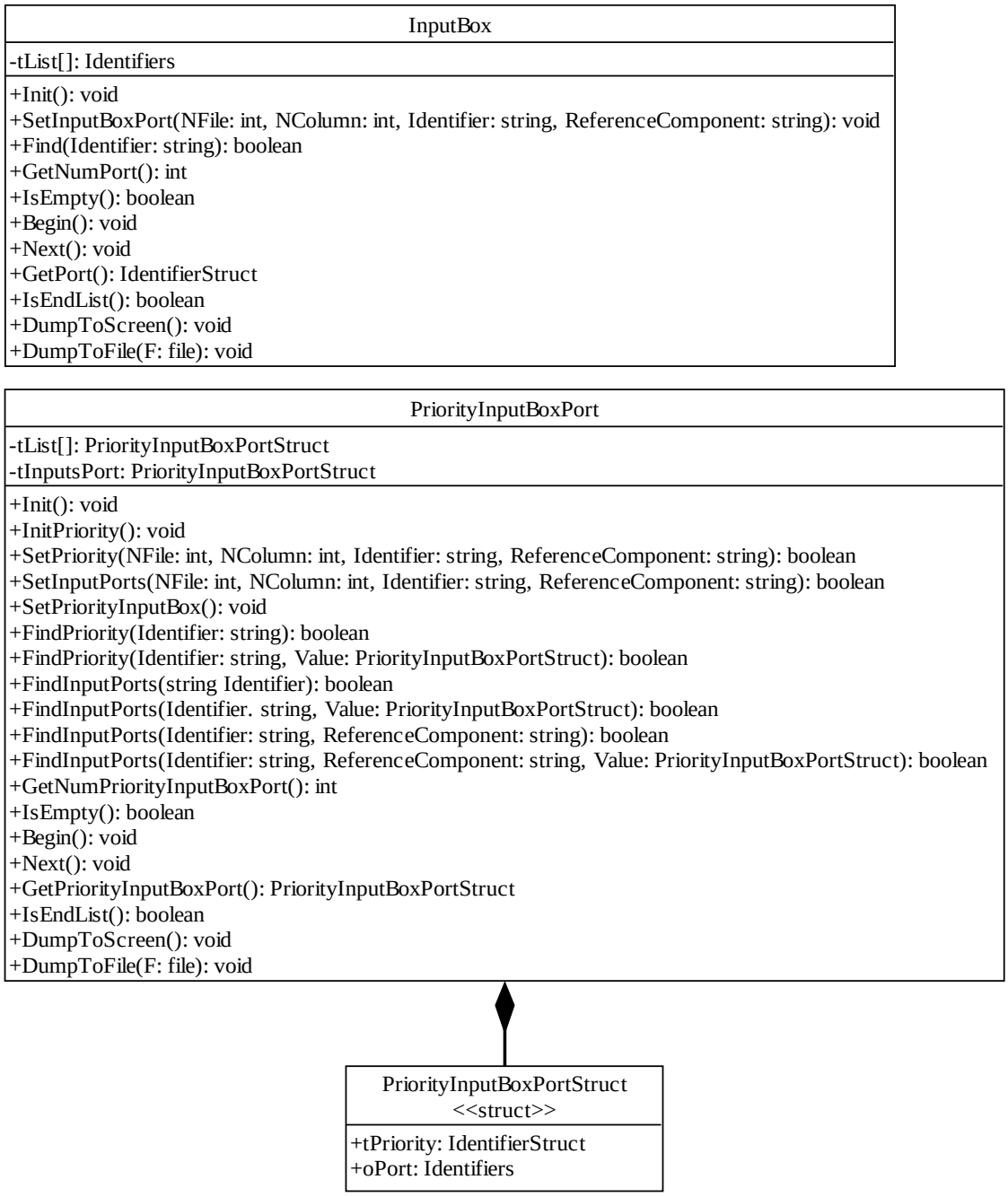


Figura 39: Clases *InputBox* y *PriorityInputBoxPort* del subsistema *PortThread*.

En la Figura 40 podemos ver el módulo Skeleton que gestiona la generación del esqueleto C++ y de la sintaxis XML.

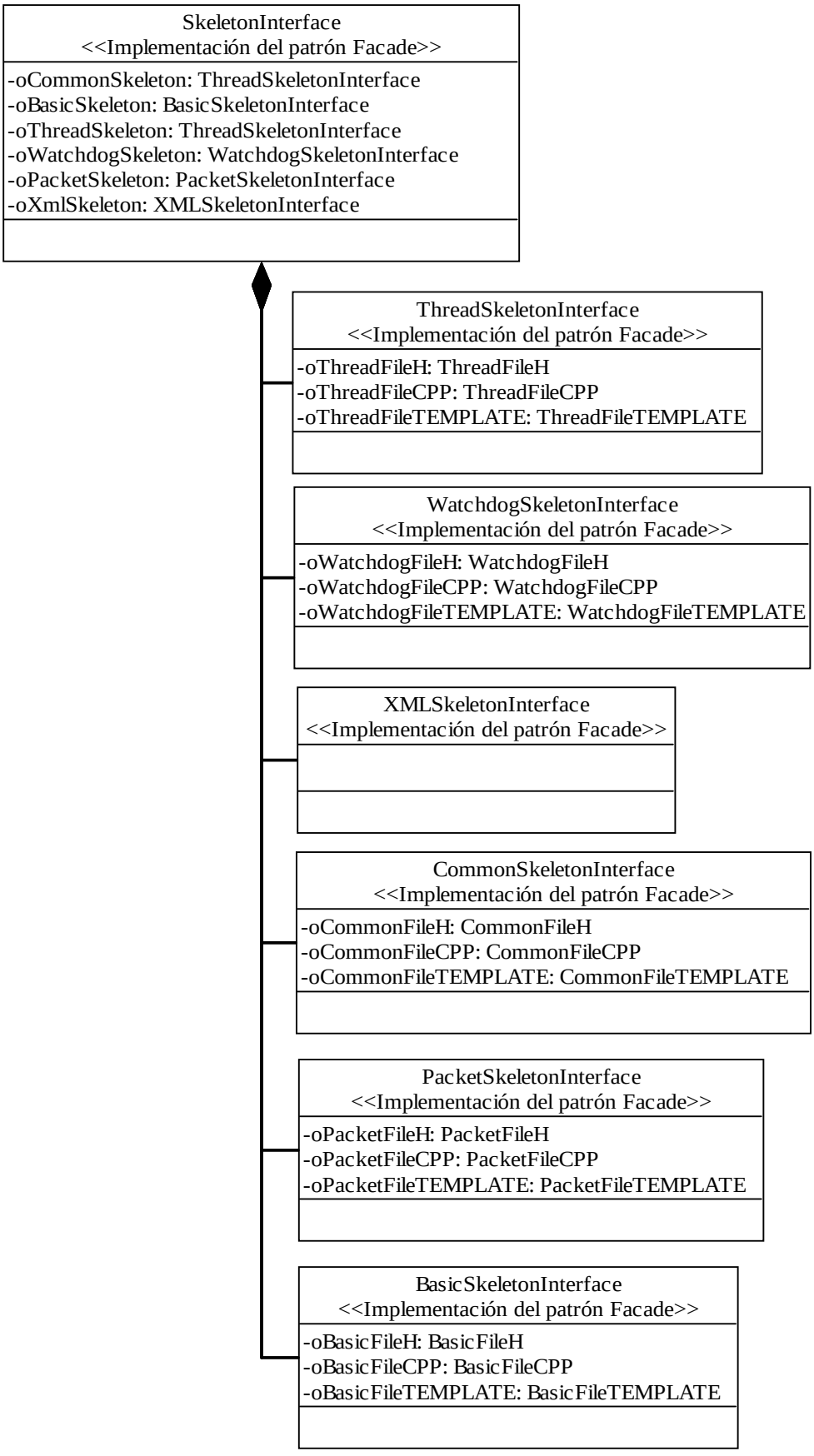


Figura 40: Módulo *Skeleton*.

5.4. Plantillas y copias de seguridad.

En el apartado 4.2.5 comentamos que los esqueletos C++ se podían dividir en dos partes o bloques: una parte que es común a todo componente y otra que varía en función de los requisitos del componente que se pretende desarrollar. También se indico que en esta última parte se podían distinguir cuatro modelos a través de los cuales se podía representar cualquier componente CoolBOT.

Partiendo de estos requisitos, en la implementación de los esqueletos C++ vamos a separar estas dos partes de la siguiente manera:

- *Parte común a todo componente*. Por cada modelo descrito en el apartado 4.2.5 se dispondrá de un fichero plantilla que contendrá trozos de esqueletos C++ que son comunes a todo componente y propios de ese modelo. Además de los trozos de esqueletos, en estos ficheros se introducirán una serie de marcas que especifican que zonas son las que varían en función de los requisitos en el esqueleto. Esto último, nos permite usar estas plantillas como guías que el compilador empleará para saber que parte de los esqueletos C++ debe generar en cada momento.
- *Parte que varía en función de los requisitos del componente*. Esta parte se encarga de generarlo el módulo *Skeletons* mostrado en el anterior apartado.

Para el caso de la generación de esqueletos XML no se da lo mencionado anteriormente, debido a que se trata de un simple proceso de traducción de lenguaje DL a sintaxis XML de un documento DLXML. El módulo *Skeletons* se encargará de realizar esta traducción.

Por último, para evitar la perdida de información en las recompilaciones (compilar un componente que ha sido compilado previamente), cada compilador dispondrá de un sistema de copias de seguridad. De este modo, si por equivocación compilamos un componente, que había sido compilado y completado posteriormente por el usuario, no se perdera la información de ese componente. En la Figura 41 se muestra el funcionamiento de este sistema de copias.

Cada vez que se detecta que ha habido una compilación previa, se procede a realizar una copia de los ficheros .h, .cpp y .xml generados en la anterior compilación. Estos ficheros son copiados añadiéndole una extensión que está formada por la fecha y hora del sistema y la extensión ".old". En la Figura 42 podemos ver un ejemplo de este sistema de copias se seguridad. Por cada recompilación se copian los ficheros .h, .cpp y

.xml de la compilación previa siguiendo el formato especificado. Este sistema de copias de seguridad es aplicado tanto en el compilador *coolbotc* como en el compilador *xmlcoolbotc*

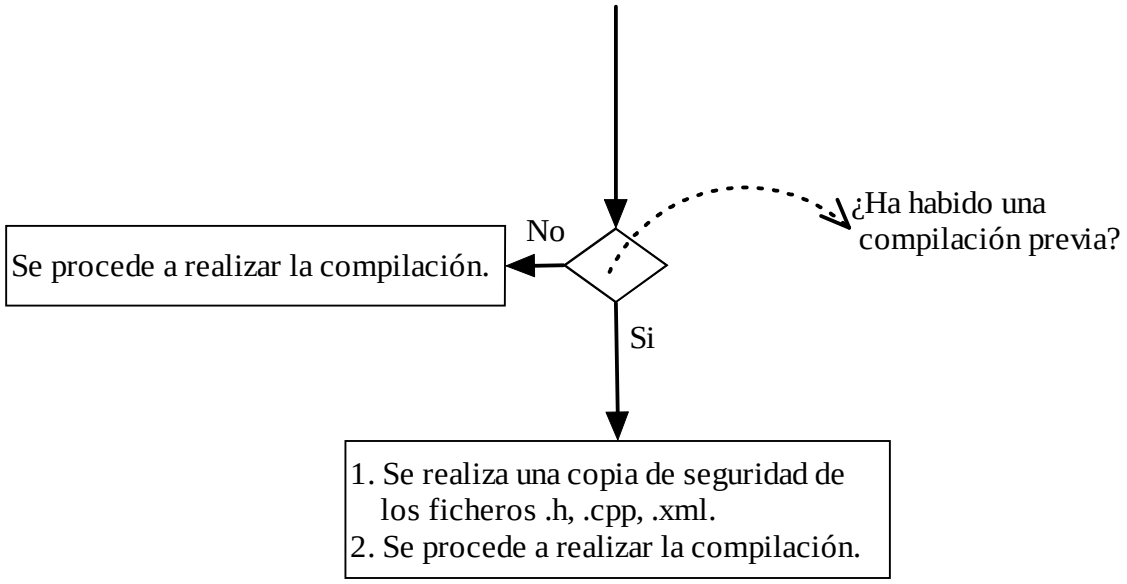


Figura 41: Sistema de copias de seguridad.

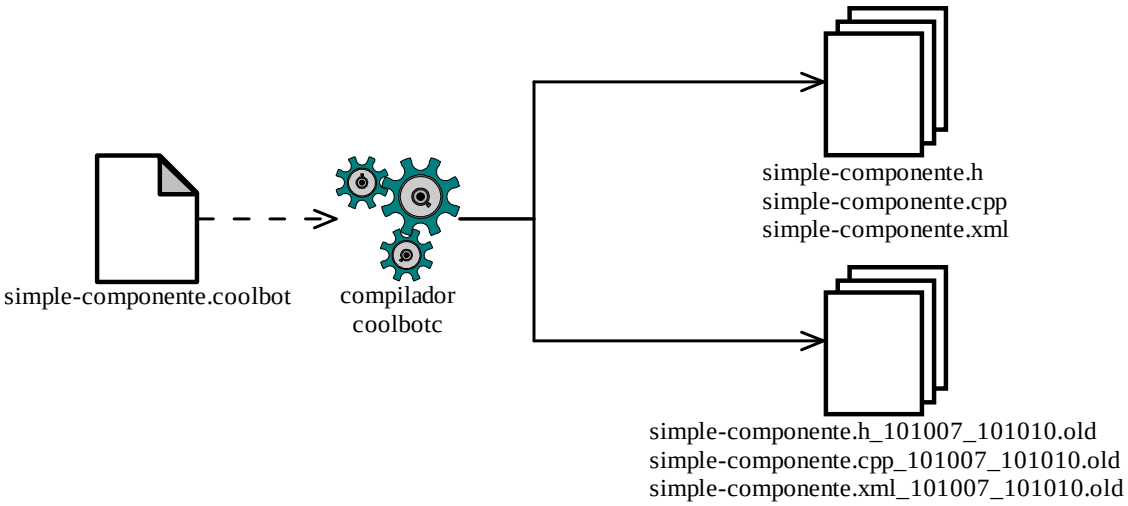


Figura 42: Recompilación del componente *SimpleComponent* en el compilador *coolbotc*.

Capítulo 6
Implementación.

La fase de implementación es un proceso de codificación donde se traducen los requisitos especificados en el apartado 4.2 siguiendo la organización establecida en el apartado anterior. En este apartado hablaremos sobre las herramientas que empleamos para llevar a cabo la implementación.

6.1. Notación Húngara.

La notación Húngara [wikipedia] fue inventada por el programador de Microsoft Charles Simonyi y consiste en una serie de reglas para dar nombres a las variables. Entre esas reglas destaca las siguientes:

- Iniciar el identificador con una o varias letras minúsculas que indiquen el tipo al que pertenece.
- Abreviación de cierto número de palabras (cada una comenzando en mayúsculas) describiendo el contenido de la variable.
- Las macros, las constantes y las enumeraciones se escriban en mayúsculas y separadas por "_" si están formadas por varias palabras.

Aplicando esta notación evitamos perder mucho tiempo en establecer los identificadores de las variables. Por esa razón se ha empleado en este proyecto.

6.2. Flex y Bison.

Flex y Bison son dos herramientas a través de las cuales vamos a desarrollar el analizador léxico y sintáctico de los compiladores *coolbotc* y *xmlcoolbotc*. A continuación se pasa a realizar una breve descripción de cada uno.

6.2.1. Flex.

Flex es un una herramienta que permite el análisis léxico de patrones lingüísticos en un determinado texto por lo que podemos desarrollar analizadores léxicos a través de él. Para ello se emplean una serie de patrones (conjunto de expresiones regulares) de los que está compuesto el texto a analizar, junto con unas acciones asociadas a cada uno de los patrones. Este esquema de patrón-acción se denomina en Flex regla y se escriben en un fichero con extensión .l. Cuando Flex reconoce un patrón ejecuta la acción asociada a dicho patrón. Esta acción se escribe en lenguaje C ya que ese es el lenguaje que posee el analizador léxico que genera Flex.

En la Figura 43 podemos ver la estructura del fichero que contiene las reglas que emplea Flex. A continuación se describe cada sección:

- *Sección de declaraciones*. Es el lugar para definir macros y para importar los archivos de cabecera escritos en C. También es posible escribir cualquier código de C aquí, que será copiado en el archivo fuente generado, y se pueden incluir atajos para definir patrones de la *Sección de reglas*, por ejemplo en vez del patrón *[0-9]** (cero o más dígitos que reconocerían cualquier número natural), se puede definir en esta sección el atajo:

  *numeros [0-9]**

  así, en la sección de código pondríamos el patrón *{numeros} {acción_en_C;}*. Con esto se clarifica la escritura del código en Flex.
- *Sección de reglas*. Es la sección más importante; asocia patrones a sentencias de C. Los patrones son simplemente expresiones regulares. Cuando Flex encuentra un texto en la entrada que es similar a un patrón dado, ejecuta el código asociado de C.
- *Sección de código*. Contiene sentencias en C y funciones que serán copiadas en el archivo fuente generado. Estas sentencias contienen normalmente el código llamado por las reglas en la sección de las reglas. En programas grandes es más conveniente poner este código en un archivo separado y enlazarlo en tiempo de compilación.

El fichero que genera Flex con el analizador léxico es un fichero en lenguaje C que se denomina *lex.yy.c*.

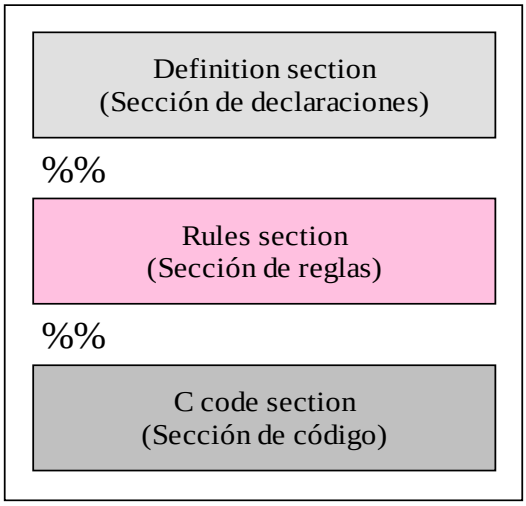


Figura 43: Estructura en que organiza Flex un fichero de reglas.

6.2.2. Bison.

Bison es una herramienta que genera analizadores sintácticos de propósito general que convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR [Pérez-Aguiar 1998], en un programa en C que analiza esa gramática.

Un archivo fuente de Bison (normalmente un fichero con extensión .y) describe una gramática. La estructura de este fichero la podemos observar en la Figura 24. Como podemos apreciar en la Figura 44, el fichero donde se describe la gramática esta formado por las siguientes secciones:

- *Sección de declaraciones*. Es el lugar para definir macros, para importar los archivos de cabecera, declaraciones de variables globales,… escritos en C. El contenido de esta sección se copia literalmente al principio del fichero que Bison genera.
- *Sección de declaraciones de Bison*. En esta sección se puede declarar el tipo de la variable *yylval*, los símbolos (terminales y no terminales) de la gramática, el axioma de la gramática, y la asociatividad y precedencia de los operadores.
- *Sección de reglas de gramática*. Es la sección más importante del fichero de especificación. Contiene las reglas de la gramática escritas en un formato concreto y con acciones asociadas a las reglas opcionalmente. Las acciones son un conjunto de instrucciones C encerradas entre llaves y que se ejecutan cada vez que se reconoce una instancia de la regla. Normalmente se sitúan al final de la regla, aunque también se admiten en otras posiciones de la regla. La mayoría

de las acciones trabajan con los valores semánticos de los símbolos de la parte derecha, accediendo a ellos mediante pseudovariables del tipo *$i* donde *i* representa la posición del símbolo en la regla. El valor semántico del símbolo no terminal de la parte izquierda de la regla se referencia con *$$*.

- *Sección de código*. En esta sección se ubican una serie de funciones y código C. En concreto, hay tres funciones que siempre tienen que aparecer:
    1. *Función main*. Función principal que invoca a la función *yyparse* la cual es la encargada de realizar el análisis sintáctico.
    2. *Función yylex*. Función invocada desde la función *yyparse* para obtener un nuevo *token* de la entrada.
    3. *Función yyerror*. Función invocada desde la función *yyparse* cuando ocurre un error sintáctico.

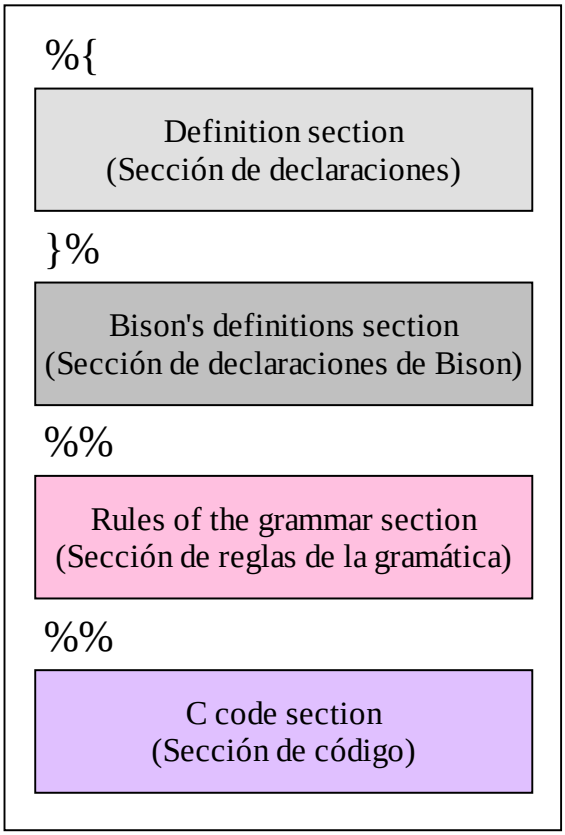


Figura 44: Estructura en que organiza Bison un fichero de reglas.

Bison genera un fichero en lenguaje C con la extensión *.tab.c* y otro, también en lenguaje C, con la extensión *.tab.h* con el analizador sintáctico. Estos ficheros junto con el fichero *lex.yy.c* se combinan con el resto de ficheros para generar los respectivos compiladores *coolbotc* y *xmlcoolbotc*.

6.3. Librería STL de C++.

Una librería es un conjunto de recursos (algoritmos) prefabricados, que pueden ser utilizados por el programador para realizar determinadas operaciones. Los lenguajes de programación suelen tener una serie de bibliotecas integradas para la manipulación de datos a nivel más básico. En C++, además de poder usar las bibliotecas de C, se puede usar la STL (*Standard Template Library*), propia del lenguaje.

La librería STL proporciona una colección de estructuras de datos contenedoras y algoritmos genéricos (que permiten efectuar operaciones sobre el almacenado de datos, procesado y flujos de entrada/salida) y dispone de cinco componentes diferentes:

- *Algoritmos*. Proporciona una amplia variedad de algoritmos comunes entre los que se incluyen los de ordenación, búsqueda y algoritmos numéricos. Los algoritmos se pueden utilizar con estructuras contenedoras de datos de la librería como vectores, o estructuras de datos definidas por el usuario provistas de iteradores.
- *Iteradores*. Una generalización del concepto de puntero que permite al programador visitar cada elemento de una estructura contenedora de datos.
- *Contenedores*. Estructuras de datos que contienen, y permiten manipular, otras estructuras de datos.
- *Funciones objeto u objetos función*. Varios de los algoritmos proporcionados por la STL, permiten pasar una función al algoritmo para adecuar su funcionalidad. Las funciones objeto son una generalización del puntero a la función del C.
- *Adaptadores*. Adapta a la interfaz de un componente para que proporcione una interfase diferente. Esto permite transformar una estructura de datos en otra que tiene las mismas facilidades pero cuya interfaze proporciona un conjunto de operaciones diferente.

En este proyecto, se han utilizado el contenedor *list* para almacenar información semántica y de generación de código. El contenedor *list* esta implementada como una lista doblemente enlazada, por tanto se trata de un contenedor secuencial que requiere de un *iterador* para realizar operaciones de inserción, modificación y eliminación de datos.

Al tratase de un contenedor secuencial el coste computacional de varias funciones miembros es de *O(N)*. Esto quiere decir que si en el contenedor hay N elementos, en un proceso de búsqueda se tardará, en el peor de los casos, lo que se tarde

en recorrer esos N elementos. Aún así, para este proyecto, es la mejor solución a emplear ya que simplifica bastante el proceso de desarrollo. En la Tabla 16 podemos ver las funciones miembros del contenedor *list* que empleamos durante la implementación de los compiladores.

Contenedor *list* de STL	
Método	**Descripción**
size	Devuelve el número de elementos almacenados actualmente en el contenedor. Función de coste computacional O(N).
empty	Devuelve verdadero si el contenedor se encuentra vacío y falso en caso contrario.
push_back	Añade el elemento al final del contenedor.
push_front	Añade el elemento al principio del contenedor.
begin	Devuelve un iterador que referencia el inicio del contenedor.
end	Devuelve un iterador que referencia el final del contenedor.
clear	Elimina todos los elementos del contenedor.

Tabla 16: Funciones miembro del contenedor list.

Además de emplease en este proyecto el contenedor *list*, se ha empleado la clase *string* para el manejo de cadenas de caracteres. Para más información sobre el contenedor list y la clase string consultar [c++ Ref] y [c++].

Capítulo 7

Prueba y Resultados.

La fase de 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 los compiladores desarrollados en este proyecto cumplan con unos mínimos de calidad y para ello se ha divido esta fase en dos partes:

- *Pruebas de incremento*: Como vimos en el apartado 3.1, en el desarrollo de los compiladores hemos empleado, como paradigma de desarrollo, el modelo incremental que nos permite ir disponiendo de partes funcionales de los compiladores (incrementos) sin tener que esperar hasta el final del proceso de desarrollo. De esta forma por cada incremento obtenido se ha realizado, en exclusiva, una fase de prueba consistente en aplicar a ese incremento una batería de pruebas. En el Fragmento de Código 51 podemos ver una prueba realizada sobre el incremento que nos genera el analizador léxico.

```
/*
 * File: test4.coolbot
 * Description: Verifica la sensibilidad de mayúsculas y minúsculas.
 * Date: 19 February 2007
 * Author: Francisco J. Santana Jorge (francisco.jesus.santana@gmail.com)
 *         Dpto. Informatica y Sistemas - Universidad de Las Palmas de Gran Canaria
 */
component Test4
{
    /* Las palabras reservadas las trata como identificadores al estar escritas en
       mayúsculas. */
     INPUT PORT Period TYPE LAST PORT PACKET PacketLong;
};
```

Fragmento de Código 51: Ejemplo de prueba para el analizador léxico.

- *Pruebas con componentes reales*. Con las pruebas de incremento nos aseguramos que los compiladores cumplan con los objetivos marcados en este

proyecto, pero las pruebas realizadas en cada incremento son agrupaciones de código sin sentido. Para proporcionar una mayor calidad al proyecto, se realizo unas pruebas con componentes reales. De este modo, al mismo tiempo que nos aseguramos que los compiladores funcionan correctamente, podemos evaluar los resultados que nos genera los compiladores frente al modelo tradicional de desarrollo de componentes CoolBOT.

En este capítulo nos centraremos en las pruebas con componentes reales. De entre las pruebas que se realizaron se escogerá una que comentaremos detalladamente.

7.1. Pruebas con componentes reales.

Descripción del componente compuesto con las capturas de pantalla (decir este componente hace tal cosa y esta formado por estos componentes atómicos que realizan tal cosa acompañado de capturas).

7.1.1. Componente PlayerRobotC.

Descripción de cada fase de la metodología de desarrollo de CoolBOT.

7.1.2. Componente GridMapC.

Descripción de cada fase de la metodología de desarrollo de CoolBOT.

7.1.3. Componente NDNavigationC.

Descripción de cada fase de la metodología de desarrollo de CoolBOT.

7.1.4. Componente ShortTermPlannerC.

Descripción de cada fase de la metodología de desarrollo de CoolBOT.

7.1.5. Resultados obtenidos.

Compilación del componente compuesto y evolución de los resultados obtenidos.

Capítulo 8

Conclusiones y trabajo futuro.

En este capítulo hablar sobre las conclusiones obtenidas en el proyecto de fin de carrera y sobre que otros trabajos se pueden realizar a partir de este proyecto.

Apéndice A

Manual de usuario.

1. Instalación.

En la Figura 45 podemos observar como esta organizado el código fuente de este proyecto. Como vemos disponemos de un directorio *models* que contiene las plantillas modelos (ver apartados 4.2.5 y 5.4) y de un directorio *source* donde se encuentra la implementación del compilador *coolbotc* y *xmlcoolbotc*. Luego aparecen varios ficheros que contienen la lógica necesaria para generar los respectivos binarios.

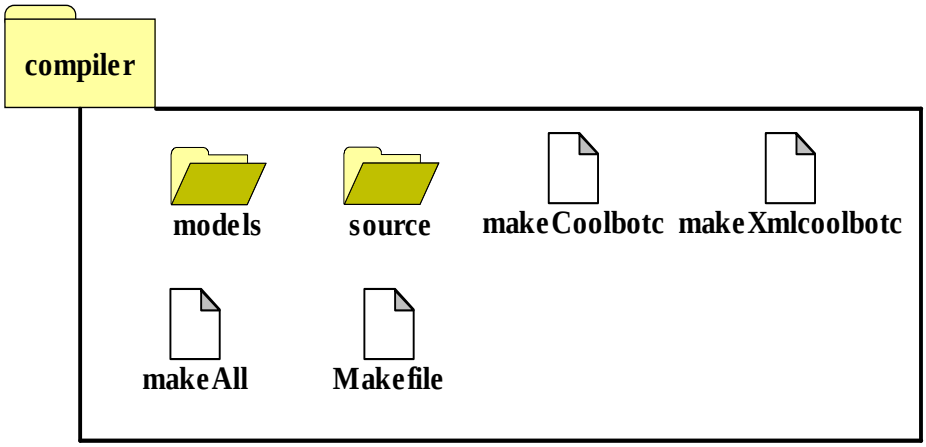


Figura 45: Organización del código fuente.

El código fuente se encuentra almacena en el repositorio *cvs*[8] del SIAS (máquina *sias.iusiani.ulpgc.es)* del Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería de la Universidad de Las Palmas de Gran Canaria y se necesita disponer de una cuenta en esa máquina para poder descargarlo. Para obtener una cuenta en esa máquina se ha de dirigir una solicitud al administrador del repositorio *cvs* del SIAS[9].

[8] http://www.cvshome.org/
[9] adominguez@iusiani.ulpgc.es

Para descargar el código fuente, simplemente hay que ejecutar la siguiente orden cvs (se requiere el software *cvs* instalado en la máquina donde se ejecuta esta orden):

cvs -d :ext:USUARIO@sias.iusiani.ulpgc.es:/home/cvsadmin/repository checkout coolbot-compiler

donde USUARIO es el nombre usuario proporcionado por el administrador del repositorio *cvs* del SIAS.

Una vez descargado el código fuente, pasamos a compilarlo. Para ello se sigue los siguientes pasos:

1. Situarnos en el directorio donde se encuentra el código fuente del compilador.
2. Invocar al script *makeAll*.

   Por ejemplo**:**

   ```
   $ cd coolbotc-compiler
   $ ./makeAll
   ```

3. Establecer la variable de entorno *COOLBOT_COMPILER_PATH*. Para ello simplemente invocamos a uno de los compiladores con la opción –p para indicar la ruta donde se encuentran instalado los compiladores (donde se han compilado).

   Por ejemplo:

   ```
   $ pwd
   /coolbotc-compiler
   $ ./coolbotc –p /coolbotc-compiler
   ```

4. Por último verificamos si los compiladores se instalaron correctamente. Para ello simplemente invocamos a cada uno de los compiladores. Por ejemplo:

```
$ coolbotc
coolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
Usage: coolbotc [options] [file]
Options:
       -h, --help: show this help.
       -v, --verbose: verbose mode on stdin.
       -vf, --verbosefile: verbose mode on file.
       -p, --path: compiler path.
       -c, --create: create the structure of directories of the platform COOLBOT.
                     Usage: (-c | --create) Name-dir Name-Componente
Examples:
          coolbotc --help
          coolbotc test.coolbot
          coolbotc -vf test.coolbot
          coolbotc -p /home/compiler
          coolbotc -c first-component FirstComponent
DONE
```

2. Modo de uso del compilador coolbotc.

El formato de invocación del compilador *coolbotc* es el siguiente:

coolbotc [opciones] [fichero-a-compilar]

y las opciones disponibles son las siguientes:

- *-h ó --help*. Muestra la información de ayuda. Si se invoca al compilador sin ningún argumento, también se muestra esta información. Por ejemplo:

```
$ ./coolbotc
coolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
Usage: coolbotc [options] [file]

Options:
   -h, --help: show this help.
   -v, --verbose: verbose mode on stdin.
   -vf, --verbosefile: verbose mode on file.
   -p, --path: compiler path.
   -c, --create: create the structure of directories of the platform COOLBOT.
                 Usage: (-c | --create) Name-dir Name-Component
Examples:
        coolbotc --help
        coolbotc test.coolbot
        coolbotc -vf test.coolbot
        coolbotc -p /home/compiler
        coolbotc -c first-component FirstComponent
```

- *(-vf | --verbosefile) fichero-a-compilar*. Activa el modo *verbose* en un fichero redireccionando la salida del verbose mode a un fichero. Este fichero tendrá como nombre el del fichero a compilar más la extensión *.output*. Por ejemplo:

```
$ coolbotc --verbosefile prueba.coolbot
coolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
File verbose mode: on
DONE.
```

- *(-p | --path) ruta*. Especifica la ruta donde se encuentra instalado el compilador. El compilador requiere de una variable de entorno para su funcionamiento que indica la ruta donde se encuentra instalado. Esta variable se denomina *COOLBOT_COMPILER_PATH*. Una vez asignada la ruta, deberemos reiniciar el *shell* para que los cambios tengan efectos. Por ejemplo:

```
$ coolbotc --path /compiler
coolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
Restartig the Bash shell changes will have effect.
DONE.
```

- *(-v | --verbose) fichero-a-compilar*. Activa el modo *verbose* en pantalla. Por ejemplo:

```
$ coolbotc --verbose prueba.coolbot
coolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
Stdin verbose mode: on
   Line 1, Column 0 : RESERVED WORD: component
   Line 1, Column 11 : IDENTIFIER: prueba
   Line 2, Column 0 : SYMBOL: {
   Line 3, Column 3 : RESERVED WORD: entry
   Line 3, Column 10 : RESERVED WORD: state
   Line 3, Column 17 : IDENTIFIER: Main
   Line 4, Column 3 : SYMBOL: {
   Line 6, Column 3 : SYMBOL: }
   Line 6, Column 5 : Syntactic rule detected: definition of user automaton state.
   Line 6, Column 4 : SYMBOL: ;
   Line 7, Column 0 : SYMBOL: }
   Line 7, Column 1 : SYMBOL: ;
   Line 7, Column 3 : Syntactic rule detected: definition of component.
   Content of Component:
       Component Name: prueba
       Header:
             Exists Header definition: NO.
       End Header.
       Port: 0 ports.
       End Port.
       Variables: 0 variables.
       End Variables.
       Exceptions: 0 exceptions.
       End Exceptions.
       Entry State:
           State Name: Main
           Transition on:
                  Identifiers found: 0 identifiers.
           End Identifiers.
       -------------------------------
       End Automaton States.
       Thread: 0 threads.
       End Thread.
   DONE.
```

- *(-c | --create) nombre-directorio nombre-componente*. Crea la estructura necesaria de directorios, ficheros y variables de entorno para poder compilar el componente generado sobre la plataforma CoolBOT (ver apartado 1.1). Esta opción crea un fichero con el nombre del componente y la extensión *.coolbot* que contiene la definición mínima exigida por el lenguaje DL para ser compilado por el compilador *coolbotc*. Cuando se ejecuta esta opción el compilador, automáticamente se crea la estructura de directorios, ficheros y variables de entorno que necesitamos para una vez desarrollado el componente y

obtenido los esqueletos poder compilarlo sobre la plataforma CoolBOT. Por ejemplo:

```
$ coolbotc simple-component SimpleComponent
coolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
DONE.
$ cd simple-component
$ ls
bin          makeMakeObjects.mak       simple-component-test.mak
examples     obj                       simple-component.coolbot
lib          simple-component-lib.mak
```

3. Modo de uso del compilador xmlcoolbotc.

El modo de uso de *xmlcoolbotc* es idéntico al de *coolbotc* con la salvedad de que no posee la opción *-c* ó *--create*. Por ejemplo:

```
$ xmlcoolbotc
xmlcoolbotc compiler - Version 1.1.0 - Francisco J. Santana Jorge
Usage: xmlcoolbotc [options] [file]

Options:
  -h, --help: show this help.
  -v, --verbose: verbose mode on stdin.
  -vf, --verbosefile: verbose mode on file.
  -p, --path: compiler path.

Examples:
          xmlcoolbotc --help
          xmlcoolbotc test.xml
          xmlcoolbotc -vf test.xml
          xmlcoolbotc -p /home/compiler

DONE.
```

Apéndice B

Diagramas sintácticos.

A continuación, se van a mostrar los diagramas sintácticos del lenguaje Description Language.

Figura 46: Inicio.

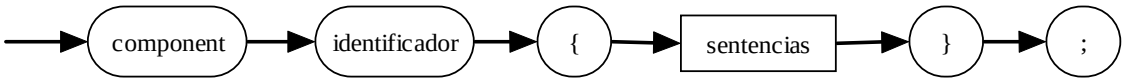


Figura 47: sentencias.

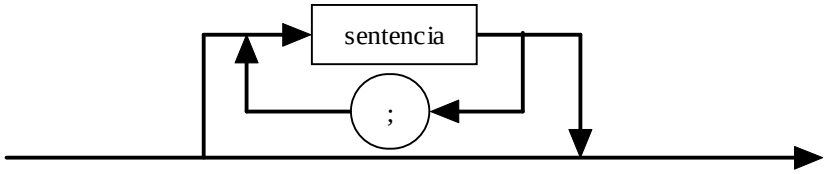


Figura 48: sentencia.

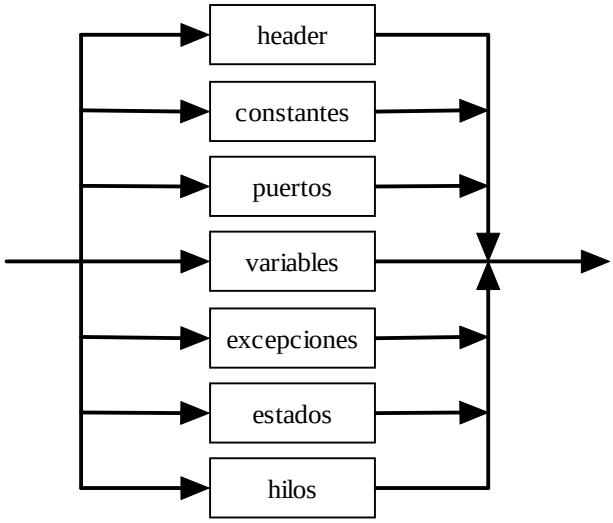


Figura 49: header.

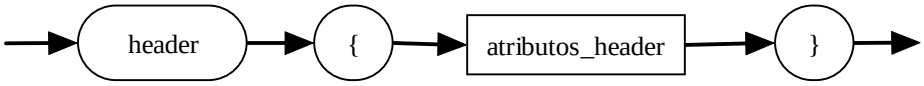


Figura 50: atributos_header.

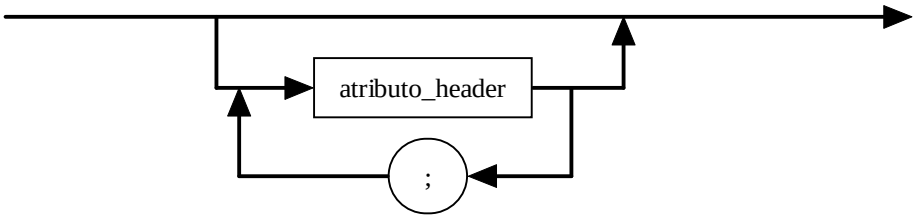


Figura 51: atributo_header.

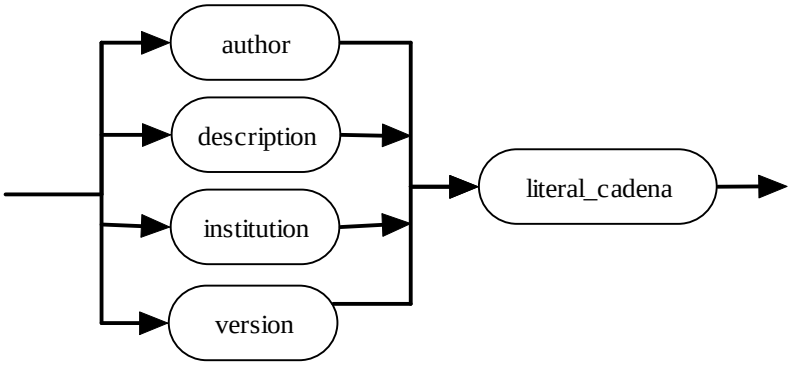


Figura 52: constantes.

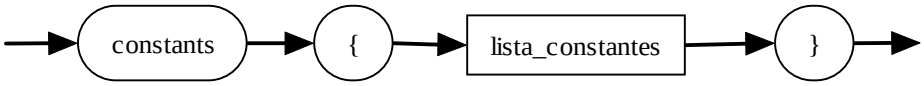


Figura 53: lista_constantes.

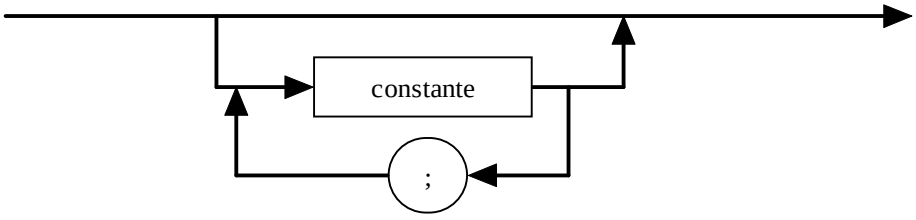


Figura 54: constante.

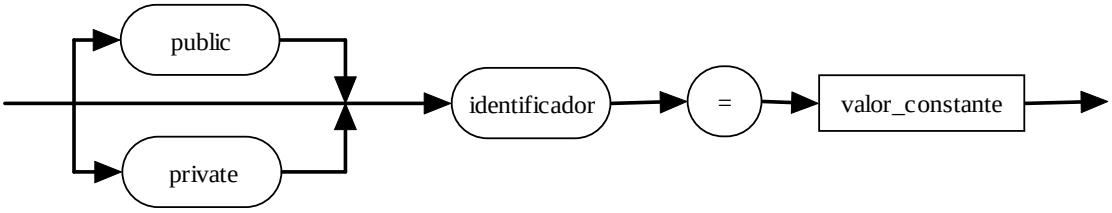


Figura 55: valor_constante.

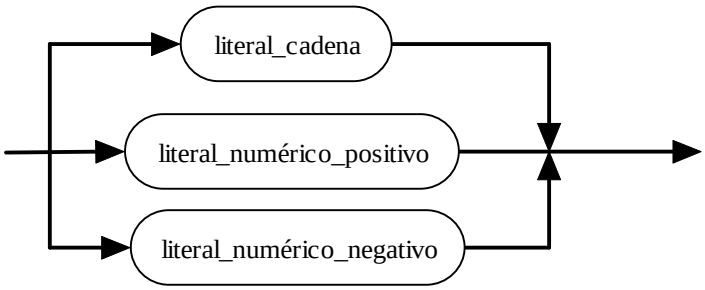


Figura 56: puertos.

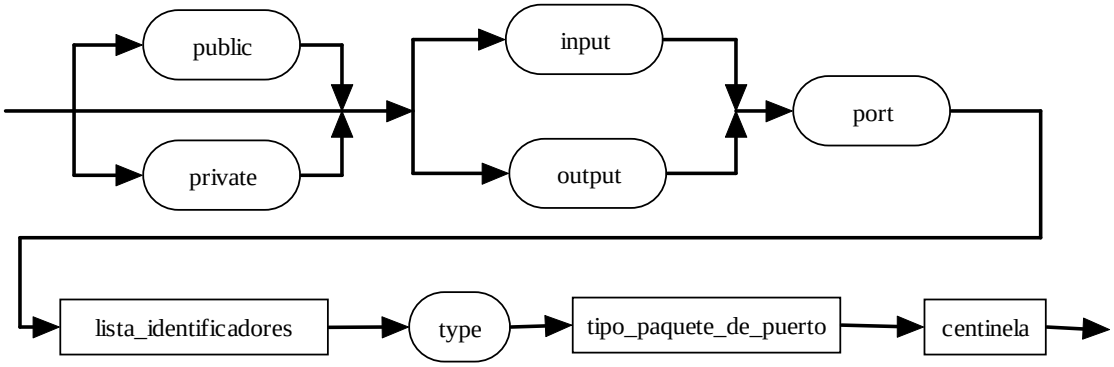


Figura 57: lista_identificadores.

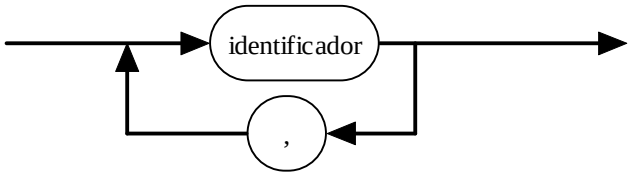


Figura 58: tipo_paquete_de_puerto.

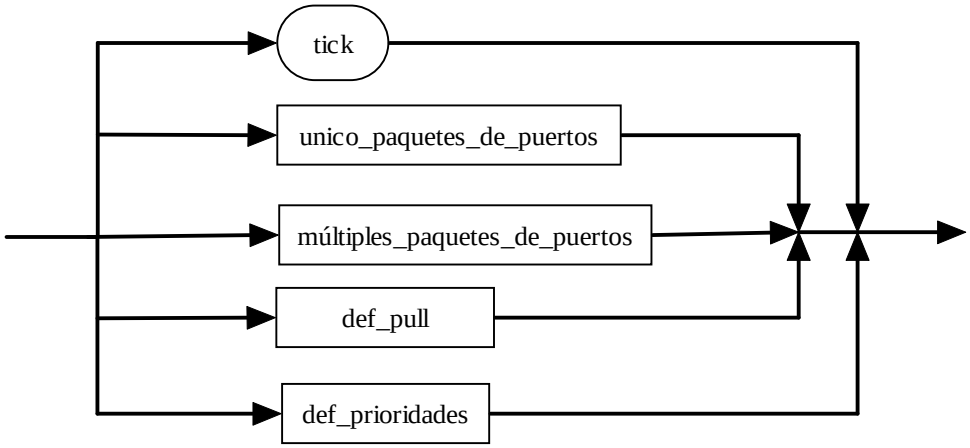


Figura 59: unico_paquetes_de_puertos.

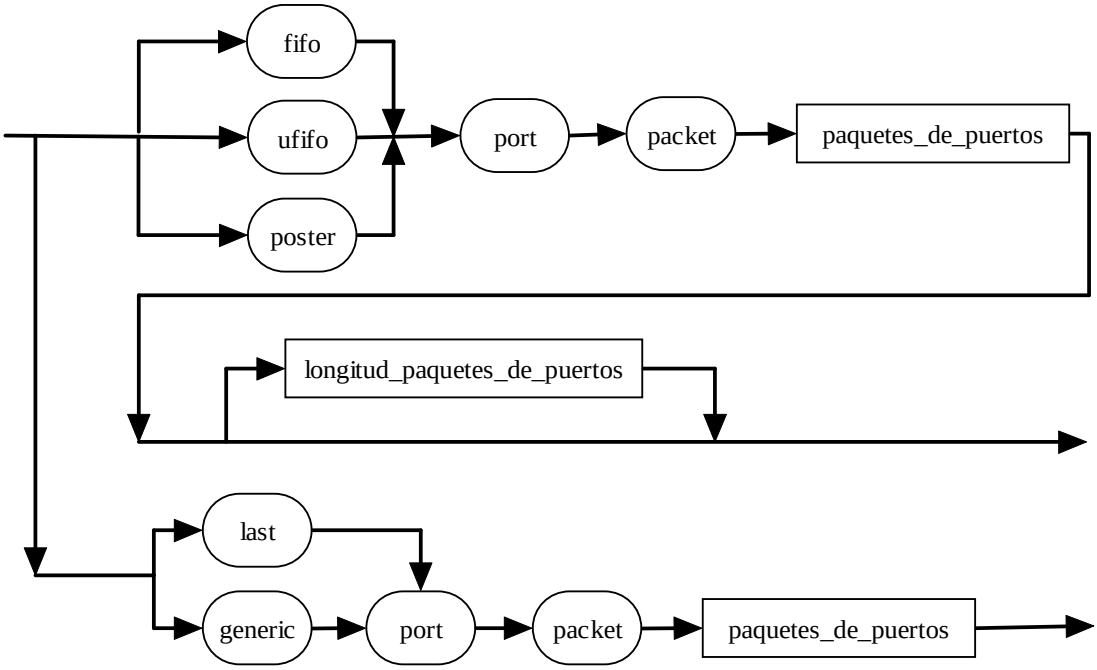


Figura 60: multiples_paquetes_de_puertos.

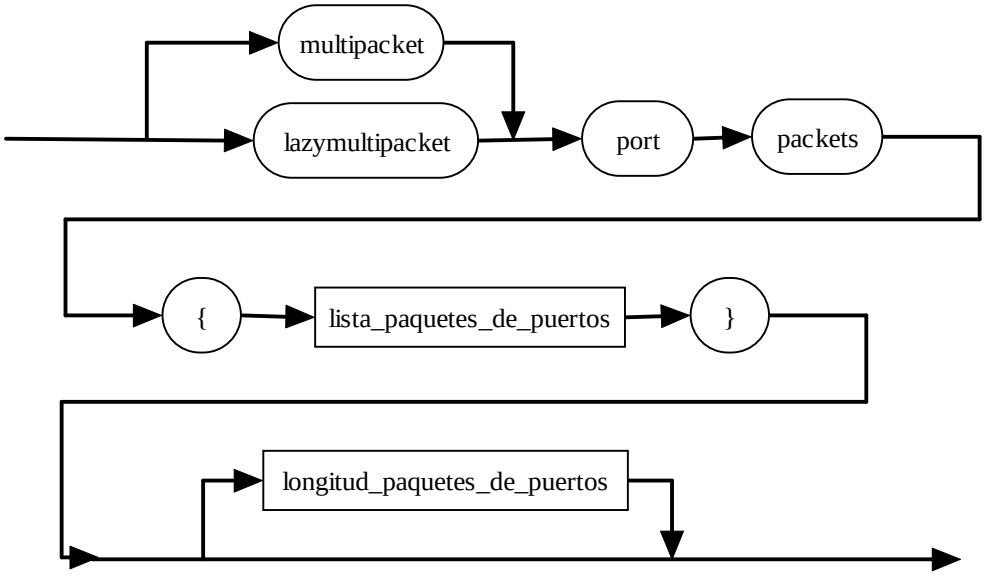


Figura 61: def_pull

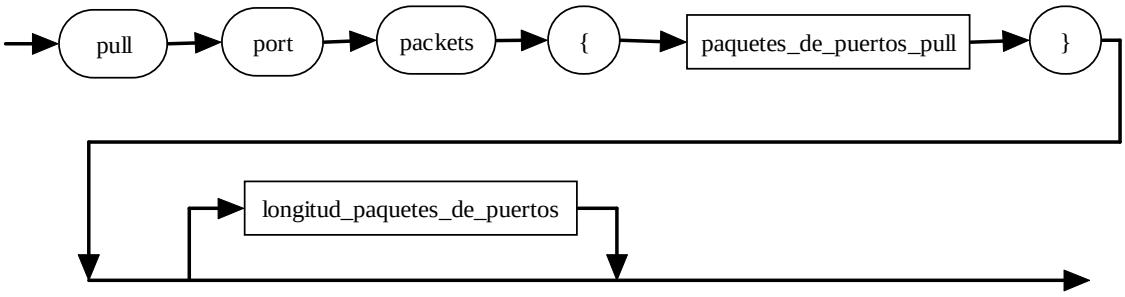


Figura 62: def_prioridades

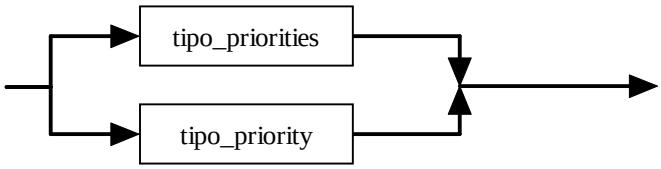


Figura 63: paquetes_de_puertos.

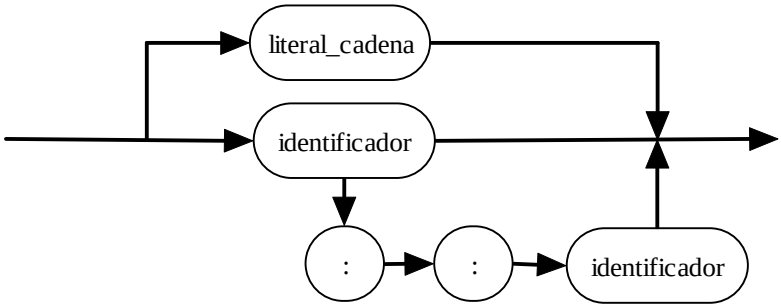


Figura 64: lista_paquetes_de_puertos.

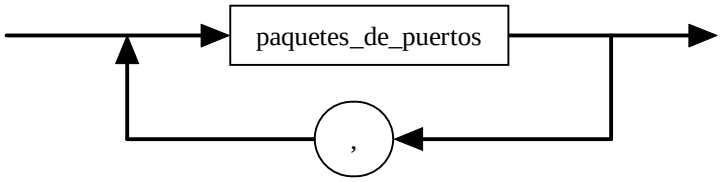


Figura 65: paquetes_de_puertos_pull.

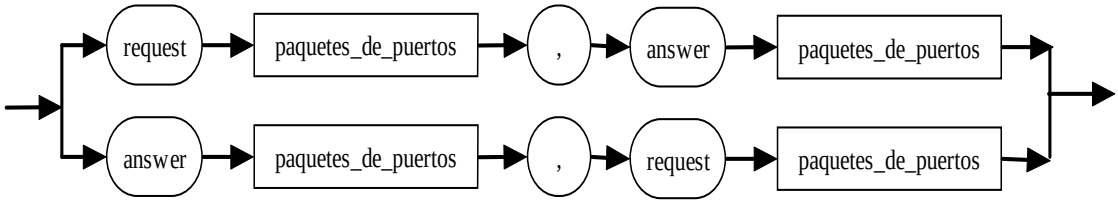


Figura 66: tipo_priority.

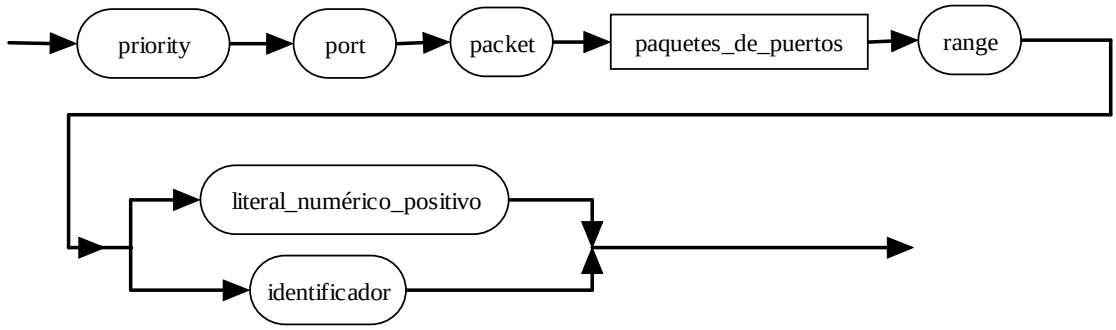


Figura 67: tipo_priorities.

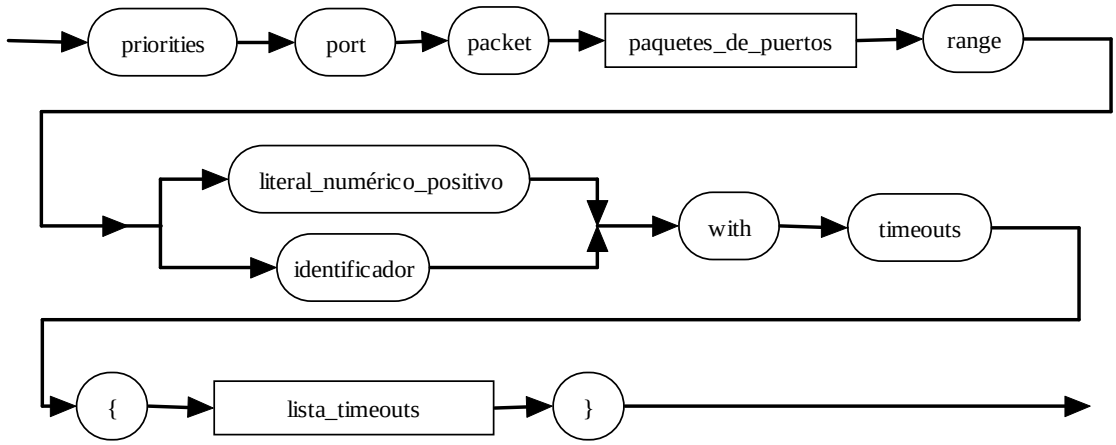


Figura 68: longitud_paquetes_de_puertos.

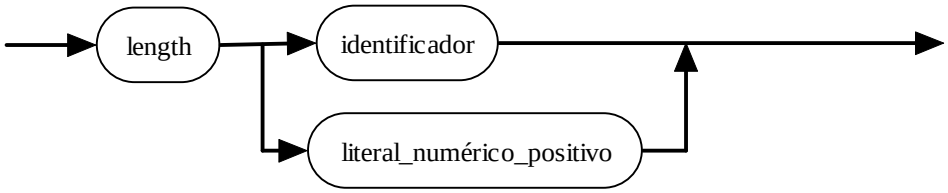


Figura 69: centinela.

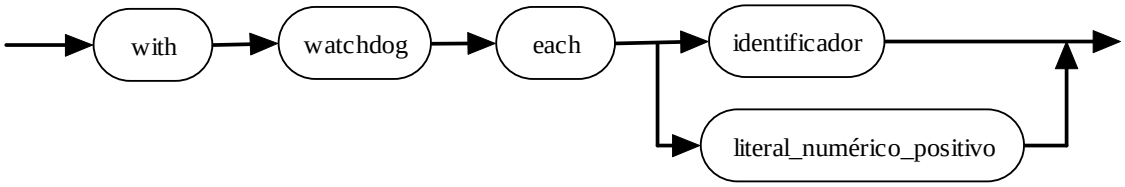


Figura 70: lista_timeouts.

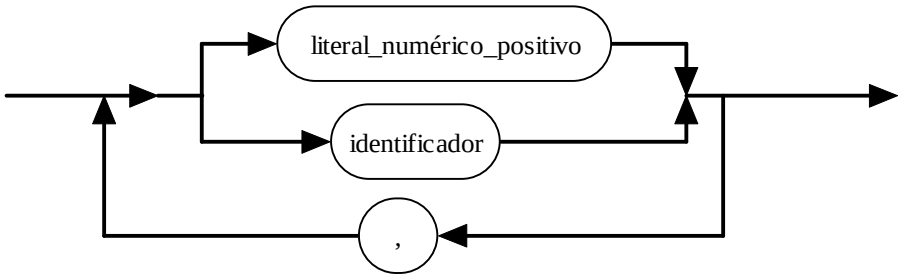


Figura 71: variables.

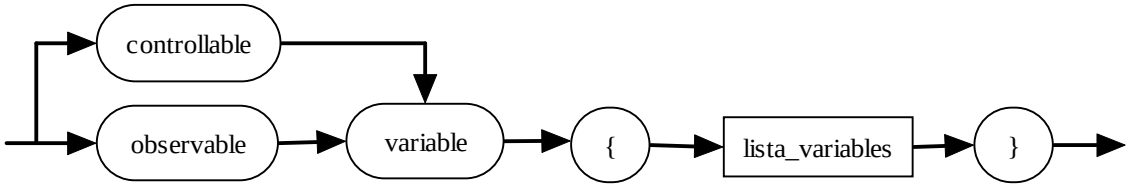


Figura 72: lista_variables.

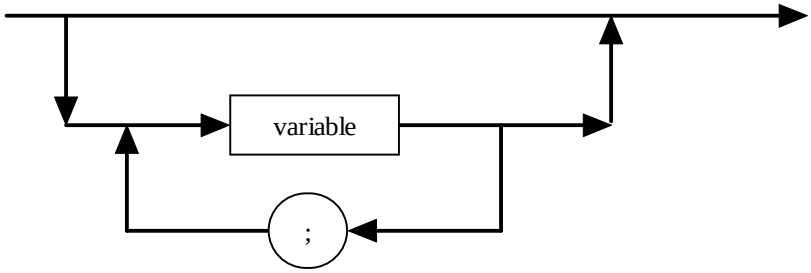


Figura 73: variable.

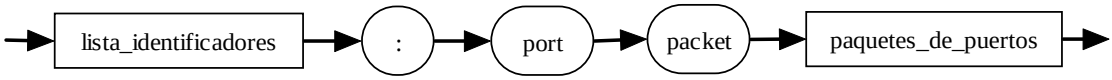


Figura 74: excepciones.

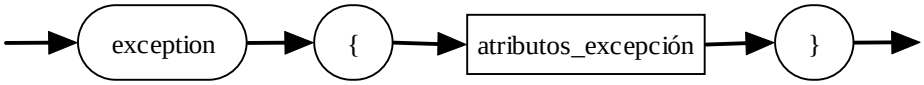


Figura 75: atributos_excepción.

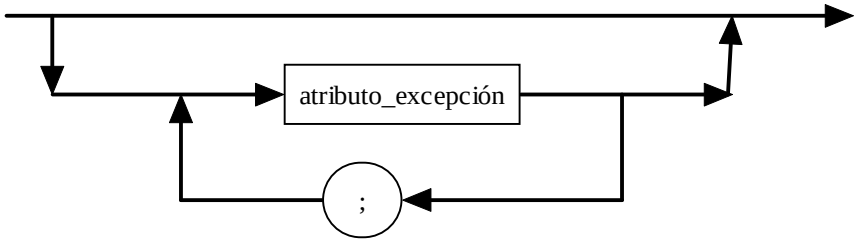


Figura 76: atributo_excepcion.

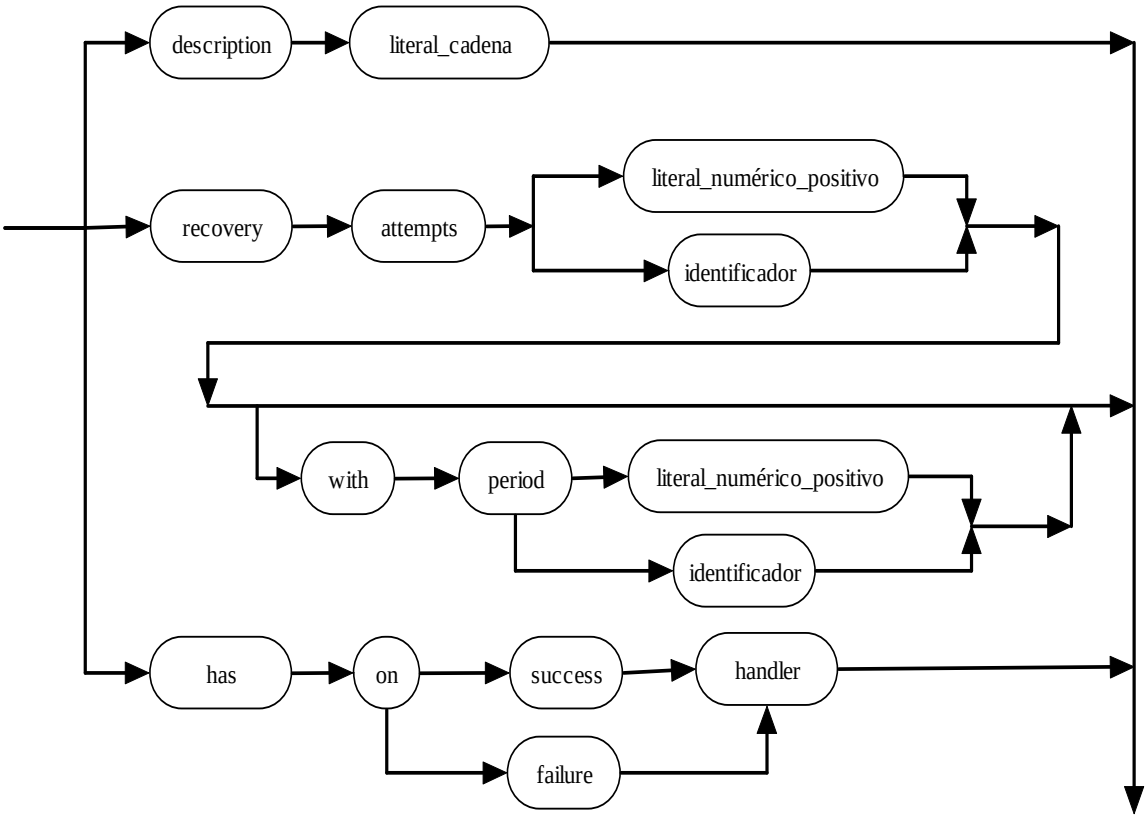


Figura 77: estados.

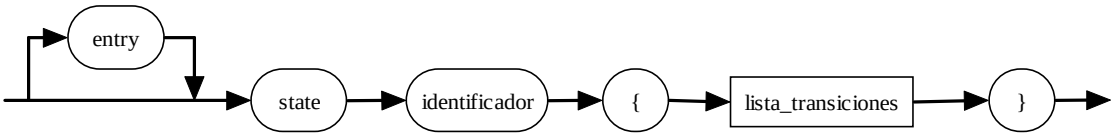


Figura 78: lista_transiciones.

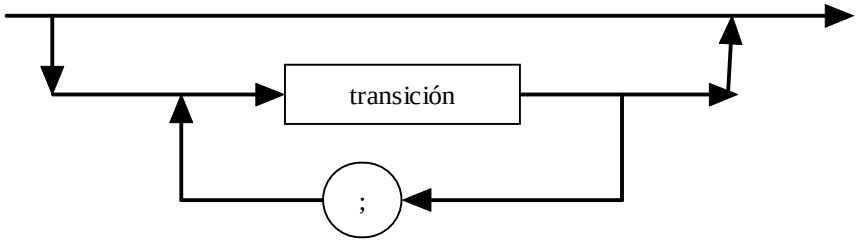


Figura 79: transición.

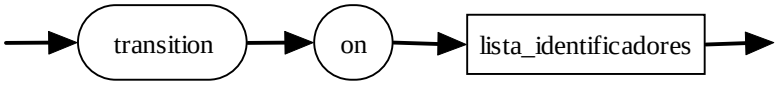


Figura 80: hilos.

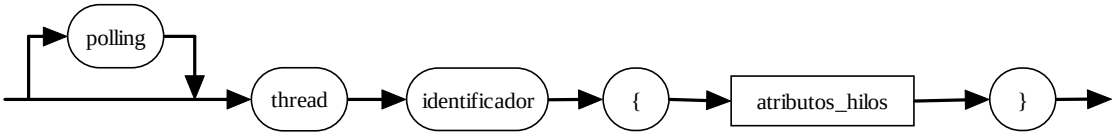


Figura 81: atributos_hilos.

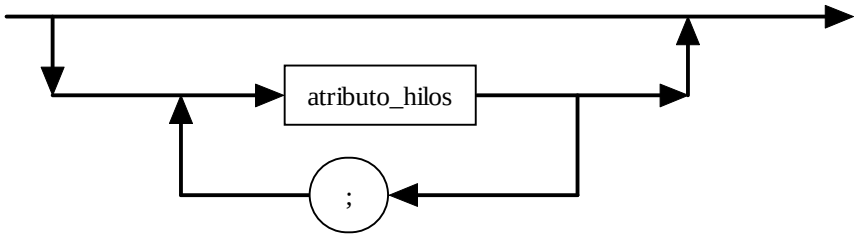


Figura 82: atributo_hilos.

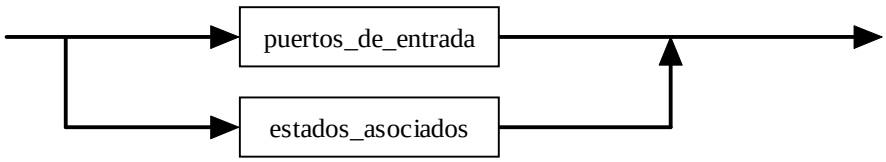


Figura 83: puertos_de_entrada.

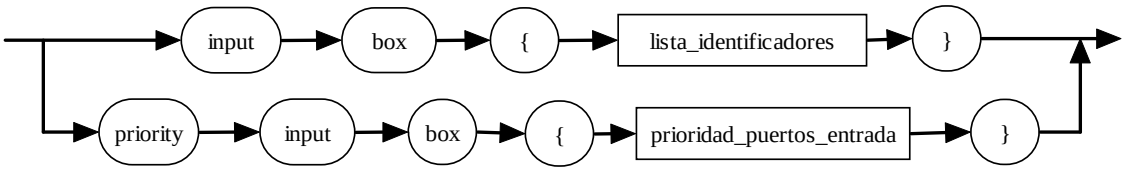


Figura 84: prioridad_puertos_de_entrada.

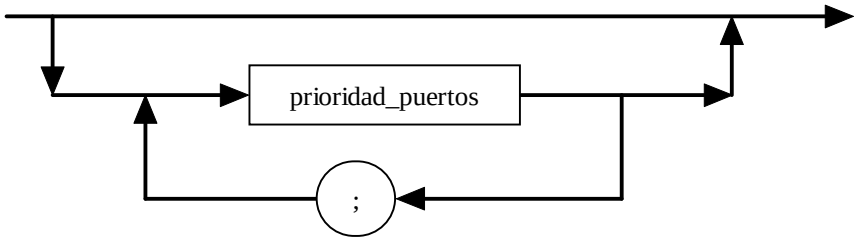


Figura 85: prioridad_puertos.

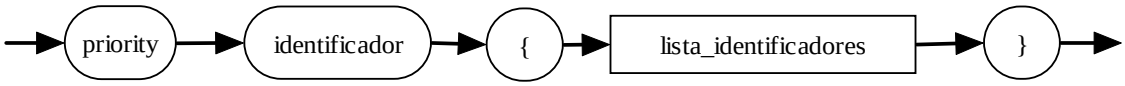
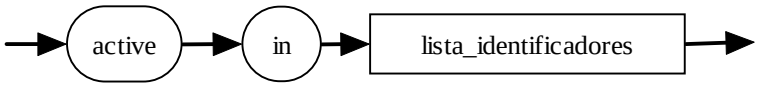


Figura 86: estados_asociados.



Apéndice C

Notación EBNF.

A continuación, se van a mostrar las reglas sintácticas en notación EBNF del lenguaje Description Language.

<inicio> ::= <especificacion_componente> <definicion_componente> ";" .

<especificacion_componente> ::= component identifier .

<definicion_componente> ::= "{" <sentencias> "}" .

<sentencias> ::= <sentencia> {“;” <sentencia> } .

<sentencia> ::= <documentacion_cabecera> | <constantes> | <puertos> | <variables> |
<excepciones> | <estados> | <hilos> .

<documentacion_cabecera> ::= <especificacion_cabecera> <definicion_cabecera> .

<especificacion_cabecera> ::= header .

<definicion_cabecera> ::= "{" <atributos_cabecera> "}" .

<atributos_cabecera> ::= <atributo_cabecera> {“;” <atributo_cabecera> } .

<atributo_cabecera> ::= (author | description | institution | version) stringliteral .

<constantes> ::= constants "{" <lista_de_constantes> "}" .

<lista_de_constantes> ::= <definicion_constante> {“;” <definicion_constante> } .

<definicion_constante> ::= <accesibilidad_constante> <definicion_de_constante> .

<accesibilidad_constante> ::= [(public | private)] .

<definicion_de_constante> ::= identifier "=" <valor_de_constante> .

<valor_de_constante> ::= (number | negativenumber | stringliteral) .

<lista_identificadores> ::= <identificador> {“;” <identificador> } .

<identificador> ::= identifier .

<puertos> ::= <especificacion_puertos> <definicion_puertos> .

<definicion_puertos> ::= <accesibilidad_puerto> <tipo_puerto> .

<accesibilidad_puerto> ::= [(public | private)] .

<tipo_puerto> ::= (input | output) port .

<definicion_puertos> ::= <lista_identificadores> type <tipo_paquete_puerto>
<centinela> .

<tipo_paquete_puerto> ::= (tick | <un_solo_paquete_puertos_sin_longitud> |
<un_solo_paquete_puertos_con_longitud> <longitud> |
<multiple_definicion_paquetes_puertos> <longitud> |
<tipo_pull> <longitud> |
<tipo_priority> |
<tipo_priorities> .

<un_solo_paquete_puertos_sin_longitud> ::= (last | generic) port packet
<paquetes_puertos> .

<un_solo_paquete_puertos_con_longitud> ::= (fifo | ufifo | poster) port packet
<paquetes_puertos> .

<multiple_definicion_paquetes_puertos> ::= (multipacket | lazymultipacket) port
packets "{" <lista_paquetes_puertos> "}" .

<tipo_pull> ::= pull port packets "{" <lista_paquetes_puertos_tipo_pull> "}" .

<tipo_priority> ::= priority port packet <paquetes_puertos> range <rango_timeouts> .

<tipo_priorities> ::= priorities port packet <paquetes_puertos> range <rango_timeouts>
with timeouts "{" <lista_de_timer_priorities> "}" .

<longitud> ::= [length <numero>] .

<numero> ::= (number | identifier) .

<paquetes_puertos> ::= (stringliteral | identifier | identifier ":"":" identifier) .

<lista_paquetes_puertos> ::= <paquetes_puertos> {“;” <paquetes_puertos> } .

<lista_paquetes_puertos_tipo_pull> ::= (<modo_answer> | <modo_request>) .

<modo_answer> ::= answer <paquetes_puertos> "," request <paquetes_puertos> .

<modo_request> ::= request <paquetes_puertos> "," answer <paquetes_puertos> .

<rango_timeouts> ::= <numero> .

<lista_de_timer_priorities> ::= <numero> {“,” <numero> } .

<centinela> ::= [with watchdog each <numero>] .

<variables> ::= <especificacion_variables> <definicion_variables> .

<especificacion_variables> ::= (observable | controllable) variables .

<definicion_variables> ::= "{" <lista_variables> "}" .

<lista_variables> ::= <definicion_variable> {“;” <definicion_variable> } .

<definicion_variable> ::= <lista_identificadores> ":" port packet <paquetes_puertos> .

<excepciones> ::= <especificacion_excepcion> <definicion_excepcion> .

<especificacion_excepcion> ::= exception identificador .

<definicion_excepcion> ::= "{" <atributos_excepcion> "}" .

<atributos_excepcion> ::= <atributos_excepcion> {“;” <atributos_excepcion> } .

<atributos_excepcion> ::= (description stringliteral |
recovery attempts <numero> [with period <numero>] |
has on (success | failure) handler) .

<estados> ::= <especificacion_estado> <definicion_estado> .

<especificacion_estado> ::= [entry] state identificador .

<definicion_estado> ::= "{" <lista_transiciones_estado> "}" .

<lista_transiciones_estado> ::= <transicion_estado> {“;” <transicion_estado> } .

<transicion_estado> ::= transition on <lista_identificadores> .

<hilos> ::= <especificacion_hilo> <definicion_hilo> .

<especificacion_hilo> ::= [polling] thread identificador .

<definicion_hilo> ::= "{" <atributos_hilo> "}"

<atributos_hilo> ::= <atributo_hilo> {“;” <atributo_hilo> } .

<atributo_hilo> ::= ((<puertos_con_prioridad> | <puertos_sin_prioridad>) |
<estados_donde_esta_activo>).

<puertos_con_prioridad> ::= priority input box "{" <puertos_de_entrada_priority> "}" .

<puertos_de_entrada_priority> ::= <puerto_priority> {“;” <puerto_priority> } .

<puerto_priority> ::= priority identificador "{" <lista_identificadores> "}" .

<puertos_sin_prioridad> ::= input box "{" <lista_identificadores> "}" .

<estados_donde_esta_activo> ::= active in <lista_identificadores> .

Apéndice D

Ejemplo de un componente completo.

Aquí va el ejemplo complete de un componente CoolBOT.

Referencias y Bibliografía.

[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.

[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.

[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.

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

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

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

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

[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.).

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

[cygwin] Página oficial del proyecto Cygwin http://www.cygwin.com/

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

[flex] Página oficial del proyecto Flex. http://www.gnu.org/software/flex/manual/

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

[c++] Curso de C++. http://www.zator.com/Cpp/

[c++ Ref] Manual de referencia de C++. http://www.cppreference.com/index.html