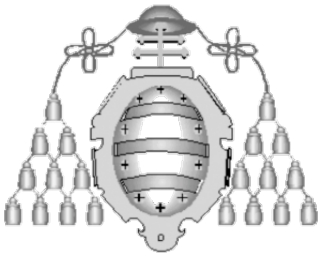

Universidad de Oviedo

Departamento de Ingeniería Eléctrica, Electrónica, de Computadores y de Sistemas

MIOOP. Modificación del estándar IEC-61131 para dar soporte al paradigma de programación orientado a objetos. Aplicación al desarrollo del control de procesos industriales.

Memoria persentada para la obtención del grado de Doctor en Ingeniería Informática

2012

Autor: Ángel Luis Sierra Díaz

Directores: Dr. Víctor Manuel González Suárez

Dr. Felipe Mateos Martín

Aí pues, ¿cuál es la virtud más importante
que debe poseer un samurai?
Varios samurais se adelantaron a contestar.
Fuerza...valor...iniciativa...lealtad.
Muy bien, dejadme ver si las acciones de vuestros
yojimbos viven según vuestras palabras.
Ujiro acepto el reto y se adelantó.
- Shushen - ¿Mi señor? - ¡Mátame!
Su katana estaba fuera de su saya y en Ujiro
antes de que nadie pudiera moverse.
Luego, sin pausa, dejó caer su katana,
desenvainó su wakizashi y cayó sobre él.
Pequeño fragmento de la novela *Invierno* de Kaita Ryoku

A mis padres y esposa.

Agradecimientos

Es mucho más correcto cuando un amigo debe enfrentarse al infortunio, estar cerca de él, visitarlo y socorrerlo. Un samurai no debe jamás, mientras viva, permitirse distanciarse de aquellos de los que es deudor espiritualmente.
Proverbio samurai

Los cimientos sobre los que se sostiene este trabajo se pierden ya en mis recuerdos. Ha sido tanta la gente a lo largo de los años que ha influido en mayor o menor medida en mí, que necesitaría decenas de páginas para poder expresar mi agradecimiento y aún así, me quedaría corto. Desde las personas o profesores que me acercaron hacia la informática y a los ingenios automáticos, hasta quienes me inculcaron la semilla de la curiosidad y el ir un poquito más lejos. Gracias a todos, y aunque no pueda decir lo agradecido que estoy a cada uno, pido disculpas a todos aquellos a los que olvido agradecer e intentaré, en las próximas líneas, recordar a las personas que más directa o indirectamente han influido en la consecución de esta memoria de tesis.

Debo comenzar los agradecimientos de forma más individual con los profesores y personal de la Universidadd de León. Les quiero agradecer que desde que llegué allí me trataran más como un amigo que como un simple alumno. El primer día que aparecí por esa Universidad, si no es por la ayuda del personal de secretaría no se que habría sido de mí, ¡os debo la vida! A los profesores Pigüi y José M.

Alija, que conseguían influenciar de forma positiva a sus alumnos con un trato gratamente cercano. Y sobre todo, a Ramón Ángel Fernández, un crack en toda la magnitud de la palabra. Profesor de 10 y persona de 11. Sois todos geniales, sín vosotros, no se que sería de la escuela de Ingeniería de León.

Al Departamento de Ingeniería Eléctrica, Electrónica, de Computación y Sistemas de la Universida de Oviedo les quiero agradecer la paciencia y atenciones que han tenido conmigo. Desde Gemma que me ha resuelto una y otra vez las papeletas burocráticas que no he sabido solucionar yo solo, pasando por Ricardo Mayo y sus charlas sobre electricidad y automatizaciones. A Felipe Mateos y su predisposición para ayudar a Victor o a mi. O a Pablo García, sin cuya ayuda desinteresada jamas hubiese llegado a hacer ninguna publicación. Todos vosotros y muchos más, sois los artífices de que la gente que no trabaja directamente con la Universidad, se sienta en el departamento como en su propia casa.

A los profesores de la Universidad de Oviedo que he tenido tanto en la carrera como a lo largo del desarrollo de mi tesis doctoral. En particular a Jose M. Labra, Belen Martinez, Francisco Ortin, Covadonga Nieto y otros tantos profesores que han influido en dotarme de los conocimientos necesarios para poder desarrollar no sólo este trabajo sino mi carrera profesional. Tampoco puedo olvidar a Ines Couso y agradecer los ratos que me ha dedicado para explicarme la mejor forma de dar un sentido matemático a los resultados experimentales que se han desarrollado en esta memoria. Si la Universidad de Oviedo es grande, es gracias a sus profesores y personal.

Que duro hubiese sido mi trabajo sin la incomiable labor de tantos becarios que pasaron antes que yo por el departamento. Sín su trabajo, jamas habría llegado si quiera a estar escribiendo estas líneas a día de hoy. Gracias en especial a todos esos proyectantes y becarios con los que sí he tenido el gusto de compartir tantas horas de trabajo como Alejandro Junquera, David Garcia, Hugo González y José Esperón. ¡Qué ingenieros ha ganado la sociedad!

Dicen que la grandeza de una persona se mide por sus logros. Yo prefiero pensar que uno es lo que es por la gente que tiene cerca. Mis queridos amigos Chip, Aarón, Muñoz, Popi, Berto y Joni,, que nunca han parado de apoyarme. A mis antiguos compañeros de estudios Tuto, Guti, Nando y Boludo, que siempre me han incitado a conseguir lo máximo. A mi cuñada *"hostil"* Pili, por su infatigable aliento. A mi suegra Mari, por su fuerza vital y a mi querido suegro Antonio al

que todos echamos de menos. A mi tía Nani, que siempre ha actuado como una segunda madre y espero hacer lo propio con mi ojito derecho Nono, mi ahijado. A mis dos abuelas, que siempre presumen de nieto y no se porqué... amor de abuelas; y a mis dos abuelos, que seguro estarán orgullosos de mí allá donde estén. A mis padres, que nunca han dejado de estar ahí cuando los he necesitado, me han proporcionado la educación que tengo y en cuyos ojos veo el orgullo de los pequeños pasitos que voy dando hacia adelante; algún día espero ser la mitad de buen padre que vosotros.

En especial quiero darle mi mas profunda gratitud a mi mejor amiga Almudena, que además tengo el gusto de llamar esposa. Le debo infinitas disculpas por las horas robadas, los malos humores, las dudas y los malos momentos que te he hecho pasar mientras desarrollaba este trabajo. Tu apoyo, tu corazón y tu forma de ser me hacen mejor persona y sin lugar a dudas, el hogar que hemos formado lo considero mi mayor logro. Este trabajo es una pequeña muestra del agradecimiento que te tengo desde el día que te conocí.

Y por último, no puedo olvidarme de una persona que ha sido capital en la consecución de este trabajo, Víctor M. González. Gracias por ayudarme a realizar esta tesis. Gracias por ser mi director de esta memoria. Gracias por dedicarme tantas horas. Gracias por tus sabios consejos. Gracias por buscar financiaciones y proyectos hasta de debajo de las piedras. Gracias y disculpas a tu esposa María por robarle tantas veces a su marido. Pero sobre todo, gracias por ser mi amigo.

Sinceramente, no tengo palabras para expresar lo agradecido que estoy a todos. Espero haber estado a la altura de vuestras expectativas. ¡Va por vosotros!

Ángel Luis Sierra Díaz

Resumen

No digas: es imposible.
Di: No lo he hecho todavía.
Proverbio samurai

La orientación a objetos facilita el desarrollo de software a gran escala y de calidad. A pesar del extendido uso de este paradigma en la Ingeniería de Software, su uso en la Ingeniería de Automatización de Procesos se encuentra con numerosas dificultades debido, entre otros motivos, al carácter fuertemente conservador del ámbito industrial y a la falta de estandirazación, tanto a nivel de hardware como de sofware, de los autómatas programables (PLCs).

Sólo la presión de poderosos clientes de productos para la automatización de procesos, como los del sector de la automoción, propició la normalización de diversos aspectos de este tipo de productos, cristalizando en 1993 con la creación del estándar IEC 61131. Aunque este estandar produjo un salto hacía adelante y en la actualidad, la mayor parte de los PLCs del mercado lo soportan, aún se siguen utilizando prácticas de programación basadas en el paradigma estructurado.

En esta memoria de tesis se presenta MIOOP, un conjunto de moficicaciones y ampliaciones a la norma IEC 61131 y en particular a su apartado 3 donde se recogen los lenguajes de programación, que permitan la aplicación del paradigma orientado a objetos en la automatización de procesos. Por otro lado, y debido al fuerte conservadurismo del sector industrial, este trabajo no pretende romper

con todo lo anterior. Por este motivo, MIOOP proporciona un conjunto de reglas necesarias para realizar la traducción de un código orientado a objetos a otro no orientado a objetos basado en la estándar IEC 61131-3.

Esta tesis se desarrolla sobre la base de una serie de nuevos conceptos definidos a lo largo del presente trabajo como son, el conjunto de ampliaciones a la norma para dar soporte a la orientación a objetos, las reglas de traducción de un código orientado a objetos a otro estructurado y la arquitectura de la herramienta software de apoyo a la aplicación sistemática de MIOOP denominada SimPLC++.

La hipótesis fundamental que se pretende demostrar con este trabajo establece que es posible programar la lógica de control de un proceso de eventos discretos secuencial bajo un paradigma orientado a objetos. Un segundo objetivo consiste en medir las implicaciones del uso de un paradigma orientado a objetos frente a uno estructurado en base a medidas cuantitativas y cualitativas.

Abstract

Object-oriented paradigms facilitate good quality software development on a large scale. Although this type of paradigms has been largely used in Software Engineering, it encounters countless difficulties in Processl Logic Control Engineering, mainly due to the strongly conservative character of the industrial field, and the lack of standardization in hardware and software programmable logic controllers (PLCs).

Only the pressure made by powerful customers gave rise to the creation of the international standard IEC 61131 in 1993. Although IEC 61131 was a breakthrough, and most current PLCs support it, structured paradigm programming styles are still used.

MIOOP is presented in this doctoral thesis as a set of modifications and extensions of the IEC 61131. In section 3 you can find the programming languages that allow for the use of object-oriented paradigms in process logic control. On the other hand, and due to the strong conservatism observed in the industrial field, this thesis does not intend to break with tradition. For this reason, MIOOP provides a set of rules to translate object-oriented codes to other non oriented ones based on the IEC 61131.

This thesis develops a set of new concepts, such as the number of extensions of the standard to support object orientation, the translation rules from an object oriented code to a structured one, and the software tool that supports the systematic use of MIOOP, called SimPLC++.

The main hypothesis that is to be proved in this thesis is that it is possible to programme an object oriented model of a control logic of a discreet event sequential

process. The second aim of this thesis is to measure the implications of using an object-oriented paradigm against a structured one, based on quantity and quality measures.

Índice general

Índice de figuras

Índice de algoritmos

Índice de cuadros

Capítulo 1

Motivación, objetivos y estructura de la tesis

Al disitnguir las ventajas de las armas de los guerreros descubrimos que, cualquiera que sea el arma, existe un momento y una situación en la que ésta es apropiada.
Miyamoto Musashi (Anillo de la tierra)

1.1. Introducción

Según la RAE (Real Academia de la Lengua Española), en su primera afección, se define automatismo como : *Desarrollo de un proceso o funcionamiento de un mecanismo por sí solo.*

Según el diccionario de Cambridge, un autómata se define como: *Una máquina que funciona sola sin la necesidad del control humano.*

Estas definiciones aún siendo correctas, no son demasiado descriptivas. Desde el punto de vista de un ingeniero, se define un sistema (máquina o proceso) automatizado como aquel capaz de reaccionar de forma automática (sin la intervención del operario) ante los cambios que se producen en el mismo, dando lugar a las acciones adecuadas para cumplir la función para la que ha sido diseñado. La figura 1.1 muestra la estructura típica de un sistema automatizado.

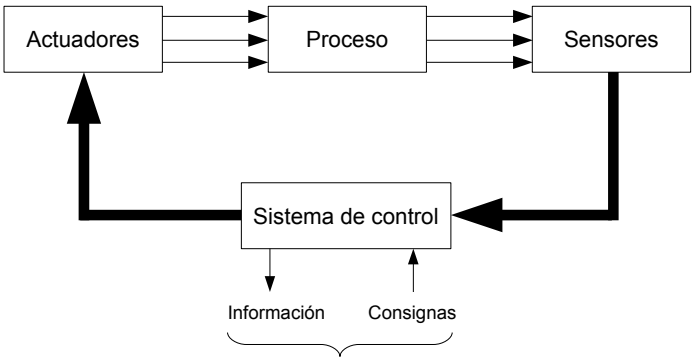

Figure 1.1: Estructura de un sistema automatizado

Como se observa, se trata de un sistema en bucle cerrado, donde la información sobre los cambios del proceso captada por los sensores, es procesada dando lugar a las acciones necesarias que se implementan físicamente sobre el proceso por medio de los actuadores. Este sistema de control se comunica eventualmente con el operador, recibiendo de éste consignas de funcionamiento (marcha, paro, cambio de características de producción, etc), y comunicándole información sobre el estado del proceso (para la supervisión del correcto funcionamiento).

Se denomina automatismo al sistema completo, aunque con este término suele hacerse referencia fundamentalmente al dispositivo físico (ya sea eléctrico, neumático, electrónico, etc.) ya que es éste el que produce de forma automática las acciones sobre el proceso a partir de la información captada por los sensores y las decisiones tomadas por el sistema de control. Las señales de entrada y de salida pueden ser de cualquier tipo, sin embargo, este trabajo se enfoca en el estudio de sistemas de eventos discretos, que son aquellos sistemas que operan sobre señales que suceden en instantes de tiempo determinados [Thi01]. En teoría de sistemas de eventos discretos, se define el *"estado"* del sistema en un instante determinado $t >= t^0$ como el conjunto de información necesaria para poder determinar la salida

del sistema partiendo de esa información y de la entrada al sistema en ese instante. El estado inicial se designa por t^o. El conjunto de estados en los que puede llegar a estar un sistema se define como espacio de estados. La evolución del sistema de un estado a otro vendrá determinada por la ocurrencia de un evento. En ese caso, el sistema de control implementa las funciones lógicas que relacionan las entradas y las salidas.

Un sistema de control se puede implementar mediante tecnología mecánica, neumática, hidráulica, eléctrica, electrónica o por computador. La idea de implementar un dispositivo programable por medio de un computador, pasa por codificar las instrucciones necesarias que permitan al sistemas tomar las decisiones oportunas a partir de las señales recibidas por los sensores. Dado que el desarrollo de la lógica de control de un sistema programable consiste básicamente en la confección de un programa con unas características especiales, se puede aprender mucho de las experiencias de desarrollo de los programas de software para ordenadores. En esta disciplina se han desarrollado, con el paso del tiempo, gran cantidad de lenguajes de programación y diversos paradigmas de programación. Uno de los paradigmas de programación en informática que significó un antes y un después fue la orientación a objetos en la que se centra este trabajo.

En los siguientes apartados se hace un recorrido histórico por la evolución de las disciplinas de la programación de ordenadores y de la automatización de procesos, para tratar de hallar las causas que justifiquen el desfase en el desarrollo de ambas tecnologías y tratar de definir de esta forma, los objetivos e hipótesis que se pretenden demostrar con la presente memoria de tesis.

1.2. Motivación

El desarrollo de programas o la programación en general, es una disciplina cuyo auge en las últimas décadas sólo encuentra parangón en la historia con el sufrido por las disciplinas de Ingeniería Mecánica, a raíz de la revolución industrial de comienzos del siglo XIX, y de Ingeniería Eléctrica con los inventos de Edison a mediados de la misma centuria. La revolución industrial, que cambió en un principio los esquemas productivos, dio lugar con el paso del tiempo a nuevas formas de organización social, como por ejemplo: cambios de horarios cotidianos para ajustarse a los ritmos productivos, desarrollo de grandes ciudades en torno a los

polos de industrialización, cambios en la estructuración en clases sociales, disminución del poder de los terratenientes y comerciantes, desarrollo de la sociedad de consumo, acceso a la educación de mayores capas sociales, un mayor nivel de democratización, etc. En definitiva, la revolución industrial provocó una transformación general de la vida tal y como se entendía hasta ese momento, dando lugar con el paso del tiempo al nuevo concepto de *"Sociedad Industrial"*.

En la actualidad, se esta asistiendo a una revolución social similar que está dando lugar a la llamada *"Sociedad de la Información"*. Esta revolución se gesta con la aparición de los primeros ordenadores a mediados del siglo XX, hecho comparable a la aparición de la máquina de vapor diseñada por Watt, la cual supuso el empujón definitivo al desarrollo de la sociedad industrial. Este paralelismo entre ambas evoluciones históricas (ver figura 1.2) se aprecia también a nivel técnico.

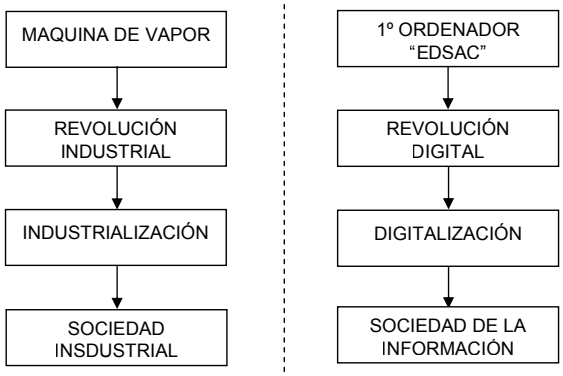

Figura 1.2: Paralelismo evolutivo entre las Sociedades Industrial y de la Información

Del mismo modo que la revolución industrial propició el desarrollo de las disciplinas mecánica y eléctrica, la revolución digital ha propiciado el desarrollo de las disciplinas de la electrónica y la programación.

La programación es la disciplina que se encarga del desarrollo de programas. Un programa está formado por una secuencia de instrucciones para comandar la ejecución de actividades en un orden determinado, por parte de un dispositivo electrónico programable, para cumplir con una tarea específica.

Por tanto, la disciplina de programación, como su propia definición indica, engloba el desarrollo de cualquier tipo de programa susceptible de ser ejecutado por un

dispositivo programable. Entre este tipo de dispositivos se pueden citar: ordenadores, microcontroladores, PDAs,... y en definitiva, cualquier dispositivo electrónico dotado de un microprocesador capaz de ejecutar las instrucciones que forman el programa almacenadas en una memoria y sobre unos datos también almacenados en la memoria.

Cuando la disciplina de programación es empleada en el entorno industrial, el objetivo que se desea que lleve a cabo el microprocesador consiste básicamente en controlar el modo de funcionamiento de un proceso productivo o máquina.

A partir de la aparición de una nueva disciplina de programación en el seno de la automatización industrial dedicada al control de procesos mediante la programación de dispositivos tipo PLCs, se produce una evolución paralela de las disciplinas de codificación de programas pero a distintas velocidades en el ámbito informática y de la automatización. Mientras que la programación de ordenadores sufre una tremenda transformación en un relativo corto periodo de tiempo, la programación industrial parece quedarse estancada en los primitivos esquemas y modelos desarrollados en la década de 1970.

A continuación, se presentan las evoluciones paralelas que han llevado los paradigmas de programación en ambas disciplinas y se hace una comparación entre las dos.

1.2.1. Evolución histórica de la automatización industrial

La formalización del tratamiento de los automatismos es muy reciente. Históricamente se puede decir que el tratamiento de los automatismos lógicos se ha basado en el álgebra de Boole [Boo47] y en la teoría de autómatas finitos. No fue hasta la década de los sesenta cuando se dispuso de herramientas como las redes de Petri (RdP)[Pet62] para el diseño y análisis de automatismos secuenciales y concurrentes.

Las primeras tecnologías disponibles para implementar controladores de sistemas de eventos discretos se basaban en la aplicación de tecnologías cableadas, lo que se denominaba automatismos cableados. Se utilizaban principalmente las tecnologías neumática y electromecánica, y tuvieron gran importancia desde comienzos del siglo XX hasta los años 50. El problema de este tipo de sistemas era su falta

de flexibilidad, lo que dificultaba la escalabilidad de estos sistemas, asi como lo complicado que resultaba encontrar errores o realizar modificaciones.

En 1950, Brown y Campbell publican el primer trabajo sobre la aplicación del computador en el control Industrial [GB50]. En dicho artículo aparece un computador controlando un sistema mediante bucle de realimentación y prealimentación. Los autores asumen que los elementos de cálculo y control del sistema deben ser computadores de cálculo analógicos, pero sugieren el posible uso de un computador digital. A partir de este momento comienzan a utilizarse los semiconductores (electrónica) en la automatización, lo que provoca una reducción del tamaño de los armarios eléctricos y el número de averías por desgaste de componentes. Pero estos sistemas siguen presentando una enorme falta de flexibilidad, es decir, un sistema de control sólo sirve para una aplicación específica y no es reutilizable.

A finales de los años sesenta, los fabricantes de automóviles necesitaban nuevas y mejores herramientas de control de la producción. Los *"nuevos controladores"* debían ser fácilmente programables por ingenieros de planta o personal de mantenimiento. El tiempo de vida del sistema debía ser largo y los cambios en el programa tenían que realizarse de forma sencilla. Además, se imponía que trabajaran sin problemas en entornos industriales adversos. La solución fue el empleo de una técnica de programación familiar y reemplazar los relés mecánicos por relés de estado sólido.

Los autómatas programables (PLC) se introducen por primera vez en la industria en 1968 aproximadamente. Bedford Associates propuso un sistema de control denominado *"Controlador Digital Modular"* (Modicon) al fabricante de automóviles General Motors.

A mediados de los 70, la potencia de los PLCs aumenta notablemente con la aparición del microprocesador, lo que permite que se puedan reprogramar sin necesidad de recablear el autómata y hacer cálculos matemáticos complejos. A partir de este momento, por cada modelo de microprocesador surge un modelo de PLC basado en el mismo y un lenguaje de programación asociado a dicho PLC. Estos primeros lenguajes de programación seguían una notación parecida a los esquemas de relés familiar para los electricistas de esa época, ya que estaba basado en diagramas de escalera y símbolos eléctricos comunes para ellos (ver figura 1.3).

Durante los años 70 y 80 se produjo su difusión en la industria. Inicialmente rechazado, el PLC fue ganando aceptación a medida que demostraba sus virtudes,

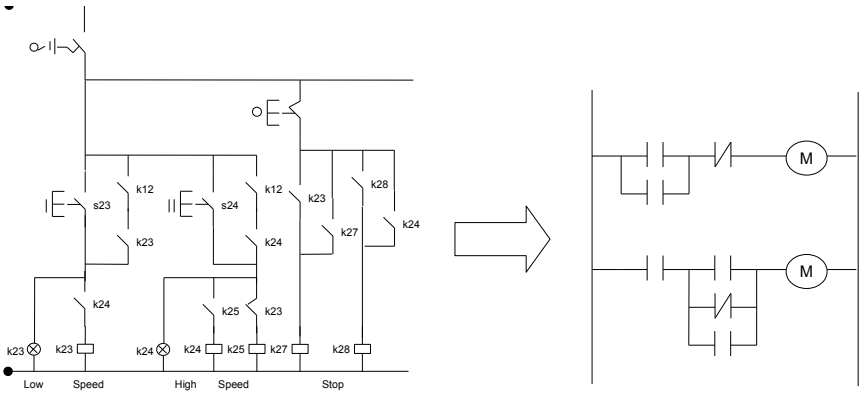

Figura 1.3: Paso de notación de reles a diagrama de escaleras

primero en un sector tan exigente y prescriptor de tecnologías innovadoras como es el de la automoción para posteriormente, y a remolque de dicho éxito, popularizarse en muchos otros ámbitos de la industria.

A finales de 1980, comienza a abordarse una problemática cuya evolución todavía continúa en nuestros días. En esa época, el nivel de utilización del PLC en el control de procesos industriales crecía de forma continuada. Sin embargo, su prácticamente nula capacidad de comunicación entre ellos y/o con otros dispositivos (que no fueran la unidad de programación) impedía la integración horizontal y vertical de los procesos de producción y fomentaba la existencia de islas de automatización en las plantas. Ésto en sí mismo no sería un problema (al fin y al cabo ocurría algo similar en la época del control basado en relés) si no fuera porque, en el ámbito de la informática de propósito general, el éxito de las redes de comunicación (con Ethernet y TCP/IP como grandes protagonistas de facto) promovió un gran interés a nivel industrial. Es por ello que, apenas 6 o 7 años después de la invención de Ethernet (allá por 1973), que los fabricantes de PLCs crearon las primeras redes de comunicaciones específicamente diseñadas para la intercomunicación de equipos de control industrial: surgen así soluciones como Modbus (de Modicon, hoy Schneider Electric), Data Highway (Allen-Bradley, hoy Rockwell Automation), Interbus (Phoenix Contact), Profibus (Siemens), etc. En base a ellas, se enuncian las primeras visiones de una fábrica intercomunicada a todos los niveles (desde el proceso hasta la gerencia) y el concepto de la pirámide CIM (acrónimo inglés de fabricación integrada por computador) que perdura hasta la actualidad.

Sin embargo, esta visión idealizada de una fábrica perfectamente intercomunicada a todos los niveles debía enfrentarse a nuevos retos para alcanzar el éxito. Uno de ellos es el de la estandarización. En 1990, el número de soluciones disponibles en el mercado para abordar la automatización de procesos y las comunicaciones industriales entre ellas era muy elevado, con prácticamente una solución propugnada por cada uno de los fabricantes que compiten en este sector tecnológico. Y lo que es peor, las arquitecturas de automatización que postulaban solian ser cerradas y propietarias, lo que impedía, o cuando menos dificultaba, la interconexión entre equipos de diferentes fabricantes y propiciaba la creación de mercados cautivos. Como cabría esperar, los clientes reaccionaron ante esta situación y comenzaron a dar prioridad en sus procesos de selección a aquellas soluciones de automatización y comunicaciones abiertas y, en la medida de lo posible, que cumplieran estándares internacionales. Surgieron en esta década normativas como la bien conocida IEC 61131-3 [IEC2a], que estandarizaba los lenguajes de programación para dispositivos de control de procesos y por cuyo cumplimiento vela la asociación PLCopen.

El uso de la norma IEC 61131 reduce el esfuerzo humano en entrenamiento, depuración, mantenimiento y consultoría de los programas. Una vez que se aprende, conoce y domina, se puede utilzar en todos los sistemas, dando la posibilidad de crear software reutilizable, minimizando el tiempo de desarrollo, el esfuerzo de codificación, y los errores de compilación y ejecución. También permite el uso de técnicas de programación usados en otros entornos no industriales, como el informático. Además, permite una coordinación eficiente entre diferentes componentes, desde distintas localizaciones, compañías o proyectos, y aumenta la conectividad facilitando la distribución del control.

En los últimos años, una de las palabras más pronunciadas por todos los responsables de marketing de los fabricantes de equipos de automatización es la de *"integración"*. Tras la estandarización o apertura en aspectos tan importantes como las comunicaciones industriales o los lenguajes de programación, los fabricantes debían diferenciarse del resto aportando nuevas funcionalidades en su familia de productos. Con la lección bien aprendida en relación a las ventajas que aporta la estandarización, se empezó a ofrecer la plena integración funcional de los equipos disponibles en su catálogo. Se pasaba así de una fase orientada a la fabricación de componentes tecnológicos, a otra orientada a la creación de herramientas (en muchos casos, software) que facilitasen al máximo la realización de sistemas y procesos más complejos basados en dichos componentes (hardware).

Entre los objetivos perseguidos por la integración se encontraba la superación de barreras para la implantación de sistemas complejos relacionadas con:

- La comunicación de los sistemas de control de procesos entre sí y con los computadores de gestión de la planta.
- La comunicación con los usuarios del proceso a través de los interfaces hombre-máquina.
- La aplicación de arquitecturas de automatización centralizadas y/o distribuidas.
- La flexibilización de los sistemas de control de cara a la automatización de sistemas híbridos (mezcla de procesos continuos y discretos).

1.2.2. Evolución de los lenguajes de programación en informática

La programación es una actividad humana que se ha desarrollado casi enteramente durante la segunda mitad del siglo XX y ha ido de la mano con los distintos avances que se han producido en la electrónica.

El desarrollo de lenguajes de programación ha evolucionado con el trabajo de diferentes grupos de investigación. No es hasta la aparición de los ordenadores y áreas como la lingüística matemática, que se han especializado las técnicas para el desarrollo de lenguajes de programación y que permiten entender mejor el lenguaje natural. En la década de 1940 (aunque esta fecha puede diferir entre las diferentes versiones que existen acerca de la historia de la computación), fue eminente la creación de lenguajes que hicieran posible la comunicación entre seres humanos y ordenadores. Durante los tiempos de guerra que prevalecieron entonces y en donde los cálculos eran fundamentales, surgen los lenguajes numéricos. Entre éstos se encuentra el lenguaje A-0 desarrollado en la *"Univac"* por un grupo cuyo líder era Grace Hopper y, por otro lado, John Backus [Bac54] desarrolla Speedcoding para la IBM701. Estos lenguajes hacían una traducción de expresiones aritméticas a código de máquina, pero lo más importante es que se empezaron a desarrollar las notaciones simbólicas. Los primeros lenguajes en aparecer son los llamados ensambladores, con los que se dio un gran avance al tener la oportunidad de programar

los ordenadores a través de mnemónicos o claves (y no *"cableando"* los equipos, ya que los primeros ordenadores que funcionaron exitosamente fueron ordenadores analógicos, los cuales efectuaban cálculos matemáticos a través de complejos circuitos electrónicos analógicos) para obtener la configuración requerida por medio de códigos numéricos, código de máquina. El empleo de mnemónicos para la elaboración de programas fue de gran utilidad, aunque la traducción a código máquina se seguía haciendo de manera manual. No pasó mucho tiempo hasta que se dio el paso decisivo para que fueran las mismas máquinas las encargadas de hacer esta traducción, lo que fue conocido como *"ensamblar el programa"* y que dio nombre a este tipo de lenguajes.

La serie de mnemónicos que utilizan los lenguajes ensambladores tienen una correspondencia prácticamente uno a uno con el lenguaje de máquina, lo que permite una programación detallada.

Pese a que los lenguajes ensambladores representaban un gran avance, los programadores aún debían ser *"verdaderos expertos"* en programación de bajo nivel. Con el paso del tiempo se encontró que los programadores recurrían a ciertos patrones de instrucciones que formaban bloques y tenían que ejecutarse de manera periódica al llevar a cabo una acción en los programas desarrollados, lo que originó la búsqueda de lenguajes que englobaran dichas instrucciones y se presentaran con un lenguaje más amigable para el usuario, más cercano al lenguaje natural.

En 1950, John Bakus dirigió una investigación que culminó en el desarrollo del conocido lenguaje Fortran. Este lenguaje ha sido considerado el primer lenguaje de alto nivel y fue utilizado en sus inicios por ordenadores IBM. Con su aparición surge el término de *"compilador"* como un sistema encargado de hacer la traducción entre un lenguaje de alto nivel y lenguaje de máquina.

En Europa, otra corriente basaba sus investigaciones en la creación de la definición de un lenguaje independiente de la máquina, implementando la teoría de gramáticas de Noam Chomsky [Cho9a, Cho9b], en particular, las gramáticas libres de contexto. Con estas ideas, un grupo de la Universidad de Münich definió el *"International Algebraic Language"*, publicado en Zurich en 1958. La implementación de este lenguaje dio como resultado el lenguaje Algol58 (Algoritmic Language). Con el lenguaje Algol evolucionaron y se crearon los nuevos conceptos de los lenguajes como: formato libre, declaración explícita de identificadores, estructuras iterativas más generales, recursividad, paso de parámetros por valor y por referencia, y

estructura de bloques. Tiempo después aparecería Algol60, que sería un lenguaje considerado como expresivo para describir algoritmos. Fue un lenguaje utilizado sobre todo en Europa y no tuvo mucho impacto comercial, sin embargo, los conceptos que se definieron en él se volvieron sumamente importantes para el futuro del desarrollo de software.

Al final de la década de 1950 se desarrollo Lisp en el MIT (Massachusetts Institute of Technology) por John McCarthy [McC60], siendo desarrollado sobre la base de estructuras generales de listas. Éste ha sido un lenguaje utilizado en inteligencia artificial y han existido múltiples variantes de él, entre las que se encuentran: MacLisp, FrazLisp, CommonLisp, Scheme.

En la decada de 1960, John Kemeny y Thomas Kurtz crean en el colegio Darmaourth el lenguaje Basic (Beginners All Purpose Symbolic Instruction Code) con la idea de desarrollar un lenguaje de programación simple para el desarrollo de aplicaciones en los sistemas de tiempo compartido. Su amplio uso a lo largo de los años ha permitido que sea un lenguaje que no ha dejado de utilizarse, tanto en escuelas para la enseñanza, como en el desarrollo de aplicaciones financieras.

El tremendo éxito que los ordenadores tuvieron a lo largo de los años sesenta fue llevando a la creación de programas cada vez más complejos que llegaban a tener miles de líneas de código. Hacer correcciones a este tipo de programas y agregarles mejoras se fue convirtiendo en una labor muy ardua. Ante este problema que amenazaba con convertir los ordenadores en máquinas estériles, surgió un grupo de científicos de la computación de los cuales, Dijkstra y Wirth [Dij68], y de forma menos teórica pero quizás con más impacto, Kernighan y Ritchie, fueron de los más destacados. Dijkstra era conocido por su baja opinión de la sentencia *"GOTO"* en programación que culminó, en 1968, con el artículo *"Goto statement considered harmful"*. Este artículo fue un importante paso hacia el rechazo de la expresión *"GOTO"* y de su eficaz reemplazo por estructuras de control tales como el bucle *"WHILE"*. El famoso título del artículo no era obra de Dijkstra, sino de Niklaus Wirth, entonces redactor de comunicaciones del ACM. Sus ideas llevaron a la creación de un nuevo concepto, la programación estructurada.

La programación estructurada propone dos ideas básicas: no repetir código y proteger las variables que usa una parte del programa de que sean modificadas accidentalmente por otras partes del programa. Una sola repetición de código es fuente probable de errores y dificulta el mantenimiento de un programa. Para no repetir

código hay que escribir funciones o procedimientos que se encarguen de realizar siempre lo que las diferentes repeticiones realizarían. Para proteger las variables hay que desarrollar lenguajes que permitan definir variables locales, es decir, que sólo pueden ser utilizadas dentro de una función o procedimiento, de esta manera, ninguna otra función del programa puede cambiarla. Como consecuencia de estas ideas, disminuye considerablemente el tamaño de los programas y éstos se hacen más confiables y fáciles de corregir o mejorar. Para facilitar la programación estructurada aparecen nuevos lenguajes, notoriamente en la década de los setenta.

En 1966 Niklaus Wirth y Tony Hoare publican Algol W a partir de los trabajos del grupo ALGOL de la IFIP (International Federation for Information Processing). Se trata de un lenguaje conciso, simple de implementar, que evita todos los defectos conocidos del lenguaje Algol e incluye sus propias características adicionales. Entre sus principales cualidades cabe destacar: Aritmética de doble precisión, números complejos, Strings, estructuras de datos dinámicas, evaluación por valor y paso de parámetros por valor.

En 1971 Wirth desarrolla Pascal, un lenguaje estructurado, con propósitos académicos, basado en las ideas de Algol y Algol W. El éxito del lenguaje Pascal fue asombroso y debido a su aceptación en los ámbitos académicos se desarrollaron aplicaciones de las más variadas, hasta que la creciente necesidad de nuevas aplicaciones sobrepasó los límites para los que fue diseñado Pascal.

En 1972 se desarrolla el lenguaje C por Dennis Ritchie [Rit78] en los Laboratorios Bell. C es un lenguaje que se basa en los lenguajes B y BCPL, y una de sus características principales es que permite el acceso, prácticamente directo, al hardware, por lo que las personas dedicadas al desarrollo de software encontraron en C un lenguaje de alto nivel que permite desarrollar funciones de un ensamblador. El lenguaje C tuvo un gran éxito y sería incluido como parte del sistema operativo UNIX y con ello, empezaría la competencia por dominar el mercado de los sistemas operativos, otra vertiente en la historia del desarrollo de software.

Pese a que ni C ni Pascal aportaron nuevos conceptos al desarrollo de lenguajes, han tenido una presencia sólida alrededor del mundo, por lo que una gran cantidad de aplicaciones se han desarrollado utilizando estos lenguajes.

Uno de los defectos de la programación imperativa es que las variables globales pueden ser utilizadas y modificar sus contenidos desde cualquier punto del programa.

Los programas que carecen de disciplina para acceder a variables globales tienden a ser inmanejables. La razón es que los módulos que acceden a estas variables no se pueden comprender completamente de forma independiente, es decir, todo está relacionado con todo.

Este problema fue detectado alrededor de 1970 por David L. Parnas [Par72] quien propuso la norma de ocultar información como solución. Su idea era encapsular cada una de las variables globales del programa en un módulo junto con sus operaciones asociadas. Sólo estas operaciones podían tener acceso a las variables globales. El resto de los módulos podrían acceder a las variables sólo de forma indirecta mediante las operaciones diseñadas a tal efecto. En la actualidad se denomina objetos a este tipo de módulos.

De finales de la década de los sesenta es Simula67, desarrollado por Kristen Nygaard y Ole-Johan Dahl en Noruega [OJDN68]. Este lenguaje fue diseñado para simulaciones y su contribución principal es el entendimiento de un concepto fundamental en los lenguajes orientados a objetos: las clases. Nygaard y Dahl no encontraban lenguajes de programación que se ajustaran a sus necesidades, así que se basaron en el lenguaje Algol60 y lo extendieron con conceptos de objetos, clases, herencia, el polimorfismo por inclusión (que se obtiene introduciendo la herencia de clases) y procedimientos virtuales. Simula67 es considerado como el primer lenguaje orientado a objetos (OO).

Otros científicos experimentaron ideas similares creando diversos lenguajes de programación orientada a objetos como Smalltalk, desarrollado entre otros por Alan Kay [Kay96]. El objetivo era crear un sistema que permitiese expandir la creatividad de sus usuarios, proporcionando un entorno para la experimentación, creación e investigación. En ellos se experimentó con otras ideas útiles como la definición de subclases que heredan las propiedades de su superclase, es decir, de la clase de la que se derivan, pero agregando variables y funciones nuevas. También surgió una idea que ayudaría a evitar los difíciles problemas que surgían en el manejo de la memoria dinámica: los constructores y destructores de objetos. Los puntos importantes de este lenguaje fueron, por un lado, adoptar el concepto de objeto y clase como núcleo del lenguaje y por otro lado, el concepto de programación interactiva, incorporando las ideas ya conocidas de lenguajes funcionales, es decir, que se tuviese un lenguaje interpretado y no compilado.

En la decada de los ochenta el paradigma orientado a objetos comenzó a madurar

como un enfoque concreto de desarrollo de software. En la última década del siglo XX, el paradigma experimentó un gran progreso, tanto en el desarrollo de programas como en la forma de presentar las aplicaciones del sistema al usuario.

A principios de los 90 se popularizó un nuevo lenguaje orientado a objetos. Se trata del C++ creado por Bjarne Stroustrup [Str94]. La idea de Bjarne Stroustrup fue crear un lenguaje orientado a objetos que heredara prácticamente toda la sintaxis y posibilidades del lenguaje que en ese momento era más popular entre los programadores, el lenguaje C. Este truco ayudó a popularizar la programación orientada a objetos y preparó el camino para la aparición de JAVA y del resto de lenguajes orientados a objetos.

Los creadores de JAVA aprendieron bien la lección de Bjarne Stroustrup. Un nuevo lenguaje para tener éxito debía ser muy parecido al que en el momento de su lanzamiento fuese el más popular. Así, JAVA se creó con un gran parecido a C++, pero JAVA es un lenguaje más puro de programación orientada a objetos. Conserva un poco del C original, pero se parece más a C++. Puede verse a C++ como un paso intermedio en la transición de la programación estructurada de C a la programación orientada a objetos más pura de JAVA. Microsoft está tratando de impulsar el lenguaje llamado C# (C sharp) que es aún más puro como lenguaje orientado a objetos. Adopta prácticamente todas las mejoras de JAVA y agrega algunas nuevas.

Actualmente la Tecnología Orientada a Objetos (TOO) no solo se aplica a los lenguajes de programación, sino que también se ha propagado a los métodos de análisis y diseño y a otras áreas, tales como las bases de datos y/o las comunicaciones y sistemas operativos. Por lo tanto, para hacer desarrollo de sistemas de software basados en el paradigma orientado a objetos (POO), hay que entender bien todos los conceptos del modelo de objetos que está detrás de ella y sus antecedentes históricos.

1.2.3. El problema

Los sistemas de automatización industrial en la actualidad se basan, en su gran mayoría, en el empleo de autómatas programables (PLCs). La creciente complejidad de los sistemas bajo control exige equipos cada vez más evolucionados, pero en cuanto al software, se vienen empleando los mismos lenguajes y herramientas de programación desde hace décadas basados fundamentalmente en lenguaje

de contactos. Este desfase se traduce en costes, tiempo de desarrollo y mantenimiento de los programas para autómatas programables excesivamente elevados, lo que constituye un auténtico cuello de botella en el desarrollo de instalaciones automatizadas. Esta situación tiene un claro precedente en el terreno de software para ordenadores personales, solventada mediante la aparición de potentes lenguajes de programación apoyados en la tecnología orientada a objetos, basada en la generación, adaptación e interconexión de *"objetos"* de software reusables.

Solamente la presión de poderosos clientes de este tipo de productos como los del sector de la automoción ha propiciado, a partir de la década de los noventa, el desarrollo de una corriente tendente a la estandarización de los PLCs, al menos en el ámbito de la programación, con el objetivo fundamental de lograr la compatibilidad en el ámbito de los programas de control. Esta corriente, canalizada por la Comisión Electrotécnica Internacional (IEC) con base en Ginebra (Suiza), cristalizó finalmente en 1993 en un estándar internacional [IEC00g, IEC00a, IEC00b, IEC2a, IEC2b, IEC93] en el que se normalizan una serie de aspectos de los controladores programables.

Mientras que el desarrollo de nuevos lenguajes de programación en el mundo de la informática ha sufrido una evolución sin pausas desde la aparición de los primeros ordenadores, llegando en la actualidad a desarrollarse sistemas distribuidos orientados a objetos, sistemas de juicio experto u objetos activos (agentes), los tímidos intentos de las instituciones de investigación de sistemas de automatización por desarrollar nuevos lenguajes y técnicas de programación e incluso nuevos lenguajes de modelado orientados al control de procesos, ha chocado frontalmente con el conservadurismo de los fabricantes de PLCs y de las empresas consumidoras de sus productos que no alcanzan a ver los beneficios que este tipo de iniciativas les podrían proporcionar.

Al comparar las curvas de evolución de los paradigmas de programación en las disciplinas de desarrollo de software en informática y en PLCs (ver figura 1.4), se observa que a pesar de la estandarización de los lenguajes de programación para en el control de procesos gracias a la norma IEC 61131, se aprecia un estancamiento en las primeras técnicas de programación de sistemas de control.

Las razones de estas distintas velocidades de evolución pueden encontrarse en los distintos tamaños y caracteres de los tipos de negocios en que se aplican ambas disciplinas. Mientras los ordenadores se aplican para facilitar la tarea en cualquier

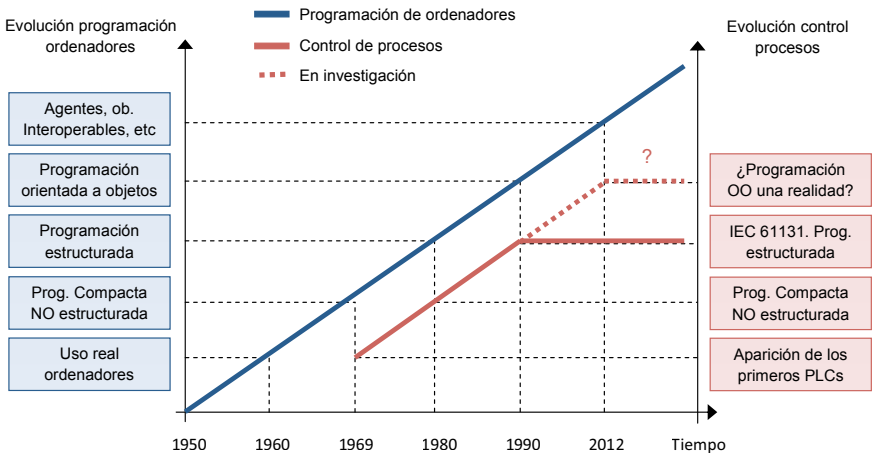

Figura 1.4: Curva de evolución de la programación informática vs control de procesos

orden de la vida, desde el sector bancario, hasta la navegación espacial, pasando el ocio, etc, los autómatas programables se aplican sólo en la automatización industrial, sector que además por lo general suele ser más conservador que algunos de los anteriores.

Otro de los motivos que pudieran justificar el retraso tecnológico en la programación de autómatas con respecto a su hermana, la programación de ordenadores, pudiera ser la falta histórica de estandarización y compatibilidad a nivel hardware y de programación de los PLCs. Mientras que en la Ingeniería del Software la plataforma hardware más impuesta ha sido el ordenador personal de arquitectura AT que ejecuta el sistema operativo Windows de la empresa Microsoft, en la automatización industrial cada fabricante ha desarrollado su propia arquitectura hardware y sistema operativo para sus PLCs.

Un factor importante que pudiera justificar en parte este desfase evolutivo, habría que buscarlo en la restricción impuesta por los consumidores de PLCs respecto a la no necesidad de conocimientos informáticos por parte del personal encargado de su programación y manejo. Ésto implica la necesidad de crear lenguajes de programación de PLCs sencillos.

Por otro lado, en la Ingeniería del Software, los lenguajes de programación han sido tradicionalmente desarrollados por centros de investigación independientes

de cualquier empresa fabricante de ordenadores, adaptándose éstas últimas a las especificaciones del lenguaje estándar y limitándose a desarrollar sus propias herramientas de programación. Por el contrario, en la ingeniería de la lógica de control, el desarrollo de los lenguajes de programación ha estado siempre en manos de los propios fabricantes de PLCs lo que ha provocado la incompatibilidad entre los mismos.

Aunque se atisba cierta tendencia hacia el desarrollo de sistemas de control orientados a objetos, principalmente promovido por empresas como 3S-Smart Software Solutions, la realidad es que el desarrollo de sistemas orientados a objetos está básicamente reducido a ámbitos académicos. Posiblemente, aún hoy siga teniendo gran peso en la decisión del desarrollo de nuevos lenguajes o técnicas de programación el hecho de que el personal que en último término lo va a emplear carezca de los conocimientos informáticos necesarios para la utilización de lenguajes y técnicas de alto nivel.

1.3. Objetivos

Se observa claramente la necesidad de realizar un esfuerzo investigador en el sentido de tratar de desarrollar o adaptar lenguajes de programación orientados a objetos que permitan el desarrollo de programas de control sobre PLCs bajo este paradigma.

El objetivo principal de esta tesis sería el de dotar a los PLCs de un conjunto de lenguajes y técnicas de programación orientadas a objetos, pero dado el fuerte conservadurismo del sector industrial, el nuevo lenguaje orientado a objetos no debería romper con todo lo anterior y comenzar desde cero, sino que sería más inteligente tratar de modificar las técnicas y arquitecturas de la actualidad al objetivo final de la orientación a objetos.

Como objetivo secundario, se pretende medir en qué grado el desarrollo de un programa de control bajo un paradigma orientado a objetos mejora (o empeora) a otro desarrollado con las técnicas tradicionales de implementación de programas en PLCs desde dos puntos de vista. Por un lado, se medirán de forma empírica los tiempos de ejecución de un experimento programado bajo los dos paradigmas que se mencionan (estructurado y OO). Por otro lado, se compararán de forma

razonada los dos estilos de programación con los que se ha desarrollado el experimento.

1.4. Estructura de la tesis

El presente documento de memoria de tesis está dividido en tres partes claramente diferenciadas, a saber:

1. Una exposición detallada de los planteamientos de partida y de los objetivos que se pretenden alcanzar con la presente investigación contenida en este capítulo.

2. Se continúa el documento en el capítulo 2 en el que se presenta una compilación de las distintas estrategias empleadas en la industria y la enseñanza referentes al paradigma de programación orientado a objetos en el control de procesos, y cómo en este paradigma se apoyan las técnicas de modelado y especificación de los sistemas de control y de informática. Se completa esta segunda parte con la descripción detallada del nuevo lenguaje orientado a objetos desarrollado en esta tesis (capítulo 3) y de la herramienta software de soporte a la aplicación de este nuevo lenguaje (capítulo 4).

3. Finalmente, en el capítulo 5 se presentan y explican los resultados experimentales obtenidos al programar sistemas de control con el nuevo lenguaje de programación orientado a objetos presentado en el capítulo 3, mediante la utilización de la herramienta software que se ha presentado en el capítulo 4. Por último, se cierra la presente tesis con una discusión crítica y razonada de las aportaciones y beneficios que potencialmente se pudieran derivar del presente trabajo, así como la sugerencia de futuros estudios que vinieran a mejorar, ampliar o completar ciertos aspectos de éste.

1.5. Conclusiones

Aunque se atisba una cierta evolución hacia la aplicación de técnicas de programación orientadas a objetos aplicadas al control de procesos, éstas están básicamente reducidas al entorno académico sobre todo si se comparan con la aplicación del

POO en el mundo informático. Además, todas estas investigaciones por lo general están enfocadas hacia un alto nivel de abstracción, es decir, consideran el problema de la automatización de procesos desde una perspectiva de integración de varios sistemas automatizados con el objetivo de optimizar la producción (quizá a este nivel, la resistencia de nuevas soluciones no es tan acentuada).

Probablemente, aún a día de hoy siga teniendo un gran peso a la hora de tomar la decisión de diseñar un nuevo lenguaje o técnica de programación, el hecho de que el personal que en último término los va a emplear carezca de los conocimientos informáticos necesarios para la utilización de lenguajes y técnicas de alto nivel.

Teniendo en cuenta que desde la ingeniera del software la programación orientada a objetos (POO) se ha impuesto en el mundo empresarial de desarrollo de nuevos programas, cabría esperar una evolución similar en la automatización de procesos que permitiera aprovecharse de la potencia de la POO tal y como ha ocurrido en los últimos años en el mundo informático, como medio para el desarrollo de sistemas más escalables, fáciles de mantener e implementables a menor coste. El problema de la adopción POO en la automatización de procesos es la necesidad del manejo y conocimiento de técnicas de modelado, análisis y POO por parte del ingeniero de control, lo que explica el salto evolutivo entre las actuales técnicas de programación en el mundo informático y de la automatización industrial.

Capítulo 2

Orientación a objetos en la informática y en el control de procesos

La observación y la percepción son dos cosas separadas;
el ojo que observa es más fuerte, el ojo que percibe es más débil.
Una especialidad de las artes marciales es ver de cerca lo que
está lejos y ver lo que está cerca con distancia.
Miyamoto Musashi (Anillo del agua)

2.1. Introducción

En la actualidad, no sería rentable generar un programa informático altamente eficaz y eficiente para solventar una necesidad determinada, si cuando cambia en algún sentido esa necesidad hay que redefinir por completo el sistema para que vuelva a ser útil. En este sentido, la reutilización de componentes ya creados es

una de las virtudes más destacables de la programación orientada a objetos, pues permite realizar cambios mínimos en los sistemas para adaptarse a los nuevos requisitos.

Schach [Sch99] responde a la pregunta de si el paradigma OO es una técnica de programación superior al resto de técnicas de programación actuales citando una investigación realiza por parte de IBM [CJ94] sobre tres proyectos desarrollados con tecnologías OO. El resultado de la investigación mostro que en casi todos los aspectos, el POO supera al paradigma estructurado. En particular, hubo importantes descensos en el número de errores al programar cambios imprevisibles en las especificaciones funcionales del sistema. Además, hubo una significativa mejora del mantenimiento adaptativo y perfectivo, así como una notoria mejora en la usabilidad.

Por otro lado, Brown [Bro97] y Koontz [Koo94] observaron dos problemas en la adopción del POO por parte de las empresas:

- Los primeros dos proyectos OO resultan ser un 30 % más caros que si se desarrollasen con técnicas estructuradas de programación. Sin embargo, los costes a partir del tercer proyecto en adelante se reducen significativamente siendo muy inferiores a los desarrollados bajo un paradigma estructurado.
- El rendimiento a nivel de computación de un programa implementado bajo un POO no mejora al desarrollado bajo un paradigma estructurado. Incluso dependiendo del compilador, el código objeto generado por un programa OO puede superar en tamaño a uno implementado con técnicas estructuradas.

A pesar de que la adopción de un POO puede resultar más cara al principio (principalmente por la necesidad de aprender nuevas técnicas), las ventajas superan con creces a las desventajas ya que una implementación OO produce sistemas más fáciles de diseñar, implementar y depurar, así como la aplicación de los pilares de la OO (encapsulación, herencia, polimorfismo, etc) permite una reutilización de código muy superior a la programación estructurada, lo que se traduce en un aumento de la productividad y una disminución de los costes de desarrollo [Pos01].

Además, la necesidad de rentabilizar al máximo todo el esfuerzo y dinero que las empresas invierten en la construcción de programas informáticos, requiere de un proceso de modelado de soluciones concretas ante problemas específicos. En este

sentido, la Ingeniería del Software se ha beneficiado de la capacidad de descripción que la orientación a objetos posee del mundo real, lo que ha impulsado el desarrollo de numerosas Métodologías orientadas a objetos y sistemas de modelado que reducen considerablemente el tiempo de desarrollo y mantenimiento de los programas.

Este mismo principio puede trasladarse al control de procesos, en el que una descomposición de un sistema grande en pequeñas partes manejables (objetos) permitiría adoptar cambios en el sistema de forma segura y sencilla, o la reutilización de esas pequeñas partes en otros sistemas distintos.

El objetivo principal de este capítulo consiste en hacer una revisión exhaustiva del grado de adaptación del paradigma de orientación a objetos en los sistemas de control de procesos, y más concretamente, en aquellos que emplean PLCs basados en la norma IEC 61131. Pero para poder llegar a entender completamente el significado de las distintas tecnologías de orientación a objetos en el ámbito del control de procesos, es necesario previamente mostrar una panorámica de todos aquellos aspectos que han conducido a que el paradigma de orientación a objetos sea actualmente el modelo más ampliamente utilizado en la industria de desarrollo de software en informática, frente al modelo de creación de software asociado al paradigma tradicional de programación estructurada, y cómo a raíz de la aparición de la orientación a objetos se ha impulsado el desarrollo de Métodologías y técnicas de análisis y modelado que faciliten la tarea de los ingenieros., tanto en el campo del desarrollo de software de informática (apoyado en la Ingeniería del Software), como en la implementación de programas de automatización industrial (sustentado en la Ingeniería de Control).

2.2. Perspectiva algorítmica versus perspectiva orientada a objetos

Los problemas que se intentan resolver mediante aplicaciones software conllevan a menudo elementos de complejidad ineludible, en los que se encuentra una cantidad ingente de requisitos que compiten entre sí. El desarrollo de estos sistemas software es una inversión considerable por lo que no es admisible el desechar un sistema existente cada vez que los requerimientos cambian. Esté o no previsto, los sistemas tienden a evolucionar en el tiempo.

El tamaño no es una gran virtud para un sistema de software. Se hace lo posible por escribir menos código mediante la invención de mecanismos ingeniosos y potentes que dan esta ilusión de simplicidad, así como mediante la reutilización de marcos estructurales de diseños y código ya existentes. Sin embargo, a veces es imposible eludir el intrínseco volumen de los requerimientos de un sistema y se plantea la obligación de, o bien escribir una enorme cantidad de nuevo software o bien, reutilizar software existente de nuevas formas.

La técnica de dominar la complejidad se conoce desde tiempos remotos: *"divide et impera"* (divide y vencerás). Cuando se diseña un sistema de software complejo, es esencial descomponerlo en partes más y más pequeñas, cada una de las cuales se puede refinar entonces de forma independiente. De este modo, se satisface la restricción fundamental que existe sobre la capacidad de canal de la comprensión humana: para entender un nivel dado de un sistema, basta con comprender unas pocas partes (no necesariamente todas) a la vez.

2.2.1. Descomposición algorítmica

Tradicionalmente, la programación fue hecha en una manera secuencial o lineal, es decir, una serie de pasos consecutivos con estructuras consecutivas y bifurcaciones.

Los lenguajes basados en esta forma de programación (ver figura 2.1) ofrecían ventajas al principio, pero el problema ocurre cuando los sistemas se vuelven complejos. Estos programas escritos al estilo *"spaghetti"* no ofrecen flexibilidad y al contener una gran cantidad de líneas de código en un sólo bloque, el mantenimiento se vuelve una tarea complicada.

Frente a esta dificultad aparecieron los lenguajes basados en la programación estructurada. La idea principal de esta forma de programación es la de separar las partes complejas del programa en módulos o segmentos que sean ejecutados conforme se requieran, es decir, en procedimientos y funciones. De esta manera, tenemos un diseño modular compuesto por módulos independientes que puedan comunicarse entre sí. Poco a poco, este estilo de programación fue reemplazando al estilo *"spaghetti"* impuesto por la programación lineal.

Bajo este estilo de programación, la unidad base de ejecución son programas o conjuntos de instrucciones ejecutables que se dividen en módulos o rutinas (utilizando la terminología informática). Una aplicación suele estar formada por una

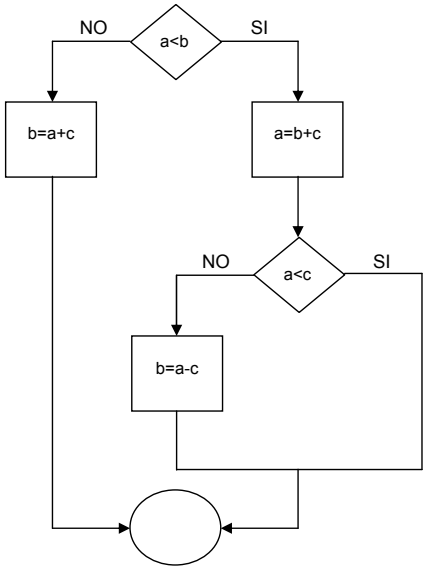

Figura 2.1: Algoritmo de programación secuencial

jerarquía más o menos definida de programas y módulos que pueden llamarse unos a otros. Esta jerarquía se organiza en torno a un programa principal desde el cual se accede a otros módulos llamados subrutinas o subprogramas. Los datos tienen un papel secundario y no son más que aquello con lo que se alimenta a los programas para que realicen su función. La programación estructurada se puede resumir en la siguiente expresión acuñada por Niklaus Wirth [Wir76]:

$$\text{Algoritmos} + \text{Estructura de datos} = \text{Programas}$$

En este paradigma de programación, la clave reside en decidir qué procedimientos se quieren y en utilizar los mejores algoritmos que se encuentren. Los lenguajes de programación estructurada dan soporte a este paradigma con facilidades para pasar argumentos a las funciones empleadas y para retornar valores desde estas funciones.

Fortran fue el lenguaje de procedimientos original; otros lenguajes como Algol60, Algol68, C y Pascal fueron invenciones posteriores en la misma tradición.

Casi todos los informáticos han sido adiestrados en el dogma del diseño estructurado descendente, y por eso suelen afrontar la descomposición como una simple

cuestión de descomposición algorítmica (descomposición de algoritmos grandes en otros más pequeños) en la que cada módulo del sistema representa un paso importante de algún proceso global. No hay nada inherentemente malo en este punto de vista, salvo que tiende a producir sistemas frágiles, puesto que cuando los requisitos cambian y el sistema crece, los sistemas construidos con un enfoque algorítmico se vuelven muy difíciles de mantener.

2.2.2. Descomposición orientada a objetos

La visión actual del desarrollo de software toma una perspectiva orientada a objetos. En este enfoque, el principal bloque de construcción de todos los sistemas software es el objeto o clase. Para explicarlo sencillamente, un objeto es un elemento generalmente traído del vocabulario del espacio del problema o del espacio de la solución. Una clase es una descripción de un conjunto de objetos similares. Todo objeto tiene identidad (puede nombrarse o distinguirse de otra manera de otros objetos), estado (generalmente hay algunos datos asociados a él) y comportamiento (se le pueden hacer cosas al objeto y él a su vez puede hacer cosas a otros objetos).

Aunque ambos diseños resuelven el mismo problema, lo hacen de formas bastante distintas. En esta segunda aproximación se ve el mundo como un conjunto de agentes autónomos (objetos) que colaboran para llevar a cabo algún comportamiento de nivel superior. No existen algoritmos concebidos como elementos independientes. En lugar de eso, son sustituidos por operaciones asociadas a los objetos pertenecientes al sistema. Cada objeto contiene su propio comportamiento bien definido. A los objetos se les pide que hagan *"algo"*, *"una acción"*, enviándoles mensajes. Puesto que esta descomposición está basada en objetos y no en algoritmos, se le llama descomposición orientada a objetos.

En contraste con la aproximación estructurada, que centra el punto de atención en las funciones o rutinas como base de ejecución, en esta nueva aproximación, son los objetos en lugar de las funciones los que forman la jerarquía básica. La programación orientada a objetos fue concebida por personas que no veían en su entorno acciones sino objetos que interactuaban unos con otros según su naturaleza (ver figura 2.2). Las acciones aplicadas a los objetos dependen de éstos, lo que representa, en términos informáticos, que los programas y subprogramas pasan a

un nivel secundario dependiendo de los datos. En este sentido, una clase son unos datos y unos métodos que operan sobre esos datos.

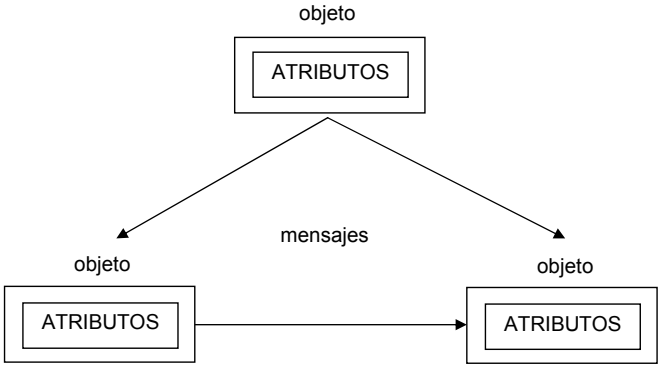

Figura 2.2: Interacción entre objetos

En síntesis, la programación orientada a objetos puede ser resumida de la siguiente manera:

Objetos + Flujo de mensajes = Programas

2.2.3. Descomposición algorítmica versus descomposición orientada a objetos

¿Cuál es la forma correcta de descomponer un sistema complejo, por algoritmos o por objetos? Según Booch [Boo94], la respuesta adecuada es que ambas visiones son importantes:

- La visión algorítmica enfatiza el orden de los eventos mientras que la visión orientada a objetos resalta los agentes que, o bien causan acciones o bien estan sujetos de esas acciones. Sin embargo, el hecho es que no se puede construir un sistema complejo de las dos formas a la vez porque son vistas completamente perpendiculares.

- La descomposición orientada a objetos tiene una serie de ventajas altamente significativas sobre la descomposición algorítmica.

- La descomposición orientada a objetos produce sistemas más pequeños a través de la reutilización de mecanismos comunes, proporcionando así una importante economía de expresión.
- Los sistemas orientados a objetos son también más resistentes al cambio y por tanto están mejor preparados para evolucionar en el tiempo porque su diseño está basado en formas intermedias estables.

En realidad, la descomposición orientada a objetos reduce en gran medida el riesgo que representa construir sistemas de software complejos, porque están diseñados para evolucionar de forma incremental partiendo de sistemas más pequeños en los que ya se tiene confianza. Es más, la descomposición orientada a objetos resuelve directamente la complejidad innata del software ayudando a tomar decisiones respecto a la separación de intereses en un gran espacio de estados.

2.3. La Orientación a Objetos (OO) como paradigma en el desarrollo de software

La Orientación a Objetos (OO), por tanto, es una nueva forma de entender el desarrollo del software que abarca un conjunto de Métodologías y herramientas para el modelado y la implementación del software, las cuales hacen más fácil la construcción de sistemas complejos a partir de componentes individuales.

La génesis de las ideas básicas de la OO se produce a principios de los años 60 y se atribuye al trabajo del Dr. Nygaard y su equipo de la Universidad de Noruega. Estos investigadores se dedicaban al desarrollo de sistemas informáticos para simular sistemas físicos, tales como el funcionamiento de motores. La dificultad con la que se encontraban era doble: los programas eran muy complejos y forzosamente tenían que ser muy modificables. Este segundo punto era especialmente problemático ya que eran muchas las ocasiones en las que se requería probar la viabilidad y el rendimiento de estructuras alternativas.

La solución que idearon fue diseñar el programa con una estructura paralela a la del sistema físico, es decir, si el sistema físico estaba compuesto por cien componentes, el programa también tenía cien módulos, uno por cada pieza. Partiendo el programa de esta manera, había una total correspondencia entre el sistema físico y

el sistema informático, dado que cada pieza tenía implementada su abstracción en un módulo informático y que los módulos se comunicaban enviándose mensajes, de la misma forma que los sistemas físicos se comunican enviándose señales. Para dar soporte a estas ideas crearon un lenguaje denominado Simula-67 [DJ67, OJDN68].

Este enfoque resolvió los dos problemas planteados. En primer lugar, ofrecía una forma natural de dividir un programa muy complejo en partes más sencillas y, en segundo lugar, se simplificaba considerablemente el mantenimiento de dicho programa, permitiendo al investigador el cambio de piezas enteras o la modificación del comportamiento de alguna de ellas sin tener que alterar el resto del programa.

En la década de los años 70, Alan Kay, de la Universidad de Utah, formó un grupo de investigación junto con Adele Goldberg y Dan Ingalls de Xerox (Palo Alto). Este grupo diseñó un entorno y un lenguaje de programación llamado Smalltalk que incorporaba las ideas de la orientación a objetos [Sav90].

En la década de los 80, Bjarne Stroustrup, de ATT-Bell, parte de Smalltalk y Simula para diseñar el lenguaje C++, como sucesor del C [Str86].

En el ámbito académico, Bertrand Meyer (1991) crea el lenguaje Eiffel, reconocido como el más completo y elegante de los lenguajes orientados a objetos (LOO).

Paralelamente, en el mundo de la inteligencia artificial se desarrolla, entre otros, el lenguaje CLOS (Common Lisp Object System), como variante orientada a objetos del lenguaje Lisp [Moo89, Ste90].

Actualmente, sin duda, uno de los lenguajes con mayor impacto en la industria de desarrollo de software es el lenguaje JAVA, lenguaje de programación OO desarrollado por Sun Microsystems en 1995. Sun Microsystems lanzó el entorno JDK 1.0 en 1996 [GS05], primera versión del kit de desarrollo de dominio público que se convirtió en la primera especificación formal de la plataforma JAVA. Desde entonces han aparecido diferentes versiones, aunque la primera comercial se denominó JDK 1.1 y fue lanzada a principios de 1997. En diciembre de 1998 Sun lanzó la plataforma JAVA 2 (conocida también como JDK 1.2). Esta versión ya representó la madurez de la plataforma JAVA, aunque se han seguido incorporando hasta la fecha nuevas funcionalidades, en este sentido, es posible acceder a JAVA SE 7 desde agosto del 2011.

La popularidad y aceptación general de las tecnologías orientadas a objetos comenzó a producirse en la década de los 90. Aparecieron nuevas revistas enteramente

dedicadas a la OO, como el Journal of Object-Oriented Programming, el Object Magazine, etc, y se comenzaron a celebrar congresos y conferencias especializadas, como la OOPSLA, centrada en los sistemas de programación y lenguajes orientados a objetos.

Las razones del rápido desarrollo de la programación orientada a objetos en los últimos 20 años ha sido debido fundamentalmente a dos razones:

- Una mejor modelación de aplicaciones del mundo real lo que permite una especificación más clara de los requisitos de las aplicaciones.
- Los propios objetos se pueden construir a partir de otros que a su vez pueden estar formados por otros objetos. Este proceso conlleva un beneficio muy importante, la reusabilidad del software durante el desarrollo de un sistema.

2.3.1. Principios de la Orientación a Objetos

En este apartado se exponen, de forma sintética, los conceptos o principios básicos de la OO: Encapsulado y ocultación de la información, clasificación, tipos abstractos de datos, herencia y polimorfismo.

2.3.2. Encapsulado y ocultación de la información

En el paradigma de programación tradicional orientada al proceso, como el análisis y el diseño estructurado de De Marco [DeM82], se produce una dicotomía entre los dos elementos constituyentes de un sistema: funciones que llevan a cabo los programas y datos que se almacenan en ficheros o bases de datos. En la OO, sin embargo, se propugna un enfoque unificador de ambos aspectos que se encapsulan en los objetos con la finalidad de modelizar y representar mejor el mundo real [Boo84, Boo94, Str88].

A nivel conceptual, un objeto es una entidad percibida en el sistema que se está desarrollando, mientras que a nivel de implementación, un objeto se corresponde con el encapsulado de un conjunto de operaciones (habitualmente denominadas métodos o servicios) que pueden ser invocados externamente y de un conjunto de variables (habitualmente denominadas atributos) que almacenan el estado resultante de dichas operaciones. Sólo el propio objeto tiene la capacidad de acceder y

modificar sus datos mediante los métodos que tiene implementados. En orientación a objetos se habla de evento cuando un método modifica un atributo de estado.

El encapsulado permite ocultar a los usuarios de un objeto los aspectos instrumentales (la propia programación), ofreciéndoles únicamente un interfaz externo mediante el cual interactuar con el objeto. Este principio de ocultación es fundamental, puesto que permite modificar los aspectos privados de un objeto sin que se vean afectados los demás objetos que interactúan con él, siempre que se conserve el mismo interfaz externo. Dicho de otro modo, el encapsulado proporciona al programador libertad en la implementación de los detalles internos de un sistema con la única restricción de mantener el interfaz abstracto que ven los usuarios externos.

Martin y Odell [MO94] reproducen una interesante analogía propuesta por David Taylor [Tay97] entre los conceptos de la OO enunciadas hasta ahora y los manejados por la citología.

David Taylor señalo que el diseño orientado a objetos refleja las técnicas de la naturaleza. Todos los seres vivos están compuestos por células. Las células son paquetes organizados que al igual que los objetos, combinan la información y el comportamiento. La información de las células está en el ADN y en las moléculas de proteínas del núcleo.

Los métodos de la célula los realizan orgánulos que rodean al núcleo. La célula está cubierta por una membrana que protege y oculta la labor celular de cualquier intrusión del exterior. Las células no pueden *"leer"* las moléculas de proteína o controlar la estructura de las demás células; sólo *"leen"* y controlan lo propio. En vez de ésto, envían solicitudes químicas a las demás células. Al empaquetar de ésta manera la información y el comportamiento, la célula se encapsula.

Taylor comentó:

> Esta comunicación basada en mensajes hace mucho más sencillo el funcionamiento celular. La membrana oculta la complejidad de la célula y presenta un interfaz relativamente sencillo al resto del organismo. Como se puede ver por la estructura celular, el encapsulado es una idea que ha estado latente durante mucho tiempo.

2.3.3. Clasificación, tipos abstractos de datos y herencia

La potente disciplina que subyace bajo el paradigma de la OO es la tipificación o clasificación de datos abstractos. Los tipos de datos abstractos, habitualmente denominados clases, definen conjuntos encapsulados de objetos reales conceptualmente similares. En los lenguajes orientados a objetos, las clases se utilizan para describir los tipos de datos abstractos y se reserva el término objeto para referir las realizaciones o casos concretos de las clases (instancias) que se generan durante la ejecución de los programas. Una clase define los datos que se están almacenando (variables o atributos) y las operaciones (métodos) soportadas por los objetos que son instancias de la clase. La idea principal de la OO es que un sistema orientado a objetos es un conjunto de objetos que interactúan entre sí y que están organizados en clases como se muestra en la figura 2.3.

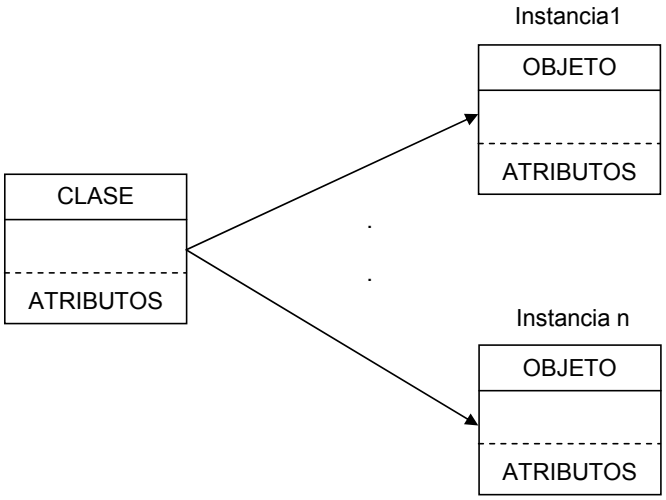

Figura 2.3: Instancias de una clase

El concepto de clase (tipo abstracto de datos) lleva a la noción de abstracción, entendida como el proceso de capturar los detalles fundamentales de un objeto mientras se suprimen o ignoran los detalles. Booch [Boo94] reproduce una afirmación de Hoare [DH72] que sugiere que:

> La abstracción surge de un reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de

concentrarse en esas similitudes e ignorar por el momento las diferencias.

La abstracción proporciona un mecanismo crucial para permitir que las personas comprendan, se comuniquen y razonen sistemas complejos. Sin abstracción, el nivel de detalle requerido para comprender un sistema hace que las personas sean incapaces de construir sus modelos mentales de cómo se estructura el sistema y cómo funciona. La noción de abstracción entraña la descomposición de un sistema complejo o complicado en sus partes más fundamentales y la descripción de esas partes con un lenguaje sencillo y preciso. Por ejemplo, las personas no piensan en un coche como centenares de elementos, sino como un objeto bien definido con un comportamiento propio. Esta abstracción permite a las personas utilizar un coche para conducirlo sin tener que preocuparse de la complejidad de las partes que forman el coche, es decir, pueden ignorar los detalles de cómo funciona el motor, los frenos o el sistema de refrigeración.

Realmente, al igual que sucede en el mundo real, lo que interesa es considerar el objeto como un todo. Al adoptar una visión del entorno orientado a objetos, el empleo de la abstracción permite utilizan bloques de información de contenido semántico cada vez mayor. Ésto es así, puesto que los objetos, como abstracciones de entidades del mundo real, representan un agrupamiento de información particularmente denso y cohesivo [Boo94].

Otra noción importante en los lenguajes orientados a objetos es el concepto de herencia. Es posible derivar nuevas clases a partir de una clase dada o realizar (instanciar) directamente objetos mediante un proceso de herencia, concepto que se define de forma parecida a la herencia en el sentido biológico. Se puede crear una nueva clase u objeto heredando los atributos y servicios de una o varias clases padre (herencia simple y múltiple, respectivamente). Las nuevas clases que se van creando por herencia configuran las denominadas "*jerarquías de clases*". En este contexto se utiliza el término *"superclase"* para referir cualquiera de las clases de orden superior en una misma jerarquía.

En la figura 2.4 se ilustra un ejemplo de una jerarquía en la que las clases hijas heredan propiedades comunes de la clase padre.

En el siguiente párrafo se ilustran los conceptos de clase y herencia [MO94], retomando la analogía propuesta por David Taylor [Tay97] entre la orientación a objetos y el funcionamiento y estructura celular:

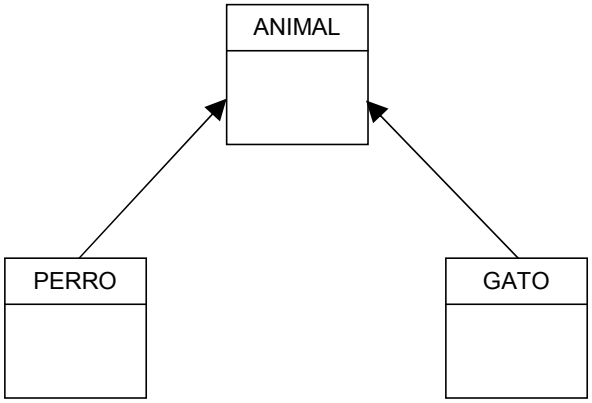

Figura 2.4: Ejemplo de jearquía de clases

> Las células son un admirable bloque universal de construcción de la naturaleza. Existen células sanguíneas que transportan sustancias químicas, células del cerebro, células óseas, células que permiten el funcionamiento de la retina del ojo y células musculares que distorsionan su forma para llevar a cabo funciones mecánicas. Los componentes de todas las plantas, insectos, peces y mamíferos están constituidos de células de estructura común que actúan según ciertos principios básicos. En principio, todo el software se podría construir de manera análoga con ciertas clases. Aunque existe una gran diversidad de células, muchas de ellas, como los objetos, tienen tipos similares. Un tipo de célula puede operar de manera similar a otra, puesto que ambas han heredado propiedades semejantes con la evolución. Las células se agrupan en órganos, como los músculos o las uñas de los pies. Los órganos se agrupan en sistemas y aparatos como, por ejemplo, el sistema nervioso. Un organismo está compuesto por varios sistemas y aparatos. Aunque forme parte de un organismo complejo, cada célula actúa por su cuenta, como un objeto, sin conocer la razón por la que se le envía un mensaje o las últimas consecuencias de su comportamiento.

La herencia es un principio muy importante de la orientación a objetos porque es precisamente en ella donde radica la reusabilidad, la extensibilidad y el bajo coste de mantenimiento de los sistemas informáticos basados en objetos.

Generalmente, los lenguajes orientados a objetos incluyen una vasta jerarquía ini-

cial de clases, a partir de la cual se pueden empezar a construir las aplicaciones. Cabe señalar que los lenguajes que no admiten la herencia múltiple generalmente permiten simularla, por ejemplo, mediante agregación de clases.

Para facilitar la definición de las clases y objetos, y de las relaciones que se establecen entre ellas (ya sean de herencia o de agregación), se acostumbra a utilizar las tarjetas CRC (Class, Responsability and Collaboration) [WBW90]. La figura 2.5 presenta el diseño de tarjeta propuesto por Graham [Gra00].

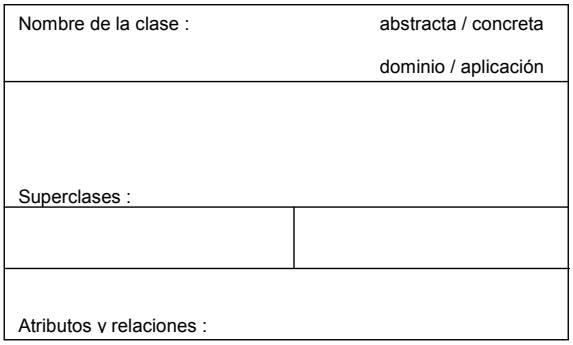

Figura 2.5: Estructura de una tarjeta CRC

Las tarjetas CRC contienen toda la información relativa a las *"responsabilidades"* (atributos y servicios) de la clase, así como las *"colaboraciones"* o relaciones que ésta mantiene con otras clases. En la propuesta de Graham [Gra95] se distingue si la clase es del *"dominio"* o de la *"aplicación"*. Una clase del dominio implica que es persistente (está almacenada en disco) y que, probablemente, será estable en la estructura durante todo el tiempo de vida del programa. Las clases u objetos de la aplicación son más volátiles, creándose y destruyéndose en tiempo de ejecución. Las clases también pueden ser *"abstractas"* o *"concretas"*, según permitan la derivación por herencia sólo de nuevas clases o también de objetos, respectivamente. Para cada clase se indica, en el apartado *"superclases"*, las clases de las cuales ha heredado parte de sus atributos y servicios. También se especifica el nombre, visibilidad (público o privado) y tipo y valor por defecto (si existe) de cada uno de los atributos de la clase. Un atributo puede ser a su vez una clase, en cuyo caso se debe escribir el nombre de ésta, estableciéndose una relación de agregación entre ambas clases. La agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (también llamado el agregado) hasta sus partes (conocidas también como atributos). A través de esta relación el agregado puede enviar mensajes a sus

partes.

Cada servicio se detalla con un nombre y una descripción de su algoritmo en pseudo-código. Cuando un servicio necesita acceder a otra clase u objeto para obtener información, se indica el nombre de dicha clase en el apartado servidores. Además, se debe establecer si el servicio es público (se puede invocar desde clases de otras jerarquías) o pertenece al comportamiento privado de la clase (sólo puede ser invocado por clases u objetos que derivan de ella).

Volviendo al concepto de encapsulado y ocultación de la información, dado que el propósito de una clase es encapsular complejidad, hay mecanismos para ocultar la complejidad de la implementación dentro de la misma. Cada método (servicio) o variable (atributo) de una clase puede ser público o privado, como ya se ha indicado. En este sentido, el interfaz público de una clase representa todo lo que los usuarios externos de la clase necesitan conocer o pueden conocer. Los métodos y variables privados sólo pueden accederse por el código que es miembro de la clase. Por consiguiente, cualquier código que no es miembro de la clase no puede acceder a un método o atributo privado. Bocch [Boo94] expone que la abstracción y el encapsulado son conceptos que se complementan: la abstracción se centra en el comportamiento observable de un objeto, mientras que el encapsulado se centra en la implementación que da lugar a ese comportamiento, ocultando todos los secretos de un objeto que no contribuyen a sus características esenciales. El encapsulado proporciona barreras explícitas entre abstracciones diferentes y por tanto conduce a una clara separación de intereses.

Finalmente, retomando la descripción de la figura 2.5, las reglas que aparecen en la parte inferior de la tarjeta se indican con notaciones simples del tipo *"SI... ENTONCES"* y expresan, por ejemplo, las condiciones que se deben cumplir para ejecutar un servicio o también para resolver los conflictos y ambigüedades que pueden aparecer durante la ejecución del sistema debidos a la herencia múltiple. Estas reglas constituyen una parte fundamental de la definición de una clase, ya que, idealmente, el soporte completo de las clases requiere que sus operaciones sean completas y correctas.

Puesto que la semántica completa de las clases sólo existe en la mente de quien la crea, en la práctica, la completitud o exactitud de la clase será tan buena como lo sea la completitud o exactitud del código que captura su conducta. En este sentido, para ayudar a expresar mejor la conducta de las clases, los lenguajes orientados

a objetos deben proporcionar construcciones que indiquen las restricciones que prueban la exactitud o completitud de las clases. Estas restricciones se pueden implementar mediante reglas asociadas a los atributos, como los *"predicados anexionados"* que se utilizan en algunos sistemas de inteligencia artificial (IA), o las *"reglas de integridad"* de los sistemas de bases de datos. Generalmente, estas reglas se activan automáticamente al acceder o modificar un atributo con el fin de comprobar que los valores de dicho atributo no infringen alguna condición relevante para el sistema.

2.3.4. Polimorfismo

Es posible definir una clase estableciendo de forma virtual algunos de sus atributos o servicios, a la espera de que se implementen en el momento de la realización de un objeto concreto o de la derivación de nuevas clases. En este caso, se puede aprovechar otra de las características más relevantes de la OO: el polimorfismo.

Así pues, un objeto que deriva de una clase que tiene definidos servicios virtuales, haciendo uso del polimorfismo, puede enviar mensajes, con el tipo de servicio que se desea obtener, a otros objetos que sabe que se lo pueden proporcionar. El modo concreto en que se ejecutará el servicio depende únicamente del objeto receptor porque es el que incorpora el comportamiento. Por servicio virtual se entiende un método cuyo comportamiento, al ser declarado *"virtual"*, es determinado por la definición de un método con la misma cabecera en alguna de las subclases de la clase a la que pertenece.

En la práctica, el polimorfismo es la propiedad que permite enviar el mismo mensaje a objetos de diferentes clases, de forma que cada uno de ellos responde a ese mismo mensaje de modo distinto dependiendo de su implementación. El polimorfismo se aplica sólo a métodos que tienen la misma *"signatura"* (nombre, tipo y número de argumentos) pero están definidos en clases diferentes. Conviene distinguir en este sentido el polimorfismo de lo que se conoce como sobrecarga de métodos que ocurre cuando una clase tiene múltiples métodos con el mismo nombre, cada uno de ellos con una *"signatura"* distinta.

El polimorfismo es una de las características que permiten la independencia de los datos, otra de las grandes ventajas de la orientación a objetos. Gracias al polimorfismo se pueden incorporar nuevas subclases en las aplicaciones, a partir de

las cuales se podrán realizar nuevos tipos de objetos y aumentar la funcionalidad del sistema sin requerir ninguna modificación del programa. Esta característica de los lenguajes orientados a objetos permite a los programadores separar los elementos que cambian de los que no cambian, y de ésta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

El polimorfismo adquiere una relevancia especial en la práctica, ya que permite crear entornos de trabajo muy ricos en clases genéricas, específicos para determinados tipos de aplicaciones, denominados *"application frameworks"*.

Actualmente, uno de los principales objetivos de los ingenieros de software es el diseño de frameworks que permitan generar aplicaciones concretas a muy bajo coste, y que posibiliten la acumulación real del trabajo y de los conocimientos en un determinado campo de investigación o de aplicación. Así, por ejemplo, Anderson [And95] describe un framework denominado *"StatTools"* que incluye una serie de jerarquías de clases que se pueden utilizar para la implementación de simulaciones en el ámbito de la estadística. Otro ejemplo, esta vez en el ámbito de la simulación de redes neuronales desde la perspectiva conexionista, es el framework que presenta Blum [Blu92] que contiene diversos algoritmos de aprendizaje modelados e implementados bajo el paradigma de la orientación a objetos. Un último ejemplo de framework, éste desde el punto de vista de las bases de datos, son Ibatis e Hibernate, desarrollados por Apache Software Foundation y GNU LGPL respectivamente. Ambos resuelven el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional).

2.4. Ingenieria del Software

Es de sobra conocida la revolución que en las últimas décadas ha producido el uso de la informática y de los ordenadores en todos los ámbitos de la vida cotidiana. Es casi seguro que la informática juega un papel importante en el desarrollo de cualquier producto comercial en el que se pueda pensar; se habrá usado algún tipo de software para su diseño, producción, gestión, distribución, etc. Esta omnipresencia del software (SW) en cualquier proceso industrial ha dado lugar a todo un conjunto de técnicas formales tendentes a regular la manera en que el SW es especificado, generado, mantenido, actualizado y documentado. Este conjunto de

técnicas, englobadas bajo el nombre común de *"Ingeniería del Software"*, se han visto fuertemente influenciadas por la aparición del paradigma de programación orientado a objetos.

Según la definición del IEEE, citada por Lewis [Lew94], "*la Ingeniería del Software es la suma total de los programas de computadora, procedimientos, reglas, la documentación asociada y los datos que pertenecen a un sistema de cómputo*". Según el mismo autor, "*un producto de software es un producto diseñado para un usuario*", en otras palabras, se considera que "*la Ingeniería de Software es la rama de la ingeniería que aplica los principios de la ciencia de la computación y las matemáticas para lograr soluciones costo-efectivas (eficaces en coste o económicas) a los problemas de desarrollo de software*", es decir, "*permite elaborar consistentemente productos correctos, utilizables y costo-efectivos*" [Cot94].

Los productos generados son programas, documentación y datos, y la Ingeniería del SW tiene como objetivo optimizar la calidad de estos productos. Para enriquecer un poco estas definiciones se pueden enumerar algunos de los temas que trata esta especialidad, como son: métodos formales, lenguajes de modelado, reutilización del SW, reingeniería, arquitecturas cliente/servidor, técnicas orientadas a objeto, análisis, diseño, pruebas y métricas de SW, estrategias de ingeniería inversa, herramientas CASE[1], marcos de trabajo, etc.

Los requisitos para los sistemas modernos de control se pueden afrontar desde la perspectiva de la Ingeniería del SW. Concretamente, los siguientes requisitos:

1. Necesidad de adopción de tecnologías emergentes y en constante evolución.
2. Flexibilidad para adaptar los diseños existentes a nuevas necesidades impuestas por cambios de proveedores, cambios de hardware, redimensionamiento del sistema,...
3. Facilidades para el mantenimiento posterior del sistema.
4. Reducción del *"time-to-market"* (TTM).
5. Reutilización de componentes para aumentar el rendimiento y la eficiencia.

[1] CASE (Computer Aided Software Engineering) o Ingeniería del SW asistida por computador. Las herramientas CASE se emplean para asistir al usuario en una o varias de las fases involucradas en el ciclo de vida del SW.

6. Uso de software y hardware COTS[2].

7. Generación de documentación para usuarios, diseñadores y mantenedores.

En definitiva, cabe pensar en la Ingeniería del SW como un conjunto de guías generales que matizan la forma en que deben hacerse las cosas en cualquiera de los dominios. Para ilustrar esta idea, se va a desarrollar uno de los conceptos fundamentales de la Ingeniería del SW como es el del ciclo de vida.

El tiempo de concepción y desarrollo de la solución de un problema software no es puntual, sino lineal o continuado porque se compone de varias fases, o en ocasiones circular, porque hay una realimentación entre fases. En un principio, si se necesitaba resolver una situación, se concebía una solución particular para ese problema concreto y no se iba más allá. Pero los requisitos actuales hacen necesario el seguimiento de un proceso cubriendo un conjunto de fases o ciclo de vida que desemboca en el producto final. En la literatura especializada [Pre96] se puede encontrar diferentes nombres, descripciones y fases englobadas en el ciclo de vida. Haciendo una recopilación amplia, se pueden enumerar las siguientes fases como posibles para un proyecto de SW:

- Concepto y definición de objetivos generales del proyecto.
- Definición y análisis de requisitos. Descripción detallada y acordada entre cliente y desarrollador sobre qué debe hacer el producto y qué características debe poseer. El análisis de esta descripción permite valorar su viabilidad y detectar incongruencias.
- Diseño de la arquitectura. Especificación de alto nivel sobre cómo se cumplirán los requisitos mediante el uso de un determinado conjunto de módulos SW y dispositivos HW.
- Diseño detallado. Especificación en detalle del SW: estructuras de datos, interfaces entre módulos, algoritmos, etc.
- Construcción (codificación manual o generación automática). Un diseño detallado da lugar a una codificación mecánica, y por tanto, más fácil de automatizar.

[2]"Commercial Off The Shelf". Adjetivo que describe productos SW ó HW que están disponibles en el mercado y que pueden usarse en la construcción de sistemas propios.

- Integración. Se deben ensamblar las diferentes partes que constituyen el sistema.
- Pruebas. Necesarias para validar el sistema porque gracias a ellas se detectan errores y se asegura el cumplimiento de los requisitos. Deben ser tanto unitarias (relativas a una parte) como de todo el conjunto del sistema.
- Instalación. Instrucciones, ficheros, documentos y código para la configuración y puesta en marcha del sistema.
- Operación. Documentos para el uso del SW en funcionamiento.
- Mantenimiento. Cambios del SW tras ser entregado al cliente.
- Corrección de errores, adaptación al entorno (Sistema Operativo, nuevos dispositivos periféricos,...).

Si bien es muy variable el número y naturaleza de las fases dependiendo del tipo de producto SW que se esté generando, también depende del modelo de gestión de proceso que se siga. Existe un amplio abanico de modelos para representar el proceso SW:

- Modelo lineal secuencial o en cascada [Roy70]. El término *"cascada"* denota el concepto idealizado de que las fases ocurren una detrás de otra, con límites entre fases claramente definidos. Fases típicas: análisis, diseño, código y pruebas.
- Modelo de construcción de prototipos [Bro95]. Útil cuando el cliente define unos objetivos generales pero no identifica los requisitos de forma detallada. Las fases se estructuran en forma de círculo: peticiones del cliente, construcción de un prototipo, pruebas y vuelta a empezar.
- Modelo RAD [KH94]. Desarrollo rápido gracias a una especificación de alto nivel con técnicas de cuarta generación (T4G[3]) y una construcción basada en componentes que enfatiza la reutilización. Este enfoque comprende las siguientes fases: modelado del flujo de información, de las características y

[3]Las técnicas de cuarta generación son un conjunto muy diverso de métodos y herramientas que tienen por objetivo el facilitar el desarrollo del software, partiendo de la especificación de alto nivel de algunas características del mismo y generando de forma automática más tarde, por medio de una herramienta de cuarta generación, el código fuente.

relaciones entre datos y del manejo de estos datos (modelado de gestión, datos y proceso, respectivamente). A partir de estos modelos se genera la aplicación reutilizando, en la medida de lo posible, componentes existentes. Finalmente, se prueban con especial atención los componentes nuevos y las interfaces.

- Modelos evolutivos. Son iterativos y permiten generar versiones cada vez más completas del producto.

- Modelo incremental [McD93]. Basado en el modelo en cascada, entrega el SW en pequeñas partes llamadas incrementos que son producidos en secuencias lineales con las fases del modelo en cascada.

- Modelo espiral [Boe88]. Desarrollo rápido de versiones incrementales del SW. Conjuga la naturaleza iterativa de construcción de prototipos con los aspectos sistemáticos del modelo en cascada. En lugar de fases, emplea las siguientes regiones de tareas: comunicación con el cliente, planificación, análisis de riesgos, ingeniería, construcción y evaluación.

- Otro campo de interés [Dye92] es el uso de lenguajes formales de especificación como *"Z"* [Spi92] para lograr una especificación matemática del SW. Estos métodos permiten especificar, desarrollar y verificar un sistema aplicando una notación rigurosa y matemática. La ambigüedad, lo incompleto y lo inconsistente se corrige más fácilmente porque aplica un análisis matemático en lugar de una revisión a propósito para el caso. Este enfoque tiene más partidarios entre los desarrolladores de SW de alta seguridad.

Para cualquiera de los modelos es cada vez más habitual el desarrollo basado en componentes con objeto de beneficiarse de las ventajas de la reutilización. A través del paradigma de orientación a objetos se pueden encapsular los datos y los algoritmos que los manejan en forma de clases reutilizables que se organizan en forma de repositorios para su reutilización. Para el desarrollo de un nuevo proyecto se identifican unas clases candidatas, se comprueba si existen y se crean si no es así.

2.4.1. Sistemas de modelado

Según Booch [BJ99], se puede decir que el modelado es una parte central de todas las actividades que conducen a la programación de buen software y por ende, de una buena ingeniería. Se construyen modelos para comunicar la estructura deseada y el comportamiento del sistema. Estos modelos permiten, además, visualizar y controlar la arquitectura del sistema. Se construyen modelos también con la finalidad de comprender mejor el sistema que se esta desarrollando, muchas veces descubriendo oportunidades para la simplificación y la reutilización. En definitiva, se construyen modelos para controlar el riesgo.

Booch [BJ99] plantea un símil extraído del contexto de la construcción de inmuebles para convencer de la importancia de modelar durante el proceso de desarrollo de software. En este sentido, afirma que, curiosamente, una gran cantidad de empresas de desarrollo de software comienzan queriendo construir *"rascacielos"*, pero enfocan el problema como si estuvieran enfrentándose a la *"caseta de un perro"*. Si realmente se quiere construir el software equivalente a una casa o a un rascacielos, el problema es algo más que una cuestión de escribir grandes cantidades de software. De hecho, el software de calidad se centra en optimizar al máximo el código, imaginar cómo escribir menos software y conseguir el mejor rendimiento sin que disminuya la eficacia. Ésto convierte al desarrollo de software de calidad en una cuestión de arquitectura, proceso y herramientas, es decir, en una cuestión de modelado.

Un modelo es una simplificación de la realidad que se construye para comprender mejor el sistema que se está desarrollando. De hecho, se construyen modelos de sistemas complejos porque no es posible comprender el sistema en su totalidad.

A través del modelado se consiguen cuatro objetivos:

- Ayudar a visualizar cómo es o debería ser un sistema.
- Especificar la estructura o el comportamiento de un sistema.
- Proporcionar plantillas que guíen la construcción de un sistema.
- Documentar las decisiones adoptadas.

Por otro lado, existen una serie de principios básicos de modelado [BJ99].

En primer lugar, es importante tener en cuenta que la elección de los modelos a crear tiene una profunda influencia sobre cómo se acomete un problema y cómo se da forma a una solución. En este sentido, los modelos adecuados pueden arrojar mucha luz sobre problemas de desarrollo muy complicados, ofreciendo una comprensión inalcanzable por otras vías; en cambio, los modelos erróneos desorientarán, haciendo que uno se centre en cuestiones irrelevantes. En el software, los modelos elegidos pueden afectar mucho a la visión del mundo que se tiene. Si se construye un sistema con la mirada de un analista que se basa en una perspectiva estructurada, probablemente se obtendrán modelos centrados en los algoritmos, con los datos fluyendo de proceso en proceso. Si se construye, en cambio, con la mirada de un desarrollador orientado a objetos, se obtendrá un sistema cuya arquitectura se centra en una gran cantidad de clases y los patrones de interacción que gobiernan cómo trabajan juntas esas clases. Cada visión del mundo conduce a un tipo de sistema diferente, con diferentes costes y beneficios, aunque la experiencia sugiere que la visión orientada a objetos es superior al proporcionar arquitecturas flexibles.

Un segundo principio básico del modelado dice que todo modelo puede ser expresado a diferentes niveles de precisión. Un ingeniero o un analísta se centrará en el qué; un desarrollador se centrará en el cómo. En cualquier caso, los mejores tipos de modelos son aquéllos que permiten elegir el grado de detalle, dependiendo de quién está viendo el sistema y por qué necesita verlo.

Un tercer principio establece que los mejores modelos están ligados a la realidad. En la Ingeniería de Software, el talón de Aquiles de las técnicas de análisis estructurado es la existencia de una desconexión básica entre el modelo de análisis y el modelo de diseño del sistema. No poder salvar este abismo hace que el sistema concebido y el sistema construido diverjan con el paso del tiempo. En los sistemas orientados a objetos, en cambio, es posible conectar todas las vistas casi independientes de un sistema en un todo semántico.

Finalmente, un cuarto principio establece que un único modelo no es suficiente de manera que cualquier sistema se aborda mejor a través de un pequeño conjunto de modelos casi independientes. La expresión *"casi independientes"* en este contexto significa tener modelos que se puedan construir y estudiar separadamente, pero aún así relacionados. Por poner un ejemplo, cuando se está construyendo un edificio se necesitan distintos modelos de planos: planos de planta, alzados, planos de electricidad, planos de calefacción y planos de cañerías; por supuesto, se pue-

den estudiar los planos eléctricos de forma aislada, pero también se puede ver su correspondencia con los planos de planta, incluso su interacción con los recorridos de las tuberías en el plano de fontanería.

Este último principio es igualmente cierto para los sistemas de software orientados a objetos. Para comprender la arquitectura de tales sistemas se necesitan vistas complementarias y entrelazadas: una vista de casos de uso (que muestre los requisitos del sistema), una vista de diseño (que capture el vocabulario del espacio del problema y del espacio de la solución), una vista de procesos (que modele la distribución de los procesos e hilos del sistema), una vista de implementación (que se ocupe de la realización física del sistema) y una vista de despliegue (que se centre en cuestiones de la ingeniería del sistema) (ver figura 2.6). Cada una de estas vistas puede tener aspectos tanto estructurales como de comportamiento. En conjunto, estas vistas representan los planos del software.

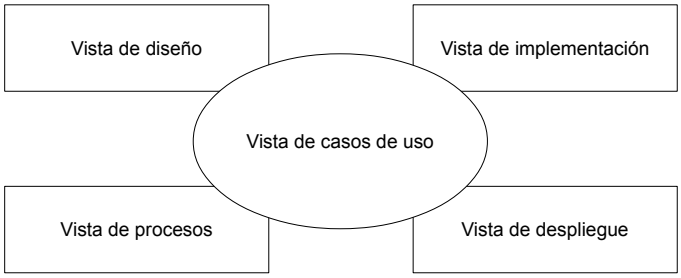

Figura 2.6: Tipos de vista en la Ingenieria del Software

Según la naturaleza del sistema, algunos modelos pueden ser más importantes que otros. Por ejemplo, en sistemas con grandes cantidades de datos dominarán los modelos centrados en las vistas de diseño estáticas. En sistemas con uso intensivo de interfaces gráficas de usuario (GUI) las vistas de casos de uso estáticas y dinámicas son bastante importantes. En los sistemas de tiempo real muy exigentes (por ejemplo, un sistema de control de tráfico) las vistas de procesos dinámicos tienden a ser más importantes.

2.4.2. Modelado orientado a objetos

Así como los planos de un arquitecto sirven para hacer un edificio, en el SW no vale sólo con una visión estática del problema debido a que los procesos, a lo

largo de su ciclo de vida, cambian su estado, por lo que se hace necesario tener en cuenta una visión dinámica. De ahí la necesidad de la utilización de modelos que proporcionen a los ingenieros unas especificaciones completas antes de la fase de implementación.

En el proceso de modelado orientado a objetos se distinguien las tres fases tradicionales de elaboración de un sistema informático [Pre96]:

1. Análisis: Se ocupa del qué, de entender el dominio del problema.
2. Diseño: Responde al cómo y se centra en el espacio de la solución.
3. Implementación: Fase en la que se adapta la solución a un entorno de programación específico.

Básicamente, los productos del análisis orientado a objetos sirven como modelos de los que se puede partir para un diseño orientado a objetos. Estos productos del diseño pueden utilizarse entonces como anteproyectos para la implementación completa de un sistema utilizando métodos de programación orientada a objetos.

En este contexto de la orientación a objetos, la transición entre los modelos de análisis y diseño se vuelve más fácil e intuitiva, puesto que los objetos del dominio del problema tienen una correspondencia con los objetos del dominio de la solución. Como señalan McGregor y Korson [MK90], durante el análisis se abordan los objetos del dominio del problema, mientras que en el diseño se especifican los objetos adicionales necesarios para obtener la solución implementada en el ordenador. Precisament,e la no consideración de ambas fases ha conducido al desastre en muchos proyectos de software.

Se han propuesto un gran número de métodos para el análisis y diseño orientado a objetos como el OOSD de Wasserman, Pircher y Muller [WM99], el método de Coad y Yourdon [CY90, CY91], el OODLE de Shlaer y Mellor [SM88], el método de Desfray [Des92], el método de Booch [Boo91, Boo94], el OMT de Rumbaugh, Blaha, Premerlani, Eddy y Lorensen [RL90], el OOSE de Jacobson, Christerson, Jonson y Övergaard [JO92], el método de Embley, Kurtz y Woodfield [EW92], el HOOD de Robinson [Rob92], el método Petch de Martín y Odell [MO94], el SOMA de Graham [Gra00], el método Fusion de Coleman [Col93], etc.

Cabe destacar el método de Booch [Boo91, Boo94], el OMT de Rumbaugh [RL90] y el OOSE de Jacobson [JO92]. Booch, Rumbaugh y Jacobson unificaron la notación

de sus respectivos métodos en 1996, dando lugar al lenguaje de modelado conocido como UML [BJ98] que sufrió un proceso de estandarización en 1997 por el Object Management Group (OMG), cuyas especificaciones han sido refrendadas por la mayoría de las empresas e instituciones desarrolladoras de software. El OMG es un consorcio a nivel internacional que integra a los principales representantes de la industria de la tecnología de información en la orientación a objetos y que tiene como objetivo central la promoción, fortalecimiento e impulso de la industria orientada a objetos. Para ello, propone y adopta por consenso especificaciones que afectan a la tecnología orientada a objetos, como es el caso de la especificación UML, que junto con el proceso unificado de desarrollo de software ha consolidado esta tecnología.

La motivación de estos tres autores por crear un lenguaje unificado de modelado (UML) se fundamentó en tres razones [BJ98]:

1. Cada uno de los métodos ya está evolucionando independientemente hacia los otros dos. Tenía sentido hacer continuar esa evolución de forma conjunta en vez de hacerlo por separado, eliminando la posibilidad de cualquier diferencia gratuita e innecesaria que confundiera aún más a los usuarios.

2. La unificación de estos métodos pretendía proporcionar cierta estabilidad al mercado orientado a objetos, permitiendo que los proyectos se pusieran de acuerdo en un lenguaje de modelado maduro y permitiendo a los constructores de herramientas que se centraran en proporcionar más características útiles.

3. Los autores esperaban que su colaboración introduciría mejoras en los tres métodos anteriores, ayudándose entre ellos a capturar lecciones aprendidas y a cubrir problemas que ninguno de sus métodos anteriores había manejado bien anteriormente.

El hecho es que cada uno de éstos era un método completo, aunque todos tenían sus puntos fuertes y débiles. En resumen, el método de Booch era particularmente expresivo durante las fases de diseño y construcción de los proyectos, OOSE proporcionaba un soporte excelente para los casos de uso como forma de dirigir la captura de requisitos, el análisis y diseño de alto nivel, y OMT-2 era principalmente útil para el análisis y para los sistemas de información con gran cantidad de datos.

Con posterioridad al desarrollo de UML, los tres autores presentaron el *"Proceso Unificado de Desarrollo de Software"* [JR99], que también se ha denominado de forma abreviada *"Proceso Unificado"*, como un estándar que cubre todo el proceso ligado al ciclo de vida de desarrollo de software. De hecho, el proceso unificado integra una amplia variedad de aportaciones, no sólo las proporcionadas por los procesos involucrados en los propios métodos de sus autores. La notación UML se integra dentro de este proceso puesto que permite modelar de forma gráfica los diferentes elementos involucrados en el desarrollo de software.

Actualmente, UML posee una gran relevancia como lenguaje de modelado en el desarrollo de software OO gracias a su nivel de estandarización, aspecto éste que permite una mayor y mejor comunicación entre desarrolladores de software.

A grandes rasgos y salvando las diferencias que existen entre los distintos métodos a los que se ha hecho referencia al principio de este apartado, sus etapas se pueden sintetizar de la siguiente manera:

1. Se identifican las clases semánticas (con sus atributos y servicios), ésto es, las que reflejan el espacio del problema. Para ello, la mayor parte de los métodos proponen un enfoque lingüístico mediante el análisis de las oraciones contenidas en la especificación inicial del modelo [Abb83]:

   a) Los sustantivos se consideran clases candidatas.

   b) Los verbos se consideran posibles servicios o métodos de las clases.

   Los adjetivos a menudo expresan atributos de la clase, etc.
   Antes de proceder al análisis lingüístico, es recomendable extraer las frases nominales cambiando los plurales por singulares, unificando los términos, etc. [PR94].
   También se propone identificar las clases partiendo de los guiones [JO92], de las listas de comprobación, de los análisis de dominio, o utilizando patrones [Coa92], layers [Gra95], etc.

2. Se establecen las interrelaciones entre las clases, sobre todo, estudiando la generalización (herencia) y la agregación entre las mismas.

3. Se añaden otras clases, como las de interfaz de usuario, las que incorporan los mecanismos de control de la aplicación y las clases base, que son independientes de la aplicación y que generalmente interactúan con el sistema operativo propietario.

4. Se implementan las clases.

En cuanto al paso de la fase de análisis a la de diseño, se pueden distinguir dos aproximaciones:

- De elaboración: el modelo de análisis se va completando con información de diseño hasta que se encuentre listo para ser codificado.
- De transformación: se aplican reglas para transformar el análisis en diseño.

Es importante resaltar el hecho de que las fronteras entre análisis y diseño son difusas, aunque el objetivo principal de ambos es bastante diferente. En el análisis orientado a objetos se persigue modelar el mundo a partir de las clases y objetos que forman el vocabulario del dominio del problema, y en el diseño orientado a objetos se establecen las abstracciones y mecanismos que proporcionan el comportamiento que este modelo requiere. La existencia de estas fronteras difusas entre la fase de análisis y la de diseño es un aspecto que deriva de la forma en que se afronta el desarrollo de software orientado a objetos. La aparición de los lenguajes de programación orientados a objetos ha hecho que se tienda a cambiar el paradigma de desarrollo empleado hasta el momento (el llamado ciclo de vida clásico) en la creación de programas.

En los años 70 y 80 aparecieron métodos de desarrollo estructurado asociados al ciclo de vida clásico. Sin embargo, los métodos de programación estructurada no se acoplan bien al paradigma orientado a objetos. Por un lado, se centran en la descomposición algorítmica, es decir, en el proceso de transformación de los datos y no en los datos mismos. Además, tampoco están pensados para manejar el proceso de encapsulado de la información ni para tratar con jerarquías.

En el ciclo de vida clásico (ver figura 2.7), en cada etapa del proceso se generan unos nuevos productos que son especificaciones de lo que se ha de hacer en la etapa siguiente. Este paradigma se ha sustituido por otros, como el ciclo de vida para orientación a objetos [AA92], más iterativos, que permiten y potencian que en diversas etapas del proceso se vaya hacia delante y hacia atrás. En palabras de Gilb [Gil88] se trata de *"un poco de análisis, un poco de diseño, un poco de programación, y vuelta a empezar"*.

El diagrama del ciclo de vida orientado a objetos (ver figura 2.8) muestra este proceso iterativo que se mueve hacia delante y hacia atrás. En ese sentido, en

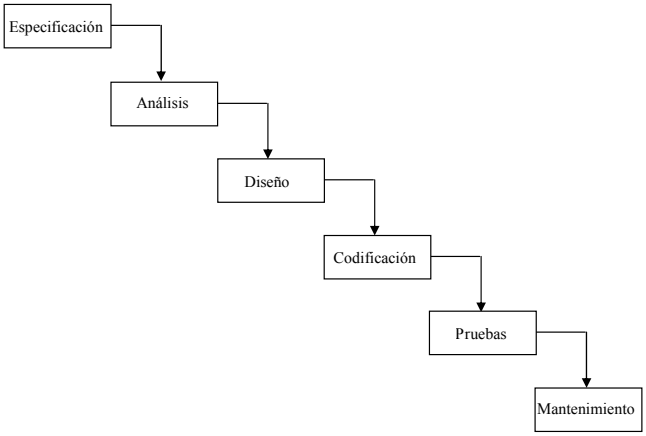

Figura 2.7: Ciclo de vida clásico en cascada

el diagrama en cuestión los diversos modelos que forman el modelo se van construyendo o rehaciendo a medida que el proceso transcurre desde el análisis de requerimientos hasta la implementación y uso del sistema.

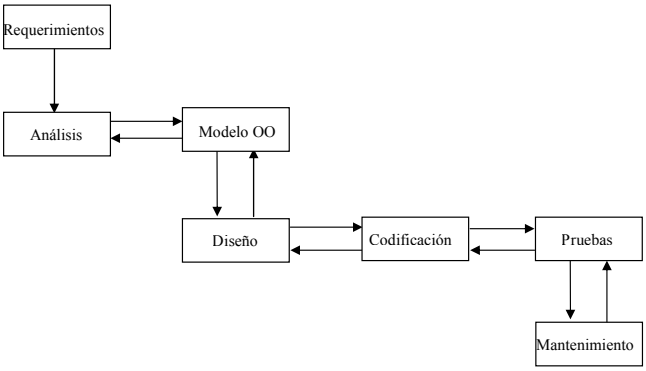

Figura 2.8: Ciclo de vida orientado a objetos

En consecuencia, de forma paralela a la adopción del paradigma orientado a objetos han ido apareciendo métodos de análisis y diseño orientados a objetos pensados para permitir el desarrollo de software siguiendo este paradigma, bastantes de los cuales han sido mencionados al comienzo de este apartado. Cada uno de estos métodos dispone de un conjunto definido de entidades, símbolos gráficos, diagramas, fichas, reglas y procedimientos de actuación. El objetivo último de todos ellos es la creación de sistemas software de calidad y, como es natural, sus Métodologías

no están aisladas ni son totalmente diferentes las unas de las otras, sino más bien al contrario.

2.4.3. Lenguaje Unificado de Modelado

El Lenguaje Unificado de Modelado o UML (Unified Modeling Language), es el lenguaje gráfico de modelado de sistemas software más conocido y utilizado en la actualidad. Provee de una sintaxis para describir los elementos importantes (llamadas artefactos en UML) de un sistema de software.

UML, en cuanto a notación de modelado, es el sucesor de la oleada de métodos de análisis y diseño orientados a objetos que aparecieron a principios de los 90. De hecho, unifica la notación de los métodos de Booch, Rumbaugh (OMT) y Jacobson (OOSE), y además, es considerado un estándar OMG [OMG03].

Sin embargo, UML no es un método. Un método consiste, en principio, en un lenguaje de modelado y un proceso. UML aporta la parte de lenguaje de modelado, pero es necesario un proceso que utilice dicho lenguaje. El Proceso Unificado de Desarrollo de Software, creado por los mismos autores de UML, aporta la parte de proceso en el que se emplea este lenguaje de modelado.

Booch [BJ98] afirma que el 80 por 100 de la mayoría de problemas puede modelarse con UML. Los elementos estructurales básicos, tales como clases, atributos, operaciones, casos de uso, componentes y paquetes, junto a las relaciones estructurales básicas, tales como dependencia, generalización y asociación, son suficientes para crear modelos estáticos para muchos tipos de dominios de problemas. Si a esta lista se añaden los elementos de comportamientos básicos, tales como las máquinas de estados simples y las interacciones, se pueden modelar muchos aspectos útiles de la dinámica de un sistema. Sólo hará falta utilizar las características más avanzadas de UML cuando se empiece a modelar aspectos relacionados con situaciones más complejas, como cuando se debe tratar con mecanismos de concurrencia y de distribución.

El vocabulario de UML incluye tres grandes bloques de construcción [BJ98, OMG03]: elementos, relaciones y diagramas. Los elementos son abstracciones fundamentales de un modelo; las relaciones ligan estos elementos entre sí; los diagramas agrupan colecciones de elementos.

En UML hay cuatro tipos de elementos, los cuales constituyen los bloques básicos de construcción orientados a objetos de este lenguaje de modelado [BJ98]. Por un lado, los elementos estructurales representan las partes estáticas de un modelo, los *"materiales de construcción"* propios del lenguaje. En total hay siete tipos de elementos estructurales: clases, interfaces, colaboraciones, casos de uso, clases activas, componentes y nodos. Los elementos de comportamiento son las partes dinámicas de los modelos UML, son los verbos de un modelo, y representan comportamiento en el tiempo y el espacio. Hay dos tipos principales de elementos de comportamiento: interacciones y máquinas de estados. Los elementos de agrupación son las partes organizativas de los modelos UML. El elemento de agrupación principal son los paquetes. Por último, existen elementos de anotación que representan las partes explicativas de los modelos. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento del modelo. Hay un tipo principal de elemento de anotación llamado nota.

2.5. Ingenieria de Control

La Ingeniería de Control es una especialidad relativamente joven en cuanto a su formulación formal pero muy antigua en cuanto a su uso. Desde hace mucho tiempo, el ser humano se ha valido de *"ingenios"* para conseguir que las cosas se comportaran de una forma determinada, que respondieran de una forma prefijada. Problemas tan diversos como el control de la posición ó velocidad de un móvil, el de la cantidad de líquido en un recipiente o el de la temperatura de un habitáculo, han sido resueltos mediante ingeniosos mecanismos a lo largo de los tiempos. Pero la verdadera historia de la Ingeniería de Control se empieza a escribir cuando se abstraen conceptos comunes de toda esta amplia casuística. Concretamente, se puede decir que el concepto de realimentación es el primero en superar los límites de lo particular para aplicarse de forma generalizada a la solución de muchos problemas de control desde principios del siglo XX [Ben93]. De hecho, algunos autores como Richard Murray [MS03] definen la Ingeniería de Control como el uso de algoritmos y realimentación en Ingeniería de Sistemas. Una definición para sistema de control realimentado es la de un dispositivo (o conjunto de ellos) que usa la medida de una magnitud para modificar el comportamiento de un sistema a través de un cálculo y una actuación.

De todos modos, no es hasta mediados del siglo XX cuando la disciplina de Ingenie-

ría de Control empieza a tomar forma y se articula como un conjunto de técnicas aplicables a un amplio rango de problemas [Ben93]. Ámbitos tan diversos como el del control de procesos, la amplificación electrónica con realimentación negativa y los servomecanismos, habían dado lugar a soluciones de control particulares que respondían a problemas diferentes y específicos. Se identifican semejanzas en todas estas soluciones y estos puntos en común impulsan el desarrollo de abstracciones que permiten utilizar el mismo conjunto de técnicas de análisis y diseño para la creación de sistemas de control en ámbitos diversos.

Así, durante la Segunda Guerra Mundial y los años posteriores, evoluciona el diseño de sistemas de control en el dominio de la frecuencia (usando relaciones de amplitud y fase entre las entradas y salidas del sistema). El conocimiento adquirido durante este periodo toma el nombre de *"Teoría Clásica de Control"*, como contraposición a la *"Teoría Moderna de Control"* que se empieza a designar en los años 60 a las técnicas basadas en el dominio temporal (modelado de los sistemas mediante ecuaciones diferenciales lineales). Sin embargo, acaban haciéndose patentes las limitaciones del uso de modelos matemáticos deterministas para el control de sistemas reales en los que aparecen no linealidades así como entradas, perturbaciones y ruidos no deterministas. Como respuesta a estas necesidades nacen las estrategias de control no lineal y los tratamientos estocásticos.

A partir de la década de los 60, con el desarrollo de la tecnología de computadores, se empieza a plantear la posibilidad de aplicar dichas máquinas al control de procesos y al estudio de las implicaciones para llevarlo a cabo. Nacen así los primeros *"Sistemas de Control Digital Directo"* (DDC) formados por un computador y sus interfaces al proceso en el centro de una red de cables en forma de estrella que los conectaban a los sensores y actuadores del proceso. Pero el elevado precio de los computadores y del cableado encarecía enormemente estos sistemas. La irrupción del computador provocó profundos cambios en la Ingeniería de Control, como son la necesidad de adaptar la naturaleza analógica y continua de las señales del entorno a la naturaleza digital y discreta del computador (uso de convertidores A/D, D/A y temporizadores). Estos cambios impulsaron el desarrollo de la *"Teoría de Control en tiempo Discreto"*.

La generalización y el abaratamiento de los microprocesadores acabó teniendo impacto en la propia arquitectura de los sistemas, ya que los *"Sistemas de Control Distribuido"* (DCS) se hacen económicamente viables y suministran una capacidad de procesamiento extremadamente alta, respuestas rápidas y tolerancia a fallos.

En los últimos años, se ha diversificado bastante el espectro de soluciones y se emplean diferentes estrategias que permiten atacar los problemas de control desde varios puntos de vista: control óptimo, adaptativo, fuzzy, redes neuronales, robusto, predictivo, multivariable, ect. De manera que una de las labores del ingeniero, es la de la elección de la técnica o conjunto de técnicas más adecuadas (por precisión, precio, sencillez, etc) para la resolución de un caso concreto.

Actualmente, el desarrollo de sistemas de control se ha convertido en una actividad complicada que cuenta con un gran número de requisitos adicionales a las clásicas especificaciones de diseño en términos de estabilidad relativa, precisión, respuesta transitoria o sensibilidad a variaciones en los parámetros [Kuo96, Oga03]. Sin perder de vista estas especificaciones tradicionales, han surgido otros requisitos a los que también hay que hacer frente [MS03]:

1. Descentralización de los algoritmos de control.
2. Garantía del cumplimiento de requisitos temporales
3. Necesidad de adopción de tecnologías emergentes y en constante evolución.
4. Flexibilidad para adaptar los diseños existentes a nuevas necesidades impuestas por cambios de proveedores, cambios de hardware, redimensionamiento del sistema,...
5. Facilidades para el mantenimiento posterior del sistema.
6. Reducción del *"time-to-market"*.
7. Reutilización de componentes para aumentar el rendimiento y la eficiencia.
8. Uso de Software y Hardware COTS.
9. Generación de documentación para usuarios, diseñadores y mantenedores.

Las técnicas tradicionales no son suficientes para cumplir con todos estos requisitos.

A pesar de que a lo largo de la historia del control se ha ido evolucionando con abstracciones sucesivas que permitían abordar cada vez mejor problemas de creciente complejidad, se ha llegado a un punto en el que son muchas las decisiones a tomar y los parámetros a controlar. Hoy en día no es suficiente con el conocimiento de una sola persona, ni existe una única abstracción que maneje la complejidad

de cualquier problema. La construcción de un sistema de control moderno abarca cuestiones que van más allá del ámbito de la teoría de control y se requiere de una aproximación multidisciplinar para abordar esta tarea.

Se trata de construir por tanto, una aproximación que permita desarrollar soluciones con un uso determinista de los recursos, basada en las nuevas tecnologías, flexible, fácil de mantener, de rápido desarrollo, con alto grado de reutilización de soluciones previas, construidas con herramientas y componentes comerciales y con documentación que explique la solución a diferentes niveles. En definitiva, hay *"otros"* requisitos no funcionales que deben ser satisfechos y por tanto *"otros"* especialistas diferentes a los de control se ven involucrados.

En la actualidad, los computadores se han impuesto como la solución optima para la automatización de los sistemas de control. En el momento que este tipo de equipos son usados para implementar el control de los sistemas, toda la teoría de control se materializa como un software. En este caso, la diferencia que marca el buen o mal funcionamiento del sistema es el diseño y desarrollo de dicho software de control.

2.5.1. Sistemas de modelado en la automatización de procesos

El uso de modelos en la automatización, al igual que en la Ingeniería del Software, resulta fundamental para una perfecta definición de un sistema de control, ya que para una correcta comprensión de un proyecto se hace necesario acudir a modelos que permitan trabajar con un nivel de abstracción menor. Esta aseveración se sostiene en las evaluaciones que realizaron Germino y Pating [GW05, Pat08] de diversos proyectos industriales, sus expertos y experimentos de laboratorio sobre los beneficios en términos de calidad y reducción del tiempo de desarrollo que se produce cauando se usan modelos en la automatización de procesos; o los trabajos de Katzke y Friedrich [KVH04, Fri09] en los que se definieron una serie de tareas para evaluar la capacidad de comprensión que los usuarios tienen de los sistemas cuando éstos usan modelos, incluyendo en el estudio diversos grupos de trabajadores como técnicos de operación, trabajadores no cualificados o ingenieros de mantenimiento.

En esta sección, se hace referencia a las RdP (Redes de Petri), GRAFET ((Graphe Fonctionnel de Commande, Etapes et Transitions) y STATECHARTS, como

lenguajes de modelado usados en la automatización de procesos.

REDES PETRI (RdP)

Carl Adam Petri [Pet62], entre los años 1960-1962, formula un modelo matemático de propósito general para la descripción de las relaciones existentes entre condiciones y eventos que acabaron recibiendo el nombre de RdP. Las RdP permiten modelar cualquier sistema de eventos discretos que exhiba características de paralelismo, concurrencia, sincronización y compartición de recursos. Además, también permiten hacer una validación cualitativa del funcionamiento de un proceso, pues el modelo puede ser empleado para explicar cómo funciona el sistema modelado.

Las RdP se desarrollan en torno al concepto de nodo. Comprenden dos tipos de nodos: lugares y transiciones. Deben tener un número entero, positivo y finito de lugares y transiciones conectados por medio de arcos orientados de forma alternativa. Cada lugar debe tener cero ó un número entero positivo de marcas.

Sobre esta definición de los componentes de una RdP se deben aplicar una serie de reglas de evolución que rigen la forma y las condiciones que permiten que las marcas se desplacen de unos lugares a otros dentro de la red. Según sean estas normas aparecen diversos tipos de RdP que se pueden clasificar de la siguiente forma según [AH92]:

- RdP no autónomas. Permiten describir el funcionamiento de un sistema cuya evolución se ve condicionada por eventos externos o por el tiempo. Básicamente estas se subdividen en:
  - RdP sincronizadas. Sirven para modelar fundamentalmente sistemas de tiempo real.
  - RdP de tiempo o estocásticas. Su campo de aplicación es el modelado de sistemas de producción y de proceso de datos. Su característica principal es que tienen en cuenta la variable tiempo.
- RdP autónomas (ver figura 2.9). En estas RdP los momentos de activación de las condiciones de transición son desconocidos o no están especificados. Por ejemplo, una RdP que represente el modelo de evolución de las estaciones

climatológicas como la mostrada en la figura 2.9, es una RdP autónoma ya que en el modelo no se indica nada sobre cuándo se va a producir la activación de las transiciones que permitirán cambiar de estación. En esta categoría se puede enmarcar las RdP autónomas continuas las cuales permiten modelar y validar cualitativamente procesos continuos.

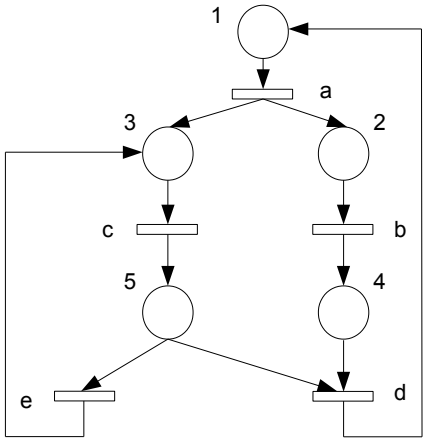

Figura 2.9: RdP autónoma

- RdP híbridas. Estas RdP pueden ser empleadas para el modelado de sistemas híbridos, esto es, con una componente de eventos discretos y una componente de continuos.

En definitiva, las RdP definen un lenguaje de modelado complejo cuyo potencial hace que sean aplicadas a muchos y variados campos científicos y sobre el que se están publicando constantemente nuevos avances y modificaciones que descubren nuevos campos de aplicación. No es de extrañar pues que sobre las bases de las RdP se definiese GRAFCET.

GRAFCET

Debido al aumento de la complejidad de los sistemas a automatizar, se hace cada vez más difícil analizar y modelar estos sistemas así como desarrollar la lógica de control que los gobierne libre de errores y fácil de comprender para el personal encargado de su posterior mantenimiento [Mor02].

Por este motivo, en 1975 el grupo *"Logic Systems Group"* en el seno de la AFCET (Association Française pour la Cybernétique Economique et Technique) decide formar una comisión de estudio denominada *"Normalización de la Representación de las Especificaciones de los Controladores Lógicos"*, encargada de unificar y racionalizar los procedimientos seguidos hasta el momento por distintas compañías y centros de investigación franceses para el análisis, modelado y elaboración de la lógica de control para controladores lógicos programables (PLC). Esta comisión produjo un informe final en Agosto de 1977 [AFC77] en el que se presentaba GRAFCET como un lenguaje de modelado de máquinas secuenciales en el sentido puramente matemático del término, independientemente de la tecnología y de la implementación final del mismo.

Posteriormente, la ADEPA (Agence Nationale pour le Développment de la Production Automatisée) tomó la definición de GRAFCET propuesta por la AFCET, la normalizó y le dio una nueva forma de representación gráfica, más alejada de las RdP y la propuso como estándar francés. En 1988 se convirtió en estándar internacional propuesto por la Comisión Electrotécnica Internacional (IEC) bajo el nombre *"Preparation of Function Charts for Control Systems"* [IEC88].

Finalmente, GRAFCET ha servido de base para el desarrollo de un nuevo lenguaje para la descripción y programación de automatismos secuenciales denominado *"Sequential Function Chart"* (SFC) publicado en 1993 como parte del estándar internacional IEC 61131-3 promovido por IEC [IEC93].

Para la elaboración de GRAFCET, la AFCET pidió a los miembros integrantes de la comisión que aportasen un documento en el que describiesen el modo o procedimiento que seguían en sus empresas o centros de investigación para describir el comportamiento de los automatismos lógicos. El resultado fueron dieciséis documentos, de los cuales quince podían ser agrupados en tres categorías fundamentalmente: diagramas de flujo, sistemas de representación de automatismos lógicos con evolución simultánea (incluidas RdP) y diagramas de estado. La decisión final fue desarrollar un nuevo modelo basado en gran medida en las RdP pero diseñado para representar de manera concisa y clara las relaciones existentes entre las entradas y las salidas del automatismo lógico.

En la figura 2.10 se muestra la diferencia de interpretación entre un GRAFCET según AFCET y ADEPA.

Además de la representación gráfica, GRAFCET toma prestado de las RdP un se-

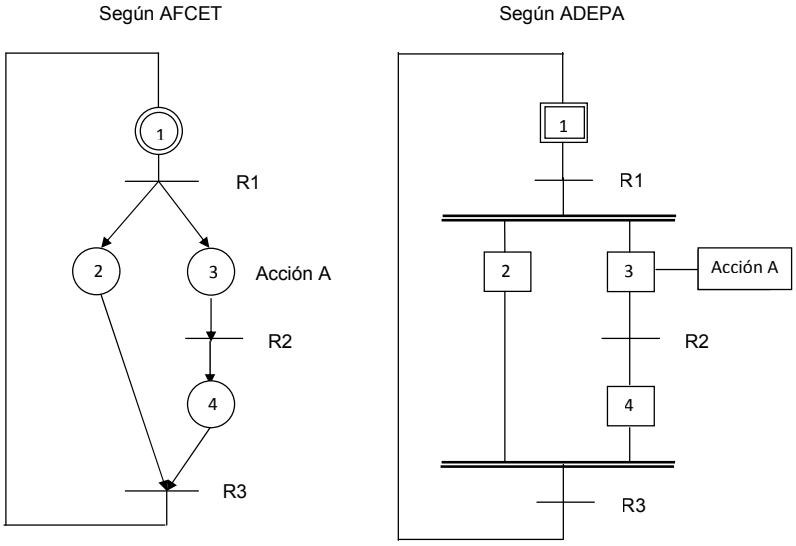

Figura 2.10: Representaciones de GRAFCET según AFCET y ADEPA

gundo concepto que es la organización del lenguaje en torno a dos tipos de nodos: etapas, transiciones y líneas de evolución orientadas que los conectan de forma alternada. Además de estos elementos de representación, GRAFCET define también el conjunto de reglas de evolución e interpretación del gráfico en las RdP interpretadas. Sin embargo, es en este conjunto de reglas donde aparecen las mayores diferencias que se pueden agrupar en tres categorías, a saber:

- El marcado de las etapas en GRAFCET es booleano, es decir, una etapa sólo puede estar marcada o no marcada, mientras que en las RdP interpretadas el marcado de los lugares es analógico, o sea, un lugar puede estar marcado cero, una, dos o más veces.

- Mientras que GRAFCET define que, todas las transiciones que en un instante dado son simultáneamente franqueables son simultáneamente franqueadas, en las RdP interpretadas no se garantiza lo mismo. Para poder asegurar ésto, es necesario introducir restricciones adicionales sobre las RdP interpretadas que desembocarán en una nueva categoría de RdP.

- Las condiciones usadas en un GRAFCET, al contrario de lo que ocurre en las RdP, pueden depender del estado de marcado de una etapa.

Tanto GRAFCET como las RdP interpretadas representan una máquina de estados secuenciales en el sentido matemático del término. Por definición, se dice que dos máquinas de estados secuenciales son equivalentes si para cada secuencia de entrada ambas producen la misma secuencia de salida.

Para superar las diferencias existentes entre GRAFCET y las RdP interpretadas, se introduce la restricción de seguridad sobre las RdP interpretadas, pasando a denominarse RdP de control interpretadas. Las reglas para interpretar estas redes son las mismas que las que rigen las RdP interpretadas y junto con la restricción de seguridad impuesta que asegura su determinismo, se puede decir que una RdP de control interpretada es equivalente a un GRAFCET.

STATECHARTS

David Harel [Har87] publica en 1987 la definición de un nuevo lenguaje visual para el modelado de sistemas complejos denominado STATECHARTS. Este nuevo lenguaje (ver figura 2.11) permite representar el comportamiento de un sistema a lo largo del tiempo, lo que incluye la dinámica de sus funcionalidades, su control y su relación con respecto al tiempo, los estados o modos en que se puede encontrar el sistema y las condiciones que hacen que evolucione de unos a otros. Provee de mecanismos para representar concurrencia, sincronización y causalidad.

Una forma natural de representar el comportamiento de un sistema es mediante máquinas de estados finitos, las cuales se pueden expresar por medio de diagramas de transición de estados. En realidad, STATECHARTS es un lenguaje de representación de diagramas de transición de estado enriquecido con términos que permiten representar jerarquía, ortogonalidad, expresiones y conexiones.

STATECHARTS se organiza en torno a estados y transiciones. El estado inicial es aquel apuntado por una transición que parte de un círculo relleno. Las transiciones, representadas por arcos orientados, son las condiciones que permiten la evolución entre dos estados conectados. Estas transiciones pueden ser directas o estar divididas en varias opciones mediante el empleo de un conector de condición, representado por un círculo conteniendo la letra *"C"*. Podría decirse que este conector es una especie de estado ficticio intermedio a dos estados que permite introducir una o varias alternativas a la transición original.

Los estados de un STATECHARTS pueden estar agrupados formando una jerar-

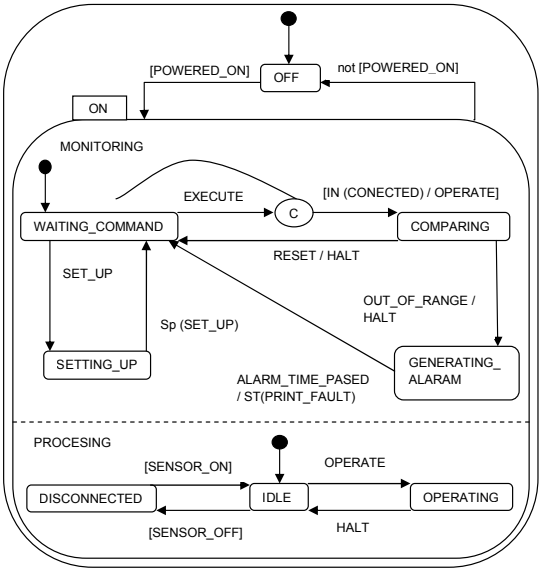

Figura 2.11: Ejemplo de STATECHART

quía. La agrupación de dos o más estados forman un nuevo estado que como tal, puede recibir y ser origen de transiciones de y hacia otros estados. El estado que agrupa a varios estados se llama padre de estos. Se dice que dos estados son hermanos si tienen el mismo estado padre. La agrupación en una jerarquía no impide que haya transiciones con su origen en un estado interno a la jerarquía y destino en un estado externo a la misma, y viceversa.

La ortogonalidad es otra característica relevante y muy potente de los STATECHARTS. Esta técnica permite representar en un solo STATECHARTS dos comportamientos que evolucionan simultáneamente (de manera independiente o no). Este tipo de estados se denominan ortogonales o estados *"Y"* (and-states), lo que quiere decir que estar en un estado *"Y"* significa estar simultáneamente en cada uno de los subestados que lo componen. Los estados ortogonales se representan por una línea de puntos que divide al estado en dos o más partes. Cada una de las partes es un estado con todas las propiedades enumeradas hasta ahora incluida la ortogonalidad. En la figura 2.11 se observa que existen dos estados ortogonales, Monitoring y Processing, que forman el estado *"On"*. Estar en el estado *"On"* significa estar simultáneamente en Monitoring y Procesing, y siguiendo la teoría vista hasta ahora, esto significa a su vez, para cada uno de estos dos estados, estar

solamente en uno de sus estados integrantes.

El lenguaje de STATECHARTS presenta un conjunto de características adicionales a las vistas hasta ahora que podrían considerars como las más relevantes y que facilitan la labor de modelar un sistema, a saber:

- Condiciones y eventos relativos a los estados. En cualquier estado del STATECHARTS es posible conocer la situación de activación o no de otro estado, o cuándo el sistema entra o sale de un estado.
- Con el objeto de aclarar la representación gráfica mediante la poda de arcos repetidos, es posible agrupar arcos mediante conectores especiales como el conector *"C"*, el conector *"S"* que permite separar o fusionar un evento en varios eventos componentes, el conector conjunción similar al anterior pero para condiciones en vez de eventos, y los conectores de diagrama que no son más que etiquetas que marcan dos puntos en el STATECHARTS que se supone estarían unidos por un arco (generalmente largo y que cruzaría por encima de otros arcos o estados enturbiando la claridad del diagrama).
- Reglas sintácticas del lenguaje textual empleado para representar expresiones.
- Reglas semánticas que definen cómo se debe interpretar cualquier STATECHART".

Tras la definición de STATECHART, Harel y su grupo de investigación han continuado trabajando en el desarrollo de lenguajes para modelado de sistemas y han desarrollado concretamente dos más que vienen a servir de herramienta para representar su concepto de modelo de proceso reactivo recogido en [HP98]. Según este modelo un sistema, está formado por tres vistas:

- Vista Funcional. Representa el qué del sistema, ésto es, la funcionalidad que tiene el sistema, expresada por medio de funciones, objetos, procesos, etc. El lenguaje que emplean para modelar esta vista se llama Activity-Charts.
- Vista de Comportamiento. Representa el cuándo, es decir, describe el comportamiento del sistema a lo largo del tiempo. El lenguaje que emplean para modelar esta vista se llama State-Charts.

- Vista Estructural. Representa el cómo, es decir, cuales son los componentes del sistema real (elemento de control, dispositivos de entrada/salida, comunicaciones, etc.) y asigna las actividades a cada uno de estos componentes. El lenguaje de modelado de esta vista se llama Module-Charts.

El modelo se completa con las reglas y mecanismos necesarios para la conexión de los distintos diagramas expresados en los tres lenguajes y un diccionario de datos.

STATECHARTS vs. otros Lenguajes de Modelado

Este lenguaje ha sido adoptado por varias Métodologías e incluso por otros lenguajes de modelado como UML, como mecanismo de representación del comportamiento del sistema. Este hecho le confiere un cierto grado de estandarización sobre todo entre la comunidad de ingenieros de software, sin embargo, entre los ingenieros de automatización su penetración no es tan alta, y tiene en GRAFCET su mayor competencia aún teniendo en cuenta que este último sólo sirve para modelar sistemas secuenciales mientras aquel sirve para representar cualquier tipo de sistema reactivo.

Probablemente, la razón fundamental de este hecho se deba a la sencillez de GRAFCET, su claridad en la representación de características como el paralelismo estructural, su intuitiva representación gráfica y a la tradición de uso.

Además, el hecho de que el estándar internacional IEC 61131 en su parte 3 [IEC93] correspondiente a los lenguajes de programación de autómatas programables, incluya SFC (GRAFCET convertido en lenguaje de programación) en vez de STATECHARTS, no ayuda a desbancarle de su posición dominante.

2.6. Sistemas de control orientados a objetos sobre PLCs

El análisis teórico y de diseño de sistemas de control ha sido objeto de varios estudios académicos, principalmente en relación con los sistemas de eventos discretos y su descripción a través de máquinas de estados finitos o redes de Petri [Pet62]. Las RdP son ampliamente utilizadas para modelar y analizar sistemas industriales principalmente debido a:

- Su semántica formal.
- Gráfica de la naturaleza.
- La expresividad.
- La disponibilidad de técnicas de análisis para demostrar propiedades estructurales (propiedades de invariancia, punto muerto, Liveness, etc).
- Posibilidad de definir y evaluar los índices de rendimiento (el rendimiento, tasas de ocupación, etc.).

Sin embargo, la construcción de modelos de RdP para sistemas complejos no es una tarea fácil y es propensa a errores. Por otra parte, desde el punto de vista de la configuración de los sistemas industriales, éstos pueden cambiar durante su ciclo de vida, por tanto, es importante poder definir un enfoque que promueva la reutilización de componentes por medio del análisis de las aplicaciones [Mur89]. Pero estos enfoques han sido rara vez aplicados en la práctica, principalmente por la difícultad para obtener un modelo exacto del sistema a automatizar.

Desde el punto de vista del desarrollo de sistemas de control orientados a objetos en la automatización industrial, varios autores, entre los que se puede citar a Duran, Bonfatti o Matfezzoni [DB94, BG95, MC99] han desarrollado diversas aplicaciones en los que se enfatiza el uso de métodologías OO para el desarrollo de sistema de control, poniendo especial enfasis en la fase de modelado y diseño que conducen a una poderosa descripción de los dispositivos físicos en los que el sistema se puede dividir. De esta forma, se obtiene una clara correspondencia entre el modelo físico del sistema a diseñar y el modelo conceptual descrito. Desgraciadamente, la descripción que se obtiene es difícil de traducir en un PLC sin introducir limitaciones o pérdida de eficacia, o sin el desarrollo de una herramienta middleware entre el lenguaje de alto nivel y el modelo de objetos de bajo nivel para controlar el dispositivo elegido.

Una posibilidad para acercar los modelos conceptuales OO a la programación de los sistemas puede pasar por dividir el sistema en pequeñas partes para luego formar un todo en el que se combine hardware (la parte de la maquinaria eléctrica y mecánica) y su software de control. A este tipo de descomposición se le llama *"Mechatronic objets"* [BS06] que pueden ser desarrollados independientemente del programa principal, posibilitando la formación de una biblioteca de objetos que

aumentan la facilidad de mantenimiento y reutilización del software. El problema de este tipo de soluciones es su dificil aplicación en el mundo real si no existe un criterio común respecto al harwdare y lenguajes de programación usados para su implementación.

De acuerdo a los estudios realizados por el grupo Advisory ARC [Gro09], el impacto de los entornos de programación de sistemas de automatización no basados en en la norma IEC 61131 en la actualidad es mínimo. El estándar internacional IEC 61131[4] [IEC93] es una norma de aplicación a los equipos autómatas programables (PLC) y sus periféricos asociados tales como herramientas de programación y depuración (PADT), elementos de interface hombre-máquina (HMI), etc, que viene a paliar la falta de un patrón común en la construcción de sistemas de control de procesos industriales y maquinaria.

Esta norma se divide en 7 partes, a saber:

- Parte 1: Visión general.
- Parte 2: Hardware.
- Parte 3: Lenguajes de programación.
- Parte 4: Guías de usuario.
- Parte 5: Comunicación.
- Parte 6: Control *"Fuzzy"* o borroso.
- Parte 8: Guías para el diseño, implementación y uso de lenguajes de programación.

Es en la parte 3, dedicada a normalizar los lenguajes de programación y el modelo de ejecución de programas de control, donde se definen ciertos principios de Ingeniería del Software para su aplicación en la programación de PLCs tales como la modularización, estructuras, etc. Una cuestión importante a discutir es cómo utilizar estos nuevos lenguajes y las estructuras de IEC 61131-3 para mejorar el desarrollo de software y el mantenimiento de los sistemas.

El estándar IEC61131 permite diseñar aplicaciones de control de forma jerárquica cuyas características principales son:

[4]Para profundizar más sobre la norma IEC 61131, ver el apéndice A.4.3

- Permite que diferentes partes del programa se puedan ejecutar con una frecuencia diferente.
- Soporta el diseño de comportamientos secuenciales complejos mediante el lenguaje Sequential Function Chart (SFC).
- Permite la definición de estructuras de datos.
- Posibilita la programación en diferentes lenguajes, concretamente ofrece tres lenguajes gráficos y dos textuales para expresar distintas partes del control de la aplicación.

La norma IEC 61131-3 proporciona lenguajes estandarizados así como métodos de ejecución de programas que permiten la programación de diferentes sistemas de control como elementos software independientes del fabricante, describiendo los programas, funciones y bloques funcionales como diferentes tipos de POUs. El concepto de POU proporciona la capacidad de reutilización, ya que una vez definidos pueden ser reutilizados en diferentes partes del control de la aplicación. Esta reutilización es debida a que en cada una de esas partes se usa una instancia diferente del POU definido una sola vez.

Desde el punto de vista de la orientación a objetos, los sistemas industriales pueden verse como objetos físicos, con su correspondiente clase en el mundo conceptual. Es decir, se podría tener una clase robot que tendría una serie de propiedades y una serie de métodos (las acciones que puede realizar). Esta correspondencia puede llevarse a cabo dentro de la norma IEC 61131-3 por medio de la estructura de programación llamada *"bloque funcional"* (FB), encapsulando los principios básicos de la orientación a objetos.

2.7. UML en la automatización de procesos

La puntualidad (timeliness) es el común denominador en los sistemas de tiempo real. Se puede definir como la capacidad que tiene un sistema de responder de la manera esperada a los estímulos externos dentro de un intervalo de tiempo aceptable. Esta característica abarca un sin número de sistemas de tiempo real, desde los puramente dirigidos por el tiempo hasta los puramente dirigidos por los eventos. Durante años, estos sistemas emplearon para su desarrollo sus propios

lenguajes, patrones de diseño y estilos de modelado, pero también tenían en común el uso de una herramienta para su modelado denominada *"ROOM"*.

ROOM es el acrónimo de *"Real-Time Object Oriented Modeling"* y es una notación de propósito específico para el modelado de los sistemas en tiempo real. Una de las grandes virtudes de ROOM radica en la definición de una serie de construcciones arquitectónicas que recaban la experiencia colectiva de varios equipos de desarrollo en varios proyectos y las bases del diseño arquitectónico para este tipo de sistemas.

En 1998, Bran Selic y James Rumbaugh [Sel98] llevaron a cabo una investigación para determinar la viabilidad de modelar Sistemas de Tiempo Real (RTS) usando una notación de propósito general: UML. En esta investigación se enfocaron en los RTS que se caracterizan por ser complejos, dirigidos por eventos y potencialmente distribuidos. Este tipo de sistemas son los empleados comúnmente en aplicaciones de telecomunicaciones, aplicaciones aeroespaciales y aplicaciones de control automático. Los proyectos para desarrollar el software asociado a los mismos demandan un gran esfuerzo inicial e involucran a grandes equipos de desarrollo y, al igual que la mayoría de los proyectos, deben adaptarse ágilmente a los inevitables cambios. Los resultados de la investigación, en pocas palabras, dieron como resultado que las construcciones definidas por ROOM podrían modelarse en UML usando simplemente sus mecanismos de extensión estándar.

La investigación mencionada en el párrafo anterior define tres construcciones para el modelado de estructura: las cápsulas, los puertos y los conectores. Las capsulas y los puertos no son otra cosa que clases con el estereotipo de *"capsule"* y *"port"* respectivamente, a los cuales se les asocian una serie de restricciones y características adicionales. Estas construcciones empleaban principalmente para su modelado diagramas de clases y diagramas de colaboración.

Así mismo, la extensión para el modelado de RTS define tres construcciones para el modelado del comportamiento: El protocolo, las máquinas de estados y los servicios de tiempo.

Con la especificación de UML 2.X se incorporaron dos nuevos diagramas que son la nueva alternativa para el modelado de los sistemas de tiempo real, es decir, los diagramas de estructura compuesta y lo diagramas de tiempo. En los diagramas de estructura compuesta, las capsulas se generalizaron en las partes mientras que los puertos y los conectores conservaron su nombre. La debilidad de UML en un enfoque fuerte en el tiempo, derivo en la incorporación de los diagramas de tiempo.

2.7.1. IEC 61131-3 bajo UML

UML es uno de los estándares más reconocidos en la Ingeniería del Software para el modelado gracias a su componente eminentemente gráfica. Antes de su utilización en el estándar IEC 61131-3 desde el punto de vista del análisis y diseño, se tendía a utilizar modelos más tradicionales de la Ingeniería del Software como el de C.M. Davidson [DW97], adaptando el OMT de Rumbaugh [RL90] para aplicarlo a la programación de PLCs sobre OO, usando modelos de ciclo de vida en cascada, aunque la realidad es que rara vez se suele realizar el modelado de las máquinas individuales que forman parte del sistema, principalmente si se trata de procesos híbridos, es decir, procesos discretos y continuos. A pesar de que los servicios fisicos se pueden modelar para facilitar la simulaciones del arranque del sistema en tiempo de ejecucion o el modelado de los procesos industriales como los propuestos por Kain [KS09], en la actualidad, la reutilizacion del software de automatizacion está mas aplicado a nivel de los FBs [KVH04], creando software que permite seleccionar elementos reutilizables de bibliotecas de FBs.

Con la aparición de UML se ha producido un rápido cambio de paradigmas hacia la notación gráfica de UML, lo que proporciona tanto al experto, como al lector ocasional de la documentación con pocos conocimientos de la terminología del paradigma, un conocimiento bastante profundo del sistema que se está implementando.

La integración de UML en los procesos de diseño de sistemas de control está haciendo que la especificación, modelado y escalabilidad de los proyectos sea más efectiva y elevada, como dice H. J. Kohler [KA00], consiguiendo desarrollos más rápidos, eficaces y baratos.

UML sirve para especificar el comportamiento dinámico de los procesos, así como la componente de tiempo real gracias a UML-Real Time (UML-RT). Su utilidad puede observarse por ejemplo en la propuesta de Lich [Lic04] para la integracion de automatas de temporizacion en la verificacion del comportamiento de tiempo real, o el framwork de Secchi [SF07] para unificar los componentes físicos del sistema y el modelado de software, con el uso de perfiles de UML-RT. ¿Pero es posible aplicar esa componente de tiempo real del software que soporta UML a la industria? En sistemas embebidos, la aplicación de UML es más fácil como demostró B.P. Douglas [Dou99] avanzando UML para su utilización en sistemas empotrados, ¿pero se puede dar soporte el estándar IEC 61131-3?

Una aproximación para responder a esa pregunta la realiza Torsten Heverhagen [HT01] que introduce el concepto de *"FBA"* (Function Block Adpater). Se trata de un FB con líneas de entrada y salida que conectan puertos u otros FBs. El propósito de los FBA es dar a UML-RT y a IEC61131-3 un lenguaje común para la especificación entre componentes de sus modelos. Los FBA no están pensados para especificar características de los FB, es decir, no especifican qué pasa después de enviar una señal a un FB. Por eso, no se le ponen estructuras del tipo IF-THEN-ELSE ni bucles. En el año 2003, Herverhagen [Hev03] realiza una revisión a su sistema adoptando nuevas especificaciones para adaptar los FBA a UML 2.0. Esta solución presenta varios inconvenientes entre los que se puede destacar, entre otros:

- Se necesita poder describir la forma en que los distintos componentes del sistema se relacionan.
- Se tiene que poder definir la lógica de negocio interna de los FBs.
- Es necesario tener en cuenta las condiciones de seguridad.

Una posible solución para estos problemas pasaría por usar los diagramas de colaboración de UML como se propone en la metodología propuesta por E. Estévez [EO07], permitiendo el intercambio de informacion entre las herramientas de desarrollo; o aplicar UML en el desarrollo de software considerando las restricciones de seguridad para las pruebas propuestas por PROFIsafe[5] (Profibus safety), como indica Dietrich [DF06].

Otra alternativa la aporta el grupo de investigación formado por Giese [GH], Gehrke [Geh05] y Burmester [BS05] para adaptar UML a los requerimientos de tiempo real y las tareas de los sistemas mecatrónicos, proporcionando además una generación automática de código ST de la norma IEC 61131.

Kleanthis Thramboulidis [TT06] utiliza UML para minimizar el cambio que se produce al transformar un FB al paradigma orientado a objetos en sistemas distribuidos. Para ello, desarrolla *"CORFU"* y lo adapta a UML. El frameWork CORFU fue presentado en el año 2002[Thr02] y a partir de ahí, fue añadiéndole mayor funcionalidad como por ejemplo, una herramienta CASE de generación automática

[5]Primera tecnologia abierta de comunicaciones funcional de seguridad para sistemas de automatiacion distribuidos. Su especificacion se publico por primera vez en primavera de 1999.

de código [DS04]. El dilatado trabajo Thramboulidis a lo largo de los años es una buena aproximación a la orientación a objetos en la automatización de procesos. El problema de esta aproximación es que se centra en sistemas distribuidos sobre la norma IEC 61499 y, como ya se ha dicho, esta tesis se centra sobre la norma IEC 61131 que es la más aceptada en la comunidad industrial.

Friedrich [Fri09] llevo a cabo una serie de experimentos para evaluar el beneficio del modelado como tal, y mas especificamente del modelado de procesos con UML y su posterior programación en un PLC. Los enfoques basados en UML y el lenguaje de control idiomatico (ICL) se compararon con las técnicas de programacion de PLCs sin ningun soporte notacional. El resultado del experimento fue que los sujetos se sintieron confusos por el número de posibles diagramas a usar de UML y eran incapaces de crear un modelado correcto. Además, la mayoria de los sujetos criticaban la falta de soporte de las herramientas de modelado, es decir, sentían que el uso de lápiz y papel para las tareas de modelado no era suficientemente flexible. La transferencia desde el modelado a la programación no era claro. Los usuarios pedían un generador de codigo automático en el caso de que el modelo fuera creado con una herramienta de software.

Katzke [KVH05] da respuesta a la falta de claridad y cantidad de diagramas que presentaba UML que reclamaban los programadores de automatas por medio de UML-PA. UML-PA fue desarrollado en el DiSPA por Fischer [FG04] como un sistema de modelado personalizado para la automatización. UML-PA está compuesto de varios sublenguajes que sirven para describir las diferentes estructuras, iteraciones y comportamientos de los sistemas.Además, posee una menor complejidad que UML, con menor número de tipos de diagrama y menos elementos de modelado. Por medio de UML-PA, los ingenieros de automatización obtienen:

- Decisión más rápida sobre el tipo específico de diagrama a usar.
- Decisión más rápida y fácil de los vinculos entre los objetos de software y los sensores/actuadores del sistema.
- Resúmen de las caracteristicas típicas de los objetos.
- Descripciones de los comportamientos de las interaciones y escenarios mas rapidos y precisos.
- Determinación del comportamiento del software del sistema completo en escenarios individuales.

Otra opción es acudir de nuevo a redes las RdP [Pet62] intentando mezclarlo de alguna forma con UML. De esta manera, se obtendrían los beneficios de modelado OO sumado con la capacidad de las RdP para definir las reglas de negocio del sistema.

Francesco Basile [BC08] presenta una métodología para la aplicación del paradigma OO en sistemas distribuidos a través de la aplicación del modelado bajo UML usando RdP. Especifica la necesidad de usar UML en los siguientes casos:

- El sistema real es complejo.
- Las relaciones entre las distintas partes del sistema a automatizar no se pueden expresa a través de leyes matemáticas.
- Haya muchos componentes que participen en el proceso (partes, máquinas, proceso).
- Los procesos presentan características comunes.
- El proceso se puede distribuir.
- El software comercial es incapaz de captar la dinámica de un sistema complejo.

Luca Ferrarini [FV03] ya profundizó en el uso de RdP con UML. Para su desarrollo, divide los sistemas en dos partes. El modelo de planta y el modelo de control. Los dos son modelos jerárquicos. En el primero se definen objetos y clases con UML y en el segundo se define un análisis mecánico, esquemas eléctricos e hidroneumaticos, dándole una concordancia con los objetos del otro modelo. Ésta es una buena aproximación, ya que además genera código automáticamente. Exactamente genera código JAVA, al igual que D.N. Ramos-Hernandez [RHF05], que desarrolla IDN (Integrated Design Notation), un entorno novedoso sobre OO para sistemas distribuidos sobre la norma IEC 61131 cuya herramienta CASE se integra perfectamente con Simulink (entorno de programación visual que funciona sobre el entorno de programación Matlab) y una notación para el proceso de control (PiCSI). El inconveniente de estas aproximaciones es que el código generado es JAVA, con lo que se sigue necesitando un intérprete para adoptar ese código a los lenguajes recogidos en IEC 61131-3 y así poder ser usados en un PLC real. Además, la aproximación de Luca Ferrarini está pensada para trabajar sobre un PC industrial corriendo sobre Linux.

2.7.2. Sistemas discretos

Existen numerosas aproximaciones a la OO en la Ingeniería de Control, pero la gran mayoría enfocadas a sistemas distribuidos. Este trabajo se centra en el estudio de la viabilidad e implicaciones del uso del paradigma de OO en el desarrollo de programas de control de sistemas de eventos discretos para ser ejecutados sobre PLCs basados en la norma IEC 61131. Bajo esta definición, no se engloban aquellos sistemas que operan sobre señales que ocurren de manera continua a medida que evoluciona el tiempo, sino al contrario, suceden en instantes de tiempo determinados por lo cual se les denomina sistemas de eventos discretos [Thi01].

La tasa de modularidad y la gestion de los cambios en los requerimientos son clave para una ingenieria de exito. El paradigma OO ha permitido el exito en el desarrollo de aplicaciones en el campo de la informática de propósito general, porque está dirigido a manejar las necesidades de los conceptos complejos en el software actual. Las ampliaciones OO de la norma IEC 61131 se especifican con clases, métodos, interfaces y herencia para hemerger como una solucion prometedora [VHK11]. Sin embargo, estas ampliaciones hacia el paradigma OO no incluyen mecanismos de eventos (usados en Java, C#, etc), ya que se basan en el procesamiento de programas síncronos.

Vogel-Heuser demuestra la viabilidad de aplicar la notación UML en la automatización de procesos, proporcionando además una herramienta CASE para la generación de código compatible con PLCs basados en la norma IEC 61131 usando la versión 1.4 de UML [VHK05]. Su editor deriva el concepto y seleccion de los diagramas de UML dando soporte a los entornos de ingeniería de CoDeSys 3.0, la actual referencia en las ampliaciones OO de la norma IEC 61131-3. Además, soporta 3 tipos de diagramas [WVH09, WK10], a saber: Diagramas de clase de descripcion de estrucutas, tablas de estado y diagramas de actividad de descripcion del comportamiento.

Fabien Chiron [CK07] va un poco más lejos que Vogel-Heuser y usa la versión 2.0 de UML para especificar orientación a objetos a través de los FBAs que son interfaces que encapsulan los FBs y la descripción de dicho interfaz (ver figura 2.12). Ya que UML no ha sido pensado para ello, Fabien Chiron redefine el diagrama de clases, añadiendo la clase *"bloque"* y usando los *"FlowPorts"*, proporcionando nuevas funcionalidades para conectar FBs y permitiendo su comunicación.

Shigeru Kajihara [KO04] presenta un entorno de trabajo para el desarrollo de

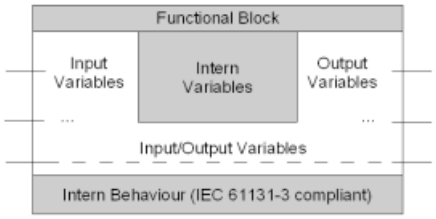

Figura 2.12: Configuración interna de un FBA

software orientado a objetos para sistemas discretos. Su entorno soporta los 3 lenguajes gráficos de IEC 61131-3 y además, permite mezclarlos proporcionando a los programadores de PLCs bibliotecas reutilizables de componentes OO.

G.Aiello [AB07] propone una herramienta (parecida a UML) para generar código IL automaticamente (ILgenerator tool) a partir de especificaciones y código en JAVA. Se basa en la métodología "*AGILE"* y UML para realizar las especificaciones del sistema. El problema que presenta es que sólo genera código IL y requiere altos conocimientos de Ingeniería del Software por parte por parte del programador en el diseño de las especificaciones, así como un alto grado de experiencia en el uso de UML y de *"AGILE"*. Además, sólo permite la entrada de código por medio del lenguaje IL limitando el uso del resto de lenguajes de más alto nivel recogidos en el estandar IEC 61131.

Una de las mayores ventajas que presenta la programación OO respecto a la programación estructurada es el desacoplamiento que se produce entre la lógica interna de los procesos y los interfaces que éstos proporcionan. Una implementación que desacopla el acceso al código de los componentes es la base fundamental para el intercambio de código a nivel de programación. Por ejemplo, un sensor de temperatura (además de su parametrización) que poee sólo un metodo que devuelve el valor de la temperatura medida. De esta forma, cada sensor puede ser fácilmente remplazado por otro componente con un interfaz compatible sin ningun tipo de modificación en la llamada. Un componente accesible por la estandarizacion de interfaces ativa la posibilidad de crear software mantenible en todo el sistema o ciclo de vida del proyecto, incluso si los componentes son sustituidos debido a un cambio de tecnologia. Diehm [Die08] describe las siguientes desventajas de la recogida de requerimientos con el lenguaje SFC respecto a hacerlo con los diagramas de estado de UML:

- Cada cambio de estado de SFC conduce a un cambio de ciclo del PLC. Por tanto, las secuencias detalladas no pueden ser modeladas en SFC, aunque se basa en un automata de estados.
- Los pseudo estados deben ser insertados antes de las secuencias de salto paralelo.
- Las rutinas de manejo de errores conducen a condiciones complejas de transicion.
- A menudo es necesario insertar saltos para realizar algoritmos particulares, lo que provoca una reducion de la legibilidad.

Una importante ampliación para el soporte de la OO es la desarrollada dentro del grupo 3S-Smart por Werner [Wer09] en la que se presentan dos nuevos tipos de FBs. Por un lado, el FB interfaz (como se muestra en la figura 2.13) que permite especificar el esqueleto de una jerarquía de clases que están obligadas a implementar los métodos que se especifican en dicho interfaz (muy parecido a la declaración de interfazes en JAVA o a las clases abstractas de C++) y por otro lado, amplia la definición de los FBs de la norma IEC 61131 permitiendo declarar e implementar métodos en su interior, tal y como se muestra en la figura 2.14.

```
INTERFACE Room

PUBLIC METHOD DayTime : VOID
END_METHOD

PUBLIC METHOD NightTime : VOID
END_METHOD

END_INTERFACE
```

Figura 2.13: Interfaz de métodos simulados de FBs

La solución que Werner propone posibilita la aplicación de tres de los pilares fundamentales de la OO, a saber, encapsulación, herencia y polimorfismo, agregando métodos en la definición de los FBs.

Estas aproximaciones, que amplían la norma IEC 61131 para posibilitar la programación OO por medio de la utilización de los FBs del estándar, presentan un problema. Al utilizar los FBs o una modificación/ampliación de los mismos, se

```
FUNCTION_BLOCK LightRoom Implements Room
VAR_IN_OUT
  Light : BOOL;
END_VAR

PUBLIC METHOD DayTime : VOID
  Light:=FALSE;
END_METHOD

PUBLIC METHOD NightTime : VOID
  Light:=TRUE;
END_METHOD

END_FUNCTION_BLOCK
```

Figura 2.14: Implementación de métodos simulados de FBs

está definiendo dos términos diferentes (un FB y una clase) por medio de un único concepto, lo que puede provocar errores en su utilización por parte de los ingenieros de control debido a que, por un lado, los FBs están pensados para encapsular un conjunto de atributos (que se asemejan a los atributos de una clase) y por otro lado, cada FB encapsula un único código o algoritmo que opera sobre dichos atributos.

La herencia es otro poderoso mecanismo de la orientación a objetos en la automatización de procesos que posibilita, por un lado la modularización de los sistemas y por otro, la reutilización de los componentes software ya desarrollados. Pocos autores desarrollan la herencia en el control de procesos por medio de la composición de jerarquías de clases. Algunas aproximaciones se basan en sistemas holónicos para definir componentes software OO que permiten generar sistemas más grandes por medio de la composición de diversos holones [NX00, BN07]. Según Koestler [Koe67] las características básicas que definen un *"Holon"* son:

- Autonomía para tomar sus propias decisiones de control.
- Cooperación con otros holones para ejecutar acciones de control coordinadas o globales.

En la actualidad se están desarrollando un gran número de investigaciones en torno a la aplicación de los conceptos sobre holones al desarrollo de sistemas de control industrial. Con este objetivo en 1994 se constituyó el consorcio *"Holonic*

Manufacturing Systems (HMS)" bajo el programa de investigación internacional *"Intelligent Manufacturing Systems (IMS) Research Program"* que agrupa a compañías y centros de investigación de Australia, Canada, Europa, Japón y Estados Unidos. El objetivo inicial de este consorcio era el desarrollo de un ensayo que demostrase la viabilidad de uso de las teorías de Koestler en el control de sistemas industriales. Fruto de esos esfuerzos se sentaron las bases de cómo debería ser un sistema holónico de producción desde varios puntos de vista: interfaz hombre máquina [KN00], modelo computacional a seguir [Dee00], flujo de materiales [SS00], etc. Desde un punto de vista práctico, se han llevado a cabo algunas implementaciones de estos conceptos entre los que se puede destacar por ejemplo, la descripción de las características básicas de una fresadora holónica [Zua00].

Además de la herencia, la agregación o composición de clases es otro poderoso mecanismo dentro del paradigma OO que permite componer un objeto grande a partír de objetos más pequeños. Victor Gonzalez [Gon02] presenta en su Métodología MLAV un sistema para la descripción y documentación de proyectos de control OO permitiendo además de la herencia, la composición de clases complejas por medio de la agregación a partir de tres tipos de objetos más sencillos: Sensores, Preaccionadores/Accionadores y Elaborados.

2.8. Conclusiones

La conclusión fundamental que se extrae de este estudio es que la adopción del POO en la Ingeniería del Software propicio una clara reducción de los costes de producción en las empresas de desarrollo de programas informáticos, gracias principalmente a la capacidad de la OO para la reutilización de código. Pero el impacto del POO en el mundo informático ha ido más lejos que la aparición de un elevado número de lenguajes OO. Desde su aparición y consolidación, han aparecido diversas Métodologías y sistemas de modelado OO que permiten unas especificaciones, análisis y diseño de los proyectos mucho más potentes y cercanas al lenguaje natural que si se aplicasen con técnicas más clásicas propias del paradigma estructurado.

Así como el POO es una realidad en el mundo empresarial informático, en la automatización de procesos y más concretamente bajo el estándar IEC 61131, el mundo industrial apenas ha comenzado a asomar la cabeza a este tipo de tecnologías. So-

lamente desde el ámbito académico están apareciendo en los últimos años tímidos intentos para adoptar algunos de los pilares en los que se basa la OO a la norma IEC 61131.

Dado el carácter conservador del mundo industrial no se espera grandes cambios en el estilo de programación de los sistemas de control en los próximos años. Hacen falta estudios que demuestren de forma empírica, la potencia que proporciona a la Ingeniería de Control la OO como se ha demostrado por medio de diversos estudios en la Ingeniería del Software.

Capítulo 3

MIOOP. Adaptación de la norma IEC 61131 al paradigma orientado a objetos

Si tenéis la sensación de estar estancados
y de no hacer ningún progreso,
abandonad vuestro estado de ánimo y pensad en
vuestro corazón que estáis empezando algo nuevo.
Miyamoto Musashi (Anillo del fuego)

3.1. Introducción

En los anteriores capítulos se ha hecho un recorrido histórico de la evolución que se ha producido en los paradigmas de programación en informática, desde

la aparición de los primeros ordenadores, hasta el desarrollo de la programación orientada a objetos y los beneficios que éste nuevo paradigma ofrece. Así mismo, se ha puesto de manifiesto la razón de la aparición de la norma IEC 61131 como respuesta a la falta de estandarización en los lenguajes de programación dentro de la Ingeniería de Automatización, y alguno de los intentos que se han hecho para paliar la brecha entre los lenguajes propuestos por el estándar y las posibilidades que ofrece la orientación a objetos en la automatización de procesos.

La introducción de las nuevas características orientadas a objetos en la norma IEC 61131 implica, tal y como se detalla a lo largo del anterior capítulo, que los programas así desarrollados no puedan ser ejecutados en los PLCs actuales. Por este motivo es necesario generar y traducir los nuevos programas orientados a objetos a las estructuras, tipos de datos y lenguajes proporcionados por el estandar IEC 61131, para posteriormente poder probar ese código con las herramientas (PLCs y soft PLCs basados en IEC 61131-3) de que se disponen en la actualidad. Por otro lado, el caracter conservador de la norma, donde prima por encima de otras características la seguridad, obiliga a que se tenga muy presente esta doctrina al a hora de traducir las características orientadas a objetos a la norma IEC 61131.

La traducción de las características orientadas a objetos de un programa de control a un código no orientado a objetos soportado por IEC 61131, en la mayoría de las ocasiones no consiste solamente en una correspondencia directa de código, sino que son necesarias acciones auxiliares que permitan dicha traducción, como por ejemplo construir la lista de propiedades de los atributos y consultar dichas listas para generar el código IEC 61131 de una forma u otra según convenga en cada caso. Estas acciones auxiliares no forman parte de las reglas que amplían la norma para dar soporte a la programación orientada a objetos, sino que serán llevadas a cabo por la herramienta que permita la traducción de código orientado a objetos al código no orientado a objetos de la norma actual.

Este tipo de traducción (ver figura 3.1) permite, por un lado, valorar las implicaciones que tiene el uso de la orientación a objetos en la automatización de procesos, y por otro lado, abre la posibilidad a los programadores de retocar el código generado si fuera necesario.

El siguiente paso, una vez concluida la traducción de un programa de control orientado a objetos a su versión compatible con la norma IEC 61131, consistiría en generar o convertir el código de la traducido a código máquina del PLC real

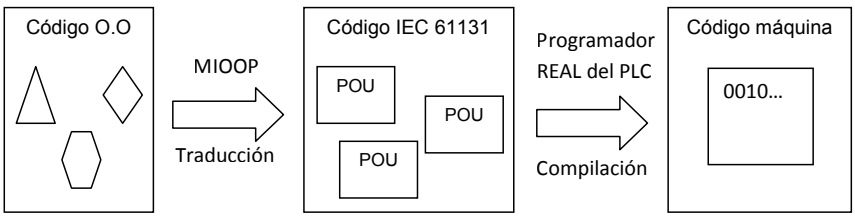

Figura 3.1: Pasos de traducción del programa de control

o soft PLC que finalmente vaya a controlar el proceso para el cual se desarrolló el código. A este mecanismo se le conoce como compilación y es realizado por la herramienta de programación del PLC real o soft PLC que se vaya a emplear para ejecutar el código, es decir, que será esta herramienta la encarga de generar el código máquina adecuado para el PLC a emplear, partiendo del código traducido a IEC 61131.

En este capítulo se presenta MIOOP (Modification of the IEC 61131 standard for Object Oriented Programming), un nuevo conjunto de modificaciones sobre la norma IEC 61131 que permita su evolución desde el paradigma de programación estructurada al paradigma de orientación a objetos, así como todos los mecanismos necesarios para su traducción y aplicación a un PLC real basado en el estándar.

La motivación que ha provocado el desarrollo de MIOOP es la de ofrecer a los ingenieros de automatización toda la potencia del paradigma de programación orientado a objetos y soportar los pilares básicos de éste: abstracción, encapsulamiento, modularidad, herencia y polimorfismo.

La definición de MIOOP se aborda en 4 partes:

1. Clase. En esta sección se definen 3 de los principios básicos que caracterizan a la orientación a objetos: encapsulación, abstracción y modularidad.

2. Herencia. En esta sección se define el concepto de jerarquía de clases y los distintos tipos de clases derivadas.

3. Polimorfismo. En esta sección se definen los conceptos básicos que permiten que un método de una clase sea invocado de diferentes formas, dependiendo de la clase que lo invoca.

4. Sobrecarga. En esta sección se definen los 2 tipos de sobrecarga que permite MIOOP: La sobrecarga de métodos y la de operadores.

La información expuesta a lo largo de este capítulo está organizada desde dos puntos de vista, a saber:

- El qué. El conjunto de ampliaciones que se proponen al estándar IEC 61131, de manera que se dé soporte al paradigma de programación orientado a objetos.
- El cómo: Las modificaciones necesarias en la norma IEC 61131 que permitan traducir las mejoras propuestas por MIOOP a código estándar de la norma de manera automática.

La mayor parte de los ejemplos que se exponen a lo largo de este capítulo se desarrollan en lenguaje ST (lenguaje de texto estructurado), ya que es un lenguaje que en comparación con otros lenguajes, permite fácilmente ver todas las características de la ampliación propuestas por MIOOP y su traducción a la norma IEC 61131.

A continuación, se enumera el conjunto de características del paradigma orientado a objetos que se proponen como ampliación a la norma IEC 61131 y que se irán analizando a los largo de este capítulo:

1. Definición de clase y de objeto.
2. Definición de método.
3. Operador punto.
4. Herencia.
5. Operador de ámbito.
6. Modelo de protección en MIOOP.
7. Constructores y destructor de objetos.
8. Conversión de tipos.
9. Casting entre clases.
10. Herencia múltiple.
11. Punteros.

12. Polimorfismo.
13. Clases abstractas.
14. Interfaces.
15. Sobrecarga de métodos.
16. Sobrecarga de operadores.

3.2. Clases y objetos

Se puede definir un objeto como el *"encapsulamiento de un conjunto de operaciones (métodos ó servicios) que pueden ser invocadas externamente, y del conjunto de variables (atributos) que almacenan el estado resultante de dichas operaciones"* [PF96].

Un objeto además de un estado interno, presenta una interfaz para poder interactuar con el exterior. Es por esto por lo que se dice que en la programación orientada a objetos *"se unen datos y procesos"* y no como en su predecesora, la programación estructurada, en la que estaban separados en forma de variables y funciones.

Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común.

Se puede definir una clase como *"un conjunto de cosas (físicas o abstractas) que tienen el mismo comportamiento y características cuyas instancias particulares son los objetos"*. [PF96].

Por tanto, se puede decir que una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (instanciación) se ha de especificar de qué clase es el objeto instanciado.

3.2.1. Definición en MIOOP

El concepto sobre el que gira toda la programación orientada a objetos es el de clase. Para poder definir una clase a partir de las estructuras proporcionadas por la norma IEC 61131, se puede optar por implementarla mediante un FB de la misma

forma que se hace en algunas aproximaciones como las de Herverhagen [Hev03] o Fabien Chiron [CK07] de la siguiente manera:

Dada una clase de la figura 3.2, equivaldría a un FB con la estructura del algorítmo 3.1.

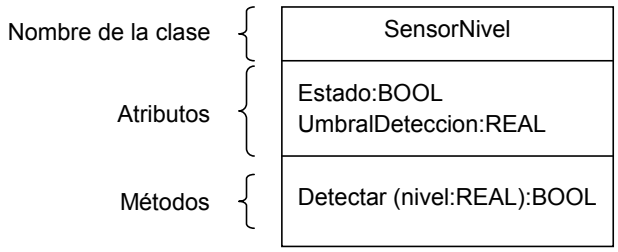

Figura 3.2: Esqueleto de una clase

Algoritmo 3.1 Equivalencia entre una clase y un FB

```
FUNCTION_BLOCK SensorNivel
  VAR_IN_OUT
    Estado : BOOL:=false;
    UmbralDeteccion : REAL:=10;
  END_VAR
    //Código del FB
END_FUNCTION_BLOCK
```

Esta aproximación se apoya en la potencia de los FBs para encapsular sus datos y poder crear instancias del mismo (ver la figura 3.3).

Cada una de las instancias de un FB se aloja en una posición distinta de memoria y por tanto, sus datos son independientes.

La definición de un FB sería equivalente a la definición de una clase, y las distintas instancias del FB serían equivalentes a los objetos, tal y como se observa en la figura 3.3.

El problema de esta solución se observa con detalle atendiendo a la definición dada por Lewis sobre un FB [Lew95] desde tres puntos de vista:

1. Conceptual. Se usa un tipo de POU ya existente en la norma para implementar un concepto nuevo con el que no es totalmente compatible.

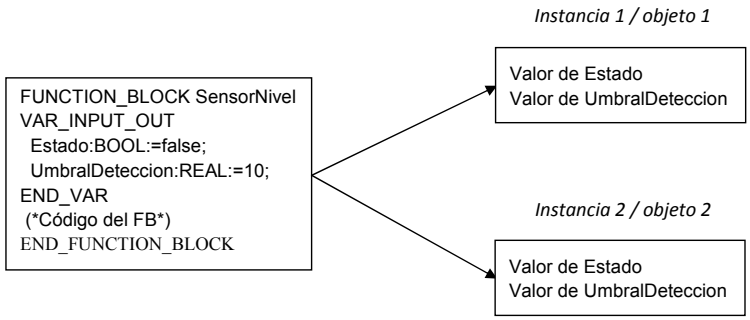

Figura 3.3: Instancias de un FB

2. Atributos. Según la propia definición de la norma IEC 61131 y de su mentor Lewis [Lew95], las variables locales a un FB no pueden ser accesibles más que desde el propio código que implementa el FB. Sin embargo, en la definición de clase del paradigma orientado a objetos, puede ser necesario permitir el acceso a los atributos que se deseen desde el exterior de la clase, es decir, desde otros objetos de otras clases.

3. Métodos. Dado que los métodos de una clase obligatoriamente deben acceder a los atributos de la clase para poder llevar a cabo sus reglas de negocio, según la definición de FB de la norma IEC61131, sólo podría lograrse este objetivo si la implementación de todos los métodos de la clase estuviesen contenidos en la propia declaración de la clase(en el FB), tal y como describe Bernhard Werner [Wer09] y se muestra en el algortítmo 3.2.
   Esta aproximación tiene tres problemas:

   a) Provoca que a la larga se tengan definiciones de clases muy extensas, ilegibles y difíciles de mantener y manejar.

   b) Obliga al programador a distinguir de alguna forma entre FBs puros y FBs que implementan clases.

   c) Obliga a modificar el estandar para poder definir métodos dentro de los FBs.

Es aquí donde radica el problema de esta última solución ya que no existe una correspondencia clara entre los métodos de una clase y el código de un FB, como se puede ver en la figura 3.4. Este problema se puede resolver añadiendo código

Algoritmo 3.2 Implementación de una clase según Werner

```
FUNCTION_BLOCK SensorNivel
  VAR_IN_OUT
     Estado : BOOL:=false;
     UmbralDeteccion : REAL:=10;
  END_VAR

  PUBLIC METHOD Detectar : BOOL
    VAR_IN_OUT
       Nivel : REAL;
    END_VAR
       IF Nivel>=UmbralDeteccion THEN
          Estado:=true
       ELSE
          Estado:=false;
       END_IF
       RETURN Estado;
  END_METHOD

END_FUNCTION_BLOCK
```

adicional a los FBs que definen las clases con el fin de poder seleccionar el método correcto ante una llamada dada, pero esto hace más difícil de entender y mantener los programas lo que implica una reducción en el rendimiento de la ejecución, como puede verse en la figura 3.5.

Otra posible solución para implementar una clase consistiría en observar cómo resolvió Stroupstrup el paso de C a C con clases [Str84]. Bjarner Stroupstrup usó el tipo definido de C llamado estructura para encapsular los atributos de una clase. Es el compilador el encargado de resolver qué atributos son accesibles, tanto por los propios métodos de la clase, como por los métodos de otras clases (ver apartado 3.9**,** modelo de protección).

En IEC 61131 existe, al igual que en C, el tipo definido por el usuario llamado estructura, de tal manera que una clase se definiría según indica el algorítmo 3.3.

Las estructuras están formadas por cualquier tipo de dato simple y pueden ser instanciadas tantas veces como el programador desee, ocupando cada instancia una posición distinta en la memoria, lo que implica absoluta independencia entre ellas. Sirven por tanto para implementar el concepto de encapsulación y de objeto.

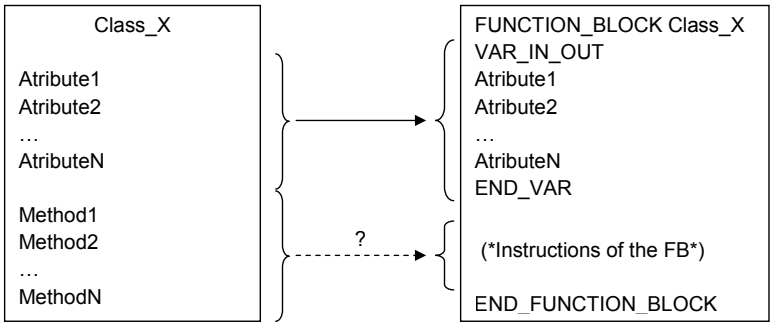

Figura 3.4: Traducción de un FB a un método

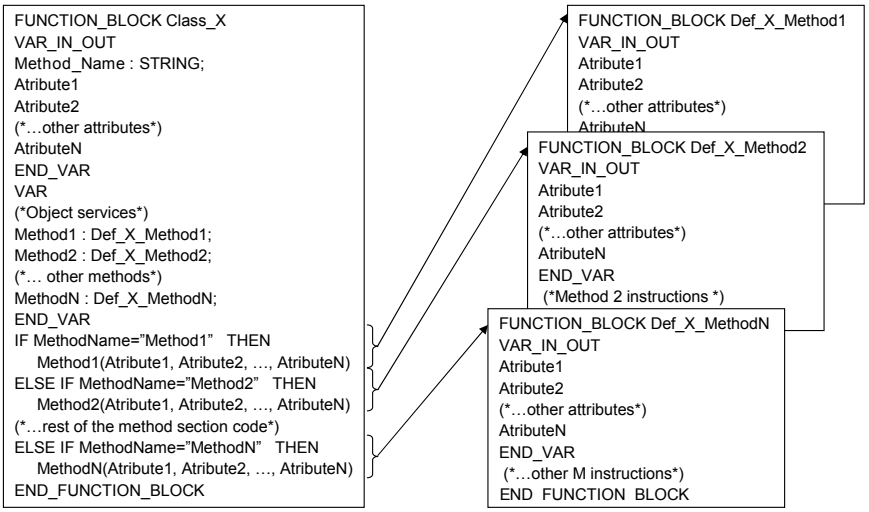

Figura 3.5: Selección del método correcto en una llamada

Algoritmo 3.3 Estructura que implmenta una clase

```
TYPE SensorNivel:
  STRUCT
    Estado : BOOL:=false;
    UmbralDeteccion : REAL:=10;
  END_STRUCT;
END_TYPE
```

Es de notar que el método *"Detectar"* declarado en la clase *"SensorNivel"* de la figura 3.2 no aparece como miembro de la estructura que implementa dicha clase. Ésto es así, porque un tipo estructura sólo puede albergar variables. En el apartado 3.3 se detalla cómo implementar los métodos de una clase y enlazarlos con las estructuras que implementan las clases a las que están asociadas.

Ésta solución es la que adopta MIOOP y a diferencia de las aproximaciones a través de un FB, en ésta, es el propio compilador el encargado de definir las reglas de acceso a los atributos de la clase en el momento de la compilación, de tal manera que un atributo privado no podrá ser accedido desde fuera del entorno del objeto, informando al programador que se ha producido un error de compilación.

Para separar semánticamente la definición de un tipo estructurado del de una clase, en MIOOP se opta por definir un nuevo tipo de POU llamado *"CLASS"* que sirva para especificar el tipo de objetos que se van a representar, describir los atributos y definir las cabeceras de los métodos, así como el acceso que el usuario tiene a tales miembros (ver algoritmo 3.4).

Algoritmo 3.4 Definición de una clase en MIOOP

```
CLASS SensorNivel()
  Estado : BOOL:=false;
  UmbralDeteccion : REAL:=10;
 METHOD Detectar (Nivel : REAL) : BOOL;
END_CLASS
```

Se ha optado por denominar a este nuevo tipo de POU como *"CLASS"* y no como estructura, en parte, por no romper con toda la historia de la informática, donde es comúnmente reconocible este término. Además, una clase contiene una definición mucho mayor, incorporando no sólo datos, sino que además añade servicios (métodos) a los objetos que la instancian. Por otra parte, la razón fundamental para la separación de una clase en un POU es debido a que una clase es semánticamente muy diferente a una estructura, ya que ésta sólo contiene datos.

El nuevo POU *"CLASS"* proporciona a los programadores un control de acceso a sus miembros (atributos y métodos). Por una lado, desde dentro de una clase, todos sus métodos no tienen ningún tipo de restricción para el acceso a los atributos de la clase. Por otro lado, por defecto y si el programador no indica lo contrario, los atributos de una clase no son accesibles desde fuera del ámbito de esta, mientras

que sus métodos si son accesibles. Es decir, por defecto, los atributos de una clase son privados y los métodos públicos (ver apartado 3.9**,** modelo de protección).

El tipo de un método (incluyendo el tipo de retorno y argumentos) es especificado en la definición del POU *"CLASS"*. El chequeo de los métodos en tiempo de compilación se hace en base a estas especificaciones. La implementación de un método se hace a parte de la definición del POU *"CLASS*" para dejar a una clase más como la declaración de un tipo de dato que como un mecanismo léxico para organizar código fuente. Este tipo de organización se realiza en otros lenguajes de programación como JAVA, donde se declara en un fichero las cabeceras de los métodos y en otro fichero se realizan las implementaciones de los mismos.

3.2.2. Traducción en IEC 61131

Tal y como se ha descrito en el apartado anterior, las clases se traducen por medio del tipo de dato recogido en la norma *"STRUCT"*, donde cada uno de los atributos de la clase se corresponde uno a uno con una variable contenida en la estructura.

Para identificar a una estructura que implementa una clase, ésta tendrá por nombre la etiqueta:

<NombreClase>

Donde:

- <NombreClase> identifica a la clase que implementa la estructura.

Por ejemplo, según esta definición, la clase *"SensorNivel*" se traduciría a IEC 61131 como la estructura que se muestra en el algorítmo 3.3.

3.3. Métodos

Los métodos son subrutinas que definen la interfaz de una clase, sus capacidades y comportamiento. Puede decirse que un método constituye un servicio que el objeto brinda a otros objetos para su uso. Para llevar a cabo su cometido, un método leerá y/o escribirá los atributos del objeto. Un método ha de tener por nombre

cualquier identificador legal distinto de los ya utilizados por otros miembros de la clase en que está definido a no ser que se quiera establecer una regla de sobrecarga (ver apartado 3.25 y apartado 3.26). Los métodos se declaran al mismo nivel que las variables de instancia dentro de una definición de clase. Todos los atributos de una clase, tanto los privados como los públicos, pueden ser accedidos a través de los métodos de la clase.

3.3.1. Definición en MIOOP

Para implementar un método desde un lenguaje no orientado a objetos, algunas aproximaciones como el paso de C a C++ [Str90] utilizan funciones simples que sólo son accesibles por parte del usuario a través de los métodos definidos en la clase. En la norma IEC 61131 existen dos tipos de POU similares, las funciones y los FB, que pueden ser utilizadas para dicho fin. Para diferenciar a un método de una clase de una función o un FB es necesario un tipo de declaración que distinga a estos dos de un método para que por un lado, exista una clara frontera entre una función o FB de la norma IEC 61131 y su utilización en la definición de métodos de una clase, y por otro lado, permita a un compilador controlar el acceso a dichos métodos para evitar invocaciones fuera del rango de actuación de la propia clase. En MIOOP se ha optado por el vocablo ingles de método, es decir, *"METHOD"*, y cuya definición es la siguiente:

> METHOD <Nombre método> (<ListaParametros>) : <Valor devuelto>

Donde:

- METHOD es la palabra reservada que identifica que un miembro de una clase es un método.
- <Nombremétodo> es el nombre del método dentro de la clase.
- (<ListaParametros>) identifica la lista de parámetros, separados por comas, que recibe el método.
- :<Valor devuelto> indica el tipo de valor que devuelve la ejecución del método en caso de devolver algún valor.

Para que una función o un FB puedan acceder a los atributos definidos en una clase pueden utilizarse tres estrategias:

1. Pasar como parámetro a la función o al FB una copia de la estructura.
2. Pasar como parámetro a la función o al FB todos y cada uno de los atributos de la estructura.
3. Pasar como parámetro a la función o al FB una referencia a la estructura.

La primera posibilidad no funcionaría ya que el método estaría trabajando con una copia de la estructura, con lo que todo cambio que se produzca en ella dentro del método, no se vería reflejado en los atributos de la clase.

Las otras dos posibles soluciones no serían factibles empleando funciones ya que una de las principales diferencias entre una función y un FB en la norma IEC 61131 radica en que, en la llamada a una función, todos los parámetros se deben pasar por copia mientras que los FB no tienen restricciones respecto a sus parámetros. Para solucionar este problema se podría optar por modificar la norma IEC 61131 para permitir a las funciones recibir parámetros por referencia pero esto obligaría a incumplir una de las premisas del estandar que es el de la seguridad. Los FBs deben ser instanciados previamente para poder ser utilizados mientras que las funciones pueden ser invocadas en cualquier momento. Este hecho, desde el punto de vista del paso de parámetros, obliga al programador a ser consciente de los efectos laterales que se pueden producir en los mismos dentro de la ejecución del POU. Por estas razones, un FB es el candidato idóneo para implementar un método dentro de la norma IEC 61131.

Una vez decidido que los FBs serán usados por MIOOP para traducir los métodos, hace falta seleccionar que tipo de paso de parámetros es más conveniente para que los métodos puedan acceder a los atributos de la clase a la que pertenecen.

El paso por valor queda descartado por los motivos que se han comentado anteriormente. En este punto hay que decidir cuál de las dos posibilidades restantes de paso de parámetros por referencia a un FB es la mejor solución.

La segunda posibilidad obligaría a realizar llamadas muy largas. Por ejemplo, la llamada al método *"Detectar"* sería:

```
Detectar (Estado, UmbralDeteccion, Apagado);
```

Esta solución sería válida tanto si se aproxima la encapsulación de los datos de una clase tal y como lo propone MIOOP (a través de una estructura) o a través de un FB. El único problema de esta solución es el aumento de líneas de código para que puedan ser accesibles los atributos de la clase, ya que estos irían situados en la parte *"VAR_INOUT"* del FB. El aumento de las líneas de código generadas con esta solución sería del orden de m^a, siendo *"m"* el número de métodos de una clase y *"a"* el número de atributos. Esta solución implicaría la necesidad de aumentar el tamaño de la memoria de los PLCs y por tanto, aumento del coste.

La tercera solución implicaría introducir una referencia a la estructura que implementa la clase en cada método y pasar en la llamada al método el propio objeto. Por ejemplo:

```
Detectar (Sensor : SensorNivel);
```

De esta manera, con un único parámetro se podría tener acceso a todos los atributos de la clase lo que implicaría mínimo consumo de memoria por parte del PLC y por tanto, esta es la solución que adopta MIOOP.

3.3.2. Traducción en IEC 61131

Cuando se traduce un método a su FB análogo se inserta como primer parámetro la clase a la que pertenece seguido de la lista de parámetros. La definición teórica en una línea de este tipo de FB sería:

```
FUNCTION_BLOCK <Nombre> (<Clase>, <Lista de parametros del
método>) : <Valor devuelto>
```

Donde:

- <Clase> es una referencia a la estructura que implementa la clase a la que pertenece el propio método.

Para identificar a un FB como un miembro de una clase, éste tendrá por nombre la etiqueta:

<NombreClase>_<NombreMétodo>

Donde:

- <NombreClase> identifica a la clase a la que pertenece el método.
- _<NombreMétodo> identifica el nombre que el usuario a dado al método dentro de la clase a la que pertenece.

Este tipo de etiquetas puede resultar muy útil al ingeniero de automatización si decidiera modificar el resultado de la traucción de MIOOP, pudiendo distinguir entre FBs miembros de una clase y los FBs creados por él.

El método detectar() se traduciría como muestra el algorítmo 3.5.

Algoritmo 3.5 Traducción de un método en IEC 61131 a través de un FB

```
FUNCTION_BLOCK SensorNivel_Detectar
  VAR_OUTPUT
    Retorno : BOOL;
  END_VAR
  VAR_IN_OUT
    THIS : SensorNivel;
  END_VAR
  VAR_INPUT
    Nivel : REAL;
  END_VAR
    IF Nivel>=THIS.UmbralDeteccion THEN
      THIS.Estado:=true
    ELSE
      THIS.Estado:=false;
    END_IF
    RETURN THIS.Estado;
END_FUNCTION_BLOCK
```

Donde *"THIS : SensorNivel"* es una referencia a la estructura de la clase a la que pertenece. En todos los métodos existe el operador *"THIS"* que referencia al objeto por el cual el método fue llamado. De esta manera, todas las propiedades de una clase son accesibles desde cualquiera de sus POUs miembro. Por esta circunstancia y gracias a que los FBs admiten el paso como parámetro de cualquier tipo de dato

primitivo por referencia, incluidas las estructuras, esta es la solución que adopta MIOOP.

El identificador *"THIS"* es una versión de la referencia *"THIS"* en Simula o el puntero *"THIS"* en C con clases. MIOOP en muchos sentidos, se basa en Simula67 [DJ67] y C++ [Str94], de ahí que tome prestado esta terminología de Simula para llamar a la referencia del objeto padre. Para saber más del operador "THIS" ver el apartado 3.7.

3.4. Operador punto

El operador punto es una de las características comunes en todos los lenguajes orientados a objetos. Por medio de este operador se pueden acceder a los atributos y métodos de un objeto.

3.4.1. Definición en MIOOP

Cuando MIOOP traduce una clase, se añade código totalmente transparente al usuario que permite la utilización de sus métodos y sus atributos (ver algoritmo 3.6).

Algoritmo 3.6 Llamada a un método a través del operador "punto"

```
FUNCTION_BLOCK Parada
  VAR_INPUT
    Consigna : REAL;
  END_VAR
  VAR
    Sensor : SensorNivel;
  END_VAR
    ...
    Sensor.UmbralDeteccion:=10;
    ...
    IF Sensor.Detectar(Consigna) THEN
    ...
END_FUNCTION_BLOCK
```

El acceso a los atributos de una clase es inmediato a través del operador punto ya que con lo que realmente está trabajando MIOOP es con una estructura. En

la norma IEC 61131 (y en la mayoría de los lenguajes de programación) el modo de acceso a las variables que se definen en una estructura se hace por medio del operador punto.

Dentro de los FBs que implementan los métodos, el acceso a los atributos de la clase es también inmediato ya que estos FB reciben por parámetro una referencia a dicha estructura.

Del mismo modo, el operador punto sirve para hacer invocaciones a métodos dentro de un código siendo el compilador el encargado de traducir dicha llamada a la invocación del FB correspondiente.

3.4.2. Traducción a IEC 61131

La llamada a un método de una clase se traduce como una llamada al FB que representa al método de la clase, pasándole la referencia de la estructura.

La función parada() del apartado anterior sería traducida a IEC 61131 como el algorítmo 3.7.

Algoritmo 3.7 Traducción de un acceso a IEC 61131 por medio del operador "punto"

```
FUNCTION_BLOCK Parada
  VAR_INPUT
    Consigna : REAL;
  END_VAR
  VAR
    Sensor : SensorNivel;
    SensorNivel_Detectar : SensorNivel_Detectar;
    //FB que implementa el método detectar
  END_VAR
    ...
    Sensor.UmbralDeteccion:=10;
    //Sensor al ser una estructura, se accede directamente
    //a sus propiedades
    ...
    IF SensorNivel_Detectar(THIS:=Sensor, Nivel:=Consigna)
    THEN
    ...
END_FUNCTION_BLOCK
```

3.5. Herencia

Cuando se construye una clase (por ejemplo la clase *"A"*), se hace atendiendo a unas necesidades concretas, pero resulta bastante común que las necesidades cambien o evolucionen, por lo que la clase *"A"* ya no serviría y se necesitaría una clase "A_1" que, manteniendo las propiedades de la clase *"A"*, se comportase como un caso particular de ésta. Ésto es, que la clase "A_1" fuese una extensión o especialización de la clase *"A"*. Ésta es una de las ventajas de la programación orientada a objeto que hace posible la reutilización de código. Para conseguir que la clase *"A_1"* cubra las nuevas y antiguas necesidades, sólo se tendrá que añadir a la clase *"A"* la funcionalidad que no posea. Esta característica se puede lograr mediante la herencia.

La herencia es el mecanismo fundamental de relación entre clases en la orientación a objetos. Relaciona las clases de manera jerárquica: una clase padre o superclase sobre otras clases hijas o subclases (ver el diagrama de clases de la figura 3.6).

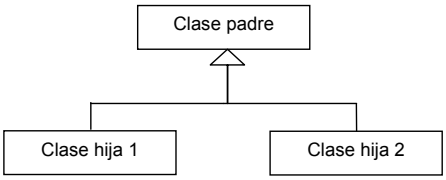

Figura 3.6: Jerarquía de clases en la herencia

Los descendentes de una clase heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables (ver modelo de protección en el apartado 3.9).

3.5.1. Definición en MIOOP

La herencia en MIOOP se define de la siguiente forma:

<NombreClase> (<Clase Base>)

Donde:

- <NombreClase> nombre de la clase que se está definiendo.

- <Clase Base> representa la clase o clases (ver apartado 3.14, herencia múltiple) de las cuales se está heredando.

Para identificar la clase de la que una clase hija hereda, se marca entre paréntesis el nombre de la clase que se hereda. Por ejemplo, dada la jearquía de clases en la que se tiene una clase base *"Letras"* y la clase *"Numeros"* que hereda de la primera (ver el algorítmo 3.8 y 3.9), se puede observar la declaración de herencia en la clase hija dentro del paréntesis.

Algoritmo 3.8 Definición de la clase *"Letras"*

```
CLASS Letras
  a : CHAR;
  b : CHAR;
  x : CHAR;
  z : CHAR;
  METHOD MétodoA() : CHAR;
END_CLASS
```

Algoritmo 3.9 Definición de la clase *"Numeros"* que hereda de *"Letras"*

```
CLASS Numeros(Letras)
  _1 : INTEGER;
  _2 : INTEGER;
  METHOD Método1() : INTEGER;
END_CLASS
```

La clase *"Numeros"* hereda todos los atributos y métodos heredables de la clase indicada entre paréntesis (ver algorítmos 3.8 y 3.9), en este caso, la clase *"Letras"*.

Supongase que todos los elementos de la clase *"Letras"* son heredables salvo los atributos *"x"* e *"y"*. Gráficamente, la clase *"Numeros"* podría representarse con una tarjeta CRC como se muestra la figura 3.7 donde se puede observar que, desde el punto de vista de un programador, la herencia se percibe como una nueva clase donde se añaden todos y cada uno de los elementos de la clase padre heredables.

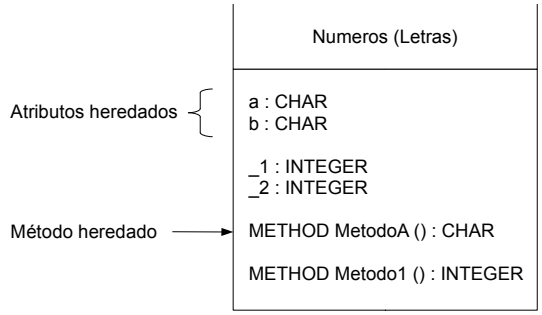

Figura 3.7: Tarjeta CRC de la clase *"Numeros"*

3.5.2. Traducción a IEC 61131

En lo referente a los atributos, para poder acceder desde la clase derivada a todos los atributos heredables de la clase base, se puede optar por dos soluciones:

1. Insertar en tiempo de compilación todos los atributos heredables de la clase base en la estructura que implementa la clase derivada. Este tipo de implementación puede verse al traducir a IEC 61131 la clase *"Numeros"* que se muestra en el algorítmo 3.9. Una vez compilada, se traduciría a la norma IEC 61131 como el código contenido en el algoritmo 3.10.

2. Insertar una referencia a la estructura que implementa la clase padre dentro de la definición de la estructura que implementa la clase hija.

   _<NombreObjetoClasePadre> : <NombreClasePadre>

   Donde:

   - _ sirve para diferenciar el nombre de una clase normal del de una clase heredada.
   - <NombreObjetoClasePadre> representa la referencia a la clase padre.
   - <NombreClasePadre> representa el nombre de la clase padre.

   La clase *"Numeros"* que se muestra en el algorítmo 3.9, una vez compilada se traduciría a la norma IEC 61131 como el código contenido en el algoritmo 3.11.

Algoritmo 3.10 Traducción de una clase con herencia en IEC 61131 por medio de la agregación de los atributos de la clase base.

```
TYPE Numeros
  STRUCT
    a : CHAR;
    b : CHAR;
    _1 : INTEGER;
    _2 : INTEGER;
  END_STRUCT;
END_TYPE
```

Algoritmo 3.11 Traducción de una clase con herencia en IEC 61131 por medio de una referencia a la clase base.

```
TYPE Numeros
  STRUCT
    _1 : INTEGER;
    _2 : INTEGER;
    _Letras : Letras;
    //Instancia a la estructura que implementa la clase
    //"Letras"
  END_STRUCT;
END_TYPE
```

La primera aproximación presenta una menor penalización en tiempo de ejecución al contener, la clase hija, los atributos de la clase padre, ya que es más rápido el acceso a una variable directamente que a través de una indirección, como es el caso de la segunda opción. La inserción de todos los atributos heredables en la estructura de la clase hija también libera al compilador de gestionar el control de acceso a los atributos no heredables de la clase padre. Sin embargo, la primera vía de implementación de la herencia obliga a duplicar los FBs que implementan los métodos heredables de la clase base en la clase hija para que los atributos de la clase hija sean accesibles desde el método heredado, lo que provoca un aumento de las lineas de código en la traducción a código estándar con la norma IEC 61131. Por otro lado, a diferencia de la informática convencional donde prima la velocidad, en el mundo de la automatización de procesos se requiere de un equilibrio entre varios aspectos:

- Ciclo de SCAN. Los programas deben de ejecutarse de forma rápida.
- Velocidad de respuesta. Los programas deben responder rápidamente a variaciones en las señales de entrada.
- Poco uso de memoria. Los programas deben ocupar poco espacio.

De esta forma, la decisión de qué tipo de traducción es más correcto para implementar la herencia se limita a un compromiso entre la velocidad de ejecución en el acceso a los atributos heredados y el tamaño de la traducción de los miembros heredables de una clase. Por tanto, en MIOOP los atributos heredables en una clase se traducen como una referencia a la clase base.

Respecto a la implementación de los métodos, para que una clase derivada pueda ejecutar los métodos de la clase base, se puede optar por dos estrategias:

1. Agregar en la clase derivada todos los métodos de las clases base. En la llamada a los métodos procedentes de la clase base se les pasaría la referencia a la estructura que implementa cada objeto de la clase derivada en vez de aquellos de la clase base. Esta solución multiplicaría el número de FBs necesarios para implementar una clase derivada, lo que aumenta innecesariamente el tamaño del código traducido a IEC 61131.
2. Otra solución sería la de utilizar los métodos de la clase base ante la invocación de un método no redefinido en la clase derivada. Para ello, en la llamada

al FB que implementa un método de la clase base, el compilador en vez de pasar como parámetro al FB la propia clase derivada, pasa la referencia a la estructura de la clase base contenida en la estructura de la derivada. Esta solución evita la multiplicación del código traducido a IEC 61131.

En el ejemplo del aparato anterior para la clase *"Numeros"* representada por la tarjeta CRC de la figura 3.7, la definición de un objeto de tipo *"Numeros"* contiene una referencia a la clase base *"Letras"*, por lo que se puede hacer una llamada al FB que implementa el método heredado, pasándole la referencia de la estructura de la clase base.

Las declaraciones de los FB que implementan los dos métodos se traduciría a la norma IEC 61131 como el FB del algorítmo 3.12 en el caso de la clase *"Letras"*. En el caso de la clase *"Numeros"* la traducción al estandar sería el FB que se muestra en el algorítmo 3.13. Ambos FBs reciben la referencia de sus respectivas clases en la variable *"THIS"* (ver apartado 3.7) que representa el objeto de la clase a la que pertenece el método.

Algoritmo 3.12 Traducción de *"MétodoA"* de la clase *"Letras"* en IEC 61131

```
FUNCTION_BLOCK MétodoA_Letras
  VAR_IN_OUT
    THIS : Letras;
  END_VAR
    //Resto del código del método
END_FUNCTION_BLOCK
```

Algoritmo 3.13 Traducción de *"Método1"* de la clase *"Numeros"* en IEC 61131

```
FUNCTION_BLOCK Método1_ Numeros
  VAR_IN_OUT
    THIS : Numeros;
  END_VAR
    //Resto del código del método
END_FUNCTION_BLOCK
```

Una llamada al método heredado *"MétodoA"* de la clase base *"Letras"* desde una instancia (objeto *"Numero"*) de la clase derivada *"Numeros"* en MIOOP sería el código que se ilustra en el algorítmo 3.14, cuya traducción a IEC 61131 produciría el codigo que se muestra en el algorítmo 3.15.

Algoritmo 3.14 Ejemplo de llamada a un método heredado

```
FUNCTION_BLOCK Llamada
  VAR
    Numero : Numeros;
  END_VAR
    Numero.MétodoA ( );
END_FUNCTION_BLOCK
```

Algoritmo 3.15 Traducción de una llamada a un método heredado

```
FUNCTION_BLOCK Llamada
  VAR
    Numero : Numeros;
    MétodoA_Letras : MétodoA_Letras;
  END_VAR
    MétodoA_Letras (Numero._Letras);
END_FUNCTION_BLOCK
```

Como se observa en el ejemplo del algorítmo 3.15, la clase *"Numeros"* adapta la llamada al método heredado, pasándole la referencia a la propia clase base. La clase *"Numeros*", al tener concatenadas todos los miembros de la clase base y de la clase derivada, puede acceder a los atributos de la clase *"Letras"* tal y como lo haría en una llamada a dicho método desde la clase base.

3.6. Operador de ámbito de acceso

Al implementar los métodos de una clase fuera de la definición de ésta, se hace necesario incorporar algún tipo de distinción que proporcione la información necesaria al compilador para determinar a qué clase pertenece cada método.

3.6.1. Definición en MIOOP

Por defecto, el convenio para el acceso a un método de una clase es acceder a aquel que esté más cercano, es decir, el método local. Si este no existe en la propia clase,

el compilador busca en los métodos definidos en las clases padres. El problema surge cuando la clase hija redefine un método de la clase padre y el programador, usando un objeto instanciado de la clase hija, desea acceder al método de la clase padre tal y como se muestra en el algorítmo 3.16.

Algoritmo 3.16 Métodos ocultados por la clase hija

```
CLASS Motor
...
  METHOD Arrancar ( ) : VOID;
  ...
END_CLASS

CLASS MotorElectrico (Motor)
  //Clase heredada de Motor
  ...
  METHOD Arrancar ( ) : VOID;
  METHOD Encendido ( ) : VOID;
  ...
END_CLASS
```

Según esta definición, si desde el método *"Encendido"* (ver algorítmo 3.17) se llama al método *"Arrancar"*, el compilador sobreentiende que se intenta invocar al método local de la clase, con lo que sería imposible hacer una llamada al método *"Arrancar"* de la clase padre (clase *"Motor"*) ya que por defecto un compilador siempre da prioridad a las variables y métodos más cercanos, es decir, a los locales.

Algoritmo 3.17 Acceso a un método local que oculta al del padre

```
METHOD Encendido ( ) : VOID;
  Arrancar ( );
  //Esta llamada oculta al método Arrancar de la clase
  //padre, es decir llama al propio método de la clase
END_METHOD
```

Este problema se resuelve en lenguajes como JAVA o los lenguajes de .NET por medio del operador *"SUPER"* que indica al compilador que al método que se desea acceder es al de la clase padre, pero esta solución sólo es útil para lenguajes orientados a objetos que únicamente permiten herencia simple. MIOOP soporta herencia múltiple y por tanto, un operador *"SUPER"* se quedaría corto para resolver la

pertenencia de un método a dos o más clases padre.

Existen varios lenguajes que permiten la herencia múltiple, como Eiffel, Clos, C++, etc, y cada de ellos resuelve la ambigüedad del ámbito de los métodos de una forma diferente. MIOOP sigue el estilo marcado por Stroupstrup en el diseño de C++ y amplia la norma IEC 61131 por medio del operador *"::"*. Este operador de ámbito es el medio que tiene MIOOP para identificar en donde están definidos los componentes de una clase; por tanto, su uso es necesario cuando alguno de estos componentes se utiliza fuera del ámbito de la clase.

El operador de ámbito proporciona al compilador toda la información para conocer la pertenencia de la definición de un método a una clase o resolver una llamada a un método definido en una clase padre.

La sintaxis del ámbito de un método por medio de este operador se define como:

<NombreClase>::<NombreMétodo>

Donde:

- <NombreClase> identifica a la clase a la que pertenece el método.
- :: representa el operador de acceso de ámbito que indica al compilador a qué clase pertenece el método que se está accediendo.
- <NombreMétodo> identifica el nombre del método definido en la clase.

De esta manera, la definición de las cabeceras de los métodos sería:

```
METHOD MotorElectrico::Encendido ( ) : VOID;
```

Y la llamada ambigüa del algorítmo 3.17, en el caso de querer llamarse al método definido en la clase padre sería:

```
Motor::Arrancar();
```

3.6.2. Traducción a IEC 61131

El operador *"::"* no posee una traducción directa a IEC 61131. Este operador dota de información al compilador para resolver las llamadas ambiguas a métodos de clases de las que se herede, de tal manera que se invoque al FB correcto de la clase en cada llamada.

Por ejemplo, dada la jerarquía formada por las clases *"Motor"* y *"MotorElectrico"* definidas anteriormente en el algoritmo 3.16 y representadas gráficamente en la figura 3.8, la llamada al método *"Arrancar"* de la clase padre por medio del uso del operador de ámbito, se realizaría como se indica en el algorítmo 3.18. En tiempo de compilación, el algorítmo 3.18 se traduciría a IEC 61131 como una llamada al FB que implementa el método "*Arrancar"* de la clase *"Motor"* como se muestra en el algorítmo 3.19.

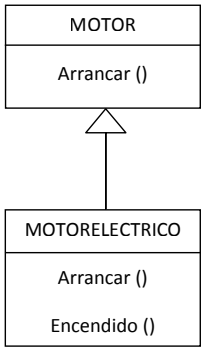

Figura 3.8: Jerarquía de clases de motores

Algoritmo 3.18 Llamada al método Arrancar de la clase MotorElectrico

```
METHOD MotorElectrico::Encendido ( ) : VOID;
  Motor::Arrancar ( );
  //Llamada al método de la clase padre
END_METHOD
```

Algoritmo 3.19 Traducción a IEC 61113 de la llamada al método Arrancar de la clase MotorElectrico

```
FUNCTION_BLOCK MotorElectrico_Encendido
  VAR_IN_OUT
    THIS : MotorElectrico;
    Motor_Arrancar : Motor_Arrancar;
    //Método invocado de la clase Motor
  END_VAR
    Motor_Arrancar (THIS._Motor);
    //Llamada al método correcto, el de la clase padre,
    //sin ambigüedad
END_FUNCTION_BLOCK
```

3.7. Operador THIS

El operador *"THIS"* representa una referencia a la instancia de una clase, de tal manera, que todo acceso a un atributo de *"THIS"* equivaldría a un acceso a un atributo del objeto al que referencia.

3.7.1. Definición en MIOOP

El acceso a los atributos de una clase a través de sus métodos se hace simplemente escribiendo el nombre del atributo. Esta sencilla forma de acceso a los atributos de una clase resulta inoperativa al traducir el código OO a IEC61131, ya que los atributos de la clase son implementados por una estructura y por definición, el acceso a estos atributos se realiza por medio del operador punto.

En lenguajes como JAVA o o los de .NET, el operador *"THIS"* sirve para que el compilador resuelva ambigüedades en las llamadas a un atributo de una clase desde un método que posee variables locales que coinciden en nombre con los atributos de la clase. En MIOOP, esta definición varia un poco, ya que esta ambigüedad se resuelve por medio del operador *"::"*. De esta forma, en MIOOP el operador *"THIS"* (que a su vez es una palabra reservada) es una referencia a la propia instancia del objeto que recibe el método de la clase traducido a IEC 61131 para poder así, acceder a los atributos de la clase y distinguirlo de una variable u objeto creado por el propio programador dentro del método. Para entender este detalle, en el algoritmo 3.20 se muestra que ocurriría si al traducir un método de la clase

"Motor" a IEC 61131 se usara el mismo nombre que la clase para la referencia a la estructura en un método que ya poseee un objeto con dicho nombre.

Algoritmo 3.20 ambigüedad en la traducción si no se usa el operador *"THIS"*

```
METHOD Motor::Diagnosticar ( ) : VOID;
  VAR
    Motor:Motor;
    //Instancia a otro motor diferente
  END_VAR
    ...
END_METHOD

FUNCTION_BLOCK Motor_Diagnosticar
  VAR_IN_OUT
    Motor : Motor;
    //Referencia a la estructura de la clase
  END_VAR
  VAR
    Motor : Motor; //ERROR
    //Instancia al nuevo motor. Nombre ambigüo
  END_VAR
    ...
END_FUNCTION_BLOCK
```

La definición de este operador de ámbito se hace como:

<THIS>.<NombreAtributo>

Donde:

- <THIS> indica al compilador que se pretende acceder a un atributo de la clase y no a una variable local del método.
- <NombreAtributo> identifica a un atributo perteneciente a la clase que define el método.

Para facilitar la labor del ingeniero programador, el acceso a los atributos de la propia clase a través de uno de sus métodos no requiere obligatoriamente el uso del operador *"THIS"* a no ser que se pueda producir una colisión ambigüa con una variable u objeto local al propio método. De este forma, si no existe ambigüedad,

el compilador entiende en tiempo de compilación que una variable no declarada dentro del método es un atributo de la propia clase realizando las traducciones necesarias para acceder a dicho atributo.

3.7.2. Traducción a IECC 61131

Todo método de una clase en MIOOP es traducido como un FB cuyo primer parámetro de entrada es una referencia a la estructura que implementa la clase. Esta referencia es etiquetada en tiempo de compilación por medio del operador *"THIS"*. Para entender mejor esta traducción se utilizará como ejemplo la clase ya definida *"SensorNivel"* y la implementación de su método *"Detectar"* tal y como se muestra en el algorítmo 3.21. En tiempo de compilación, el algorítmo 3.21 se traduce al algoritmo 3.22.

Algoritmo 3.21 Implementación del método Detectar

```
METHOD SensorNivel::Detectar (Nivel : REAL) : BOOL;
  VAR
     Estado : BOOL;
  END_VAR
     Estado:=false;
     //Acceso a la variable local
     THIS.Estado:=true;
     //Acceso campo Estado de la estructura que contiene
     //la instancia SensorNivel
END_METHOD
```

3.8. Utilización de un objeto en MIOOP

Para definir una clase en MIOOP se debe:

- Crear un nuevo POU clase, identificándolo con un nombre único dentro del proyecto.
- Definir los atributos y métodos que constituyen la clase.
- Programar cada uno de los métodos definidos en la clase.

Algoritmo 3.22 Traducción del método Detectar a IEC 61131

```
FUNCTION_BLOCK SensorNivel_Detectar
  VAR_INPUT
    Nivel : REAL;
  END_VAR
  VAR_IN_OUT
    THIS : SensorNivel;
  END_VAR
  VAR_OUTPUT
    Retorno : BOOL;
  END_VAR
  VAR
    Estado : BOOL;
  END_VAR
    Estado:=false;
    //Acceso a la variable local
    THIS.Estado:=true;
    //Acceso al atributo de la clase SensorNivel
END_FUNCTION_BLOCK
```

Una vez definida una clase, puede ser instanciada (ver figura 3.9) y utilizada desde cualquier POU y en cualquiera de los 5 lenguajes definidos en el estándar IEC 61131. Tanto para acceder a los atributos del objeto como a sus métodos se empleará el operador *"punto"* (ver figura 3.10 y 3.11).

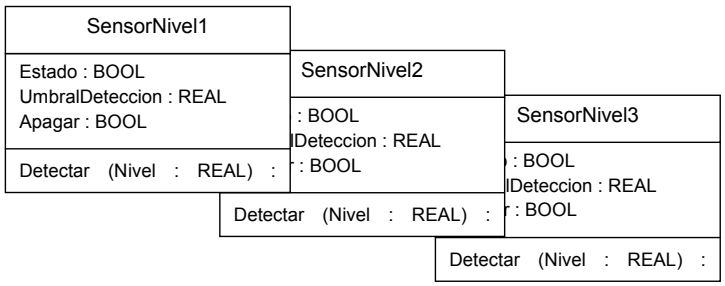

Figura 3.9: Ejemplo de instancias de una clase

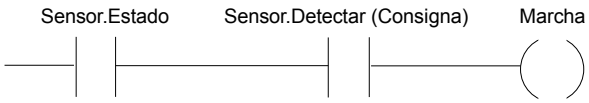

Figura 3.10: Ejemplo de llamada a un método desde un ejemplo en LD

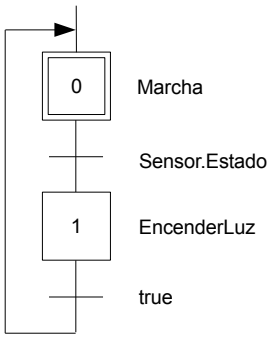

Figura 3.11: Ejemplo de consulta del valor de un atributo del objeto en GRAFCET

3.9. Modelo de protección

El modelo de protección es un mecanismo que permite definir quien puede o no acceder a los atributos y métodos de una clase, es decir, define los niveles de acceso para elementos de esa clase. Existen cuatro niveles de acceso:

- Público: Se permite el acceso a todos los elementos definidos en una clase desde el exterior de ésta, es decir, desde otro POU independiente de la clase. Éste es el nivel de protección más bajo.
- Protegido: El acceso a los elementos de la clase está restringido a los métodos de las clases heredadas, es decir, los métodos de esa clase y todas las subclases.
- Amigo: Permite el acceso a los elementos de la clase por parte de una entidad externa que se declara como amiga como si se tratase de un método propio de la clase.
- Privado: El acceso a los elementos de la clase está restringido a los métodos de esa clase solamente. Éste es nivel más alto de protección y el compilador

prohíbe el acceso a los elementos de la clase a partir de una instancia de ésta y el operador *"."*.

3.9.1. Definición en MIOOP

En MIOOP, el acceso a los atributos de una clase por parte de los métodos de la misma no tiene restricción alguna. Esta propiedad cambia cuando se habla de visibilidad de los miembros de una clase desde fuera del ámbito de ésta. Por defecto y si no se indica nada, MIOOP declara los atributos de un objeto como inaccesibles o invisibles desde fuera del ámbito de la misma, mientras que sus métodos son completamente accesibles desde otros objetos o rutinas. Aunque esta regla resulta útil para no tener que indicar el nivel de protección de cada uno de los atributos y métodos de una clase, las buenas prácticas de programación aconsejan siempre indicar el nivel de protección de cada miembro de una clase de forma explícita por medio de un modificador específico. Ésto permite además manterner el caracter fuertemente tipado y extricto del estandar IEC 61131. Por este motivo, en los ejemplos que se detallan a continuación, siempre se indica el nivel de protección de todos los componentes de las clases.

La definición del modelo de protección depende del modificador de acceso que acompañe a la declaración de cada atributo y de cada método. Dependiendo del tipo de acceso especificado, el compilador debe tomar distintas decisiones respecto al acceso que permíte los atributos y métodos definidos en una clase. Los tipos de modificadores de acceso definidos en MIOOP son:

1. Acceso público, definido con la palabra reservada *"PUBLIC"*.
2. Acceso protegido, definido con la palabra reservada *"PROTECTED"*.
3. Acceso amigo, definido con la palabra reservada *"FRIEND"*.
4. Acceso privado, definido con la palabra reservada *"PRIVATE"*.

En el algorímo 3.23 se muestra un ejemplo del modelo de protección de la clase *"SensorNivel"*.

Con respecto a la herencia, los miembros de una clase que son declarados como privados no son marcados como heredables y por tanto, son inaccesibles en la jerarquía de clases. El resto modificadores de ámbito, permiten que los miembros de la clase con los que son etiquetados sean declarados heredables.

Algoritmo 3.23 Definición de niveles de acceso en una clase

```
CLASS SensorNivel
  PUBLIC Estado : BOOL:=false;
  //Atributo público
  PUBLIC UmbralDeteccion : REAL:=10;
  //Atributo público
  PRIVATE Apagado : BOOL:=false;
  //Atributo privado
  PROTECTED Reiniciado : BOOL:=true;
  //Atributo protegido
  PUBLIC METHOD Detectar (Nivel : REAL) : BOOL;
  //Método público
  PRIVATE METHOD VerApertura () : REAL;
  //Método privado
END_CLASS
```

PUBLIC

La declaración sería:

PUBLIC <Nombre atributo/método> : <Tipo de dato>;

Donde:

- PUBLIC es la palabra reservada que identifica que un miembro de una clase puede ser libremente accedido.
- <Nombre atributo/método> es el nombre del atributo o método de la clase que puede ser accesible desde cualquier ámbito.
- <Tipo de dato> identifica al tipo de dato del atributo o el tipo de dato devuelto por un método.

En el algoritmo 3.24 se muestra un ejemplo de acceso a los atributos de una clase, uno de forma correcta y otro de forma erronea.

PRIVATE

La declaración sería:

Algoritmo 3.24 Ejemplo de acceso a un miembro público

```
FUNCTION_BLCOK LlamadaMiembroPublico
  VAR
    Sensor : SensorNivel;
  END_VAR
    ...
    Sensor.UmbralDeteccion:=5; //OK
    ...
    Sensor.Apagado:=false; //ERROR. Atributo inaccesible
END_FUNCTION_BLOCK
```

PRIVATE <Nombre atributo/método> : <Tipo de dato >;

Donde:

- PRIVATE es la palabra reservada que identifica que un miembro de una clase solo puede ser accedido a través de un miembro de la misma clase.
- <Nombre atributo/método> es el nombre del atributo o método de la clase que es no es accesible desde otra clase o instancia.
- <Tipo de dato> identifica al tipo de variable que define a un atributo o el tipo de valor devuelto por un método.

En el algorítmo 3.24 se ilustra la imposibilidad de acceder a un elemento declarado privado desde cualquier código que no este encapsulado dentro de los márgenes definidos por la clase.

PROTECTED

Un elemento de la clase protegido se comporta de manera parecida a un miembro privado, salvo que estos son accesibles dentro de la clase que lo posee y desde las clases derivadas, pero no desde los objetos instanciados a raíz de dichas clases.

La declaración sería:

PROTECTED <Nombre atributo/método> : <Tipo de dato>;

Donde:

- PROTECTED es la palabra reservada que identifica que a un elemento protegido.
- <Nombre atributo/método> es el nombre del atributo o método a proteger.
- <Tipo dato> identifica al tipo de variable que define a un atributo o el tipo del dato devuelto por un método.

Para entender el funcionamiento de este nivel de protección, se describe la clase *"SensorNivelUltrasonico"* que es una especialización de la clase *"SensorNivel"* heredando todos sus métodos y atributos. Desde cualquier método de esta nueva clase, los miembros protegidos de la clase *"SensorNivel"* son públicos, sin embargo, estos miembros son privados si se intenta acceder a ellos desde una entidad externa (ver algorítmo 3.25).

Algoritmo 3.25 Ejemplo de acceso a un miembro protegido

```
METHOD SensorNivelUltraSonido::ReiniciarSeñal
   ...
   Reiniciado := true; //OK
   //Atributo heredado y accesible por ser protegido
   ...
   Apagado:=false; //ERROR
   //Atributo inaccesible de la clase padre y no
   //heredado por ser privado
END_METHOD

FUNCTION_BLOCK LlamadaMiembroProtegido
  VAR
     SensorUltraSonido : SensorNivelUltraSonido;
  END_VAR
     ...
     SensorUltraSonido.Reiniciado:=false; //ERROR
     //Atributo inaccesible
END_FUNCTION_BLOCK
```

FRIEND

El modificador *"FRIEND"* puede aplicarse a clases, POUs estándar de la norma IEC 61131 o a métodos de otra clase, para inhibir el sistema de protección, es decir, para que las clases y funciones declaradas como amigas, puedan tener acceso a todos los elementos de la clase como si estos fueran públicos. Esta declaración de amistad debe ser declarada dentro de la clase, definiendo explícitamente el acuerdo de amistad con el POU, clase o método externo. Las relaciones de *"amistad"* entre clases son parecidas a las amistades entre personas:

- La amistad no puede transferirse. Si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C.
- La amistad no puede heredarse. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C.
- La amistad no es simétrica. Si A es amigo de B, B no tiene por qué ser amigo de A.

La declaración sería:

FRIEND <Tipo de POU> <Nombre> ( ) : <Valor devuelto>;

Donde:

- FRIEND es la palabra reservada que identifica una relación de amistad con la clase que la declara miembro.
- <Tipo de POU> identifica el ámbito del contrato de amistad que se suscribe.
- <Nombre> es el nombre del FB, método o clase con el que se suscribe el contrato de amistad. Si la relación de amistad es con el método de una clase, el nombre se divide en el de la clase y métodos amigos separados por "::".

  <NombreClase>::<NombreMétodo>

POU FRIEND

El caso más sencillo es el de una relación de amistad con un POU de la norma IEC 61131. Este mecanismo permitiría que una función, un FB o un programa

puediesen acceder a todos los miembros de un objeto a partir del POU *"amigo"*. Este POU *"amigo*" se declara en la propia clases y su implementación se desarrolla fuera de la misma, teniendo acceso a todos los atributos y métodos de la clase de igual manera que si se tratase de un método propio de la clase (ver algorítmo 3.26).

Algoritmo 3.26 Definición de POUs amigos en una clase

```
CLASS ActuadorNeumatico( )
  PRIVATE MaximaApertura : REAL:=50;
  //Atributo privado que será accedido
  PRIVATE PorApertura : INTEGER:=0;
  PUBLIC Movimiento : BOOL:=false;
  PUBLIC METHOD Abrir (Cantidad : INTEGER) : VOID;
  PUBLIC METHOD Cerrar (Cantidad : INTEGER) : VOID;
  PUBLIC METHOD Estado () : INTEGER;
  FRIEND VerApertura (Actuador : ActuadorNeumatico) : REAL;
  //POU amigo
END_CLASS
```

Una vez definida la clase donde se estable el contrato de amistad, se implementa el POU amigo (en este caso, un FB) que tiene acceso a toda la clase *"Actuador-Neumático"* tal y como se puede observar en el algorítmo 3.27.

Algoritmo 3.27 Implementación de un POU amigo

```
FUNCTION_BLOCK VerApertura
  VAR_IN_OUT
     Actuador : ActuadorNeumatico;
  END_VAR
  VAR_OUTPUT
     ValorApertura : REAL;
  END_VAR
    RETURN Actuador.MaximaApertura;
    //OK. Acceso permitido a un atributo privado
    //de la clase
END_FUNCTION_BLOCK
```

Métodos FRIEND

Este tipo de relación es algo más compleja que la de los POU *"amigos"*. En este caso, la clase que subscribe el contrato de amistad proporciona acceso público a

una clase externa a través de uno o varios de sus métodos que se declaran *"amigos"*.

La declaración de amistad de un método se define en la clase que desea permitir libre acceso a sus elementos, pero la implementación de este método corresponde a la clase externa.

Para liberar de ambigüedades al compilador, es necesario indicar la clase a la que pertenece cada método *"amigo"* por medio del operador de ámbito *"::"* como se muestra en el algoritmo 3.28.

Algoritmo 3.28 Definición de un método FRIEND

```
CLASS ActuadorNeumatico( )
  PRIVATE MaximaApertura : REAL:=50;
  PRIVATE PorApertura : INTEGER:=0;
  PUBLIC Movimiento : BOOL:=false;
  PUBLIC METHOD Abrir (Cantidad : INTEGER) : VOID;
  PUBLIC METHOD Cerrar (Cantidad : INTEGER) : VOID;
  PUBLIC METHOD Estado () : INTEGER;
  FRIEND METHOD SensorNivel::VerApertura (Actuador
        : ActuadorNeumatico)  : REAL;
  //Método FRIEND de la clase SensorNivel
END_CLASS
```

En el algoritmo 3.29 se muestra la declaración de la clase que posee el método de la clase *"ActuadorNeumatico"* como *"FRIEND"*. En el algoritmo 3.30 se muestra la implementación del método *"VerApertura"* en MIOOP. Finalmente, en el algorítmo 3.31 se muestra un ejemplo de una functión en la que se utiliza el método *"amigo"* para el acceso a los elementos de la clase *"ActuadorNeumatico"*.

Algoritmo 3.29 Clase que implementa el método FRIEND a otra clase

```
CLASS SensorNivel( )
  PUBLIC Estado : BOOL:=false;
  PUBLIC UmbralDeteccion : REAL:=10;
  PRIVATE Apagar : BOOL:=false;
  PUBLIC METHOD Detectar (Nivel : REAL) : BOOL;
  PUBLIC METHOD VerApertura (Actuador
        : ActuadorNeumatico) : REAL;
END_CLASS
```

Algoritmo 3.30 Implementación del método FRIEND *"VerApertura"*

```
METHOD SensorNivel::VerApertura(Actuador
        : ActuadorNeumatico)
  RETURN Actuador.MaximaApertura;
  //OK. Acceso permitido a un atributo privado
END_METHOD
```

Algoritmo 3.31 Ejemplo de utilización de un método FRIEND

```
FUNCTION_BLOCK Consultar
  VAR
    Actuador : ActuadorNeumatico;
    Sensor : SensorNivel;
    Apertura : REAL;
  END_VAR
    Apertura:=Sensor.VerApertura (Actuador);
    //Al ser un método amigo accede a todos los elementos
    //del objeto
    Apertura:=Actuador.VerApertura(); //OK
END_FUNCTION_BLOCK
```

El único método de la clase *"SensorNivel"* que tiene un contrato de amistad es *"VerApertura"* y por tanto, posee libre acceso a todos los elementos de la clase *"ActuadorNeumático"*. Si se intentase un acceso a alguno de los elementos privados de la clase *"ActuadorNeumatico"* por parte de otro método de la clase *"SensorNivel"* con el que no se tiene amistad, el compilador mostraría un error de acceso a un elemento privado.

Clases FRIEND

Una clase declarada como *"amiga"* permite que ésta tenga acceso a todos los elementos de la clase que subscribe el contrato. Es decir, desde cualquiera de los métodos de la clase *"amiga"* se puede operar con cualquiera de los elementos de la clase que define la amistad (ver algorítmo 3.32). La declaración de la clase *"FRIEND"* se muestra en el algorítmo 3.33 y en el algoritmo 3.34 se puede observar la implementación de uno de sus métodos en el que se puede ver cómo se tiene acceso a todos los elementos de la clase *"ActuadorNeumatico"*.

Algoritmo 3.32 Definición de una clase FRIEND dentro de otra clase

```
CLASS ActuadorNeumatico( )
  PRIVATE MaximaApertura : REAL:=50;
  PRIVATE PorApertura : INTEGER:=0;
  PUBLIC Movimiento : BOOL:=false;
  PUBLIC METHOD Abrir (Cantidad : INTEGER) : VOID;
  PUBLIC METHOD Cerrar (Cantidad : INTEGER) : VOID;
  PUBLIC METHOD Estado () : INTEGER;
  FRIEND CLASS SensorNivel;
  //Clase amiga de ActuadorNeumatico
END_CLASS
```

Algoritmo 3.33 Definición de la clase declarada FRIEND

```
CLASS SensorNivel( )
  PUBLIC Estado : BOOL:=false;
  PUBLIC UmbralDeteccion : REAL:=10;
  PRIVATE Apagar : BOOL:=false;
  PUBLIC METHOD Detectar (Nivel : REAL) : BOOL;
END_CLASS
```

Algoritmo 3.34 Ejemplo de utilización de una clase FRIEND

```
METHOD SensorNivel::Detectar (Nivel : REAL) : BOOL;
  RETURN Actuador.Movimiento;
  //OK. Acceso permitido a un atributo privado de la
  //clase ActuadorNeumatico
END_METHOD
```

3.9.2. Traducción a IEC 61131

El modelo de protección no genera código adicional producir la traducción a código de IEC 61131 desde el modelo OO original. Este mecanismo es utilizado por el compilador para crear una jerarquía de acceso a los elementos de las clases, de tal manera que un acceso no permitido aborte la traducción y muestre el pertinente error.

3.9.3. Modelo de protección en la herencia

El modelo de protección permite a una clase definir que atributos y métodos son heredables para sus clases derivadas.

Por ejemplo, dada la clase *"Tolva"* y la clase *"TolvaConCalentador"* que hereda de la primera (ver algorímos 3.35 y 3.36). El modelo de protección asegura que sólolos atributos y métodos públicos y protegidos de la clase *"Tolva"* son heredados por la clase *"TolvaConCalentador*" que podría representarse gráficamente como se muestra en la tarjeta CRC de la figura 3.12.

3.10. Constructor y Destructor

El constructor de una clase es un método estándar para inicializar sus objetos que se ejecuta siempre al crear un objeto siempre y cuando el programador así lo necesite como se verá más adelante. El nombre de los constructores suele coincidir con el nombre de la clase a la que pertenecen y no suelen proporcionar ningún valor devuelto. Su función principal es la de brindar al usuario un instrumento que le permita ejecutar las acciones que desee justo a continuación de ser creada una instancia y antes de que se ejecute ningún método.

Algoritmo 3.35 Definición de la clase *"Tolva"*

```
CLASS Tolva
  PUBLIC Lleno : BOOL;
  PUBLIC Vacio : BOOL;
  PROTECTED nivel : REAL;
  PRIVATE Limpio : BOOL;
  PUBLIC METHOD LlenarTolva ();
  PUBLIC METHOD EstaLlena () : BOOL;
  PUBLIC METHOD ParарLlenado ();
  PRIVATE METHOD Recircula ();
END_METHOD
```

Algoritmo 3.36 Definición de la clase *"TolvaConCalentador"*

```
CLASS TolvaConCalentador (Tolva)
  PUBLIC Calentador : REAL;
  PRIVATE Temperatura : REAL;
  PUBLIC METHOD ArrancarCalentador ();
  PUBLIC METHOD ParарCalentador ();
  PUBLIC METHOD GetTemperatura () : BOOL;
  PRIVATE METHOD CalcularTemperatura ();
END_METHOD
```

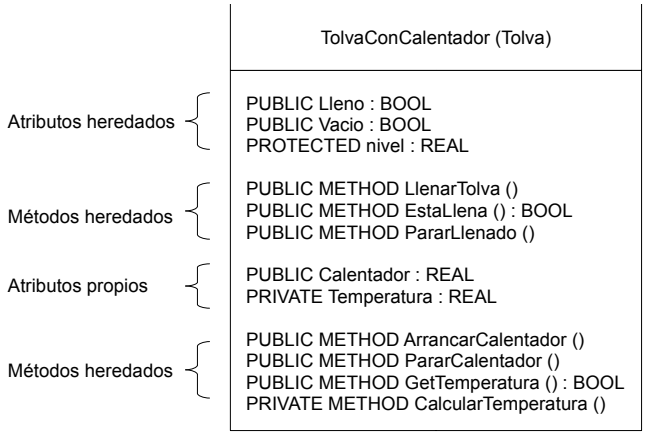

Figura 3.12: Esquema de la clase *"TolvaConCalentador"* en formato CRC

El destructor es un método que se invoca automáticamente cuando el objeto se destruye, es decir, al finalizar la ejecución del POU que instancia el objeto. Su principal función, al igual que la de los constructores, es la de dotar al usuario de un mecanismo que le permita ejecutar las acciones que desee a continuación de dejar existir un objeto.

3.10.1. Definición en MIOOP

El constructor, por defecto, se ejecuta en primer lugar cuando un objeto instancia una clase. Es decir, la primera instrucción que se añade a un POU que instancia una clase es una llamada al constructor de dicha clase. Esta línea es añadida por el compilador en tiempo de compilación. Esta acción funciona perfectamente cuando se trata de objetos dinámicos que se crean en un momento determinado. En este caso, una instrucción especial desencadena la reserva de memoria del objeto y el compilador añadiría la llamada al constructor. MIOOP se centra en sistemas estáticos que son los que reconoce la norma IEC 61131 y por tanto, hay que decidir en que punto se debe invocar al constructor. La solución pasaría por llamar una única vez al constructor en el primer lugar donde este se instancie y que el compilador lleve la cuenta para evitar que se llame más de una vez.

El destructor de un objeto es la última línea que se ejecuta de un POU que instancia una clase.Es decir, en tiempo de compilación, el traductor añade al final del código de dicho POU una llamada al destructor de cada objeto que se ha instanciado en él. El problema de los destructores es mayor que el de los constructores ya que no tiene sentido que un sistema estático elimine instancias de objetos y por tanto, jamas se invocaría el destructor. De este modo, MIOOP desarrolla el mecanismo del destructor pero deja su utilización para futuras investigaciones en el campo de sistemas dinámicos sobre OO.

Para poder usar el constructor y destructor de forma análoga a como se haría en un sistema dinámico habría que poder declarar objetos dentro de una función pero esta posibilidad esta prohibida dentro de la norma IEC 61131 y solo es posible instanciar FBs y por ende, métodos, dentro de otro FB o de un programa.

El mecanismo de constructores y destructores no es nuevo. Es una adaptación del mecanismo de inicialización de clases de Simula67 [DJ67], permitiendo introducir código en ambos por parte del programador.

El nombre, tanto del constructor como el destructor, coincide con el nombre de la clase a la que pertenecen. La diferencia radica en el prefijo "~" que sirve para distinguir al destructor del constructor.

El constructor se identifica como un método con el nombre de la clase a la que pertenece, sin indicar si este es público o privado. La definición del constructor sería de la forma:

<NombreClase> (<ParametrosEntrada>)

Donde:

- < NombreClase> identifica que el método declarado es un constructor.
- (<ParametrosEntrada>) identifica la lista de parámetros que recibe el constructor.

Para definir el destructor se introduce un método sin parámetros de entrada ni de salida, identificándolo con el nombre de la clase a la que pertenece precedida del identificador "~". La definición del destructor seguiría la etiqueta:

~<NombreClase>

Donde:

- ~<NombreClase> El símbolo "~" sirve para identificar que el método es un destructor.

Un ejemplo del uso de constructores y destructores se ilustra en el algorítmo 3.37.

Algoritmo 3.37 Ejemplo de utilización del constructor y destructor

```
CLASS SensorNivel
  METHOD SensorNivel () : VOID;  //Constructor por defecto
  METHOD ~SensorNivel () : VOID;  //Destructor de la clase
  ...
END_CLASS
```

MIOOP soporta la sobrecarga de constructores, pero sólo permite la ejecución de uno en cada instancia. Por defecto, si el programador no indica otro constructor,

el sistema ejecuta aquel que menos parámetros de entrada tenga. Sin embargo, MIOOP sólo permite la definición de un único destructor.

Tanto el constructor como el destructor pueden ser programados por parte del usuario, pero si no se define ningún constructor o destructor, el compilador no incorporaría ninguna llamada automática a estos en los POUs que instancian las clases.

3.10.2. Traducción a IEC 61113

Tanto el constructor como el destructor de una clase son métodos y por tanto, su traducción es similar a la de cualquier otro método de una clase.

En el caso de los constructores su definición sería:

<NombreClase>_<NombreClase>

Donde:

- <NombreClase> identifica a la clase a la que pertenece el método implementado a través del FB.
- _< NombreClase> identifica que el método definido es especial, en este caso un constructor.

La traducción del destructor seguría la definición:

~<NombreClase>_<NombreClase>

Donde:

- ~<NombreClase> identifica a la clase a la que pertenece el método implmentado a través del FB. El símbolo "~" sirve para identificar que el método es un destructor.
- _< NombreClase> identifica que el método definido es especial.

3.11. Conversión de tipos o *"casting"*

Las conversiones de tipos de datos permiten que una variable o resultado de evaluar una expresión de un tipo, sea tratada como una variable o el resultado de evaluar una expresión de otro tipo. En algunos lenguajes como C, dichas conversiones pueden ser implícitas, cuando se realizan de forma automática, o explícitas, en las que es necesario que el programador indique la conversión deseada (ver algorimo 3.38).

La norma IEC 61131 obliga a que todas las conversiones sean explícitas debido a su caracter tan conservador. Estas conversiones se hacen por medio de funciones que proporciona el propio estandar y aseguran que el usuario es consciente de las perdidas de precisión que se pueden perder al hacer una conversión de tipos.

Algoritmo 3.38 Ejemplo de conversión implícita y explícita

```
FUNCTION Ejemplo
  VAR
    entero : INTEGER;
  END_VAR
    entero:=3.5; //Conversión implícita
    entero:=REAL_TO_UINT(3.5); //Conversión explícita
END_FUNCTION
```

3.11.1. Definición en MIOOP

Al igual que la norma IEC 61131, por motivos de seguridad, MIOOP define todas las conversiones como explicitas. La sintaxis propuesta por MIOOP sería la siguiente:

<NombreVariableConvertida> := (<NuevoTipo>) <NombreVariable_A_Convertir>

ó

<NombreExpresiónConvertida> := (<NuevoTipo>) <NombreExpresión_A_Convertir>

Donde:

- <NombreVariableConvertida> | <NombreExpresiónConvertida> es la variable o expresión donde se va a alojar el dato una vez convertido.

- (<NuevoTipo>) es el tipo de la variable identificada por <NombreVariable>. El tipo va siempre entre paréntesis para permitir al compilador distinguirlo de una variable común.

- <NombreVariable_A_Convertir> | <NombreExpresión_A_Convertir> es la variable o expresión que contiene el dato que se desea convertir.

3.11.2. Traducción a IEC 61131

La norma IEC 61131 sólo recoge la conversión de tipos explícita por medio de funciones como por ejemplo, REAL_TO_UINT. Estas funciones son proporcionadas por el entorno de programación del PLC y sigue la síntaxis: *<valor a convertir> _TO_ <valor resultado> (<Variable>)* (ver algorítmo 3.39).

Algoritmo 3.39 Ejemplo de conversión de tipos de IEC 61131

```
FUNCTION Conversion
  VAR
    Entero : INTEGER;
    Decimal : REAL:=10.5;
  END_VAR
    Entero:=REAL_TO_UINT(Decimal);
END_FUNCTION
```

MIOOP define la misma síntaxis para todas las conversiones, tanto de tipos de datos simples como entre objetos (ver apartado 3.12). De ésta manera, no es preciso conocer las funciones de conversión de IEC 61131. Automáticamente, al programar un *"casting"* entre tipos simples de datos, el compilador las traduce a las correspondientes funciones de conversión de IEC 61131.

Por ejemplo, la función *"Conversion"* del apartado anterior (ver algoritmo 3.40), al compilarse se traduciría como muestra el algorítmo 3.41.

Algoritmo 3.40 Conversión de tipos simples según MIOOP

```
FUNCTION Conversion
  VAR
    Entero : INTEGER;
    Decimal : REAL;
  END_VAR
    Entero:=10;
    Decimal:=5.4;
    Decimal:=(REAL) Entero;
    //Decimal contiene el valor 10.0
    Entero:=20;
    Entero:=(INTEGER) Decimal;
    //Entero contiene el valor 10
END_FUNCTION
```

Algoritmo 3.41 Traducción a IEC 61131 de la conversión de tipos simples

```
FUNCTION Conversion
  VAR
    Entero : INTEGER;
    Decimal : REAL;
  END_VAR
    Entero:=10;
    Decimal:=5.4;
    Decimal:=INT_TO_REAL (Entero);
    //Decimal contiene el valor 10.0
    Entero:=20;
    Entero:=REAL_TO_INT (Decimal);
    //Entero contiene el valor 10
END_FUNCTION
```

3.12. Casting entre objetos

El casting entre clases funciona de forma análoga a la conversión de tipos básicos de datos pero aplicado a clases que guardan una relación de herencia. Es un mecanismo por el cual, un objeto de *"A"* pase a ser un objeto de *"B"* siempre y cuando exista una relación de herencia entre ambas. En caso contrario, el compilador mostrará un mensaje de error.

Es de resaltar que en el proceso de conversión puede haber perdida de precisión si la clase que se desea convertir es *"mayor"* que la clase destino a la que se quiere transformar. Es decir, ésto suele ocurrir cuando la clase convertida es la clase padre en una jerarquía de herencia.

3.12.1. Definición en MIOOP

Cuando un programador escribe un casting entre objetos que tienen una relación de herencia, MIOOP añade en cada clase, de forma totalmente transparente al ingeniero de automatización, un método que se traducirá como un FB que recibe por parámetro los objetos origen y destino de la conversión. Dicho método se inserta en la clase destino de la conversión y realiza asignaciones de los atributos del objeto origen al objeto resultado. El nombre del FB lleva la etiqueta:

> Casting_<ClaseTipoDestino>_<ClaseTipoOrigen> (<ObjetoTipoDestino>, <ObjetoTipoOrigen>)

Donde:

- Casting identifica a un FB de casting.
- _<ClaseTipoDestino> representa la clase que contendrá el resultado de la conversión.
- _<ClaseTipoOrigen> representa la clase del objeto origen que se desea convertir.
- <ObjetoTipoDestino> representa una instancia del objeto resultado de la conversión.

- <ObjetoTipoOrigen> representa una instancia del objeto origen que se desea convertir.

Un ejemplo del casting entre clases se muestra en el algorítmo 3.42.

Algoritmo 3.42 Ejemplo de casting entre clases

```
CLASS Numericos
  PUBLIC uno : INTEGER;
END_CLASS

CLASS AlfaNumericos(Numericos)
  //La clase AlfaNumericos hereda de la clase Numericos
  PUBLIC a : CHAR;
END_CLASS

FUNCTION_BLOCK Ejemplo
  VAR
    AlfaNumerico : AlfaNumericos;
    Numerico : Numericos;
  END_VAR
    AlfaNumerico.uno:=1;
    AlfaNumerico.a:='A';
    Numerico:=(Numericos) AlfaNumerico;
    //El objeto Numerico pierde el valor del atributo
    //Alfanumerico.a
END_FUNCTION_BLOCK
```

Al hacer el casting ascendente de la clase *"AlfaNumericos"* a la clase padre, se está informando al compilador que se desea hacer un *"CAST"* con pérdida de precisión, que en el caso de una clase significa la pérdida de atributos. Al existir una relación de herencia entre las dos clases, la clase hija posee todos los atributos de la clase padre, con lo que es posible en este ejemplo, asignar a la clase *"Numericos"* todos sus valores .

En el caso de hacer un *"casting"* descendente (sin pérdida de precisión) de una clase padre a una clase hija, sólo se podrán asignar aquellos atributos de la clase padre (que son los que como mínimo posee la clase hija), dejando el resto con su valor por defecto.

3.12.2. Traducción en IEC 61131

En tiempo de compilación, al detectar un *"casting"* entre dos clases, el compilador de MIOOP realiza dos tareas:

1. Añade los métodos de *"casting"* a las clases implicadas, como se puede ver en el algorimo 3.43 en el que el compilador añade en la clase *"Numéricos"* el método *"Casting_ Numericos_ AlfaNumericos"* para realizar el casting.
2. Cambia la instrucción de "casting" por la llamada al FB que implementa dicho *"casting"*.

Algoritmo 3.43 Método añadido por el compilador para realizar el casting de clases

```
CLASS Numericos
  PUBLIC uno : INTEGER;
  METHOD Casting_Numericos_AlfaNumeros (ObjetoDestino
         : Numericos, ObjetoOrigen : Alfanumericos);
  //Método añadido por el compilador para realizar
  //el casting
END_CLASS
```

La no introducción de los métodos de *"casting"* en todas las clases tiene su fundamento en el ahorro de líneas de código y en hacer el programa de control más ligero. Sólo aquellas clases involucradas en operaciones de *"casting"* poseerán este tipo de métodos.

La traducción del ejemplo del apartado anterior (ver algoritmo 3.42), genera el código en IEC 61131 para el FB *"ejemplo"* que se ilustra en el algoritmo 3.44. El FB *"Casting_ Numericos_ AlfaNumericos"* del algorimo 3.44 que realiza la conversión entre las dos clases tendría el código que se muestra en el algorítmo 3.45.

3.13. Uso de WARNINGS

MIOOP proporciona un chequeo de tipos y variables no utilizadas debido a que la memoria de los PLC suele ser limitada. Este tipo de chequeo representa una

Algoritmo 3.44 Traducción del casting entre clases a IEC 61131

```
FUNCTION_BLOCK Ejemplo
  VAR
    AlfaNumerico : AlfaNumericos;
    Numerico : Numericos;
    Casting_Numericos_AlfaNumericos
            : Casting_Numericos_AlfaNumericos;
    //FB que Realiza el casting entre los dos objetos
  END_VAR
    AlfaNumerico.uno:=1;
    AlfaNumerico.a:='A';
    Casting_Numericos_AlfaNumericos (Numerico
            , AlfaNumerico);
    //Numerico.uno contiene el valor 1 y se pierde el
    //valor de AlfaNumerico.a
END_FUNCTION_BLOCK
```

Algoritmo 3.45 Implementación del FB _Casting_Numericos_AlfaNumericos

```
FUNCTION_BLOCK Casting_Numericos_AlfaNumericos
  VAR_IN_OUT
    ObjetoDestino : Numericos;
    ObjetoOrigen : AlfaNumericos;
  END_VAR
    ObjetoDestino.uno:=ObjetoOrigen.uno;
END_FUNCTION_BLOCK
```

ayuda de cara a la depuración y a minimizar la memoria utilizada para albergar el código y variables del programa de control.

Los tipos de WARNING que lanzan son dos:

1. Cuando se hace una conversión con pérdida de precisión. En este caso el WARNING que muestra el compilador es por la posible pérdida de datos en la conversión como se muestra en el algorítmo 3.46.
2. Cuando se declara una variable de tipo simple, un objeto o un FB, y no se usa en el POU que lo instancia, como se puede ver en el algorítmo 3.47.

Algoritmo 3.46 WARNING por conversiones de tipos

```
FUNCTION_BLOCK EjemploWaring
  VAR
    Entero : INTEGER;
    Decimal : REAL;
  END_VAR
    ...
    Entero:=(INTEGER)Decimal;
    //WARNING. Conversión de REAL a INTEGER. Posible
    //pérdida de información
    ...
END_FUNCTION_BLOCK
```

Algoritmo 3.47 WARNING por no utilización de elementos declarados

```
FUNCTION_BLOCK EjemploWaring
  VAR
    Sensor : SensorNivel;
    Entero : INTEGER;
    Decimal : REAL;
    Instancia_FB_De_Ejemplo : FB_De_Ejemplo;
  END_VAR
    Decimal:=10.5;
    //WARNING. Objeto Sensor declarado no utilizado.
    //WARNING. Variable Entera declarada no utilizada.
    //WARNING. Instancia a FB_De_Ejemplo no utilizada.
END_FUNCTION_BLOCK
```

3.14. Herencia múltiple

La herencia múltiple es un mecanismo que permite que una clase herede atributos y métodos de más de una clase.

La razón fundamental para considerar la herencia múltiple es la de permitir que dos o más clases se combinen dentro de otra, dando como resultado una nueva clase con características de ambas clases base. Parafraseando a Stroupstrup [Str86]:

> Un ejemplo razonable del uso de herencia múltiple, sería el proporcionar dos librerías de clases *"Filtro"* y *"Regulador"* para representar objetos que controlen el paso de aire limpio y su presión a una bomba neumática respectivamente. Un programador podría crear las clases como el algorítmo 3.48.
>
> Usando la herencia simple, habría que aumentar el número de clases que heredan de *"Regulador"* y *"Filtro"* para proporcionar los mismos beneficios que una herencia múltiple".

Algoritmo 3.48 Ejemplo de definición de herencia múltiple

```
CLASS Mi_Filtro_Regulador (Regulador, Filtro)
   ...
END_CLASS

CLASS Mi_Filtro (Filtro) //No es un regulador
   ...
END_CLASS

CLASS Mi_ Regulador (Regulador) //No es un filtro
   ...
END_CLASS
```

La herencia múltiple es un poderoso mecanismo que permite ahorrar gran cantidad de código de una forma elegante, pero que por el contrario, presenta dos problemas:

1. La herencia repetida. Es el caso en que dos de las clases base tengan un ancestro común. En el ejemplo del algorítmo 3.48, supóngase que tanto la clase *"Filtro"* como la clase *"Regulador"* heredan de la clase *"Elemento"*.

Gráficamente la jerarquía puede verse como en la figura 3.13.
El problema surge cuando el compilador pretende saber si debe juntar los atributos y métodos iguales de las clases base en *"Filtro_ Regulador"* o si deben estar separadas. Es decir, si se deben concatenar sólo los elementos distintos de las clases base *"Elemento"* en *"Filtro_ Regulador"* o juntar todos los elementos de las clases bases dándoles nuevos nombres. Ambas estrategias son correctas y dependen del contexto.

2. Ambigüedad con respecto al nombre de atributos y métodos idénticos en las clases base. En el ejemplo de la figura 3.13, supóngase que la clase *"Elemento"* posee un único atributo *"Caudal"*. De esta forma, la clase *"Filtro_ Regulador"* hereda dicho atributo tanto de la clase *"Filtro"* como de la clase *"Regulador"*. Al instanciar un objeto de la clase *"Filtro_ Regulador"* y acceder al atributo *"Caudal"*, el compilador no sabría si debe usar el atributo heredado de la clase *"Filtro"* o de la clase *"Regulador"* (ver figura 3.14).

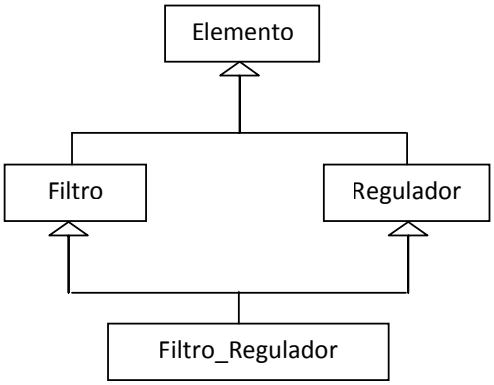

Figura 3.13: Ejemplo de herencia repetida

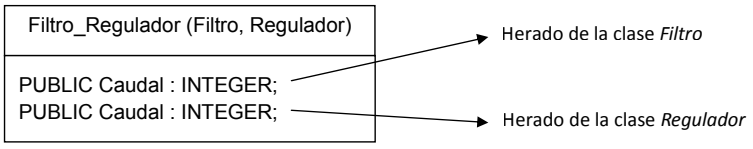

Figura 3.14: Ejemplo de ambigüedad en atributos y métodos heredados por várias vías

Varios lenguajes solucionan estos problemas de diversas maneras:

- C++ requiere que el programador resuelva la ambigüedad indicando de qué clase base vendrá la característica a usar. Para ello, el usuario debe emplear el operador *"::"* para indicar al compilador la procedencia del atributo o método al que se debe acceder.
- CLOS proporciona al programador control total del método de combinación, y si no es suficiente, el protocolo de *"MetaObjetos"* ofrece al programador formas de modificar la herencia: envío de métodos, instanciación de clases, y otros mecanismos internos sin afectar a la estabilidad del sistema.
- Eiffel permite al programador explicitar si junta o separa características que son heredadas de superclases. Eiffel juntará características automáticamente si tienen el mismo nombre e implementación. El programador tiene la opción de renombrar las características para separarlas.
- SmallTalk no soporta directamente multi-herencia pero posibilita declarar alias a métodos que permiten renombrar y acceder a métodos que quedarían ocultados por el mecanismo de resolución de conflictos convencional.
- Perl usa la lista de clases para heredar de una lista ordenada. El compilador usa el primer método que encuentra mediante búsqueda en profundidad por la lista de clases base.

3.14.1. Resolución de la ambigüedad de la herencia múltiple

MIOOP resuelve la ambigüedad de la herencia múltiple por medio del operador de ámbito *"::"* (ver apartado 3.6). Por ejemplo, dada la jerarquía de clases definidia en el algoritmo 3.48. Supongase que tanto la clase *"Regulador"* como la clase *"Filtro"* poseen ambas un método heredable llamado *"Cerrar"*. La llamada a uno de los métodos de mismo nombre (*"Cerrar"* en el ejemplo) desde la clase *"Filtro_ Regulador"* se resuelve como se puede observar en el algorítmo 3.49.

Algoritmo 3.49 Resolución de ambigüedad en la herencia múltiple

```
PUBLIC METHOD Filtro_Regulador::Medir()
  Regulador::Cerrar();
END_METHOD
```

3.14.2. Traducción a IEC 61131

Al igual que en la herencia simple, en la herencia múltiple cada una de las clases base se implementan en la estructura de la clase heredada como una referencia a dicha clase. De esta forma, la traducción a IEC 61131 de la clase *"Filtro_Regulador"* sería la que se muestra en el algorítmo 3.50.

Algoritmo 3.50 Traducción de una clase con herencia múltiple en IEC 61131

```
TYPE Filtro_Regulador:
  STRUCT
    _Filtro : Filtro;
    _Regulador : Regulador;
  END_STRUCT;
END_TYPE
```

Al compilar las clases definidas en el apartado anterior, el método *"Medir"* (ver algorítmo 3.49) se traduce a la norma IEC 61131 cómo el FB que se muestra en el algorítmo 3.51.

Algoritmo 3.51 Traducción de un método con herencia múltiple a IEC 61131

```
FUNCTION_BLOCK Filtro_Regulador_Medir
  VAR_IN_OUT
    THIS : Filtro_Regulador;
    Regulador_Cerrar : Regulador_Cerrar;
  END_VAR
    Regulador_Cerrar (THIS._Regulador);
END_FUNCTION_BLOCK
```

3.15. Inicialización de miembros heredados

Con la introducción de la herencia múltiple, la sintaxis de inicialización de clases bases y miembros debe ser también extendida. Por ejemplo, dada la definición de clases del algorítmo 3.52.

La sintaxis de la inicialización es idéntica a la sintaxis para la inicialización de objetos por medio de los constructores proporcionados por MIOOP, tal y como se

Algoritmo 3.52 Implementación de constructores en la herencia múltiple

```
CLASS A
  PUBLIC METHOD A (numero : INTEGER) : VOID;
END_CLASS

CLASS B
  PUBLIC METHOD B (numero : INTEGER) : VOID;
END_CLASS

CLASS X
  PUBLIC xx : INTEGER;
  PUBLIC METHOD X (numero : INTEGER) : VOID;
END_CLASS

//Implementación del constructor de la clase X
FUNCTION_BLOCK X_X
  VAR_IN_OUT
    THIS : X;
    NumeroA : INTEGER;
    NumeroB : INTEGER;
    NumeroX : INTEGER;
    A_A : A_A;
    B_B : B_B;
  END_VAR
    A_A (numeroA);  //Inicializa la base A
    B_B (numeroB);   //Inicializa la base B
    THIS.xx:=numeroX;  //Inicializa el miembro xx
END_FUNCTION_BLOCK
```

puede observar en el algorítmo 3.53 y su traducción a IEC 61131 en el algoritmo 3.54.

Algoritmo 3.53 Inicialización de objetos con herencia múltiple

```
FUNCTION_BLOCK EjemploInicializacion
  VAR
    A : A;
    B : B;
  END_VAR
    A.A(1);
    B.B(2);
END_FUNCTION_BLOCK
```

Algoritmo 3.54 Traducción de la inicialización de objetos con herencia múltiple en IEC 61131

```
FUNCTION_BLOCK EjemploInicializacion
  VAR
    A : A;
    B : B;
    A_A : A_A; //Constructor de la clase A
    B_B : B_B; //Constructor de la clase B
  END_VAR
    A_A(1);
    B_B(2);
END_FUNCTION_BLOCK
```

3.16. Controversia de la herencia múltiple

Existen muchas razones de controversia en la herencia múltiple [Car91a, Car91b, Wal91, Wal93, Sak92]. Los argumentos en contra se centraron en la complejidad real e imaginaria del concepto, su utilidad y el impacto de la herencia múltiple en otros rasgos de los lenguajes y en la construcción de herramientas:

1. Smalltalk no soporta la herencia múltiple y para muchas personas que consideran a Smaltalk como el paradigma OO a seguir, conjeturan que si Smaltalk

no lo tiene, la herencia múltiple debe ser mala o innecesaria, pero parafraseando a Stroupstrup [Str94], esta aseveración no se sostiene. Quizás Smaltalk podría beneficiarse de la herencia múltiple o quizás no, esa no es la cuestión. Sin embargo, está claro que algunas de las técnicas que los seguidores de Smaltalk recomendaban como alternativas a la herencia múltiple no se pueden aplicar a lenguajes tipificados con grandes restricciones de seguridad, tales como los de la norma IEC 61131. Por ejemplo, permitiendo renombrar métodos de otras clases que necesitan ser accedidos saltandose los mecanismos de privacidad de acceso.

2. Algunos expertos consideran que la herencia múltiple es una mala opción, fundamentalmente porque es demasiado difícil de usar y eso lleva a un diseño pobre y a un código lleno de *"bugs"*. Se puede abusar de la herencia múltiple, pero también de cada rasgo de un lenguaje de programación. Lo que parece más importante es que, en programas reales, el uso de la herencia múltiple ha llevado a una estructura que los programadores consideran superior a las alternativas de herencia simple, donde no existe otra alternativa obvia que simplifique la estructura del programa o su mantenimiento. Como asevera Stroupstrup [Str94], muchos de los se quejan de que la herencia múltiple conduce a error se basan exclusivamente en experiencias con lenguajes que no ofrecen un gran nivel de detección de errores en la compilación.

3. Otros expertos consideran la herencia múltiple como un mecanismo demasiado débil y a veces señalan la delegación como una alternativa. La delegación es un mecanismo para dirigir operaciones a otro objeto en tiempo de ejecución [Str87]. El problema es que la delegación complica el diseño y convierte los sistemas en menos eficientes, siendo una estrategia a seguir cuando no es posible implementar la herencia.

4. La herencia múltiple es un mecanismo aceptable que potencia las características de los lenguajes que lo implementan pero que a la vez, convierte en una tarea dura el diseño de herramientas que la utilicen. Sin embargo, este sobrecoste de los compiladores se recoge con beneficios por la mayor potencia que proporciona la herencia múltiple en el diseño e implementación de programas de control.

El defecto fundamental de estos argumentos es el de tomar la herencia múltiple con demasiada seriedad. La herencia múltiple no soluciona todos los problemas,

pero no tiene porqué hacerlo, ya que es un recurso que resulta ser bastante barato. A veces es muy conveniente tener herencia múltiple. Grady Booch [Boo91] expresa esta misma idea con un ejemplo:

> La herencia múltiple es como un paracaídas, no lo necesitas muy a menudo, pero cuando lo necesitas es esencial.

Su opinión está parcialmente basada en la experiencia adquirida de la reimplementación de los componentes de Booch de ADA en C++. Esta biblioteca de clases de contenedores y operaciones asociadas, implementadas por Grady Booch y Mike Vilot, es uno de los mejores ejemplos del uso de herencia múltiple [bV90, bV93].

3.17. Relaciones entre objetos

Durante la ejecución de un POU, los diversos objetos que lo componen han de interactuar entre sí para lograr una serie de objetivos comunes. Este tipo de interacción representa la comunicación entre diversos objetos mediante la mutua invocación de servicios.

MIOOP recoge 2 tipos de relaciones posibles entre objetos distintos:

1. Relaciones de asociación. Se dan cuando un objeto invoca los servicios (métodos) que ofrece otro objeto interactuando de esta forma con él (ver figura 3.15). Un ejemplo de asociación entre objetos se muestra en el algorítmo 3.55.
   Este tipo de relación representa la relación entre objetos con menos riqueza semántica.

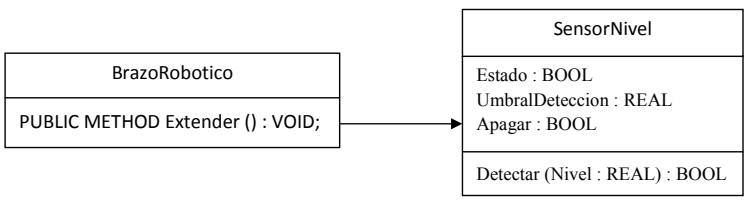

Figura 3.15: Relación de asociación entre objetos

Algoritmo 3.55 Implementación de la relación de asociación entre objetos

```
CLASS BrazoRobotico
  ...
  PUBLIC METHOD Extender (Sensor : SensorNivel) : VOID;
  ...
END_CLASS

PUBLIC METHOD BrazoRobotico::Extender (Sensor
        : SensorNivel ): VOID
  IF Sensor.Detectar(10) THEN
  //Llamada a un método de la clase SensorNivel
  ...
END_METHOD
```

2. Relación de agregación. Este tipo de relación se define cuando una clase es incluida como parte de otra clase, es decir, cuando una clase *"A"* incluye a otra clase *"B"* como un atributo más. A través de la agregación se definen objetos compuestos sumando las funcionalidades proporcionadas por los objetos agregados.

   A diferencia de la herencia, donde se añaden los atributos y métodos heredables de la clase base a la clase derivada, en la agregación el objeto no se encuentra físicamente dentro de la clase que lo contiene, sino que lo que se tiene es una referencia a dichas clases, con la ventaja de que puede estar agregado en más de una clase a la vez. Para ello, la clase *"A"* que agrega la clase *"B"* debe tener un atributo para almacenar la referencia al objeto de la clase agregada (en este caso, la clase *"B"*) y recibir una referencia de dicho objeto en algún momento, normalmente como un parámetro en el constructor de la clase *"A"*.

   En el algorítmo 3.56 se muestra un ejemplo de agregación y en el algorítmo 3.57 la traducción a la norma IEC 61131 tras la compilación.

   De este modo, los elementos de la clase agregada son accesibles desde la clase contenedora por medio del operador *"punto"*. La capa de protección de la clase agregada es manejada por MIOOP en tiempo de compilación, así como el acceso a los métodos de la clase contenedora. Ambos accesos se garantizan por el compilador por medio de los operadores de acceso de ámbito *"PUBLIC"*, *"PRIVATE"* y *"PROTECTED"*.

Algoritmo 3.56 Ejemplo de agregación de clases

```
CLASS Letras
  PUBLIC a : CHAR;
  PRIVATE z : CHAR;
END_CLASS

CLASS Numeros
  PUBLIC _1 : INTEGER;
  PRIVATE Letra : Letras; //Agregación de una clase
END_CLASS
```

Algoritmo 3.57 Traducción de agregación de clases en IEC 61131

```
TYPE : Letras
  STRUCT
    a : CHAR;
    z : CHAR;
  END_STRUCT;
END_TYPE

TYPE : Numeros
  STRUCT
    _1 : INTEGER;
    Letra : Letras;
  END_STRUCT;
END_TYPE
```

La diferencia entre asociación y agregación radica en que en la asociación sólo se tiene relación con la clase asociada a través del método que declara la asociación, y en la agregación todos los métodos de la clase tienen acceso a la clase agregada.

La necesidad de la existencia de la agregación y asociación entre objetos en MIOOP como otro medio de relación entre clases diferente a la herencia se justifica porque en la herencia, la clase heredada tiene total acceso a los elementos de la clase base salvo a los elementos privados, en cambio una clase agregada o asociada sólo se tiene acceso a los elementos públicos, no a los elementos protegidos.

3.18. Punteros

Un puntero es una variable que referencia a una posición de memoria donde se almacena otra variable. En otras palabras, un puntero es una variable cuyo valor es la dirección de memoria de otra variable. Si se tiene una variable *"p"* de tipo puntero que contiene una dirección de memoria en la que se encuentra almacenado un valor *"v"*, se dice que *"p"* apunta a *"v"* (ver figura 3.16).

Los punteros permiten de una forma rápida y eficiente el manejo de la memoria permitiendo trabajar con la dirección memoria de las variables y no con los datos que éstas contienen, lo que otorga a los lenguajes de programación que los soportan una gran potencia. Por otro lado, el uso de punteros es una fuente de errores si no se usan de forma adecuada y complica en cierta medida la lectura del código fuente.

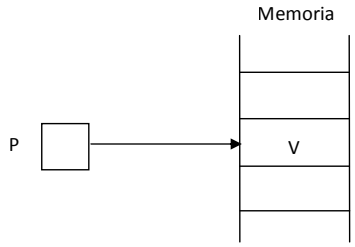

Figura 3.16: Segemento de memoria con punteros

Los punteros permiten implementar un grafo, donde los nodos de éste son los datos residentes en memoria y los arcos que unen los nodos son los propios punteros.

Dependiendo del lenguaje escogido, los punteros se implementan de diversas maneras, pero todas ellas se traducen de la misma forma a ensamblador. Cuando se declara un puntero, el compilador reserva memoria para éste, asignándole un nombre simbólico. Cuando se carga un valor en la variable del puntero (una dirección de memoria), por ejemplo, en el ensamblador de la máquina 80x86 lo interpreta haciendo una asignación directa de memoria a la variable por medio de un *"OFFSET"* a la posición de memoria. Al acceder al valor a que apunta el puntero, el ensamblador de la máquina 80x86 traduce dicho acceso por medio de una indirección al contenido de la variable por medio de la etiqueta [<variable>].

La mejor forma de ver el trato que da el ensamblador a los punteros, es por medio de un ejemplo en el lenguaje C, uno de los más versátiles en la utilización de punteros.

Primero, se declara el puntero. Ésta no es más que una variable de tipo puntero para almacenar la dirección de memoria de otra variable:

```
Unsigned char *p;
```

A continuación, se consigue la dirección de memoria de la variable a la que se quiere acceder:

```
p=&c;
```

Posteriormente, se puede asignar un dato indirectamente en la variable *"c"* a través del puntero *"p"*, como por ejemplo el valor 0xff:

```
*p=0xff;
```

Las instrucciones anteriores se traduciría a ensamblador de la máquina 80x86 como una asignación a la variable *"p"* de una dirección de memoria:

```
MOV P, OFFSET C
```

Y la asignación del valor 0xff a la variable *"c"* a través del puntero *"p"* quedaría traducida a ensamblador de la máquina 80x86 de la forma:

```
MOV [P], 0xff
```

3.18.1. Definición en MIOOP

Los punteros son una herramienta delicada ya que al estar trabajando con direcciones de memoria y no con variables, se vulnera todos los principios de encapsulación de datos y seguridad que proporciona la norma IEC 61131. Es decir, si por ejemplo en un FB se declara como parámetro de entrada un puntero y este se pasa en la parte *"VAR_IN"*, cualquier modificación que se haga sobre este puntero modificará el valor real al que está referenciando, hecho que va en contra de la naturaleza del tipo de parámetro *"VAR_IN"* (sólo entrada). Ésta es la razón por la que MIOOP no permite a los programadores declarar ni manejar punteros de forma directa. Su uso queda restringido como parte de la lógica interna de MIOOP para el manejo de ciertas características de la orientación a objetos que se verán más adelante.

Para dar soporte a los punteros es necesario ampliar la norma IEC 61131 añadiendo los operadores *"OFFSET"* y *"@"* como parte del lenguaje IL. El operador *"OFFSET"* permitiría asignar a una variable una dirección de memoria y el operador *"@"* permitiría acceder al valor contenido en la posición de memoria a la que apunta la variable. Estas dos nuevas palabras reservadas funcionan igual que sus equivalentes en ensamblador, utilizando la palabra reservada *"MOVE"* para asignar valores entre registros. Por tanto, el ejemplo del algorítmo 3.58 se traduciría a IEC 61131 como se muestra en el algorítmo 3.59.

Algoritmo 3.58 Punteros en lenguaje ensamblador de la máquina 80x86

```
MOVE C, 100
//Se carga en el en c el valor 100
MOVE P, OFFSET C
//Asignación a la variable p de la dirección de memoria
//de c
MOVE [P], 999
//Asignación al dato apuntado por p el valor 999
//Ahora c contiene el valor 999
```

Ya que el lenguaje ensamblador tiene semejanza con el lenguaje IL de la norma IEC 61131 y siendo los otros 4 lenguajes del estándar (LD, ST, FBD y SFC) de más alto nivel, parece razonable establecer una frontera en el manejo de punteros entre el lenguaje IL y el resto. De esta manera, MIOOP define las palabras reservadas * y & para los lenguajes ST, LD, FBD y SFC. La palabra reservada * se traduce a IL como un *"MOVE"* al valor del puntero por medio de la palabra reservada @. La

Algoritmo 3.59 Utilización de punteros en IEC 61131

```
MOVE C, 100
//Se carga en el en c el valor 100
MOVE P, OFFSET C
//Asignación a la variable p, la dirección de memoria
//de c
MOVE @P, 999
//Asignación al dato apuntado por p el valor 999
```

palabra reservada & se traduce a IL como otro *"MOVE"* a la dirección marcada por la variable a través de la palabra reservada *"OFFSET"*.

Para declarar un puntero en ST, LD, FBD o SFC, en la parte de declaración de variables de un POU, se precede el tipo de la variable de la palabra reservada *"*"*, lo que indica al compilador que esa variable es un puntero. Esta definición posee la estructura:

<NombreVariable>:*<TipoVariable>

Donde:

- <NombreVariable> identifica el nombre de la Variable.
- :* indica que la variable será de tipo puntero.
- <TipoVariable> identifica el tipo de variable al que apunta el puntero.

3.18.2. Traducción a IEC 61131

MIOOP, continuando con la filosofía estricta y fuertemente tipada de la norma IEC 61131 y para evitar errores de programación, no permite el manejo de punteros por parte del ingeniero programador por ser una potencial fuente de errores, al igual que ocurre con lenguajes como JAVA o los pertenecientes a .NET que gestionan internamente los objetos como punteros pero estos no pueden ser usados por los usuarios de forma directa.

Las traducciones que se hacen de los punteros a código IL por parte de un compilador no son necesarias para entender en el funcionamiento de MIOOP, sin embargo,

en este caso se muestran para que el lector pueda entender perfectamente como funcionan los punteros.

Ya que no tiene sentido mostrar un ejemplo sencillo de manejo de punteros, para mostrar el funcionamiento de éstos, se usa como ejempo un un ARRAY de punteros a objetos del que se hablará en la sección 3.20. Dada la función para el manejo de un sensor de nivel que se muestra en el algorítmo 3.60. Al compilarse la función anterior, se modifican los accesos al vector de objetos por accesos a punteros como se puede observar en el algorítmo 3.61. Este último algorítmo se traducía a lenguaje IL como se aprecia en el algorítmo 3.62.

Algoritmo 3.60 Ejemplo de punteros en MIOOP

```
FUNCTION_BLOCK UtilizacionPunteros
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF SensorNivel;
    //Array de objetos
  END_VAR
    //Inicialización de los objetos y resto del código
    //de la función
    Vector[1]:=Sensor;
    *Vector[1].UmbralDeteccion:=2;
END_FUNCTION_BLOCK
```

En el ejemplo del algorítmo 3.60, el ARRAY contiene la dirección de memoria del objeto por medio de un puntero. Cualquier acción sobre este puntero equivaldría a manejar directamente el objeto real (ver figura 3.17).

3.19. Punteros a funciones

En lenguajes como C o C++ es posible hacer la invocación de una función a partir su dirección de memoria contenida en un puntero. Esta dirección de memoria apunta al punto del segmento de código donde comienza el codigo de la función, con lo que el marcador del programa realiza un salto a dicha posición y comienza la ejecución de la función.

Algoritmo 3.61 Traducción de punteros a IEC 61131

```
FUNCTION_BLOCK UtilizacionPunteros
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF *SensorNivel;
    //Array de punteros a un objeto
  END_VAR
    //Inicialización de los objetos y resto del código
    //de la función
    Vector[1]:=&Sensor;
    //Se asigna la posición de memoria de la instancia
    //del objeto
    *Vector[1].UmbralDeteccion:=2;
    //Se accede al elemento real del objeto
END_FUNCTION_BLOCK
```

Algoritmo 3.62 Traducción a IL de punteros

```
FUNCTION_BLOCK UtilizacionPunteros
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF *SensorNivel;
    //Array de objetos
  END_VAR
    LD Sensor
    MOVE Vector[1], OFFSET Sensor
    //El vector[1] apunta al objeto Sensor
    MOVE @Vector[1],2
    //Asignación a la dirección que apunta vector[1]
    //un valor directo
END_FUNCTION_BLOCK
```

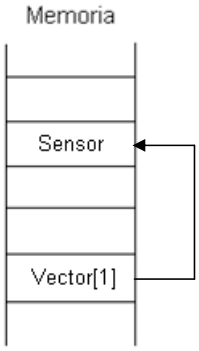

Figura 3.17: Sección de datos de un array de punteros

3.19.1. Definición en MIOOP

De igual forma que MIOOP maneja variables a través de punteros, es posible hacer llamadas a funciones, FBs o métodos de clases a través de punteros al segmento de código. Pero al igual que con los punteros a variables, su uso queda prohibido a los programadores y sólo el compilador realiza estas llamadas cuando precise, colocando directamente el código IL de la llamada a un POU a través de un puntero.

La llamada a una función, FB o método a través de punteros sigue la siguiente sintaxis:

<*NombrePOU> (<ListaParametros>)

Donde:

- (<*NombreVariable>) identifica el nombre de la función, FB o método de la clase que se desea llamar.
- (<ListaParametros>) indica la lista de parámetros de entrada de la función, FB o método separados por comas.

3.19.2. Traducción a IEC 61131

En el algorítmo 3.63 se observa la llamada a un método de un objeto a través de un ARRAY en MIOOP (ver sección 3.20). Al compilarse la función del algoritmo 3.63, se modifican las líneas en negrita por accesos a punteros como se muestra en el algorítmo 3.64. Este último algorítmo se traduciría a lenguaje IL como se aprecia en el algorítmo 3.65.

Algoritmo 3.63 Ejemplo de puntero a función en MIOOP

```
FUNCTION_BLOCK LlamadaConPunteros
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF SensorNivel;
    //Array de objetos
  END_VAR
    //Inicialización de los objetos y resto del código
    //de la función
    Vector[1]:=Sensor;
    //Se asigna la posición de memoria de la instancia
    //del objeto
    Vector[1].Detectar(2);
    //Se hace la llamada al método Detectar pasándole
    //como parámetro el valor 2
END_FUNCTION_BLOCK
```

De esta forma, se podrían implementar cualquier tipo de dato elemental, ARRAY u objeto, así como una llamada a una función o FB por medio de un puntero.

3.20. Punteros a objetos

Las clases pueden considerarse un tipo de dato más complejo que los datos de tipo simple como enteros o cadenas de caracteres, pero de igual forma que éstos, los objetos son susceptibles de ser accedidos a través de un puntero que apunta a la cabecera del primer elemento de la clase.

Por ejemplo, sea una clase *"A"* que almacena 3 enteros (ver figura 3.18). Dado un puntero *"p"* a un objeto de la clase *"A"*, este puntero estaría apuntando al primer

Algoritmo 3.64 Traducción de puntero a función en IEC 61131

```
FUNCTION_BLOCK LlamadaConPunteros
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF *SensorNivel;
    //Array de punteros a un objeto
  END_VAR
    //Inicialización de los objetos y resto del código
    //de la función
    Vector[1]:=&Sensor;
    //Se asigna la posición de memoria de la instancia
    //del objeto
    *Vector[1].Detectar(2);
    //Se hace la llamada al método Detectar pasándole
    //como parámetro el valor 2
END_FUNCTION_BLOCK
```

Algoritmo 3.65 Traducción a IL de puntero a función

```
FUNCTION_BLOCK LlamadaConPunteros
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF *SensorNivel;
    //Array de punteros a un objeto
  END_VAR
    //Inicialización de los objetos y resto del código
    //de la función
    LD Sensor
    MOVE Vector[1], OFFSET Sensor
    //El vector[1] apunta al objeto Sensor
    CAL @Vector[1] (2)
    //Llamada al segmento de código que almacena el FB
    //que implementa el método Detectar pasándole como
    //parámetro un número 2
END_FUNCTION_BLOCK
```

atributo del objeto (ver figura 3.19). El acceso a los elementos de la clase *"A"* sería el mismo desde el propio objeto de la clase *"A"* que desde el puntero *"p"*.

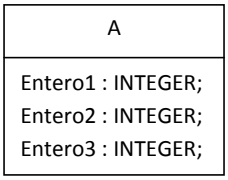

Figura 3.18: CRC de clase para su acceso a través de un puntero

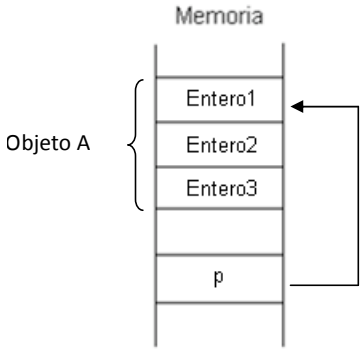

Figura 3.19: Segemento de datos de un puntero a un objeto

3.20.1. Definición en MIOOP

MIOOP no permite bajo ninguna circustancia el uso de punteros. Cuando el programador requiere manejar un ARRAY de objetos, el compilador puede optar por insertar los objetos en el ARRAY de dos formas diferentes:

1. Insertar en cada posición del ARRAY una copia de cada objeto.
2. Insertar en cada posición un puntero que apunta a la posición de memoria de cada objeto.

La primera solución tiene dos problemas. Por un lado, es una gran perdida de memoria y por otro lado, cualquier modificación que se haga en el objeto contenido en una posición de memoria no se vería reflejado en el objeto original por tratarse de una copia. Además, ante una asignación del tipo:

```
Objeto:=ARRAY[numero]
```

es necesario tener sobrecargado el operador *"="* para que funcione correctamente.

La segunda solución es óptima tanto en el tamaño de memoria consumida como en el hecho de que una modificción en el objeto al que apunta el puntero de cada posición del ARRAY queda reflejada en el objeto original.

MIOOP opta por la segunda posibilidad cuando el programador declara un ARRAY de objetos insertando de forma transparente el código necesario para manejar los punteros a los objetos de tal forma que de cara al usuario, éste trabaja directamente con los objetos del ARRAY. Cuando se instancia un ARRAY de objetos el compilador sustituye dicha declaración por un ARRAY de punteros.

3.20.2. Traducción a IEC 61131

Dada la clase *"SensorNivel"* definida en el apartado 3.2 y una función que declara un array a esta clase (ver algorítmo 3.66), el compilador traduciría automáticamente y de forma transparente al programador los accesos al array por el código contenido en el algoritmo 3.67.

Algoritmo 3.66 Función de ejemplo de un array de objetos

```
FUNCTION_BLOCK ArrayDeObjetos
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF SensorNivel;
    //Array de objetos
  END_VAR
    //Código de la función e inicialización del objeto
    //Sensor
    Vector[1]:=Sensor;
END_FUNCTION_BLOCK
```

3.21. Polimorfismo

La palabra *"polimorfismo"* proviene del griego (poli=varios y morfos=formas) y significa que posee varias formas diferentes. En POO, el polimorfismo se refiere

Algoritmo 3.67 Traducción de una función de ejemplo con un array de objetos a IEC 61131

```
FUNCTION_BLOCK ArrayDeObjetos
  VAR
    Sensor : SensorNivel;
    Vector : ARRAY [1..1] OF *SensorNivel;
    //ARRAY de punteros a SensorNivel
  END_VAR
    //Código de la función e inicialización del objeto
    //Sensor
    Vector[1]:=&Sensor;
    //Dirección de memoria del objeto Sensor
END_FUNCTION_BLOCK
```

a la posibilidad de definir clases diferentes que tienen métodos denominados de forma idéntica, pero que se comportan de manera distinta.

Una de las ventajas del polimorfismo es que se puede hacer una solicitud de una operación sin conocer el método que debe ser llamado. Estos detalles de ejecución quedan ocultos para el programador y la responsabilidad de decidir qué método debe ser invocado recae en el compilador.

Para entender mejor el concepto de polimorfismo es preferible ver un ejemplo.

Dada una jerarquía de clases de motores eléctricos como la mostrada en la figura 3.20.

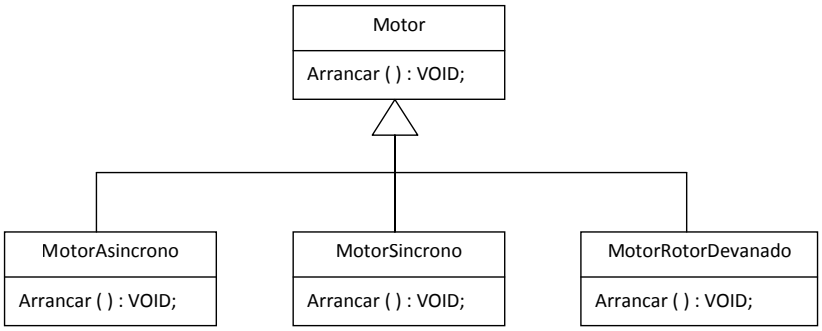

Figura 3.20: Diagrama de clases de motores eléctricos

Los motores asíncronos, síncronos y de rotor devanado heredan de una clase base común con la que comparten y redefinen el método *"Arrancar"*. Gracias al poli-

morfismo, sería posible poner en marcha los distintos tipos de motores a partir de invocaciones del método *"Arrancar"* de la clase base *"Motor"* y que sea el compilador quien decida cuál de los métodos de las clases derivadas se debe invocar.

Por ejemplo, dada la función del algorítmo 3.68 donde se define un ARRAY de *"Motores"*, es posible arrancar los motores derivados usando un bucle FOR. Sin el uso de polimorfismo, habría que realizar un arranque individual para cada uno de los distintos motores. En el ejemplo anterior, es el compilador quien decide qué tipo de motor debe arrancar.

Algoritmo 3.68 Ejemplo de arranque polimórfico de motores eléctricos

```
FUNCTION_BLOCK ArranqueMotores
  VAR_INPUT
    MAsincrono : MotorAsincrono;
    MSincrono : MotorSincrono;
    MRotorDevanado : MotorRotorDevanado;
  END_VAR
  VAR
    M : Motor;
    Vector : ARRAY[1..3] OF Motor;
    i : INTEGER;
  END_VAR
    Vector[1]:=MAsincrono;
    Vector[2]:=MSincrono;
    Vector[3]:=MRotorDevanado;
    FOR i:=1 TO 3 BY 1 DO
      M:=Vector[i];
      M.Arrancar(); //Llamada polimórfica
    END_FOR
END_FUNCTION_BLOCK
```

3.21.1. Definición en MIOOP

Un tipo abstracto de datos se define como una especie de caja negra. Una vez que ésta ha sido definida, realmente no interactúa con el resto del programa. No hay ninguna manera de adaptación si no es a través de la modificación de la definición. Ésto puede conllevar una gran inflexibilidad.

Supóngase definido una tipo *"Motor"* para el manejo de motores eléctricos. Se

asume por el momento que el sistema soporta motores asincronos, sincronos y de rotor devanado.

Se podría definir una clase *"Motor"* como el algorítmo 3.69.

Algoritmo 3.69 Primera aproximación al polimorfismo para la clase Motor

```
TYPE TipoMotor :
  (Sincrono, Asincrono, RotorDevanado)
  //Declaración de un tipo enumerado para ayudar a la
  //decisión del tipo de objeto que se recibe
END_TYPE

CLASS Motor
  PROTECTED SentidoGiro : BOOLEAN;
  PROTECTED Velocidad : INTEGER;
  PROTECTED T : TipoMotor;
  PUBLIC Motor () : VOID;
  PUBLIC METHOD RegularVelocidad (n : INTEGER) : VOID;
  PUBLIC METHOD Arrancar (T : TipoMotor) : VOID;
END_CLASS

METHOD Motor::Motor () : VOID;
  Vector[1]:=MAsincrono;
  Vector[2]:=MSincrono;
  Vector[3]:=MRotorDevanado;
END_METHOD

METHOD Motor::RegularVelociad (n : INTEGER) : VOID;
  Velocidad:=n;
END_METHOD
```

El tipo *"T"*, como propiedad de la clase *"Motor"*, es necesario para permitir operaciones como *"Arrancar"* y determinar el tipo de motor de que se trata. El método *"Arrancar"* deberíaía definirse como se ilustra en el algorítmo 3.70 para garantizar que el sistema funciona correctamente.

Este tipo de aproximación resulta ser muy confusa. Los métodos como *"Arrancar"* deberían conocer todos los tipos de motores que hay en el momento de escribir el código. Ésto es tremendamente inflexible ya que para poder arrancar nuevos tipos de motores futuros, sería necesario reprogramar el método *"Arrancar"* de la clase base *"Motor"* añadiéndole un nuevo caso para el nuevo tipo de motor.

Algoritmo 3.70 Implementación del méotdo "Arrancar"

```
METHOD Motor::Arrancar (T : TipoMotor) : VOID;
  SWITCH (T)
    CASE Sincrono
    //Se arranca un motor sincrono
    CASE Asincrono
    //Se arranca un motor asincrono
    CASE Asincrono
    //Se arranca un motor de rotor devanado
END_METHOD
```

Esta solución genera código fuertemente acoplado, ya que como se muestra en el ejemplo 3.71, es el método "Arrancar" de la clase *"Motor"* el que se encarga de invocar al método *"Arrancar"* apropiado de la clase derivada según el tipo del objeto. Este acoplamiento rompe con el concepto de encapsulación de la OO. Lo lógico es que cada objeto defina e implemente la forma en que se debe arrancar con independencia de cómo se haga el resto.

En el ejemplo 3.70, el código para cada método crece cada vez que se añade un nuevo tipo de motor al sistema. Si se define un nuevo tipo de motor, toda operación sobre la clase motor debe ser revisada (y posiblemente modificada). No se podría añadir un nuevo tipo de motor al sistema, a menos que se dispusiese del código fuente para todas las operaciones.

Los mecanismos adaptados de Simula y C++ en MIOOP proporcionan los instrumentos necesarios para solventar este problema mediante el operador *"VIRTUAL"*.

El concepto de métodos virtuales es una de las mayores aportaciones de la programación orientada a objetos. Fue introducido primeramente por Simula [DJ67] y posteriormente adaptado por Stroupstrup [Str86, Str88] para dar soporte en C++ al polimorfismo, modificando la idea de Simula de una manera que fuese fácilmente implementada.

Los métodos virtuales son aquellos métodos que se definen en una clase, pero que su implementación puede que se difiera a una clase heredada. *"VIRTUAL"* es el término usado tanto en Simula como en C++ mediante el cual el programador indica que un método puede ser definido más adelante, en una clase que deriva de ésta. Este concepto es muy importante para el desarrollo del polimorfismo y se

denomina ligadura dinámica, ya que no se conoce a qué método debe llamarse en tiempo de compilación, sino dinámicamente, en tiempo de ejecución.

El concepto de método *"VIRTUAL"* soluciona el problema que surge cuando una clase derivada hereda de una clase base. Un objeto de la clase derivada puede ser referido tanto como del tipo de la clase base como del tipo de la clase derivada. Si hay métodos de la clase base redefinidos por la derivada, aparece un problema cuando un objeto derivado ha sido promocionado como del tipo de la clase base. Cuando un objeto derivado es referido como del tipo de la base, el comportamiento de la llamada al método deseado es ambigüo. Distinguir entre un método *"VIRTUAL"* y otro no *"VIRTUAL"* sirve para resolver este problema. Si el método en cuestión es designado como *"VIRTUAL"*, se llamará al método de la clase derivada (si existe). Si no es *"VIRTUAL"*, se llamará al método de la clase base.

La definición de un método virtual tiene la sintaxis:

VIRTUAL METHOD <NombreMétodo> (<ListaParametros>) ;

Donde:

- VIRTUAL METHOD identifica que el método que se declara a continuación es virtual. Hay que destacar que los métodos virtuales son por definición públicos, ya que de nada serviría definir un método virtual privado al que ninguna otra clase pudiera acceder.
- <NombreMétodo> identifica el nombre del método de la clase.
- (<ListaParametros>) indica la lista de parámetros de entrada al método separados por comas.

Para poder utilizar el polimorfismo en el ejemplo anterior, primero se definen las propiedades generales de todos los motores y se etiquetan los métodos virtuales que las clases derivadas implementarán (ver algotírmo 3.71). Dada esta definición, se pueden escribir métodos generales para el manejo de motores particulares tal y como se muestra en el algorítmo 3.72. Para definir motores particulares se debe indicar que ese nuevo motor es heredado de la clase *"Motor"* y especificar en el las propiedades particulares de dicho motor, incluyendo los métodos virtuales que deben ser implementados/redefinidos (ver algorítmo 3.73).

Algoritmo 3.71 Clase "Motor" con métodos virtuales

```
CLASS Motor
  PROTECTED SentidoGiro : BOOLEAN;
  PROTECTED Velocidad : INTEGER;
  PUBLIC METHOD RegularVelocidad (n : INTEGER) : VOID;
  VIRTUAL METHOD Arrancar () : VOID;
END_CLASS
```

Algoritmo 3.72 Ejemplo de arranque de "Motores" polimórficos

```
FUNCTION_BLOCK ArrancarTodo
  VAR_INPUT
    Vector : ARRAY[1..2] OF Motor;
  END_VAR
  VAR
    i : INTEGER;
  END_VAR
    FOR i:=1 TO 2 BY 1 DO
      Vector[i].Arrancar();
    END_FOR
END_FUNCTION_BLOCK
```

Algoritmo 3.73 Definición de un "Motor" particular

```
CLASS MotorAsincrono (Motor)
  PUBLIC METHOD CambioSentido () : VOID;
  VIRTUAL METHOD Arrancar () : VOID;
END_CLASS
```

3.21.2. Traducción a IEC 61131

La clave para la implementación de los métodos virtuales es asociar a cada clase que contenga este tipo de métodos y sus clases derivadas, una única tabla global con las llamadas a los métodos correctos que implementan los métodos virtuales que se define en una jerarquía de clases superior. Esta tabla de métodos virtuales es creada en tiempo de compilación y está oculta al programador. A esta tabla se le da el nombre de *"VMT"* (tabla de métodos virtuales). Además de esta tabla, se necesita un marcador por cada clase que indique a qué método de la tabla *"VMT"* se debe llamar, denominado *"vPointer"* (puntero a métodos virtuales).

La tabla *"VMT"* es un vector de punteros (ver figura 3.21). Su tamaño se decide en tiempo de compilación dependiendo del número de métodos etiquetados como virtuales. Por cada clase que posea métodos virtuales se añade una entrada a la tabla *"VMT"*. Cada puntero de la tabla *"VMT"* contiene la dirección de memoria de cada uno de los métodos de la clase derivada que redefine los métodos virtuales de la clase base. En tiempo de compilación, el compilador al encontrarse con una clase que posee métodos virtuales realiza las siguientes acciones:

1. Inserta en la tabla *"VMT"* un puntero por cada uno de los métodos virtuales al segmento de código donde se encuentra la implementación de dichos métodos, es decir, inserta en cada posición del vector de la tabla *"VMT"*, la posición de memoria de cada uno de los FBs que implementan los métodos virtuales y siempre en el mismo orden. Este último aspecto es de vital importancia, ya que al tener la tabla *"VMT"* las posiciones de memoria del segmento de código de los métodos en el mismo orden, permite estandarizar las llamadas a cada uno de los métodos en todas las clases que implementan los métodos virtuales. Esta inicialización de la tabla "*VMT*" se sitúa al principio del código del programa principal de forma automática por parte del compilador y de forma transparente para el ingeniero programador.

2. Añade un puntero *"vPointer"* que apunta al primer elemento insertado en la tabla *"VMT"* en el punto anterior. *"vPointer"* será el primer atributo de la clase que define los métodos virtuales y de las que heredan de esta, es decir, será el primer campo de la estructura que implementa los atributos de la clase.

3. Añade de forma automática y transparente al ingeniero programador, en el

constructor de la clase, los mecanismos necesarios para inicializar el puntero *"vPointer"* de cada objeto al principio de la tabla *"VMT"* que apunta al primer método virtual del objeto.

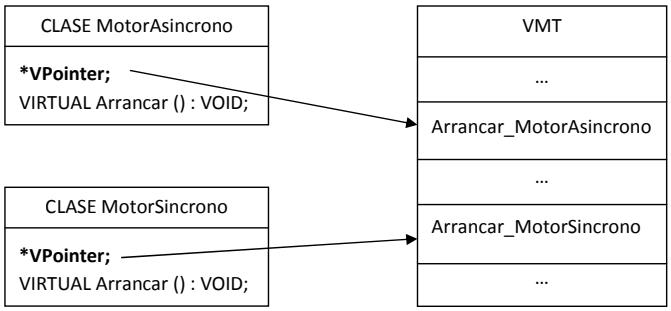

Figura 3.21: Esquema de memoria de la tabla *"VMT"*

El vector que implementa la tabla *"VMT"* tiene la sintaxis:

VMT : ARRAY[1..n] of *VOID;

Donde:

- VMT identifica que se trata de la tabla *"VMT"*.
- ARRAY[1..n] of *VOID indica al compilador que la tabla *"VMT"* es un ARRAY de punteros. El tamaño del array se decide en tiempo de compilación y será el del número total de métodos virtuales que se hayan definido en el sistema.

El puntero *"vPointer"* definido en cada clase que apunta a la entrada adecuada en la tabla *"VMT"* tiene la siguiente sintaxis:

vPointer : *VOID;

Donde:

- vPointer : *VOID identifica que se trata de un puntero a la tabla *"VMT"*.

A continuación, se muestra un ejemplo que ilustra estos conceptos:

Dada la jerarquía de clases que se muestra en la figura 3.22. Independientemente de la implementación de cada método virtual, a la clase *"A"*, *"B"* y *"C"* se les añade un puntero *"vPointer"* a la tabla *"VMT"* por poseer métodos virtuales. La traducción a IEC 61131 de estas tres clases sería la que se ilustra en el algorítmo 3.74.

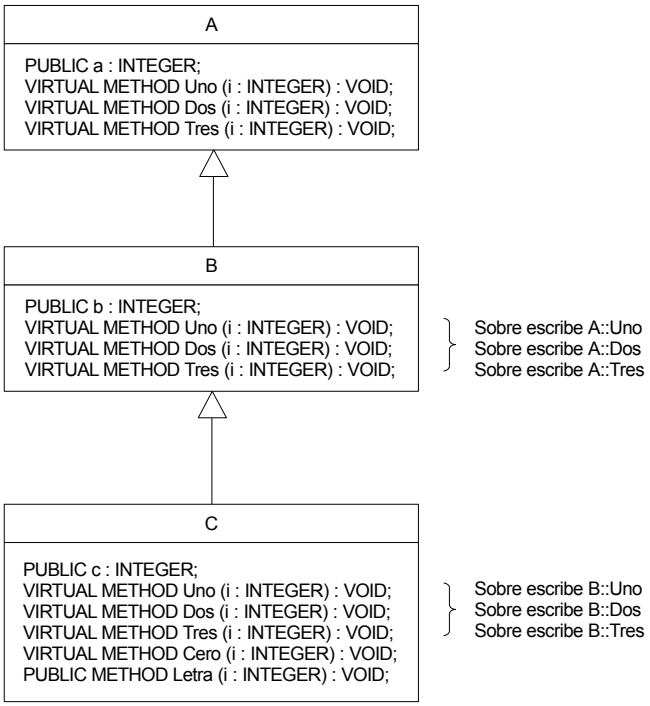

Figura 3.22: Diagrama de clases con métodos virtuales

En este ejemplo, la clase *"C"* no necesita definir sus métodos heredados como virtuales ya que no existe ninguna clase que herede de ella a la que pueda postergarse la implementación de sus métodos. En cuanto a la tabla *"VMT"*, el compilador le añade tres punteros a los métodos virtuales de la clase *"A"* y otros tres a la clase *"B"* ya que hereda de la clase *"A"*. Para la clase *"C"* se insertan cuatro elementos, tres por herencia de la clase *"B"* y otro más por el método virtual nuevo *"Cero"*. El método *"Letra"* queda sin insertar en la tabla *"VMT"* ya que ni es heredado ni es virtual. La duda surge cuando el compilador debe decidir el orden en el que

Algoritmo 3.74 Traducción de clases con métodos virtuales a IEC 61131

```
TYPE A :
  STRUCT
    vPointer : *VOID;
    //Puntero a la tabla VMT que contiene tres elementos
    //para esta clase por tener tres métodos virtuales
    a : INTEGER;
  END_STRUCT
END_TYPE

TYPE B :
  STRUCT
    vPointer : *VOID;
    //La tabla VMT heredada contiene los mismos elementos
    //que la de la clase base
    b : INTEGER;
    _A : A;
  END_STRUCT
END_TYPE

TYPE C :
  STRUCT
    vPointer : *VOID;
    //La tabla VMT heredada contiene un elemento adicional
    //para el método virtual N
    c : INTEGER;
    _B : B;
  END_STRUCT
END_TYPE
```

introducir las direcciones de memoria del segmento de código donde se encuentran los métodos virtuales en la tabla *"VMT"*. La solución es siempre introducirlas en el mismo orden en que se han declarado los métodos virtuales en la clase base, en este caso, la primera posición de *"VMT"* sería para el método *"Uno"* (declarado en la clase base) y la última para el método *"Cero"* (declarado en la clase *"C"*), tal y como puede apreciarse en la figura 3.23.

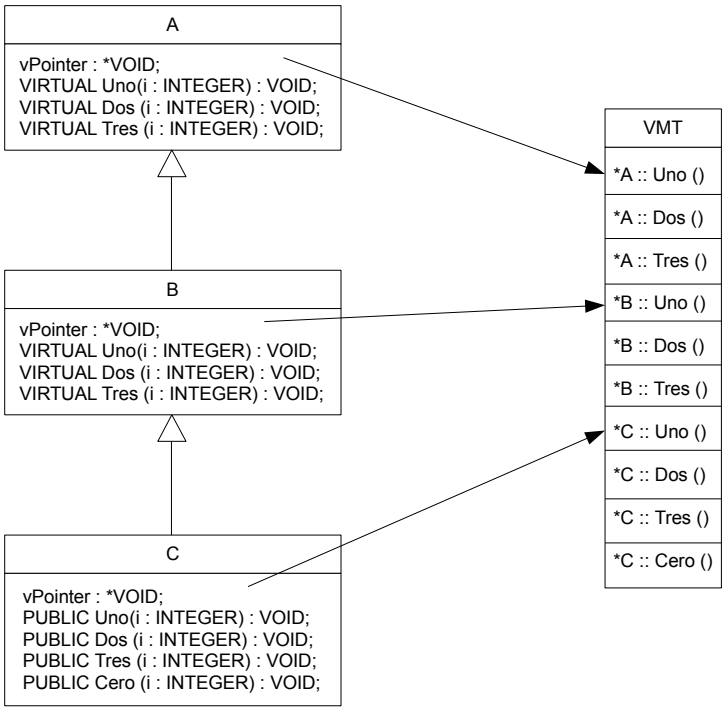

Figura 3.23: Esquema de memoria de la tabla *"VMT"* de la figura 3.22

A continuación se muestra un ejemplo en MIOOP de la potencia del polimorfismo a través de los métodos virtuales declarados en las clases *"A"*, *"B"* y *"C"* (ver algorítmo 3.75).

Al declararse un ARRAY de objetos, la variable *"Vector"* se traduce como un ARRAY de punteros a objetos de tipo *"A"*.

```
Vector : ARRAY [1..2] OF *A;
```

El ARRAY declarado sólo admite objetos de tipo *"A"* y sus derivados, por tanto,

Algoritmo 3.75 Ejemplo de utilización del polimorfismo

```
FUNCTION_BLOCK Ejemplo
  VAR
    ObjB : B;
    ObjC : C;
    Vector : ARRAY [1..2] OF A;
  END_VAR
    Vector[1]:= ObjB;
    Vector[2]:= ObjC;
    FOR i:=1 TO 2 BY 1 DO
      Vector[i].Tres (i);
    END_FOR
END_FUNCTION_BLOCK
```

el compilador traduce automáticamente la segunda y tercera sentencia por las direcciones de memoria de cada uno de los objetos:

```
Vector[1]:= &ObjB;
Vector[2]:= &ObjC;
```

Al traducirse a IEC 61131 la última línea contenida dentro del bucle FOR (Vector[i].Tres(i);), el compilador sustituye esa sentencia por una llamada a la dirección de memoria que contiene el puntero *"vPointer"* del objeto más un offset que introduce el compilador. Este offset viene marcado por el lugar que ocupa cada método virtual en la tabla *"VMT"* y es el mismo para toda la jerarquía de clases. En este ejemplo, el offset es 2 ya que se está llamando al tercer método virtual de la clase base (el elmento cero es el primer item marcado por *"vPointer"*), con lo que esa sentencia se traduciría como:

```
(*Vector[1].vPointer)+2 (i);
```

Esta línea de código traducida a IEC 61131 se compone de tres partes:

1. (*Vector[1].vPointer) ; Es una llamada al FB que contiene el puntero alojado en la posición 1 del ARRAY por medio de un acceso al puntero *"vPointer"* que apunta a la tabla *"VMT"*.

2. +2 ; Es el offset insertado en tiempo de compilación por el compilador. En toda la jerarquía de herencia de la clase *"A"*, cualquier invocación al método *"tres"* llevaría este offset.

3. (i) ; Son los parámetros del método *"tres"* que en este caso es un número entero.

Para que estas llamadas tengan efecto, el compilador añade en tiempo de compilación la inicialización del puntero *"vPointer"* en el constructor de cada objeto, como se ilustra en el algorítmo 3.76 para la clase *"A"*, en el algoritmo 3.77 para la clase *"B"* y en el algorítmo 3.78 para la clase *"C"*.

Algoritmo 3.76 Constructor de la clase *"A"*

```
FUNCTION_BLOCK A_A
  VAR_IN_OUT
    THIS : A;
  END_VAR
    THIS.vPointer:=&vPointer;
END_FUNCTION_BLOCK
```

Algoritmo 3.77 Constructor de la clase "B"

```
FUNCTION_BLOCK B_B
  VAR_IN_OUT
    THIS : B;
  END_VAR
    THIS.vPointer:=&vPointer+3;
END_FUNCTION_BLOCK
```

Algoritmo 3.78 Constructor de la clase "C"

```
FUNCTION_BLOCK C_C
  VAR_IN_OUT
    THIS : C;
  END_VAR
    THIS.vPointer:=&vPointer+6;
END_FUNCTION_BLOCK
```

La tabla *"VMT"* es una variable global a la que tienen acceso los dos constructores. La inicialización de los punteros *"vPointer"* de las dos clases se hace asignando la posición de memoria de la tabla más un offset que sitúa al puntero en la primera posición de la tabla que marca el primer método virtual de cada uno de las clases.

El comienzo de la función del ejemplo del algorítmo 3.75 se traduciría a la norma IEC 61131 como se muestra en el algorítmo 3.79.

Algoritmo 3.79

```
FUNCTION_BLOCK Ejemplo
  VAR
    ObjB : B;
    ObjC : C;
    *Vector[1..2] : A;
    B_B : B_B;
    //Constructor de la clase B
    C_C : C_C;
    //Constructor de la clase C
  END_VAR
    B_B (ObjB);
    //Inicialización del puntero vPointer del objeto ObjB
    C_C (ObjC);
    //Inicialización del puntero vPointer del objeto ObjC
    ...
    //Resto del código del FB
END_FUNCTION_BLOCK
```

Como en esta función de ejemplo del algoritmo 3.75 sólo se llama a un método virtual, el compilador no tiene la necesidad de inicializar las tablas *"VMT"* con el resto de valores que no va a utilizar con lo que hay un notable ahorro de código traducido.

3.22. Métodos virtuales puros

Un método virtual puro es un método virtual que sólo presenta la definición, es decir, no tiene implementaciñon y por tanto necesita ser implementado por una clase derivada.

3.22.1. Definición en MIOOP

Hay ocasiones, cuando se desarrolla una jerarquía de clases, que algún comportamiento está presente en todas ellas pero se materializa de forma distinta para cada una. Por ejemplo, olviendo al ejemplo del arranque de los motores de la jerarquía de clases descrita en el apartado 3.21. El método arrancar se lleva a cabo sobre toda la jerarquía de motores pero las operaciones concretas para llevarla a cabo dependen del tipo de motor en concreto (de su clase). Por otra parte, la acción arrancar un motor no tiene sentido para la clase genérica *"Motor"* porque esta clase representa una abstracción del conjunto de motores posibles.

Para resolver esta nueva característica deseable, MIOOP proporciona los métodos virtuales puros que representan la definición del método pero donde no está permitido su implementación, dejando ésta para una clase heredada.

Los métodos virtuales puros proporcionan al programador:

- Ayuda para encontrar errores que surgen en el diseño de los roles de una clase genérica y su rol representando objetos concretos.
- El soporte para un estilo de diseño de jerarquías basado en la separación de la especificación de clases genéricas y de sus implementaciones.

Volviendo al ejemplo del *"Motor"*. Se amplía la declaración de la clase *"Motor"* que representa el conjunto general de motores de alterna. Esta clase declara dos nuevos métodos virtuales, además del heredado *"Arrancar"*, que son los métodos *"Parar"* y *"CambiarSentidoGiro"*. Naturalmente, no tiene sentido que haya objetos de la clase *"Motor"*, sólo deberían existir objetos específicos derivados de ésta. Las reglas de MIOOP especifican que los métodos virtuales como *"Parar"* y *"CambiarSentidoGiro"* deben ser definidos en la clase en la que primero son declarados. La razón para este requerimiento es el de asegurar una perfecta unión entre una clase general y sus especializaciones, y evitar la posibilidad de llamar a métodos virtuales que no han sido implementados. Así, el programador podría definir una clase como la que se muestra en el algorítmo 3.80.

Este tipo de definición asegura que errores inocentes como olvidar implementar un método *"Parar"* por una clase derivada de *"Motor"* o errores tontos como crear una clase *"Motor"* sencilla e intentar usarla causen errores en tiempo de ejecución. Incluso cuando tales errores no se comenten, la memoria de un PLC puede fácilmente

Algoritmo 3.80 Definición de una clase con simulación de métodos virtuales puros

```
CLASS Motor
  PROTECTED SentidoGiro : BOOLEAN;
  PROTECTED Velocidad : INTEGER;
  PROTECTED Velocidad : INTEGER;
  PUBLIC METHOD RegularVelocidad (n : INTEGER) : VOID;
  VIRTUAL METHOD Arrancar ( ) : VOID;
  VIRTUAL METHOD Parar ( ) : VOID;
  VIRTUAL METHOD CambiarSentidoGiro ( ) : VOID;
END_CLASS

METHOD Motor::Parar () :VOID;
  //Mensaje de error si se intenta parar un motor base
END_METHOD

METHOD Motor::CambiarSentidoGiro () :VOID;
  //Mensaje de error si se intenta cambiar el sentido de
  //giro de un motor base
END_METHOD
```

colapsarse con innecesarias entradas en la tabla virtual (*"VMT"*) a métodos que nunca son llamadas de la clase *"Motor"* como *"Parar"* y *"CambiarSentidoGiro"*.

La solución para evitar estos errores es simplemente permitir que el ingeniero de control especifique cuándo desa que un método virtual no tenga implementación en la clase en que se define por primera vez, es decir, proveerlo de un instrumento que le permita declarar métodos virtuales puros. En MIOOP esto se puede hacer con un inicializador a "=0" en dicho método virtual puro según la siguiente síntaxis:

VIRTUAL METHOD <NombreMétodo> (<ListaParámetros>) = 0;

Donde:

- VIRTUAL METHOD indica que el método es virtual.
- <NombreMétodo> identifica a un método.
- (<ListaParametros>) recibe la lista de parámetros de entrada al método virtual.

- = 0 indica al compilador que el método virtual se declara como puro y que es obligatorio su implementación en una clase derivada. Además, indica que este método no puede ser invocado en una instancia de la clase a la que pertenece.

La definición de métodos virtuales puros por medio de la sintaxis *"=0"*, se escogió sobre otras palabras o conceptos más obvios por no introducir nuevas palabras reservadas que hicieran más confuso el concepto de MIOOP. En matemáticas, el número 0 significa *"ausencia de"* o *"carente de"*, por lo que al indicar que un método virtual es igual a 0, se puede entender fácilmente de forma intuitiva que dicho método carece de implmentación en esa clase y que por tanto, su implementación se retrasa a una clase derivada.

Con el nuevo concepto de métodos virtuales puros, la clase *"Motor"* implementada en el algorítmo 3.80 puede ser redefinida etiquetando los métodos virtuales que posee como puros, tal y como se muestra en el algorítmo 3.81.

Algoritmo 3.81 Definición de una clase con métodos virtuales puros

```
CLASS Motor
  PROTECTED SentidoGiro : BOOLEAN;
  PROTECTED Velocidad : INTEGER;
  PROTECTED Velocidad : INTEGER;
  PUBLIC METHOD RegularVelocidad (n : INTEGER) : VOID;
  VIRTUAL METHOD Arrancar ( ) : VOID;
  VIRTUAL METHOD Parar ( ) : VOID = 0;
  VIRTUAL METHOD CambiarSentidoGiro ( ) : VOID = 0;
END_CLASS
```

3.22.2. Traducción a IEC 61131

La traducción de un método virtual puro a la norma IEC 61131 es idéntica a la traducción de un método virtual normal (ver apartado 3.21, polimorfismo). La única diferencia radica en el control que ejerce sobre este tipo de métodos el compilador, no permitiendo accesos a estos métodos desde instancias de la clase que los define y obligando a su implementación en las clases que heredan.

3.23. Clases abstractas

Una clase en la que todos sus métodos se declaran como virtuales puros se denomina clase abstracta. Una clase abstracta sólo puede ser usada como base de otra clase. Ésto implica que no es posible instanciar objetos de ella. Una clase derivada de una abstracta es la encargada de implementar sus métodos virtuales puros. Se puede decir que las clases abstractas definen tipos (de datos) abstractos.

3.23.1. Definición en MIOOP

En MIOOP no existe técnicamente una diferencia entre una clase en la que todos sus métodos son virtuales y una clase abstracta. La diferencia es más sintáctica que semántica de tal forma que cuando un ingeniero de automatización se refiere a una clase abstracta entiende que ningúno de sus métodos esta implementado, sólo han sido definidos y que por tanto, su implementación debe desarrollarse en una clase hija (ver algorítmo 3.82).

Algoritmo 3.82 Definición de clase abstracta

```
CLASS Motor
  PROTECTED SentidoGiro : BOOLEAN;
  PROTECTED Velocidad : INTEGER;
  VIRTUAL METHOD RegularVelocidad (n : INTEGER) : VOID = 0;
  VIRTUAL METHOD Arrancar ( ) : VOID = 0;
  VIRTUAL METHOD Parar ( ) : VOID = 0;
  VIRTUAL METHOD CambiarSentidoGiro ( ) : VOID = 0;
END_CLASS
```

Es de destacar que todos los métodos de la clase abstracta *"Motor"* son virtuales puros, es decir, que no serán implementados por esta clase. Solo las clases derivadas de *"Motor"* están obligadas a implementar estos métodos. Por ejemplo, una clase *"MotorSincrono"* que herede de *"Motor"* podría definirse como se muestra en el algorítmo 3.83.

Bajo esta nueva definición, es posible utilizar una clase abstracta como base para el polimorfismo de las clases derivadas de la misma forma que se describió en el apartado 3.21, polimorfismo. Como aplicación práctica se muestra en el algoritmo 3.84 un arranque de un motor asíncrono que hereda de la clase abstracta *"Motor"*.

Algoritmo 3.83 Implementación de una clase derivada de otra abstracta

```
CLASS MotorSincrono (Motor)
  PUBLIC METHOD RegularVelocidad (n : INTEGER) : VOID;
  PUBLIC METHOD Arrancar ( ) : VOID;
  PUBLIC METHOD Parar ( ) : VOID;
  PUBLIC METHOD CambiarSentidoGiro ( ) : VOID;
END_CLASS
```

Algoritmo 3.84 Ejemplo de polimorfismo con clases abstractas

```
FUNCTION_BLOCK ArrancarMotorSincrono
  VAR_IN_OUT
    ObM : Motor;
  END_VAR
    ObM.Arrancar();
    //Llamada al método arrancar a través de polimorfismo
END_FUNCTION_BLOCK

FUNCTION_BLOCK Ejemplo
  VAR
    ObMotorSincrono : MotorSincrono;
  END_VAR
    ArrancarMotorSincrono (ObMotorSincrono);
END_FUNCTION_BLOCK
```

Como se indicó previamente, no es posible crear instancias de objetos de una clase abstracta. Este error se captura en tiempo de compilación (ver algorítmo 3.85).

Algoritmo 3.85 Error al instanciar una clase abstracta

```
FUNCTION_BLOCK Ejemplo
  VAR
    ObM : Motor;
    ObM2 : Motor;
    //Error. No se pueden instanciar una clase abstracta
  END_VAR
    ObM.Parar();   //Ok
END_FUNCTION_BLOCK
```

La importancia del concepto de las clases abstractas radica en que permiten definir una clara separación entre el esqueleto de una clase base y las clases derivadas encargadas de implementar dicho esqueleto. Además, Stroupstrup demostró que la utilización de clases abstractas en sistemas de tamaño medio-grande reduce el tiempo de diseño y compilación por un factor de 10 [Str89].

3.23.2. Traducción en IEC 61131

Al igual que ocurre con los métodos virtuales puros, las clases abstractas no poseen una traducción a parte de la descrita en el apartado 3.21, polimorfismo. Es el compilador el encargado de asegurarse que los métodos de la clase abstracta han sido implementados en las clases derivadas y que no se han hecho instancias de dicha clase.

3.24. Interfaz

El concepto de clases abstractas (ver sección 3.23) puede llevarse un poco más lejos, definiendo un tipo de clase abstracta en la cual todos sus métodos están sin implementación y no posee ni atributos, constructores, ni destructores. Estas clases no aportan ni estado ni comportamiento, simplemente proporcionan declaraciones de métodos que están obligados a implementar cualquier clase que herede de el.

3.24.1. Definición en MIOOP

La definición de clases abstractas es suficiente para declarar clases que sirven como esqueleto para sus clases derivadas, pero para su definición, es obligatorio que el programador declare todos sus métodos como virtuales puros. Este trabajo puede llevar a errores ya que no existe diferencia sintáctica entre una clase normal, otra que implemente algún método virtual puro y una clase abstracta. Para resolver este problema y no romper con el fuerte sentido tipificado de la norma IEC 61131, se define en MIOOP el tipo de dato *"INTERFAZ"* mediante el cual, un usuario puede distinguir de una forma clara una clase normal de una clase abstracta.

La declaración de un *"INTERFAZ"* sigue la sintaxis:

INTERFAZ <NombreClase>

Donde:

- INTERFAZ indica que la clase declarada actúa como una clase de interfaz.
- <NombreClase> identifica a una clase que actúa como un interfaz.

Los interfaces no pueden poseer ningún tipo de variable declarada en su parte de definición de atributos. Los métodos que se declaran dentro de la definición de un *"INTERFAZ"* son todos métodos virtuales puros, aunque no es necesario etiquetarlos con *"VIRTUAL"* ni tampoco terminar la definición del método con el sufijo *"=0"*. Este tipo de errores son resueltos en tiempo de compilación (ver algorítmo 3.86).

Algoritmo 3.86 Ejemplo de definición de interfaz

```
INTERFAZ A
  METHOD g ( ) : VOID;
  METHOD f ( ) : VOID;
  METHOD h ( ) : VOID;
END_INTERFAZ
```

3.24.2. Traducción en IEC 61131

Podría pensarse que un *"INTERFAZ"*, al representar nada más que un contrato por el cual toda clase que derive de él está obligada a implementar todos sus

métodos, carecer de atributos y no poseer implementación alguna de los métodos que define, no debería tener ningún tipo de traducción a la norma IEC 61131 o como mucho, traducirse simplemente a una estructura (traducción de una clase. Ver apartado 3.2, Clases y objetos) sin atributos. Nada más lejos de la realidad. Los interfaces son traducidos en tiempo de compilación como una estructura con un único atributo que es el puntero *"vPointer"* que contiene la dirección de memoria de la tabla *"VMT"* (ver apartado 3.21, polimorfismo). Este único atributo es necesario para garantizar la ligadura dinámica en tiempo de ejecución cuando se realiza una llamada a un método definido en el interfaz desde una clase derivada de éste. Para entender esta definición, se empleará un contra ejemplo en el que un *"INTERFAZ"* no se traduce como una estructura a IEC 61131.

Dado el interfaz para definir motores de alterna que se muestra en el algorítmo 3.87 y dada una clase que derivada de dicho interfaz (ver algorítmo 3.88). Si los *"INTERFAZ"* no se tradujesen como una estructura con el puntero *"vPointer"* apuntando al primer elemento de la tabla *"VMT"*, un intento de llamada a un método del *"INTERFAZ"* como el del algorítmo 3.89 fallaría. Para comprender el error que ocurre en el algorítmo 3.89 hay que observar el intento de llamada que se produce en la traducción de la función *"Arrancar_Motor_ASincrono"* a IEC 61131 (ver algorítmo 3.90).

Algoritmo 3.87 Definición del interfaz *"Motor"*

```
INTERFAZ Motor
  METHOD RegularVelocidad (n : INTEGER) : VOID;
  METHOD Arrancar ( ) : VOID;
  METHOD Parar ( ) : VOID;
  METHOD CambiarSentidoGiro ( ) : VOID;
END_INTERFAZ
```

Como puede observarse en el algorítmo 3.90, el compilador inserta de forma automática y transparente al programador, una llamada a un FB a través de la dirección de memoria alojada en el puntero *"vPointer"* que sustituye a la llamada al método del *"INTERFAZ"* del algorítmo 3.89 (como se ha explicado en el capítulo 3.21, polimorfismo). Al no existir el campo *"vPointer"* en la estructura que implementa el *"INTERFAZ"* es imposible realizar la ligadura dinámica con la llamada al método correcto.

Por tanto, es necesario que el *"INTERFAZ"* definido para el motor génerico sea

Algoritmo 3.88 Clase "MotorAsincrono" que implementa el interfaz "Motor"

```
CLASS MotorAsincrono (Motor)
  PRIVATE SentidoGiro : BOOLEAN;
  PRIVATE Velocidad : INTEGER;
  PRIVATE Estado : STRING ;
  PUBLIC METHOD RegularVelocidad (n : INTEGER) : VOID;
  PUBLIC METHOD Arrancar ( ) : VOID;
  PUBLIC METHOD Parar ( ) : VOID;
  PUBLIC METHOD CambiarSentidoGiro ( ) : VOID;
END_CLASS
```

Algoritmo 3.89 Intento fallido de acceso a un método de un interfaz

```
FUNCTION_BLOCK ArrancarMotorASincrono
  VAR_INPUT
    ObM : Motor;
  END_VAR
    ObM.Arrancar();
    //Error. No se conoce el atributo de llamada vPointer
END_FUNCTION_BLOCK

FUNCTION_BLOCK Ejemplo
  VAR
    ObMotorASincrono : MotorASincrono;
  END_VAR
    ArrancarMotorASincrono(ObMotorASincrono);
END_FUNCTION_BLOCK
```

Algoritmo 3.90 Traducción del intento fallido de acceso al método de un interfaz a IEC 61131

```
FUNCTION_BLOCK ArrancarMotorASincrono
  VAR_INPUT
    ObM : Motor;
    //OK. Estructura que implementa el interfaz
  END_VAR
    ObM.vPointer)( );
    //Error. vPointer es un campo desconocido
END_FUNCTION_BLOCK
```

traducido a una estructura con el puntero *"vPointer"* como único campo de dicha estructura (ver algorítmo 3.91).

Algoritmo 3.91 Estructura que implementa la definición de un interfaz

```
TYPE Motor:
  STRUCT
    vPointer : *VOID; //Puntero a la tabla VMT
  END_STRUCT
END_TYPE
```

3.25. Sobrecarga de métodos

Cuando una variable puede ser de diferentes tipos, el programador debe decidir si permite un modo mixto de aritmética o requiere una conversión explícita de los operandos a un tipo común. Por ejemplo, supóngase la formula $F = M * A$. Si *"M"* es de tipo entero y *"A"* es de tipo doble, se puede aceptar la multiplicación y deducir que *"M"* debe ser promocionada a tipo doble antes de la multiplicación o que se requiera que el programador escriba algo como $(DOUBLE)M * A$. Sin embargo, si *"A"* fuese de tipo char, el compilador produciría un error ya que se trataría de un tipo de operación no permitida. Para evitar este problema, el compilador requeriría que el programador definiese algún tipo de rutina que permitiese multiplicar un char y un entero.

Esta problemática puede extenderse al uso de objetos, donde los usuarios quieren herramientas de programación que no enturbien excesivamente la comprensión del código, pero que por otro lado, no oculten errores de programación.

La sobrecarga en la orientación a objetos se entiende como la posibilidad de que un objeto proporcione una serie de servicios con el mismo nombre pero distintas reglas de negocio. La única diferencia de estos servicios radicaría en el tipo y numero de los parámetros recibidos por los métodos que implementan dichos servicios.

3.25.1. Definición en MIOOP

La sobrecarga de métodos se refiere a la posibilidad de tener dos o más métodos con el mismo nombre pero funcionalidad diferente dentro de una misma clase (ver

algorítmo 3.92). Es decir, dos o más métodos con el mismo nombre que realizan acciones diferentes. El compilador, ante una llamada a un método sobrecargado, direcciona dicha llamada a uno u otro método dependiendo de los parámetros de entrada usados. Para un compilador estos métodos no tienen nada en común a excepción del identificador, por lo que se trata en realidad de un recurso semántico del lenguaje. El ámbito del nombre sobrecargado es el de la clase.

Algoritmo 3.92 Ejemplo de sobrecarga de métodos

```
CLASS A
  PUBLIC METHOD AA (numero : INTEGER);
  PUBLIC METHOD AA (numero1 : INTEGER, numero2 : INTEGER);
END_CLASS
```

La norma IEC 61131 se puede decir que en cierta forma también soporta la sobrecarga en funciones y FBs, pero sólo de dos maneras:

1. En la llamada a una función o FB variando el orden de los parámetros de entrada, siempre que se indique el tipo de variable que se pasa como parámetro (haciendo una llamada explícita). Si se cambiase el tipo de éstos, el compilador lanzaría un mensaje de error (ver algorítmo 3.93).

Algoritmo 3.93 Funciones sobrecargadas por los parámetros de entrada en IEC 61131

```
FUNCTION FuncionSobrecargada
  VAR_INPUT
    numero : INTEGER;
    ch : CHAR;
  END_VAR
    //Código de la function
END_FUNCTION

FUNCTION Llamada
  FuncionSobrecargada (numero := 10, ch :="a" ); //OK
  FuncionSobrecargada (ch := "a", numero := 10); //OK
  FuncionSobrecargada (numero :="a", ch :="a");
  //Error. Se esperaba un entero
END_FUNCTION
```

2. Añadiendo a los tipos de datos de los parámetro el prefijo *"ANY"*, lo que permite que sea el sistema el que resuelva en tiempo de ejecución el tipo de dato que se recibe. Este tipo de sobrecarga sólo sirve para las funciones estándar de la norma IEC 61131, como adición (ADD), susbstracción (SUB), etc.
   Existen 4 tipos de datos *"ANY"* (ver tabla 3.1) en la norma IEC 61131 y para que funcione este tipo de sobrecarga, no se pueden mezclar entre sí, en cuyo caso, el compilador mostrará un error en tiempo de compilación.

ANY_BIT	**ANY_INT**	**ANY_REAL**	**ANY_DATE**
BOOL	INT | UINT	REAL	DATE
BYTE	SINT | USINT	LREAL	TIME_OF_DAY
WORD	DINT | UDINT		DATE_AND_TIME
DWORD	LINT | ULINT		
LWORD			

Cuadro 3.1: Tabla de tipos de funciones sobre cargadas con *"ANY"*

Este tipo de sobrecarga definida por IEC 61131 es limitada, ya que no permite sobrecargar ningún tipo de función definida por el usuario.

Para proporcionar a los ingenieros de automatización una mayor flexibilidad que la que aporta la norma IEC 61131, MIOOP amplia el concepto de sobrecarga de la norma y lo extiende a las clases.

La sobrecarga de métodos fue introducida por primera vez por C++ en 1986 [Str86] y revisada en 1989 por Stroupstrup [Str89]. Esta versión permitió la resolución del problema que se produce con tipos de datos o clases que son demasiado similares y aumentó la independencia del orden de la declaración.

Cuando se realiza la invocación de un método sobrecargado, es decir, que existen otros con idéntico nombre en el mismo ámbito, el compilador decide cuál de ellos se utilizará mediante un proceso denominado *"resolución de sobrecarga"* aplicando ciertas reglas para verificar cuál de las declaraciones se ajusta mejor al número y tipo de los argumentos utilizados. Es decir, donde existe máxima concordancia entre los argumentos actuales y formales. El proceso sigue unas reglas denominadas de *"congruencia estándar de argumentos"*. Son las siguientes (en el orden de precedencia señalado):

1. Concordancia exacta en el número y el tipo de los parámetros de entrada.

2. Concordancia una vez realizadas las promociones de los tipos asimilables a enteros, por ejemplo: SHORT; BOOL; enumerados a INTEGER. También de tipos REAL a DOUBLE.

3. Concordancia una vez realizadas las conversiones estándar. Por ejemplo: INTEGER a DOUBLE; WORD a DOUBLE; clase-derivada a superclase.

4. Concordancia una vez realizadas las conversiones explicitadas por el usuario.

Si el análisis conduce a que dos métodos distintos concuerdan al mismo nivel, entonces se produce un error y la sentencia es rechazada por el compilador, declarando que existe una ambigüedad.

Cuando los métodos sobrecargados tienen dos o más argumentos, se procede a obtener la mejor concordancia para cada uno, utilizando las reglas anteriores. Si un método tiene mejor concordancia que los demás para un argumento y mejor o igual concordancia para los restantes, entonces éste se invoca. En caso contrario, la llamada se rehúsa declarándola ambigüa. El orden en que hayan sido declarados los métodos sobrecargados no influye en la precedencia anterior.

Si tras la aplicación de las reglas anteriores ocurre una situación de ambigüedad se pueden desarrollar varias estrategias. Un posible mecanismo de resolución de ambigüedades dependería del orden de la declaración de cada método. En tiempo de compilación, la declaración correcta que primero tenga una coincidencia es la que *"gana"*. Para hacer esto tolerable, sólo las conversiones no limitadas son aceptadas en una coincidencia. Por ejemplo, dada una clase cualquiera de ejemplo y dados dos métodos sobrecargados cuya única diferencia es el tipo de parámetro recibido (ver algorítmo 3.94), ante una serie de llamadas al método *"Accion"* (ver algoritmo 3.95), la primera llamada provoca un error en tiempo de compilación, ya que el primer método *"Accion"* espera como parámetro de entrada un entero y recibe un doble. Para solventar este problema, el compilador puede añadir de forma transparente al programador, una línea de conversión de tipos, bien por medio de los casting proporcionados por MIOOP o mediante las funciones de conversión de la norma IEC 61131, con la correspondiente perdida de precisión al transformar una variable doble a entero. En el siguiente ejemplo se muestran las dos posibles alternativas de traducción de la línea *"ObjetoEjemplo.Accion (2.0);*" :

1. Conversión por medio de funciones proporcionadas por IEC 61131.

```
ObjetoEjemplo.Accion (DOUBLE_TO_INT (2.0));
```

2. Conversión por medio del casting definido por MIOOP.

```
ObjetoEjemplo.Accion ((DOUBLE) 2.0);
```

Algoritmo 3.94 Ejemplo de dos métodos sobrecargados en MIOOP

```
METHOD ClaseEjemplo::Accion  //Método Accion(entero)
  VAR_INPUT
    numero : INTEGER;
  END_VAR
    //Código del método
END_METHOD

METHOD ClaseEjemplo::Accion  //Método Accion(doble)
  VAR_INPUT
    numero : DOUBLE;
  END_VAR
    //Código del método
END_METHOD
```

Algoritmo 3.95 Llamada a un método sobrecargado en MIOOP

```
FUNCTION_BLOCK Llamada
  VAR
    ObjetoEjemplo : ClaseEjemplo;
  END_VAR
    ObjetoEjemplo.Accion(2.0);
    //Llamada a ObjetoEjemplo.Accion(entero).Error en el
    //tipo de los parámetros
    ObjetoEjemplo.Accion (2);
    //Llamada a ObjetoEjemplo.Accion(entero).OK
END_FUNCTION_BLOCK
```

Esta regla para solventar la ambigüedad es fácil de expresar, simple de entender para los programadores, eficiente en tiempo de compilación, trivial para una implementación correcta y sin embargo, es una constante fuente de errores y confusión.

Revisando la declaración del algorítmo 3.94, el orden podría cambiar completamente el significado del código observando el algorítmo 3.96 y las llamadas del algorítmo 3.97 respectivamente.

Algoritmo 3.96 Modificación de los métodos *"Accion"* en MIOOP

```
METHOD ClaseEjemplo::Accion  //Método Accion(doble)
  VAR_INPUT
    numero : DOUBLE;
  END_VAR
    //Código del método
END_METHOD

METHOD ClaseEjemplo::Accion  //Método Accion(entero)
  VAR_INPUT
    numero : INTEGER;
  END_VAR
    //Código del método
END_METHOD
```

Algoritmo 3.97 Modificación de la llamada a los métodos *"Accion"* en MIOOP

```
FUNCTION_BLOCK Llamada
  VAR
    ObjetoEjemplo : ClaseEjemplo;
  END_VAR
    ObjetoEjemplo.Accion(2.0);
    //Llamada a ObjetoEjemplo.Accion(doble). OK
    ObjetoEjemplo.Accion (2);
    //Llamada a ObjetoEjemplo.Accion(doble). Necesario
    //hacer una conversión de tipos de entero a doble
END_FUNCTION_BLOCK
```

En el ejemplo de los algoritmos 3.96 y 3.97, el orden cambia sustancialmente el resultado, ya que al convertir una variable de tipo entero a doble, no hay pérdida de precisión. Básicamente, el orden de precedencia es propenso a errores. La solución es ir hacia una vista de programación en la que el orden de aparición/definición de los métodos no influya en las reglas de llamadas al método correcto sobrecargado.

La dificultad de encontrar un nuevo mecanismo de resolución de ambigüedades es la de dotar al compilador de una independencia del orden de definición de los

métodos y que se mantenga una absoluta compatibilidad con la norma IEC 61131. En particular, la de la regla que dice:

> Si una expresión tiene dos posibles representaciones legales, ésta es ambigua y por tanto ilegal.

Por ejemplo, según esa regla, la anterior llamada al método *"Accion"* sería ambigüa e ilegal. Por tanto, se necesita una notable mejora en las reglas de coincidencia en las que se permita una declaración del tipo de conversión a aplicar (en el ejemplo anterior, la conversión de tipos de doble a entero) que pierda menos precisión en el cambio.

Este tipo de reglas debe minimizar los errores a los programadores de autómatas en las conversiones implícitas de tipos. El problema fundamental radica en que el grafo del algoritmo de compilación de construcción de las conversiones implícitas contiene ciclos. Por ejemplo, no sólo existen conversiones de tipo short a entero, sino que también de entero a short. Ésto potencia la aparición de sutiles errores en los programas de control y por tanto, es el programador el encargado de desarrollar esquemas de conversión que prevengan este tipo de errores. En lugar de eso, el compilador debe diseñar un sistema de coincidencia entre los tipos de parámetros de entrada encontrados en la declaración de los métodos y los tipos de los métodos coincidentes. De esta forma, las coincidencias implican conversiones menos propensas a errores y con menor número de errores de cara al programador, es decir, el objetivo final es evitar cualquier posible fuente de errores de programación de tal manera que si existe ambigüedad, se debe producir un error que debe ser corregido por el programador (el ingeniero de automatización debe ser siempre consciente de lo que hace). Por tanto, implementar un compilador que cumpla con estos objetivos implica primero definir las siguientes cuatro reglas de coincidencias:

1. Coincidencias sin conversión o con conversiones inevitables (por ejemplo variable a constante).
2. Coincidencias usando promociones integrales (como define el estándar ANSI C, de carácter a entero, short a entero; y sus correspondientes versiones sin signo) y de real a doble.
3. Coincidencias usando conversiones estándar (por ejemplo, de entero a doble, de tipos de datos sin signo a sus respectivas versiones con signo, como por ejemplo de entero sin signo a entero).

4. Coincidencias para conversiones de tipos definidos por el usuario (constructores y operadores de conversión).

Considerando un método de un único argumento de entrada, la regla de coincidencias es simple. Se elige siempre la mejor coincidencia que es la más alta en la lista anterior. Si existen dos coincidencias que están en el mismo nivel de la lista, la llamada es ambigüa y por tanto, existe un error en tiempo de compilación.

Se necesita una regla algo más compleja que la anterior para soportar métodos con más de un argumento. Para éstos, un método se escoge respecto a otro con tal de que tenga un primer argumento con una mejor coincidencia (según la lista anterior) que otro método. Si todos los argumentos coinciden como mejor coincidencia, la llamada es ambigüa como en el caso de un único argumento.

Por ejemplo, dada la definición de una clase *"NumeroComplejo"* que se muestra en el algorítmo 3.98, cuyo fin es el de poder realizar calculos con números complejos[1].

Algoritmo 3.98 Clase "Complejo" con métodos sobrecargados

```
CLASS NumeroComplejo
  PUBLIC METHOD Suma (entero : INTEGER, doble : DOUBLE);
  PUBLIC METHOD Suma (Doble : DOUBLE, entero : INTEGER);
  PUBLIC METHOD Suma (obComlejo : Complejo, entero
        : INTEGER);
  PUBLIC METHOD Suma (entero : INTEGER, obComplejo
        : Complejo);
  PUBLIC METHOD Suma (obComplejo : NumeroComplejo,
    entero1 : INTEGER, entero2 : INTEGER);
END_CLASS
```

En el algorítmo 3.99 se muestra una serie de llamadas a la clase *"NumeroComplejo"* y a que método se llamaría una vez que el compilador resolviese las ambigüedades.

La limitación de doble a entero en la última llamada provoca un *"WARNING"* al compilar. Esta limitación es permitida en este caso ya que no se pierde precisión en la conversión y para preservar compatibilidad entre MIOOP y la norma IEC 61131, y no limitar así a los programadores, pero se emite un menaje de aviso para alertar al programador de la potencial fuente de errores. En este caso, el

[1]Para simplificar el ejemplo, sólo se muestran las cabeceras de las métodos junto con los parámetros de entrada de cada uno.

Algoritmo 3.99 Ejemplo de llamadas a métodos sobrecargados de la clase "Complejo"

```
FUNCTION_BLOCK LlamadaMétodosSobreCargaNumeroComplejo
  VAR_INPUT
    C : NumeroComplejo;
  END_VAR
    C.Suma (1, 2.0);  //Suma(entero, double)
    C.Suma (1.0, 2);  //Suma (double, entero)
    C.Suma (c, 1);  //Suma(Complejo, entero)
    C.Suma (c, 1, 3);  //Suma(Complejo, entero, entero)
    C.Suma (1, c);  //Suma(int, NumeroComplejo)
    C.Suma (1, 1);
    //ERROR. Llamada ambigua. ¿Suma(entero, double) ó
    //Suma(double, entero)?
END_FUNCTION_BLOCK
```

compilador debe escoger siempre aquella coincidencia que menos precisión pierda en la conversión.

3.25.2. Traducción en IEC 61131

En la comunidad informática existe cierto respeto a la utilización de la sobrecarga. Este miedo puede trasladarse a la Ingeniería de Control debido fundamentalmente a dos cuestiones:

1. A la posibilidad de que puedan ocurrir ambigüedades indetectables, lo que esta totalmente prohibido en la automatización de procesos.

2. A la posibilidad de que un programa no pueda ser correctamente compilado a menos que el programador declare explícitamente (en sintonía con la norma IEC 61131) que métodos van a ser sobrecargados.

Muchos de los problemas y miedos son resueltos gracias a la independencia en el orden de las reglas de sobrecarga, pudiendo delegar en el compilador la decisión de qué métodos son sobrecargados y liberando a los usuarios de tal acción.

Para mantener la compatibilidad con la norma IEC 61131 que dicta que no pueden existir dos FBs con el mismo nombre, el compilador debe añadir información

adicional en la traducción de los métodos sobrecargados. Así, en el momento de la compilación, a los nombres de los métodos sobrecargados se les añade la información necesaria en su nombre para distinguir unos de otros y evitar que tengan el mismo identificador.

En tiempo de compilación, cuando se traduce la definición de los métodos y se detectan ambigüedades, éstos son renombradas por medio de la síntaxis:

NombreClase_NombreMétodo _P_<ListaTiposParametros>

Donde:

- NombreClase identifica la clase a la que pertenece el método.
- _NombreMétodo identifica el método declarado por el usuario que es ambigüo por causa de una sobrecarga.
- _P identifica el comienzo de la lista de los tipos de parámetros de entrada del método.
- _<ListaTiposParametros> contiene la lista de parámetros que recibe el método, pudiendo ser de 1 a n. Esta lista tiene la forma:

  <b| y | c | s | i | d | r | l | w | uy | us | ui | ud | ul | t | a | p | e | o | h | f | d | g>

  Donde cada elemento identifica al tipo de dato de cada parámetro.
  Por ejemplo, la declaración del método:

```
METHOD Clase::Método (entero1 : INT, entero2 : INT,
       doble : DOUBLE)
```

  Se traduciría según esta regla a:

```
FUNCTION_BLOCK Clase_Método_P_iid
```

En la tabla 3.2 se muestra los elementos que identifican la parametrización de un método sobrecargado con los tipos de datos de la norma IEC 61131. El tipo *"P"* y el *"O"* son casos especiales, ya que IEC 61131 no permite trabajar directamente con punteros, ni reconoce las clases. Estos dos tipos se tratan igual que si se pasase

un valor numérico (en el caso de un puntero) o un tipo enumerado definido por el usuario (en el caso de un objeto). Su función es meramente informativa para que en caso de que el ingeniero acceda directamente al código generado tras la resolución de ambigüedades de concordancia por parte del compilador, se puedan identificar fácilmente los métodos que originaron las ambigüedades.

Nombre parámetro	**Tipo de dato**
B	BOOLEAN
Y	BYTE
S	SHORT INT
US	UNSIGNED SHORT INT
C	CHAR
I	INTEGER
UI	UNSIGNED INTEGER
D	DOUBLE
UD	UNSIGNED DOUBLE
R	REAL
L	LONG INTEGER
UL	UNSIGNED LONG INT
W	LONG REAL
T	STRING
A	ARRAY
P	PUNTERO
E	TYPE
O	CLASS
H	TIME
F	DATE
D	TIME_OF_DAY
G	DATE_AND_TIME

Cuadro 3.2: Parametrización de un método sobrecargado

A continuación se muestra un ejemplo de traducción de llamadas a métodos sobrecargados a IEC 61131. Dadas las siguientes declaraciones y las llamadas a dichas funciones que se muestran en el algorítmo 3.100. Las tres sentencias que se compilan correctamente de la función *"LlamadaDeSobrecarga"* se traducirían a código de la norma IEC 61131 como se muestra en el algorítmo 3.101.

Algoritmo 3.100 Ejemplo de clase y llamada sobrecargada en MIOOP

```
CLASS SobreCarga
  PUBLIC METHOD Método (entero : INTEGER);
  PUBLIC METHOD Método (doble : DOUBLE, entero : INTEGER);
  PUBLIC METHOD Método (entero : INTEGER, cadena : STRING);
END_CLASS

FUNCTION_BLOCK LlamadaDeSobrecarga
  VAR
    SC : SobreCarga;
  END_VAR
    SC.Método (1); //OK
    SC.Método (2.0, 1); //OK
    SC.Método (1, 'abc'); //OK
    SC.Método (1, 1);
    //Error de compilación. No se encuentra definido el
    //método
END_FUNCTION_BLOCK
```

Algoritmo 3.101 Ejemplo de clase y llamada sobrecargada en IEC 61131

```
FUNCTION_BLOCK LlamadaDeSobrecarga
  VAR
    SC : SobreCarga;
    Método_SobreCarga_P_i : Método_SobreCarga_P_i;
    Método_SobreCarga_P_di : Método_SobreCarga_P_di;
    Método_SobreCarga_P_it : Método_SobreCarga_P_it;
  END_VAR
    Método_SobreCarga_P_i (1);
    Método_SobreCarga_P_di (2.0, 1);
    Método_SobreCarga_P_it (1, 'abc');
END_FUNCTION_BLOCK
```

3.26. Sobrecarga de operadores

Al igual que sucede con los métodos, los operadores como la suma, multiplicación, etc, son también susceptibles de ser sobrecargados. Realmente, la norma IEC 61131 ya recoge la sobrecarga de operadores simples. Por ejemplo, el operador suma *"+"* permite sumar variables primitivas de tipo entero, doble, etc (aunque no se denomine explícitamente sobrecarga).

En POO, la sobrecarga de operadores se da cuando alguno o todos los operadores artiméticos (como +, -, =, etc) tienen diferentes implementaciones dependiendo del contexto en que se usan, es decir, dependen del tipo de dato de sus argumentos.

El principal objetivo de este tipo de sobrecarga es el de permitir a los usuarios especificar el significado de todo operador, tan largo como se haya definido la sentencia, y que no interfiera con la semántica predefinida. Ésta sería una tarea fácil si se permitiera la sobrecarga de todos los operadores sin excepción o deshabilitando todo operador que tenga un significado predefinido para el objeto de una clase. Los mayores problemas se encuentran relacionados con operadores que no se corresponden con el modelo usual o predefinido de los operadores binarios o aritméticos.

3.26.1. Definición en MIOOP

MIOOP permite al programador sobrecargar a su vez los operadores para sus propios usos o para sus propios tipos de clases por medio de la siguiente síntaxis:

<tipo> OPERATOR <NombreOperador>

(<ListaTiposParametros >)

Donde:

- Tipo identifica el tipo de variable u objeto donde se guarda el resultado de la operación sobrecargada.
- OPERATOR es la palabra reservada que sirve al compilador para identificar un operador sobrecargado y sustituir dicha operación por la llamada al método correspondiente que realiza la operación sobrecargada.

- <NombreOperador> identifica al operador de la norma IEC 61131 que se sobrecarga. Es necesario que este nombre coincida con el operador de la norma para que el compilador pueda sustituirlo por la llamada correcta.

- (<ListaTiposParametros>) son los parámetros que se admiten en la operación sobrecargada separados por comas, pudiendo ser de 1 a n.

Un operador sobrecargado, a efectos prácticos, no es más que un método de una clase que define el operador que recibe una serie de parámetros de entrada de cualquier tipo (tipos de datos primitivos y clases) y que debe ser definido, diseñado e implementado por el programador.

Como en el caso de los métodos sobrecargados de una clase, ésta es la forma en que se define un operador sobrecargado y que un compilador puede detectar cuando compila el código, pero dichas sentencias no son compatibles con IEC 61131. Para ello, en tiempo de compilación, cuando el compilador detecta que una operación está sobrecargada, sustituye dicha operación por una llamada al método que implementa la operación. Dicho FB es la traducción del método *"OPERATOR"* definido por el programador en la clase.

Por ejemplo, dada la declaración que se muestra en el algorítmo 3.102, una llamada al método "*OPERATOR +"* definido por la clase *"X"* quedaría como se ilustra en el algorítmo 3.103.

Algoritmo 3.102 Ejemplo de clase con sobrecarga de operadores

```
CLASS X
  X OPERATOR + (x1 : X, x2 : X);
END_CLASS
```

MIOOP también soporta la sobrecarga de varios operadores con el mismo nombre, es decir, por ejemplo, el ingeniero de automatización podría definir varias sobrecargas sobre el operador suma indicando en la lista de parametros diferente número y/o tipo de éstos. Las reglas que el compilador debe aplicar para solventar posibles ambigüedades, son las descritas en la sobrecarga de métodos (ver apartado 3.25) una vez aplicadas las reglas de sobrecarga de operadores.

Algoritmo 3.103 Ejemplo de llamada a un operador sobrecargado

```
FUNCTION_BLOCK FBDeSobreCarga
  VAR
    x : X;
    x1 : X;
    x2 : X;
  END_VAR
    x:=x1+x2;
END_FUNCTION_BLOCK
```

3.26.2. Traducción a IEC 61131

Los operadores sobrecargados son traducidos por MIOOP a IEC 61131 como métodos ordinarios de la clase con la siguiente sintaxis:

<Clase>_<Tipo>_OPERATOR_<NombreOperador>_
(<ListaTiposParametros >)

Donde:

- <Clase> identifica el nombre de la clase a la que pertenece el operador sobrecargado.
- _<Tipo> identifica el tipo de variable u objeto donde se guarda el resultado de la operación sobrecargada.
- _OPERATOR indica que el FB definido es un operador sobrecargado
- _<NombreOperador> identifica al operador de la norma IEC 61131 que se sobrecarga.
- _<ListaTiposParametros> contiene la lista de parámetros que recibe el FB que implementa el método, pudiendo ser de 1 a n. Esta lista tiene la forma y es idéntica a la utilizada para la sobrecarga de métodos del apartado 3.25:

  <b| y | c | s | i | d | r | l | w | uy | us | ui | ud | ul | t | a | p | e | o | h | f | d | g>

Bajo esta definición, en el ejemplo ilustrado en el algorítmo 3.102 del apartado anterior, el compilador entiende que siempre que se aplique una suma para un objeto de la clase *"X"*, lo que se quiere es utilizar la operación sobrecargada definida por el usuario, con lo que la función del algorítmo 3.103 se traduciría a la norma IEC 61131 como se muestra en el algorítmo 3.104.

Algoritmo 3.104 Traducción de llamada a un operador sobrecargado en IEC 61131

```
FUNCTION_BLOCK FBDeSobreCarga
  VAR
    x : X;
    x1 : X;
    x2 : X;
    X_X_Operator_+_oo : X_X_Operator_+_oo;
    //FB de sobrecarga para la suma
  END_VAR
    x:=X_X_Operator_+_oo (x1,x2);
END_FUNCTION_BLOCK
```

3.27. Sobrecarga del operador de asignación

La sobrecarga del operador de asignación *"="* es un caso particular de sobrecarga de operadores que permite invocar el método que redefine la asignación en toda operación que incluya la clase que lo sobrecarga. Esta operación resulta útil cuando la asignación simple no es adecuada o incompleta en el trabajo con objetos.

3.27.1. Definición en MIOOP

En la norma IEC 61131, la inicialización y asignación de variables son por defecto definidos como una copia a nivel de bit. Ésto provoca grandes problemas cuando un objeto de una clase con asignación sobrecargada es usado como un miembro de una clase que no tiene definida la asignación (ver algorítmo 3.105).

Con la instrucción *"X1:=X2"* del algorítmo 3.105, lo que se hace es copiar el objeto *"X2"* en *"X1"* bit a bit. Ésto evidentemente es un error, ya que no hay forma de copiar las clases agregadas. Por tanto, es necesario ampliar el concepto de copia

Algoritmo 3.105 Ejemplo de sobrecarga de asignación

```
CLASS X
  PUBLIC Numero : INTEGER;
  PUBLIC ObjetoY : Y ;
  //Agregación de un objeto de la clase Y
  X OPERATOR = (x1 : X, x2 : X)
END_CLASS

CLASS Y
  PUBLIC Letra : STRING;
END_CLASS

METHOD X::Operator=(X : x1, X : x2);
  x1:=x2;
END_METHOD
```

de objetos y entenderla más como una copia de miembros que como una copia bit a bit, en la que es necesario tener sobrecargada la operación asignación en los miembros agregados y llamar a dicho método.

Esta nueva definición obliga a redefinir el anterior operador de asignación e indicar de alguna manera al compilador que no se desea hacer una copia bit a bit. Ésto se hace indicando que el objeto está realizando una operación sobrecargada por medio de la palabra reservada de clase *"OPERATOR"*, lo que permite realizar una asignación bit a bit entre todos los atributos de la clase de tipo simple y llamar a los operadores sobrecargados de las clases agregadas (ver algorítmo 3.106).

Algoritmo 3.106 Ejemplo de sobrecarga del operador asignación en MIOOP

```
METHOD X::Operator=(X : x1, X : x2);
  x1.OPERATOR:=x2.OPERATOR;
END_METHOD
```

Hay que tener en cuenta que la no existencia de la operación de sobrecarga en la clase agregada mostraría un error de compilación.

3.27.2. Traducción a IEC 61131

El compilador, al encontrar la instrucción de asignación entre dos objetos que poseen la palabra reservada *"OPERATOR"* para sobrecargar la asignación, sustituye dicha sentencia por una asignación normal de las estructuras que implementan los objetos que se están asignando y posteriormente, realiza las llamadas a cada uno de los operadores sobrecargados de las clases agregadas.

Esta definición se ve claramente al traducir a IEC 61131 el algorítmo 3.106 del apartado anterior (ver algorítmo 3.107).

Algoritmo 3.107 Traducción de sobrecarga del operador de asignacion en IEC 61131

```
FUNCTION_BLOCK X_X_Operator_=_oo
  VAR_IN_OUT
    THIS : X;
    X2 : X;
  END_VAR
  VAR
    Y_Y_Operator_oo : Y_Y_Operator_oo;
    //FB de sobrecarga de la clase Y
  END_VAR
    THIS:=X2;
    Y_Y_Operator_oo(THIS.ObjetoY, X2.ObjetoY);
END_FUNCTION_BLOCK

FUNCTION_BLOCK Y_Y_Operator_=_oo
  VAR_IN_OUT
    THIS : Y;
    Y2 : Y;
  END_VAR
    THIS:=Y2;
    Y_Y_Operator_oo(THIS.ObjetoY, X2.ObjetoY);
END_FUNCTION_BLOCK
```

3.28. Operadores unitarios / Sobrecarga de operadores unitarios

Una de las mayores ventajas aritméticas que poseen algunos lenguajes como C, son las notaciones preincremento, postincremento, predecremento y postdecremento. Su funcionamiento se basa en la suma o resta por el mismo valor de la variable, alojando el resultado de la operación en dicha variable, pero con sutiles diferencias, ya que los preincrementos y predecrementos actualizan el valor de la variable sobre la que actúan antes de realizar cualquier operación adicional con la variable, y los posincrementos y posdecrementos realizan operaciones adicionales con la variable antes de actualizar su valor (ver algorítmo 3.108).

Algoritmo 3.108 Ejemplo de sobrecarga de operadores unitarios

```
a:=0;
b:=++a;
//Se incrementa primero "a" y luego se asigna a "b".
//a=b=1
b:=a++;
//Se asigna a "b" el valor de "a" y luego se incrementa
//este. a=2 y b=1
```

3.28.1. Definición en MIOOP

MIOOP proporciona este tipo de operaciones porque resultan útiles para minimizar el número de líneas de código en operaciones aritméticas, al condensar varias operaciones en una única línea (ver algorítmo 3.108) y no van encontra de la filosofía restrictiva de la norma. Para ello, cuando el compilador detecta una operación unitaria, sustituye dicha sentencia por una equivalente en la norma IEC 61131 de forma totalmente transparente al ingeniero programador.

Este tipo de operadores es también susceptible de ser sobrecargado. Para definirlo, es necesario añadir algún tipo de sintaxis que permita que el compilador identifique qué tipo de operación (prefija o postfija) se ha elegido. Para ello, se puede optar por añadir dos nuevas palabras reservadas como *"Prefija"* y *"Postfija"* que identifiquen inequívocamente cada acción (ver algorítmo 3.109).

Algoritmo 3.109 Notación "_PreFija" y "_PostFija" en una clase

```
CLASS X
  X OPERATOR PreFija++ ( );
  X OPERATOR PostFija++ ( );
END_CLASS
```

Sin embargo, esta notación requiere que el usuario aprenda estas dos nuevas palabras reservadas. Además, para aquellos ingenieros acostumbrados a programar en C, resulta relativamente fácil comprender el concepto de los operadores unitarios sin añadir aditivos al lenguaje.

Para un compilador resulta sencillo diferenciar entre notaciones prefijas o postfijas cuando se está traduciendo el código. En consecuencia, es posible definir la sobrecarga unitaria de operadores de la misma forma en que se definen el resto de operadores sobrecargados.

Bajo este nuevo concepto, la sobrecarga de operadores unitaria se definiría como se muestra en el algorítmo 3.110.

Algoritmo 3.110 Notación "_PreFija" y "_PostFija" en MIOOP

```
CLASS X
  X OPERATOR ++ ( );
  X ++ OPERATOR ( );
END_CLASS
```

3.28.2. Traducción a IEC 61131

La norma IEC 61131 no proporciona operaciones de tipo unitarias, por lo que deben ser traducidas en tiempo de compilación a las sentencias correspondientes estándar. Por ejemplo, el algorítmo 3.108 se traduciría a la norma IEC 61131 como el algorítmo 3.111.

Respecto a la traducción de la sobrecarga de este tipo de operador, se sigue las mismas reglas descritas que en el apartado 3.26.

Algoritmo 3.111 Traducción de la sobre carga de operadores unitarios en IEC 61131

```
a:=0;
a:=a+1;
b:=a;
//Equivaldría a: b:=++a;
b:=a;
a:=a+1;
//Equivaldría a: b:=a++;
```

3.29. Conclusiones

A lo largo de este capítulo se ha presentado MIOOP (Modification of the IEC 61131 standard for Object Oriented Programming), un nuevo conjunto de modificaciones sobre la norma IEC 61131 que permiten la evolución del paradigma de programación estructurada, en el que se basa el estándar, al paradigma de orientación a objetos, así como todos los mecanismos necesarios para su traducción.

Su principal innovación es la de ofrecer a los ingenieros de automatización toda la potencia de la POO y soportar los pilares básicos de éste: abstracción, encapsulamiento, modularidad, jerarquía y polimorfismo.

La definición de las características de MIOOP se divide en dos partes fundamentalmente:

- El qué. El conjunto de ampliaciones que se proponen al estándar IEC 61131, de manera que se dé soporte al paradigma de programación orientado a objetos.
- El cómo: Las modificaciones necesarias en la norma IEC 61131 que permitan traducir las mejoras propuestas por MIOOP a código estándar de la norma de manera automática.

Finalmente, MIOOP se completa con una herramienta software denominada SimPLC++ (descrita en el capítulo 4) que permite la programación de sistemas de control basados en la norma IEC 61131 ampliada con las características del paradigma OO descritas en este capítulo. Además, SimPLC++ incorpora un conjunto de herramientas que la convierten en un completo banco de ensayos que permite comprobar

empíricamente la diferencia de rendimiento entre un estilo de programación estructurado y un estilo de programación OO.

Capítulo 4

Banco de ensayos (SimPLC++)

Aquellos que prefieren los sables largos poseen sus propias razones, pero únicamente es lógico para ellos. Desde el punto de vista de la auténtica vía del mundo, ésto es ilógico. ¿Es inevitable perder utilizando un sable más corto y no un sable muy largo?
Miyamoto Musashi (Anillo del viento)

4.1. Introducción

SimPLC++ (Simulador de PLC orientado a objetos) es el nombre que recibe la herramienta software que permite el diseño y ejecución de programas de control para PLCs basados en la norma IEC 61131 y en la ampliación propuesta por MIOOP.

La herramienta SimPLC++ nace de la necesidad detectada dentro del grupo de investigación *GENIA* de la *Universidad de Oviedo*, de proporcionar a los alumnos de ingeniería una herramienta para el aprendizaje de los diversos lenguajes y

métodologías existentes para el desarrollo de proyectos de automatización basados en autómatas programables compatibles con la norma IEC 61131-3 y con las ampliaciones introducidos por MIOOP.

SimPLC++ se concibe como una herramienta dividida en 5 módulos que interactúan entre sí para brindar a los alumnos e investigadores, un banco de aprendizaje y ensayos de programas de control sobre la norma IEC 61131 y las ampliaciones propuestas por MIOOP (ver figura 4.1):

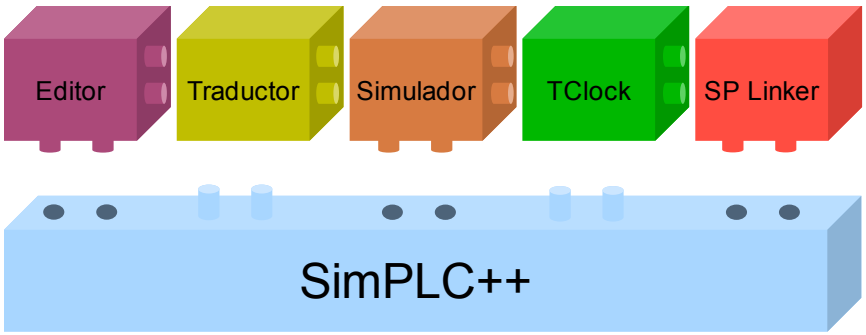

Figura 4.1: División de SIMPLC++ en módulos

1. Editor de programas. SimPLC++ facilita un editor para el desarrollo de programas de control bajo los cinco lenguajes definidos en la norma IEC 61131 (IL, LD, ST, FBD, SFC). Así mismo, SimPLC++ soporta las ampliaciones a la norma propuestas por MIOOP proporcionando una ampliación de los 5 lenguajes de la norma IEC 61131 (IL++, LD++, ST++, FBD++ y SFC++).

2. Traductor. SimPLC++ permite implementar un programa de control bajo el paradigma orientado a objetos, sin embargo la norma IEC 61131 no reconoce dicho paradigma. Es por ello que siempre que se escribe un programa de control, SimPLC++ traduce todo el código a un lenguaje que cualquier PLC basado en IEC 61131 reconozca. Por convenio, este lenguaje se ha decidido que sea IL ya que es el lenguaje de más bajo nivel de la norma.

3. TCLOCK. Para poder hacer una comparativa objetiva de la diferencia entre un programa codificado bajo un POO y el mismo programa implementado siguiendo un paradigma estructurado, SimPLC++ permite la introducción

de directrices de control de tiempos de los ciclos de SCAN de los programas y sacar dichos resultados a un fichero de texto para su posterior análisis.

4. Simulador de programas. Como herramienta de aprendizaje, SimPLC++ permite la simulación de la ejecución de los programas de control que se confeccionan.
   La simulación ofrece enormes ventajas en el proceso docente. A través de esta técnica se puede lograr emular, reproducir o replicar con un alto grado de similitud, procesos, objetos y fenómenos del mundo real, algunos de los cuales no podrían ser presentados a los alumnos, mucho menos permitirían interactuar con ellos [AT91]. Entre las características principales de los simuladores, según Alessi y Trollip, se pueden destacar:

   a) Aumento de la motivación, ya que en la misma, el estudiante es un participante activo, lo que posibilita aplicar el principio de *"aprender haciendo"*.

   b) Ayuda a construir un modelo mental de parte del mundo real y da la posibilidad de probarlo sin riesgos. Sirve, en ocasiones, para eliminar complicaciones que podrían oscurecer la compresión de principios más importantes como por ejemplo, poner a voluntad escalas de tiempo, lo cual permitiría trabajar con procesos cuya ocurrencia en la vida real sea por escaso tiempo o con una lentitud muy grande para ser observados.

   c) Es el único modo de suministrar al alumno una visión segura y a un costo razonable de ciertos fenómenos que de otra manera sería inviable.

5. SP Linker. SimPLC++ incorpora un módulo capaz de crear un patrón XML a partír del código IL generado por el módulo traductor que puede ser leído por el módulo UDE del fabricante Schneider y de esta forma, poder exportar un proyecto desarrollado en SimPLC++ a un PLC real de dicha marca.

SimPLC++ se engloba dentro de un proyecto más ambicioso del grupo GENIA mediante el cual se pretende dotar al Laboratorio de Automatización Virtual (LAV) de las herramientas necesarias para la programación y simulación de controladores lógicos programables a bajo coste. LAV es el nombre de la herramienta software que permite la aplicación sistemática de la Métodología MLAV [Gon02]. Su función

es la de integrar todas las fases de desarrollo de un proyecto de automatización (análisis, diseño, programación, documentos, etc) en un único *"Entorno Virtual de Ingeniería"* (VEE), lo que permite mantener toda la información del proyecto centralizada, sirviendo además como punto de encuentro de los distintos grupos de ingenieros que toman parte en el desarrollo de un proyecto de automatización, ayudando a eliminar las barreras de entendimiento existentes entre los mismos y facilitando la toma de decisiones [Gru99].

En los siguientes apartados se describe la arquitectura interna de SimPLC++ atendiendo a las cinco partes en que ésta herramienta se divide. Por otro lado, en el anexo A se hace un pequeño recorrico sobre los distintos equipos de control que se pueden utilizar para la automatización de procesos y se detalla un breve resumen de la norma IEC 61131.

4.2. Módulo editor

El módulo de edición consiste básicamente en cinco editores de programas de control, uno por cada tipo de lenguaje recogido en la norma IEC 61131. Dos de ellos son editores de texto, es decir, permiten al usuario escribir las instrucciones que constituirán un programa de control usando el teclado. Los otros tres son editores gráficos que proporcionan al usuario los mecanismos de selección, configuración y conexión de las instrucciones que han de formar el programa de control de forma gráfica.

Un programa de control puede ser implementado en cualquiera de los 5 lenguajes de la norma (ver sección A.4.3 para una descripción más detallada de estos lenguajes) e incluso mezclándolos según las necesidades o gustos del usuario. En cualquier caso, todos ellos permiten guardar el trabajo editado hasta el momento y recuperar ediciones hechas con anterioridad.

El módulo de edición permite además crear o editar la estructura del programa, ésto es, los POUs necesarios del tipo deseado, conteniendo las instrucciones y variables que el usuario estime oportuno.

Igualmente, permite crear o editar el módulo de ejecución del programa. Ésto incluye determinar cuántas configuraciones tiene un PLC, cuántos recursos por cada configuración, cuántas tareas en cada recurso y finalmente, instanciar los POUs y asociar esas instancias con las tareas deseadas.

Por supuesto, el módulo de edición brinda al usuario los mecanismos para poder crear los tipos de datos que desee y las tablas de variables globales a las configuraciones, a los recursos y a los programas que sea menester.

4.2.1. Tipos de POUs

El módulo editor reconoce 4 tipos de POUs. Tres de ellos son los proporcionados en la norma IEC 61131, a saber, funciones, bloques funcionales y programas. El cuarto tipo de POU es exclusivo de SimPLC++ y se denomina *"CLASS"*.

Para poder utilizar en cualquier proyecto un POU creado con anterioridad, SimPLC++ posee una base de datos de todas los POUs que el usuario ha ido creando a modo de biblioteca de componentes, como se observa en la figura 4.2.

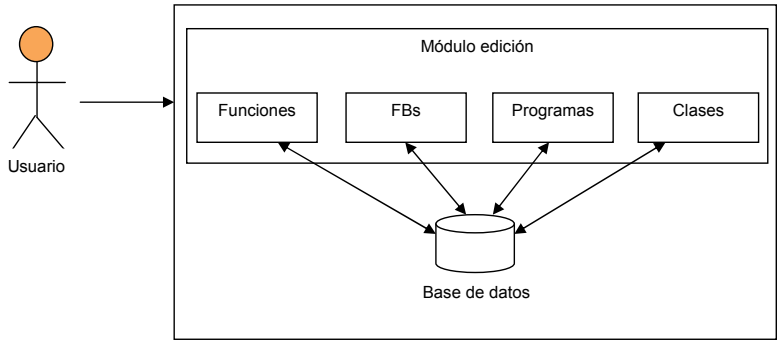

Figura 4.2: Biblioteca de componentes de SimPLC++

Funciones

El módulo de edición permite crear funciones definidas por el usuario. Estas funciones, al igual que las definidas en la norma IEC 61131, no pueden contener ninguna información de estado interno, es decir, que la invocación de una función con los mismos argumentos (parámetros de entrada) debe suministrar siempre el mismo valor (salida).

El módulo de edición permite crear funciones en cualquiera de los lenguajes soportados por IEC 61131 salvo en SFC, ya que la norma no permite su invocación desde

este tipo de lenguaje. También se permite la posibilidad de llamar a otras funciones, pero al igual que ocurre con la norma IEC 61131, SimPLC++ no permite crear instancias a FBs u objetos desde su código interno.

La invocación de una función desde el lenguaje ST se realiza escribiendo el nombre de ésta junto con los parámetros de entrada de la misma y asignando su valor de retorno a otra variable, dirección de memoria o salida física. En IL se apilan previamente las variables que se pasan como parámetros y posteriormente se escribe la palabra reservada *"CAL"* seguida del nombre de la función. En los lenguajes gráficos, SimPLC++ asocia una imagen con el nombre de la función con tantas líneas de entrada como parámetros tenga y una única línea de salida, como se observa en la figura 4.3. Es necesario conectar todas las líneas definidas en la función para que se pueda compilar el programa.

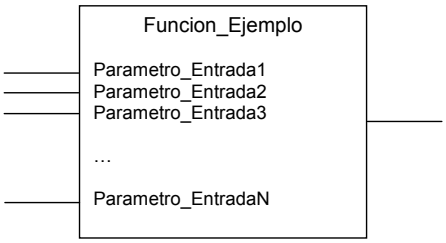

Figura 4.3: Representación gráfica de una función en SimPLC++

Además de las funciones que el usuario puede crear, SimPLC+ proporciona una serie de funciones predefinidas (como funciones aritméticas, de manejo de strings, etc) que pueden ser invocadas en cualquier momento.

Bloques funcionales (FB)

Como en el caso de las funciones, el módulo de edición permite al programador crear sus propios FBs y utilizarlos en cualquier parte del programa de control o desde otro POU. Estos FBs, al contrario que las funciones, son un tipo de POU que posee memoria interna lo que provoca que no tengan un comportamiento determinista.

El tratamiento que el módulo de edición da a los FBs es idéntico al de las funciones con la diferencia que en este caso es preciso instanciar previamente el FB mediante una variable. Para su invocación, desde los lenguajes textuales se debe escribir el

nombre de éste junto a sus parámetros de entrada y de salida. En los lenguajes gráficos, al igual que con las funciones, SimPLC++ asocia una imagen con el nombre del FB que posee tantas líneas de entrada como parámetros tenga de entrada, y tantas líneas de salida como parámetros de salida (ver figura 4.4). Por otro lado, es necesario conectar todas las líneas definidas en el FB para que SimPLC++ no lance un mensaje de error cuando compila el programa.

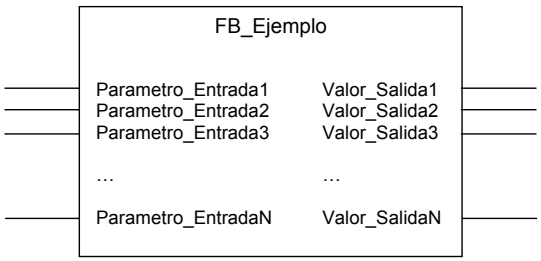

Figura 4.4: Representación gráfica de un FB en SimPLC++

Además de los FBs que el usuario puede crear, SimPLC+ proporciona una serie de FBs predefinidos tales como contadores, temporizadores, etc.

Programas

El módulo editor trata los programas de la misma forma que en la norma IEC 61131, dándoles la misma consideración que un FB. Sin embargo y a diferencia de éstos, no se permite crear instancias de un programa por lo que para su utilización es necesario asociarlos a tareas. Bajo esta definición, el módulo editor permite asociar n programas a una tarea.

Clases

Este nuevo tipo de POU encapsula las características más básicas propuestas en MIOOP para que la norma IEC 61131 soporte la orientación a objetos. Su objetivo es por tanto la definición de las clases que serán instanciadas desde otros POUs en cualquiera de los lenguajes soportados por SimPLC++.

El POU clases se subdivide a su vez en 3 partes claramente diferenciadas:

- Clase/s de las que se hereda. SimPLC++ soporta herencia múltiple y para su inclusión se permite asociar la lista de clases heredables separadas por comas en la cabecera de la clase que se está definiendo.
- Atributos. Vienen a ser una analogía a las variables locales de un FB con la diferencia de que en este caso se puede indicar el nivel de protección que el ingeniero programador desee, a saber, público, privado, protegido y amigo. Para saber más sobre los niveles de protección soportados por MIOOP ver el apartado 3.9.
- Métodos. Los métodos son los servicios que proporciona la clase y mediante los cuales se interactúa con los atributos de la misma. Del mismo modo que con los atributos, en el caso de los métodos también es posible indicar el nivel de protección que se desee para cada uno.

Para poder utilizar una clase es necesario instanciarla previamente. Esta instanciación se realiza por medio de una variable que comúnmente se denomina *"objeto"*. Al igual que ocurre con los FBs, los objetos tienen la capacidad de recordar su estado, es decir, el valor de sus atributos mientras exista en memoria.

Los POUs clase pueden ser programados en cualquiera de los 5 lenguajes de la norma IEC 61131 y a su vez, pueden ser utilizados en cualquier momento por un programa, un FB u otras clases. SimPLC++ sólo limita su utlización dentro de las funciones debido a que la norma IEC 61131 no permite instanciar FBs desde ellas ya que un FB guarda el estado de sus variables internas. Los objetos, al igual que los FBs, deben conservar el estado de sus atributos durante todo su ciclo de vida. Por tanto, y siguiendo la filosofía restrictiva del estándar, SimPLC++ restringe también el uso de objetos dentro de funciones.

A diferencia del POU FB que únicamente encapsula atributos y código, el POU clase permite encapsular atributos y un conjunto de servicios que pueden operar sobre dichos atributos (los métodos), así como un conjunto de características propias del paradigma orientado a objetos (herencia, polimorfismo, agregación, etc). Buscando una similitud entre un FB y una clase, se podría decir que una clase contiene un conjunto de FBs que pueden interactuar con los atributos definidos en la clase.

Otra característica del POU clase que no poseen los otros tres POUs de la norma IEC 61131, es la capacidad de poder usar toda la funcionalidad de otra clase.

Este recurso se logra por medio de la herencia en la que una clase padre marca como heredables los métodos y atributos que se crean necesarios. La clase hija que hereda podrá acceder a ellos sin necesidad de tener que volver a programarlos. Para marcar la herencia, SimPLC++ permite indicar entre paréntesis la/s clase/s de las que se hereda separadas por comas.

El acceso a los atributos y métodos de un objeto en los lenguajes textuales se hace por medio del nombre de dicho objeto seguido del operador *"."* y el atributo/servicio solicitado. En los lenguajes gráficos, el módulo editor asocia a cada método de la clase una imagen cuyo nombre es el de la clase seguido del operador *"."* y el nombre del método. Dicha imagen contiene tantas líneas de entrada como parámetros de entrada tenga el método y tantas líneas de salida como parámetros de salida se definan en él, como se observa en la figura 4.5.

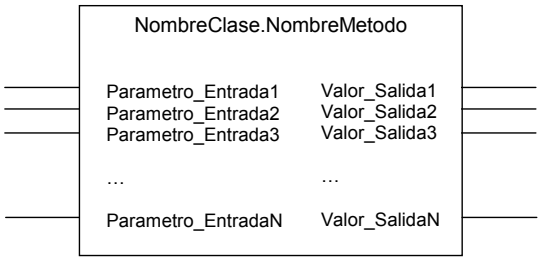

Figura 4.5: Representación gráfica de un método en SimPLC++

4.2.2. POUs estándar

La norma IEC 61131 define una serie funciones y FBs estándar comunes en todos los lenguajes. El módulo editor reconoce los mismos elementos comunes ya programados y accesibles de diferentes formas tanto en los lenguajes de la norma como en sus versiones orientadas a objetos. Estos POUs estándar se guardan en una librería a modo de base de datos junto con las palabras reservadas e identificadores reconocidos por el módulo editor de SimPLC++ cuya funcionalidad es doble, tal y como se observa en la figura 4.6. Por un lado, sirven como una biblioteca de utilidades para el programador en la que se encuentran todos los elementos comunes y estructuras de programación de SimPLC++, y por otro lado, sirve al módulo traductor para generar su tabla de símbolos cuando *"parsea"* el código del programa de control.

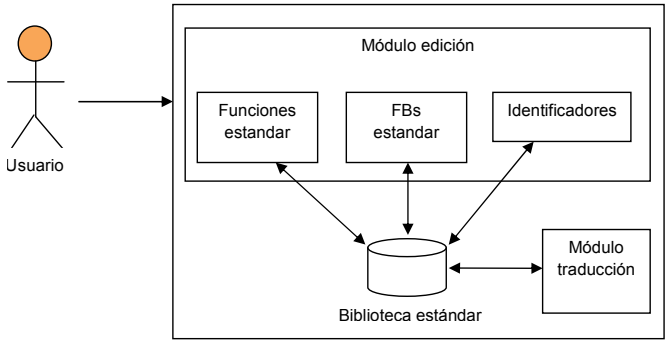

Figura 4.6: Biblioteca de POUs estandar

4.2.3. Elementos comunes

El módulo editor reconoce todos los tipos de datos simples (INTEGER, BOOLEAN, STRING, etc) recogidos en la norma IEC 61131. Así mismo, el módulo editor amplia el tipo ARRAY de la norma permitiendo la definición de ARRAYs de objetos tal y como se implementan en MIOOP (ver apartado 3.20). La definición de este nuevo tipo de ARRAY es análogo al ARRAY de la norma IEC 61131, con la diferencia de que en éste, lo que se asigna a cada posición del ARRAY es un puntero a la direcciones de memoria de cada objeto. Esta asignación es **totalmente transparente** al programador cuya única obligación es la de indicar el tipo de objeto que desea insertar en el ARRAY. El módulo traductor es el encargado de insertar el código necesario para que esta asignación sea coherente con el modelo OO de SimPLC++ propuesto por MIOOP.

Aunque el ARRAY de objetos funciona por medio de punteros a objetos y por tanto SimPLC++ soporta el manejo de punteros, el módulo editor prohíbe la definición y utilización de punteros por parte del programador. Los punteros son, por tanto, un recurso interno no disponible para el programador.

4.2.4. Tipos de secuencias

Dado el carácter secuencial del tipo de procesos a los que se circunscribe MIOOP, SimPLC++ presenta 5 tipos de secuencias comunes para los 5 lenguajes de la norma IEC 61131 y sus ampliaciones OO.

Cada una de estas secuencias puede ser representada por un GRAFCET en el que las acciones asociadas a las etapas se vinculan con los preaccionadores/accionadores del proceso, y los cambios de estado de los sensores serán las receptividades de las transiciones que permitirán la evolución de una etapa a otra. Cada una de estas secuencias puede estar organizada de varias formas distintas como se recoge en el estándar internacional IEC 848 [IEC88].

Para un mejor entendimiento, los tipos de secuencias serán representadas por un GRAFCET que controla el proceso del ciclo de llenado de un objeto *"Tanque"* cuya representación en una tarjeta CRC se muestra en la figura 4.7.

Figura 4.7: Tarjeta CRC de una clase *"Tanque"*

Inicialmente las dos electroválvulas se encontrarán cerradas. Al comienzo del proceso, el ingeniero forzará la apertura de la electroválvula de entrada del tanque. Para ello, el objeto *"Tanque"* proporciona el servicio *"Llenar"*. Ésto provocará que el tanque se comience a llenar, lo que al cabo de cierto tiempo provocará la activación del sensor de nivel *"Máximo"*. Cuando el ingeniero observe este hecho, forzará la electroválvula de entrada para que se cierre por medio del servicio *"Tapar"*, quedándose a la espera de que por efecto del consumo de líquido del tanque, que no se puede controlar, se vacíe el mismo, lo que provocará que el sensor de nivel *"Mínimo"* se desactive. Es en este momento cuando se dan las condiciones iniciales y el ciclo de control converjería de nuevo.

Secuencias lineales

Es el caso más sencillo de todos. Consiste en la implementación secuencial de un conjunto de instrucciones. Primero se ejecuta la que el ingeniero estime que debe ser ejecutada antes. Cuando ésta finalice, se pasa a ejecutar la siguiente y así sucesivamente hasta que se termine con la ejecución de la última. La representación

en lenguaje GRAFCET de este tipo de secuencias se muestra en la figura 4.8.

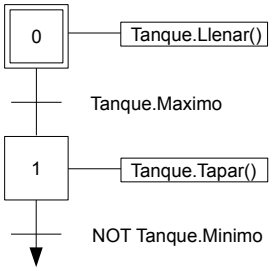

Figura 4.8: Secuencia lineal

Secuencias paralelas

Cuando el ingeniero de proceso determina que varias secuencias de ejecución de servicios deben realizarse *"simultáneamente"*, esto significa que esas secuencias deben ser ejecutadas en paralelo. La representación de este tipo de estructura en lenguaje GRAFCET recibe el nombre de *"paralelismo estructural"* y se expresa como se muestra en la figura 4.9.

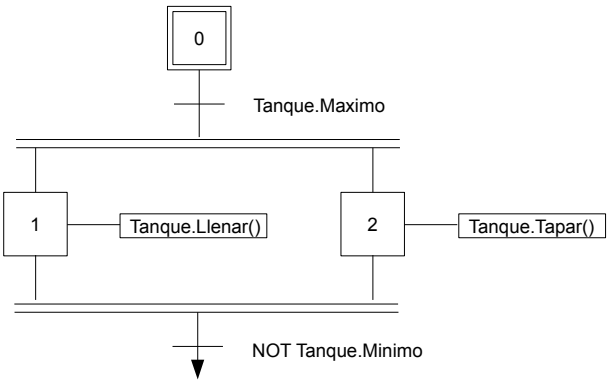

Figura 4.9: Secuencia paralela

No se puede asegurar que la ejecución de las etapas que constituyen las distintas ramas del paralelismo estructural tarden el mismo tiempo, o lo que es lo mismo, no se puede asegurar que se alcance la última etapa de cada una de las secuencias que constituyen el paralelismo al mismo tiempo. Sin embargo, la norma IEC 848

[IEC88] exige la unicidad de modo de ejecución para evitar inconsistencias, lo que implica que es necesario asegurar que todas las secuencias que constituyen el paralelismo han alcanzado la última etapa antes de permitir la evolución hacia una etapa externa al paralelismo.

Secuencias alternativas

Cuando el experto en el proceso indica que un conjunto de servicios o de secuencias de servicios, al menos dos, *"no pueden ejecutarse a la vez"*, el ingeniero de automatización puede pensar en ejecutarlos secuencialmente. Si el experto en el proceso indica que no a dicha sugerencia, la única posibilidad que resta es que sean alternativos o mutuamente excluyentes, es decir, que sólo una de ellas se puede ejecutar en un momento dado, y tras su ejecución se pase a ejecutar otra etapa o secuencia de etapas no perteneciente a este conjunto. Se trata pues de tomar una decisión acerca de cuál de esos servicios o de esas secuencias se debe ejecutar. La toma de esta decisión dependerá del valor de una condición lógica para cada una de las ramas alternativas de la secuencia. Estas condiciones deben ser mutuamente excluyentes para asegurar que no se podrán ejecutar servicios de más de una rama simultánea. De lo contrario, se estaría en el caso de secuencias paralelas. La representación en lenguaje GRAFCET de este tipo de secuencias se muestra en la figura 4.10 y equivaldría a una instrucción IF-THEN-ELSE en lenguaje ST.

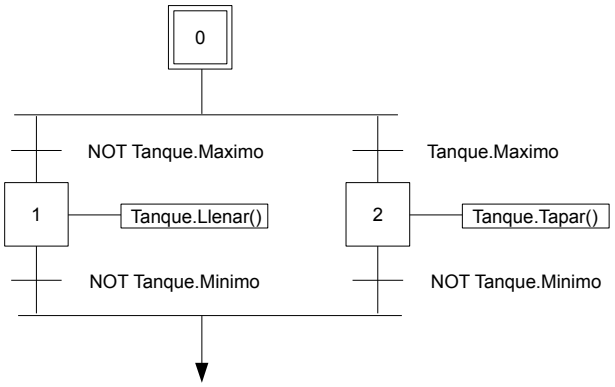

Figura 4.10: Secuencias alternativas

Secuencias cíclicas

En ocasiones, la estructura de una secuencia se repite de manera cíclica. Ésto es, cuando se alcanza el último estado de la secuencia y se verifica una determinada condición, se comienza de nuevo la secuencia por alguno de los estados anteriores al último, generalmente el primero.

La representación en lenguaje GRAFCET de este tipo de secuencias es la que aparece en la figura 4.11. Un caso particular de las secuencias cíclicas son las secuencias iterativas que se estudian en la siguiente sección.

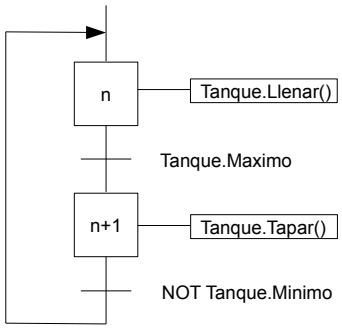

Figura 4.11: Secuencias cíclicas

Para evitar la inserción de líneas que puedan dificultar la lectura del programa de control, SimPLC++ permite secuencias cíclicas sin la necesidad de unir físicamente una transición con una etapa por medio de una línea. En su lugar puede asociarse una secuencia cíclica a una etiqueta cuyo nombre coincide con la etapa a la que se quiere unir la transición, como se muestra en la figura 4.12.

Secuencias iterativas

Las secuencias iterativas son un caso particular de las secuencias cíclicas en las que el número de veces que se repite la secuencia es finito. La utilización de este tipo de secuencias responde a la necesidad expresada por el experto del proceso de aplicar un servicio o conjunto de servicios u operaciones de manera reiterada y consecutiva un número determinado de veces.

El número de veces que se ejecutará esta secuencia de operaciones vendrá dado por:

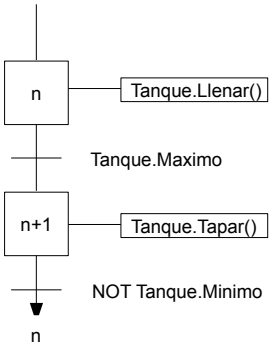

Figura 4.12: Secuencia cíclica con salto a etiqueta lejana

1. Un número determinado de antemano y fijado por el experto.
2. *"Mientras que"* o *"Hasta que"* no se produzca una determinada condición.

Este tipo de estructura suele ser empleada para aplicar las mimas operaciones sobre un grupo de datos distinto en cada iteración, sin necesidad de tener que escribir las mismas operaciones tantas veces como las queramos aplicar cambiando solamente los datos sobre los que se aplica.

Existen tres tipos de secuencias iterativas cuya representación en lenguaje GRAFCET es la que aparece en las figuras 4.13, 4.14, y cuya interpretación es la siguiente:

- Bucle del tipo *"Para"*, ver figura 4.13. Equivaldría a un bucle FOR en lenguaje ST. En este tipo de bucle las operaciones son aplicadas un número finito y determinado de veces. Para ello es necesario llevar la cuenta del número de veces que se han aplicado las operaciones.
- Bucle del tipo *"Hasta que"*, ver figura 4.14. Equivaldría a un bucle DO-UNTIL en lenguaje ST. En este tipo de bucle se asegura que las tareas propias del bucle se ejecutan al menos una vez, ya que la condición de *"Fin de Bucle"* no es analizada *"Hasta Que"* estas tareas no se hayan realizado.
- Bucle del tipo *"Mientras que"*. Equivaldría a un bucle WHILE en lenguaje ST. En este otro tipo de bucle, las tareas propias del bucle sólo son realizadas *"Mientras Que"* la condición de fin de bucle sea cierta. Si la primera vez que se evalúa la condición esta da como resultado falso, entonces el bucle no

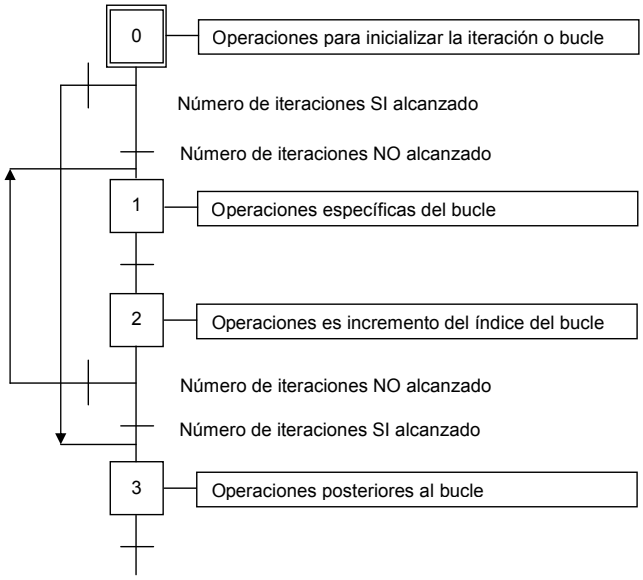

Figura 4.13: Bucle de tipo *"para"*

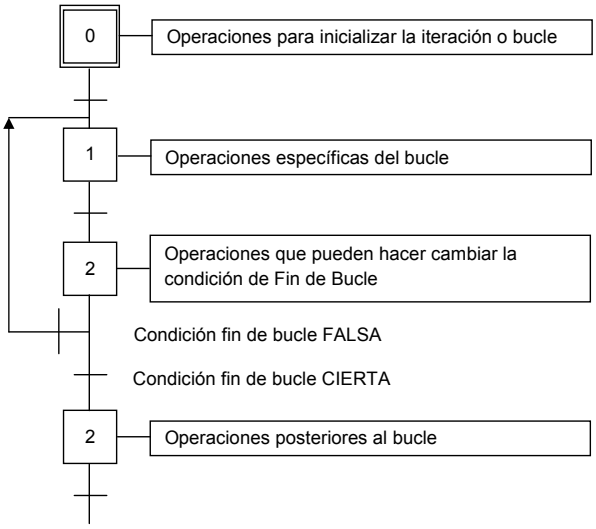

Figura 4.14: Bucle de tipo *"Hasta que"*

se ejecutará y por tanto las tareas propias del bucle tampoco. Este tipo de bucle se parece al bucle *"Para"* con la diferencia que el bucle *"Mientras que"* se ejecuta al menos una vez.

4.2.5. Operaciones básicas

El módulo editor incorpora una serie de operaciones frecuentemente usadas por los ingenieros de automatización y accesibles desde los todos los lenguajes soportados por SimPLC++ que se detallan a continuación.

Operaciones lógicas

La evolución del proceso entre dos estados consecutivos se produce por el cambio de valor en alguna variable del proceso. Puede tratarse de una sola variable o de más de una, en cuyo caso es necesario especificar la combinación lógica de dos o más variables por medio de paréntesis.

Las condiciones AND, OR y NOT son suficientes para representar todas las operaciones lógicas, ya que el resto pueden ser creadas mediante la combinación de éstas.

Operaciones aritméticas

Aunque el dominio del problema al que se circunscribe MIOOP es el de los sistemas de eventos discretos [Thi01] en los cuales la mayoría de las variables que se manejan son digitales de tipo *"todo/nada"*, es necesario que el módulo editor proporcione los mecanismos necesarios que permitan al ingeniero especificar las operaciones aritméticas básicas *"suma"*, *"resta"*, *"división"* y *"multiplicación"*. Para ello, SimPLC++ soporta los operandos de asignación *"="*, multiplicación *"*"*, división *"/"*, suma *"+"*, resta *"-"*, etc, así como funciones incorporadas en la biblioteca de funciones estándar que permiten las mismas operaciones.

Operaciones de conteo

Es una de las situaciones típicas que el experto en el proceso va a querer representar. Consiste en la posibilidad de llevar la cuenta del número de veces que se produce un evento determinado. El elemento que se emplea para llevar a cabo esta operación se denomina contador y ofrece al experto varias funcionalidades básicas a partir de los FBs estándares incorporados en la biblioteca de utilidad de SimPLC++, a saber:

- Incremento de la cuenta.
- Decremento de la cuenta.
- Puesta a cero o reseteo de la cuenta.
- Asignación del valor máximo de la cuenta.
- Consulta del valor actual de la cuenta.
- Consulta de fin de cuenta alcanzada.
- Comparación de valor de cuenta.

Operaciones de temporizaciones

En no pocas ocasiones, el experto de automatización deseará llevar a cabo la cuenta de una determinada cantidad de tiempo. A esta operación se la denomina temporización, y al elemento que permite llevarla a cabo temporizador. Este tipo de operador viene incorporado en el módulo editor como un conjunto de FBs estándar en la biblioteca de SimPLC++. El temporizador brinda al experto las siguientes funcionalidades:

- Puesta en marcha de la temporización.
- Parada de la temporización.
- Puesta a cero o reseteo de la temporización.
- Asignación de la cantidad de tiempo a temporizar.
- Consulta del valor actual de la temporización.
- Consulta de fin de temporización.
- Comparación de valor actual de la temporización.

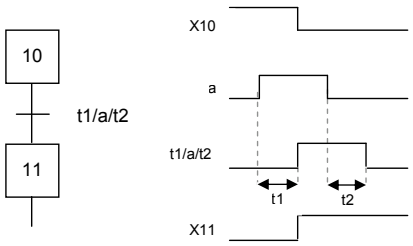

Figura 4.15: Dependencia temporal

4.2.6. Condiciones de transición

La norma internacional IEC 848 [IEC88] define la posibilidad de establecer la condición de transición entre dos estados no sólo como el valor lógico *"1"* de la condición de transición, sino también como:

1. Dependencia Temporal. Representada por *"t1/a/t2"*. Dada la condición de transición *"a"* (*"a"* es una variable o una combinación lógica de variables), será verdadera (valor lógico *"1"*) después del retardo t1 contado a partir del flanco ascendente en *"a"*. La condición de transición permanecerá con valor lógico *"1"* hasta un tiempo t2 posterior al flanco descendente en *"a"*, como se muestra en la figura 4.15.

2. Flancos. La condición de transición de una expresión lógica se hace cierta en función de la evolución dinámica del valor lógico de la condición. Existen dos tipos:

   a) Flanco Ascendente.

   b) Flanco Descendente.

3. Negado. La condición de transición se hace cierta (valor lógico *"1"*) cuando el valor lógico de su expresión lógica asociada es *"0"*.

Cuanlificadores

La norma internacional IEC 848 [IEC88] define un conjunto cualificadores o formas diferentes de ejecutar una acción asociada a una etapa de un GRAFCET. Estos modificadores son ampliamente empleados por el ingeniero del proceso en

sus descripciones textuales sobre cómo el mismo se debe comportar, por lo que el módulo editor proporciona los mecanismos para recogerlos. Estos modificadores son:

- S/R ≡ Stored ≡ Memorizado. Si al invocarse una acción, ésta necesita continuar ejecutándose incluso después de que la etapa haya dejado de estar activa, se dice que la operación es memorizada. Equivale a la acción Set o Reset de un biestable.
- L ≡ Limited ≡ Limitado en el Tiempo. La acción u operación que se lleva a cabo en una etapa se ejecuta durante un período de tiempo limitado mientras el mismo permanezca activo. Una vez transcurrido el tiempo marcado y si la etapa aún permanece activa, la acción dejará de ser ejecutada. Si la etapa deja de ser activa antes de que haya transcurrido el tiempo establecido, la acción dejará de ejecutarse.
- D ≡ Delayed ≡ Retardada. La acción u operación que se lleva a cabo asociada a una etapa no comienza a ejecutarse inmediatamente cuando la etapa es activada sino un instante de tiempo definido después. Si la etapa deja de ser activa antes de que haya transcurrido el tiempo indicado, nunca se llegará a ejecutar la acción asociada a la etapa.
- P ≡ Pulse ≡ De pulso. La acción u operación que se lleva a cabo asociada a una etapa se ejecuta una única vez con independencia del tiempo que la etapa esté activa.
- N ≡ Non Stored ≡ Condicionada. La acción u operación que se lleva a cabo asociada a una etapa se ejecuta mientras la etapa esté activa.

4.3. Módulo TCLOCK

Uno de los objetivos principales de esta memoria de tesis es el de aportar las modificaciones y ampliciones necesarias para permitir adaptar los lenguajes del estandar IEC 61131 a la POO. Por otro lado, un segundo objetivo de este trabajo es el de conocer de forma empírica la eficiencia que produce el paradigma OO en la Ingeniería de Control si se compara con el paradigma de programación estructurado. Esta medida de eficacia se puede conseguir comparando ambos paradigmas desde

un punto de vista numérico. Es decir, tomando medidas estadísticas del número de líneas de código escritas en un programa de automatización y los tiempos de ejecución y comparándolos.

De esta necesidad de medir tanto las líneas de código como los tiempos de ejecución surge como respuesta el módulo TCLOCK de SimPLC++.

El módulo TCLOCK sólo empieza a funcionar si se ejecuta el programa de control mediante el módulo de simulación de SimPLC++. Consiste en un proceso que funciona por debajo del programa de control tomando medidas de tiempos de ejecución. Cada invocación que se hace al módulo TCLOCK desde el programa de control creará un hilo de medición llamado *"THREAD-TIME"* que almacena información acerca de quien ha invocado el hilo (tipo de POU, punto del código donde ha sido invocado, etc) y comienza un contador del tiempo de ejecución. Cada *"THREAD-TIME"* es independientemente de los otros hilos arrancados, tal y como se ve en la figura 4.16.

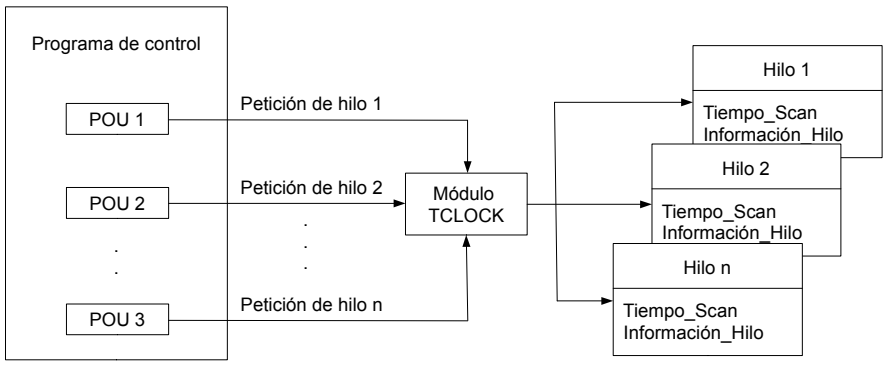

Figura 4.16: Diagrama de petición de hilos *"THREAD-TIME"*

El ciclo de vida de cada hilo se cirscunscribe al ámbito del tipo de TCLOCK que se declara. De esta forma, un hilo se crea cuando se invoca un TCLOCK y el hilo se cierra cuando se introduce la orden de finalización del hilo o el POU al que pertenece. Cuando un hilo termina, éste vuelca sus datos a una estructura de datos de tipo *"lista enlazada"*. Esta estructura está gestionada por el propio módulo TCLOCK y es la encargada de almacenar la información de cada hilo, es decir, cada nodo de la estructura contiene los datos que almacena cada hilo. Al finalizar la ejecución del programa de control con el módulo de simulación, el módulo TCLOK vuelca la *"lista enlazada"* donde ha almacenado los hilos a un

fichero de texto (ver figura 4.17) para su posterior análisis.

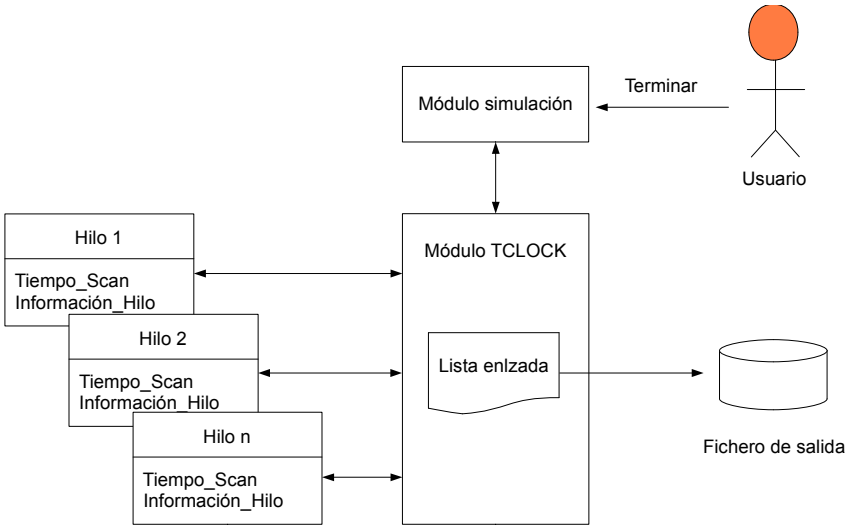

Figura 4.17: Diagrama de volcado de la lista de hilos a fichero

El módulo TCLOCK produce una distorsión en el tiempo total de ciclo de SCAN desde tres puntos de vista.

- Añade dos instrucciones simples al código que deben ejecutarse. Una instrucción para arrancar el hilo y otra para pararlo, lo que implica una complejidad temporal de $O(2)$.

- Al parar un hilo, el módulo TCLOCK inserta en la *"lista enlazada"* un nuevo registro con los datos del hilo. Para minimizar los tiempos de inserción en la estructura de datos, el módulo TCLOCK implementa una *"cola"* para que las inserciones tengan una complejidad temporal de $O(1)$.

- Al terminar la simulación de la ejecución de un programa, el módulo TCLOCK vuelca todos los datos de la *"cola"* que almacena los hilos *"THREAD-TIME"* a un fichero de texto. Dicha proceso, al hacerse fuera del ciclo de SCAN del programa de control, no introduce ningún tipo de alteración en el mismo.

4.3.1. Tipos de TCLOCKs

El módulo TCLOCK incorpora tres tipos de medidores de tiempos de ejecución. Uno a nivel de código, otro a nivel de POU y otro a nivel de método. Todos ellos pueden ser invocados desde cualquiera de los lenguajes del módulo de edición y mezclados entre sí, es decir, es posible utilizar un tipo de TCLOCK a nivel de POU y dentro utilizar varios a nivel de código.

TCLOCK a nivel de código

Este tipo de TCLOCK sirve para analizar el tiempo de ejecución de un trozo de código en particular. Entre la información que almacena el módulo TCLOCK en este caso se encuentra el tiempo de ejecución y desde cuál de los POUs soportados por SimPLC++ fue lanzado el hilo. En el caso de los lenguajes textuales, además se incluye el número de línea de código donde se comenzó el temporizador del TCLOCK y el número de línea de código donde se paró el temporizador. En el caso de los lenguajes gráficos LD y FBD, se incluye el nombre de la red donde comenzó el hilo del TCLOCK y el nombre de la red donde se paró el mismo. En el caso de SFC, se incluye el nombre de la etapa donde comenzó el hilo y el nombre de la etapa donde se paró.

Este tipo de TCLOCK requiere que el programador escriba explícitamente dentro del código de control las instrucciones de arranque y parada del TCLOCK. Si por error u omisión no se introdujese la instrucción de parada, el módulo traductor insertaría dicha instrucción como la última línea de código del POU traducido.

TCLOCK a nivel de POU

Este tipo de TCLOCK sirve para analizar el tiempo de ejecución de todo un POU incluyendo en esta definición a los programas, FBs, funciones y clases. Entre la información que almacena el módulo TCLOCK en este caso se encuentra el tiempo de ejecución y desde qué POU fue lanzado el hilo.

SimPLC++ permite al usuario declarar un TCLOCK a nivel de POU cuando se crea un POU (del tipo que sea) por medio de las herramientas de edición del módulo editor. De esta manera, el programador se olvida de tener que insertar de forma explícita las llamadas con código para arrancar y parar el TCLOCK. Es el

propio módulo traductor el encargado de insertar dichas llamadas. El arranque del hilo *"THREAD-TIME"* sería en la primera instrucción que ejecutaría el POU y la parada del hilo sería la última instrucción de dicho POU.

En el caso del TCLOCK a nivel de clase, el módulo TCLOCK considera a cada método de la clase como un POU inpendiente, insertando en cada método las instrucciones de arranque y parada de los hilos. Este tipo de TCLOCK a nivel de clase permite ingeniero de automatización olvidarse de colocar un TCLOCK en cada método.

TCLOCK a nivel de método

Este tipo de TCLOCK tiene un funcionamiento parecido al de a nivel de tipo POU con la diferencia de que sólo se lanza cuando se invoca el método al que se asocia este tipo de TCLOCK. La información que almacena el módulo TCLOCK sería el tiempo de ejecución, desde qué objeto se ha lanzado la invocación del método y desde qué método fue lanzado el hilo.

Como en el caso del TCLOCK a nivel de POU, en el momento de la creación de un método se indica que se quiere incorporar un TCLOCK asociado a éste. Posteriormente, el módulo traductor insertará como primera instrucción del método el arranque del módulo TCLOCK y como última línea del método la parada del TCLOCK.

4.4. Módulo traductor

Los equipos electrónicos de control basados en el computador se fundamentan en la ejecución de instrucciones codificadas en lenguaje máquina. La programación de sistemas de control directamente en código máquina resulta un esfuerzo enorme que puede aliviarse por la utilización de lenguajes de programación de alto nivel. El problema de estos lenguajes de alto nivel es que no son comprensibles directamente por el computador. Este problema se soluciona mediante la compilación. Un compilador es un programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación generando un programa equivalente que la máquina es capaz de interpretar. En la figura 4.18 se muestra un diagrama de bloques del proceso de compilación dividido en tres fases.

En la primera se realiza una traducción a un código intermedio. Si al traducir este código se encuentran errores, se interrumpe la compilación. La segunda fase optimiza éste código intermedio para que el código resultante de la tercera fase, el código máquina que ejecuta el computador, resulte más rápido.

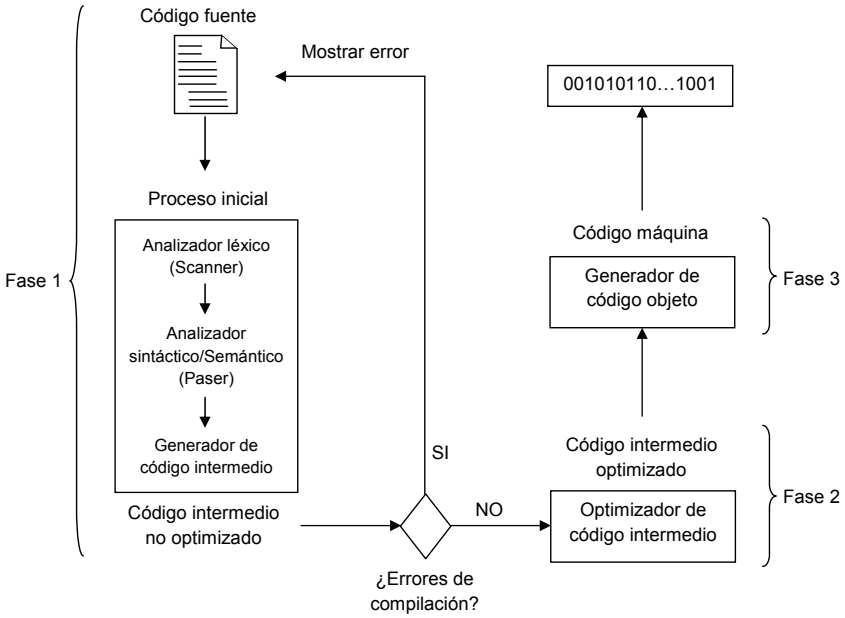

Figura 4.18: Diagrama de bloques de una compilación

El tipo de equipo de control más impuesto en la industria en este momento, tal y como se ha descrito en capítulos anteriores, es el PLC. Por lo general, estos autómatas se programan empleando los lenguajes de programación proporcionados por el fabricante y cada vez más, los lenguajes definidos por la norma IEC 61131. En cualquiera de estos dos casos, los fabricantes proporcionan los editores necesarios para implementar tanto el código de control en un lenguaje de alto nivel y los compiladores que traducen dicho lenguaje a otro que el PLC pueda interpretar y ejecutar.

Debido a que el lenguaje máquina de cada PLC es específico de cada fabricante, SimPLC++ realiza una compilación de los programas de control basados tanto en la norma IEC 61131 y su ampliación OO propuesta por MIOOP para las fases 1 y 2 de la figura 4.18. De esta manera, el módulo traductor se encarga de generar

un código intermedio estándar para todos los PLCs basados en la norma IEC 61131 independientemente del hardware interno que éstos tengan, a partir de un programa de control implementado en cualquiera de los lenguajes de la norma IEC 61131 (incluyendo sus versiones OO).

4.4.1. Lenguaje intermedio del módulo traductor

Para poder introducir un programa de control desarrollado con SimPLC++ en un PLC real se necesitaría un driver o las especificaciones técnicas del PLC del fabricante, o exportar el código intermedio generado por el módulo traductor al editor del fabricante para compilar dicho código desde ese editor. Por lo tanto, este código intermedio generado por el módulo traductor debe ser estándar y reconocible por cualquier editor de programas de control basados en la norma IEC 61131, independientemente del fabricante del mismo.

Por un lado, ninguno de los lenguajes de programación de PLCs, ni los definidos en la norma, ni los de los fabricantes son orientados a objetos, por lo que un programa de control desarrollado en SimPLC++ con componentes OO, no podría ser introducido directamente en un PLC real o exportado como tal a un editor de un fabricante para su compilación a código máquina. Sería necesaria una transformación previa de ese programa definido en OO a un programa equivalente codificado en los lenguajes del estándar.

Por otro lado, todos los lenguajes recogidos en la norma IEC 61131 pueden ser traducidos a IL, que es el lenguaje de más bajo nivel de la norma y puede ser considerado como el lenguaje ensamblador de la misma.

De esta manera, el módulo traductor convierte la implementación desarrollada con el módulo editor a lenguaje IL estándar de la norma. Esta traducción se realiza en una serie de fases bien diferenciadas:

1. Evaluación léxica, sintáctica y semántica del código del módulo editor.

2. Traducción de las estructuras y algoritmos OO a su versión no orientado a objetos siguiendo los principios marcados por MIOOP.

3. Traducción del código no orientado a objetos a IL.

Evaluación de los programas de control

La evaluación del programa de control tiene como fin encontrar errores en la implementación de todos los POUs del proyecto, tanto a nivel léxico, como sintáctico y semántico. Cualquier tipo de error abortará el proceso y mostrará el error encontrado para que el programador lo subsane.

Traducción de código OO a no orientado a objetos

Cuando el módulo traductor detecta cualquier tipo de implementación OO, traduce dicho código a su versión no orientada a objetos siguiendo las premisas impuestas por MIOOP. A partir de este momento, cuando éste documento se refiera a las versiones OO de los lenguajes de la norma, se hará por medio del nombre de cada lenguaje seguido del símbolo *"++"*, es decir, IL++, ST++, LD++, FBD++ y SFC++.

Debido a que el módulo editor permite la implementación de código OO desde cualquiera de los 5 lenguajes recogidos en la norma IEC 61131, el módulo traductor es capaz de traducir estos 5 lenguajes a sus versiones no OO tal y como se observa en la figura4.19.

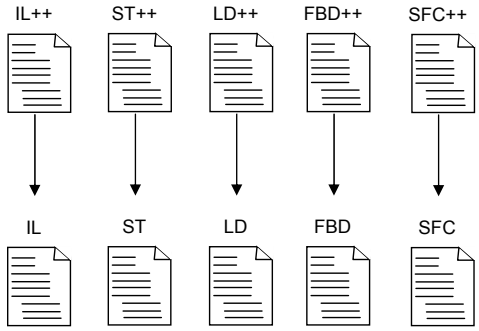

Figura 4.19: Traducción de lenguajes OO a sus versiones no OO

Traducción de lenguajes no OO a IL

Una vez que el módulo traductor tienen traducido el programa de control a

una versión no OO, el siguiente paso consiste en traducir el código a lenguaje IL de la norma IEC 61131.

Este código IL generado sería el resultado final de la compilación. SimPLC++ exporta automáticamente este código a un fichero de texto para que el ingeniero de control pueda observarlo o retocarlo si así lo deseara. Así mismo, SimPLC++ permite la importación de este tipo de ficheros para realizar un proceso de compilación inversa para la obtención de código IL++ (ver figura 4.20).

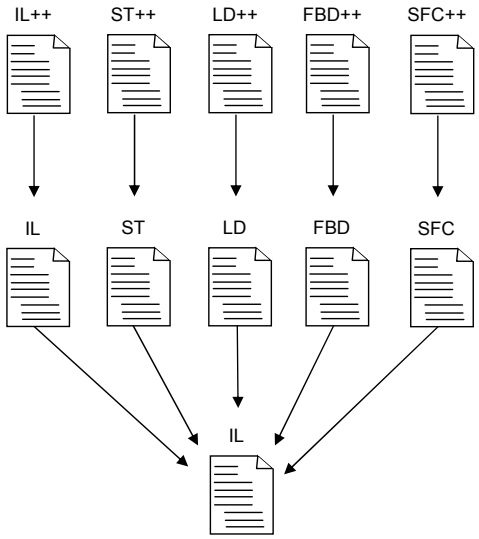

Figura 4.20: Traducción final de un lenguaje OO a IL

4.4.2. Reglas de traducción

El módulo traductor transforma todos los lenguajes que soporta SimPLC++ (a saber, los 5 lenguajes de la norma IEC 61131 y sus versiones OO) al lenguaje de más bajo nivel en el estándar IEC 61131, el lenguaje IL. Para ello, el módulo traductor aplica una serie de patrones dependiendo de qué tipo de lenguaje esté traduciendo.

Lenguaje estructurado (ST)

- Cada variable es traducida a una variable con el mismo nombre a IL.
- Las llamadas a funciones, FBs y métodos de clases se realizan por medio de la palabra reservada *"CAL"*.
- Las estructuras *"IF-THEN-ELSE"* se traducen como un salto condicional al trozo del código *"THEN"* si se cumple la condición del *"IF"* o al trozo de código *"ELSE"* en caso contrario.
- La estructura *"WHILE"* y *"DO-UNTIL"* se traducen como un salto condicional. La diferencia entre las dos instrucciones radica en el lugar de la comparación del salto. En el caso del *"WHILE"* es en la primera instrucción del bucle y en el caso del *"DO-UNTIL"* en la última instrucción.

Lenguaje de contactos (LD) y bloques funcionales (FBD)

El módulo traductor asocia un grupo de instrucciones diferentes a cada bloque gráfico de estos dos lenguajes, dependiendo de la naturaleza del bloque:

- Si el bloque es una variable se traduce con el mismo nombre a IL.
- Si es una función, FB o método de una clase se realiza una llamada a dicho POU por medio de la palabra reservada *"CAL"*.

El sentido de traducción de las redes es de arriba hacia abajo, de tal modo que la primera red que el módulo traductor encuentra implementada en un POU será el primer conjunto de instrucciones codificadas en IL.

En lenguaje LD, en una misma red, la asociación en la misma línea de una misma red de diversos bloques se traduce como la secuencia de las instrucciones asociadas a dichos bloques unidos por instrucciones *"AND"*. La asociación de bloques en líneas paralelas se traduce como la secuencia de las instrucciones asociadas a dichos bloques unidos por instrucciones *"OR"*.

Diagrama funcional secuencial (SFC)

El módulo traductor aplica las mismas reglas de traducción en el lenguaje SFC que en los otros dos lenguajes gráficos de la norma para los bloques gráficos. En

cambio, no se aplican los mismos patrones respecto al sentido de traducción de un GRAFCET que en los lenguajes LD o FBD. En SFC no es suficiente con una traducción de arriba hacia abajo del grafo dirigido, además es necesario conocer el estado de cada etapa y qué transiciones son franqueables en cada momento.

Para traducir el flujo de ejecución de un GRAFCET se utiliza un método sistemático dividido en tres fases:

1. Evaluación de transiciones. A cada transición del GRAFCET se le asocia una variable booleana "TR_i". Esta variable está asociada a la transición que separa la etapa X_i de la $Xi+1$. La ecuación:
   $TRi{:=}X_i * <$ *condición de la transición* $i >$
   refleja si una transición es franqueable o no.

2. Secuenciación de etapas. Para cada etapa se establecen dos ecuaciones que indiquen las condiciones de activación y desactivación de dicha etapa. Las ecuaciones tienen la forma:
   $S(X_i) := TR_i - 1$ para un *"SET"*.
   $R(X_i) := TR_i$ para un *"RESET"*.

3. Ejecución de acciones. Con el conjunto de ecuaciones de los apartados anteriores es suficiente para implementar un secuenciador, es decir, un dispositivo capaz de seguir una secuencia dada, pero el módulo editor de SimPLC++ permite asociar a cada etapa un conjunto de acciones. Esta asociación también se puede representar por una fórmula matemática del tipo:
   $<$ *acción* $>:= X_i$

Para comprender el método de traducción de un GRAFCET se propone un ejemplo que ilustre dicha conversión por medio de las 3 fases citadas:

Dado el GRAFCET de la figura 4.21 que consta de 2 etapas separadas por una transición (la condición de franqueabilidad sería por ejemplo "$(a+b)*c$" y una acción asociada a la etapa 1 que acciona el contactor *"KM1"*).

Al aplicar el método se obtienen los siguientes resultados por cada fase:

1. La transición que separa las etapas 0 y 1 tiene la condición: $(a+b)*c$. A esta transición se le asocia la variable TR0, y a las etapas se les asigna respectivamente las variables X0 y X1. La ecuación que reflejaría la situación

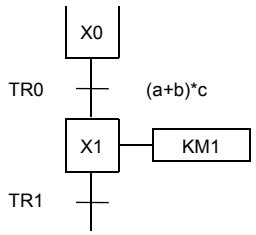

Figura 4.21: Ejemplo de traducción de un GRAFCET

de franqueabilidad de la citada transición quedaría como: $TR0 := X0 * ((a + b) * c)$

2. El *"SET"* de la etapa 1 se producirá si y sólo si la transición que separa la etapa 0 y 1 es franqueable, por tanto: $S(X1) := TR0$. El *"RESET"* de la etapa 1 se producirá si y sólo si la transición que separa la etapa 1 de la etapa 2 (a la que por ejemplo se llama TR2) es franqueable, por tanto: $R(X1) := TR1$

3. La ejecución de las acciones asociadas a una etapa se produce solo cuando la etapa esta activa. De esta forma, la ecuación que activa el contactor sería: $KM1 := X1$

4.4.3. Estructura lógica del parser de traducción

Como se ha indicado anteriormente, el módulo traductor permite convertir cualquier lenguaje soportado por SimPLC++ a lenguaje IL. Para poder realizar esta traducción, el módulo traductor necesita definir una tabla de símbolos, un generador de código y un analizador semántico, sintáctico y léxico para cada uno de los lenguajes soportados por SimPLC++.

Los analizadores léxicos y sintácticos de los lenguajes que proporciona SimPLC++ no tienen ninguna relación, pero no sucede lo mismo con respecto a la tabla de símbolos, ya que en todo momento se pueden usar todos los tipos de datos en todos lenguajes. El analizador semántico también puede ser igualmente compartido ya que todos los lenguajes están sujetos a las mismas restricciones semánticas. Por último, el generador de código va a generar código en IL según la norma IEC 61131, por tanto su funcionalidad es similar para todos los lenguajes.

Debido a las similitudes con respecto a cada uno de los elementos comentados anteriormente, el módulo editor encapsula toda la funcionalidad de la tabla de símbolos, del analizador semántico y del generador de código en tres librerías que son accesibles por los traductores de todos los lenguajes. Por otro lado, cada librería de traducción léxica y sintáctica es común para cada lenguaje estructurado y OO. Por tanto, las versiones estructurada y OO comparten analizador léxico y sintáctico.

De esta forma, el módulo editor incorpora 8 sub módulos, 5 dedicados a los analizadores léxicos/sintácticos y los otros 3 para la tabla de simbolos, analizador semántico y el generador de código (ver figura 4.22).

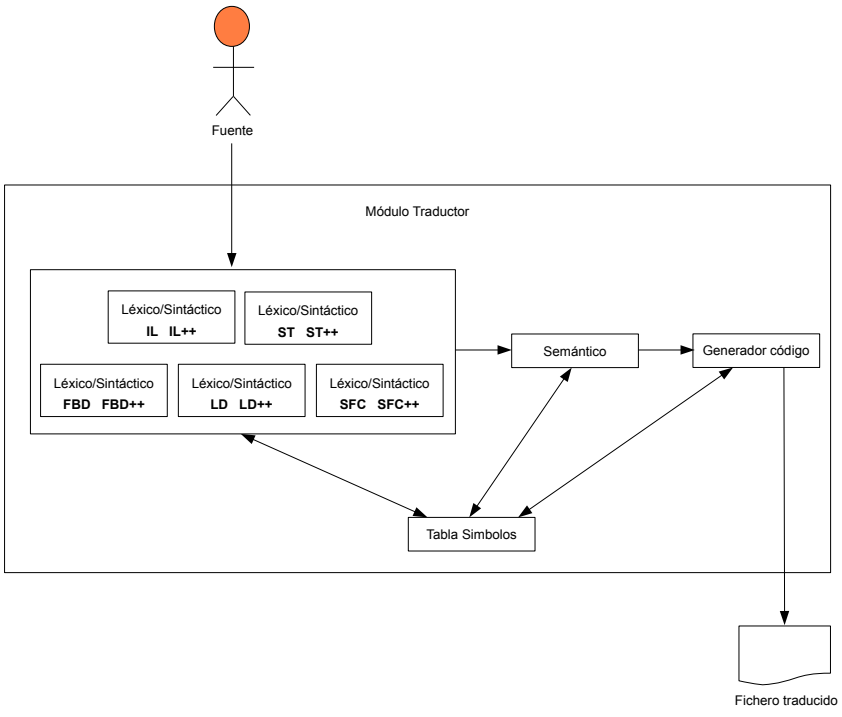

Figura 4.22: Estructura de las librerías del traductor

4.5. Módulo simulador

Una vez que el usuario implementa el programa de control y se compila comprobando que éste es correcto, existe la posibilidad de ejecutarlo para verificar que éste cumple con las especificaciones funcionales requeridas o deseadas.

Existen dos posibilidades a la hora de realizar la ejecución del programa de control:

- Realizar una ejecución del programa de control sin que existan más entidades que el propio SimPLC++.
- Ejecutar el programa de control en conexión con otras entidades remotas usando el laboratorio de automatización virtual (LAV) a través del software MEGADRIVER (ver anexo B).

En el primer caso, SimPLC++ proporciona un visor de la evolución de los estados de las variables a lo largo del ciclo de SCAN.

En el segundo caso, es necesario publicar en MEGADRIVE las variables de salida y de entrada casándolas con las etiquetas del otro cliente con el que se comunica SimPLC++.

4.5.1. Ejecución desde el módulo de simulación

Para poder llevar a cabo una ejecución del programa de control con SimPLC++, el módulo simulador proporciona un servicio de ejecución llamado *"SM-Execute"* que lleva a cabo las reglas de ejecución de las tareas especificadas mediante el módulo de edición. Para cumplir con este cometido, el *"SM-Execute"* se divide en dos componentes, a saber:

- Planificador de tareas (SM-Schedule). Se encarga de determinar qué tarea debe ser ejecutada en cada momento. Este planificador se implementa siguiendo los estudios de Ohman [JA98].
- Intérprete de instrucciones (SM-Player). Una vez que el planificador de tareas determina qué tarea se debe ejecutar en un momento dado, se procede a interpretar de manera secuencial las instrucciones de la instancia de cada POU asociado a la misma comenzando por la primera de ellas hasta llegar

a la última.
Una instrucción por lo general presenta la siguiente sintaxis:
<operando><operando> [;< operando >].
Interpretar una instrucción significa descifrar qué acción indica el operador que se debe llevar a cabo sobre el operando u operandos que la acompañan y ejecutarla. Los operandos son típicamente variables del programa que ocupan una posición en la memoria.

El servicio de ejecución permanece indefinidamente planificando qué tareas se deben ejecutar e interpretando sus instrucciones mientras el usuario no detenga la simulación.

Por otro lado, el *"SM-Execute"* necesita acceder al juego de instrucciones de los POUs asociados al *"SM-Schedule"* y poder acceder a las variables instanciadas. De esta forma, el módulo simulador proporciona una máquina virtual llamada *"SM-VirtualMachine"* que se encarga de gestionar y cargar en memoria el código de los POUs que están asociados a tareas y de gestionar un bloque de memoria reservado para la simulación. Es decir, el *"SM-VirtualMachine"* implementa un segmento de código y un segmento de datos dentro del módulo simulador.

Internamente, el *"SM-VirtualMachine"* se divide en dos componentes o segmentos, a saber:

- Segmento de código (SM-Code). En el *"SM-Code"* se cargan todas y cada unas de las instrucciones de los POUs asociados a las tareas.
  Debido a que en la ejecución de un conjunto de instrucciones pueden producirse saltos hacia adelante o hacía atras y que el acceso a cada instrucción debe tener una complejidad temporal de $O(1)$, la estructura de datos usada es un ArrayList (Code-Array) en el que cada nodo es una instrucción a ejecutar por el *"SM-Player"*.
  Cuando el usuario ejecuta el módulo simulador, el *"SM-Schedule"* inicializa un *"SM-Code"* por cada tarea y en tiempo de ejecución, dependiendo de la prioridad de cada una de ellas, indica al *"SM-Player"* a qué *"SM-Code"* debe acceder.

- Segmento de datos (SM-Data). Es necesario que el módulo simulador almacene internamente el valor de las variables y gestione una pila por cada *"SM-Code"* que el *"SM-Player"* maneja. Esta tarea recae en el *"SM-Data"*.

Por otro lado, el "*SM-Data*" almacena internamente el valor de *"PC"* (contador de programa) del *"SM-Code"* para poder saber qué instrucción es la siguiente a ser ejecutada por el *"SM-Player"*.
La estrutura de datos que almacena el valor de las variables es una tabla Hash (Data-Hash). Y tanto la pila, como la tabla hash, como *"PC"* se asocian a un *"SM-Code"* y son creados por el *"SM-Schedule"* en el momento que comienza la simulación.

La arquitectura interna del módulo simulador se puede observar en las figuras 4.23 y 4.24.

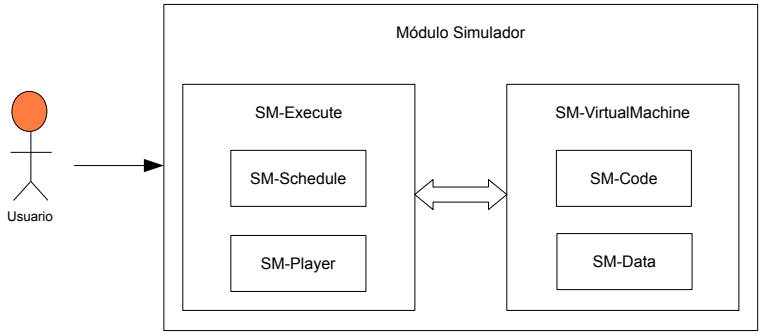

Figura 4.23: Arquitectura del módulo simulador

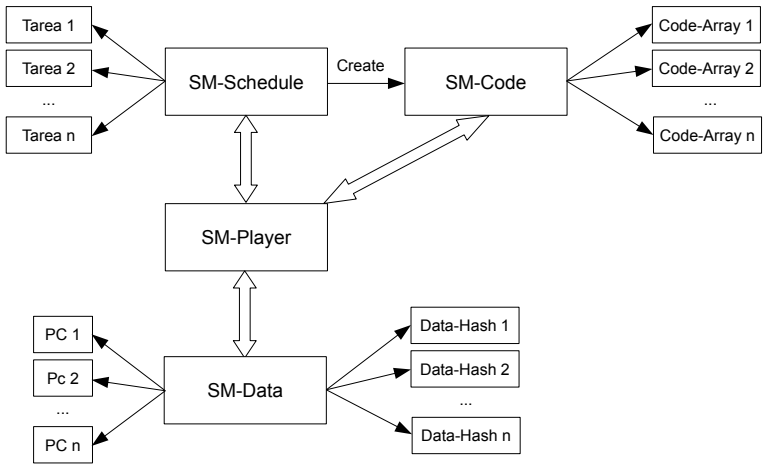

Figura 4.24: Esquema de ejecución del módulo de simulación

4.5.2. Consideraciones finales del módulo simulador

El módulo simulador ejecuta en un bucle infinito las tareas asociadas al *"SM-Schedule"*, de esta manera, cuando se termian de ejecutar todas las instrucciones del ciclo de SCAN, el planificador comienza de nuevo con la primera tarea. Para evitar que por errores de programación o que el lazo de ejecución entre en un punto del código del que no puede salir, el *"SM-Schedule"* implementa un perro guardian (Watchdog) que vigila que el tiempo de ejecución de un programa no exceda un determinado tiempo máximo (tiempo de ciclo máximo).

Debido a que el módulo simulador tiene como fin ver el correcto funcionamiento del programa de control, el usuario puede parar en cualquier momento la simulación e interactuar con el valor de las variables almacenadas en el *"SM-Data"* para provocar diferentes tipos de situaciones y poder observar cómo el programa de control evoluciona a partir de este momento. Cuando el usuario pausa la ejecución del módulo simulador, éste envía una orden al módulo TClock para que detenga a su vez todos los hilos de tiempos. De esta forma se evita que una parada no prevista del sistema por parte del usuario produzca mediciones incorrectas por el módulo TClock.

4.6. Módulo SP Linker

SimPLC++ proporciona a los usuarios la posibilidad de transferir un programa de control (tanto en una versión estructurada como OO) a un PLC real. Para ello, el módulo SP Linker permite generar un archivo XML que usa los servicios proporcionados por el Unity Developer's Edition (UDE) [1] para enviar el programa de control a un PLC de la marca Schneider.

La secuencia de acciones que realiza el módulo SP Linker es la siguiente:

1. Carga en memoria el programa de control generado por el módulo traductor (ver sección 4.4) en lenguaje IL de la norma IEC 61131.

2. Genera un archivo XML traduciendo cada instrucción a un conunto de marcas.

[1] En el anexo C se realiza un breve resumen sobre el framwork UDE

3. Asocia al archivo XML generado la configuración del PLC donde va a ser traspasado y la IP del mismo.

Los dos primeros pasos se realizan de forma automática una vez que el usuario solicita la ejecución del módulo SP Linker. Una vez que se lanza el módulo, se solicita al usuario que introduzca la configuración hardware del PLC y la IP que tiene éste dentro de la red. Estos datos son necesarios para que el UDE pueda transferir el código compilado por los servicios del UDE.

La IP debe ser introducida a mano por parte del usuario, pero la configuración de los PLCs se carga de una base de datos que contiene un gran número de modelos de PLCs de la marca Scheneider así como distintos módulos de comunicación, etc. De esta forma, por medio de un sencillo interfaz, el usuario puede seleccionar el PLC exacto al que va a transferir y si en el futuro aparecen nuevos modelos, la BD puede ser editada.

La secuencia de acciones que realiza el SP Linker se puede observar en la figura 4.25.

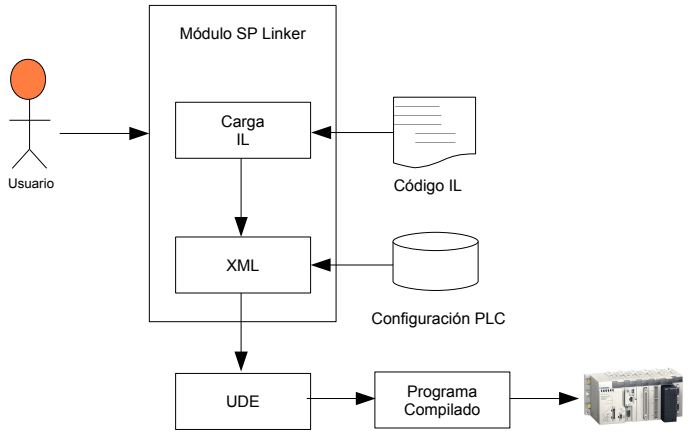

Figura 4.25: Esquema de funcionamiento del módulo SP Linker

4.7. Conclusiones

A lo largo de este capítulo se han presentado las características arquitectónicas de los módulos que componen la herramienta software SimPLC++ que permite no

sólo una programación basada en la norma IEC 61131, sino que además posibilita la programación de sistemas de control OO mediante la aplicación sistemática de MIOOP. Además de los módulos de edición y ejecución de programas de control basados en IEC 61131 y MIOOP, SimPLC++ proporciona un módulo que permite verificar la viabilidad de la presente memoria de tesis de una forma empírica. Para ello, permite tomar medidas de los ciclos de SCAN de los programas de control que se desarrollan a medida que estos se ejecutan por medio del módulo simulador que porporciona SimPLC++.

Se puede concluir diciendo que SimPLC++ no sólo proporciona un excelente banco de pruebas para validar la presente memoria de tesis, además proporciona una herramienta software que se integra dentro del LAV del grupo GENIA, lo que posibilita disponer de un conjunto de herramientas totalmente abiertas, escalables e independiente de cualquier fabricante de PLCs que el futuro implementador podrá seguir o no dependiendo de sus gustos y conocimientos.

Capítulo 5

Resultados experimentales

Sin ninguna confusión de espíritu, sin relajarse en
ningún momento, puliendo la mente y la atención,
afilando el ojo que observa y el ojo que ve, uno llega
al vacio real como el estado en el que no hay oscuridad
y las nubes de la confusión han desaparecido.
Miyamoto Musashi (Anillo del vacío)

5.1. Introducción

En este capítulo se presenta un ejemplo de aplicación de MIOOP programado con la herramienta SimPLC++. Mediante este ejemplo se pretende demostrar la viabilidad de programar un sistema de control por medio de un programa implementado bajo un paradigma orientado a objetos basado en las ampliaciones realizadas a la norma IEC 61131 relatadas en el capítulo 3. Así mismo, se dispondrá de una versión del ejemplo siguiendo el estandar clásico de IEC 61131 de programación estructurada (programada con SimPLC++) para que sirva como punto de comparación entre ambos paradigmas. Tambíen se detallan las distintas

mediciones tomadas en ambos experimentos por medio del módulo TCLOCK proporcionado por SimPLC++ (ver apartado 4.3). Dichas mediciones servirán para realizar una valoración objetiva y cuantitaviva de los beneficios o perjuicios que el paradigma OO tiene sobre el estructurado, lo que servirá como conclusiones finales de la presente memoria de tesis.

El experimento que se ha programado con SimPLC++ está dividio en una serie de procesos que se describen en los siguientes apartados. Cada apartado, a su vez contiene los siguientes subapartados:

1. Descripción del proceso.

2. Etapas del proceso. Se realiza una descripción de las diferentes fases que conforman el proceso que se está detallando.

3. Diagrama de clases. Esquema de las clases que conforman el proceso de la versión programada siguiendo el paradigma OO.

4. Implementación de los procesos. Se muestran las programaciones del proceso siguiendo los dos paradigmas de programación que se pretenden comparar en este capítulo. El lenguaje de programación que se ha elegido para realizar el experimento es SFC y su versión orientada a objetos ampliada por MIOOP, SFC++.

5. Resultados de tiempos. Las medidas de tiempos tomadas por el módulo TCLOCK se presentan en dos partes. Por un lado, se muestra de forma numéria y en una tabla los tiempos de ejecución de todas las etapas que conforman el proceso. Por otro lado y debido a la dificultad que puede presentar el comparar directamente valores numéricos por parte del lector, se muestran los tiempos de ejecución de cada etapa del proceso por medio de los diagramas de cajas (ver figura 5.1). Estos diagramas de cajas permiten comparar de forma visual los percentiles de tiempos de cada paradigma posibilitando comprobar que etapa es más rápida globalmente en un golpe de vista.
   Además de los diagramas de cajas, en el apartado de tiempos de cada proceso se muestra la media, desviación típica y covarianza de cada una de las etapas que conforman cada proceso para los dos paradigmas de programación.

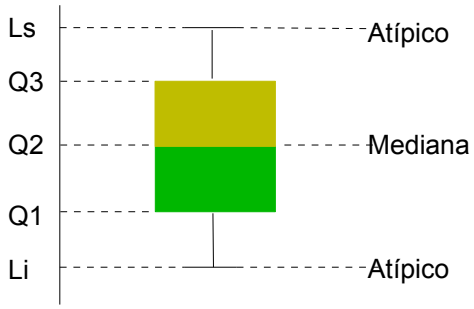

Figura 5.1: Ejemplo de diagrama de cajas

Debido al tamaño del experimento, en este capítulo sólo se hará una breve descripción del mismo, mostrando las partes del código OO y estructurado más relevantes, dejando un detalle más profundo para los anexos (ver anexo D). De igual manera, en los anexos sólo se mostrará el código OO y su traducción a código IL de una de las partes del experimento con el fin de no abrumar al lector con la extensión de la prueba experimental.

5.2. Descripción del proceso

El proceso que se pretende automatizar se corresponde la célula de fabricación exible FMS-200 (ver figura 5.2) del proveedor multinacional con sede en Japón SMC, especialista en el desarrollo de componentes neumáticos.

La célula flexible FMS-200 lleva a cabo el montaje de las diferentes piezas mediante el ensamblado de los componentes mostrados en la figura 5.3, para montar el producto terminado que se muestra en la figura 5.4. Para ello, el sistema completo está constituido por ocho estaciones, realizándose en cada una de ellas el proceso de inserción de un determinado componente.

Las diferentes estaciones están adaptadas para el montaje de una gran diversidad de conjuntos, introduciendo variaciones de material en las piezas, tamaño de éstas, color o altura. La combinación de todas estas posibilidades permite crear un total de veinticuatro posibles combinaciones. Ésto da la posibilidad de realizar varias estrategias de producción.

El proceso de fabricación incluye toda una serie de operaciones de alimentación,

Figura 5.2: Célula flexible FMS-200

manipulación, verificación y carga, realizados mediante componentes de diferentes tecnologías (neumática, hidráulica, electrotecnia, robótica, etc.).

Todas las estaciones se encuentran adosadas alrededor de un sistema de transferencia, que mediante unas cintas transportadoras, realiza el transvase del conjunto de piezas de una estación a otra.

Se enumeran a continuación las ocho estaciones de que consta la célula:

- Estación 1 - Colocación de bases.
- Estación 2 - Inserción de rodamientos.
- Estación 3 - Prensa hidráulica.
- Estación 4 - Inserción de ejes.
- Estación 5 - Inserción de tapas.
- Estación 6 - Inserción de tornillos.
- Estación 7 - Atornillado robotizado.
- Estación 8 - Almacén.

Figura 5.3: Vista de los Componentes a Ensamblar por la FMS 200

Figura 5.4: Muestra del producto final.

Cada una de las estaciones está gobernada por un PLC, y todos estos a su vez son coordinados por un PLC central (el transfer) que actúa como director de orquesta. La topología se muestra en la figura 5.5.

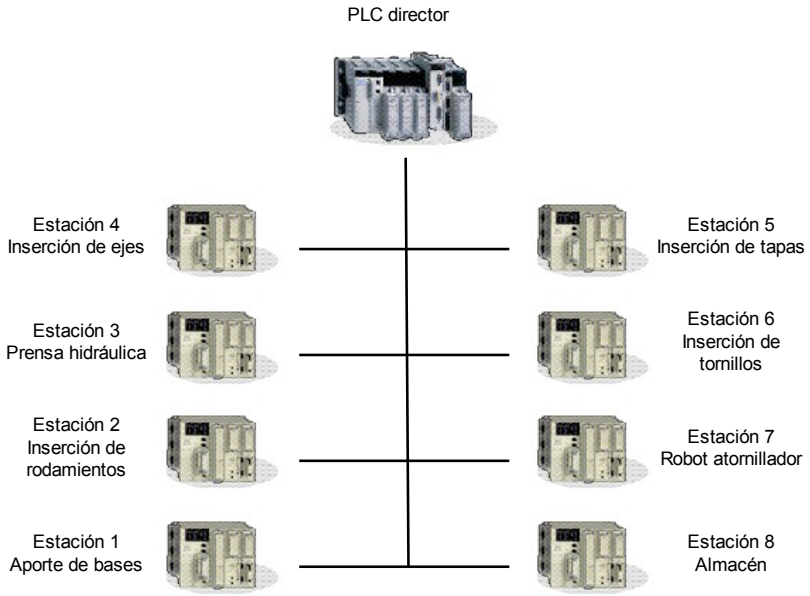

Figura 5.5: Disposición de los PLCs de la célula FMS-200

Sólo son objeto de estudio las estaciones numeradas de la uno a la seis, debido a que todas estas estaciones son controladas por un PLC y presentan un diseño en el cual predominaban los actuadores basados en la tecnología de fluidos. Sin embargo, la estación 7 (estación robotizada de atornillado) hace uso de una nueva tecnología de amplia difusión, como es la robótica, tecnología que no entra dentro del estudio de la presente memoria de tesis. Por otro lado, la estación ocho (estación de almacenaje) no presenta la suficiente complejidad para ser estudiada de forma aislada y su lógica de negocio está incluida dentro del *"transfer"*. Por último, en este capítulo se muestra un trozo de la programación del *"transfer"*, aunque no sus tiempos de ejecución debido a los tiempos de espera de las 8 estaciones. En particular, el proceso que se estudia del *"transfer"* es el encargado de invocar la ejecución de las distintas estaciones (en las dos versiones, estructurada y OO) para que el lector pueda observar la diferencia que existe de programación usando el polimorfismo y compararlo con el estilo estructurado.

5.3. Detalles de implementación

Se disponene de dos versiones programadas de cada estación totalmente funcionales. Ambas han sido programadas con la herramienta SimPLC++ y medidas con el módulo TClock en diferentes puntos. Las clases han sido definidas en lenguaje ST++ y el resto del código, tanto en la versión OO como en la estructurada, han sido programadas con SFC++ y SFC respectivamente.

La versión OO del experimento se compone de un conjunto de objetos sencillos y otros más complejos que se forman a partir de los más sencillos y de otros objetos genéricos de los que heredan. A parte, cada elemento físico de la estación es caracterizado en el modelo orientado a objetos como una clase. Las clases genéricas contienen diversos métodos virtuales así como servicios y atributos que heredan los objetos complejos. En la figura 5.6 se muestra el diagrama de clases de la herencia entre la clase base genérica y las clases complejas que son usadas por las estaciones y que se componen además de objetos simples. Así mismo, cada estación hereda de un interfaz (ver figura 5.7) que posee varios métodos virtuales como el método *"Run"*, *"InicializarElementos"*, *"GetFinProceso"*, etc. Desde el *"transfer"*, la invocación del método *"Run"* de cada estación provoca la ejecución de la misma de forma polimórfica. Del mismo modo, la clase *"ControlProcesoEstacion"* es casi exclusiva de cada estación pero comparte muchos elementos comunes entre las 6 estaciones a estudiar. Por este motivo, las clases *"ControlProcesoEstacion"* heredan de la clase *"ObjetoBaseComplejo"* e implementa y especializa sus métodos virtuales en cada estación (ver figura 5.8).

Cada estación OO posee un método *"Run"* que encapsula toda la lógica interna de cada estación. En las subsecciones en las que se comparan los dos paradigmas se mostrará una parte de ese método *"Run"* de la versión OO, así como los métodos de las clases que están involucradas en los procesos que se quieren comparar. En la versión estructurada sólo se muestra el trozo de las ramas donde se realizan los procesos análogos a la versión orientada a objetos.

La versión estructurada de cada estación está desarrollada en un único FB programado en lenguaje SFC. El *"transfer"* invoca la ejecución del FB que representa cada estación a medida que el sistema va evolucionando.

En las sucesivas secciones se pasa a detallar algunas partes de cada una de las estaciones que conforman el experimento bajo el paradigma OO, comparándolas

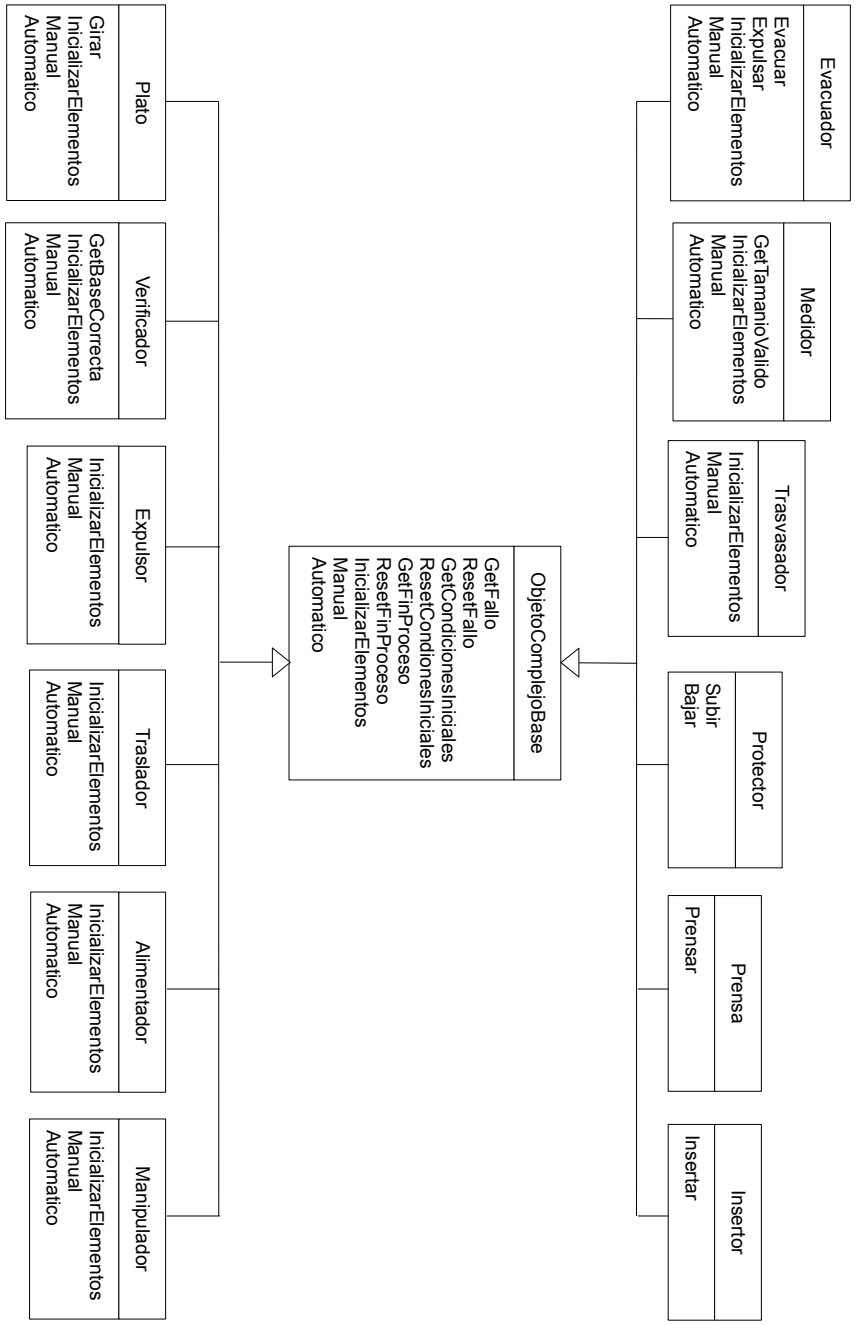

Figura 5.6: Esquema de herencia de clases

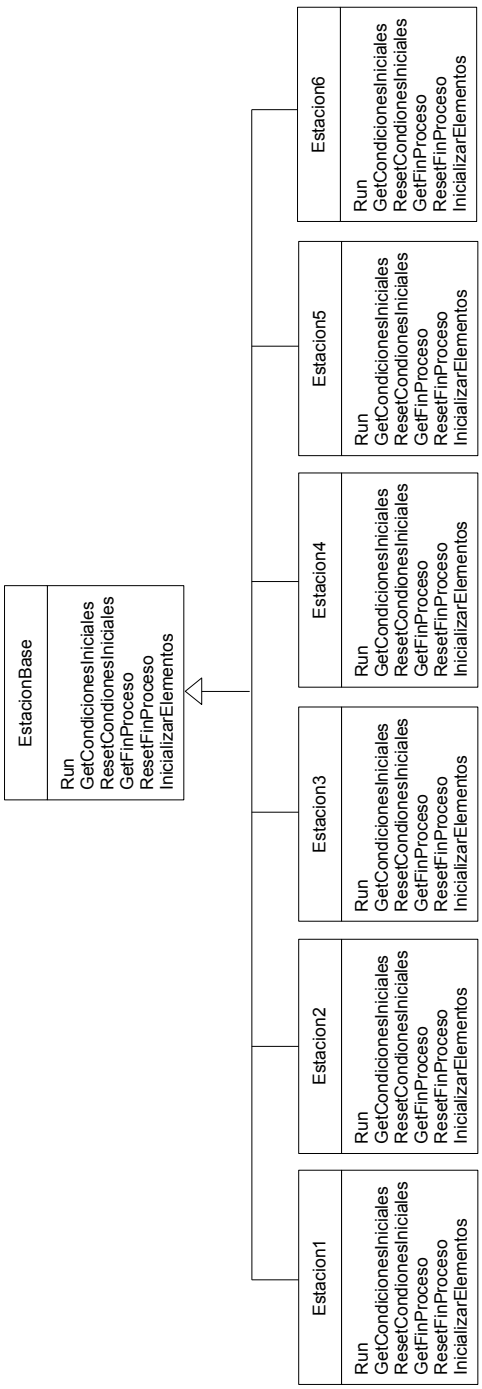

Figura 5.7: Herencia del interfaz *"EstacionBase"*

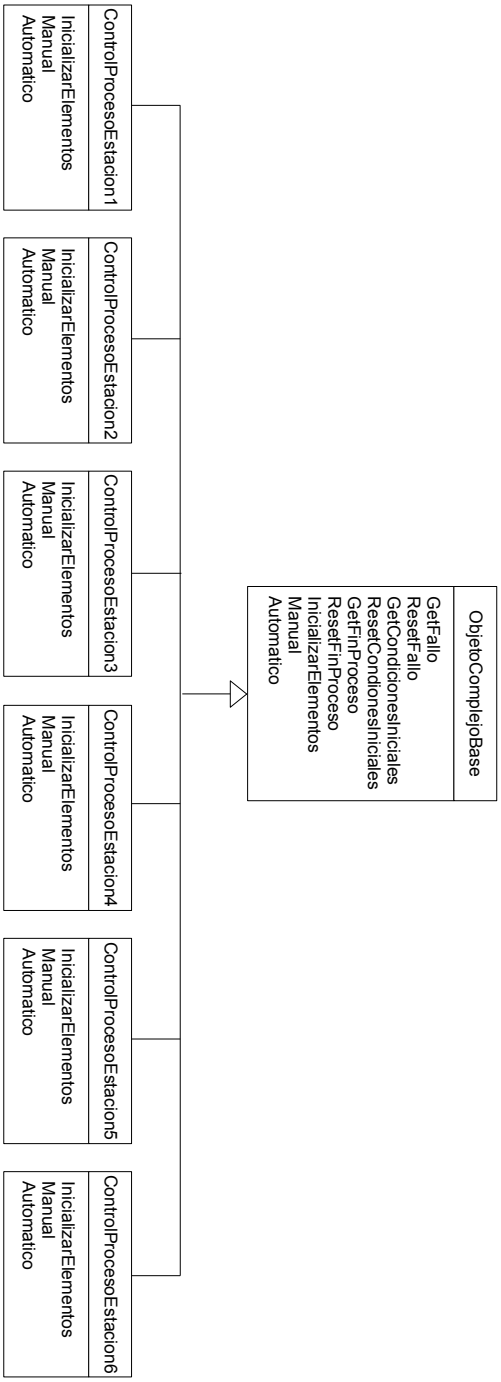

Figura 5.8: Herencia de las clases "*ControlProcesoEstacion*"

con su versión estructurada análoga para poder comprobar la diferencia de estilos entre ambas.

5.4. Estación transfer

El transfer de material conecta las estaciones de proceso para llevar a cabo de forma secuencial el montaje del dispositivo de giro. Además de suministrar la pieza ensamblada a cada estación, el *"transfer"* es el encargado de gestionar cada una de las estaciones indicando a cada una cuándo debe ejecutarse, controlar posibles errores de éstas, y en definitiva, supervisar todo el proceso de montaje.

Como muestra en la figura 5.9, se trata de un transfer de circulación rectangular, con recirculación continua de los palets, alrededor del cual se distribuyen las estaciones de proceso que realizan las diferentes fases de montaje.

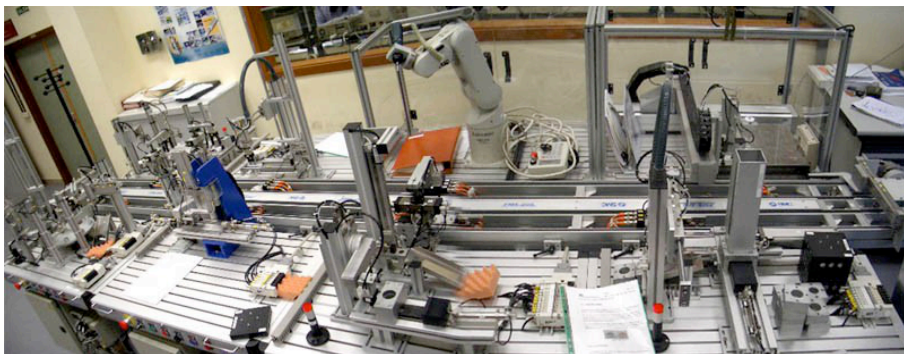

Figura 5.9: Transfer de material

El montaje de las estaciones se realiza mediante uniones atornilladas dispuestas al efecto, tanto en la estación como en el transfer, que permiten el ensamblado de forma rápida y precisa. Frente a cada estación se situa un tope mecánico para retener los palets, un sistema de lectura del código identificativo, y según el proceso a realizar, otra serie de elementos de elevación, centraje, giro, etc.

Tanto el punto de retención del palet, como la situación de la estación sobre éste último, pueden variarse para modificar de forma sencilla la distribución de los puestos que componen la célula.

5.4.1. Diagrama de clases

La versión OO del transfer se compone de 13 clases (incluyendo las 6 estaciones diferentes) y su diagrama de clases se muestra en la figura 5.10.

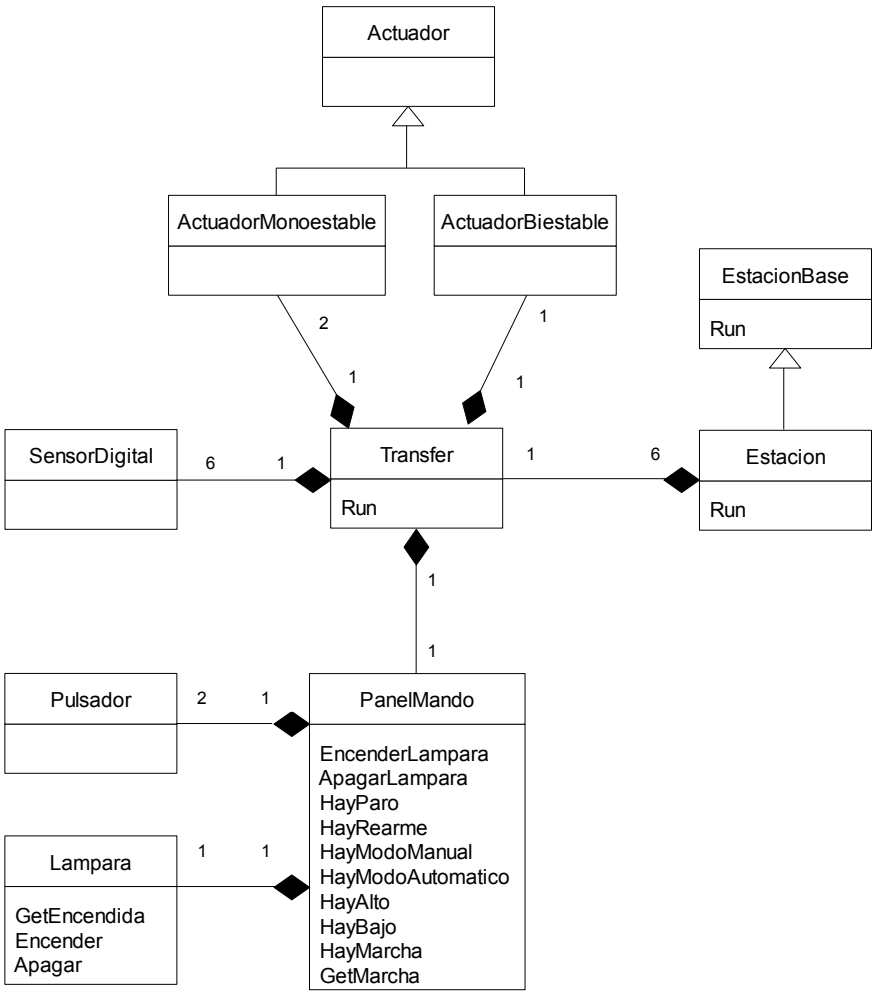

Figura 5.10: Diagrama de clases del transfer

5.4.2. Programación OO vs programación estructurada

En esta subsección se mostrará el proceso que permite la ejecución de las estaciones si el palet se encuentra en la estación correcta y ésta no está ocupada (proceso *"Run"*).

Proceso de ejecución de estaciones de la versión OO

En la figura 5.11 se muestra las ramas concernientes al proceso que permite ejecutar las distintas estaciones.

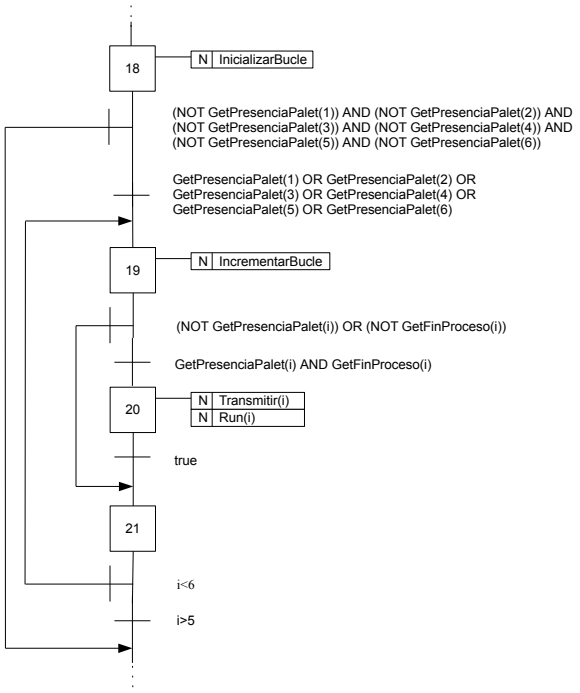

Figura 5.11: Proceso de ejecución de las estaciones en la versión orientada a objetos

Las etapas 18 y 19, así como las transiciones asociadas a la etapa 21, tienen la misión de simular un bucle *"FOR"* para la ejecución de las estaciones usando el polimorfismo.

La etapa 18 tiene una acción denominada *"InicializarBucle"* que contiene el código ST:

```
i:=0;
```

La etapa 19 tiene una acción denominada *"InicializarBucle"* que contiene el código ST:

```
i++;
```

La transición asociada a la etapa 18 cotiene una llamada al método *"GetPresenciaPalet"* perteneciente a la clase *"transfer"* (ver algoritmo 5.1). Para que dicho método pueda devolver el estado de los sensores mapeados por medio de las variables *"ppA"*, *"ppB"*, *"ppC"*, *"ppD"*, *"ppE"* y *"ppF"* que indican la presencia del palet en cada estación, previamente hay que que invocar el método *"SetPresenciaPalet"* (ver algoritmo 5.2) que actualiza el valor del vector *"Vector_pp"* en el que cada posición del array está asociado a cada uno de los sensores, a saber:

- Vector_pp[1] -> ppA
- Vector_pp[2] -> ppB
- Vector_pp[3] -> ppC
- Vector_pp[4] -> ppD
- Vector_pp[5] -> ppE
- Vector_pp[6] -> ppF

Algoritmo 5.1 Método *"GetPresenciaPalet"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::GetPresenciaPalet (i:BYTE)
  RETURN Vector_pp[i];
END_METHOD
```

La transición asociada a la etapa 18 cotiene una llamada al método *"GetFinProceso"* perteneciente a la clase *"Transfer"* (ver algoritmo 5.3). Este método, que manipula el vector *"Vector_Estacion"*, permite invocar cada uno de los métodos particulares *"GetFinProceso"* de cada estaciones gracias a la ligadura dinámica. El vector *"Vector_Estacion"* contiene una referencia a cada uno de los objetos de las

Algoritmo 5.2 Método *"SetPresenciaPalet"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::SetPresenciaPalet (i:BYTE)
  CASE i OF
      1 :  Vector_pp[i]:=ppA; RETURN true;
      2 :  Vector_pp[i]:=ppB; RETURN true;
      3 :  Vector_pp[i]:=ppC; RETURN true;
      4 :  Vector_pp[i]:=ppD; RETURN true;
      5 :  Vector_pp[i]:=ppE; RETURN true;
      6 :  Vector_pp[i]:=ppF; RETURN true;
  ELSE
     RETURN false;
  END_CASE
END_METHOD
```

Algoritmo 5.3 Método *"GetFinProceso"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::GetFinProceso (i:BYTE)
  RETURN Vector_Estacion[i].GetFinProceso;
END_METHOD
```

Algoritmo 5.4 Método *"InicializarVectorEstaciones"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::InicializarVectorEstaciones ( )
   Vector_Estacion(1):=Estacion1;
   Vector_Estacion(2):=Estacion2;
   Vector_Estacion(3):=Estacion3;
   Vector_Estacion(4):=Estacion4;
   Vector_Estacion(5):=Estacion5;
   Vector_Estacion(6):=Estacion6;
END_METHOD
```

estaciones. En el algoritmo 5.4 se muestra el método *"InicializarVectorEstaciones"* donde se hace la inicialiación del vector. En cada posición del vector se inserta un objeto estación. Las 6 estaciones son atributos privados de la clase *"Transfer"*.

En la etapa 20 se hace una llamada a al método *"Transmitir"* (ver algoritmo 5.5). Éste método coloca un bit en la posición de memoria compartida por todos los PLCs de las estaciones para comenzar la ejecución de éstas. En el caso del *"transfer"*, estas posiciones de memoria están mapeadas a las variables *"mm"*, a saber:

- mmA -> Estación de montaje de bases.
- mmB -> Estación de montaje de rodamientos.
- mmC -> Estación de prensado.
- mmD -> Estación de inserción de ejes.
- mmE -> Estación de montaje de tapas.
- mmF -> Estación de inserción de tornillos.

Por otro lado, en la misma etapa 20 se hace una invocación del método *"Run"* de la propia clase *"Transfer"*. En dicho método (ver algoritmo 5.6) se hacen las invocaciones a los métodos *"Run"* particulares de cada estación por medio del polimorfismo.

Proceso de ejecución de estaciones de la versión estructurada

En la figura 5.12 se muestrala parte del GRAFCET que controla la ejecución de las estaciones.

Las variables denominadas *"estacion"* son flags booleanos que indican al transfer si una estación esta en ejecución. La comprobación inicial del estado de las estaciones se hace previamente al GRAFCET que se muestra, así como la inicialización de las estaciones.

Las variables etiquetadas como *"pp"* indican si el palet ha llegado a una estación, siendo:

- *"ppA"* detecta la presencia del palet en la estación de bases (estación 1).

Algoritmo 5.5 Método *"Transmitir"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::Transmitir (i:BYTE)
  mmA:=0;
  mmB:=0;
  mmC:=0;
  mmD:=0;
  mmE:=0;
  mmF:=0;
  CASE i OF
    1 : mmA:=1;
    2 : mmB:=1;
    3 : mmC:=1;
    4 : mmD:=1;
    5 : mmE:=1;
    6 : mmF:=1;
  END_CASE
END_METHOD
```

Algoritmo 5.6 Método *"Run"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::Run (i:BYTE)
  Vector_Estacion[i].Run();
END_METHOD
```

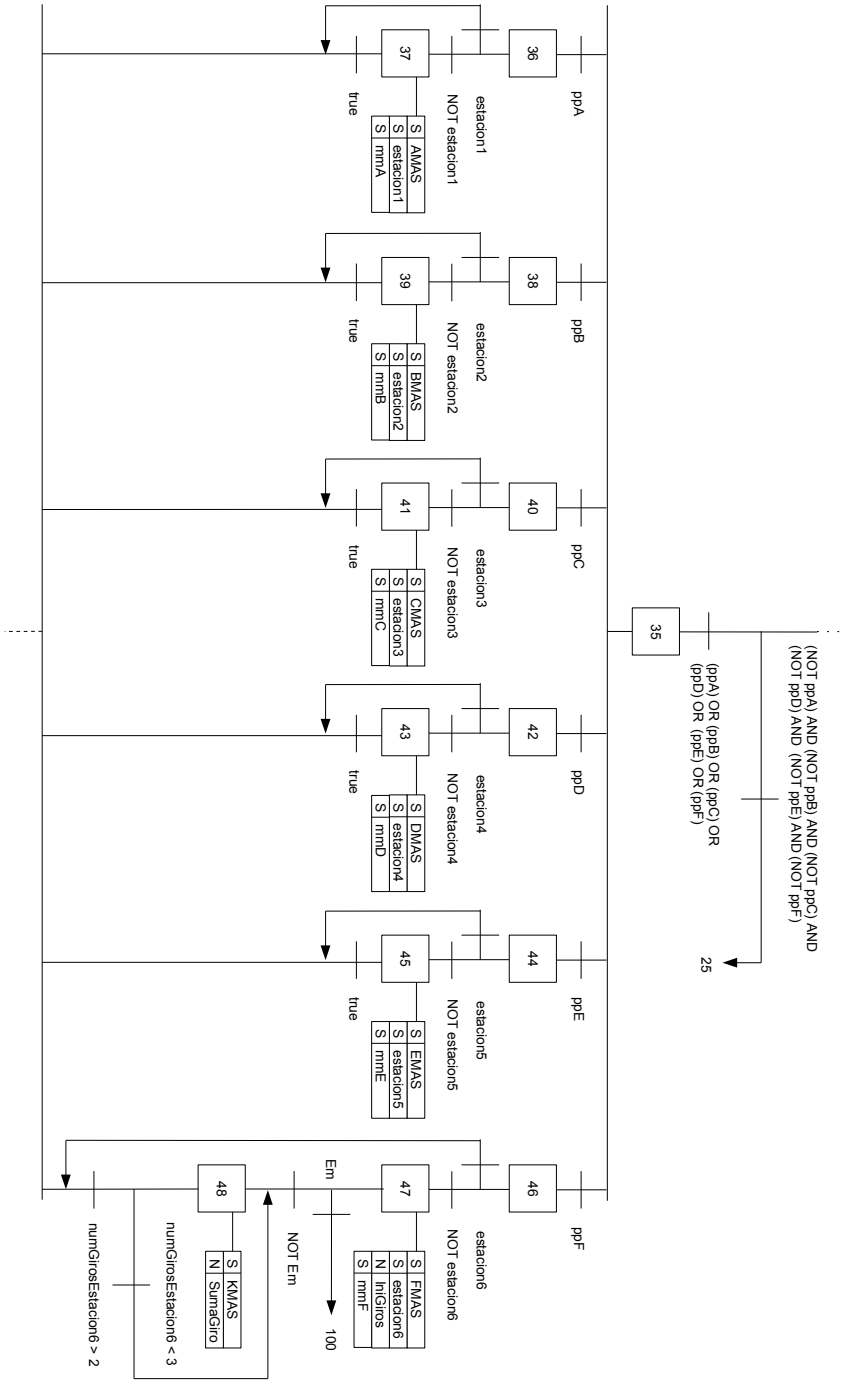

Figura 5.12: Proceso de ejecución de las estaciones en la versión estructurada

- *"ppB"* detecta la presencia del palet en la estación de rodamientos (estación 2).
- *"ppC"* detecta la presencia del palet en la estación de prensa (estación 3).
- *"ppD"* detecta la presencia del palet en la estación de ejes (estación 4).
- *"ppE"* detecta la presencia del palet en la estación de tapas (estación 5).
- *"ppF"* detecta la presencia del palet en la estación de tornillos (estación 6).

Las variables etiquetadas como *"mm"* están mapeadas a las posiciones de memoria compartidas por todos los PLCs que marcan el comienzo de la ejecución de cada estación, siendo:

- *"mmA"* posiciona un bit en la memoria compartida por la estación de montaje de bases (estación 1).
- *"mmB"* posiciona un bit en la memoria compartida por la estación de inserción de rodamientos (estación 2).
- *"mmC"* posiciona un bit en la memoria compartida por la estación de prensa (estación 3).
- *"mmD"* posiciona un bit en la memoria compartida por la estación de inserción de ejes (estación 4).
- *"mmE"* posiciona un bit en la memoria compartida por la estación de montaje de tapas (estación 5).
- *"mmF"* posiciona un bit en la memoria compartida por la estación de inserción de tornillos (estación 6).

La etapa 37 tiene asociada la acción *"AMAS"* que se corresponde con la señal de avance del tope de la estación de bases (estación 1).

La etapa 39 tiene asociada la acción *"BMAS"* que se corresponde con la señal de avance del tope de la estación de rodamientos (estación 2).

La etapa 41 tiene asociada la acción *"CMAS"* que se corresponde con la señal de avance del tope de la estación de prensa (estación 3).

La etapa 43 tiene asociada la acción *"DMAS"* que se corresponde con la señal de avance del tope de la estación de ejes (estación 4).

La etapa 45 tiene asociada la acción *"EMAS"* que se corresponde con la señal de avance del tope de la estación de tapas (estación 5).

La etapa 47 tiene asociada la acción *"FMAS"* que se corresponde con la señal de avance del tope de la estación de tornillos (estación 6), y la acción *"iniGiros"* que contiene el código en ST siguiente:

```
numGirosEstacion6:=0;
```

La etapa 47 tiene asociada la transición *"Em"* que es un sensor que captura la presión de la seta de emergencia.

La etapa 48 tiene asociada la acción *"KMAS"* que se corresponde con la señal de giro del palet de tornillos, y la acción "sumaGiro" que contiene el código en ST siguiente:

```
numGirosEstacion6:=numGirosEstacion6+1;
```

5.5. Estación 1 - Estación de montaje de bases

Esta estación desempeña el primero de los procesos de la línea de montaje. Su función es la de situar la base sobre un palet vacío en la cinta de transferencia. Esta base es la parte estructural del producto final ya que servirá como soporte al dispositivo de giro (ver figura 5.13). Por esta razón, su funcionamiento es indispensable sea cual sea la referencia o especificación de la pieza en proceso. Su puesta en marcha o parada determinará la continuidad del ciclo productivo.

El ciclo de trabajo comienza con la extracción de una base del alimentador de petaca. Tras comprobar su orientación, la pieza es trasladada hacia la zona de montaje donde, en función del resultado de la prueba, será desechada mediante un cilindro empujador o montada en el palet, previamente en situación de espera en el sistema de transferencia.

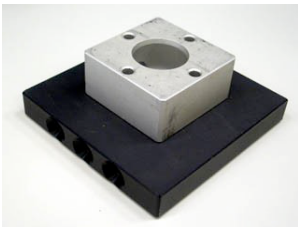

Figura 5.13: Base montada

5.5.1. Etapas del proceso

Todo el proceso requiere de una serie de operaciones que se pueden separar en etapas bien diferenciadas:

Extracción de una base del alimentador

El dispositivo alimentador es de *"tipo petaca"* o también denominado por gravedad. Las piezas, almacenadas unas encima de otras, irán cayendo por su propio peso a medida que se sustraen las bases desde la zona inferior de la columna de almacenaje. El avance de un cilindro neumático facilita la extracción de la pieza. La forma específica de su empujador asegura la obtención de una sola base y no permite que caigan nuevas piezas hasta que haya retrocedido completamente.

Comprobación de la orientación

La base dispone de un hueco circular en el centro donde irán alojados posteriormente rodamiento, eje y tapa. Para que la inserción de estos componentes sea posible, la base deberá situarse en el palet con la cara con el hueco de mayor diámetro hacia arriba. Dado que la pieza proveniente del alimentador puede tener cualquier orientación, es necesario comprobar la orientación de ésta, asegurando el correcto ensamblaje posterior y evitando interferencias que puedan dañar los equipos.

La verificación se realiza mediante el avance de un cilindro cuyo vástago dispone de un empujador con forma cilíndrica que se ajusta al alojamiento circular de la base. Una orientación incorrecta impide la carrera completa del cilindro y la consiguiente activación del detector magnético correspondiente. Tras una cierta espera, el programa de control del PLC identifica dicha base como mala, para poder ser desechada posteriormente.

Traslado a la zona de trasvase

Para su posterior tratamiento, la base es transferida hacia la zona de trasvase mediante un cilindro neumático de sección rectangular, con el fin de evitar el giro del empujador. El carril de desplazamiento está plano y rodeado por guías para asegurar un transporte correcto de la pieza.

Rechazo de base incorrecta

Si el proceso comprobador realizado anteriormente, puso de manifiesto una orientación incorrecta de la base, ésta es desechada. Para ello, se emplea un cilindro de simple efecto que expulsa la pieza hacia una rampa de desalojo de material, dejando vacía la zona de trasvase. Tras este punto se reinicia el ciclo de la estación, cuyos elementos están libres de interferencias y preparados para el tratamiento de una nueva pieza.

Montaje de la base en el palet

Si la orientación de la base es la adecuada, ésta deberá ser colocada en un palet vacío situado en el sistema de transfer de material. Para ello, se dispone de un manipulador de dos ejes, cada uno de ellos constituido por un cilindro de vástagos paralelos. El eje horizontal permite el movimiento del eje vertical entre el punto de trasvase y la vertical del palet, mientras que el eje vertical permite tomar o dejar la pieza en su posición inferior y elevarla para el trasvase en su posición superior.

Finalmente, el eje vertical incorpora una plataforma de sujeción por vacío que incluye cuatro ventosas telescópicas con el fin de absorber posibles desalineaciones en altura.

La manipulación de la pieza se consigue por aspiración mediante un eyector de vacío, disponiéndose a su vez de un vacuostato que proporciona una señal al PLC en caso de que la sujeción sea correcta. El caso contrario pone de manifiesto la ausencia de la pieza propiciada por una falta de material en la columna de alimentación, la cuál deberá ser subsanada por un operario.

5.5.2. Diagrama de clases

La versión OO de la estación 1 se compone de 14 clases y su diagrama de clases se muestra en la figura 5.14.

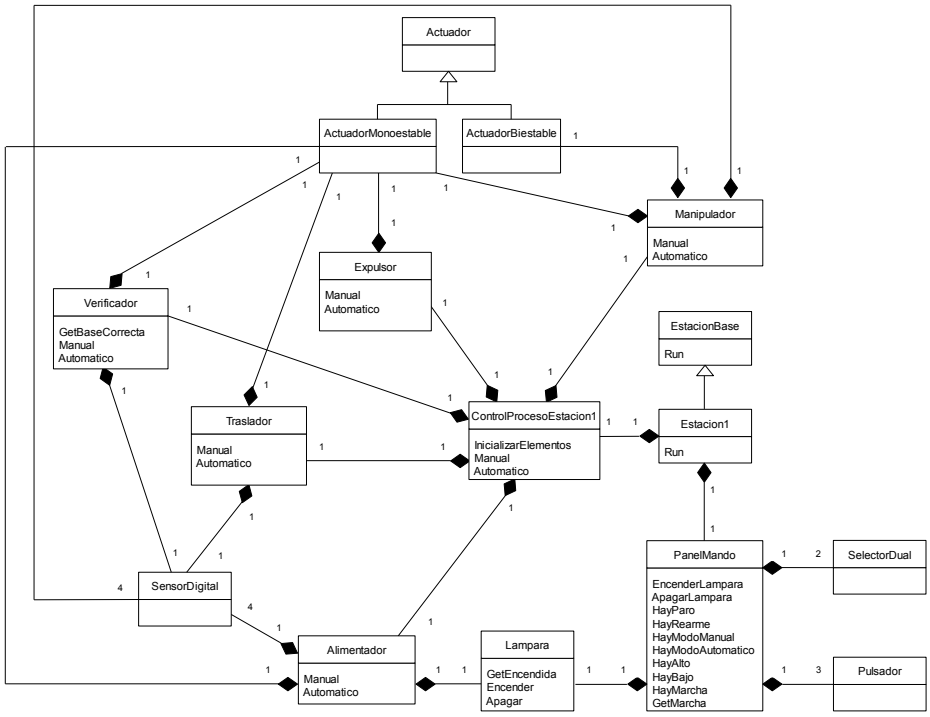

Figura 5.14: Diagrama de clases de la estación 1

5.5.3. Programación OO vs programación estructurada

En esta subsección se mostrarán los procesos de extracción de una base del alimentador (proceso *"alimentador"*) y la comprobación de la orientación de la base donde irán alojados los ejes y rodamientos (proceso *"verificador"*).

Método *"Run"* de la clase Estacion1

En la figura 5.15 se muestra las ramas concernientes a la ejecución automática de la estación 1.

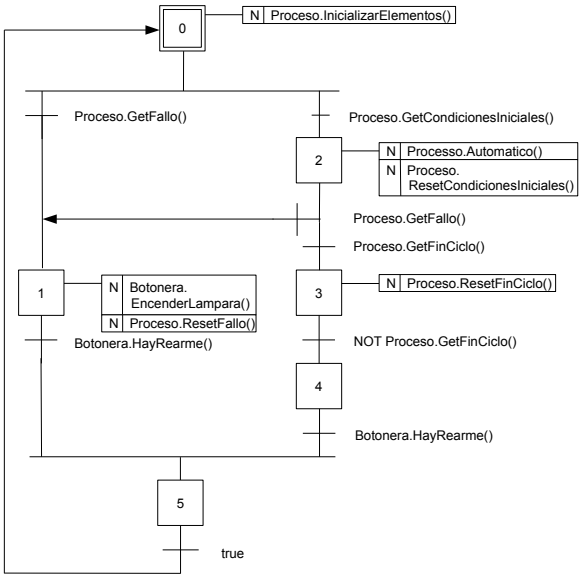

Figura 5.15: Método *"Run"* de la clase *"Estación1"*

Proceso alimentador de la versión OO

La etapa 2 del método *"Run"* de la versión OO de la estación 1(ver figura 5.15), tiene una acción invocación al método *"Automatico"* perteneciente a la clase *"ControlProcesoEstacion1"* que se encarga de realizar la ejecución automática de la estación. La clase *"ControlProcesoEstacion1"*, instanciada por medio del objeto *"Proceso"*, engloba todos los procesos necesarios para llevar a cabo los acciones que debe realizar la estación 1. En la figura 5.16 se muestra el trozo de código

del método de ejecución automática, *"Automatico"*, correspondiente al proceso de extracción de una base.

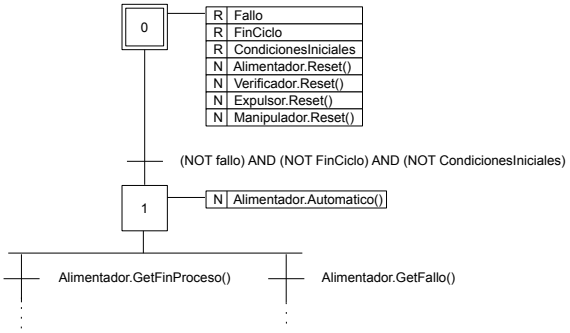

Figura 5.16: Proceso de ejecucion automatico del alimentador

La parte interesante del método *"Automatico"* correspondiente a la acción de extracción de una base, se encuentra en la etapa 1 donde se realiza una llamada al método *"Automatico"* de la clase *"Alimentador"* que es quien realiza la tarea de extracción de la base de forma automática (ver figura 5.17).

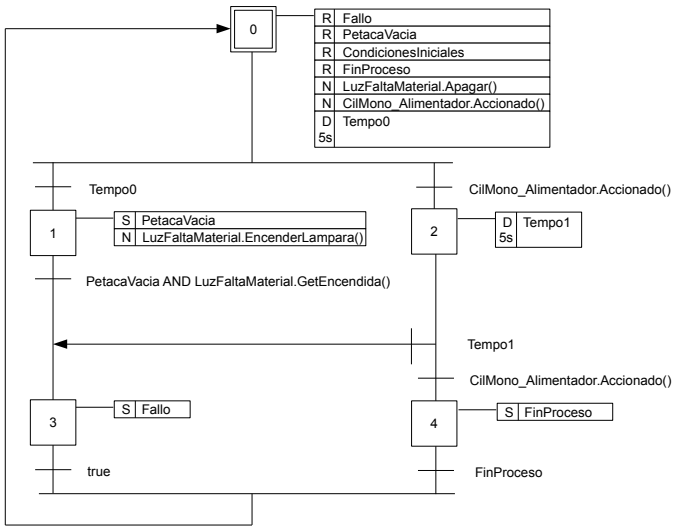

Figura 5.17: Método *"Automatico"* de la clase *"Alimentador"*

Los objetos *"CilMono_ Alimentador"* y *"LuzFaltaMaterial"* son dos objetos agregados a la clase *"Alimentador"* que se corresponden con las clases *"ActuadorMo-*

noestable" y *"Lampara"* respectivamente.

Las variables *"Fallo"*, *"PetacaVacia"*, *"CondicionesIniciales"* y *"FinProceso"* son atributos privados boolean de la clase *"Alimentador"*.

Proceso alimentador de la versión estructurada

En la figura 5.18 se muestra un trozo del GRAFCET que controla la extracción de las bases del alimentador.

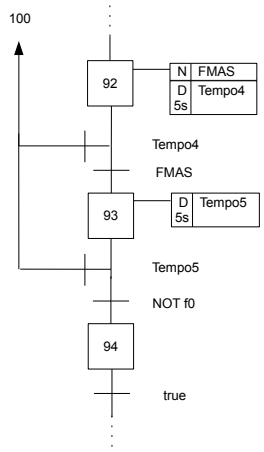

Figura 5.18: Proceso alimentador de la versión estructurada

La etapa 92 tiene asociada la acción *"FMAS"* que se corresponde con la señal de avance del alimentador.

La etapa 100, que no se muestra en la figura 5.18, tiene asociada la transición *"DEFECTO"* que se corresponde con el piloto luminoso por defecto de la estación.

La etapa 93 tiene asociada la transición *"f0"* que se corresponde con el final de carrera del vastago del alimentador cuando se encuentra atrás.

Proceso verificación de la versión OO

En la figura 5.19 se muestra otro trozo del código correspondiente al método *"Automatico"* del objeto *"Proceso"*. Esta parte del código se encarga de comprobar la orientación correcta de la base.

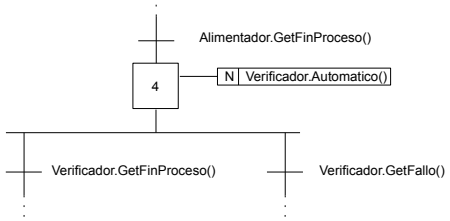

Figura 5.19: Proceso de verificación del método *"Automatico"* de la clase *"ControlProcesoEstacion1"*

La acción de verificación de forma automática de la orientación de la base se corresponde con la llamada al método *"Automatico"* de la clase *"Verificador"* que es la que realiza la tarea(ver figura 5.20).

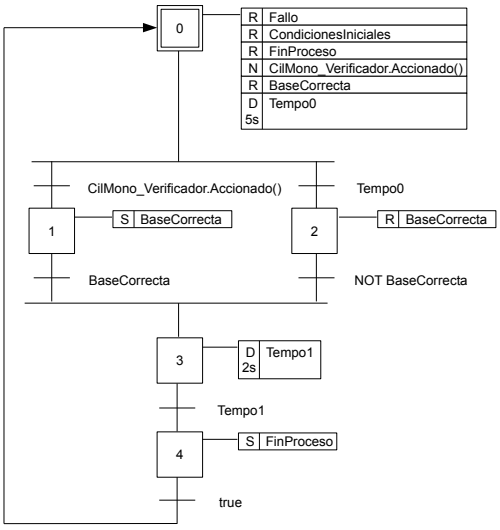

Figura 5.20: Método *"Automatico"* de la clase *"Verificador"*

Las acciones de la etapa 0 se corresponden con:

- "*Fallo*", *"BaseCorrecta"*, *"CondicionesIniciales"* y *"FinVerificador"* son atributos boolean de la clase.
- *"CilMono_ Alimentador.Accionado"* es una llamada al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

Proceso verificación de la versión estructurada

En la figura 5.21 se muestra un trozo del GRAFCET que controla la verificación de la orientación de las bases.

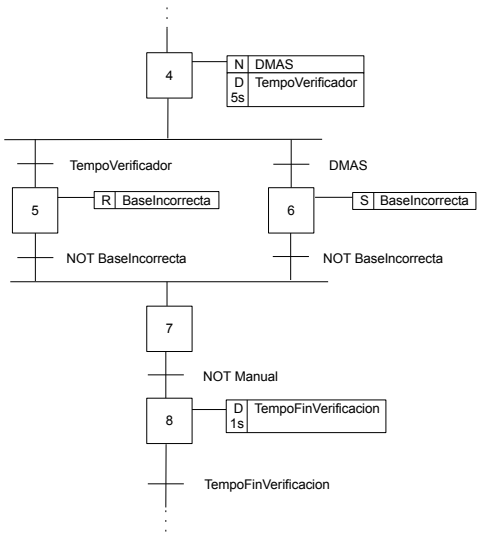

Figura 5.21: Proceso de verificación de la versión estructurada

La etapa 4 tiene asociada la acción *"DMAS"* que se corresponde con la señal de avance del verificador de posición.

5.5.4. Resultados de tiempos del módulo TCLOCK

Los tiempos de ejecución de la estación de montaje de bases está medida tras la realiación de 20 mediciones de las cuales, en 10 se aceptaba la pieza y en las otras 10 la pieza era rechazada. En las tablas se muestran, por una parte el número de ciclos de SCAN que se han necesitado para completar un proceso completo, y por otro el tiempo de ejecución de cada proceso medido en milisegundos.

En la tabla 5.22 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma OO.

En la tabla 5.23 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma estructurado.

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Verificación	Traslado	Expulsión	Manipulación
4631	2.12	26	2113	1124	1753	0	4525
4662	2.11	26	2113	1122	1755	0	4535
4721	2.11	27	2128	1124	1754	0	4538
4660	2.11	27	2102	1121	1754	0	4527
4757	2.11	29	2267	1127	1751	0	4535
4701	2.11	28	2176	1124	1754	0	4544
4696	2.11	29	2119	1121	1753	0	4537
4727	2.11	27	2107	1125	1753	0	4535
4666	2.11	28	2136	1125	1752	0	4531
4622	2.11	26	2123	1125	1753	0	4536
3536	2.12	26	2088	3010	1754	106	0
3513	2.11	29	2262	3012	1750	107	0
3548	2.12	29	2186	3015	1753	105	0
3594	2.11	28	2249	3012	1751	108	0
3444	2.12	26	2165	3012	1751	108	0
3400	2.12	29	2127	3012	1756	107	0
3517	2.12	27	2208	3010	1752	107	0
3485	2.11	28	2303	3012	1753	107	0
3481	2.11	26	2150	3010	1752	105	0
3446	2.11	29	2134	3011	1752	107	0

Figura 5.22: Tiempos de ejecución de la estación 1 en su versión OO

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Verificación	Traslado	Expulsión	Manipulación
4713	2.08	19	2088	1124	1755	0	4554
4702	2.09	19	2141	1118	1754	0	4533
4688	2.09	19	2168	1123	1749	0	4529
4633	2.09	19	2158	1124	1749	0	4520
4698	2.09	17	2114	1122	1751	0	4526
4670	2.09	18	2087	1122	1749	0	4524
4623	2.09	19	2116	1115	1748	0	4523
4662	2.09	18	2099	1117	1750	0	4515
4602	2.09	18	2079	1119	1749	0	4512
4599	2.09	21	2081	1119	1748	0	4511
3556	2.09	19	2169	3008	1753	103	0
3687	2.09	20	2317	3009	1750	104	0
3519	2.09	19	2167	3008	1750	105	0
3468	2.09	18	2149	3008	1749	104	0
3513	2.09	19	2275	3005	1748	103	0
3452	2.09	19	2154	3004	1751	102	0
3466	2.09	18	2088	3008	1751	102	0
3467	2.09	19	2220	3007	1747	103	0
3468	2.09	18	2127	3007	1751	102	0
3443	2.09	18	2146	3007	1748	102	0

Figura 5.23: Tiempos de ejecución de la estación 1 en su versión estructurada

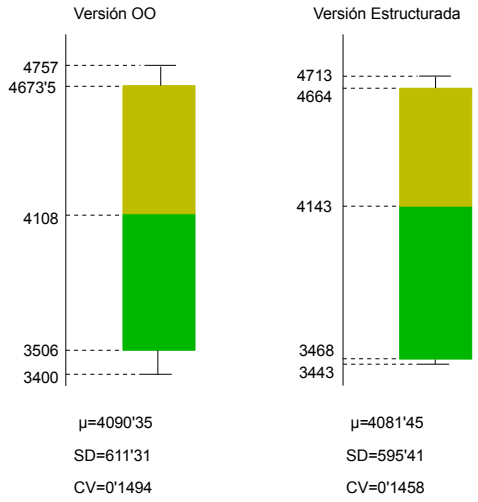

Figura 5.24: Diagrama de cajas del proceso *"número de ciclos"* de la estación 1

A continuación, se muestra de forma gráfica, usando los diagramas de caja, las cuartiles de los tiempos de ejecución de los distintos procesos que conforman la estación de montaje de bases, así como sus medias, desviación típica y covarianza de dichos tiempos.

En la figura 5.24 se puede observar el diagrama de cajas de los ciclos ejecutados por la estación para llevar a cabo el montaje de bases.

En la figura 5.25 se muestra el diagrama de cajas del proceso que permite inicializar los objetos que conforman la estación dejando la posición inicial todos los elementos de la misma.

En la figura 5.26 se muestra el diagrama de cajas del proceso que permite sacar una base del alimentador *"tipo petaca"*.

En la figura 5.27 se muestra el diagrama de cajas del proceso que verifica la correcta orientacion de las bases.

En la figura 5.28 se muestra el diagrama de cajas del proceso que transfiere la base a la zona de trasvase.

En la figura 5.29 se muestra el diagrama de cajas del proceso que saca de la zona de trasvase las bases mal orientadas.

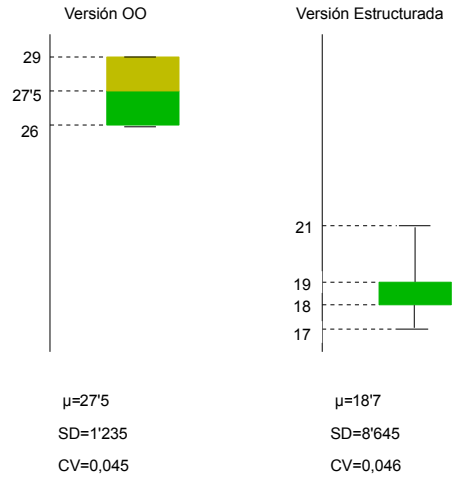

Figura 5.25: Diagrama de cajas del proceso *"Inicialización"* de la estación 1

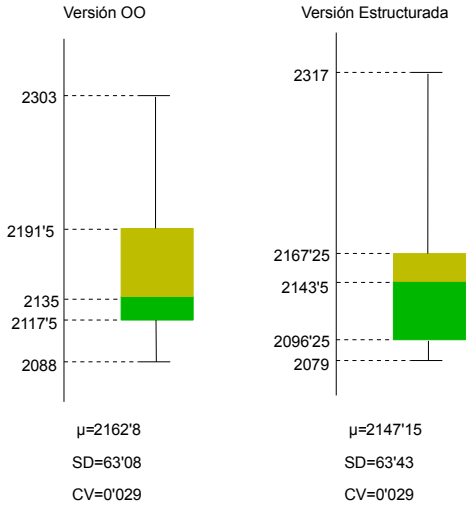

Figura 5.26: Diagrama de cajas del proceso *"Alimentación"* de la estación 1

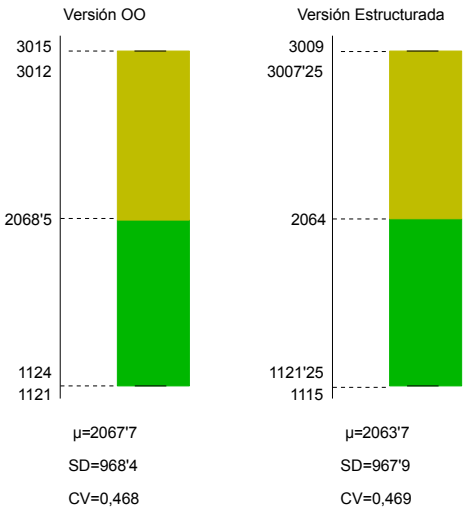

Figura 5.27: Diagrama de cajas del proceso *"Verificación"* de la estación 1

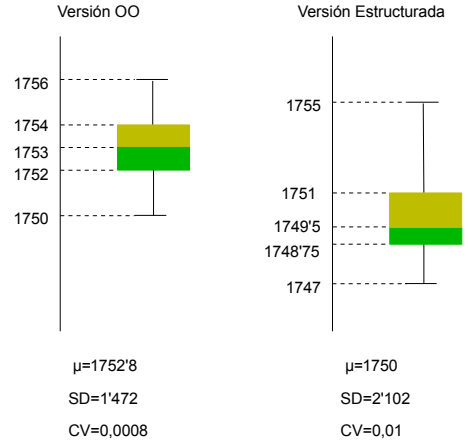

Figura 5.28: Diagrama de cajas del proceso *"Traslado"* de la estación 1

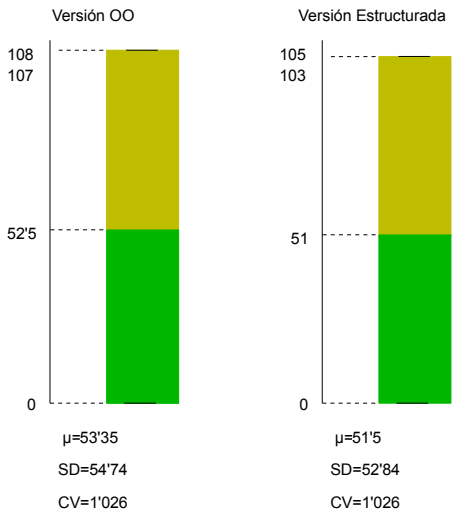

Figura 5.29: Diagrama de cajas del proceso *"Expulsión"* de la estación 1

En la figura 5.30 se muestra el diagrama de cajas del proceso que coloca una base correcta (bien orientada) en el palet.

5.6. Estación 2 - Estación de montaje de rodamientos

La operación realizada por la segunda de las estaciones consiste en la colocación de un rodamiento dentro del alojamiento realizado al efecto en la base (ver figura 5.31).

Dicha tarea de colocación se realizará sobre el palet llegado a través del sistema de transferencia, con la base situada previamente en la estación anterior. La inserción del rodamiento exige que el palet sobre el que se coloca la base, se encuentre en una posición determinada con una cierta precisión. Un centraje está realizado por medio de cuatro pines que se insertan en unos alojamientos realizados al efecto en la parte inferior del palet.

La estación 2 permite también al usuario seleccionar el tipo de rodamiento a insertar entre dos posibles: alto y bajo. El sistema, mediante una medición, expulsa el rodamiento si se no corresponde al pedido o sigue el tratamiento si está bien.

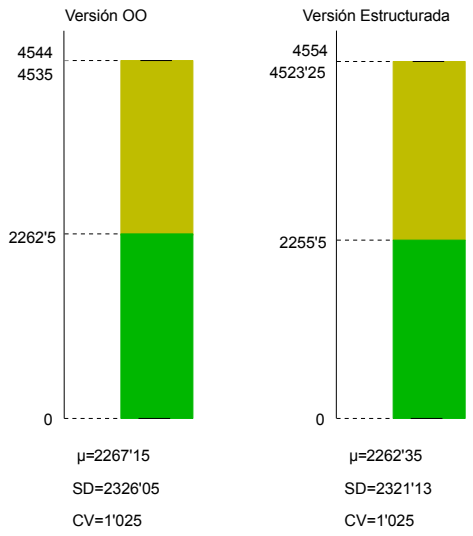

Figura 5.30: Diagrama de cajas del proceso *"Manipulación"* de la estación 1

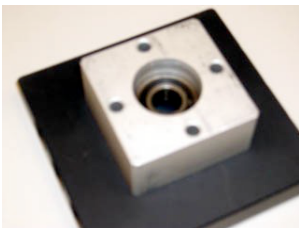

Figura 5.31: Rodamiento dentro de la base

5.6.1. Etapas del proceso

La inserción del rodamiento precisa de la realización de una serie de operaciones llevadas a cabo por los siguientes módulos:

Alimentación del rodamiento

Los rodamientos se encuentran ubicados en un alimentador por gravedad. El almacén está constituido por una columna que guía los rodamientos y de un cilindro empujador en la parte inferior que extrae el cojinete en el momento que se desea comenzar el ciclo. Al igual que en la primera estación, el empujador no deja caer a los rodamientos hasta que el cilindro esta en posición inicial.

En este caso, se dispone de un sensor de presencia de rodamiento realizado mediante un microrruptor que permite al PLC verificar que tras realizar una alimentación, se ha extraído realmente un cojinete. Dicho elemento posibilita determinar en que momento han agotado los rodamientos cargados en el alimentador, en cuyo caso el usuario tiene que solucionar la falta material.

Trasvase a la estación de medida

Para el desplazamiento del rodamiento desde el punto de alimentación al lugar donde se realiza la siguiente operación, se utiliza un manipulador realizado a partir de un actuador de giro tipo piñón – cremallera, que describe un ángulo de 180°.

Al eje del actuador de giro se dispone de un brazo encargado de desplazar una pinza de dos dedos de apertura paralela, la cual sujeta el rodamiento por la parte interna. Este brazo alberga en su interior un mecanismo construido por una correa dentada y dos piñones, consiguiéndose de esta manera que a lo largo de todo el giro la pinza no cambie de orientación, de forma que el rodamiento se deposite en el punto final sin ángulo alguno de inclinación.

Medición de la altura

La estación contempla la posibilidad de alimentar rodamientos con alturas distintas, habiéndose incluido un módulo de medición para diferenciarlas. El rodamiento se deposita en una plataforma, sobre un centrador accionado mediante un cilindro neumático que lo sitúa en un punto inicial con alta precisión.

Dicha plataforma es elevada mediante un cilindro neumático sin vástago, de forma que el rodamiento contacta con un palpador a través del cual se obtiene una medición de su altura. El palpador está constituido por un potenciómetro lineal cuya salida es tratada mediante un módulo analógico incluido en el PLC. Este módulo manda una señal al autómata para que el pueda comparar el pedido con lo que hay.

Tras realizar la medición, el elevador vuelve a su posición original, momento en el cual, en caso de que la altura no coincida con la deseada, un cilindro expulsor empuja el rodamiento hacia un recipiente de recogida. Después de este rechazo, el sistema empieza de nuevo desde el principio con otro rodamiento. Si la altura corresponde, la estación sigue el tratamiento.

Inserción del rodamiento

La última de las operaciones es realizada por un manipulador compuesto por una unidad rotolineal. Dicha unidad dispone de un brazo en el cual se ha colocado una pinza de dos dedos. Tras coger el rodamiento, se eleva el brazo para posteriormente realizar un giro de 180° que al realizar una nueva bajada inserta el rodamiento en el alojamiento realizado en la base.

La pinza siempre está hacia abajo porque el brazo solamente gira, baja o sube. De este modo no hay ningún problema para la inserción del rodamiento en la base.

5.6.2. Diagrama de clases

La versión OO de la estación 2 se compone de 14 clases y su diagrama de clases se muestra en la figura 5.32.

5.6.3. Programación OO vs programación estructurada

En esta subsección se mostrarán los procesos de desplazamiento del rodamiento desde el punto de alimentación (proceso *"Trasvase"*) y medición de los rodamientos (proceso *"Medidor"*) .

Método *"Run"* de la clase Estacion2

En la figura 5.33 se muestra las ramas concernientes a la ejecución automática de la estación 2.

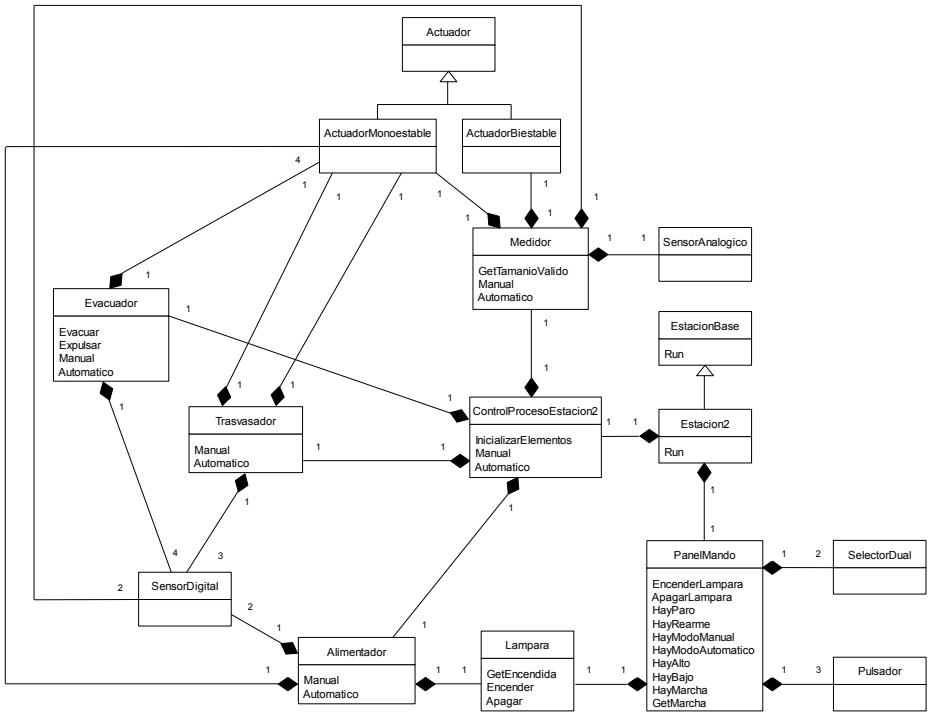

Figura 5.32: Diagrama de clases de la estación 2

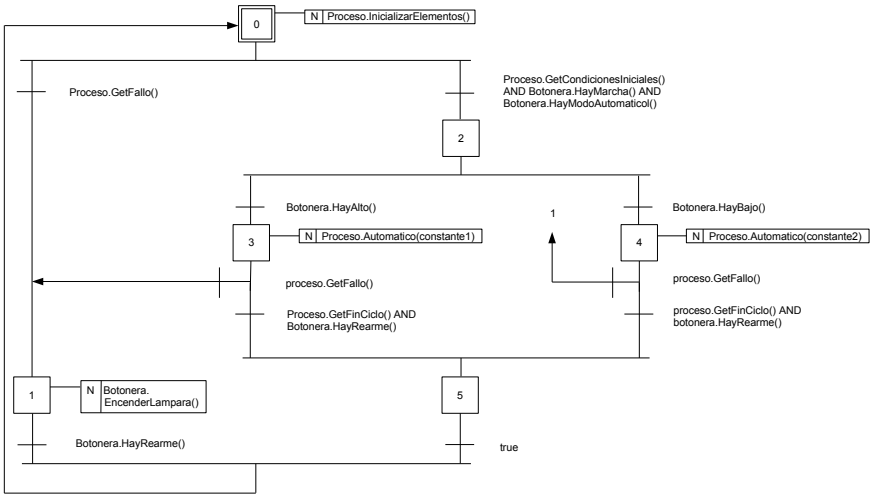

Figura 5.33: Método *"Run"* de la clase *"Estacion2"*

Proceso trasvasador de la versión OO

La etapa 3 y 4 del método *"Run"* de la versión OO de la estación 2 (ver figura 5.33) tienen asociados una llamada al método *"Automatico"* de la clase *"ControlProcesoEstacion2"*, pasandole las constantes *"Constante1"* y *"Cosntante2"* respectivamente. Estas constantes contienen el tamaño del rodamiento y su selección depende de la posición que tenga el selector de tamaño de rodamientos. En la figura 5.34 se muestra el trozo de código del método *"Automatico"* correspondiente al proceso de desplazamiento del rodamiento de forma automática.

La etapa 2 tiene asociada una invocación al método *"Automatico"* de la clase *"Trasvasador"* que es quien realiza la tarea de mover el rodamiento (ver figura 5.35).

Las acciones de la etapa 0 se corresponden con:

- "*Fallo", "CondicionesIniciales" y "FinProceso"* son atributos boolean de la clase.
- *"CilBi_Brazo.Accionado"* es una llamada al método *"Accionado"* de la clase *"ActuadorBiestable"*.

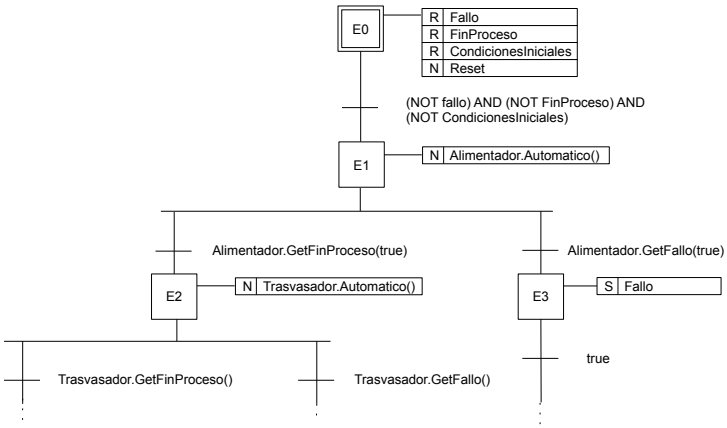

Figura 5.34: Método *"Automatico"* de la clase *"ControlProcesoEstacion2"*

La etapa 1, 3, 4 y 6 tienen una transición previa de franqueo asociada a una llamada al método *"Activado"* de la clase *"SensorDigital"*.

La etapa 1,4 y 5 tienen una acción en la que se llama al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

Proceso trasvasador de la versión estructurada

En la figura 5.36 se muestra un trozo del GRAFCET que controla el proceso de alimentación de la prensa.

La etapa 112 tiene asociada la acción *"BMENOS"* que se corresponde con el retroceso del manipulador de trasvase.

La etapa 112 tiene asociada la transición *"b0"* que se corresponde con la detección de la posición inicial del manipulador de trasvase.

La etapa 121 tiene asociada la acción *"CMAS"* que se corresponde con la apertura de la pinza del manipulador de trasvase.

La etapa 122 tiene asociada la acción *"BMAS"* que se corresponde con el avance del manipulador de trasvase.

La etapa 122 tiene asociada la transición *"NOT IMAS"* que se corresponde con la apertura de la pinza del manipulador de inserción.

La etapa 124 tiene asociada la transición *"b1"* que se corresponde con la detección de la posición central del manipulador de trasvase.

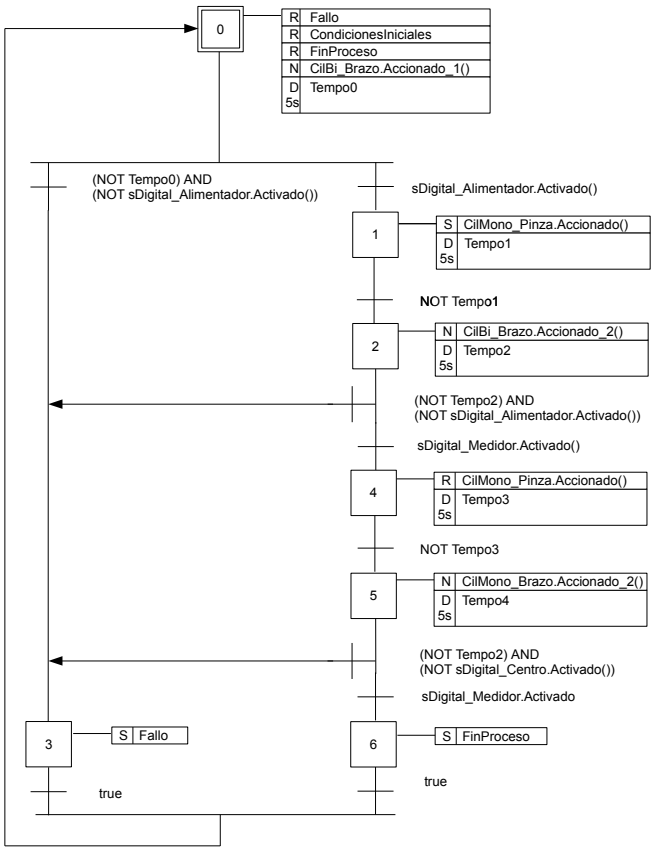

Figura 5.35: Método de ejecucion automatica de la clase *"Trasvasador"*

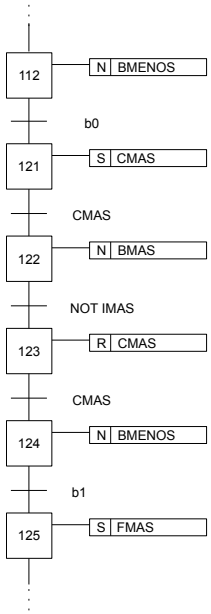

Figura 5.36: Proceso trasvasador de la versión estructurada

La etapa 125 tiene asociada la acción *"FMAS"* que se corresponde con el avance del centrador.

Proceso Medidor de la versión OO

En la figura 5.37 se muestra otro trozo del código correspondiente al método *"EjecucionAutomatica"* de la clase *"ControlProceso"*. Esta parte del código se encarga de la medicción de los rodamientos.

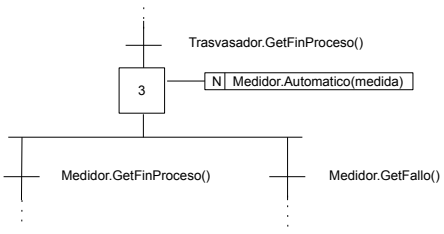

Figura 5.37: Método de ejecucion automatica de la clase *"ControlProceso"* para el medidor

La acción de los rodamientos se corresponde con la acción llamada al método *"Automatico"* de la clase *"Medidor"* que es la que realiza la tarea (ver figura 5.38).

Las acciones de la etapa 0 se corresponden con "*Fallo*","*CondicionesIniciales*", *"FinProceso"* y *"TamanioValido"* son atributos boolean de la clase. La acción *"CilMono_Centrador.Accionado"* es una llamada al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

La acción *"CilBi_Elevador.Accionado"* de la etapa 1 es una llamada al método *"Accionado"* de la clase *"ActuadorBiestable"*.

La transición *"sDigital_Abajo.Accionado"* de la etapa 5 devuelve el valor de la clase *"SensorDigital"* al igual que la transición *"sDigital_Arriba.Accionado"* de las etapas 2 y 3.

La acción *"Medicion"* de la etapa 2 contiene la instrucción en código ST++: *"TamanioValido:=(referencia=sAnalog_Palpador.Valor);"*

Proceso Medidor de la versión estructurada

En la figura 5.39 se muestra un trozo del GRAFCET que controla la medición de los rodamientos.

La etapa 126 tiene asociada la acción *"DMAS"* que se corresponde con la subida del elevador.

La transición *"d1"* de la etapa 126 es el final de carrera encargado de detectar la posición arriba del elevador.

La etapa 145 tiene asociada la acción *"AlturaCorrecta"* que es modifica el valor de la variable booleana del mismo nombre.

La transición *"SelectorTipo"* de la etapa 145 contiene una variable booleana del mismo nombre mapeada a la entrada I2.0.

La etapa 147 tiene asociada la acción *"DMENOS"* que se corresponde con la bajada del elevador.

La transición *"d0"* de la etapa 1476 es el final de carrera encargado de detectar la posición abajo del elevador.

La etapa 126 tiene asociada la acción *"FMAS"* que se corresponde el avance del centrador.

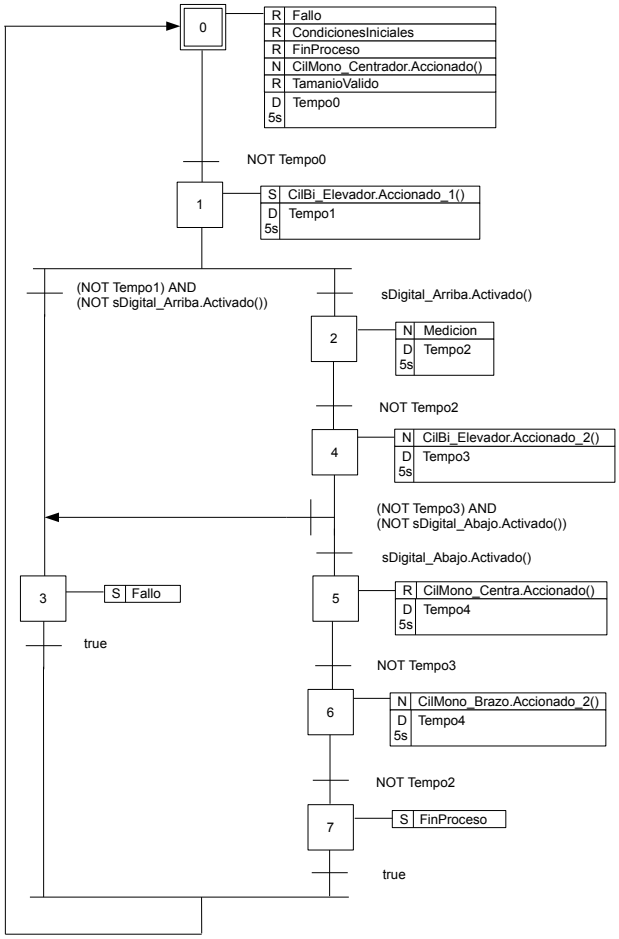

Figura 5.38: Método de ejecucion automatica de la clase *"Medidor"*

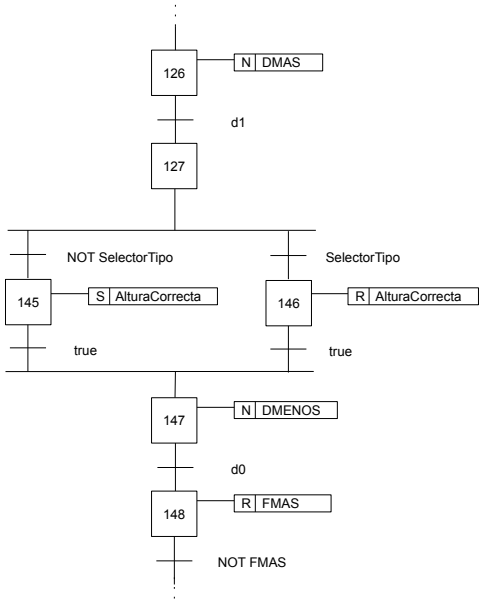

Figura 5.39: Proceso de medidor de la versión estructurada

5.6.4. Resultados de tiempos del módulo TCLOCK

Los tiempos de ejecución de la estación de montaje de rodamientos está medida tras la realiación de 20 mediciones, 10 en las que se aceptaba la pieza y otras 10 en las que la pieza era rechazada. En las tablas se muestran, por una parte el número de ciclos de SCAN que se han necesitado para completar un proceso completo, y por otro el tiempo de ejecución de cada proceso medido en milisegundos.

En la tabla 5.40 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma OO.

En la tabla 5.41 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma estructurado.

A continuación, se muestra de forma gráfica, usando los diagramas de caja, las cuartiles de los tiempos de ejecución de los distintos procesos que conforman la estación de montaje de rodamientos, así como sus medias, desviación típica y covarianza de dichos tiempos.

En la figura 5.42 se puede observar el diagrama de cajas de los ciclos ejecutados por la estación para llevar a cabo el montaje de bases.

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Traslado	Medición	Expulsión	Manipulación
6909	2.17	25	4283	6072	4582	0	5828
6766	2.17	26	4130	5926	4589	0	5760
6809	2.16	25	4130	6105	4584	0	5739
6758	2.17	24	4216	6013	4581	0	5736
6792	2.17	26	4199	5988	4576	0	5744
6771	2.18	25	4203	6000	4578	0	5731
6764	2.16	26	4148	5944	4596	0	5735
6843	2.17	26	4161	5960	4604	0	5732
6846	2.17	23	4258	6051	4595	0	5722
6816	2.17	27	4179	5977	4602	0	5732
4714	2.17	24	4086	5880	4606	1007	0
4703	2.17	27	4250	6047	4599	1007	0
4619	2.18	24	4113	5908	4602	1006	0
4683	2.18	26	4269	6067	4600	1006	0
4651	2.18	26	4174	5971	4598	1007	0
4651	2.17	24	4160	5955	4600	1007	0
4652	2.17	27	4327	6123	4600	1007	0
4622	2.17	26	4203	6007	4612	1008	0
4653	2.17	26	4197	5994	4601	1006	0
4654	2.18	28	4195	5994	4601	1007	0

Figura 5.40: Tiempos de ejecución de la estación 2 en su versión OO

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Traslado	Medición	Expulsión	Manipulación
6946	2.10	15	2298	5766	4474	0	5761
7044	2.10	14	2437	5909	4464	0	5730
7012	2.10	14	2429	5904	4472	0	5719
7004	2.11	14	2426	5902	4475	0	5721
6961	2.10	16	2423	5893	4469	0	5715
7010	2.10	14	2415	5886	4477	0	5711
6933	2.11	14	2385	5855	4481	0	5720
6953	2.10	16	2416	5882	4474	0	5717
6931	2.10	16	2419	5886	4472	0	5707
6986	2.10	14	2410	5882	4481	0	5711
4834	2.10	14	2396	5866	4477	1004	0
4813	2.10	16	2415	5888	4480	1006	0
4846	2.10	15	2392	5864	4484	1005	0
4856	2.10	14	2414	5885	4474	1004	0
4851	2.10	14	2410	5891	4482	1004	0
4844	2.10	15	2400	5877	4479	1005	0
4824	2.10	14	2401	5873	4479	1005	0
4846	2.10	14	2398	5874	4481	1004	0
4837	2.10	14	2400	5878	4481	1004	0
4821	2.10	15	2398	5866	4474	1005	0

Figura 5.41: Tiempos de ejecución de la estación 2 en su versión estructurada

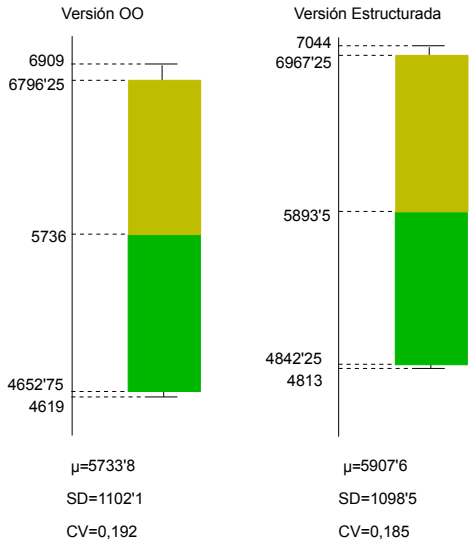

Figura 5.42: Diagrama de cajas del proceso *"número de ciclos"* de la estación 2

En la figura 5.43 se muestra el diagrama de cajas del proceso que permite inicializar los objetos que conforman la estación dejando la posición inicial todos los elementos de la misma.

En la figura 5.44 se muestra el diagrama de cajas del proceso que permite sacar un rodamiento del lugar donde se almacenan.

En la figura 5.45 se muestra el diagrama de cajas del proceso que traslada el rodamiento desde la zona de alimentación hasta al lugar donde se realiza la siguiente operación.

En la figura 5.46 se muestra el diagrama de cajas del proceso que identifia si el rodamiento posee la altura que le usuario ha seleccionado desde el panel de mando.

En la figura 5.47 se muestra el diagrama de cajas del proceso que saca de la zona de medición los rodamientos que no poseen la altura correcta.

En la figura 5.48 se muestra el diagrama de cajas del proceso que coloca el rodamiento en el interior de la base.

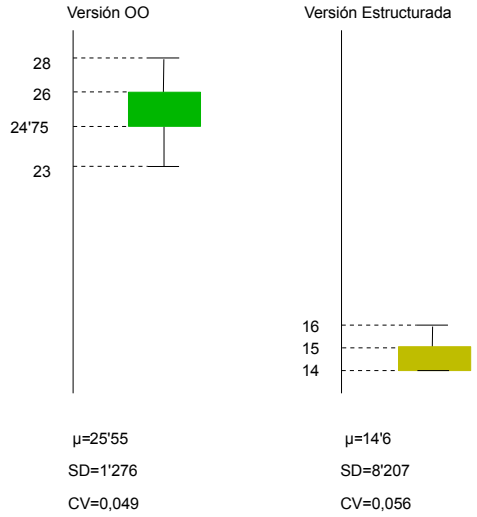

Figura 5.43: Diagrama de cajas del proceso *"Inicialización"* de la estación 2

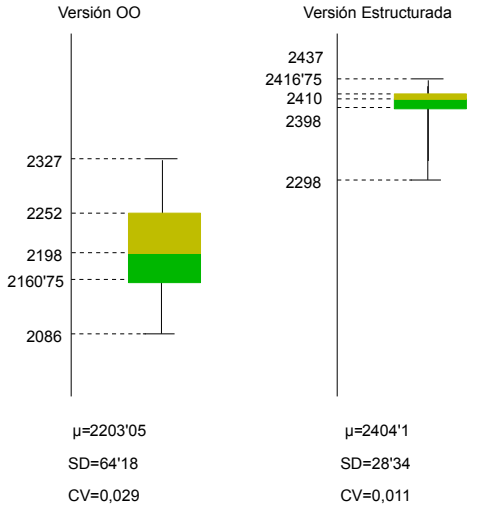

Figura 5.44: Diagrama de cajas del proceso *"Alimentación"* de la estación 2

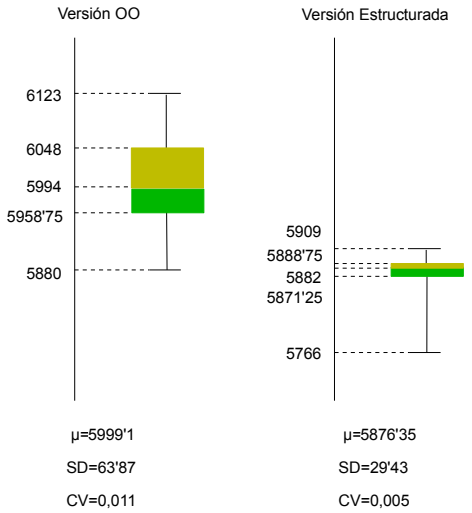

Figura 5.45: Diagrama de cajas del proceso *"Traslado"* de la estación 2

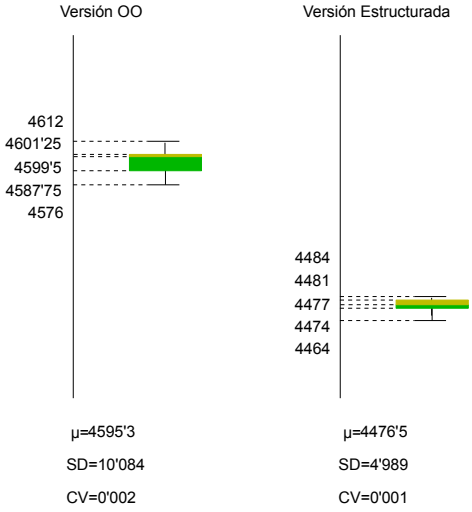

Figura 5.46: Diagrama de cajas del proceso *"Medicion"* de la estación 2

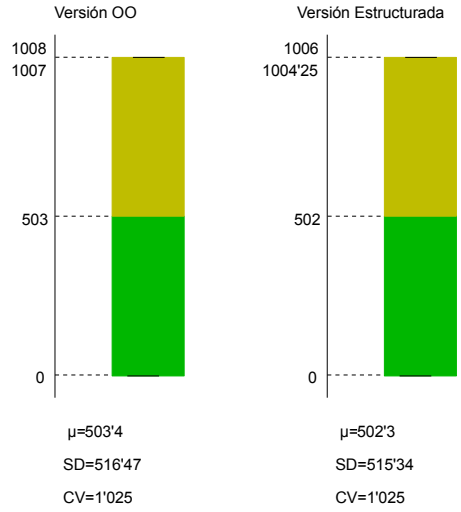

Figura 5.47: Diagrama de cajas del proceso *"Expulsion"* de la estación 2

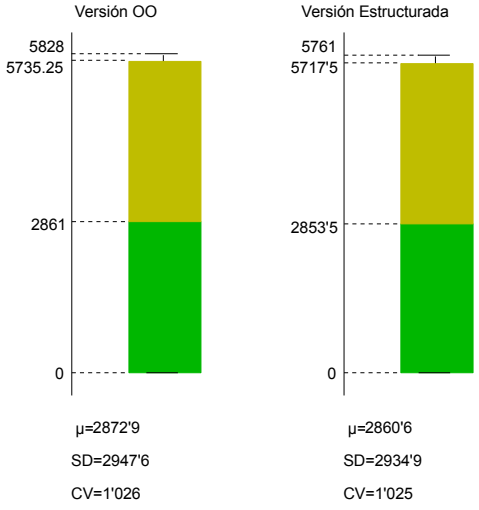

Figura 5.48: Diagrama de cajas del proceso *"Manipulacion"* de la estación 2

5.7. Estación 3 - Estación de prensado

La tercera de las estaciones permite realizar el proceso necesario para prensar el conjunto de base y rodamiento.

Esta estación posibilita sincronizar los procesos de separación de la base-rodamiento del palet e implementar los mecanismos de seguridad (bajada de la mampara) necesarios para realizar el prensado.

5.7.1. Etapas del proceso

La realización del prensado exige llevar a cabo una serie de operaciones según se describe a continuación.

Inserción / extracción del conjunto

La primera de las operaciones a realizar consiste en insertar la base con el rodamiento en su interior desde el palet retenido en el *"transfer"*, hasta un punto de descarga en el interior de la estación. Esta manipulación y el proceso inverso de descarga tras haber finalizado el prensado, se realiza mediante un actuador de giro neumático. Dicho actuador incorpora un brazo con un grupo de cuatro ventosas en su extremo, encargadas de sujetar la pieza a través del vacío generado en su interior mediante un eyector. Con el fin de mantener la base siempre en posición horizontal durante todo el movimiento de giro, este brazo incorpora en su interior un mecanismo basado en piñón y correa dentada, similar al utilizado en la estación de inserción del rodamiento.

Alimentación de la prensa

La base a introducir en la prensa es depositada en una plataforma que dispone de dos cilindros neumáticos de doble efecto. El primero realiza el traslado desde el punto de carga / descarga al de prensado y el segundo el proceso inverso una vez que la prensa ha finalizado su operación.

Prensado del cojinete

En el momento en que la base con el rodamiento en su interior ha sido situado bajo el cilindro hidráulico, en primer lugar desciende una mampara de protección

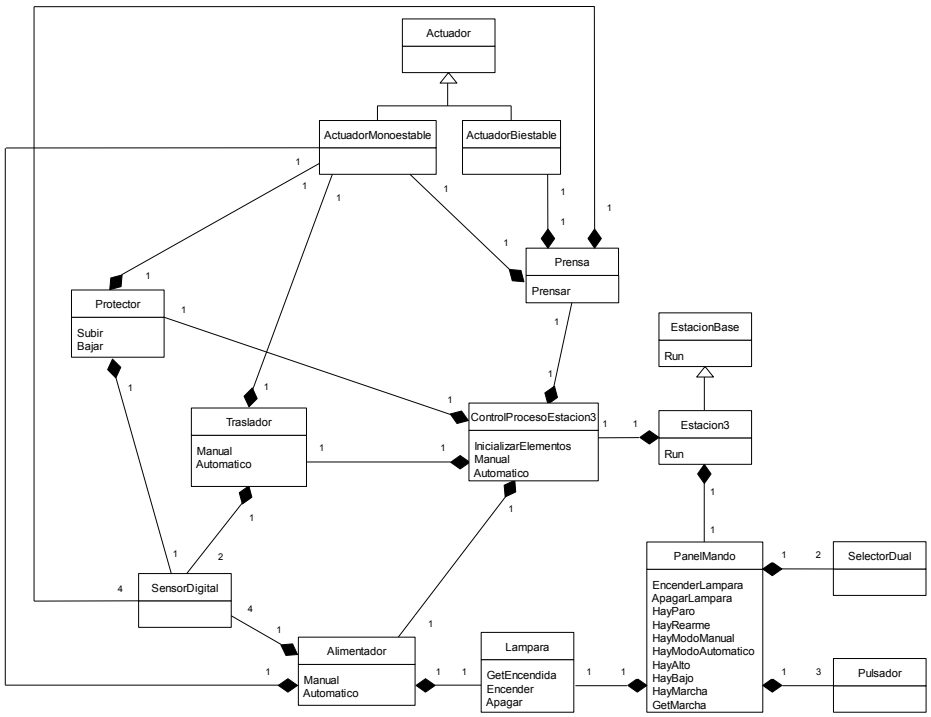

Figura 5.49: Diagrama de clases de la estación 3

accionada mediante un cilindro neumático. De esta forma se logra una protección del usuario frente a eventuales riesgos de accidente, a la vez que se muestra un mecanismo de seguridad ampliamente utilizado en este tipo de aplicaciones.

A continuación, mediante una válvula distribuidora hidráulica 4/3, el cilindro de prensado baja realizando una fuerza regulable a través de la válvula limitadora de presión que incorpora el grupo hidráulico. Una vez finalizado, este cilindro retorna a su posición original, se eleva la protección y se empuja el conjunto hasta el punto de descarga.

5.7.2. Diagrama de clases

La versión OO de la estación 3 se compone de 13 clases y su diagrama de clases se muestra en la figura 5.49.

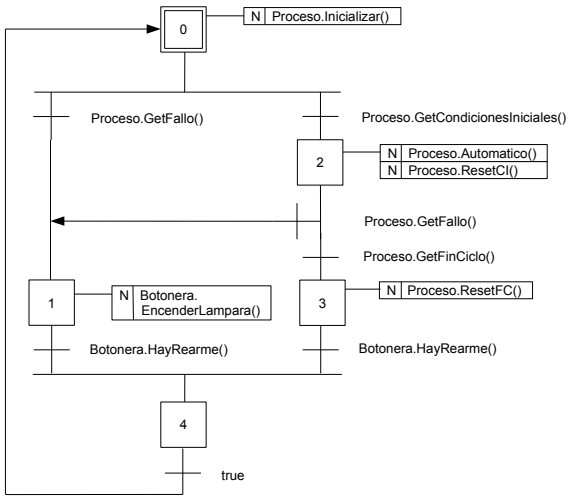

Figura 5.50: Método *"Run"* de la clase estacion3

5.7.3. Programación OO vs programación estructurada

En esta subsección se mostrarán los procesos de alimentación de la prensa (proceso *"traslador"*) y bajada del protector para el prensado (proceso *"protector"*).

Método *"Run"* de la clase Estacion3

En la figura 5.50 se muestra las ramas concernientes a la ejecución automática de la estación 3.

Proceso traslador de la versión OO

La etapa 2 del método *"Run"* de la versión OO de la estación 3 (ver figura 5.50) tiene asociada una llamada al método *"Automatico"* de la clase *"ControlProcesoEstacion3"*. En la figura 5.51 se muestra el trozo de código del método *"Automatico"* correspondiente al proceso de alimentación de la prensa.

La etapa 2 tiene asociada invocación al método *"Automatico"* de la clase *"Traslador"* que es quien realiza la tarea de alimentar la prensa (ver figura 5.52).

Las acciones de la etapa 0 se corresponden con:

- "*Fallo", "CondicionesIniciales" y "FinProceso"* son atributos boolean de la clase.

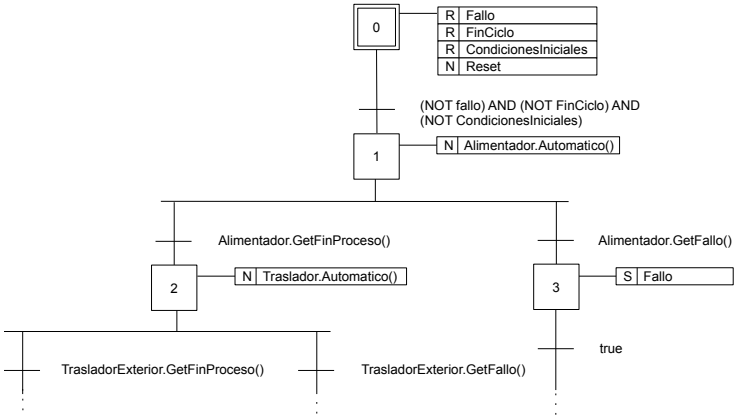

Figura 5.51: Método *"Automatico"* de ControlProcesoEstacion3

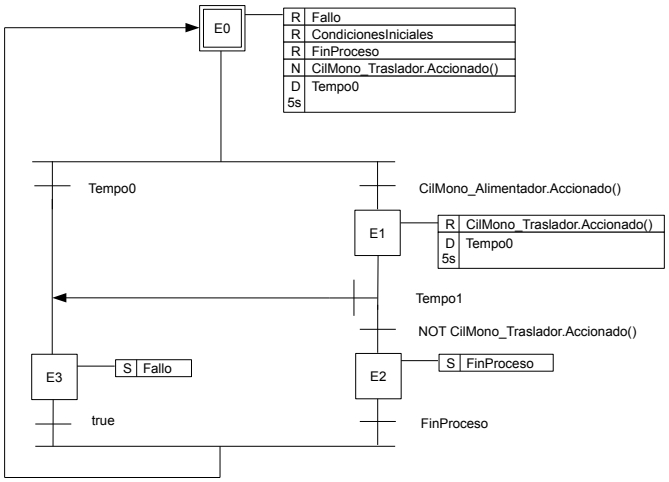

Figura 5.52: Método de ejecucion automatica de la clase *"Traslador"*

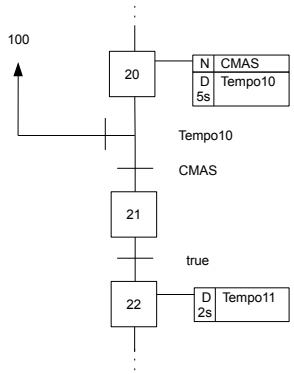

Figura 5.53: Proceso de alimentación de la prensa de la versión estructurada

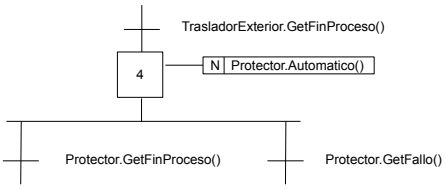

Figura 5.54: Método *"Automatico"* de la clase ControlProceso para la bajada del protector

- *"CilMono_ Traslador.Accionado"* es una llamada al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

Proceso traslador de la versión estructurada

En la figura 5.53 se muestra un trozo del GRAFCET que controla el proceso de alimentación de la prensa.

La etapa 20 tiene asociada la acción *"CMAS"* que se corresponde con la señal de avance del cilindro traslador exterior.

Proceso protector de la versión OO

En la figura 5.54 se muestra otro trozo del código correspondiente al método *"Automatico"* de la clase *"ControlProceso"*. Esta parte del código se encarga de descender la mampara protectora.

La acción de bajar la mampara protectora se corresponde con la llamada al método

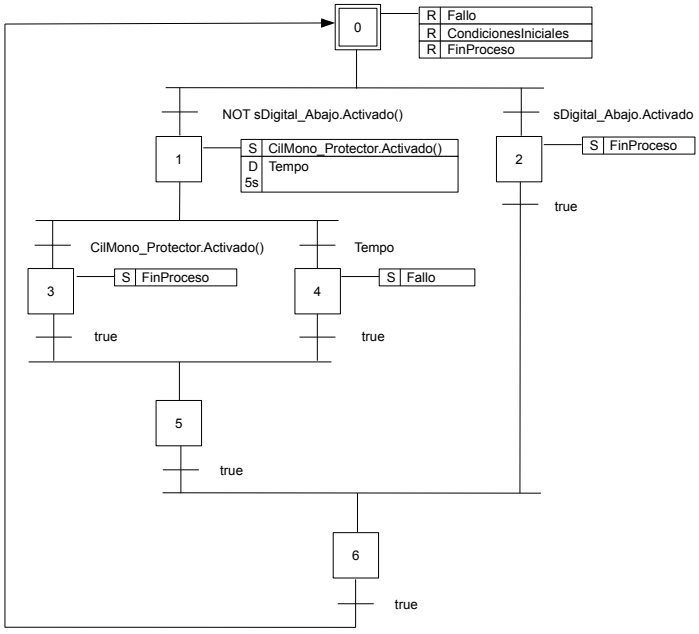

Figura 5.55: Método *"Automatico"* de la clase *"Protector"*

"Automatico" de la clase *"Protector"* que es la que realiza la tarea (ver figura 5.55).

Las acciones de la etapa 0 se corresponden con "*Fallo","CondicionesIniciales" y "FinProceso"* son atributos boolean de la clase.

La transición *"CilMono_ Alimentador.Accionado"* de la etapa 1 es una llamada al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

La transición *"sDigital_ Abajo.Accionado"* de la etapa 2 devuelve el valor de la clase *"SensorDigital"*.

Proceso Protector de la versión estructurada

En la figura 5.56 se muestra un trozo del GRAFCET que controla la bajada de la mampara de protección.

La etapa 31 tiene asociada la acción *"DMAS"* que se corresponde con la bajada de la mampara protectora de la prensa.

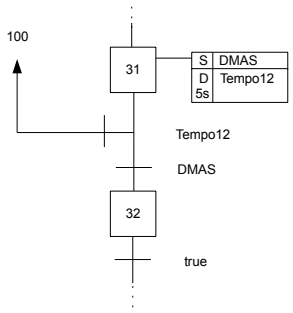

Figura 5.56: Proceso de bajada del protector de la versión estructurada

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Manipulación	Traslado	Bajar protector	Prensado	Subir protector	Traslado	Manipulación
7603	2'11	1266	4166	1161	404	3013	1010	1350	3309
7512	2'11	1272	2899	1167	404	3014	1011	1349	3303
7505	2'11	1203	3938	1169	407	3013	1010	1352	3306
7442	2'12	1228	3932	1175	405	3012	1008	1359	3313
7571	2'11	1207	3909	1174	406	3012	1008	1356	3318
7456	2'11	1304	3914	1167	404	3012	1007	1349	3315
7452	2'11	1221	3990	1168	407	3014	1012	1354	3318
7517	2'11	1234	4028	1170	405	3011	1010	1349	3323
7557	2'10	1301	3903	1170	404	3011	1009	1346	3313
7519	2'11	1202	3944	1171	408	3010	1009	1346	3311

Figura 5.57: Tiempos de ejecución de la estación 3 en su versión OO

5.7.4. Resultados de tiempos del módulo TCLOCK

Los tiempos de ejecución de la estación de prensado está medida tras la realiación de 10 mediciones. En las tablas se muestran, por una parte el número de ciclos de SCAN que se han necesitado para completar un proceso completo, y por otro el tiempo de ejecución de cada proceso medido en milisegundos.

En la tabla 5.57 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma OO.

En la tabla 5.58 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma estructurado.

A continuación, se muestra de forma gráfica, usando los diagramas de caja, las cuartiles de los tiempos de ejecución de los distintos procesos que conforman la

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Manipulación	Traslado	Bajar protector	Prensado	Subir protector	Traslado	Manipulación
6951	2'09	2094	2763	1345	402	3006	2005	1164	3344
6984	2'09	2201	2759	1339	401	3008	2004	1154	3365
7023	2'09	2149	2910	1351	403	3007	2004	1163	3350
7144	2'09	2178	2858	1353	402	3007	2005	1171	3350
7188	2'09	2087	2912	1357	402	3006	2005	1162	3352
6978	2'09	2163	2865	1352	403	3007	2007	1165	3350
6988	2'08	2187	2865	1355	403	3008	2004	1161	3357
7081	2'08	2112	2945	1360	404	3006	2005	1155	3347
7034	2'09	2138	2971	1354	401	3008	2005	1156	3367
7189	2'09	2302	3058	1359	401	3010	2004	1160	3412

Figura 5.58: Tiempos de ejecución de la estación 3 en su versión estructurada

estación de prensado, así como sus medias, desviación típica y covarianza de dichos tiempos.

En la figura 5.59 se puede observar el diagrama de cajas de los ciclos ejecutados por la estación para llevar a cabo el prensado.

En la figura 5.60 se muestra el diagrama de cajas del proceso que permite inicializar los objetos que conforman la estación, dejando en la posición inicial todos los elementos de la misma.

En la figura 5.61 se muestra el diagrama de cajas del proceso que traslada la base con el rodamiento en su interior desde el palet hasta el interior de la estación.

En la figura 5.62 se muestra el diagrama de cajas del proceso que permite trasladar el conjunto base-rodamiento hasta la zona de prensado.

En la figura 5.63 se muestra el diagrama de cajas del proceso que baja la mampara de protección de la prensa.

En la figura 5.64 se muestra el diagrama de cajas del proceso querealiza el prensado del conjunto base-cojinete.

En la figura 5.65 se muestra el diagrama de cajas del proceso que sube la mampara de protección de la prensa.

En la figura 5.66 se muestra el diagrama de cajas del proceso que permite trasladar el conjunto base-rodamiento fuera de la zona de prensado.

En la figura 5.67 se muestra el diagrama de cajas del proceso que traslada el conjunto base-rodamiento prensado hasta el palet del *"transfer"*.

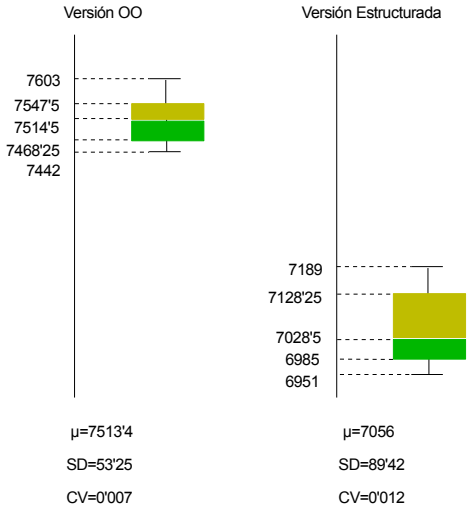

Figura 5.59: Diagrama de cajas del proceso *"número de ciclos"* de la estación 3

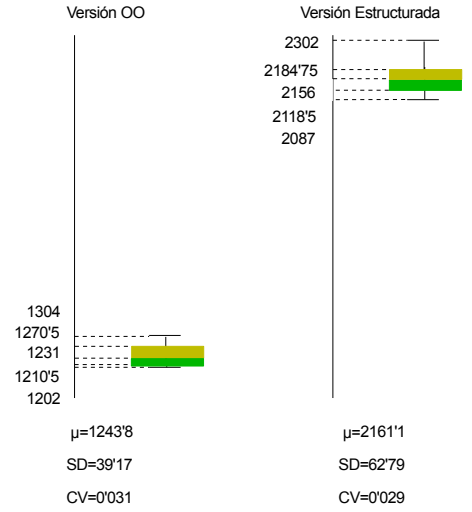

Figura 5.60: Diagrama de cajas del proceso *"Inicialización"* de la estación 3

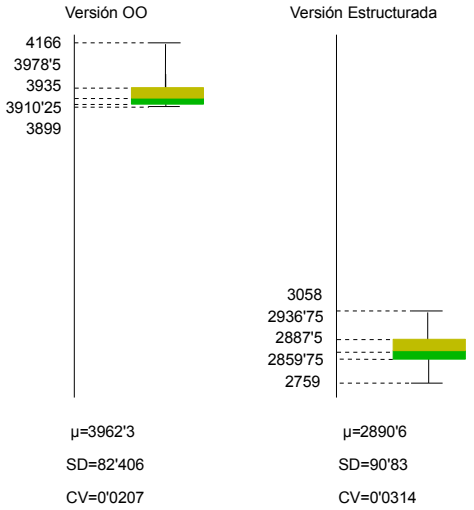

Figura 5.61: Diagrama de cajas del proceso *"Manipulación1"* de la estación 3

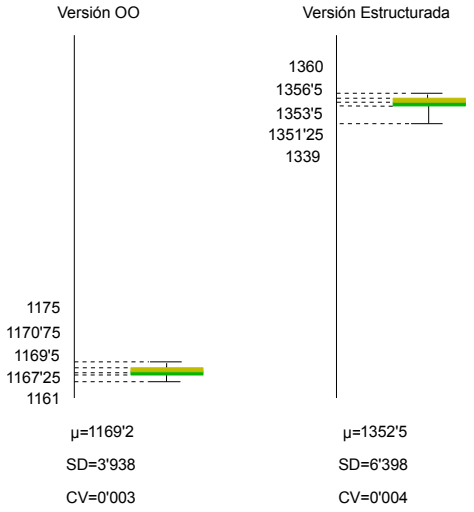

Figura 5.62: Diagrama de cajas del proceso *"Traslado1"* de la estación 3

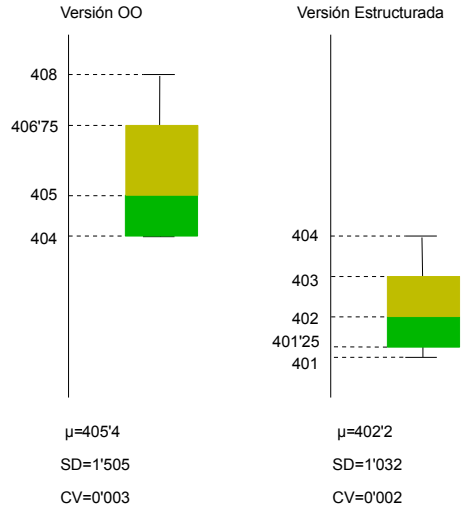

Figura 5.63: Diagrama de cajas del proceso *"BajarProtector"* de la estación 3

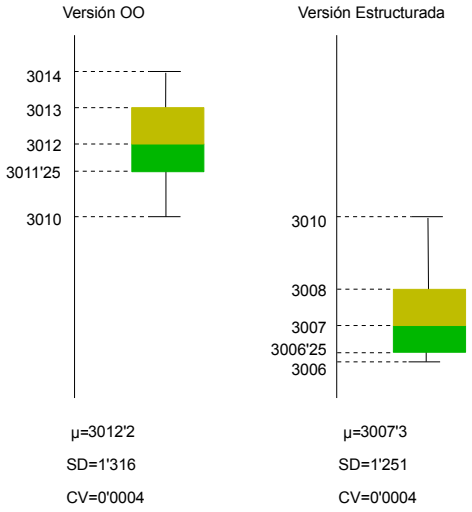

Figura 5.64: Diagrama de cajas del proceso *"Prensado"* de la estación 3

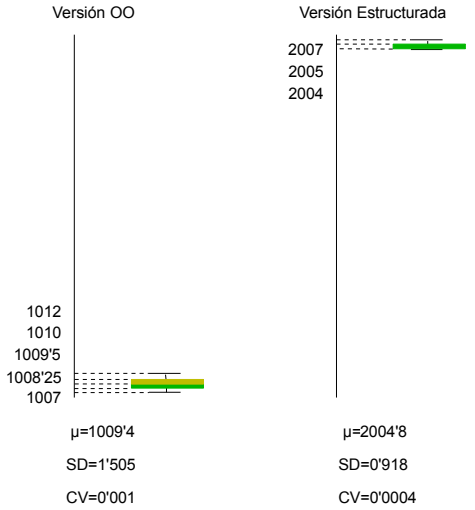

Figura 5.65: Diagrama de cajas del proceso *"SubirProtector"* de la estación 3

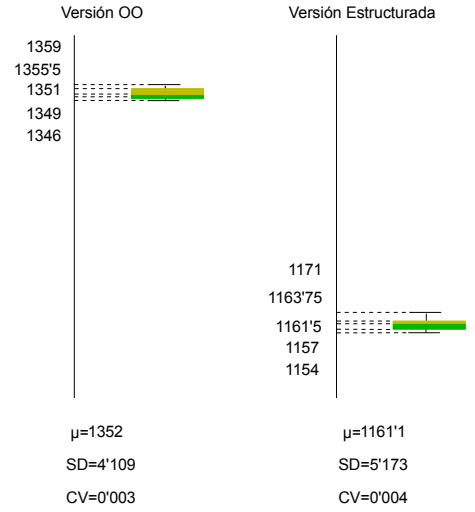

Figura 5.66: Diagrama de cajas del proceso *"Traslado2"* de la estación 3

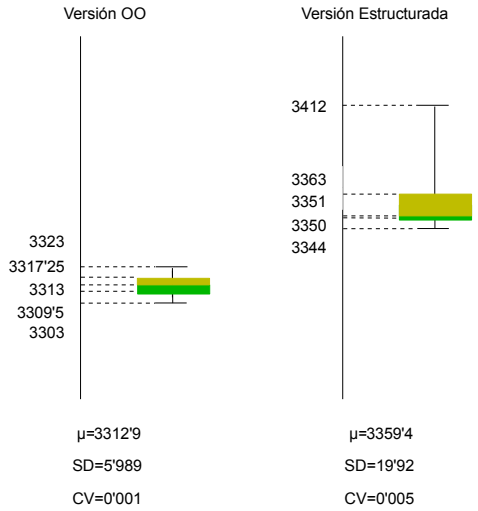

Figura 5.67: Diagrama de cajas del proceso *"Manipulación2"* de la estación 3

5.8. Estación 4 - Estación de inserción de ejes

La cuarta de las estaciones permite realizar el proceso necesario para el montaje de un eje, situándolo dentro del rodamiento insertado con anterioridad.

Con esta estación se aumenta la flexibilidad desde el punto de vista de la variedad de conjuntos que es posible fabricar, ya que permite la colocación de ejes con dos materiales diferentes; aluminio y nylon.

La existencia de dichas variantes exige que a las operaciones tradicionales de alimentación, manipulación e inserción, se les sumen las de comprobación del tipo de material y extracción de los ejes, en el caso de que no coincida con el deseado.

5.8.1. Etapas del proceso

El número adicional de operaciones a realizar supone un aumento en la complejidad de la estación, de forma que su estructura cambia notablemente respecto a las anteriores. En este caso, los componentes se encuentran distribuidos en torno a un plato divisor con ocho estaciones (figura 5.68).

A lo largo de dichas subestaciones se van realizando las sucesivas operaciones. A continuación se describe la forma en que se llevan a cabo:

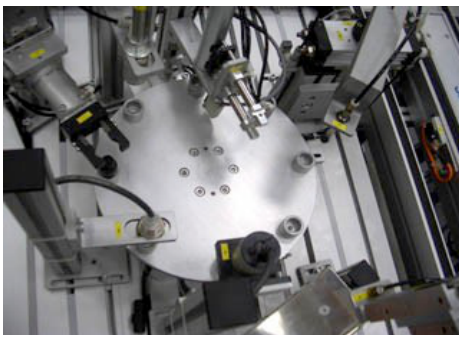

Figura 5.68: Plato divisor (vista superior)

Plato divisor

sistema de movimiento giratorio alternativo, de forma que a cada movimiento de giro se produce un avance de un número de grados correspondiente a la división de la circunferencia entre el número de posiciones definidas.

Para conseguir este efecto, se incluye un cilindro neumático empujador con movimiento oscilante mediante el cual se consigue el avance del ángulo deseado. A su vez, se dispone de otros dos cilindros de tope funcionando alternativamente, uno de ellos móvil que sujeta el plato mientras se produce el giro, y otro fijo que lo bloquea cuando el movimiento ha terminado, de forma que el plato permanece sujeto firmemente y el cilindro empujador puede retornar a su posición inicial en espera de un nuevo ciclo.

Alimentación de ejes

Los ejes almacenados en un alimentador tipo petaca, son depositados en la primera de las estaciones del plato mediante un sistema de alimentación paso a paso, realizado mediante dos cilindros neumáticos. Estos cilindros se encuentran situadas en todo momento en posiciones contrapuestas, de manera que cuando el de la parte inferior libera el último eje del cargador, el de la parte superior sujeta al resto.

Para evitar posibles atascos en la alimentación, la subestación ha sido modificada colocándole un detector óptico en lógica negativa, situado sobre la primera de las posiciones del plato.

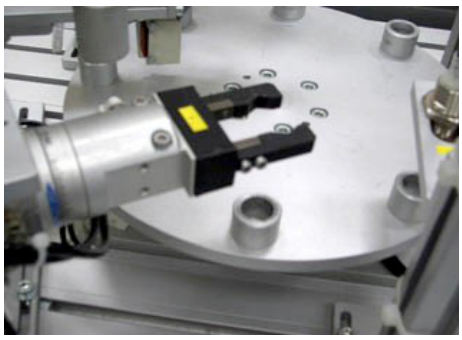

Figura 5.69: Giro de ejes

Medición de la altura del eje

El eje a introducir no presenta una forma simétrica, por lo cuál es necesario que sea montado sobre el conjunto en una posición determinada. Con el objeto de determinar si el eje proveniente de la estación de alimentación ha sido introducido correctamente se realiza una medida de su altura. Para ello, se utiliza un cilindro neumático que al disponer de un detector magnético permite discernir si en el avance contacta con el eje o si llega al final de su carrera debido a que no ha sido colocado en su posición.

Giro de ejes

En caso de que en la segunda de las subestaciones del plato divisor, se determine que el eje ha sido depositado en posición invertida, el manipulador de la figura 5.69 se encargará de darle la vuelta. Ésto se consigue sujetando el eje con una pinza de dos dedos, elevándolo a continuación mediante un cilindro de vástagos paralelos, para finalmente voltearlo utilizando un actuador de giro de 180^{o} y volver a depositarlo en el alojamiento en la posición adecuada.

Detección material eje

La siguiente de las operaciones llevadas a cabo en esta estación, se realiza en dos subestaciones consecutivas del plato divisor. El propósito de la misma consiste en determinar el material con que está construido el eje, siendo necesario distinguir entre aluminio y nylon. Para ello, se dispone en las subestaciones tercera y cuarta de sendos detectores inductivo y capacitivo mediante los cuales es posible realizar la diferenciación entre estos dos tipos de materiales.

Evacuación eje incorrecto

Como se ha mencionado anteriormente, esta estación contempla la posibilidad de realizar en un nivel superior de gestión de la célula, una elección del material del eje montado en los diferentes conjuntos a construir. Por ello resulta necesario disponer de algún elemento que deseche el eje en caso de que no corresponda con el tipo indicado. Dicha operación se realiza en la quinta subestación del plato divisor, para lo cual se dispone de un manipulador que al recibir la orden correspondiente y se encarga de extraer el eje del plato. Se trata de un manipulador de dos ejes que dispone como elemento terminal de una ventosa con la cual sujeta el eje por su parte superior. Cada uno de los ejes está compuesto por un cilindro neumático de vástagos paralelos mediante los cuales se realizan los movimientos de elevación del eje y conducción hasta una rampa de evacuación. La sujeción del eje se lleva a cabo mediante la técnica de vacío, utilizando para ello una ventosa, un eyector para conseguir el vacío necesario y un vacuostato que suministra al PLC una señal indicando que la sujeción es la adecuada.

Inserción eje en conjunto

La inserción del eje, llevada a cabo en la última de las subestaciones del plato divisor, se realiza mediante un manipulador de tipo rotolineal. Éste permite realizar las operaciones de recogida del eje, desplazamiento hasta el punto de descarga e inserción con un único componente. Dicho cilindro dispone de la posibilidad de comandar tanto el movimiento de avance y retroceso del vástago, como el giro a izquierdas y derechas del mismo de forma independiente.

Al igual que en las anteriores manipulaciones realizadas utilizando la técnica de vacío, se incluyen para ello un eyector y un vacuostato.

5.8.2. Diagrama de clases

La versión OO de la estación 4 se compone de 16 clases y su diagrama de clases se muestra en la figura 5.70.

Por no volver demasiado complicado el diagrama no se ha puesto la clase padre de los clases *"ActuadorMonoestable"* y *"ActuadorBiestable"*. Al igual que ocurre con las 3 estaciones anteriores, estas clases heredan de la clase *"Actuador"*.

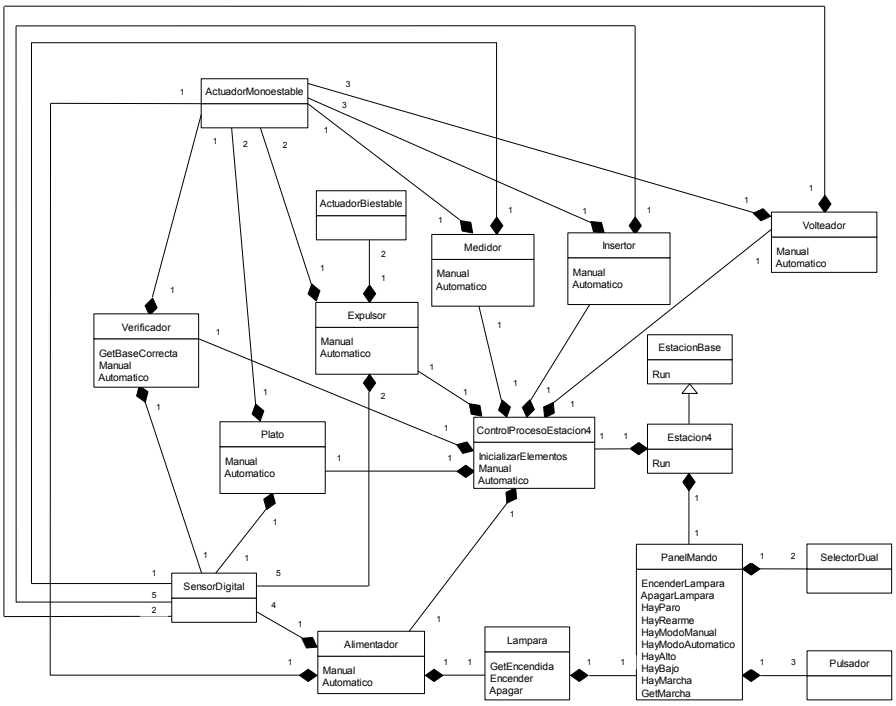

Figura 5.70: Diagrama de clases de la estación 4

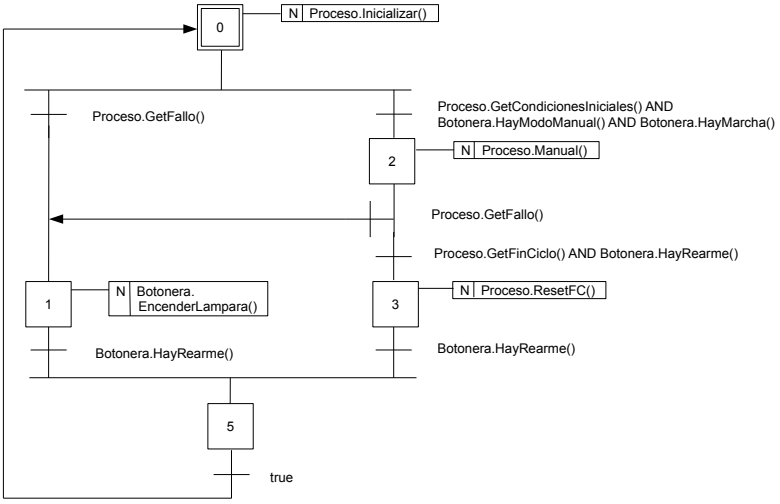

Figura 5.71: Método *"run"* de la clase estacion4

5.8.3. Programación OO vs programación estructurada

En esta subsección se mostrarán los procesos de movimiento giratorio alternativo del plato (proceso *"Plato"*) y el proceso encargado de dar la vuelta al eje en caso de detectarse que este ha sido introducido al reves (proceso *"Volteador"*).

Método *"Run"* de la clase Estacion4

En la figura 5.71 se muestra las ramas concernientes a la ejecución manual de la estación 4.

Proceso plato de la versión OO

La etapa 2 del método *"Run"* de la versión OO de la estación 4 (ver figura 5.71) tiene una llamada al método *"Manual"* de la clase *"ControlProcesoEstacion4"* encargado de realizar la tarea de girar el plato. En la figura 5.72 se muestra el trozo de código del método *"Manual"* correspondiente al proceso del movimiento del plato giratorio.

En la figura 5.72 sólo se muestra una parte del proceso de girado ya que después de cada acción se ejecuta una etapa que tiene asociada una invocación al método *"Girar"* de la clase *"Plato"*. En la figura 5.73 se puede ver el método *"Girar"* que gobierna el movimiento de dicho plato.

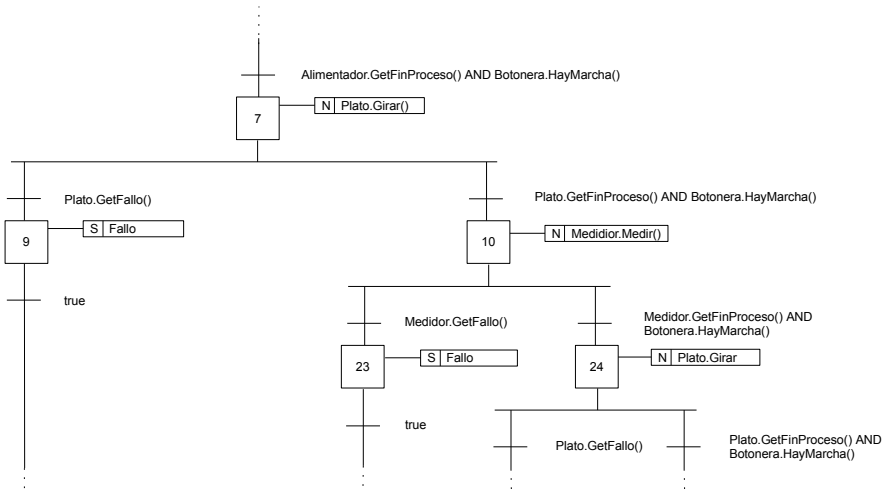

Figura 5.72: Método *"Manual"* de la clase *"ControlProcesoEstacion4"*

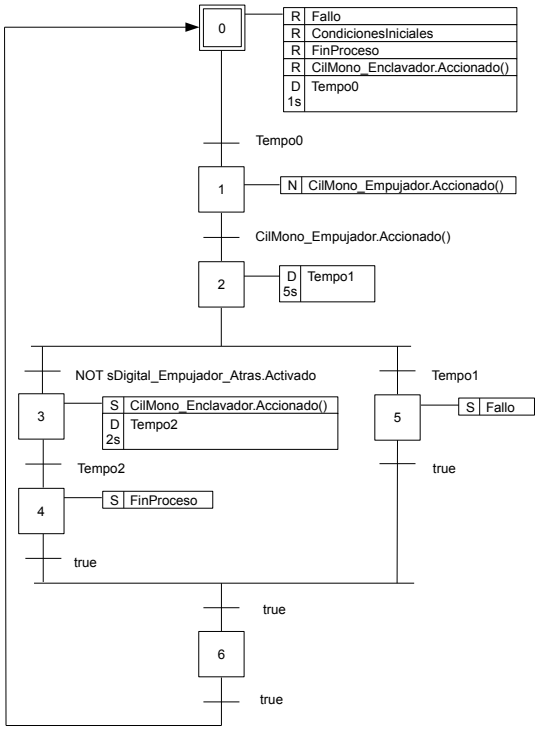

Figura 5.73: Método de ejecucion *"Manual"* de la clase *"Plato"*

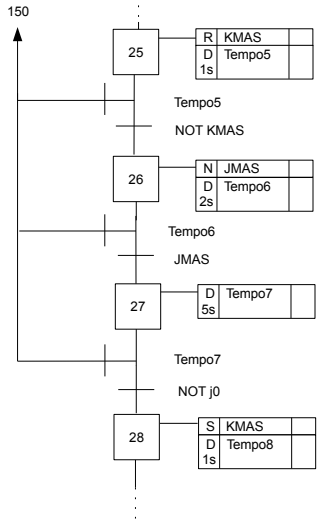

Figura 5.74: Proceso de giro del plato de la versión estructurada

Las acciones de la etapa 0 se corresponden con:

- "*Fallo", "CondicionesIniciales" y "FinProceso"* son atributos boolean de la clase.
- *"CilMono_ Enclavador.Accionado"* es una llamada al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

La etapa 1 tiene asociado una acción que se corresponden con una llamada al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

Proceso plato de la versión estructurada

En la figura 5.74 se muestra un trozo del GRAFCET que controla el proceso del movimiento del plato para una acción ya que al igual que ocurre con la versión OO, tras cada acción de la estación se produce un movimiento rotatorio del plato.

La etapa 25 tiene asociada la acción *"KMAS"* que se corresponde con la señal de avance de enclavamientos del plato.

La etapa 26 tiene asociada la acción *"JMAS"* que se corresponde con la señal de avance del empujador del plato.

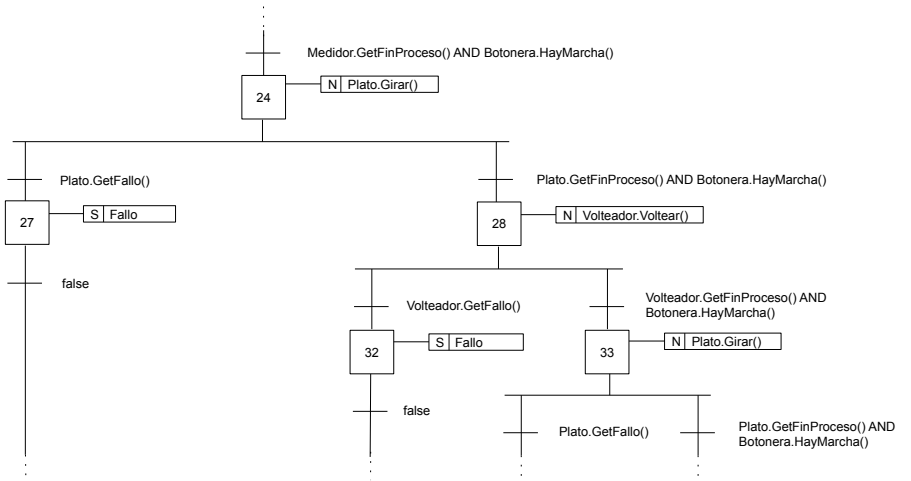

Figura 5.75: Método *"Automatico"* de la clase *"ControlProceso"* para la bajada del protector

La etapa 27 tiene asociada la acción *"j0"* que se corresponde con la señal de detección de empujador del plato atrás.

La etapa 150 tiene asociada la transición *"DEFECTO"* que se corresponde con el piloto luminoso por defecto de la estación.

Proceso volteador de la versión OO

En la figura 5.75 se muestra otro trozo del código correspondiente al método *"Automatico"* de la clase *"ControlProceso"*. Esta parte del código se encarga de voltear el eje en caso de detectarse que este está en posición invertida.

La acción de girar el eje se corresponde con la llamada al método *"Automatico"* de la clase *"Volteador"* que es la que realiza la tarea (ver figura 5.76).

Las acciones de la etapa 0 se corresponden con "*Fallo*","*CondicionesIniciales*" y *"FinProceso"* son atributos boolean de la clase.

La transición *"CilMono_ Bajar.Accionado"*, *"CilMono_ Pinza.Accionado*" y *"CilMono_ Giro.Accionado"* son llamadas al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

La transición *"sDigital_ Arriba.Accionado"* devuelve el valor de la clase *"SensorDigital"*.

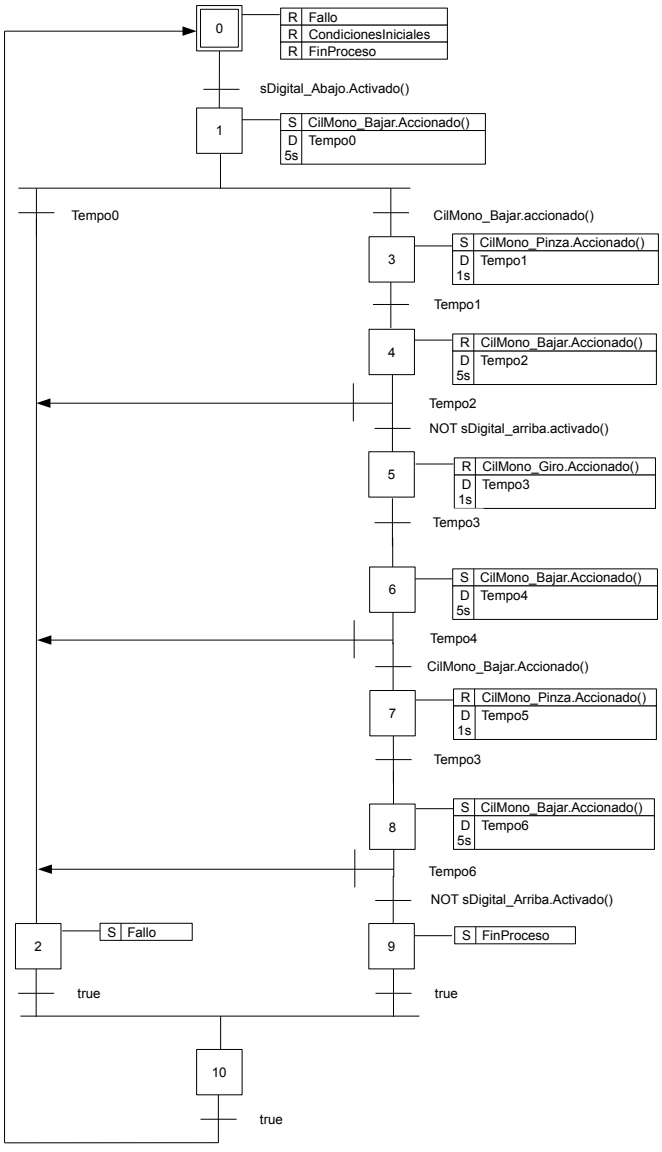

Figura 5.76: Método *"Automatico"* de la clase *"Volteador"*

Proceso voltear de la versión estructurada

En la figura 5.77 se muestra un trozo del GRAFCET que controla el giro de los ejes en caso de que se hayan introducido al revés.

Las etapas 44, 47, 49 y 51 tiene asociada la acción *"GMAS"* que se corresponde con la bajada del cilindro de volteo.

Las etapas 46 y 50 tiene asociada la acción *"IMAS"* que se corresponde con el cierre de la pinza del manipulador de volteo.

La etapa 48 tiene asociada la acción *"HMAS"* que se corresponde con el giro del cilindro de volteo.

5.8.4. Resultados de tiempos del módulo TCLOCK

Los tiempos de ejecución de la estación de inserción de ejes está medida tras la realiación de 30 mediciones, 10 en las que se aceptaba el eje, otras 10 volteando y aceptando el eje, y 10 mediciones rechazando el eje. En las tablas se muestran, por una parte el número de ciclos de SCAN que se han necesitado para completar un proceso completo, y por otro el tiempo de ejecución de cada proceso medido en milisegundos.

En la tabla 5.78 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma OO.

En la tabla 5.79 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma estructurado.

A continuación, se muestra de forma gráfica, usando los diagramas de caja, las cuartiles de los tiempos de ejecución de los distintos procesos que conforman la estación de montaje de rodamientos, así como sus medias, desviación típica y covarianza de dichos tiempos.

En la figura 5.80 se puede observar el diagrama de cajas de los ciclos ejecutados por la estación para llevar a cabo la inserción de ejes.

En la figura 5.81 se muestra el diagrama de cajas del proceso que permite inicializar los objetos que conforman la estación dejando la posición inicial todos los elementos de la misma.

En la figura 5.82 se muestra el diagrama de cajas del proceso que permite sacar un eje del alimentador *"tipo petaca"* donde se encuentran ubicados.

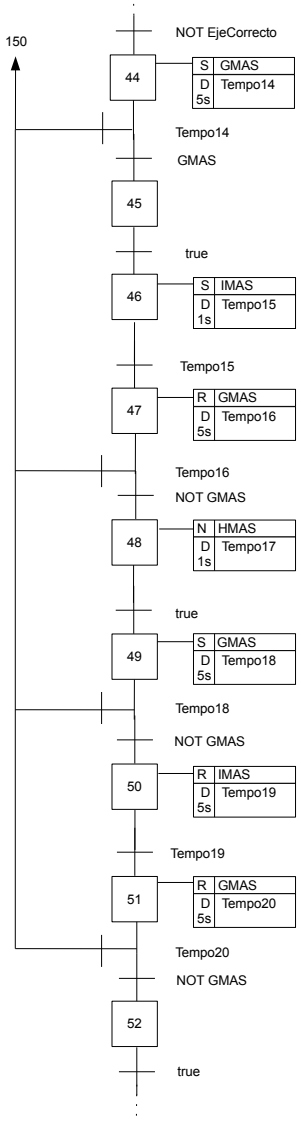

Figura 5.77: Proceso de volteo de la versión estructurada

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	1º giro plato	Medición	2º giro plato	Volteo	3º giro plato	Detección metal	4º giro plato	Detección nylon	5º giro plato	Expulsión	Inserción
9292	2.17	30	1007	2299	1189	2298	0	2298	4	2295	7	4598	0	3780
9292	2.17	31	1010	2296	1187	2298	0	2298	6	2298	6	4598	0	3748
9243	2.17	30	1008	2298	1186	2296	0	2298	4	2298	6	4600	0	3644
9197	2.17	31	1008	2299	1182	2300	0	2296	4	2298	6	4604	0	3610
9157	2.17	32	1010	2296	1183	2303	0	2298	4	2296	6	4603	0	3572
9195	2.17	30	1010	2296	1180	2304	0	2296	5	2300	8	4598	0	3561
9138	2.17	31	1011	2296	1183	2296	0	2298	4	2296	6	4599	0	3519
9142	2.17	30	1009	2300	1183	2299	0	2296	4	2296	6	4606	0	3512
9098	2.17	32	1008	2298	1183	2302	0	2298	5	2296	7	4603	0	3495
9130	2.17	32	1012	2297	1182	2303	0	2298	5	2299	8	4600	0	3486
12092	2.17	31	1011	2297	3011	2300	4623	2301	4	2299	6	4603	0	3493
12093	2.17	31	1010	2298	3010	2298	4618	2298	5	2300	7	4598	0	3479
12026	2.17	31	1008	2298	3009	2299	4618	2297	4	2299	6	4597	0	3468
12015	2.17	30	1010	2299	3009	2298	4620	2297	4	2300	8	4600	0	3459
12089	2.17	31	1010	2298	3010	2299	4620	2302	4	2296	6	4604	0	3464
12093	2.17	32	1009	2301	3010	2301	4624	2297	4	2296	6	4598	0	3458
12053	2.17	32	1009	2297	3010	2300	4621	2301	4	2300	6	4608	0	3457
12093	2.17	30	1009	2296	3010	2298	4624	2297	6	2296	6	4603	0	3465
12057	2.17	33	1008	2302	3009	2304	4615	2300	4	2299	6	4599	0	3455
11982	2.17	31	1009	2296	3008	2300	4620	2303	4	2298	6	4596	0	3450
7831	2.17	32	1012	2295	1187	2399	0	2298	4	2298	7	2304	2851	0
7776	2.17	32	1009	2298	1186	2300	0	2298	4	2297	7	2300	2846	0
7835	2.17	31	1009	2296	1188	2300	0	2300	6	2296	6	2302	2832	0
7748	2.17	32	1011	2299	1188	2300	0	2298	4	2300	6	2301	2836	0
7797	2.17	31	1009	2297	1187	2302	0	2299	4	2296	8	2302	2832	0
7797	2.17	31	1009	2298	1183	2298	0	2296	4	2298	6	2301	2830	0
7783	2.17	30	1011	2298	1184	2300	0	2298	4	2297	6	2301	2838	0
7786	2.17	30	1008	2302	1184	2302	0	2299	6	2298	8	2302	2832	0
7691	2.17	31	1011	2299	1180	2301	0	2299	4	2296	6	2302	2836	0
7753	2.17	30	1007	2300	1182	2296	0	2297	4	2296	8	2300	2834	0

Figura 5.78: Tiempos de ejecución de la estación 4 en su versión OO

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	1º giro plato	Medición	2º giro plato	Volteo	3º giro plato	Detección metal	4º giro plato	Detección nylon	5º giro plato	Expulsión	Inserción
11604	2.13	18	1007	3294	177	3294	0	3294	4	3294	6	6596	0	3499
11596	2.13	18	1005	3298	176	3292	0	3295	4	3296	4	6589	0	3456
11548	2.13	19	1006	3296	175	3295	0	3299	4	3293	4	6594	0	3447
11517	2.13	20	1005	3293	175	3296	0	3299	3	3295	5	6589	0	3441
11574	2.13	19	1006	3294	175	3295	0	3292	5	3297	3	6599	0	3445
11511	2.13	19	1008	3297	176	3298	0	3293	7	3294	4	6597	0	3427
11543	2.13	20	1006	3297	177	3298	0	3295	3	3294	5	6594	0	3427
11568	2.13	20	1007	3296	175	3297	0	3297	5	3292	4	6589	0	3463
11504	2.13	19	1006	3294	174	3296	0	3295	4	3296	4	6597	0	3437
11512	2.13	20	1006	3294	175	3296	0	3297	4	3297	4	6592	0	3442
15105	2.13	20	1007	3295	3004	3298	4613	3298	4	3293	3	6591	0	3462
15105	2.13	18	1007	3297	3003	3296	4609	3296	3	3295	4	6592	0	3437
15094	2.13	21	1006	3300	3008	3297	4609	3294	6	3295	5	6593	0	3431
15117	2.13	18	1007	3296	3004	3298	4611	3295	4	3292	4	6590	0	3428
15037	2.13	19	1008	3298	3004	3296	4608	3293	4	3294	7	6591	0	3424
15067	2.13	21	1007	3290	3005	3300	4612	3295	4	3297	3	6594	0	3429
15053	2.13	20	1007	3297	3004	3293	4610	3300	4	3294	5	6597	0	3423
15083	2.13	18	1005	3295	3004	3296	4611	3296	4	3296	4	6587	0	3412
15073	2.13	20	1008	3297	3004	3294	4612	3294	6	3296	4	6591	0	3411
15075	2.12	21	1007	3293	3007	3297	4613	3296	5	3296	3	6595	0	3419
9746	2.13	21	1008	3296	180	3298	0	3295	4	3294	4	3298	2839	0
9764	2.13	18	1005	3293	177	3296	0	3299	4	3295	4	3297	2830	0
9686	2.13	18	1007	3296	176	3295	0	3294	3	3293	5	3296	2826	0
9753	2.13	19	1007	3299	178	3298	0	3298	3	3299	3	3299	2837	0
9722	2.13	21	1007	3293	177	3295	0	3290	4	3294	6	3302	2827	0
9723	2.13	18	1006	3295	177	3298	0	3295	4	3297	4	3298	2820	0
9705	2.13	18	1007	3294	176	3294	0	3299	5	3296	4	3300	2824	0
9772	2.13	20	1007	3292	175	3294	0	3295	4	3291	4	3303	2823	0
9764	2.12	18	1006	3292	175	3294	0	3291	4	3295	3	3300	2844	0
9774	2.13	18	1004	3294	176	3294	0	3292	4	3292	4	3299	2829	0

Figura 5.79: Tiempos de ejecución de la estación 4 en su versión estructurada

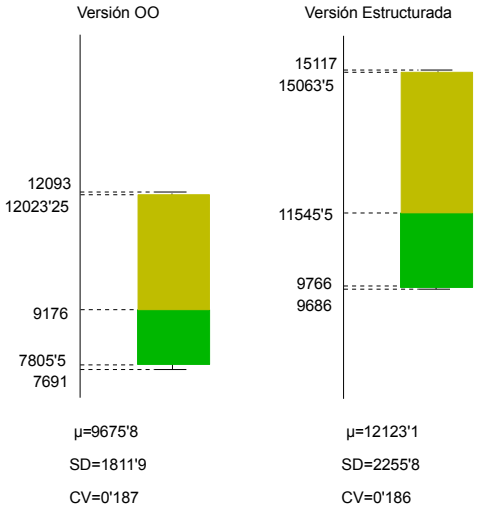

Figura 5.80: Diagrama de cajas del proceso *"número de ciclos"* de la estación 4

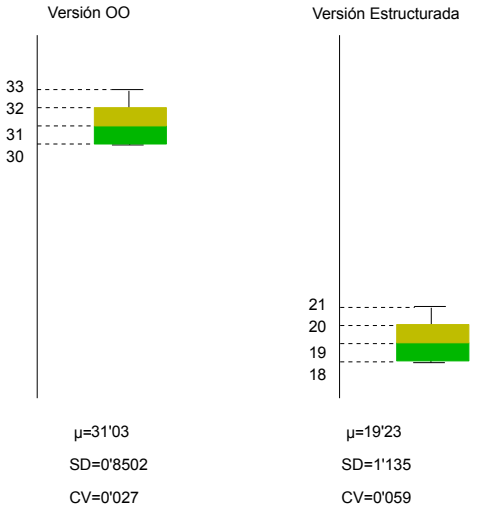

Figura 5.81: Diagrama de cajas del proceso *"Inicialización"* de la estación 4

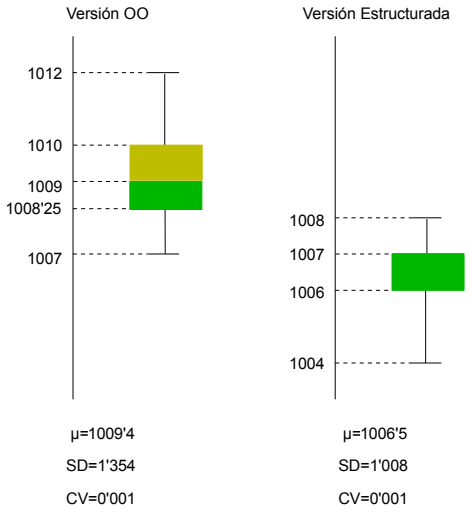

Figura 5.82: Diagrama de cajas del proceso *"Alimentación"* de la estación 4

En la figura 5.83 se muestra el diagrama de cajas del proceso que realiza un primer giro al plato divisor.

En la figura 5.84 se muestra el diagrama de cajas del proceso que mide la altura del eje para comprobar si ha sido insertado correctamente.

En la figura 5.85 se muestra el diagrama de cajas del proceso que realiza un segundo giro al plato divisor.

En la figura 5.86 se muestra el diagrama de cajas del proceso que gira el eje en caso de que se haya detectado que ha sido introducido de forma incorrecta.

En la figura 5.87 se muestra el diagrama de cajas del proceso que realiza un tercer giro al plato divisor.

En la figura 5.88 se muestra el diagrama de cajas del proceso que determina si el eje insertado es metálico.

En la figura 5.89 se muestra el diagrama de cajas del proceso que realiza un cuarto giro al plato divisor.

En la figura 5.90 se muestra el diagrama de cajas del proceso que comprueba si el material del que se compone el eje es nylon.

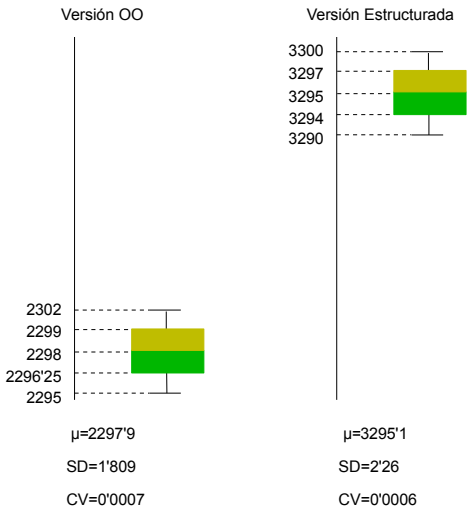

Figura 5.83: Diagrama de cajas del proceso *"1º giro al plato divisior"* de la estación 4

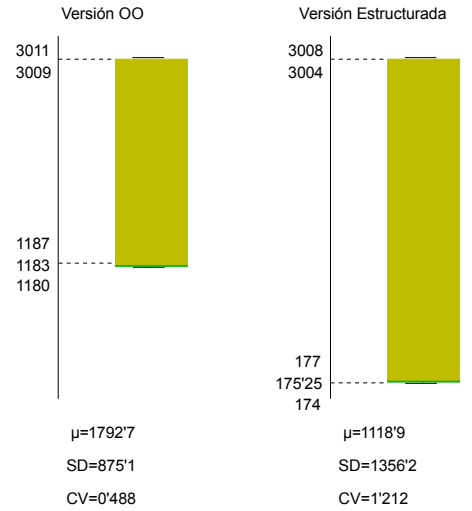

Figura 5.84: Diagrama de cajas del proceso *"Medicion"* de la estación 4

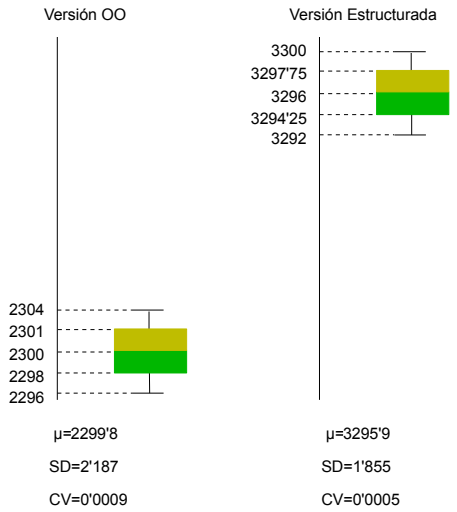

Figura 5.85: Diagrama de cajas del proceso *"2º giro al plato divisior"* de la estación 4

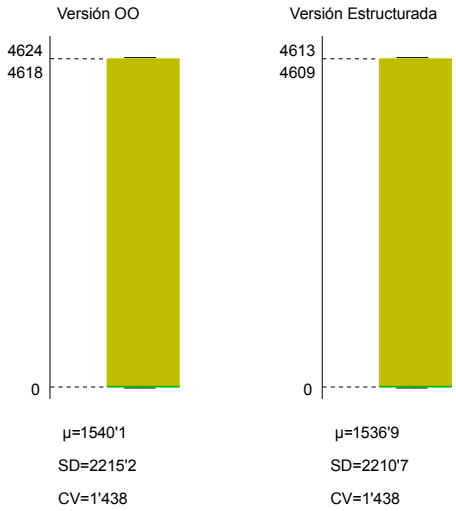

Figura 5.86: Diagrama de cajas del proceso *"Volteo"* de la estación 4

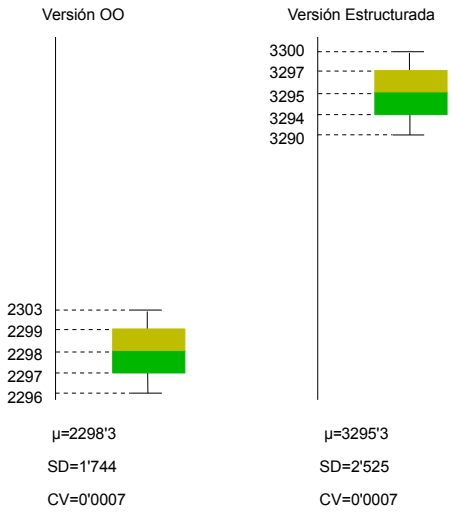

Figura 5.87: Diagrama de cajas del proceso *"3º giro al plato divisior"* de la estación 4

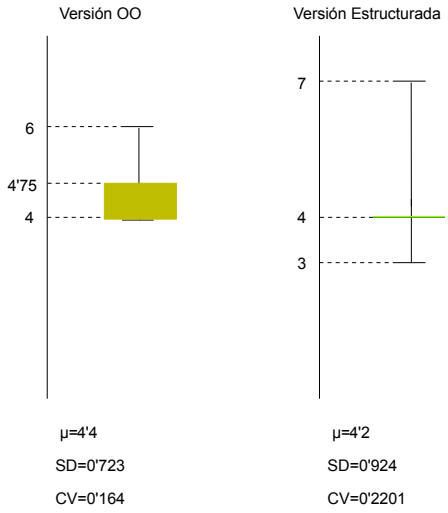

Figura 5.88: Diagrama de cajas del proceso *"Detectar eje metal"* de la estación 4

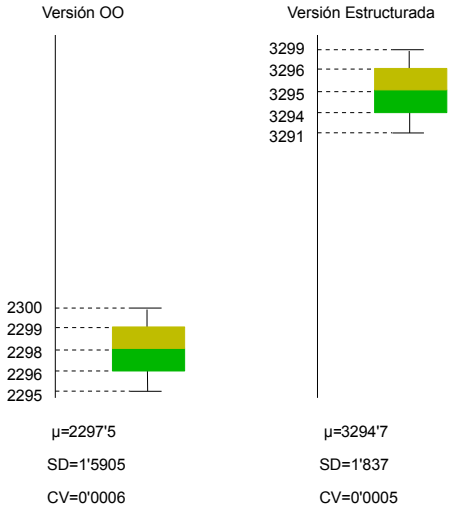

Figura 5.89: Diagrama de cajas del proceso *"4º giro al plato divisior"* de la estación 4

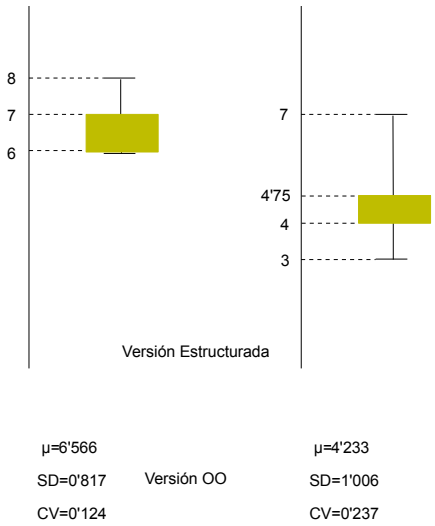

Figura 5.90: Diagrama de cajas del proceso *"Detectar eje nylon"* de la estación 4

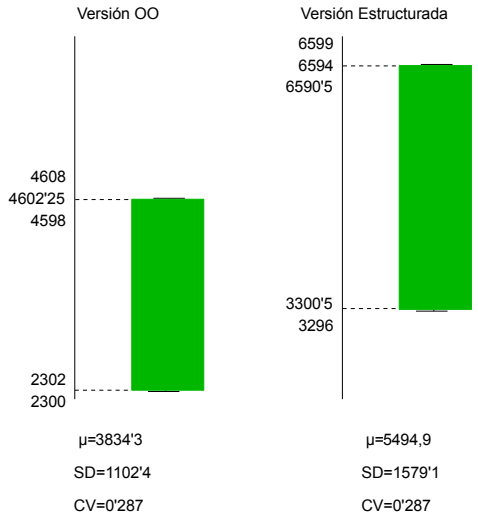

Figura 5.91: Diagrama de cajas del proceso *"5º giro al plato divisior"* de la estación 4

En la figura 5.91 se muestra el diagrama de cajas del proceso que realiza un quinto giro al plato divisor.

En la figura 5.92 se muestra el diagrama de cajas del proceso que desecha el eje en el caso de que el tipo detectado no coincida con el que había seleccionado el usuario.

En la figura 5.93 se muestra el diagrama de cajas del proceso que inserta un eje correcto en el conjunto.

5.9. Estación 5 - Estación de montaje de tapas

El quinto de los componentes que se procede a ensamblar se corresponde con una tapa que se deposita sobre la base. Ésta se encaja en un alojamiento dispuesto a tal efecto y tiene el propósito de retener el eje del dispositivo de giro, previamente montado en la estación anterior.

Si ya en la cuarta estación se introducía una serie de variantes en los conjuntos finales montados, al existir la posibilidad de utilizar ejes de dos materiales diferentes, en esta estación se multiplica el número de variaciones. En este caso son

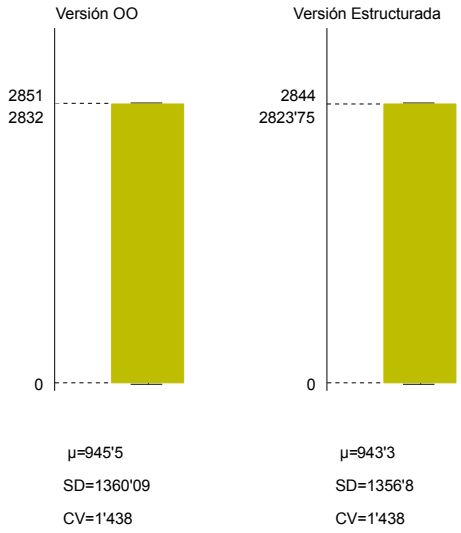

Figura 5.92: Diagrama de cajas del proceso *"Expulsion"* de la estación 4

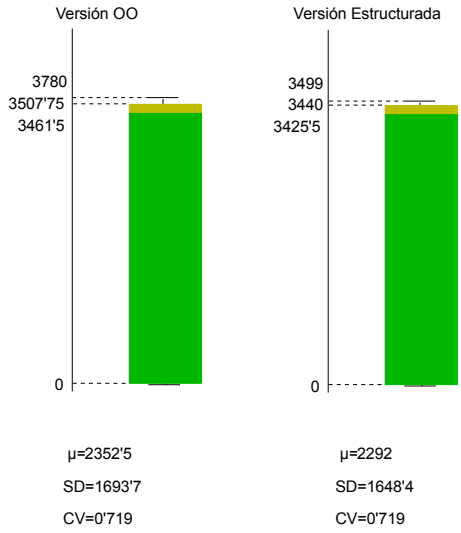

Figura 5.93: Diagrama de cajas del proceso *"Inserción"* de la estación 4

tres los materiales con los que se han fabricado las tapas; aluminio, nylon blanco y nylon negro. Además, se añade la característica de que estos tres tipos de tapas pueden presentar dos alturas distintas, completando un total de seis combinaciones de posibles piezas a ensamblar.

La necesidad de realizar las verificaciones pertinentes para determinar qué tipo de tapa va a ser montada en cada ciclo, hace que esta estación presente la mayor complejidad en cuanto a las operaciones a realizar. Igualmente, es necesario que el control de la misma realice las operaciones de selección de pieza de forma coordinada, en función de las órdenes proporcionadas por el *"transfer"* encargado de la gestión de la producción de la FMS-200. Para este fin, recibe como parámetros desde el *"transfer"* el tipo de pieza detallada realizando un número máximo de 8 intentos de colocación, suponiendo a partir de ahí que no se encuentra en el cargador ninguna tapa de acuerdo con las especificaciones dadas desde el maestro de producción.

5.9.1. Etapas del proceso

La estructura adoptada para la realización del proceso de montaje de la tapa es similar a la presentada en la estación anterior. Esta estructura basada en la inclusión de un plato divisor con ocho estaciones, permite mejorar el método de trabajo a realizar en la estación. Por una parte, ahorra espacio respecto a otras distribuciones espaciales de los puestos de trabajo posibilitando a su vez que todas las manipulaciones se puedan realizar de forma simultánea, optimizando el proceso llevado a cabo.

El número de operaciones a realizar, así como los componentes que intervienen en las mismas se detallan a continuación:

Plato divisor

El plato divisor de ocho posiciones que de forma alternativa produce el intercambio de piezas entre las sucesivas estaciones, presenta idéntica estructura al utilizado en la estación anterior. Para su funcionamiento se incluyen dos cilindros de tope fijo y móvil respectivamente, junto a un cilindro empujador que produce el giro, funcionando según la secuencia detallada con anterioridad.

Módulo alimentación

Para el almacenamiento y posterior alimentación de las tapas se utiliza un cargador de gravedad provisto de un empujador accionado mediante cilindro neumático que realiza la extracción.

En esta subestación, al contrario que en la de ejes, se han de introducir las tapas de forma correcta, ya que no existe ningún elemento que posteriormente la coloque de forma correcta. Y lo que es mas grave, tampoco es posible su extracción del proceso provocando inexorablemente un atasco en el mismo.

Subestación de carga

La alimentación de material descrita en el punto anterior, realiza el suministro de tapas que posteriormente serán montadas en el conjunto preparado en el palet. Sin embargo, tras esta fase, resulta necesario realizar la carga de la tapa sobre el plato divisor. Para ello, se dispone de un manipulador compuesto por un cilindro rotolineal encargado de realizar la elevación y posterior giro de un brazo con una pinza de dos dedos de apertura paralela. El giro de 180^{o} realizado por este elemento sitúa la pinza sobre el punto exacto de descarga de la tapa en la primera estación del plato divisor.

Subestaciones de detección de material

Como ya se ha indicado anteriormente, esta subestación brinda la posibilidad de trabajar con tapas de aluminio, nylon blanco y nylon negro respectivamente.

Para poder diferenciar las tapas del primero de estos tipos, la segunda de las subestaciones del plato divisor está dotada con un captador inductivo. Éste proporciona una señal al PLC, únicamente en el caso de que la tapa que el plato sitúa frente a él cada vez que realiza un giro sea de aluminio.

La detección de las tapas realizadas en nylon se lleva a cabo a través de otro sensor capacitivo que a diferencia del anterior, proporciona también una señal de detección, al situar frente a él piezas que no sean metálicas.

Con el fin de diferenciar finalmente las tapas de nylon blanco de las de color negro, la subestación dispone de un detector fotoeléctrico. Dicho componente proporciona una detección únicamente de las tapas de color blanco.

Figura 5.94: Encoder

Subestación medición tapa

El hecho de disponer de tapas con dos alturas diferentes hace necesario disponer de un elemento que realice una medición de las mismas. Debido a la finalidad didáctica con la cual se ha realizado el diseño de esta célula, se han adoptado soluciones variadas para llevar a cabo operaciones similares. En otras estaciones, para operaciones de medición se recurre a componentes como cilindros neumáticos con detectores de altura correcta o palpadores con salida analógica. En este caso, se ha considerado de interés utilizar un transductor digital que proporciona una salida por pulsos, como es el encoder lineal.

La figura 5.94 muestra el aspecto del componente utilizado, el cuál consta de un cilindro neumático que desplaza el palpador hasta contactar con la tapa, en combinación con un encoder lineal integrado en el mismo cilindro. Mediante la cuenta de los pulsos suministrados por el encoder realizada mediante una entrada de contaje rápido del PLC, es posible determinar la distancia que avanza al cilindro hasta contactar con la tapa. Disponiendo de este dato, es posible determinar de forma directa la altura de la tapa.

Evacuación tapa incorrecta

La penúltima subestación del plato divisisor es la encargada de rechazar el producto si este no coincide con el material y altura seleccionado por el planificador.

Para ello, se dispone de un manipulador de dos ejes que en caso de recibir la orden correspondiente, recoge la tapa del plato divisor y la deposita sobre una rampa de evacuación. Los componentes que forman parte de este manipulador son dos cilindros neumáticos de vástagos paralelos a modo de ejes sobre los que se ha fijado como elemento terminal una placa con tres ventosas para sujeción por vacío.

Inserción tapa

Al igual que sucedía en la subestación anterior, la última de las subestaciones está encargada del montaje de la tapa sobre el conjunto retenido en la cinta transportadora.

El manipulador usado para realizar esta tarea presenta idénticas características que el utilizado para depositar la tapa sobre el plato divisor. Dispone de una pinza de apertura paralela para sujetar la tapa, que es elevada y desplazada realizando un giro hasta el punto de descarga mediante un actuador neumático rotolineal.

5.9.2. Diagrama de clases

La versión OO de la estación 5 se compone de 15 clases y su diagrama de clases se muestra en la figura 5.95.

5.9.3. Programación OO vs programación estructurada

En esta subsección se mostrarán los procesos de montaje de tapas en el palet (proceso *"Cargador"*) y el proceso encargado de detectar si la tapa es metálica (proceso *"DetectorMetal"*).

Método *"Run"* de la clase Estacion5

En la figura 5.96 se muestra las ramas concernientes a la ejecución automática de la estación 5.

Proceso plato de la versión OO

La etapa 2 del método *"Run"* de la versión OO de la estación 5 (ver figura 5.96)

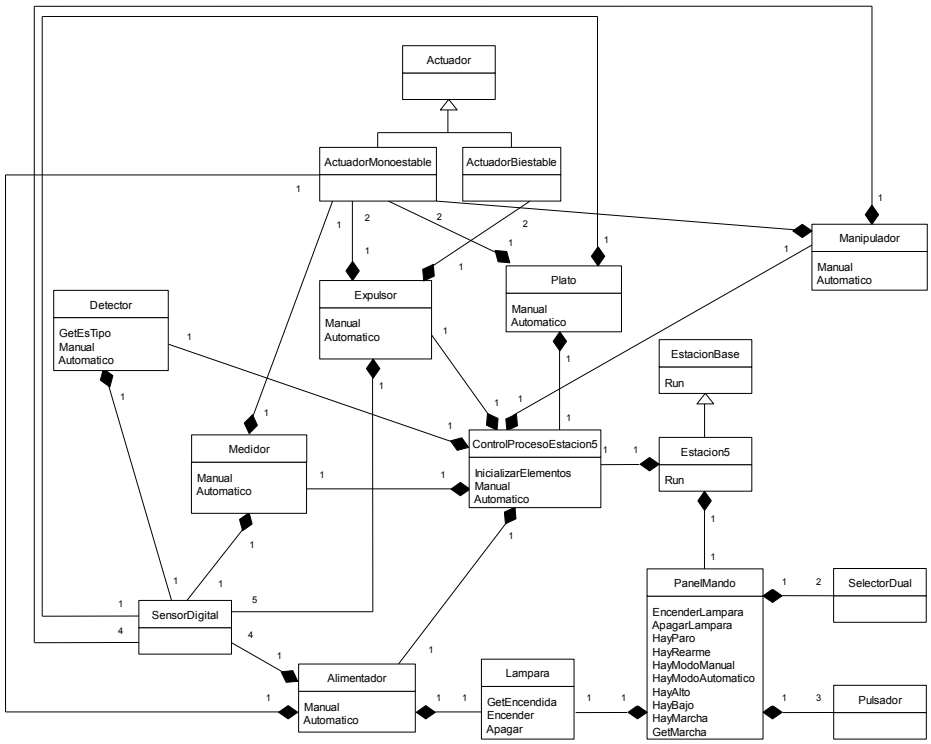

Figura 5.95: Diagrama de clases de la estación 5

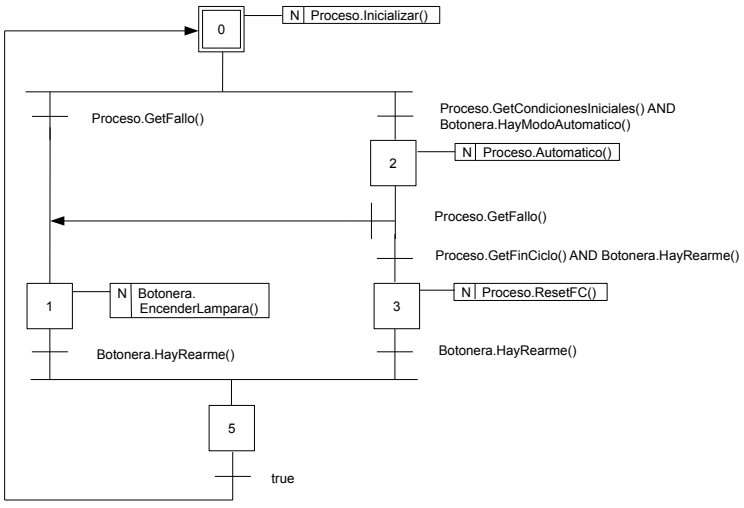

Figura 5.96: Método *"Run"* de la clase *"Estacion5"*

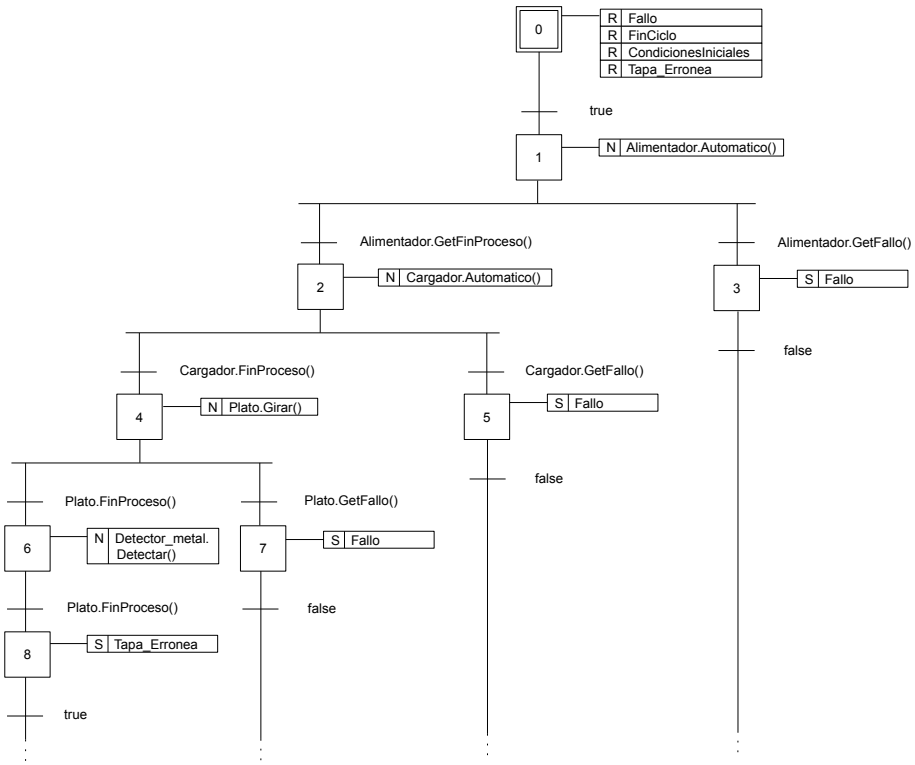

Figura 5.97: Método de ejecución automática de ControlProcesoEstacion5

tiene asociada una acción llamada al método *"Automatico"* de la clase *"Control-ProcesoEstacion5"*. En la figura 5.97 se muestra el trozo de código del método *"Automatico"* correspondiente al proceso de montaje de tapas en el palet.

La parte interesante del método *"Automatico"* correspondiente a la acción montaje de tapas se corresponde con la etapa 2 que tiene asociada una invocación al método *"Automatico"* del objeto *"Cargador"* que es una instancia de la clase *"Manipulador"* que es quien realiza la tarea de montaje de tapas (ver figura 5.98).

Las acciones de la etapa 0 se corresponden con "*Fallo", "CondicionesIniciales" y "FinProceso"* son atributos boolean de la clase.

"CilMono_ Avance.Accionado", *"CilMono_ Bajada.Accionado"* y *"CilMono_ Pinza.Accionado"* son llamadas al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

La etapa 1, 2, 3, 5, 6, 7, 8 y 9 tienen asociado una acción que se corresponden con una llamada al método *"Activado"* de la clase *"SensorDigital"*.

Proceso cargador de la versión estructurada

En la figura 5.99 se muestra un trozo del GRAFCET que controla el proceso montaje de las tapas.

La etapa 242 tiene asociada la acción *"BMAS"* que se corresponde con la señal de avance de giro del manipulador de carga.

La etapa 243 tiene asociada la acción *"AMAS"* que se corresponde con la señal de bajada del manipulador de carga.

La etapa 243 tiene asociada la transición *"b0"* que se corresponde con la señal de detección del inicio del giro del manipulador carga.

La etapa 244 tiene asociada la acción *"CMAS"* que se corresponde con la señal de cierre de la pinza del manipulador de carga.

La etapa 244 tiene asociada la transición *"a1"* que se corresponde con la señal de detección del manipulador de carga abajo.

La etapa 245 tiene asociada la transición *"a0"* que se corresponde con la señal de detección del manipulador de carga arriba.

La etapa 150 tiene asociada la transición *"DEFECTO"* que se corresponde con el piloto luminoso por defecto de la estación.

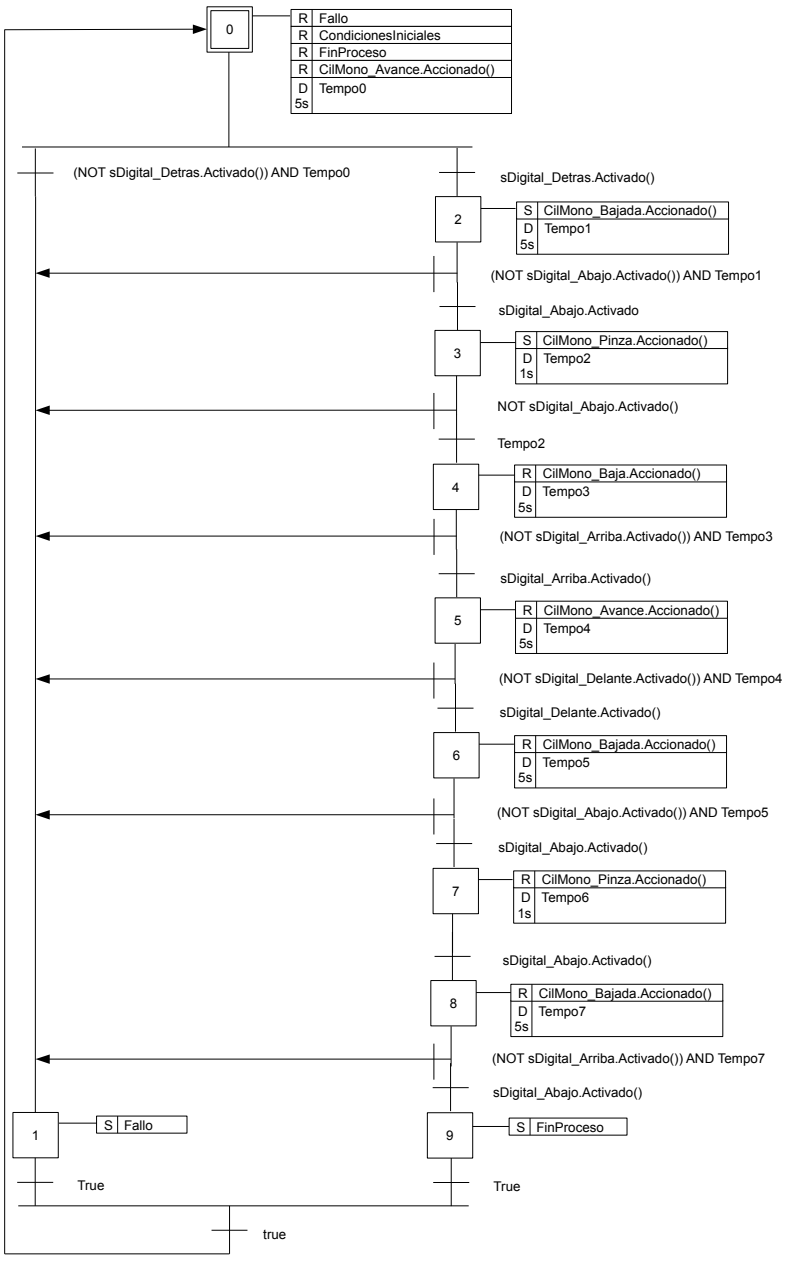

Figura 5.98: Método de ejecucion automatica de la clase *"Manipulador"*

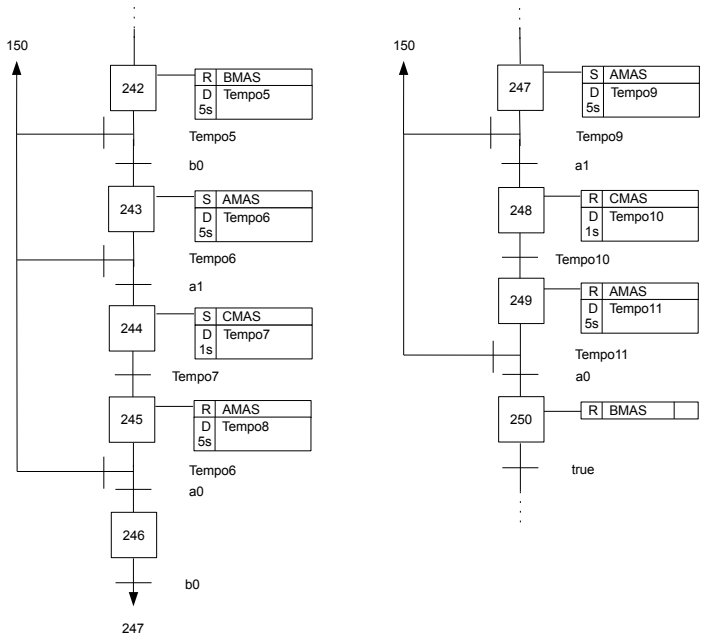

Figura 5.99: Proceso cargador de la versión estructurada

Proceso detector metal de la versión OO

En la etapa 6 de la figura 5.97 se realiza la acción de detectar si la tapa es de metal por medio de la acción *"DetMetal"*, que es una llamada al método *"Detectar"* de la clase *"Detector"* que es la que realiza la tarea (ver figura 5.100).

Las acciones de la etapa 0 se corresponden con "*EsTipo"* y *"CondicionesIniciales"* son atributos boolean de la clase.

Proceso detector metal de la versión estructurada

En la figura 5.101 se muestra un trozo del GRAFCET que controla si la tapa es de tipo metal.

La etapa 35 tiene asociada la acción *"dm"* que se corresponde con el sensor de detección de tapa metálica.

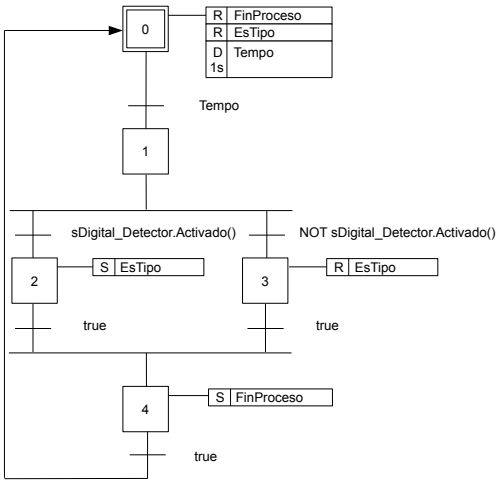

Figura 5.100: Método de detección de estructura metálica de la clase *"Detector"*

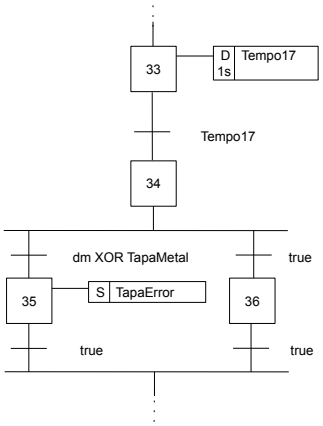

Figura 5.101: Proceso de detectar metal de la versión estructurada

5.9.4. Resultados de tiempos del módulo TCLOCK

Los tiempos de ejecución de la estación de montaje de tapas está medida tras la realiación de 20 mediciones de las cuales, en 10 se aceptaba la pieza y en las otras 10 la pieza era rechazada. En las tablas se muestran, por una parte el número de ciclos de SCAN que se han necesitado para completar un proceso completo, y por otro el tiempo de ejecución de cada proceso medido en milisegundos.

En la tabla 5.102 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma OO.

En la tabla 5.103 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma estructurado.

A continuación, se muestra de forma gráfica, usando los diagramas de caja, las cuartiles de los tiempos de ejecución de los distintos procesos que conforman la estación de montaje de bases, así como sus medias, desviación típica y covarianza de dichos tiempos.

En la figura 5.104 se puede observar el diagrama de cajas de los ciclos ejecutados por la estación para llevar a cabo el montaje de tapas.

En la figura 5.105 se muestra el diagrama de cajas del proceso que permite inicializar los objetos que conforman la estación dejando la posición inicial todos los elementos de la misma.

En la figura 5.106 se muestra el diagrama de cajas del proceso que permite sacar una tapa del alimentador *"tipo petaca"*.

En la figura 5.107 se muestra el diagrama de cajas del proceso que realiza la carga de la tapa sobre el plato divisor.

En la figura 5.108 se muestra el diagrama de cajas del proceso que ejecuta el primer giro del plato divisior.

En la figura 5.109 se muestra el diagrama de cajas del proceso que detecta si el material de la tapa es de tipo metálico.

En la figura 5.110 se muestra el diagrama de cajas del proceso que ejecuta el segundo giro del plato divisior.

En la figura 5.111 se muestra el diagrama de cajas del proceso que detecta si el material de la tapa es de tipo nylon blanco.

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Carga	1º giro plato	Detección metal	2º giro plato	Detección nylon	3º giro plato	Detección color	4º giro plato	Medición	5º giro plato	Expulsión	Inserción
12835	2,16	1036	7591	5476	3228	7	1613	7	1614	7	1615	3009	3229	0	4541
12769	2,17	1033	7594	5485	3228	5	1613	9	1615	6	1611	3010	3230	0	4535
12831	2,16	1033	7592	5481	3226	3	1612	7	1613	6	1614	3008	3227	0	4532
12863	2,15	1032	7574	5463	3228	5	1614	7	1610	6	1614	3011	3235	0	4526
12842	2,16	1035	7579	5469	3224	4	1612	8	1615	6	1612	3010	3230	0	4522
12800	2,16	1033	7576	5465	3224	3	1613	5	1616	5	1613	3008	3228	0	4526
12878	2,15	1030	7577	5469	3232	3	1613	7	1610	9	1613	3010	3233	0	4526
12723	2,17	1034	7568	5461	3226	3	1611	8	1612	6	1611	3009	3229	0	4519
12819	2,16	1034	7569	5458	3230	6	1613	9	1613	7	1613	3009	3230	0	4526
12798	2,16	1035	7570	5461	3224	3	1612	6	1611	8	1614	3009	3230	0	4518
12299	2,15	1035	7578	5468	3226	5	1612	7	1612	7	1612	3011	1616	4597	0
12173	2,16	1034	7569	5456	3226	3	1612	8	1610	8	1612	3011	1617	4594	0
12148	2,15	1035	7568	5458	3232	3	1614	6	1614	8	1615	3008	1618	4591	0
12147	2,15	1034	7540	5425	3226	3	1612	6	1611	9	1616	3008	1619	4587	0
12114	2,16	1033	7565	5454	3224	5	1612	6	1615	6	1614	3012	1617	4594	0
12104	2,16	1036	7556	5446	3229	4	1614	5	1613	7	1613	3009	1617	4590	0
12170	2,16	1035	7559	5449	3228	5	1611	6	1610	8	1614	3012	1620	4591	0
12104	2,16	1034	7564	5454	3226	5	1611	5	1614	5	1613	3011	1620	4586	0
12056	2,16	1035	7562	5451	3225	5	1609	6	1612	6	1611	3010	1616	4596	0
12109	2,16	1034	7562	5452	3227	3	1612	6	1613	6	1612	3009	1621	4592	0

Figura 5.102: Tiempos de ejecución de la estación 5 en su versión OO

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Carga	1º giro plato	Detección metal	2º giro plato	Detección nylon	3º giro plato	Detección color	4º giro plato	Medición	5º giro plato	Expulsión	Inserción
12667	2'08	1017	5612	5612	3321	4	1608	4	1608	4	1608	3004	3221	0	4553
12641	2'08	1014	6692	5593	3321	5	1608	4	1608	4	1606	3005	3225	0	4534
12685	2'07	1015	6682	5580	3317	4	1610	4	1610	5	1607	3006	3224	0	4521
12620	2'08	1015	6619	5517	3316	6	1610	6	1610	6	1608	3004	3223	0	4532
12657	2'07	1017	6624	5521	3321	4	1607	4	1607	4	1608	3007	3219	0	4520
12656	2'08	1015	6625	5521	3324	6	1609	4	1609	3	1609	3005	3223	0	4529
12612	2'08	1013	6625	5522	3322	4	1606	6	1606	4	1606	3005	3223	0	4534
12605	2'08	1016	6611	5510	3318	5	1608	4	1608	4	1606	3004	3223	0	4529
12629	2'08	1017	6617	5515	3320	4	1607	4	1607	4	1608	3006	3223	0	4540
12600	2'08	1014	6589	5486	3320	4	1607	4	1607	4	1608	3006	3224	0	4531
12304	2'08	1014	6592	5490	3323	4	1608	4	1608	4	1608	3004	1609	5594	0
12340	2'08	1015	6616	5514	3319	4	1610	3	1610	4	1609	3006	1609	5584	0
12404	2'08	1016	6589	5488	3323	4	1610	6	1610	7	1607	3009	1613	5591	0
12299	2'08	1016	6577	5476	3323	4	1611	5	1611	4	1610	3006	1608	5593	0
12286	2'08	1014	6569	5468	3320	4	1606	4	1606	4	1609	3003	1608	5583	0
12306	2'08	1018	6574	5471	3320	5	1606	5	1606	4	1608	3004	1609	5592	0
12342	2'08	1017	6566	5466	3318	4	1606	4	1606	6	1609	3007	1612	5588	0
12339	2'07	1016	6558	5458	3316	6	1609	7	1609	6	1607	3006	1611	5591	0
12308	2'07	1016	6581	5482	3319	4	1609	4	1609	4	1609	3006	1612	5591	0
12335	2'07	1016	6577	5475	3318	4	1607	4	1607	4	1609	3005	1611	5583	0

Figura 5.103: Tiempos de ejecución de la estación 5 en su versión estructurada

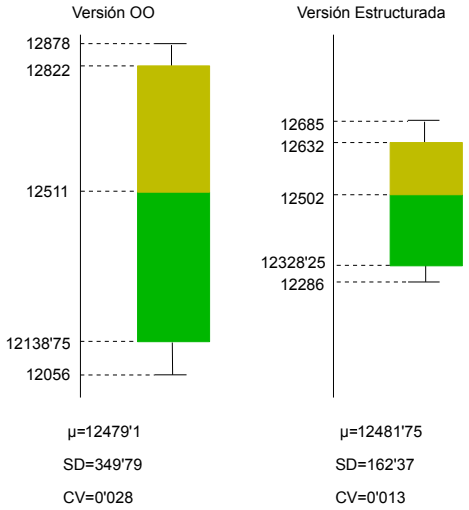

Figura 5.104: Diagrama de cajas del proceso *"número de ciclos"* de la estación 5

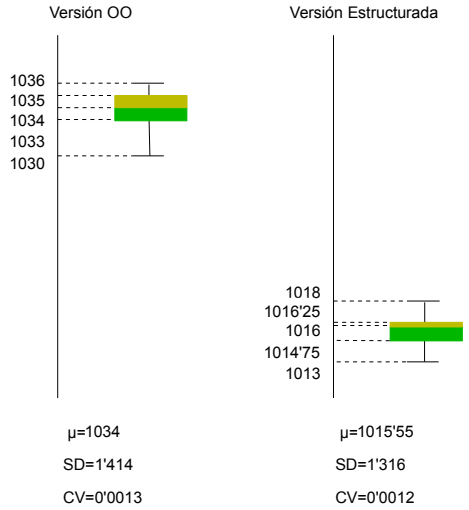

Figura 5.105: Diagrama de cajas del proceso *"Inicialización"* de la estación 5

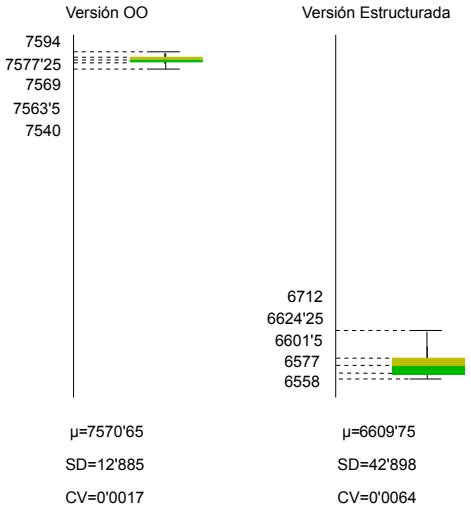

Figura 5.106: Diagrama de cajas del proceso *"Alimentación"* de la estación 5

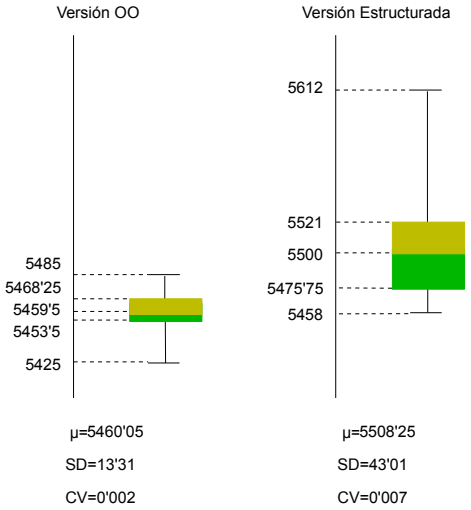

Figura 5.107: Diagrama de cajas del proceso *"Carga"* de la estación 5

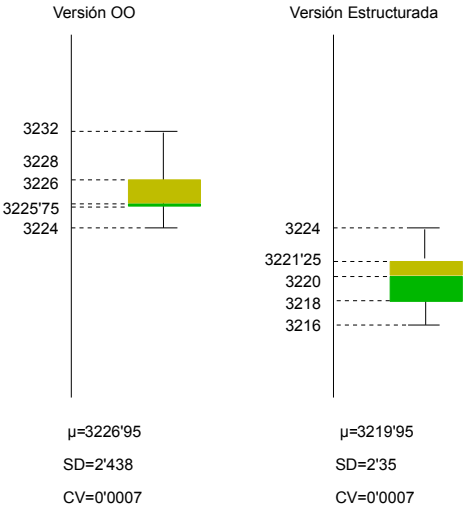

Figura 5.108: Diagrama de cajas del proceso *"1GiroPlato"* de la estación 5

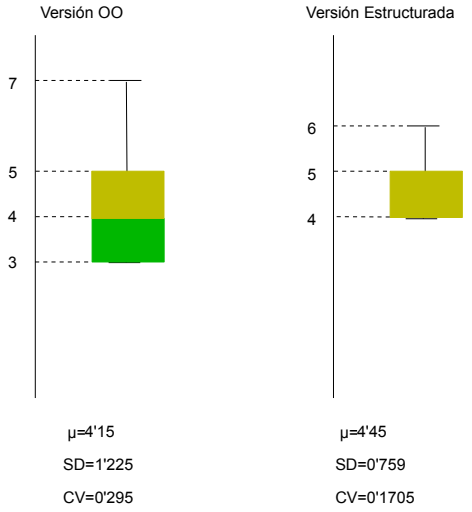

Figura 5.109: Diagrama de cajas del proceso *"DeteccionrMetal"* de la estación 5

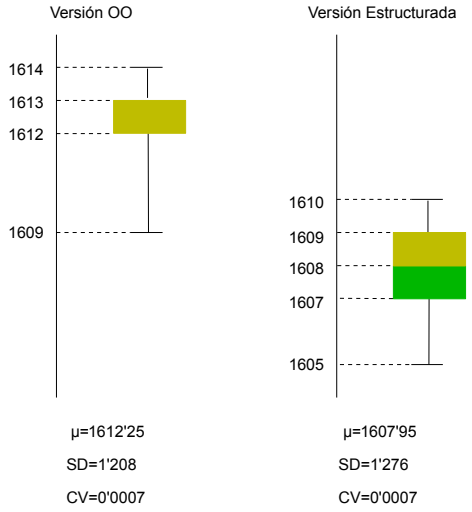

Figura 5.110: Diagrama de cajas del proceso *"2GiroPlato"* de la estación 5

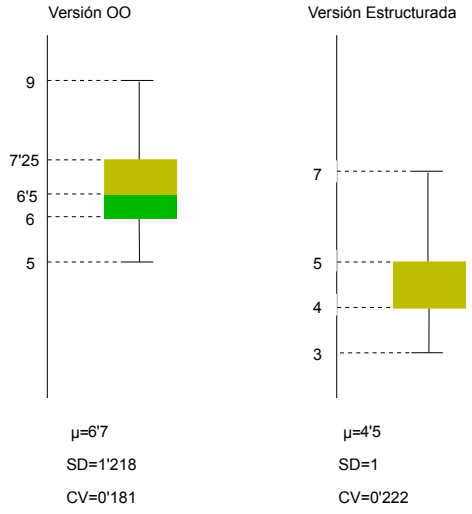

Figura 5.111: Diagrama de cajas del proceso *"DeteccionrNylon"* de la estación 5

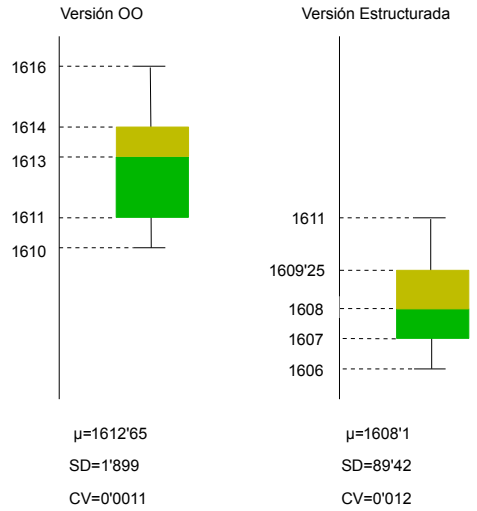

Figura 5.112: Diagrama de cajas del proceso *"3GiroPlato"* de la estación 5

En la figura 5.112 se muestra el diagrama de cajas del proceso que ejecuta el tercer giro del plato divisior.

En la figura 5.113 se muestra el diagrama de cajas del proceso que detecta si el material de la tapa es de tipo nylon negro.

En la figura 5.114 se muestra el diagrama de cajas del proceso que ejecuta el cuarto giro del plato divisior.

En la figura 5.115 se muestra el diagrama de cajas del proceso que realiza la medición de la tapa.

En la figura 5.116 se muestra el diagrama de cajas del proceso que ejecuta el quinto giro del plato divisior.

En la figura 5.117 se muestra el diagrama de cajas del proceso que expulsa la tapa en caso de que esta tenga una altura incorrecta y/o este compuesta de un material diferente al seleccionado por el usuario.

En la figura 5.118 se muestra el diagrama de cajas del proceso que realiza el montaje de la tapa sobre el conjunto retenido en la cinta transportadora.

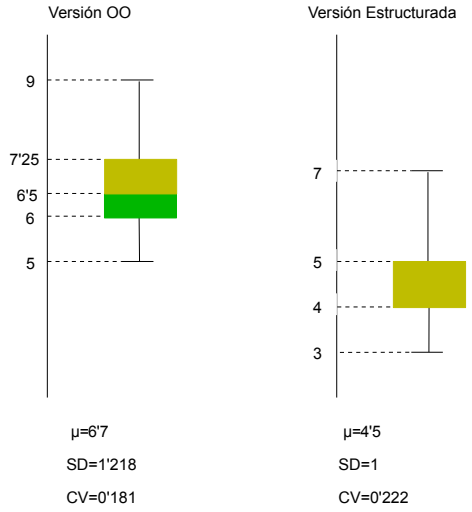

Figura 5.113: Diagrama de cajas del proceso *"DeteccionrNylon"* de la estación 5

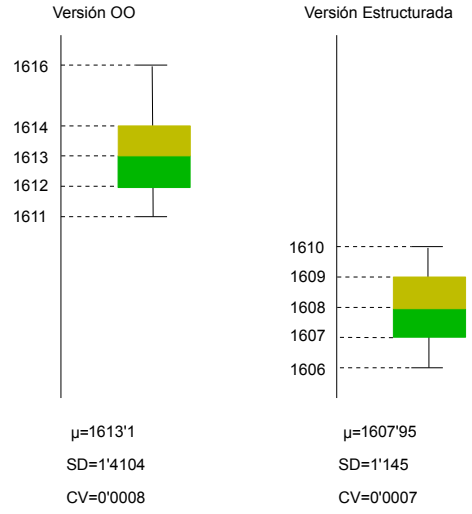

Figura 5.114: Diagrama de cajas del proceso *"4GiroPlato"* de la estación 5

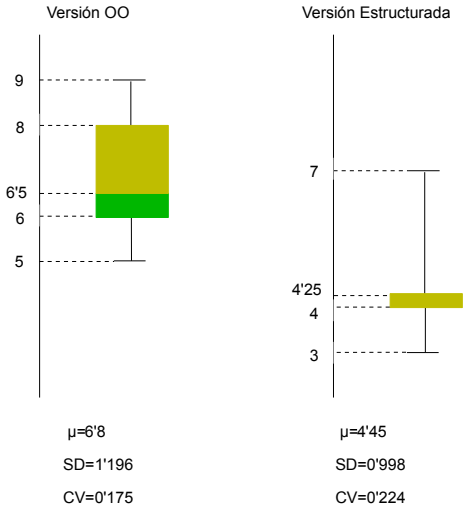

Figura 5.115: Diagrama de cajas del proceso *"Medicion"* de la estación 5

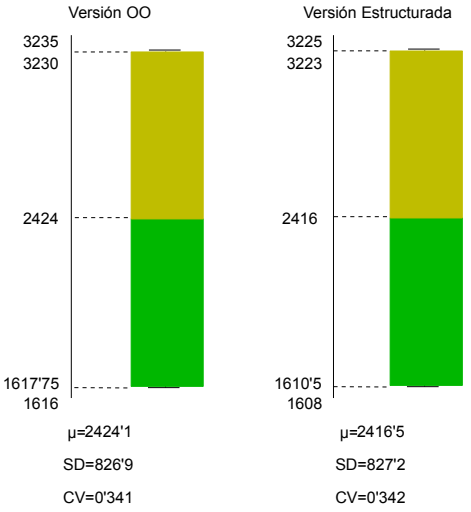

Figura 5.116: Diagrama de cajas del proceso *"5GiroPlato"* de la estación 5

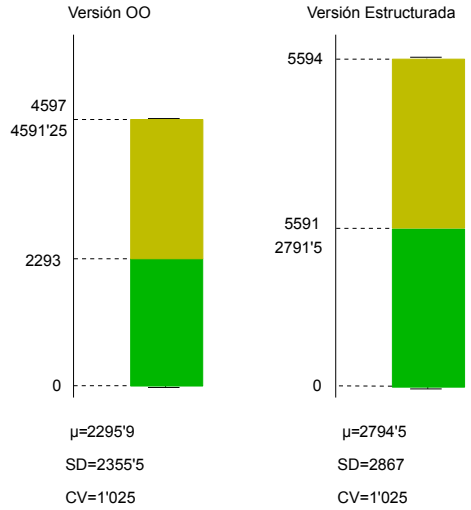

Figura 5.117: Diagrama de cajas del proceso *"Expulsion"* de la estación 5

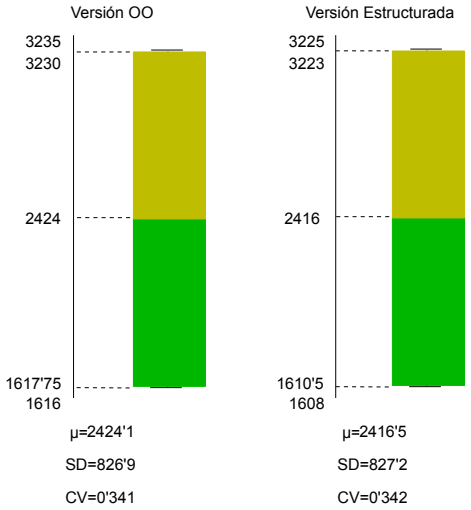

Figura 5.118: Diagrama de cajas del proceso *"Insercion"* de la estación 5

5.10. Estación 6 - Estación de inserción de tornillos

Esta estación está encargada de realizar la última de las operaciones de inserción de componentes en el conjunto final a construir. Dichos componentes consisten en cuatro tornillos que son depositados en otros tantos orificios roscados dispuestos en la base del dispositivo de giro.

La tecnología utilizada para materializar los movimientos de esta estación, basada en diferentes cilindros neumáticos, posibilita únicamente realizar la descarga de los tornillos en un único punto. Resulta necesario por tanto incluir un componente adicional en el *"transfer"* que realice sucesivos giros en el palet, de forma que al realizar cuatro ciclos de inserción mediante esta estación de trabajo, queden colocados los cuatro tornillos.

Los componentes utilizados para esta operación consisten en un cilindro neumático elevador sobre el que se encuentra montado un actuador de giro de idéntica tecnología.

A diferencia de estaciones anteriores en las cuales la comunicación entre las mismas y el *"transfer"* se limitaba al envío de mensajes de puesta en marcha e indicaciones de final de ciclo, en este caso, la necesidad de coordinación entre ambos se ve ampliada. Debido a ello, se potencia la utilización de las posibilidades que la red de autómatas ofrece para el intercambio de información entre los diferentes puestos. Como se detallara mas adelante en el apartado referido al código se realiza un proceso secuencial en la espera de las distintas fases de giro del palet. Incluyendo en estas fases las peticiones de comunicación entre el autómata que rige el transfer y el que gobierna la estación que se analiza en este apartado.

5.10.1. Etapas del proceso

Para realizar las tareas de esta estación se llevan a cabo una serie de operaciones detalladas a continuación.

Alimentación de tornillos

Los tornillos se encuentran almacenados en un cargador vertical por gravedad. Todos ellos son descargados sobre un alojamiento a través de un sistema de alimentación paso a paso construido mediante dos cilindros neumáticos de doble efecto

trabajando de forma contrapuesta. En el momento que retrocede el de la parte inferior dejando caer el último tornillo, el de la parte superior avanza para sujetar a todo el resto. Una vez que el tornillo ha caído, ambos cilindros retornan a la posición original.

Módulo transvase

El alojamiento en el cual se depositan los tornillos, se encuentra situado sobre un cilindro neumático de vástagos paralelos y doble vástago. Este alojamiento cuenta con un detector de fibra óptica tipo barrera para verificar la presencia del tornillo. La construcción del alojamiento permite su fijación por las placas situadas en sus extremos, de forma que es el cuerpo del cilindro el que se desplaza a modo de carro. Dicho cilindro es utilizado para trasladar los tornillos desde el punto en que son alimentados, hasta un punto donde el siguiente módulo realizará su recogida para el posterior montaje sobre el conjunto.

Manipulador inserción tornillo

Una vez que los elementos anteriores han depositado un tornillo y lo han trasladado hasta el punto de recogida, se realiza la carga del mismo sobre uno de los orificios de la base del dispositivo de giro retenido en el transfer.

Este manipulador realizado en base a dos cilindros neumáticos de vástagos paralelos, presenta dos grados de libertad, correspondientes a los ejes horizontal y vertical. Como elemento terminal dispone de una pinza neumática de dos dedos de apertura paralela mediante la cual se realiza la sujeción de los tornillos.

5.10.2. Diagrama de clases

La versión OO de la estación 6 se compone de 12 clases y su diagrama de clases se muestra en la figura 5.119.

5.10.3. Programación OO vs programación estructurada

En esta subsección se mostrarán la alimentación de tornillos (proceso *"alimentador"*) y el proceso encargado de insertar los tornillos en los orificios de la base (proceso *"insertar"*).

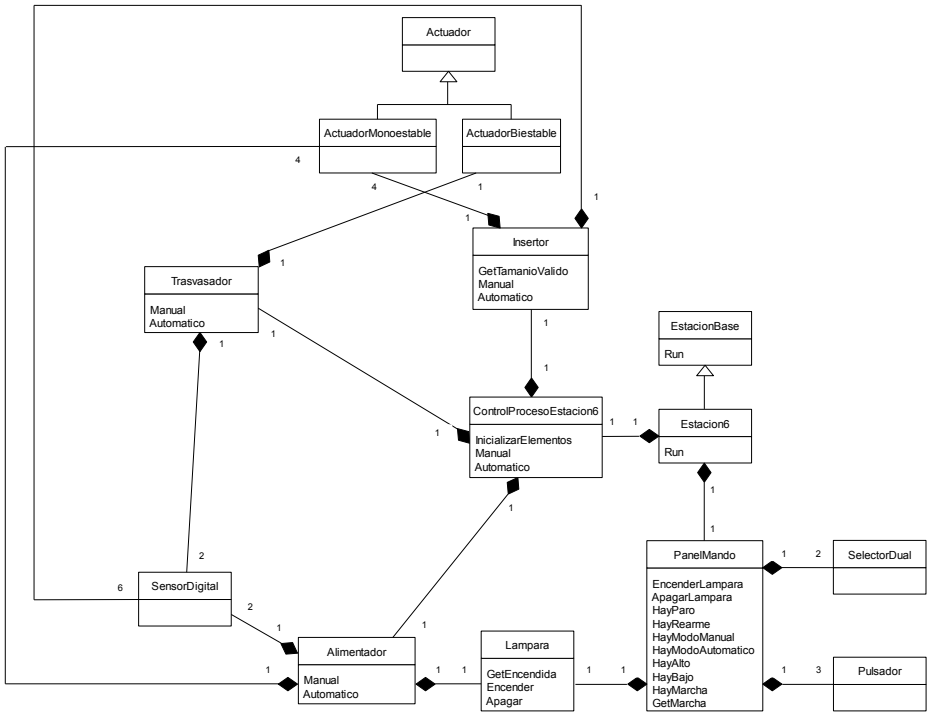

Figura 5.119: Diagrama de clases de la estación 6

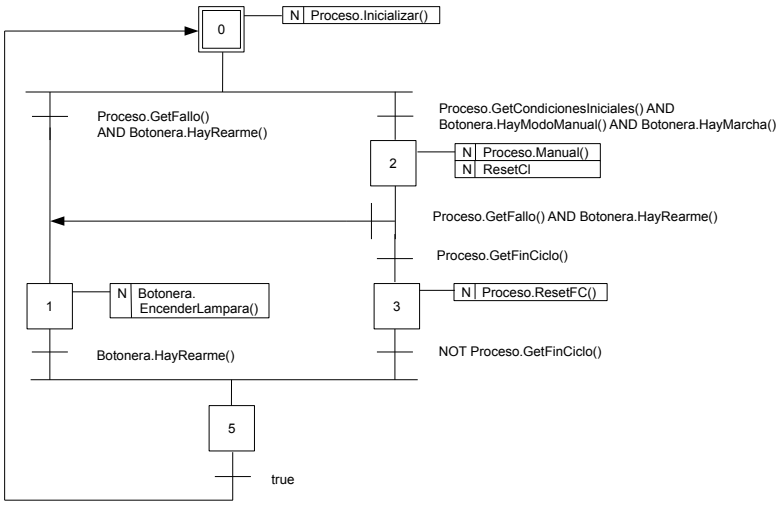

Figura 5.120: Método *"Run"* de la estación 6

Método *"Run"* de la clase Estacion6

En la figura 5.120 se muestra las ramas concernientes a la ejecución automática de la estación 6.

Proceso plato de la versión OO

La etapa 2 del método *"Run"* de la versión OO de la estación 6 (ver figura 5.120) tiene asociada una acción llamada al método *"Automatico"* de la clase *"ControlProcesoEstacion6"*. En la figura 5.121 se muestra el trozo de código del método *"Automatico"* correspondiente al proceso de alimentación de tornillos.

La parte interesante del método *"Automatico"* correspondiente a la acción de alimentación de tornillos se encuentra en la etapa 1 que tiene asociada una invocación al método *"Automatico"* de la clase *"Alimentador"* que es la encargada de realizar la alimentación de tornillos y que ha sido descrita en la sección 5.5.3 (ver figura 5.17).

Proceso alimentador de la versión estructurada

En la figura 5.122 se muestra un trozo del GRAFCET que controla la alimentación de tornillos.

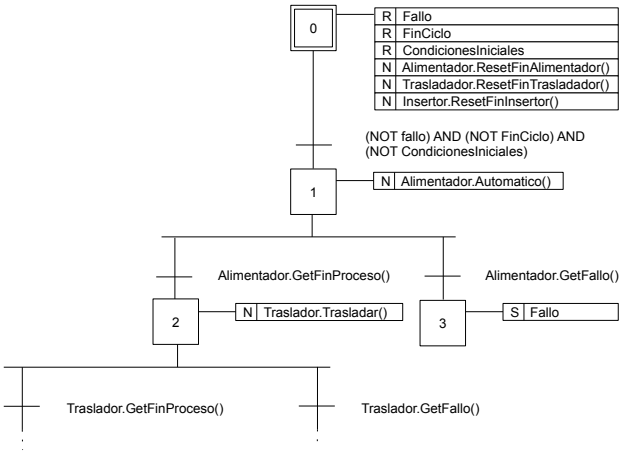

Figura 5.121: Método de ejecucion automática de ControlProcesoEstacion6

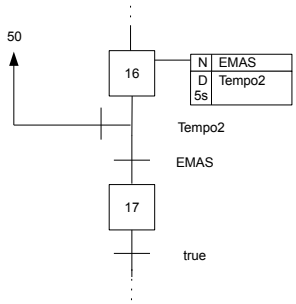

Figura 5.122: Proceso alimentador de la versión estructurada

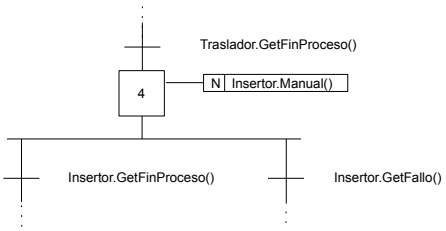

Figura 5.123: Método de ejecucion manual de la clase *"ControlProceso"* para el insertor

La etapa 16 tiene asociada la acción *"EMAS"* que se corresponde con la señal de avance del cilindro de traslado.

La etapa 50 tiene asociada la transición *"DEFECTO"* que se corresponde con el piloto luminoso por defecto de la estación.

Proceso insertar de la versión OO

En la figura 5.123 se muestra otro trozo del código correspondiente al método *"Automatico"* de la clase *"ControlProceso"*. Esta parte del código se encarga de insertar tornillos en los orificios de la base.

La acción de insercción de tornillos se corresponde con la llamada al método *"Manual"* de la clase *"Insertor"* que es la que realiza la tarea (ver figura 5.124).

Las acciones de la etapa 0 se corresponden con "*Fallo*","*CondicionesIniciales" y "FinProceso"* son atributos boolean de la clase.

La transición *"CilMono_ Bajar.Accionado"*, *"CilMono_ Cerrar_ pinza.Accionado*" y *"CilMono_ Avanzar.Accionado"* son llamadas al método *"Accionado"* de la clase *"ActuadorMonoestable"*.

Proceso insertar de la versión estructurada

En la figura 5.125 se muestra un trozo del GRAFCET que controla la bajada de la mampara de protección.

La etapa 26 tiene asociada la acción *"BMAS"* que se corresponde con la bajada del manipulador de inserción.

La etapa 27 tiene asociada la acción *"CMAS"* que se corresponde con el cierre de la pinza del manipulador de inserción.

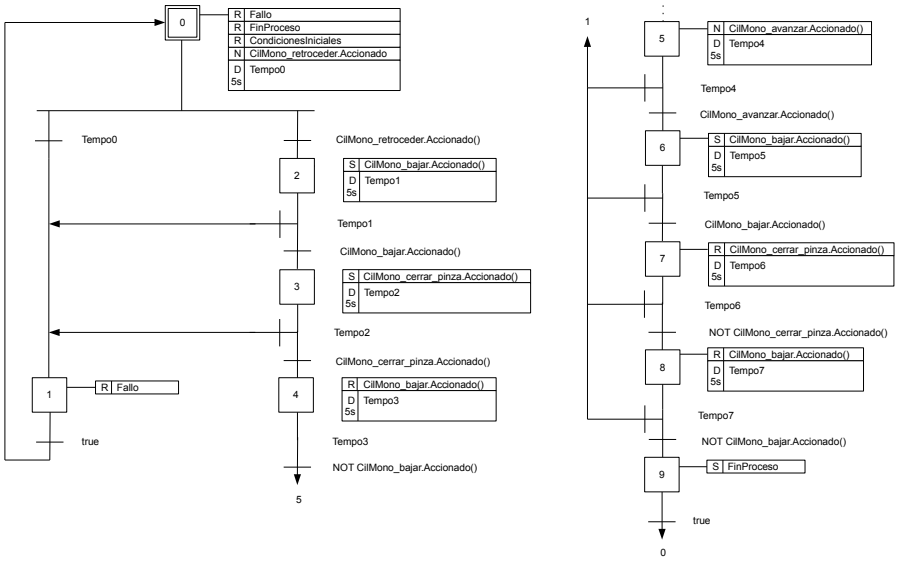

Figura 5.124: Método de ejecucion manual de la clase *"Insertor"*

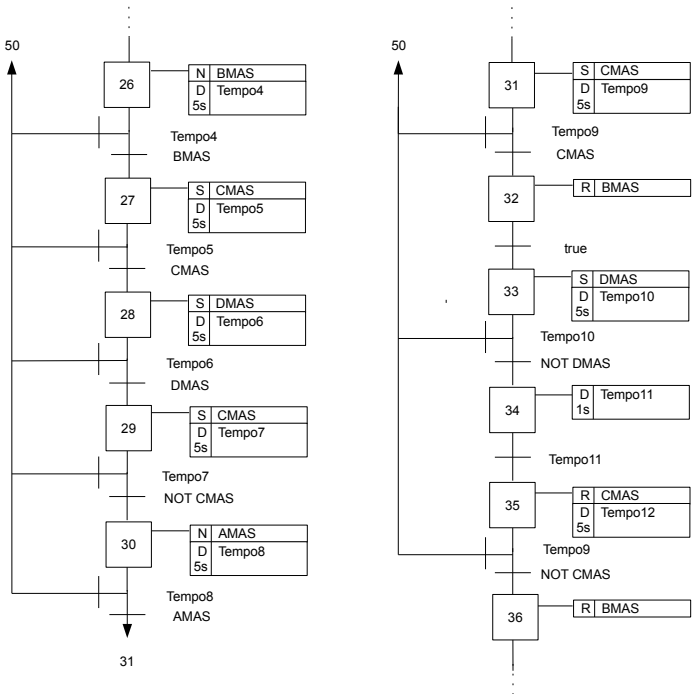

Figura 5.125: Proceso insertar de la versión estructurada

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Traslado	Inserción
2961	2.18	363	1006	280	4573
3100	2.17	369	1007	279	4558
2963	2.17	368	1010	282	4559
2971	2.17	371	1008	282	4565
2970	2.17	366	1008	279	4586
2972	2.18	368	1006	283	4601
3013	2.18	371	1007	280	4591
2943	2.18	364	1005	278	4553
2923	2.18	367	1007	280	4570
2937	2.17	370	1008	283	4572

Figura 5.126: Tiempos de ejecución de la estación 6 en su versión OO

La etapa 28 tiene asociada la acción *"DMAS"* que se corresponde con la alimentación de tornillos.

La etapa 30 tiene asociada la acción *"AMAS"* que se corresponde el avance del manipulador de insercción.

5.10.4. Resultados de tiempos del módulo TCLOCK

Los tiempos de ejecución de la estación de inserción de tornillos está medida tras la realización de 10 mediciones. En las tablas se muestran, por una parte el número de ciclos de SCAN que se han necesitado para completar un proceso completo, y por otro el tiempo de ejecución de cada proceso medido en milisegundos.

En la tabla 5.126 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma OO.

En la tabla 5.127 se muestran los tiempos de ejecución de las versión de la estación programada siguiendo un paradigma estructurado.

A continuación, se muestra de forma gráfica, usando los diagramas de caja, las cuartiles de los tiempos de ejecución de los distintos procesos que conforman la estación de inserción de tornillos, así como sus medias, desviación típica y covarianza de dichos tiempos.

En la figura 5.128 se puede observar el diagrama de cajas de los ciclos ejecutados por la estación para llevar a cabo la inserción de los tornillos.

Nº de ciclos	Duración media ciclo SCAN	Inicialización	Alimentación	Traslado	Inserción
3030	2.11	361	1004	280	4562
2960	2.11	360	1005	278	4555
3019	2.11	358	1003	280	4554
2976	2.11	358	1002	279	4556
3000	2.11	358	1005	278	4557
2990	2.11	354	1002	276	4551
2978	2.10	356	1005	278	4552
2996	2.11	359	1001	279	4553
3003	2.11	354	1002	277	4546
2984	2.11	356	1003	278	4547

Figura 5.127: Tiempos de ejecución de la estación 6 en su versión estructurada

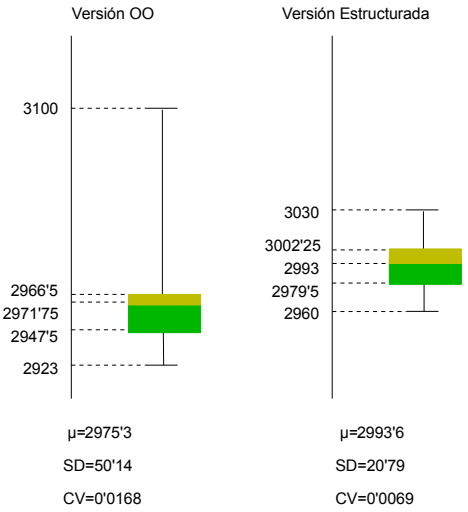

Figura 5.128: Diagrama de cajas del proceso *"número de ciclos"* de la estación 6

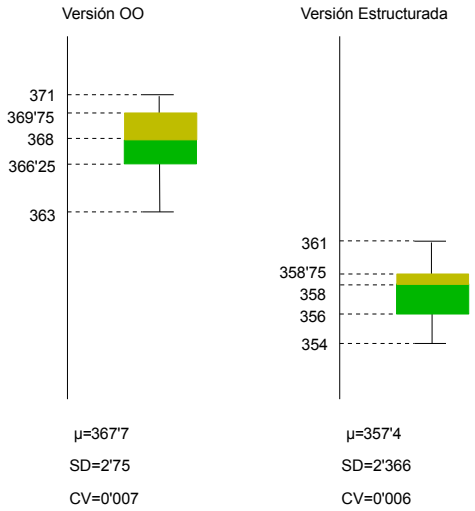

Figura 5.129: Diagrama de cajas del proceso *"Inicialización"* de la estación 6

En la figura 5.129 se muestra el diagrama de cajas del proceso que permite inicializar los objetos que conforman la estación dejando la posición inicial todos los elementos de la misma.

En la figura 5.130 se muestra el diagrama de cajas del proceso que saca los tornillos del cargador vertical por gravedad donde se almacenen.

En la figura 5.131 se muestra el diagrama de cajas del proceso que transfiere los tornillos desde el lugar de alimentación hasta la zona donde se realizará su recogida para el posterior montaje.

En la figura 5.132 se muestra el diagrama de cajas del proceso que inserta los tornillos en el conjunto.

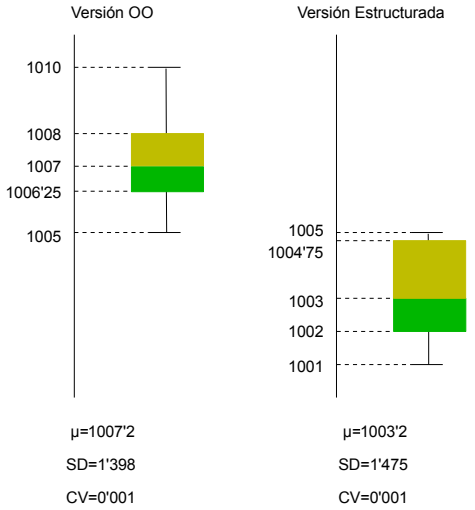

Figura 5.130: Diagrama de cajas del proceso *"Alimentación"* de la estación 6

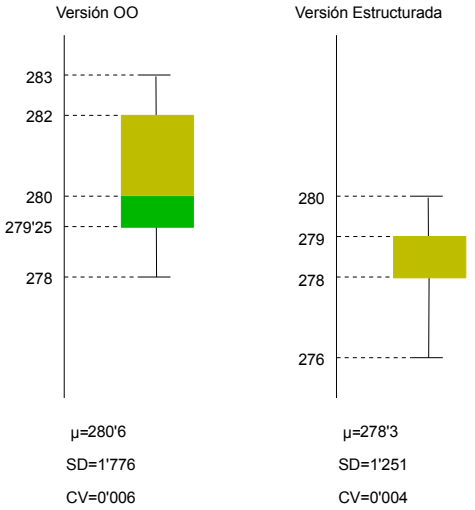

Figura 5.131: Diagrama de cajas del proceso *"Traslado"* de la estación 6

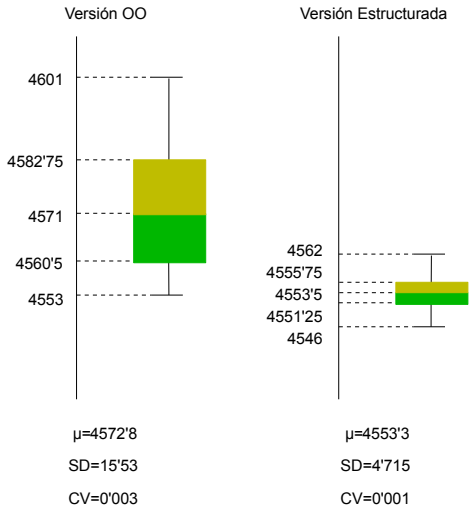

Figura 5.132: Diagrama de cajas del proceso *"Inserción"* de la estación 6

Capítulo 6

Discusión final y conclusiones

Una persona con una visión profunda evaluará tanto el inicio como el final de una situación y considerará continuamente cada faceta como importante.
Takeda Shingen

6.1. Introducción

A lo largo de este trabajo se ha mostrado el panorama actual en el que se encuentra el desarrollo de software en el campo de la Ingeniería de Automacización y las razones del salto evolutivo que existe, si se comparan las técnicas de programación para el desarrollo de programas de control, respecto a la Ingeniería de Software de propósito general en informática. Uno de los saltos más evidentes radica en la no utilización del paradigma orientado a objetos cuando se aplica el estándar IEC 61131 en la Ingeniería de Control. Es en este marco donde se engloba la aportación principal de esta memoria (ver capítulo 3), dando otro enfoque al estándar IEC 61131 y a los lenguajes de programación que en él se recogen, para dotar a la norma de los mecanísmos necesarios para realizar un desarrollo OO lo más cercano posible al usado en la informática de proposito general.

Este capítulo se divide en los siguientes apartados:

1. Las conclusiones que se obtienen tras la realización de los experimentos mostrados en el capítulo 5.
2. Las principales aportaciones que se derivan de las modificaciones propuestas por MIOOP al estándar IEC 61131.
3. Una discusión razonada de las principales desventajas y ventajas del uso de MIOOP.
4. Los futuros trabajos que se derivan de la presente memoria de tesis.

6.2. Conclusiones de los resultados experimentales del capítulo 5

En el capítulo 5 se ha mostrado un experimento que pretende mostrar la viabilidad de la aplicación de MIOOP en la automatización de procesos. En dicho experimento se ha desarrollado la programación de la célula de fabricación flexible FMS-200 siguiendo dos paradigmas de programación. El estructurado, soportado por la norma IEC 61131 y el OO, compatible con el estándar IEC 61131 ampliado por MIOOP. En este apartado se pretende analizar el resultado de la programación de la célula FMS-200 siguiendo los dos paradigmas. Pero dicho análisis debe hacerse desde dos puntos de vista: el objetivo y el subjetivo. Por un lado, se analizan los tiempos de ejecución totales de cada uno de los procesos que conforman la célula (análisis objetivo). Por otro lado, se analiza el estilo de programación usado cuando se aplica un paradigma estructurado u orientado a objetos (análisis subjetivo).

6.2.1. Análisis de los tiempos de ejecución

En los apartados dedicados a mostrar los tiempos de ejecución del capítulo 5, se presentan los tiempos de cada una de las etapas que conforman cada proceso del experimento. A continuación, se muestran los totales de ejecución de cada una de las estaciones por medio de los diagramas de cajas para que el lector pueda

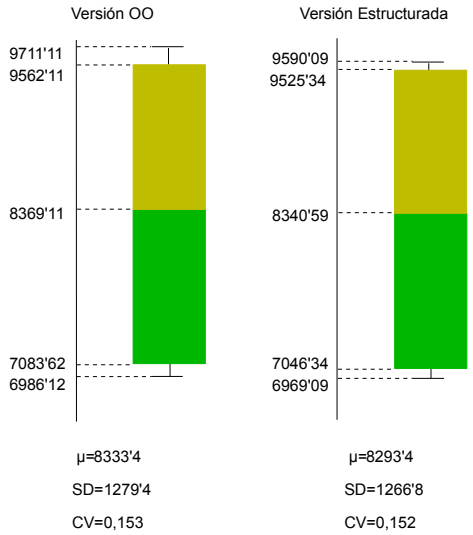

Figura 6.1: Diagrama de cajas del tiempo total de la estación 1

observar de forma visual los cuartiles de las muestras (incluyendo los mínimos, máximos, mediana, media, desviación típica y covarianza).

En la figura 6.1 se muestran los tiempos de ejecución de la estación de montaje de bases (estación 1). Los tiempos totales de las dos versiones son muy parecidos (diferencias inferiores 1 % para los cuartiles Q1 y Q3) y en ambas, la distribución de los valores obtenidos es proporcional, existiendo una desviación del tiempo total por debajo de la mediana del 51 % en la versión OO y del 52 % en la versión estructurada. En global, la versión orientada a objetos presenta una media de tiempos de ejecución total de un 0'09 % superior a los de la versión estructurada.

En la figura 6.2 se muestran los tiempos de ejecución de la estación de montaje de rodamientos (estación 2). Los tiempos totales de las dos versiones difieren en hasta 5012'43 milisegundos en el cuartil 1 y de hasta 1103'94 milisegundos en el caso del cuartil 3. Así mismo, la distribución de los valores obtenidos no es proporcional en las dos versiones. En la versión OO, sólo el 6 % del tiempo total de ejecución se encuentra por debajo de la mediana, mientras que en la versión estructurada, este valor decrece al 0'003 % del tiempo total de ejecución. En global, la versión estructurada presenta una media de tiempos de ejecución total de un 19 % superior a los de la versión orientada a objetos.

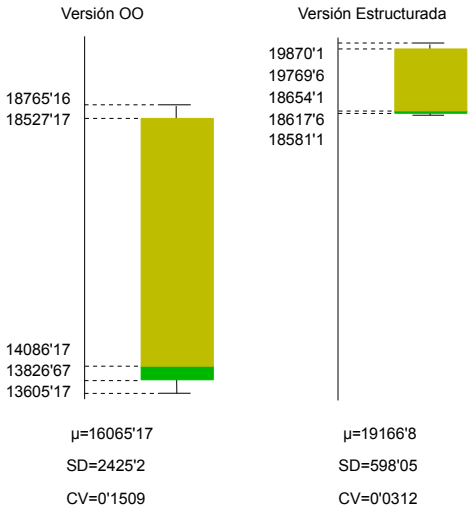

Figura 6.2: Diagrama de cajas del tiempo total de la estación 2

En la figura 6.3 se muestran los tiempos de ejecución de la estación de prensado (estación 3). Los tiempos totales de las dos versiones difieren en hasta 1276'75 milisegundos en el cuartil 1 y de hasta 1244'75 milisegundos en el caso del cuartil 3. Por otro lado, la distribución de los valores obtenidos es proporcional, existiendo una desviación del tiempo total por debajo de la mediana del 46 % en la versión OO y del 34 % en la versión estructurada. En global, la versión estructurada presenta una media de tiempos de ejecución total de un 8 % inferior a los de la versión orientada a objetos.

En la figura 6.4 se muestran los tiempos de ejecución de la estación de inserción de ejes (estación 4). Los tiempos totales de las dos versiones difieren en hasta 13607'46 milisegundos en el cuartil 1 y de hasta 17558'21 milisegundos en el caso del cuartil 3. Por otro lado, la distribución de los valores obtenidos es relativamente proporcional en la versión OO existiendo una desviación del tiempo total por debajo de la mediana del 32 %. Sin embargo, en el caso de la versión estructurada, la desviación del tiempo total por debajo de la mediana decrece hasta el 17 %. En global, la versión orientada a objetos presenta una media de tiempos de ejecución total de un 42 % inferior a los de la versión estructurada.

En la figura 6.5 se muestran los tiempos de ejecución de la estación de montaje de tapas (estación 5). Los tiempos totales de las dos versiones difieren únicamente

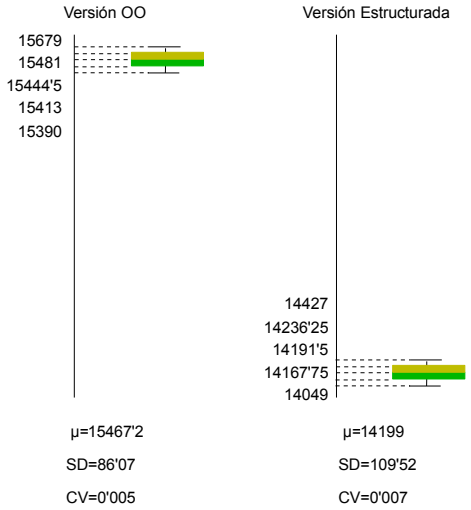

Figura 6.3: Diagrama de cajas del tiempo total de la estación 3

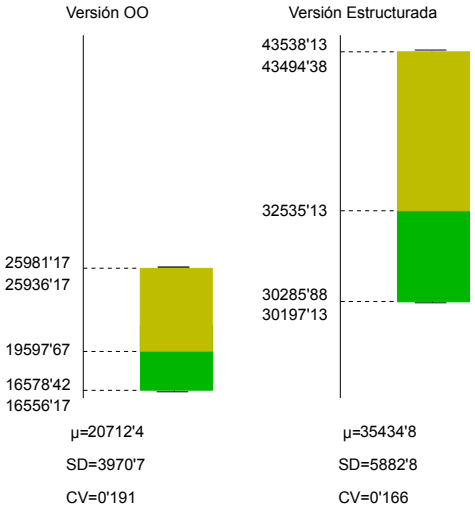

Figura 6.4: Diagrama de cajas del tiempo total de la estación 4

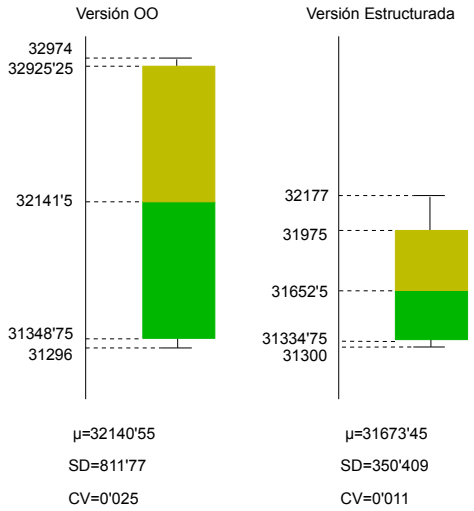

Figura 6.5: Diagrama de cajas del tiempo total de la estación 5

en 14 milisegundos en el caso del cuartil 1, sin embargo, en el caso del cuartil 3, las diferencias suben hasta los 950'25 milisegundos. La distribución de los valores obtenidos es proporcional en ambas versiones, existiendo una desviación del tiempo total por debajo de la mediana del 50'2 % en la versión OO y del 49'7 % en la versión estructurada. En global, la versión orientada a objetos presenta una media de tiempos de ejecución total de un 14 % superior a los de la versión estructurada.

En la figura 6.6 se muestran los tiempos de ejecución de la estación de inserción de tornillos (estación 6). Los tiempos totales de las dos versiones son bastante parecidos, siendo las diferencias inferiores al 1 % para los cuartiles Q1 y Q3. Respecto a la distribución de los valores obtenidos, la versión OO presenta una desviación del tiempo total por debajo de la mediana del 29 %, mientras que la versión estructurada muestra una mejor distribución de los valores arrojando una desviación de tiempos un 67 % por debajo de la mediana. En global, la versión estructurada presenta una media de tiempos de ejecución total de un 0'05 % inferior a los de la versión orientada a objetos.

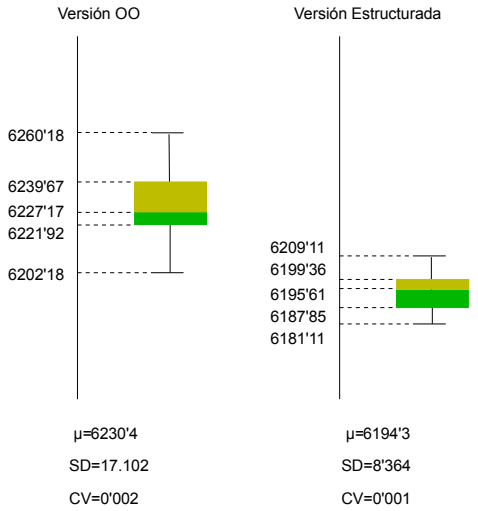

Figura 6.6: Diagrama de cajas del tiempo total de la estación 6

6.2.2. Análisis de los estilos de programación

A la vista de los resultados obtenidos por los tiempos de ejecución de los procesos que conforman cada estación de la célula flexible FMS-200, así como los tiempos totales de ejecución de cada estación de forma individual, no se puede decir de forma categórica que un paradigma de programación sea mejor que otro atendiendo únicamente a valores numéricos. Esta aseveración se sostiene en el hecho de que no existe un claro ganador si se analizan los tiempos de ejecución de las estaciones. En unos casos, los procesos OO presentan tiempos de ejecución inferiores a sus versiones estructuradas. En otros casos, se da la situación contraria. Lo mismo ocurre cuando se suman los tiempos de ejecución de todos los procesos que conforman cada estación.

Las desviaciones en los tiempos de ejecución que se observan entre las dos versiones de programación pueden tener un fondo más mental de los desarrolladores que únicamente empírico, en el sentido de que un paradigma proporcione unas ventajas en tiempos de ejecución respecto al otro. Aspectos tales como gustos de los ingenieros en las técnicas de implementación, capadidad de asimilación de los modelos del sistema, mayor familiaridad con un paradigma de programación, etc, pueden provocar que en el desarrollo de alguno de los procesos aparezcan más líneas de código siguiendo un paradigma que en otro. Por otro lado, y atendiendo

a la evolución de la electrónica y en concreto a la mejora que con el tiempo se vaya produciendo en el HW y sistemas operativos de los PLCs, es de esperar que aunque exista cierta diferencia de tamaño en los programas de control si se comparan los paradigmas de programación estructurado y OO, estas disparidad de tiempos resulte insignificante.

Otra medida que se puede tener en cuenta es la de la velocidad de programación propiamente dicha. Aunque el estudio del tiempo de desarrollo es un experimento que no se contempla dentro de esta memoria de tesis, atendiendo a los estudios realizados por Post [Pos01] se puede decir que una implementación OO produce sistemas más fáciles de diseñar, implementar y depurar, así como la aplicación de los pilares de la OO (encapsulación, herencia, polimorfismo, etc) permite una reutilización de código muy superior a la programación estructurada, lo que se traduce en un aumento de la productividad y una disminución de los costes de desarrollo.

Analizando el desarrollo de la programación de las estaciones siguiendo un paradigma OO, se puede llegar a la conclusión de que la implementación de la primera o las dos primeras estaciones, puede resultar algo más costosa que si se desarrollase de forma estructurada, debido principalmente a la identificación de clases y programación de las mismas la primera vez. Sin embargo, gracias a los pilares de la OO, éste trabajo previo reduce enormemente el tiempo de desarrollo de las otras estaciones al poder usar una biblioteca de objetos programados anteriormente.

6.3. Aportaciones

El conjunto de contribuciones que el presente trabajo aporta a la comunidad pueden ser divididas en tres categorías, aportaciones directas, indirectas y técnicas, que se enumeran en las siguientes secciones.

6.3.1. Aportaciones directas

En esta sección se detallan las aportaciones inmediatas y productos palpables del presente trabajo:

- La aportación de un nuevo enfoque orientado a obejtos a la hora de abordar la automatización de procesos.

- Un método que sistematiza las modificaciones necesarias que se deben realizar en el estándar IEC 61131 para dotar a todos sus lenguajes de una funcionalidad orientada a objetos.
- Una herramienta software que permite el desarrollo OO de programas de control con un 100 % de compatibilidad con el estándar IEC 61131.
- La definición de nuevos conceptos en la automatización de procesos como virtual, sobrecarga, clase abstracta, ligadura dinámia, etc.
- La aproximación al concepto de clase a partir del tipo de dato *"estructura"* aportado por el estándar.
- Un compendio exhaustivo de métodologías, lenguajes y sistemas de modelado de sistemas de procesos de uso en la actualidad.

Todo este conjunto de aportaciones directas genera a su vez un grupo de beneficios o aportaciones indirectas entre las que se puede citar las siguientes.

6.3.2. Aportaciones indirectas

La comunidad de ingenieros que potencialmente podrían beneficiarse de las aportaciones de este trabajo se pueden clasificarse en dos grupos bien diferenciados, a saber:

- Los ingenieros de automatización que trabajan en empresas de consultoría e ingeniería, cuyo principal cometido es la automatización de procesos mediante la reutilización de piezas de software previamente desarrolladas y funcionales.
- Los ingenieros docentes cuya principal tarea es la formación de futuros ingenieros en las técnicas de automatización industrial, y más concretamente en la programación de automatas.

Los beneficios que estos dos grupos de profesionales podrían obtener por la aplicación de las aportaciones de este trabajo se pueden agrupar en dos grandes grupos:

- Por un lado, los profesionales del mundo de la ingeniería podrían ver incrementado el nivel competitivo de sus empresas debido fundamentalmente a la disminución de los tiempos de desarrollo, y por tanto, de los costes, al aplicar MIOOP al desarrollo sus programas de control.

  El uso de MIOOP les permitiría una reutilización de código muy superior al que tendrían usando técnicas de programación estructurada. Es decir, el ingeniero de control se beneficiaría de piezas programas ya desarrolladas en el pasado.

- Por otro lado, los profesionales docentes podrían beneficiarse fundamentalmente de la utilización de la herramienta software SimPLC++ por dos motivos:

  1. Las características de la herramienta SimPLC++ haría que los alumnos se vieran más motivados a la hora de abordar la resolución de un problema de automatización. Ésto sería debido principalmente a la integración de SimPLC++ dentro del LAV y sus herramientas de simulación de procesos (ver anexo B), las cuales permiten un mayor nivel de concentración en el problema a resolver y son un mecanismo mucho más intuitivo que los usados en la actualidad.

     Esta motivación provocaría que el nivel de complejidad de los procesos a automatizar aumentase lo que a su vez permitiría alcanzar un mayor nivel de conocimientos por parte de los alumnos. Es decir, que indirectamente, la calidad de la enseñanza se vería mejorada por el uso de MIOOP.
  2. El uso de SimPLC++ permitiría al profesor mejorar las explicaciones de los ejercicios de automatización a los alumnos al poder hacer una analogía casi de 1 a 1 entre los elementos físicos del sistema de control y la lógica de control encargada de manejarlos.

6.3.3. Aportaciones técnicas

Desde el punto de vista técnico, MIOOP aporta una serie de beneficios a la hora de abordar proyectos de control de procesos:

- Descomposición de los sistemas en pequeñas partes más manejables (las clases) que se pueden refinar y mejorar de forma independiente.

- Acoplamiento de diferentes clases por medio de la herencia y la agregación, lo que permite componer nuevas clases con funcionalidades ya depuradas.
- Acceso a la funcionalidad de los objetos a través de los servicios proporcionados por éstos, lo que permite delimitar una clara separación de la lógica interna de los procesos por medio de interfaces (los métodos).
- Al desarrollar una clase, el programador tiene la capacidad de decidir el grado de acceso que proporciona a los servicios y datos de ésta (modelo de protección).
- Posibilidad de definir clases diferentes que tienen métodos denominados de forma idéntica pero que se comportan de manera distinta (polimorfismo). Ésta técnica permite desarrollar software de forma elegante, clara y ahorrando código, permitiendo hacer invocaciones de métodos sin conocer en tiempo de diseño que método debe ser llamado (ligadura dinámica).
- Declaración de clases que sirvan como esqueleto para sus clases derivadas (clases abstractas e interfaces). Este tipo de clases permiten al programador enfocarse en el diseño de clases sin prestar atención a su implementación. Serán las clases que hereden de éstas las encargadas de desarrollar sus métodos.
- Eliminación de las restricciones en la elección del nombre de los métodos (sobrecarga de métodos). MIOOP permite que varios métodos de una clase tengan un nombre idéntico diferenciandose únicamente en el tipo y número de sus parámetros.
- Ampliación de los operadores estándar de la norma IEC 61131 (suma, resta, multiplicación, etc) soportando operadores definidos por el programador (sobrecarga de operadores).

6.4. Discusión general

A continuación, se analiza de manera formal y razonada las desventajas y ventajas de la aplicación de MIOOP y en general, del paradigma de programación orientado a objetos en el desarrollo de proyectos de automatización desde varios puntos de

vista, como por ejemplo, la capacidad de generación de modelos, la reutilización de código, el tamaño del programa final, la velocidad de desarrollo, etc.

6.4.1. Desventajas de MIOOP

La aplicación de MIOOP tiene varias desventajas que se agrupan en dos tipos:

Desventajas desde el punto de vista de la lógica de control

MIOOP presenta varias desventajas desde un punto de vista del desarrollo de programas control:

1. MIOOP traduce los programas de control OO a sus versiones estructuradas basadas en el estándar IEC 61131-3 de forma automática. Esta traducción, al igual que ocurre con muchas herramientas de generación de código del mercado, puede producir un código complejo de entender. Este código traducido, desde el punto de vista de la Ingeniería de Automatización, donde el mantenimiento de los sistemas automatizados por lo general, está en manos de personal distinto al que lo desarrolló y con unos conocimientos limitados de técnicas de programación, así como la imposibilidad de acceder a un compilador de forma sencilla, puede convertirse en un auténtico problema. Este inconveniente puede paliarse en gran medida gracias a la herramienta de desarrollo proporcionada por MIOOP, de tal forma que las futuras modificaciones que los usuarios finales tuvieran que hacer, no se serían con la herramienta del fabricante, delimitando el problema a una cuestión de aprendizaje del paradigma OO y de manejo de SimPLC++.

2. Los programas de control obtenidos trás la traducción de un compilador basado en MIOOP, serán más grandes que si los mismos programas se hubiesen desarrollado empleando programación estructurada (principalmente debido a la aparición de nuevas estructuras usadas para la traducción). Los equipos de control más usados en el mundo de la automatización son los autómatas programables, los cuales por lo general suelen disponer de una cantidad de memoria limitada, lo que puede provocar que este aumento del tamaño del programa traducido sea problemático.
   Por otro lado, dados los potenciales beneficios que el análisis, modelado y la reutilización de código que la programación orientada a objetos pueden

proporcionar, el precio del hardware no debería ser una causa para detener su desarrollo o implantación. De hecho, fabricantes como Schneider proporcionan PLCs que pueden tener procesadores y memorias de gran capacidad, incluso suminstran económicos sistemas para ampliaciones de la memoria física para el alojamiento del programa de control.

3. El aumento del tamaño del código traducido y el uso de la ligadura dinámica[1], produce programas de control OO potencialmente más lentos que si se desarrollasen directamente siguiendo un paradigma estructurado. A pesar de ello, y a la vista de los resultados obtenidos de los tiempos de ejecución de los experimentos del capítulo 5, este impacto en la velocidad no es muy grave y en algunos casos, inexistente.

Desventajas desde el punto de vista de la programación orientada a objetos

MIOOP al estar basado en un paradigma orientado a objetos presenta los mismos inconvenientes que se aprecian en la Ingeniería del Software tales como:

1. La orientación a objetos proporciona a los ingenieros una poderosa herramienta para modularizar los sistemas en pequeñas partes (divide et vinces), pero puede darse el caso de producirse una excesiva atomización del proyecto lo que produce una mayor dificultad para comprender el problema de una forma global.

2. Muchas compañías de desarrollo entiende los beneficios que un sistema orientado a objetos puede proporcionarles e invierten gran cantidad de recursos para luego comenzar a darse cuenta que han impuesto una nueva cultura que es ajena a los ingenieros programadores que poseen en sus plantillas. Específicamente los siguientes problemas suelen aparecer repetidamente:

    - Curvas de aprendizaje largas. Un sistema orientado a objetos ve al mundo en una forma única. Involucra la conceptualización de todos los elementos de un programa, desde subsistemas a los datos, en la forma de objetos. Toda la comunicación entre los objetos debe realizarse en

[1] Una llamada a un FB a través de un puntero resulta más lenta que realizar directamente la invocación del FB debido a la resolución de la dirección de memoria del POU en el segmento de código (ver apartados 3.4 y 3.21 del capítulo 3).

la forma de mensajes. Al hacer la transición de un sistema estructurado a uno orientado a objetos, la mayoría de los programadores deben capacitarse nuevamente antes de poder aplicarlo.

- Dependencia del lenguaje. A pesar de la portabilidad conceptual de los objetos en un sistema OO, en la práctica existen muchas dependencias. Muchos lenguajes orientados a objetos están compitiendo actualmente para dominar el mercado. Cambiar el lenguaje de implementación de un sistema orientado a objetos no es una tarea sencilla. Por ejemplo, C++ soporta el concepto de herencia múltiple mientras que SmallTalk no lo soporta. En consecuencia, la elección de las características orientadas a objetos que un lenguaje soporta tiene implicaciones de diseño muy importantes.
- Determinación de las clases. Una clase es un molde que se utiliza para crear nuevos objetos. En consecuencia, es importante crear el conjunto de clases adecuado para un determinado proyecto. Desafortunadamente, la definición de las clases es más un arte que una ciencia. Si bien hay muchas jerarquías de clases predefinidas, usualmente se deben crear clases específicas para el proyecto que se esté desarrollando. Luego, en 6 meses ó 1 año se puede dar el caso que las clases que se establecieron no son posibles; en ese caso será necesario reestructurar la jerarquía de clases devastando totalmente la planificación original.

Los problemas derivados por la aplicación/adopción de la orientación a objetos son de difícil solución y se salen del ámbito del problema al que pretende dar solución MIOOP. La adopción de métodologías OO, sistemas de modelado OO, formación de las plantillas y en general, la aplicación de una buena Ingeniería de Software a la automatización de procesos, debería solventar en gran medida estos inconvenientes.

6.4.2. Ventajas de MIOOP

Asimiliación y generación de modelos

La transición entre las fases de análisis y diseño en la orientación al objeto es mucho más suave que en las métodologías estructuradas, no habiendo tanta diferencia entre las etapas [PF96]. Es difícil determinar dónde acaba el análisis orientado a

objetos (AOO) y dónde comienza el diseño orientado a objetos (DOO), siendo la frontera entre ambas totalmente inconsistente, de forma que lo que algunos autores incluyen en el AOO otros lo hacen en el DOO. El objetivo del AOO es modelar la semántica del problema en términos de objetos distintos pero relacionados. Por su parte, el DOO conlleva reexaminar las clases del dominio del problema, refinándolas, extendiéndolas y reorganizándolas, para mejorar su reutilización y tomar ventaja de la herencia. El análisis casa con el dominio del problema y el diseño con el dominio de la solución; por lo tanto el AOO enfoca el problema en los objetos del dominio del problema y el DOO en los objetos del dominio de la solución.

Se puede definir el AOO como *"el proceso que modela el dominio del problema identificando y especificando un conjunto de objetos semánticos que interaccionan y se comportan de acuerdo a los requisitos del sistema*" [Mon92].

Se puede definir el DOO como *"el proceso que modela el dominio de la solución, lo que incluye a las clases semánticas con posibles añadidos, y las clases de interfaz, aplicación y utilidad identificadas durante el diseño"* [Mon92].

La aplicación de MIOOP en el desarrollo de programas de control y por ende, el uso del paradigma orientado a objetos, permite el uso de modelos OO en las primeras fases de los proyectos. Entre estos modelos, cabe destacar el lenguaje de modelado UML, que como demostró Douglas[Dou99], permite una escalabilidad y definición de los proyectos mucho más elevada que si se usaran técnicas de documentación clásicas, consiguiendo desarrollos más rápidos (50 % o más), eficaces y baratos.

Por otro lado, el carácter jerárquico de los modelos obtenidos, divide el problema global en sucesivos niveles de decreciente complejidad lo que facilita la asimilación de la estructura de la lógica de control por personas ajenas a aquellas que la desarrollan. Así mismo, el hecho de que los objetos se inspiren en los objetos reales que forman el proceso que se desea controlar, constituye un factor fundamental que propicia la caída en la pendiente de la curva de asimilación del proyecto por parte del grupo de trabajo.

La orientación a objetos ya ha demostrado sus benefícios en la disciplina de Ingeniería del Software, pero en la disciplina de Ingeniería de Automatización será aún más palpable ya que el mantenimiento de los sistemas automatizados por lo general está en manos de personal distinto al que lo desarrolló y con unos conocimientos limitados de técnicas de programación.

Por tanto, cuanto menos abstracto sea el programa de control, es decir, cuanto

más próximo a la realidad esté, más fácilmente será asimilado.

Reutilización de código

El término reutilización fue originalmente postulado por M.D. McIlroy en la conferencia de la NATO de 1968 sobre Ingeniería del Software [McI76], y desde entonces la reutilización del software ha sido, y sigue siendo, uno de los principales temas de investigación en el campo de la Ingeniería del Software, citándose a menudo como una de las principales técnicas para incrementar la productividad de los desarrolladores de software. Así Mili et al. [MM95] llegan a afirmar que:

> La investigación en estas décadas en los campos de la Ingeniería del Software y de la Inteligencia Artificial ha dejado algunas alternativas, pero la reutilización es la *"única"* aproximación realista para llegar a los índices de productividad y calidad que la industria del software necesita.

Aunque por otra parte, este mismo autor reconoce que no se han conseguido grandes avances en la adopción sistemática de la reutilización en el proceso de construcción del software.

El propósito de la reutilización es mejorar la eficiencia, la productividad y la calidad del desarrollo software. Así, la reutilización puede definirse como *"cualquier procedimiento que produce o ayuda a producir un sistema mediante el nuevo uso de algún elemento procedente de un esfuerzo de desarrollo anterior"* [Fre87] o como *"la utilización de elementos software existentes durante la construcción de un nuevo sistema software"* [Kru92].

Los procesos físicos se constituyen sobre la base de la técnica de reutilización de elementos básicos (motores, sensores, actuadores, etc) ensamblados adecuadamente para cumplir con la función deseada. Uno de los principales beneficios de la tecnología orientada al objeto es la reutilización de código. Las clases proporcionan los mecanismos de encapsulación, abstracción y ocultación de la información, además de ser un componente elemental en la reutilización. Las clases aportan los mecanismos de reutilización en dos niveles: por un lado, como representación de una abstracción de diseño que se puede extender o especializar, y por otro lado, como fábrica de objetos que comparten la estructura y el comportamiento definido por la clase. La herencia es la propiedad fundamental que permite que nuevas

clases puedan compartir comportamiento y representación a partir de las clases existentes. Es el concepto del paradigma objetual sobre el que se asientan la reutilización y la extensibilidad del software. A través de la herencia, los ingenieros de desarrollo pueden construir nuevos elementos software sobre una jerarquía de elementos existentes, permitiendo abordar el proceso de diseño y construcción del software sin tener que partir de cero [Mar95].

La herencia múltiple es la segunda vía para crear nuevas clases a partir de las existentes. Es posible declarar clases derivadas de las existentes especificando que heredan los miembros de una o más clases antecesoras. Este sistema tiene también sus ventajas e inconvenientes, pero es muy flexible, ya que incluso pueden cambiarse los componentes en tiempo de ejecución.

La tercera vía para la reutilización de código es la agregación y la composición. Este tipo de relación entre clases permite componer objetos a partir de otros más sencillos, de modo que la colección completa representa un todo. Las relaciones de agregación/composición se especifican entre clases y se reflejan en instancias de objetos.

Compatibilidad con la norma

La mayoría de los PLCs comerciales no incorporan ningún tipo de lenguaje de programación orientado a objetos. Ésto implica que el modelo de la lógica de control orientado a objetos de un programa siguiendo el paradigma OO debe ser traducido o compilado a alguno de los lenguajes de programación no orientado a objetos del PLC que se vaya a emplear para automatizar el proceso.

MIOOP aporta la herramienta SimPLC++ como una solución intermedia que pasa por añadir una ampliación de los cinco lenguajes proporcionados por el estándar IEC 61131-3 dotandolos de características orientadas a objetos. Cada uno de estos lenguajes (denominados en SimPLC++ como IL++, ST++, LD++, FBD++ y SFC++), soportaría las características aportadas por MIOOP en el capítulo 3 (ver figura 6.7) y serían traducidos a los 5 lenguajes soportados por el estándar IEC 61131, a saber IL, ST, LD, FBD y SFC.

Las ampliaciones propuestas por MIOOP son empleadas solamente para representar las características orientadas a objetos que al compilarse son traducidas a estructuras de datos y módulos de programa adicionales. Entre estas características se pueden enumerar la declaración de objetos, la ligadura dinámica, el mapeo

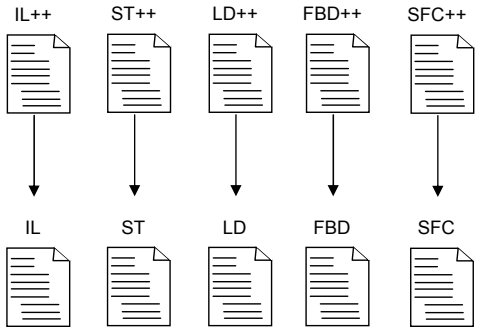

Figura 6.7: Traducción de lenguajes OO a sus versiones no OO

de atributos de instancias de objetos a señales físicas de entrada o salida del equipo de control, etc.

Sin embargo, los cinco lenguajes de la norma pueden seguir siendo empleados para implementar las estructuras de control propiamente dichas. Por ejemplo, un servicio de un objeto podría ser implementado empleando el lenguaje SFC o IL añadiendo características estructuradas a la lógica de control orientada a objetos, es decir, SimPLC++ no limíta la mezcla de módulos desarrollados bajo un primas OO con módulos propios de la norma IEC 61131.

De esta forma, el programa de control final posee la misma estructura que propone MIOOP al mismo tiempo que los módulos que realmente hacen el control del proceso están programados en los lenguajes en los que los ingenieros de automatización y el personal de mantenimiento están ya acostumbrados y conocen.

6.5. Estudios futuros

En este apartado se relatan una serie de futuros trabajos de investigación que se derivan de la actual memoria de tesis, a saber, sistemas dinámicos, sistemas distribuidos y agentes fuzzy.

6.5.1. Sistemas dinámicos

Las clases y objetos descritos en este trabajo tienen un funcionamiento estático, es decir, se conoce el número de objetos que se van a utilizar en tiempo de diseño y

su número no varía a lo largo del ciclo de ejecución del programa de control. Ésto es debido a que el mundo industrial es muy predecible y se conocen a priori las necesidades de un programa, es decir, se sabe de antemano el número de válvulas, el número de sensores, etc. Este concepto de OO estático presentado por MIOOP es suficiente para desarrollar programas de control orientados a objetos, pero existe, por otro lado, otro tipo de sistemas en los que un modelo estático se antoja insuficiente. Este tipo de sistemas pueden variar en el número de objetos (sensores, actuadores, etc) a lo largo del ciclo de vida del programa de control y deben ser capaces de adaptarse sin tener que volver a reescribirse el código. Por ejemplo, un modelo dinámico resultaría útil para desarrollar un programa que controle la fabricación de distintos tipos de componentes mediante la reordenación del flujo de los materiales a lo largo de las distintas máquinas que componen el proceso en función de las necesidades de manipulación que cada una precise.

Se entiende un sistema dinámico, como aquel sistema complejo que presenta un cambio o evolución de su estado a lo largo del tiempo. En computación, un sistema orientado a objetos dinámico es aquel que permite crear y destruir objetos en tiempo de ejecución. Para poder gestionar la creación y destrucción de objetos se necesita un espacio de memoria destinado para este fin. En informática, a dicho espacio se le conoce como *"Heap"* o espacio dinámico.

Normalmente la memoria se organiza de una forma similar a como se muestra en la figura 6.8.

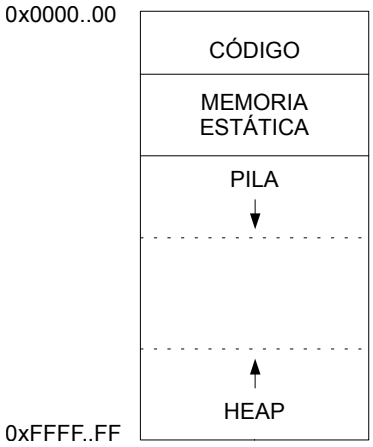

Figura 6.8: Organización de memoria

Es habitual que la pila crezca *"hacia abajo"* en programas informáticos debido a que muchos procesadores tienen instrucciones que facilitan la creación de pilas *"descendentes"*. Dado que la pila puede colisionar con el *"Heap"*, el compilador debería generar código para que en caso de colisión el programa pueda pedir memoria adicional al sistema o terminar adecuadamente. Por otro lado, la norma IEC 61131 no permite el uso de la recursividad, con lo que no existiría el riesgo de que la pila crezca de forma inesperada, pero tampoco permite el manejo del espacio de memoria de *"Heap"*, con lo que habría que desarrollar estructuras de datos que implementen dicho espacio de memoria.

Para indicar al compilador que se desea implementar la zona de memoria dinámica, se debería agregar algún tipo de instrucción especial al código que indique al compilador donde comienza y acaba la memoria dinámica simulada, como por ejemplo *"SHeap"* (Heap simulada). *"SHeap"* sería una palabra reservada que funcionaría como una variable global almacenando el comienzo de la memoria dinámica en la memoria física del PLC. Toda variable que se mapee después del rango asignado a *"SHeap"* produciría un error en tiempo de compilación, ya que el límite por arriba de memoria dinámica asignada a un programa lo limita el propio PLC y su memoria física (ver algorítmo 6.1).

Algoritmo 6.1 Error al acceder a una posición incorrecta de la memo

```
SHeap #10000    //Comienzo de la memoria dinámica
Valvula1 AT #200   //OK
Sensor1 AT #10050
//Error. No se puede mapear una variable dentro de SHeap.
```

De esta forma, todas las variables que se asocien mediante un puntero a la zona de memoria reservada para *"SHeap"* podrían simular el uso normal del *"Heap"* en informática.

En conclusión, una posible línea de investigación consistiría en decidir cuál es la mejor estrategia para implementar una memoria dinámica que se adapte al estándar IEC 61131 y las consecuencias que de esta se derivan.

6.5.2. Gestión de memoria dinámica

Cuando se mapea una variable a una posición física de un PLC, hay que tener en cuenta el tamaño de dicha variable. Esta acción resulta fácil al manejar un número limitado de variables y ser este constante. La dificultad surge cuando no se sabe a priori qué variables se van a almacenar en *"SHeap"* y se necesita realizar una gestión del tamaño que van a ocupar dichas variables en tiempo de ejecución.

- Una aproximación podría consistir en que cada clase implemente sus propios métodos para gestionar sus inserciones dinámicas, es decir, si se entiende la memoria como una pila, cada objeto poseería los métodos *"push"* y *"pop"* para insertar y sacar un dato de la memoria respectivamente. El punto fuerte de esta solución sería el de que cada objeto encapsularía los mecanismos necesarios para autogestionar el modelo dinámico. Por el contrario, esta solución aumentaría el tiempo de compilación y el tamaño del código traducido de MIOOP a IEC 61131.

- Otra posible solución pasaríaa por sacar fuera de las clases el manejo de la memoria dinámica, traspasando dicha gestión a una segunda clase. Ésta sería la encargada de manejar las inserciones de variables en la memoria, eliminación de variables de la memoria bien porque se destruye un objeto o por petición expresa del programador, y la gestión eficiente de la memoria conociendo el estado de ésta en todo momento y sabiendo en cada instante que dato eliminar o en qué posición insertar un elemento. Esta solución resultaría más compleja de implementar y comprender, pero permitiría una clara diferenciación de los objetos respecto de la memoria, ofreciendo ésta, servicios de inserción, consulta y eliminación de elementos en la misma.

La memoria física de un PLC se divide en celdas de bytes, por lo que un elemento mayor no puede ser ubicado en una única celda. Cuando se direccionan las variables, este problema es resuelto por el programador asignando memoria a una variable dependiendo de su tamaño o por el compilador en tiempo de compilación asignando espacios de memoria inferiores a *"SHeap"*. El problema surge cuando se realiza una operación sobre la memoria de *"SHeap"* (inserción, eliminación o consulta). En este caso, es el gestor de memoria dinámica el encargado en tomar las decisiones de localización y ubicación de cada variable atendiendo a su tamaño.

La mejor forma en que el gestor de memoria de *"SHeap"* pudiese ofrecer servicios, sería envolviéndolo en una capa de abstracción, es decir, en una clase encargada de las operaciones de inserción, consulta y eliminación de elementos, así como de la gestión de los huecos que debe albergar cada elemento que se insertase. A esta capa encargada de la gestión de *"SHeap"* se la podría llamar *"MemoryManager"*.

Esta nueva clase *"MemoryManager"* debería contener dos grupos de servicios:

1. Los servicios de conversión de un tipo primitivo de dato (entero, carácter, etc) a byte, debido a que estos son los únicos datos validos en cada una de las celdas.
2. Los servicios de operación sobre la memoria, entre los que se encontrarían:
   - Servicio de inserción de un byte en una posición de la memoria.
   - Servicio de eliminación de un elemento de la memoria.
   - Servicio de consulta de una posición dada de la memoria. Este servicio devolvería el dato alojado en la posición que se le pasa como parámetro en el formato correcto (entero, float, string, etc).

Los servicios de conversión tendrían que ser una serie de métodos que dividen el tipo de variable recibido como parámetro de entrada, a una ristra de bytes que se alojaran en celdas de la memoria de *"SHeap"* de forma consecutiva. Las reglas de conversión deberían ser:

Algoritmo 6.2 Conversión de entero a byte

```
Byte1 := Entero / 256;  //byte más significativo
Byte2 := MOD(Entero, 256);  //byte menos significativo
```

- Los tipos primitivos de variables simples se componen de n bytes. Así, por ejemplo, un entero es el resultado de la concatenación de dos bytes.
  Cuando se solicitase un servicio de conversión de un tipo simple a byte, se invocaría un algoritmo de conversión característico para cada tipo de dato. Por ejemplo, un entero se convertiría a byte por medio del algoritmo 6.2.
- Los strings son un caso especial de tipo simple de dato pero cuyo conversión a bytes no es muy diferente. Cada uno de los n-1 elementos que forman el

> string es un carácter (que se implementa con un byte) y el primer elemento del string es un entero que identifica la longitud del mismo. De esta forma, el primer byte que se almacenase en la memoria sería el de la longitud del string y en los consecutivos (tantos como marque su longitud) se almacenaría el código ASCII traducido a byte de cada elemento.

Estos métodos servirían como apoyo a los servicios de inserción y eliminación. Del mismo modo, el gestor de memoria dinámica también tendría que poseer los métodos que transformasen un byte en otro tipo de elemento, convirtiendo una ristra de bytes a un único tipo de variable. Este proceso es inverso a la separación en bytes de un tipo de dato y por tanto usarían el mismo algoritmo de separación pero en diferente orden.

Los servicios de operación sobre la memoria consistirían en tres métodos que operarían directamente sobre la memoria.

- Método insert.
- Método delete.
- Método element.

El método *"element"* recibiría una posición de memoria y el tipo de elemento que se desea recibir. Internamente tendría que acceder a la posición de memoria y convertiría la ristra de bytes alojados en dicha posición al elemento que se le pide utilizando los servicios de conversión de la propia clase *"MemoryManager"*.

El método *"delete"* recibiría una posición de memoria y un tipo de elemento, y dependiendo de éste, eliminaría de la memoria dinámica tantos bytes como ocupe ese tipo de dato a partir de la posición que se le pasa al método.

El método *"insert"* recibiría una variable para que se alojase en la memoria, dividiéndola en bytes por medio de los servicios de conversión y devolviendo la posición de memoria donde ha ubicado el bit más significativo.

En conclusión, la línea de investigación que se abriría a partir de este punto se basaría en definir la mejor forma en la que se puede implementar el gestor de memoria, qué servicios tendría que soportar y cómo se debería acoplar en los sistemas de control basados en IEC 61131.

6.5.3. Estrategias de operación sobre la memoria

A la hora de implementar la clase *"MemoryManager"* se podrían utilizar dos tipos de estrategias. El tipo de implementación escogida dependería para qué tipo de PLC se desean hacer los programas de control y las restricciones tanto de tiempo y de espacio (en tamaño del programa) que se tuvieran.

Gestión explicita de la memoria

En este tipo de estrategia, es el programador el encargado de llamar explícitamente a una función para liberar los bloques de memoria. Existirían tres métodos fundamentalmente para poder realizarlo:

- Lista ordenada de bloques libres.
- Bloques etiquetados en los extremos.
- Bloques compañeros.

En el primer método, se utilizaría una lista de bloques libres ordenada por la dirección inicial del bloque, en la que cada bloque tiene una cabecera con su tamaño y un puntero al siguiente bloque de la lista. Esta lista serviría para conocer el estado de *"SHeap*" en todo momento, ya que una inserción o borrado en *"SHeap"* se realizaría sobre esta lista ordenada también.

Figura 6.9: Bloque libre de lista ordenada de bloques libres

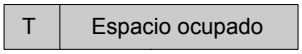

Figura 6.10: Bloque ocupado de lista ordenada de bloques libres

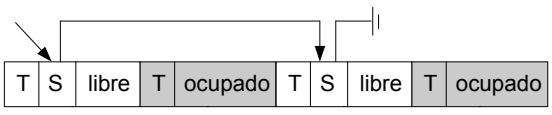

Figura 6.11: Lista ordenada de bloques libres

Un bloque libre tendría la estructura que se muestra en la figura 6.9. Un bloque ocupado tendría la estructura que se puede observar en la figura 6.10. Una lista ordenada de bloques tendría la estructura que indica la figura 6.11.

El problema de las listas ordenadas de bloques en una lista es que hay que realizar búsquedas secuenciales para pedir memoria y liberarla, produciendo fragmentación en *"SHeap"*.

El segúndo método (bloques etiquetados en los extremos) reduciría el problema de las búsquedas secuenciales, modificando la lista de bloques anterior e implementándola como una lista doblemente enlazada, guardando información en cada bloque sobre si los bloques contiguos están o no libres.

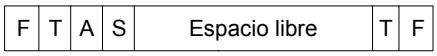

Figura 6.12: Bloque libre de bloques etiquetados en los extremos

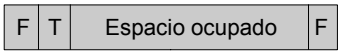

Figura 6.13: Bloque ocupado de bloques etiquetados en los extremos

Un bloque libre tendría la estructura que se muestra en la figura 6.12, donde *"F"* indica el flag libre / ocupado de los bloques contiguos, *"T"* el tamaño del bloque, y *"P"* y *"S"* son punteros a los nodos adyacentes. Un bloque ocupado tendría la estructura que se puede observar en la figura 6.13.

El tercer método (bloques compañeros) permite eliminar las búsquedas al pedir y liberar memoria. Para ello, los bloques tienen que ser familias de tamaño 2n Bytes, con lo que la búsqueda de memoria quedará reducida a un ARRAY de listas de bloques libre de 32 posiciones como máximo.

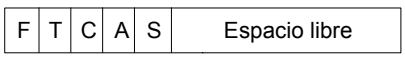

Figura 6.14: Bloque libre de bloques compañeros

Un bloque libre tendría la estructura que se observa en la figura 6.14, donde *"C"* indica el código del bloque. Un bloque ocupado tendría la estructura que se muestra en la figura 6.15.

Figura 6.15: Bloque ocupado de bloques compañeros

Este algoritmo sería más rápido (debido al tiempo de las búsquedas) que los dos primeros métodos, pero solo se aplica a sistemas con memoria virtual que permiten reservar espacio de direcciones sin reservar memoria física. Además, estos algoritmos al liberar memoria producen un problema de fragmentación de los bloques libres que ocurre cuando la lista contiene dos bloques libres contiguos. Ésto sería erróneo y el sistema debería fusionar todos los bloques libres que se encontrasen juntos.

Gestión de memoria implícita

En este tipo de estrategia, es el gestor de memoria el encargado de liberar los bloques de memoria que no utilice el programa. La ventaja de esta estrategia sería que el programador podría olvidarse de la gestión de la memoria con lo que se produciría un desarrollo más rápido y se reducirían los errores en la programación. La desventaja que presentaría esta estrategia sería que produciría unos métodos lentos y desaprovecharían la memoria.
Existirían dos métodos fundamentalmente para implementar esta estrategia:

- Contenedores de referencias.
- Recolectores de basura.

Los dos métodos se basarían en que cuando un bloque no posee un puntero que lo referencia quiere decir que el programa no lo utiliza y por tanto podría ser liberado automáticamente.

El método por contador de referencia se basaría en contar el número de punteros que referencian cada bloque de memoria de *"SHeap"*. Cada bloque tendría un contador de referencias y se conocerían los punteros que contiene. Cada creación, modificación o destrucción de un puntero a una variable de *"SHeap"* supondría actualizar los contadores de los bloques a los que apunta. Cuando el contador de un bloque llegase a cero, éste se liberaría automáticamente. Este método es el empleado para liberar memoria en C++. El problema de esta estrategia radicaría al liberar de memoria estructuras cíclicas.

La idea para la implementación de los recolectores de basura se basaría en que el programa pediría memoria al sistema hasta que no quedase más. En este momento, se activaría el recolector de memoria que liberaría la memoria que no utilizase el programa.

El recolector de basura tendría que poder recorrer toda la memoria de *"SHeap"* que estría usando el programa y detectar que partes no están siendo referenciadas por un puntero.

Para su implementación, se deberían usar datos encapsulados en nodos o celdas para facilitar la identificación de los apuntadores en su interior. Cada nodo o celda tendría que poseer un descriptor de su contenido. Los punteros que no se encuentrasen referenciados en *"SHeap*" se considerarían como punteros externos. El algoritmo encargado de localizarlos se llama *"marcar y barrer*" y actuaría en dos pasos:

1. Marcar. Marca todos los bloques a los que tiene acceso el programa.
2. Barrer. Libera los bloques no marcados.

En el caso de implementar esta estrategia, *"SHeap"* debería variar su estructura primitiva para que cada bloque de memoria contuviera la siguientes estructuras:

- Descriptor del bloque.
- Marca.
- Apuntadores.
- Otros datos.

Un bloque libre tendría un descriptor que indicaría que es libre y los datos que necesita el gestor de memoria explícito para poder pedir bloques (lista de libres). Cuando se pidiera un bloque de memoria libre, se tendría que especificar de qué tipo es la variable (descriptor) y su tamaño. Se buscaría el bloque en una lista de libres, tal y como se haría con un gestor de memoria explicito, y al encontrar el bloque libre se inicializaría éste y se devolvería un puntero a su posición en *"SHeap"*. En caso de que no se encuentrase el bloque, se activaría el recolector de basura por medio del algoritmo *"marcar y barrer"*. El algoritmo *"marcar"* buscaría

los punteros externos asignados en *"SHeap"* y el algoritmo *"barrer"* recorrería toda la memoria dinámica y para cada nodo marcado, lo desmarcaría, y para cada nodo no marcado, lo liberaría.

En conclusión, la línea de investigación que se abre en este caso buscaría definir la mejor estrategia para gestionar la memoria dinámica así como los impactos que esta gestión produciría en los ciclos de SCAN de los proyectos.

6.5.4. Reserva y liberación de memoria dinámica. NEW y DESTROY

Por defecto, todos los objetos que se instancian en un POU son alojados en la memoria estática del PLC. Ésto implica que es el programador el encargado de instanciar los objetos en tiempo de diseño. Si a priori no se conoce el número de objetos que se van a utilizar, es necesario almacenar estos objetos en la memoria dinámica *"SHeap"*. Para poder insertar un objeto en *"SHeap"* se podría optar por dos estrategias:

1. Que el ingeniero programador realice las llamadas de inserción y eliminación de objetos con el gestor de memoria.
2. Que sea el propio sistema el encargado de hacer las inserciones y eliminaciones de objetos.

La primera posibilidad obligaría al programador a hacer las peticiones de inserción de un objeto al gestor de memoria y cerciorarse de que se copia a *"SHeap"* todos sus atributos. Esta solución no sería muy eficiente y sería un foco de errores, ya que el usuario podría intentar utilizar un objeto que no hubiese sido debidamente insertado en *"SHeap"*, o realizar la inserción de forma incorrecta, etc. Y por supuesto, incluso si se hubiese hecho todo correctamente, cualquiera que modificase el programa estaría expuesto a cometer esos mismos errores. Una gran parte de los problemas de la programación de sistemas dinámicos con objetos tiene su origen en la inicialización incorrecta de objetos en el *"Heap"*.

La segunda estrategia dejaría toda la gestión de la inserción y eliminación de los objetos en *"SHeap"* al sistema. Para ello, el ingeniero tendría que indicar al compilador que se querría hacer una inserción o eliminación de un objeto en *"SHeap"*.

Para informar al compilador de la agregación o eliminación de un objeto dinámico se puede optar por las soluciones que desde la informática se ha dado a este problema. En lenguajes tales como Java, o los de .Net, si se quiere hacer una reserva o liberación de memoria dinámica, se hace por medio de las palabras reservadas *"New"* y *"Destroy"*. Un sistema dinámico para el control de procesos no debería de romper con la historia de la programación OO en informática y debería usar unas palabras reservadas parecidas.

El operador *"New"* permitiría al ingeniero programador, mediante una única sentencia, realizar todo el trabajo y cálculos necesarios para la creación de un objeto dinámico dejando todo el trabajo trás el telón al sistema. Ésto se consiguiría introduciendo las llamadas necesarias al gestor de memoria y asignando el puntero devuelto por el gestor a la variable a la que se asigna. Por otro lado, el complemento al operador *"NEW"* es la expresión *"DESTROY"*, que primero llamaría al destructor propio del objeto (para ejecutar cualquier código introducido por el usuario) y después liberaría la memoria ocupada por dicho objeto en *"SHeap"*.

Como conclusión, se hace necesario una investigación que permita adoptar los operadores *"NEW"* y *"DESTROY"* del mundo informático en un entorno industrial basado en IEC 61113 tratando de aprobechar la potencia de MIOOP.

6.5.5. Sistemas de control distribuido

Se observa una cierta tendencia en el mundo de la automatización industrial hacia el uso cada vez mayor de sistemas de control distribuido. En este sentido, la institución IEC está desarrollando dos nuevos estándares aprovechando el eco que ha alcanzado la norma IEC 61131.

El primero de estos estándares, denominado *"Fieldbus"* para uso en sistemas de control industrial ([IEC00c, IEC00d, IEC00f, IEC00g, IEC00e]), normaliza todos los aspectos relacionados con la denición de un bus de campo para comunicaciones industriales (capa física, protocolo de enlace de datos, capa de aplicación, etc.) el cual es imprescindible para implementar un sistema de control distribuido.

El segundo estándar denominado *"Bloques Funcionales para sistemas de control y medida de procesos industriales"* ([IEC00h, IEC01, IEC02]) vendría a normalizar los distintos aspectos necesarios para poder implementar un sistema de control distribuido. En el libro *"Modelado de sistemas de control mediante IEC 61499. Apli-*

cación de bloques funcionales a sistemas distribuidos." de R. W. Lewis ([Lew01]), se describen las características de esta norma de manera amena y de fácil comprensión. En resumen, se puede decir que esta norma define una arquitectura hardware y software distribuida formada por configuraciones, recursos, aplicaciones y programas, y un lenguaje de programación basado en el empleo de una nueva forma de bloques funcionales evolucionados de los bloques funcionales definidos en la norma IEC 61131-3 para cubrir sus carencias.

En conclusión, una posible línea de investigación consistiría en estudiar cómo puede afectar esta norma a la utilización de MIOOP y qué adaptaciones o ampliaciones serían necesarias para su uso.

6.5.6. Objetos distribuidos

En los sistemas Cliente/Servidor, un objeto distribuido es aquel que esta gestionado por un servidor y sus clientes invocan sus métodos utilizando un *"método de invocación remota"*. El cliente invoca el método mediante un mensaje al servidor que gestiona el objeto, se ejecuta el método del objeto en el servidor, y el resultado se devuelve al cliente en otro mensaje.

Dentro de las tecnologías orientadas a objetos distribuidas (OOD), tres de las más importantes y más usadas en el mundo de la informática son:

- RMI. Remote Invocation Method. Fue el primer fremework para crear sistemas distribuidos de Java. El sistema de Invocación Remota de Métodos (RMI) de Java permite a un objeto que se está ejecutando en una Máquina Virtual Java (VM), llamar a métodos de otro objeto que está en otra VM diferente. Esta tecnología está asociada al lenguaje de programación Java, es decir, que permite la comunicación entre objetos creados en este lenguaje.

- DCOM. Distributed Component Object Model. El Modelo de Objeto Componente Distribuido esta incluido en los sistemas operativos de Microsoft. Es un juego de conceptos e interfaces de programa, en el cual los objetos de programa del cliente pueden solicitar servicios de objetos de programa servidores en otros ordenadores dentro de una red. Esta tecnología esta asociada a la plataforma de productos Microsoft.

- CORBA. Common Object Request Broker Architecture. Tecnología introducida por el Grupo de Administración de Objetos OMG, creada para establecer una plataforma para la gestión de objetos remotos independiente del lenguaje de programación.

Dado que MIOOP permite la programación OO de sistemas de control basados en el estandar IEC 61131 y las tecnologías RMI, DCOM y CORBA son de uso común en el mundo de la informática, la idea pasaría por encontrar una forma de mezclar dichas tecnologías. A este nuevo sistema se le podría denominar *"MIOOP-D"* (MIOOP distribuido) y debería de resolver los siguientes problemas de una forma totalmente transparente al usuario:

- Ejecución de servicios. Permitir que dos computadores ejecuten un servicio en otra máquina diferente y esta devuelva el resultado de la operación.
- Inestabilidad de la red. Cada computador debe gestionar un protocolo de comunicación que evite latencias de tiempo en la transferencia de mensajes a través del medio o caídas repentinas de una máquina.
- Seguridad. Deben existir diversos criterios de seguridad para permitir la ejecución de estos servicios remotos ya que pueden estar sujetos a un ambiente hostil, por el hecho de encontrarse en red.
- *"Marshalling"* y *"Unmarshalling*". Si se ejecuta un programa en una sola computadora se tiene la seguridad que la representación de datos esta conforme a esa plataforma independientemente del sistema operativo que posea. Sin embargo, el paso de parámetros complejos entre computadoras de fabricantes distintos presenta un gran problema de compatibilidad. El *"Marshalling"* y *"Unmarshalling"* es el proceso por el que debe pasar toda información para que ésta sea utilizable en ambientes heterogéneos.

Desde un punto de vista arquitectónico, CORBA (ver figura 6.17) y RMI (ver figura 6.16) presentan gran similitud. En ambos tecnologías, uno de los primeros detalles para aislar al programador de los cuatro puntos anteriormente mencionados, involucra el concepto utilizado en los lenguajes OO: la separación entre interface e implementación. El interface es únicamente una declaración del método(s) definido en la implementación donde se encuentra definida la lógica de

negocio, es decir, el interface funciona como una declaración. En RMI estos interfaces se declaran por medio de una biblioteca de clases propia mientras que en el caso de CORBA, al ser multi plataforma, requiere la definición de los servicios través de un lenguaje llamado IDL (Interface Definition Language).

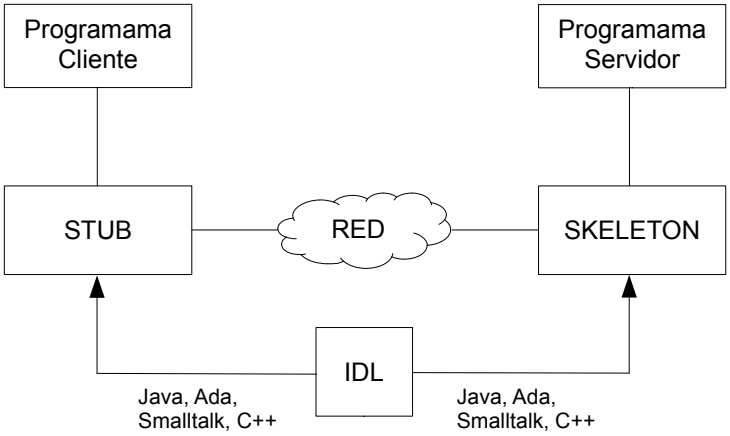

Figura 6.16: Arquitectura CORBA

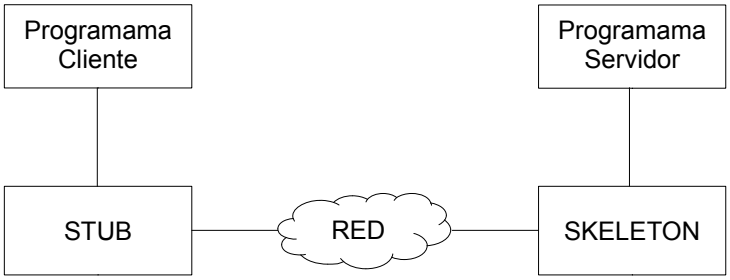

Figura 6.17: Arquitectura RMI

Una vez definidos los interfaces e implementaciones de todos los servicios, es necesario definir otro elemento para ejecutar | invocar servicios remotos: Los llamados *"Stubs"* y *"Skeletons"*. Estos *"Stubs"* y *"Skeletons"* permiten que en el momento de ser invocado un servicio remoto este pueda ser simulado localmente. El *"Stub"* funciona como un simulador para todos los servicios que están siendo invocados y pertenecen a la implementación del servidor, mientras el Skeleton funciona como simulador para recibir parámetros de la implementación del cliente. Otra labor importante que cumplen los *"Stubs"* y *"Skeletons"* es el *"Marshalling"* y *"Unmars-*

halling" de información.

La definición de un lenguaje intermedio tipo IDL no sería necesario en *"MIOOP-D"* ya que al traducir a código estructurado compatible con la norma IEC 61131 los programas que se compilan, se resolvería el problema de heteregeonidad entre computadoras de distintos fabricantes. Bastaría con definir una clase base que al ser heredada, indicase al compilador que el tipo de dato *"INTERFAZ"* (ver apartado 173) soportado por MIOOP debe ser considerado como un OD (objeto distribuido). El nombre de dicha clase base podría ser el acrónimo de Interoperable Network Distributed Object (INDO). Dicha palabra clave estaría reservada y podría declararse como se puede observar en el algorítmo 385. En tiempo de compilación, el compilador traduciría dichas clases y generaría de forma automática los *"Stubs"* y *"Skeletons"*.

Algoritmo 6.3 Ejemplo de definición de interfaz INDO

```
INTERFAZ ClaseDistribuida (INDO)
  METHOD servicioUno () : VOID;
  METHOD ServicioDos () : VOID;
  METHOD servicioTres () : VOID;
END_INTERFAZ
```

Por otro lado, el compilador debe proveer al ingeniero de un apartado donde poder asociar los objetos distribuidos con una dirección IP, para que cuando un computador reclame un servicio remoto, el sistema conozca la ubicación donde éste se debe ejecutar. Dicha información se tendría que asociar a cada *"Stub"* y *"Skeleton"* de forma totalmente transparente y en tiempo de compilación.

"MIOOP-D" debería poder gestionar en tiempo de ejecución los *"Stubs"* y *"Skeletons"* de forma dinámica ya que no tendría sentido tener en la memoria del PLC todos los objetos remotos, porque podría transcurrir mucho tiempo hasta que otra máquina reclamase un servicio de ellos. Para ello, *"MIOOP-D"* debería proporcionar una segunda capa de abstracción, un middleware que se ejecutase por debajo del programa principal y que se encarguase de manejar la creación, ejecución y destrucción de los *"Stubs"* y *"Skeletons"* de forma dinámica. Dicha capa de abstracción pasaría por crear un gestor de peticiones que podría recibir el nombre de Remote Connections Manager (RCM). La labor de este gestor sería:

- Envio y recepción de los mensajes.

- Gestión de errores, fallos y seguridad de los tiempos de envio de mensajes.
- Creación, ejecución y destrucción de los *"Stubs"* y *"Skeletons"*.

En conclusión, esta nueva línea de investigación requeriría que se desarrollase e investiguase un sistema dinámico estable con MIOOP o alguna alternativa OO, para poseteriormente definir los protocolos de comunicación entre máquinas a través de la red, establecer la arquitectura de *"MIOOP-D"* así como del *"RCM"*, definir las traducciones necesarias para hacer compatible el lenguaje desarrollado a alto nivel con la norma IEC 61131 y obtener las implicaciones que tendría esta tecnología en tiempos de ciclo de SCAN y facilidad de uso si se compara con las actuales tendencias, que desde el mundo académico van dirigidas a sistemas distribuidos estructurados bajo el estándar IEC 61499.

6.5.7. Agentes fuzzy

Un agente inteligente, es una entidad capaz de percibir su entorno, procesar tales percepciones y responder o actuar en su entorno de manera racional, es decir, de manera correcta y tendiendo a maximizar un resultado esperado. Según Nikola Kasabov [kG97] los agentes deben exhibir las siguientes características:

- Aprender nuevos problemas e implementar posibles soluciones.
- Capacidad de adaptación en línea y en tiempo real.
- Ser capaz de analizar condiciones en términos de comportamiento, el error y el éxito.
- Aprender y mejorar a través de la interacción con el medio ambiente (realización).
- Aprender rápidamente de grandes cantidades de datos.
- Deben estas basados en memoria de almacenamiento masivo y ser capaces de recuperar la información.

Desde el punto de vista de la automatización de procesos, podría ser interesante por ejemplo, disponer de máquinas controladas por PLCs con comportamiento variable autoregulado dependiendo de la situación del lazo de producción, o un

sistema capaz de calcular la carga de trabajo de los distintos computadores y balancear la ejecución de una serie servicios de una máquina a otra, o que ante un fallo de un computador, los propios agentes reconfiguren el sistema sin que éste se tenga que parar.

Como conclusión, esta línea de investigación final usaría la tecnología aportada por MIOOP, junto con la los sistemas dinámicos y los sistemas distribuidos orientados a objetos para mezclarlos con los sistemas de lógica difusa recogidos en el apartado 7 de la norma IEC 61131 [IEC00a], y de esta manera obtener un sistema de agentes expertos distribuidos orientados a objetos.

6.6. Conclusiones

Por un lado, MIOOP se presenta como un instrumento eficaz para el desarrollo de programas de control de procesos siguiendo un paradigma OO, describiendo los mecanismos necesarios para traducir dichos programas a sus versiones estructuradas compatibles con la norma IEC 61131-3.

Por otro lado, MIOOP aporta la herramienta software SIMPLC++ para el desarrollo de programas de control OO. Dicha herramienta permite traducir programas de control basados en MIOOP a programas que se ajustan a la norma IEC 61131-3, traspasar dicho código a PLCs de la marca Scheneider y medir de forma empírica el impacto que dicha traducción tiene en el código final que se produce tras la compilación, tanto en número de líneas de código como en tiempos de ciclo de SCAN.

Apéndice A

Norma IEC 61131

El verdadero guerrero siempre cuenta con tres armas: la radiante espada de la pacificación; el espejo de la valentía, la amistad y la sabiduría; y la piedra preciosa de la iluminación.
Morihei Ueshiba

Hace tiempo que las empresas de ingeniería han abandonado el uso de la lógica cableada en pos de la lógica programada como medio de llevar a cabo el control de procesos. La lógica programada recibe este nombre porque con ella la lógica de control se presenta por medio de instrucciones que pueden ser organizadas para ser ejecutadas por un *"equipo de control programable"*. Este tipo de equipos de control se basan en el empleo de microprocesadores para la ejecución de las instrucciones que forman la lógica de control, a las cuales se las llama comúnmente *"programa de control"* o simplemente *"programa"*.

A.1. Sistemas de control distribuido (DCS)

En este tipo de sistemas de control no existe un único sistema central encargado de ejecutar un programa de control al cual llegan todas las señales de los sensores

del proceso y del cual parten todas las señales que van hasta los preaccionadores y accionadores. La *"intelegiencia"*, entendida como la capacidad de un dispositivo de ejecutar un programa de control, o lo que es lo mismo, de tomar una decisión en base a las señales de los sensores del proceso en un instante determinado, se haya distribuida entre una serie de nodos interconectados por una red de comunicaciones que les permite el intercambio de información.

A.2. Ordenadores personales industriales

Consisten en emplear ordenadores personales con una estructura constructiva capaz de resistir las condiciones hostiles del entorno industrial en que van a ser empleados. Estos equipos por lo general ejecutan un sistema operativo en tiempo real sobre el cual se ejecuta el programa de control necesario para controlar el proceso de la manera deseada.

Su principal ventaja con respecto al resto reside en sus mayores capacidades de procesamiento que permiten el empleo del mismo equipo para el control del proceso y para la ejecución de aplicaciones auxiliares, como por ejemplo sistemas SCADA, gestores de bases de datos, aplicaciones de tratamiento estadístico, etc.

Además, presentan una total independencia del hardware con respecto al software, tanto del sistema operativo como de los lenguajes de programación empleados para programar la lógica de control, lo que garantiza que las empresas podrán emplear para un proceso dado la solución deseada con independencia del resto de soluciones utiizadas en el resto de la planta.

A.3. Microcontroladores

El empleo de este tipo de soluciones se está extendiendo en el ámbito de los sistemas empotrados. La razón fundamental es su bajo coste y la necesidad de que el sistema controlado y el sistema controlador formen un única unidad física. Ejemplos de este tipo de sistemas son los empleados en los electrodomésticos (microondas, lavadora, vídeo, ...), en los vehículos (ABS, SRS, ...), etc.

A.4. Autómatas programables

También se les conoce con el nombre de PLCs. Surgen en los anales de la década de 1960 en EE.UU. en el sector de la automoción como respuesta a la necesidad de simplificar, agilizar, abaratar y facilitar el desarrollo y mantenimiento de los sistemas de control hasta aquel entonces resueltos mediante lógica cableada.

Hoy en día, constituyen la solución más empleada en la automatización de procesos, debido fundamentalmente a su adaptación al medio industrial, su robustez y la facilidad de interface con el proceso.

Su principal desventaja la constituye la incompatibilidad existente entre los distintos PLCs de los distintos fabricantes tanto a nivel de hardware como de software de programación. Esta incompatibilidad nace de la decisión de estrategia comercial de los fabricantes de este tipo de equipos para fidelizar a sus clientes a base de hacer sus sistemas incompatibles con los de la competencia. De esta forma, cuando una empresa de producción industrial decide automatizar sus procesos mediante el empleo de autómatas programables de una determinada marca, deberá continuar consumiendo los productos de esa marca para solucionar sus futuros problemas de automatización, aún cuando le fuese más beneficioso o rentable emplear los productos de otras marcas.

Para tratar de paliar esta situación se han llevado a cabo muchos esfuerzos encaminados casi todos a la definición de estándares de comunicación de señales (Profibus, AS-i, Ethernet Industrial, CAN...) para tratar de asegurar la compatibilidad de los sistemas automatizados a nivel del intercambio de las señales de control que manejan. Ésto debería permitir configurar redes de comunicación de sistemas automatizados con PLCs de distintos fabricantes, pero en la práctica sucede que dos implementaciones del mismo estándar de comunicación llevadas a cabo por dos fabricantes distintos no son compatibles al 100 %.

También se han hecho algunos esfuerzos para tratar de unificar o estandarizar la arquitectura hardware y los lenguajes de programación de los PLCs con mayor o menor éxito de aceptación, entre los que cabe citar: Osaca, MatPLc, IEC 61131, etc. cuyas características más importantes se detallan a continuación.

A.4.1. OSACA

OSACA (Open System Architecture for Controls within Automation Systems [OSA96] es el nombre de una asociación paneuropea sin ánimo de lucro que nace en 1992 con el objetivo de tratar de denir una arquitectura de sistema abierto para aplicarla en la implementación de sistemas de automatización, y tratar de combatir de esta forma la incompatibilidad existente entre los distintos tipos de sistemas de automatización de los diferentes fabricantes. En esta asociación participan tanto instituciones de investigación, como empresas de desarrollo, empresas consumidoras de sistemas automatizados y empresas fabricantes de equipos de automatización de varios países europeos.

La definición de sistema abierto para automatización propuesta por OSACA se basa en la definición de este tipo de sistemas recogida en el estándar IEEE 1003.0 de 1.990 [IEE90], según la cual :

> Un sistema abierto provee de las capacidades necesarias para garantizar que cualquier aplicación correctamente diseñada pueda ser ejecutada en diversas plataformas proporcionadas por distintos fabricantes, que pueda interaccionar con otras aplicaciones y que presente un estilo consistente de interacción con el usuario.

El objetivo final perseguido por OSACA es definir una arquitectura abierta para aplicar al diseño de sistemas de automatización que describa una topología de control dependiente solamente del proceso, intercambiable, interoperable, escalable y portable.

Según OSACA, esta arquitectura proporcionaría una serie de beneficios tanto a los usuarios finales (reducción de costes, interfaces estandarizados, integrabilidad,...), como a los fabricantes de maquinaria (desarrollo independiente del proveedor, reducción de costes, más facilidad en la implementación de las peculiaridad de control de cada cliente,...) e incluso a los fabricantes de equipos de control (reusabilidad de software, posibilidad de emplear hardware de otros fabricantes,...).

La arquitectura Osaca sigue una topología cliente-servidor organizada en varios niveles como se puede apreciar en la figura A.1. Según esta arquitectura, en el nivel más bajo están situados una serie de componentes electrónicos que son los que implementan las funciones básicas del sistema coordinados por un sistema

operativo que proporciona servicios a un módulo de comunicaciones al interface de programación de aplicaciones (API). Los servicios proporcionados por este API y por el módulo de comunicaciones serán empleados por el programador de objetos de arquitectura (AOs) para llevar a cabo el control del proceso. Cada uno de estos AO actuará como un servidor para el resto de objetos AO del sistema de control los cuales podrán invocar a los servicios que este proporciona cuando lo estimen oportuno. Desde este punto de vista se dice que actúan como clientes.

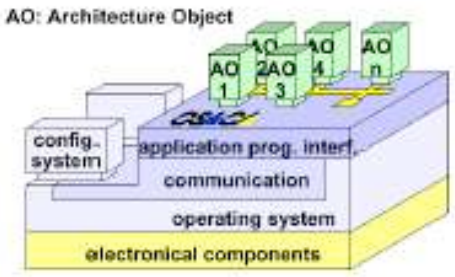

Figura A.1: Arquitectura OSACA

Los resultados de esta iniciativa no han tenido mucha repercusión a nivel internacional prueba de lo cual es el hecho de que los trabajos de la asociación han sido congelados.

A.4.2. MATPLC

Originariamente este proyecto recibía el nombre de PINPLC pasando posteriormente a adoptar la nueva denominación de MATPLC.

El proyecto MATPLC [Sou01] surge como una iniciativa popular alentada por el éxito del desarrollo de Linux fruto del esfuerzo desinteresado de miles de colaboradores anónimos. Al amparo del foro de intercambio de información sobre temas de control alojado en la página web www.control.com nace la idea de llevar a cabo un nuevo proyecto bajo licencia GNU (General Public License), para desarrollar un sistema de control industrial basado en PC. Este tipo de sistemas es lo que se conoce comúnmente como SoftPLC, es decir, se trata de aplicaciones software ejecutadas sobre un ordenador personal que ejecutan un programa de control imitando la forma en que lo hace un PLC real.

Este tipo de solución está a medio camino entre los PLCs físicos y los PCs industriales, y se está haciendo muy popular en los últimos años al tomar lo mejor de las dos soluciones: por un lado la forma de trabajo de los PLCs que está comprobado el éxito de su aplicación y por otro lado, la capacidad de procesamiento y la flexibilidad de utilizar un sistema basado en un ordenador personal. Todos los grandes fabricantes de equipos de automatización ofrecen hoy en día soluciones SoftPLC compatibles con sus líneas de productos.

MATPLC nace con un doble objetivo: por un lado tratar de romper con el carácter cerrado y con la incompatibilidad introducida por los fabricantes de equipos de automatización y por otro lado, tratar de buscar una alternativa al uso de Windows como sistema operativo sobre el que se ejecutan los SoftPLCs ofrecidos por estos fabricantes. El grupo de personas que preconizan el desarrollo de MATPLC considera que Windows no es un sistema operativo suficientemente estable para llevar a cabo el control de un proceso de manera ininterrumpida durante largos periodos de tiempo, además de consumir gran cantidad de recursos en tareas innecesarias a la hora de emplear el PC para llevar a cabo el control de un proceso. Por este motivo, MATPLC fue desarrollado para ser ejecutado sobre Linux que es un sistema operativo gratuito, estable y que puede ser adaptado a las necesidades del usuario, ya que los módulos que lo forman pueden ser desinstalados si el usuario no los necesita. Por ejemplo, si para llevar a cabo el control de un proceso no se necesita ejecutar ningún vídeo, en Linux es posible desinstalar el módulo gestor de vídeos para evitar que consuma recursos, mientras que en Windows no es posible.

Si fuese necesario MATPLC se podría ejecutar sobre LinuxRT, un sistema operativo con la misma arquitectura que Linux pero que funciona en tiempo real por lo que es muy adecuado para la ejecución de aplicación de control.

Aunque MATPLC emula el funcionamiento de un PLC real, su arquitectura es completamente diferente hasta el punto de que no es necesario seguir una estructura cíclica de ejecución como ocurre en los PLCs (lectura de entradas, ejecución de programa y actualización de salidas) aunque si se desea es posible configurar los programas de control en MATPLC para que funcionen de esta forma (ver figura A.2).

MATPLC se basa en una arquitectura modular, para permitir la ampliación del mismo sin necesidad de tener que modificar las características básicas del kernel. Cada uno de estos módulos autónomos se ejecutan por defecto en procesos dife-

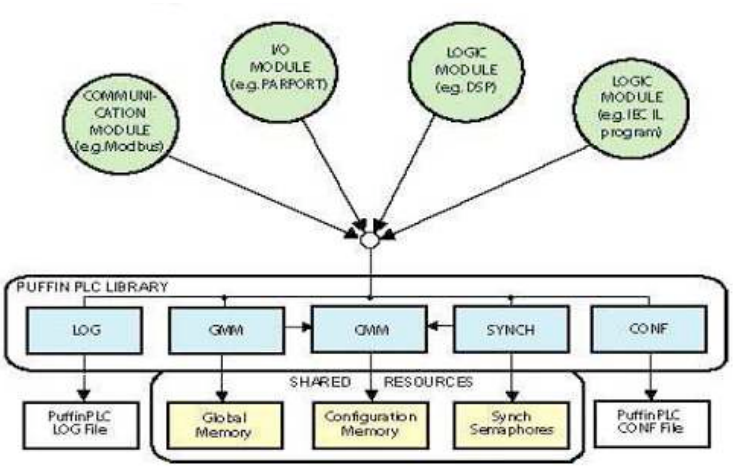

Figura A.2: Arquitectura MATPLC

rentes y tienen la libertad de decidir si desean ejecutarse imitando el ciclo de scan de los PLCs o no.

Se puede disponer por ejemplo de módulos para la lectura y escritura de entradas y salidas, módulos para implementar un lazo de regulación PID, módulos para llevar a cabo un filtro sobre una señal, etc. Estos módulos se comunican por medio del uso de los servicios proporcionados por una librería que forma el kernel de MATPLC.

La librería de MATPLC está formada por cinco componentes encargados de proveer a los distintos módulos de ciertos servicios fundamentales, a saber:

- Gestor de la configuración de memoria, encargado de gestionar la memoria compartida en la que se almacena la información de configuración.
- Gestor de la memoria global, el cual se encarga de administrar el uso de la memoria empleada para almacenar el valor de las variables empleadas en el programa de control.
- Librería de sincronización. Implementa los servicios necesarios para sincronizar el acceso a la memoria global.
- Librería de configuración. Sus servicios son empleados para chequear las opciones de configuración.

- Módulo de registro de actividades. Brinda servicios básicos para permitir a los módulos registrar en archivos sus actividades. Es usado fundamentalmente para llevar a cabo depuraciones sobre el programa de control.

MATPLC presenta una versión finalizada y estable desde Mayo de 2006, pero el hecho de estar centrado en el sistema operativo Linux le proporciona una resonancia limitada todavía en el mundo de la automatización.

Sin embargo, algunas de las ideas que inspiran su arquitectura son muy interesantes y aportan una nueva forma de entender el control de procesos mediante equipos programables.

A.4.3. IEC 61131[1]

La norma internacional IEC 61131 constituye el primer intento serio de estandarizar los distintos aspectos constitutivos de los autómatas programables. Surge al auspicio de la Comisión Electrotécnica Internacional la cual a comienzos de los años 1990 creó un grupo de estudio denominado *"WG7: Programable Control Systems"* perteneciente al subcomité *"SC65B: Devices"* el cual a su vez estaba encuadrado en el comité técnico *"TC65: Industrial Process Measurement and Control"*. Dada la envergadura del trabajo que se iba a abordar se decidió dividir el grupo *"WG7"* en varios grupos de trabajo cada uno de los cuales se encargaría de estudiar, definir y publicar un documento en el que se recogieran las guías a seguir en un determinado aspecto constitutivo de los PLCs. Estos grupos son los siguientes:

- Parte 1: Información general. Presenta la definición de los conceptos básicos y de la terminología a emplear a lo largo del estándar [IEC2a]. Su año de publicación fue 1992.
- Parte 2. Requisitos del equipamiento y pruebas. Estandariza los conceptos relativos a la electrónica y la parte de construcción mecánica además de establecer las baterías de pruebas que el equipo debe pasar [IEC2b]. Su año de publicación fue 1992.

[1] Esta sección no pretende ser un recorrido exhaustivo por las características de la norma IEC 61131 ni mucho menos. Por ello se recomienda al lector la consulta a los documentos de la propia norma y de los libros que la explican en detalle (citados más adelante). El objetivo de esta sección es simplemente dar unas nociones básicas acerca de los principios en que se basa la misma y más concretamentesu en su parte 3 donde se describen los lenguajes de programación soportados por la norma.

- Parte 3. Lenguajes de programación. Define y estandariza los lenguajes de programación de PLCs, la estructura que los programas de control deben seguir y la forma en que deben ser ejecutados [IEC93]. Su año de publicación fue 1993.
- Parte 4. Guías de usuario. Recoge una serie de guías o consejos a la hora de seleccionar, instalar y mantener un PLC. Su año de publicación fue 1995.
- Parte 5. Comunicaciones. Estandariza la forma en que se deben producir la comunicación entre varios PLCs mediante el empleo del protocolo MAP [IEC00g]. Su año de publicación fue 2000.
- Parte 6. Comunicaciones vía Fieldbus. Estandariza la forma en que se deben producir la comunicación entre varios PLCs mediante el protocolo Fieldbus. (Publicación pendiente de la definición total del estándar Fieldbus).
- Parte 7. Programación de control Fuzzy. Define una librería de bloques funcionales que permite la implementación de programas de control de PLC basados en lógica difusa [IEC00a, MR06]. Su año de publicación fue 2000.
- Parte 8. Guías para la implementación de lenguajes de programación para PLCs. Define las normas que se deberían seguir a la hora de crear nuevos lenguajes de programación para PLCs. Su año de publicación 2000.

De todas estas partes, la que más impacto y aceptación ha tenido no solo entre los usuarios sino incluso entre los fabricantes de PLCs, es la parte 3 referente a los lenguajes de programación. La aparición de la fundación internacional *"PLCopen"* (www.plcopen.org) cuyo principal cometido es fomentar el uso del estándar en todos sus aspectos y concretamente en el apartado 3, y el hecho de que representantes de las más grandes compañías fabricantes de PLCs del mundo (Siemens, Allen-Bradley, Telemechanique, ...) formen parte de su patronato directivo, está produciendo que cada día más y más compañías estén optando por la adopción de este estándar como guía de desarrollo de sus nuevos productos tanto a nivel hardware como software.

Es la parte 3 la que en mayor medida afecta a este trabajo por lo que sólo se van a presentar a continuación las características fundamentales de esta parte (a partir de este momento cuando se diga *"la norma"* se estará haciendo referencia a la parte 3 solamente). Si el lector lo desea, puede ampliar lo aquí expuesto

en los documentos que exponen las distintas partes del estándar. En lo que a la parte 3 se refiere, se sugiere al lector la lectura del libro *"Programming Industrial Control Systems usingIEC 1131-3"* de R. W. Lewis [Lew95] en el cual se explica de forma amena y entendible el contenido de esta parte. La lectura de artículos como [JA98, Nes99] por citar sólo dos, pueden servir para comprender de manera sucinta el alcance de esta parte.

En la práctica, la norma se divide en dos partes como se muestra en la figura A.3: elementos comunes y lenguajes de programación.

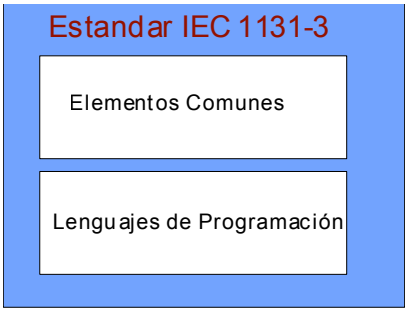

Figura A.3: Partes de la norma IEC 61131-3

Elementos comunes

Es el conjunto de definiciones de elementos de programación empleados para crear un programa de control independientemente del lenguaje de programación utilizado. Entre estos elementos cabe destacar:

1. Conjunto de caracteres. Define el conjunto de caracteres válidos a emplear para representar el resto de elementos de la norma.

2. Identificadores. Especifica la forma válida de de construir los identificadores de variables, constantes y en definitiva, de cualquier elemento que pueda ser utilizado en un programa de control.

3. Palabra clave. Recoge la lista de palabras o identificadores que tienen un significado específico dentro de los distintos lenguajes de programación y que por tanto no pueden ser utilizados por el usuario como identificadores de variables.

4. Tipos de datos. Los tipos de datos definen la naturaleza de la información que las instrucciones de cada lenguaje de programación puede manejar. Hacen referencia a su tamaño y a su significado. Los tipos de datos básicos son: booleanos, bytes, números enteros, números reales, palabras, horas del día, fechas y cadenas (strings). Sobre estos tipos de datos básicos, el usuario puede definir sus propios tipos de datos (tipos de finidos y estructuras) y elementos dimensionales (arrays).

5. Variables. Las variables permiten identificar aquellos datos cuyo contenido puede cambiar, por ejemplo, los datos asociados a entradas, salidas o memoria del autómata programable. Una variable se puede declarar como uno de los tipos de datos elementales definidos en la norma o como uno de los tipos de datos derivados. De este modo se crea un alto nivel de independencia con el hardware, favoreciendo la reusabilidad del software. El alcance de las variables es normalmente limitado a la unidad de organización del programa en la cual son declaradas.Ésto significa que sus nombres pueden ser reutilizados en otras partes sin ningún conflicto, eliminando así otra fuente de errores, que son las variables temporales. Si la variable requiere un alcance global, debe ser explícitamente declarado con la directiva *"VAR_ GLOBAL"*.

6. Modelo de organización de programas de control. La norma define un modelo de organización del programa de control que se desea ejecutar en un PLC. Para ello, define 3 tipos de estructuras capaces de albergar instrucciones ejecutables escritas en alguno de los lenguajes de programación de la norma, cada uno de los cuales recibe el nombre genérico de POU (Unidad de Organización de Programa). Éstos son:

   a) Funciones. El término de función en la norma tiene el mismo significado que el del término matemático función, ésto es, una instrucción que puede ser invocada tantas veces como se desee para resolver una operación (generalmente matemática). La norma define una serie de funciones estándar (sqrt, abs, cos, sen, add,...) que pueden ser ampliadas por funciones creadas por el usuario que una vez definidas pueden ser usadas repetidamente.

   b) Bloques Funcionales (FBs). Un bloque funcional según lo define la norma puede ser considerado como una caja negra que encapsula una funcionalidad del programa de control. A diferencia de las funciones, los

FBs no son deterministas, es decir, que para el mismo conjunto de valores de entrada no se asegura que el FB produzca la misma salida. Ésto es debido a la capacidad que tienen los FBs de almacenar su estado interno de manera persistente aún después de terminar su ejecución. Se pueden crear tantas copias de un FB como se desee. A cada copia, la norma le asigna el nombre de *"instancia"*. Cada instancia mantendrá la información correspondiente al estado en que se encuentra la misma. La norma define por defecto un conjunto de FBs básicos como por ejemplo contadores, temporizadores, biestables, etc. A este conjunto básico, el usuario puede añadirle los FBs que desee programándolos en cualquiera de los lenguajes del estándar.

c) Programas. Los programas constituyen los POUs de más alto nivel. Conceptualmente son muy parecidos a los FBs ya que igual que estos, proveen de un mecanismo para la reutilización del software y pueden contener datos, instrucciones, entradas y salidas. Su mayor diferencia radica en que no se puede invocar un programa desde otro.

Una vez que el programa de control está organizado en POUs, es el momento de especificar de qué forma se desea que se ejecuten las mismas para lograr controlar el proceso como se desee. Para ello, la norma define el modelo de ejecución.

7. Modelo de ejecución. Este modelo se organiza en torno a tres conceptos como se muestra en la figura A.4:

   a) Configuración. Al más alto nivel, el elemento software requerido para solucionar un problema de control particular puede ser formulado como una configuración. Una configuración es específica para un tipo de sistema de control, incluyendo las características del hardware: procesadores, direccionamiento de la memoria para los canales de entrada/salida y otras capacidades del sistema.

   b) Recurso. Dentro de una configuración, se pueden definir uno o más recursos. Se puede entender el recurso como un procesador capaz de ejecutar programas IEC.

   c) Tarea. Asociado a cada recurso pueden estar definidas una o más tareas. Las tareas controlan la ejecución de un conjunto de programas y/o

bloques funcionales. Es decir, la sola definición de un programa o un FB no implica su ejecución, sino que una instancia del mismo debe ser asociado a una tarea para garantizar que este será ejecutado en algún momento. Cada tarea puede ser configurada para ser ejecutada periódicamente o por una señal de disparo especificada.
Por lo general, un PLC convencional está formado por una única configuración que contiene un único recurso (una CPU) la cual ejecuta una única tarea cíclica. Puede también tener definidas tareas asociadas a la presencia de determinados eventos que reciben comúnmente el nombre de interrupciones.

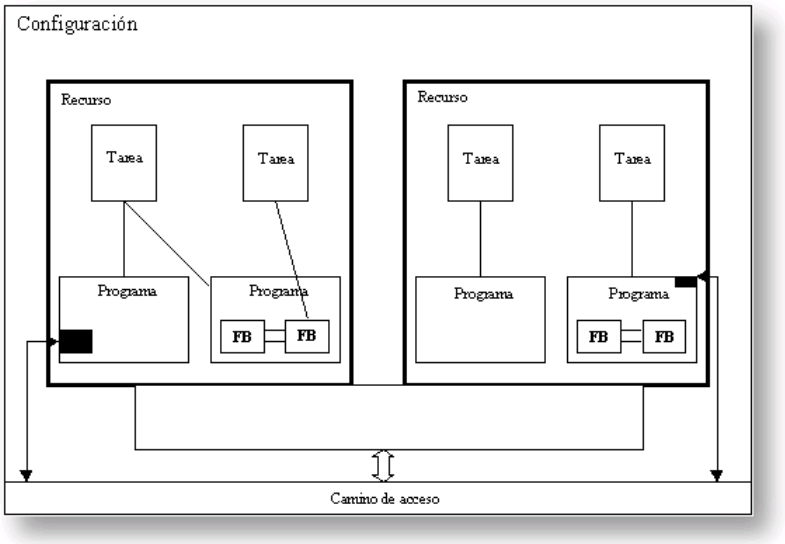

Figura A.4: Modelo de organización de la norma IEC 61131-3

8. Lenguajes de programación. Recoge la definición léxica, sintáctica y semántica del conjunto de lenguajes de programación disponibles en la norma. La norma no permite la introducción de particularidades distintivas, lo que daría lugar a la aparición de dialectos. Evidentemente, cualquier compañía es libre para crear su propio dialecto pero en ese caso no recibiría la certificación de compatibilidad con la norma por parte de PLCopen.
   Los lenguajes que define la norma pueden ser divididos en dos categorías como se muestra en la figura A.5:

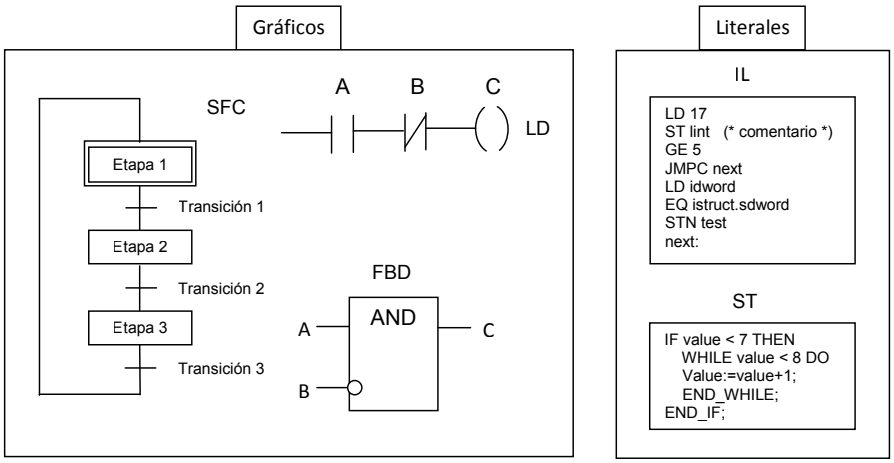

Figura A.5: Lenguajes de programación de la norma IEC 61131-3

a) Gráficos:

- Diagrama de contactos (LD). Tiene sus orígenes en los Estados Unidos. Está basado en la representación gráfica de la lógica de reles (ver figura A.6).

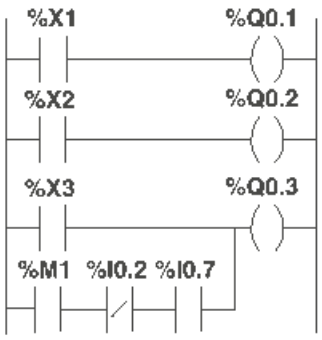

Figura A.6: Diagrama de contactos

- Diagrama de bloques funcionales (FBD). Las funciones y FBs se inspiran en el concepto de circuito integrado para ser cableados

entre sí de forma análoga al esquema de un circuito. Es ampliamente utilizado en Europa y su uso es muy común en aplicaciones que implican un flujo de información o datos entre componentes de control (ver figura A.7).

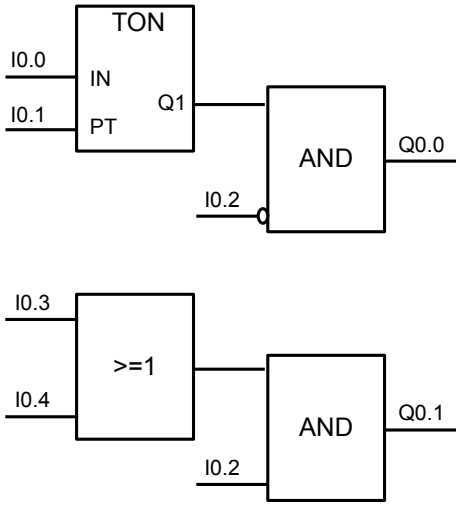

Figura A.7: Diagrama de bloques funcionales

- Diagrama funcional secuencial (SFC). Está diseñado para cumplir con un doble objetivo. Por un lado, proveer de un mecanismo para representar una lógica de control coherente con el resultado de un análisis estructurado y descendente de un problema. Por otro lado, servir como lenguaje de programación para ser empleado en la implementación de cualquier módulo secuencial.
  SFC deriva del lenguaje de modelado GRAFCET [IEC88] al cual se le han añadido los elementos necesarios para convertirlo en un lenguaje de programación, fundamentalmente reglas para expresar la correcta ejecución de POUs en un autómata programable y la introducción del concepto de *"bloque de acción"*. Estos bloques de acción son conceptualmente equivalentes a las acciones de GRAFCET y al igual que éstas, son asociadas a las etapas del SFC. Es sobre estos bloques de acción sobre los que se especifican las reglas de ejecución. Tanto los bloques de acción como las transiciones

pueden ser programados en cualquiera de los lenguajes de programación definidos en la norma, incluido SFC (ver figura A.8).

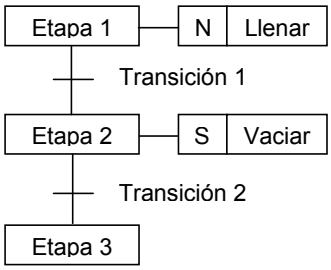

Figura A.8: Diagrama funcional secuencial

b) Literales:

1) Lista de instrucciones (IL). Su origen está en Alemania y es un lenguaje ensamblador para PLCs basado en un acumulador (ver figura).

```
LD    %10.0
AND   %10.0
AND   %M3
AND   %10.5
AND   %10.4
S     %M0
LD    %M2
AND   %10.5
OR    %10.2
R     %M0
LD    %M0
ST    %Q0.0
```

Figura A.9: Lista de instrucciones

2) Texto estructurado (ST). Es un lenguaje de alto nivel inspirado en Ada, Pascal y C. Puede ser utilizado para codificar expresiones complejas e instrucciones anidadas con gran facilidad. Este lenguaje dispone de estructuras para bucles (REPEAT-UNTIL, WHILE-DO,...), ejecución condicional (IF-THEN-ELSE, CASE), funciones (SQRT, SIN, ...), etc.

Cualquier programa de control se podría implementar con mayor o menor esfuerzo empleando cualquiera de los cinco lenguajes. La elección del lenguaje de programación a emplear dependerá de:

- Los conocimientos del programador.
- El tipo de problema a tratar (secuencial o no).
- El nivel de descripción del proceso.
- La estructura del sistema de control.
- La coordinación con otras personas o departamentos.

Los cinco lenguajes están interrelacionados y permiten su empleo para resolver conjuntamente un problema común según la experiencia del usuario.

Apéndice B

LAV

La ilusión me hace un experto,
la realidad me hace un principiante.
Proverbio samurai

SimPLC++ se engloba dentro de un proyecto más ambicioso de ISA, mediante el cual se pretende llegar a disponer de las herramientas necesarias para dotar adecuadamente y a bajo coste un laboratorio de automatización virtual (LAV) [Gon02], orientado a la programación y supervisión de automatismos en base a controladores lógicos programables.

Un laboratorio de automatización virtual consistiría en todos aquellos componentes hardware y software necesarios para el aprendizaje de las técnicas propias de la Ingeniería de Automatización, tales como programación, supervisión de procesos, etc. Son tres las partes fundamentales de un LAV: proceso a controlar; controlador programable encargado del control del proceso, y sistema para manejo y supervisión del proceso controlado. Este esquema en la actualidad está formado por:

- Proceso: En algunas ocasiones se podrá disponer del propio proceso real que se desea controlar. Ésto es poco frecuente y muy caro. En otras ocasiones

se podrá disponer de una maqueta que representa el proceso. Ésto es más frecuente, aunque se sigue presentando el problema del coste y la falta de flexibilidad. Una tercera opción es la construcción de paneles de entrada/salida. Esta última opción es la más frecuente, presenta un menor coste y una gran flexibilidad, aunque es una forma poco intuitiva de representación de los procesos.

- Equipo de Control: Autómatas programables de distintos fabricantes. Aunque presentan características hardware similares, el software de programación y depuración es muy distinto. Son relativamente caros.
- Supervisión del Proceso: Para la supervisión del proceso se suelen utilizar aplicaciones software SCADA. Son muy abundantes pero por lo general son muy caras.

La visión del LAV por el grupo GENIA (Entornos Integrados de Automatización) perteneciente al área ISA de la Universidad de Oviedo, implementa los anteriores componentes del siguiente modo (ver figura B.1):

- Proceso: Aplicación software para la simulación del proceso. Nunca será tan intuitivo como el proceso real, pero sí está al mismo nivel que las maquetas y muy por debajo de sus costes, siendo además mucho más flexibles desde el punto de vista de la variedad de procesos que se pueden representar.
- Equipo de Control: Simulación de un autómata programable basado en el estándar IEC 61131-3 de programación, ampliamente aceptado por los principales fabricantes de autómatas. Esta aplicación será mucho más barata que los autómatas reales lo que permitirá una mejor dotación de los laboratorios de prácticas. En ningún momento se pretende que este software sustituya totalmente a los autómatas reales, se pretende que los complemente. Una vez que un programa de control haya sido diseñado y simulado se podrá transferir a cualquier autómata real para que los alumnos comprueben el funcionamiento.
- Supervisión del Proceso: Se realizará por medio de la aplicación SCALIBUR NXG, en fase de desarrollo en el área ISA.

En la figura B.1 se muestra el conjunto de herramientas hardware-software que integran un LAV según la visión de GENIA. Para la supervisión del proceso se

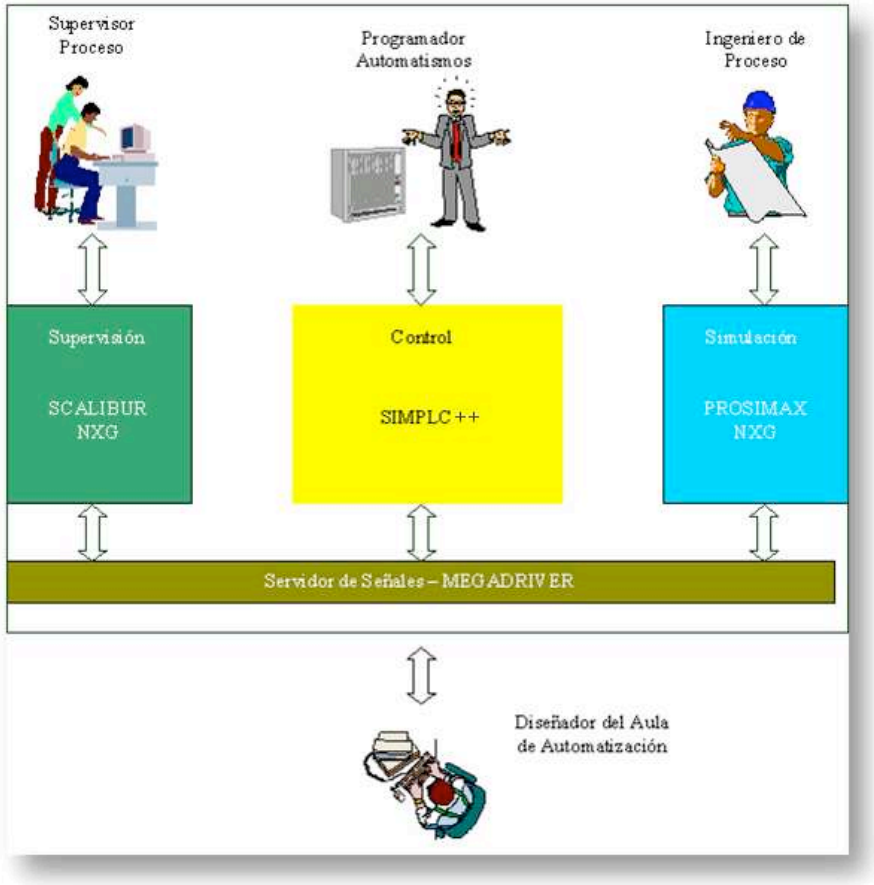

Figura B.1: Laboratorio de Automatización Virtual

tiene la herramienta software SCALIBUR NXG, en fase de desarrollo por GENIA. Para la parte de proceso se tiene la herramienta de simulación PROSIMAX NXG. Para la parte del diseño del programa de control se tienen la herramienta software SimPLC++. En esta herramienta se integra el módulo simulador que permite la simulación de un equipo de control que se adapta al estándar IEC 61131-3 de programación. La parte que se encarga de la comunicación entre todas las herramientas software es el servidor de señales MEGADRIVER.

B.1. MEGADRIVER

MEGADRIVER es una aplicación cliente-servidor que crea un sistema distribuido abierto que se comporta como un servidor de señales y ofrece un conjunto de servicios de red que permite a los clientes intercambiarse información.

Los servicios de red y sistema de comunicaciones con el servidor son encapsulados a los clientes en una librería de enlace dinámico. Cada servicio, dependiendo de las características físicas y de sobrecarga de la red, tienen un *"timeout"* configurable por el usuario de cada cliente, independientemente si clientes y servidor trabajan en una misma máquina (localhost) o en red.

MEGADRIVE permite la comunicación entre diversos dispositivos:

- Simuladores de PLCs propiedad de fabricantes como *"WinS5"* propiedad de Siemens Simatic S5. Para su utilización, el fabricante del simulador debe proporcionar un interfaz, un middleware que entienda a MEGADRIVE. Este tipo de configuración también es válida para comunicar PLCs reales con el módulo de simulación de SimPLC++.
- Diversos módulos de simulación de SimPLC++, tanto a nivel de local (localhost) como en máquinas distintas.
- Módulo de simulación de SimPLC++ y PROSIMAX

El protocolo de comunicación utilizado para la comunicación entre los clientes y MEGADRIVE es TCP/IP (Transmision Control Protocol/Internet Protocol) lo que asegura la independencia de las comunicaciones del sistema operativo y del hardware, ya que TCP/IP es un estándar disponible por defecto en todos los sistemas operativos con posibilidad de conexión en red.

MEGADRIVER cumple básicamente con tres funciones, a saber:

- Proveer de los servicios necesarios para permitir la interacción de los distintos clientes con el servidor.
- Proporcionar un servidor de señales.
- Conectividad entre los clientes

A continuación se pasan a describir cada una de estas funciones en detalle.

B.1.1. Servicios de Interacción

Dado que en un proyecto de automatización será empleado por varios clientes para llevar a cabo distintas tareas, es necesario que MEGADRIVER provea a los mismos de un conjunto de servicios básicos, a saber:

- Localización del Servidor. MEGADRIVE provee a los clientes con un servicio que les permite identificar la dirección de red donde está localizado el servidor con el que desean conectarse.
- Conexión. Una vez que un cliente ha localizado un servidor con el que desea conectarse, debe solicitar entrar en conexión con ese servidor por medio de un servicio que MEGADRIVE le proporciona.
- Desconexión. En cualquier momento el usuario puede determinar que ya no es necesario usar un cliente en la ejecución del programa de control. En este caso, el servidor proporciona un servicio a ese cliente que le permita desconectarse del mismo.

B.1.2. Servidor de señales

La principal interacción que se produce entre los distintos clientes de MEGADRIVE consiste en el intercambio de señales. Por señal se entiende una variable empleada en un programa de control, el identificador de un objeto de la simulación del proceso, un ítem asociado a una acción en una aplicación SCADA, un registro de la base de datos manejada por el paquete de elaboración de presupuestos, etc.

MEGADRIVE provee de servicios que permitan a los clientes añadir y borrar señales del programa de control, y consultar y modificar el valor de una señal. Este intercambio de información entre al menos los módulos simuladores del proceso y del equipo de control, debe ser tan rápido como sea posible e idealmente se podría decir que en tiempo real.

Para hacer este proceso lo más rápido posible MEGADRIVE introduce un mecanismo de aviso del cambio de valor de una variable del sistema a los módulos clientes necesarios como se muestra en la figura B.2. Esta optimización consiste en llevar un registro para cada cliente de qué señales de otros clientes son las que realmente le interesan. Ésto implica que el servidor de señales incorpora un tipo de servicio que se podría definir como *"subscripción"* y que permite a un cliente informar al servidor de qué variables le interesan, es decir, publica en MEGADRIVE las variables que son accesibles desde otro cliente.

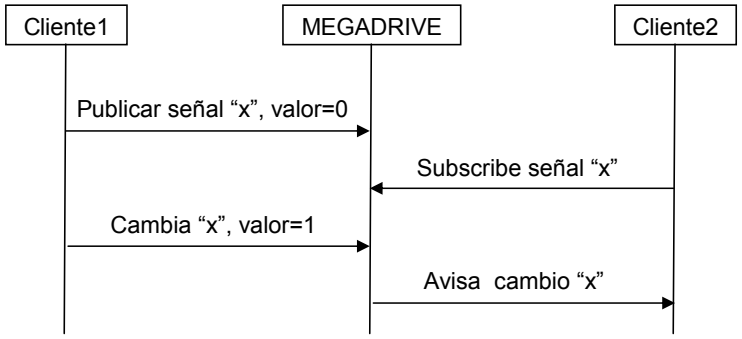

Figura B.2: Paso de mensajes entre clientes a través de MEGADRIVE

Para poder llevar a cabo una subscripción es necesario que todos los clientes conozcan que variables están disponibles en el proyecto, lo que exige que cada cliente tenga a su disposición un servicio para poder *"publicar"* en MEGADRIVE las variables que le interesa y que el resto de clientes las conozcan. Este servicio, en el caso de SimPLC++, es proporcionado por el módulo simulador en el que se asocian variables de salida del programa de control con etiquetas publicables de MEGADRIVE.

B.1.3. Conectividad

MEGADRIVE permite la conexión entre distintos tipos de equipos de control reales o de simulación. Ésto implica que para cada equipo de control que se desee conectar a MEGADRIVE, es necesario programar un cliente pasarela que permita traducir el protocolo MEGARIVE al protocolo de ese equipo o tarjeta de adquisición de datos empleada como interface de un proceso, como se muestra en la figura B.3.

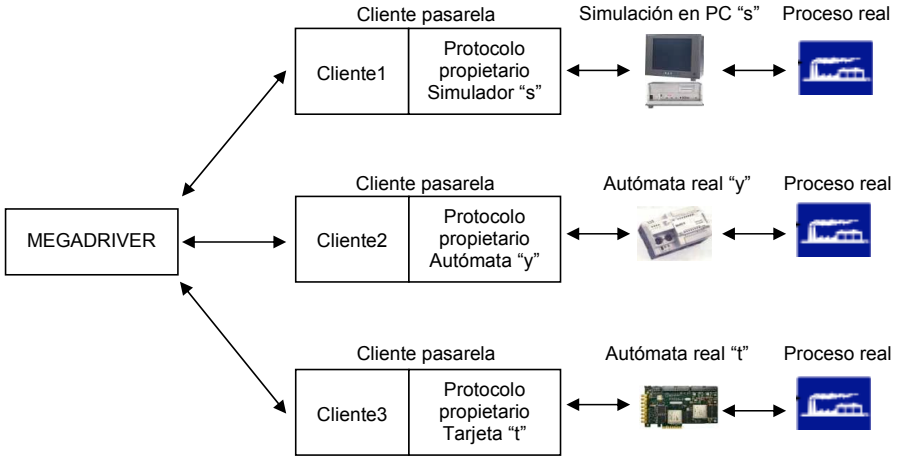

Figura B.3: Pasarela de conexión con MEGADRIVE propia de cada cliente

Para facilitar este tipo de conexión y no tener que desarrollar un nuevo cliente pasarela cada vez que se desee emplear un nuevo equipo de control o una nueva tarjeta de adquisición, MEGADRIVE emplea el protocolo OPC (OLE for Process Control). Este protocolo fue desarrollado inicialmente por Microsoft pero posteriormente fue liberado convirtiéndose en un estándar público mantenido y desarrollado por la fundación OPC (www.opcfoundation.org) de la cual son miembros más de 300 empresas multinacionales del sector informático e industrial, lo que demuestra su grado de aceptación e implantación.

El estándar OPC normaliza entre otras cosas la forma en que una aplicación puede acceder a los datos de otra. Ésto es de extrema importancia cuando se habla de equipos de control, ya que el no disponer de un mecanismo estándar de acceso a los datos de los mismos, se suscita la necesidad de tener que programar un driver de comunicaciones cada vez que se desee leer o forzar una de sus variables. Con

la llegada de OPC cada fabricante provee de un servidor OPC con el equipo, lo cual provoca que ya no sea necesario desarrollar un nuevo driver para cada nuevo equipo.

MEGADRIVE y el módulo simulador de SimPLC++ usan OPC, proporcionado un módulo cliente de OPC que permite leer y escribir variables de cualquier equipo de control o aplicación que provea de un servidor OPC sin tener que hacer ningún desarrollo adicional. En la figura B.4 se muestra como OPC afecta a la arquitectura que se observa en la figura B.3.

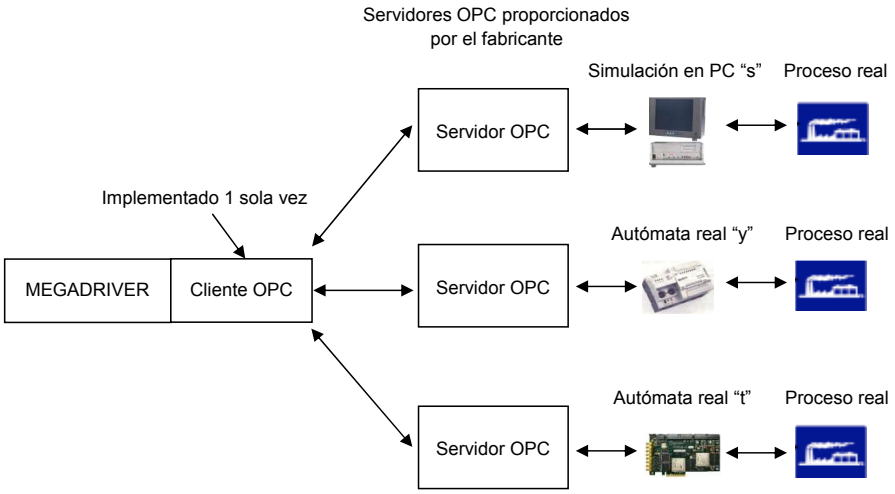

Figura B.4: Pasarela de conexión con MEGADRIVE con un servidor OPC

Con el fin de facilitar el posible intercambio de información con aplicaciones desarrolladas por terceras compañías, MEGADRIVE proporciona un módulo servidor de OPC lo que facilita el acceso a las variables manejadas en un proyecto de automatización.

B.2. PROSIMAX NXG

PROSIMAX NXG es una aplicación que se encarga de la simulación de procesos industriales guiados a través de programas de control de autómatas reales o simulados, por medio de objetos que representan los elementos físicos del sistema real que se está simulando. Estos objetos permiten al programador seguir visualmente

el cambio de estado de las distintas partes que conforman el sistema real a medida que se ejecuta el programa de control.

PROSIMAX NXG integra una completa librería de objetos tales como válvulas, bombas, sensores, etc, así como un editor de objetos personalizados. Toda esta funcionalidad permite al programador construir una versión del sistema que se desea simular por medio de la composición de los distintos objetos, asociando cada uno de estos objetos con las señales de entrada y de salida del proceso real o simulado.

Para poder realizar una simulación de una planta es necesario conectar los objetos entre sí, permitiendo, que los objetos puedan conocer: cuál es el estado en el que se encuentra otro objeto, pasar el valor de las propiedades de un objeto a otro y responder a eventos generados por otros objetos. Para ello, PROSIMAX NXG proporciona un servicio de conexión a través de MEGADRIVE que permite la publicación/suscripción de las variables que van a formar parte de los objetos que simulan la planta en PROSIMAX NXG, tal y como se muestra en la figura B.5.

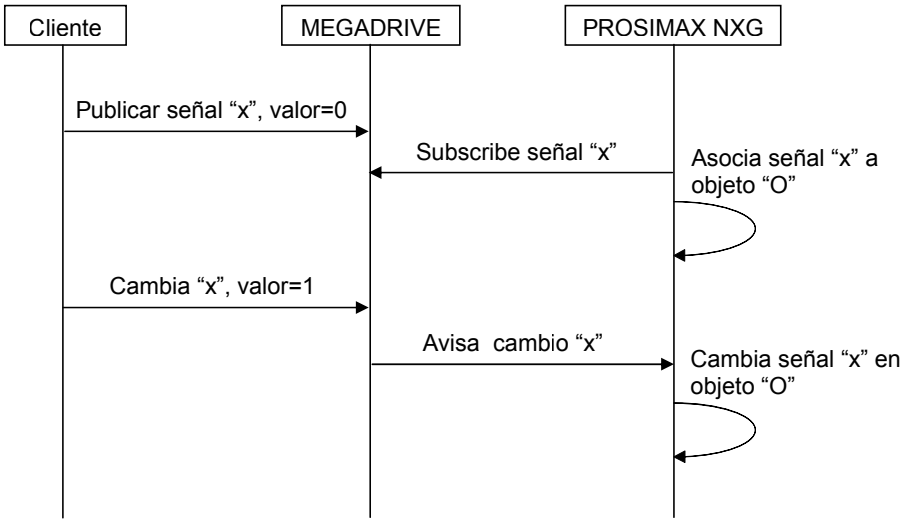

Figura B.5: Comunicación cliente-MEGADRIVER-PROSIMAX

Cuando se realiza una conexión, los objetos de PROSIMAX que forman parte de ella pasan a tomar el rol o papel de objeto productor de información y objeto consumidor de información, de tal manera que las propiedades del objeto que desempeña el papel de productor son conocidas por el objeto que desempeña el

papel de consumidor. El conocimiento de estas propiedades se logra gracias al mecanismo de publicación y suscripción de propiedades. De igual modo ocurre con los eventos, es decir, un objeto es capaz de responder a la ocurrencia de un evento de otro objeto en caso de que el segundo objeto esté subscrito al evento que es capaz de generar el primer objeto.

B.2.1. Subscripción de eventos y propiedades

Cuando el usuario realiza la conexión entre dos objetos, indica qué propiedades del objeto consumidor quedarán subscritas a las propiedades que publica el objeto productor. Gracias a la realización de la conexión y a la suscripción de propiedades, PROSIMAX NXG puede realizar el trasiego de información entre los objetos en el estado de simulación.

Los conceptos de suscripción y publicación son aplicables también a los eventos que un objeto puede lanzar hacia otros objetos. Cuando se realiza la conexión entre objetos, el objeto consumidor de la conexión puede subscribirse a cualquiera de los eventos que el objeto productor publica, de tal forma que cuando el objeto productor informa a la aplicación de la ocurrencia de un evento, ésta sabe que tiene que informar a los objetos subscritos a ese evento.

Cuando el sistema se encuentra en el estado de simulación, los objetos productores generan información que será consumida por los objetos consumidores.

B.2.2. Estados de PROSIMAX NXG

PROSIMAX NXG consta de tres estados, a saber:

1. Estado de edición. El estado por defecto cuando se abre o se crea un proyecto es el de edición. En este estado se puede realizar la configuración del aspecto del sistema, la ubicación de los objetos dentro de ésta, la configuración del diseño gráfico y el comportamiento de dichos objetos. A grandes rasgos se puede decir que las acciones que se pueden realizar en este estado son las siguientes:

    - Añadir objetos a la planta.
    - Eliminar objetos de la planta.

- Editar las propiedades de un objeto. Por editar las propiedades de un objeto se entiende la configuración de los valores de las propiedades que definen la representación gráfica y el comportamiento de dicho objeto. Como por ejemplo la rotación del objeto, el tipo de objeto a generar (en el caso de un generador de objetos), la velocidad máxima (en el caso de una carretilla), etc.

En este estado se pueden iniciar las comunicaciones con un servidor de señales, siendo posible realizar las acciones permitidas por dicho servidor. Desde este estado se podrá pasar al estado de conexión o al estado de simulación.

2. Estado de conexión. Este estado permite realizar la conexión de los objetos, es decir, se realiza la suscripción de las propiedades de los objetos que desempeñan el papel de consumidores de las propiedades publicadas por los objetos que desempeñan el papel de productores. Además, se puede visualizar y editar las conexiones de un objeto productor.
   Al igual que en el estado de edición, en este estado se pueden iniciar las comunicaciones con un servidor de señales, siendo posible realizar las acciones permitidas por dicho servidor.

3. Estado de simulación. Este estado permite realizar la simulación de la planta industrial diseñada. En este estado, la intervención del usuario se reduce a la interacción con aquellos objetos de la planta que permitan modificar su comportamiento como por ejemplo, cambiar el sentido del movimiento, vaciar carretillas, etc.

Apéndice C

Unity Developer's Edition (UDE)

Para perfeccionarme, yo no debo olvidar, ni por un solo momento, que debo estar al servio del mundo
Jigoro Kano

Unity Developer's Edition (UDE) es un software especializado para los ingenieros de desarrollo de automatización que ofrece acceso a todos los servidores de objetos de software de Unity Pro y Unity Studio de la marca Scheneider. Permite desarrollar soluciones a medida para facilitar la colaboración de Unity con el sistema CAD eléctrico, el generador de códigos y variables, etc. Unity ofrece interfaces de programación para entornos de desarrllo lo que facilita la colaboración entre Unity y cualquier otra herramienta de software de terceros.

Unity proporciona una serie de servidores. Cada servidor gestiona su propia base de datos permitiendo obtener información en tiempo real (ver figura C.1). Estos servidores son:

- Unity Pro Server.

- Unity Studio Manager Server.
- Unity Library Server.
- OPC Factory Server.

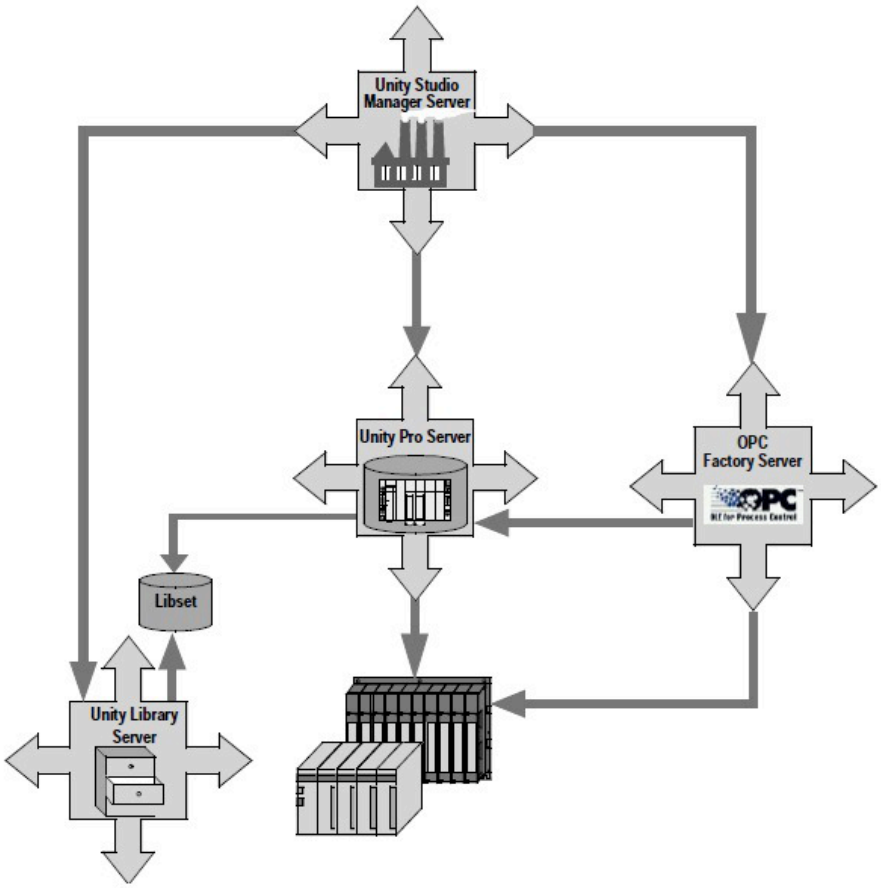

Figura C.1: Servidores de Unity Developer's Edition

Los servidores Unity proporcionan sus servicios a través de interfaces COM. El intercambio de información con los servidores puede ser realizado de dos formas:

- Dinámica mediante una interface COM/DCOM (Distributed Component Object Model).
- Estática mediante intercambio de datos en formato XML.

C.1. Unity Pro Server (UPS)

Unity Pro Server (UPS) está diseñado para su uso con herramientas externas para acceder o modificar la información relativa a un proyecto Unity.

Ofrece a las aplicaciones cliente los métodos y propiedades que les permitan:

- Manejar proyectos Unity.
- Crear, leer o modificar proyectos Unity.
- Manejar la consistencia de los datos entre las aplicaciones cliente y una o varios proyectos Unity.
- Acceder a Unity Pro Server en modo GUI.

C.1.1. Gestión de aplicaciones cliente

Un cliente puede invocar varias instancias de Unity Pro Server, pero un determinado proyecto Unity sólo podrá estar asociado a una determinada instancia del UPS. En un momento dado, sólo un cliente tiene derechos de escritura sobre el proyecto Unity. El uso de la misma instancia de UPS por varios clientes permite a estos clientes saber qué modificaciones se han realizado sobre el proyecto (ver figura C.2).

Dentro de Unity Pro Server se encuentra el componente Unity Pro Broker (UPB) encargado de gestionar una o varias instancias así como los aspectos mono o multicliente de un proyecto Unity. Se ofrecen dos vías para el acceso a los proyectos mediante servicios destinados a:

- Creación de un proyecto.
- Apertura de un proyecto existente.

Cuando varios clientes acceden a un mismo proyecto sólo se permite el acceso en modo escritura a un cliente a la vez, de esta forma se pretende evitar que un cliente deshaga lo que ha hecho otro. Las aplicaciones cliente deben solicitar a UPS el acceso de escritura, pero si ya existe una conexión con permisos de escritura se devuelve un error.

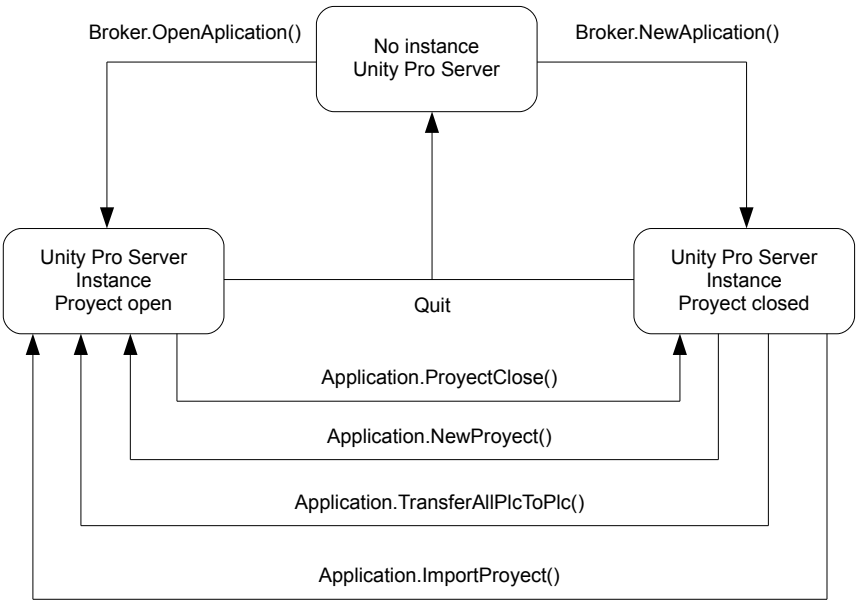

Figura C.2: Gesión del proyecto con UPS

Si el acceso de escritura queda libre se envía una notificación, de manera que otro cliente la pueda obtener. Un cliente con acceso en modo lectura que desea cambiar al modo escritura tiene que obtener una nueva interfaz del objeto del proyecto con derechos de escritura (ver figura C.3).

Para que los cambios realizados en un proyecto surtan efectos debe llevarse a cabo una operación de guardado (se salvan los cambios en un fichero .STU).

C.1.2. Gestión de la consistencia de datos

Un proyecto Unity es identificado por varias firmas constituidas por un número formado por varios campos. Estas firmas son generadas automáticamente por el sistema y cualquier modificación del proyecto dará lugar a la modificación de una o varias firmas. En un proyecto se pueden encontrar tres tipos de firmas:

- File Signature. Se corresponde con el proyecto. Ésta se guarda en el archivo . STU y se crea la primera vez que el proyecto se guarda, actualizandose con cada salvado posterior.

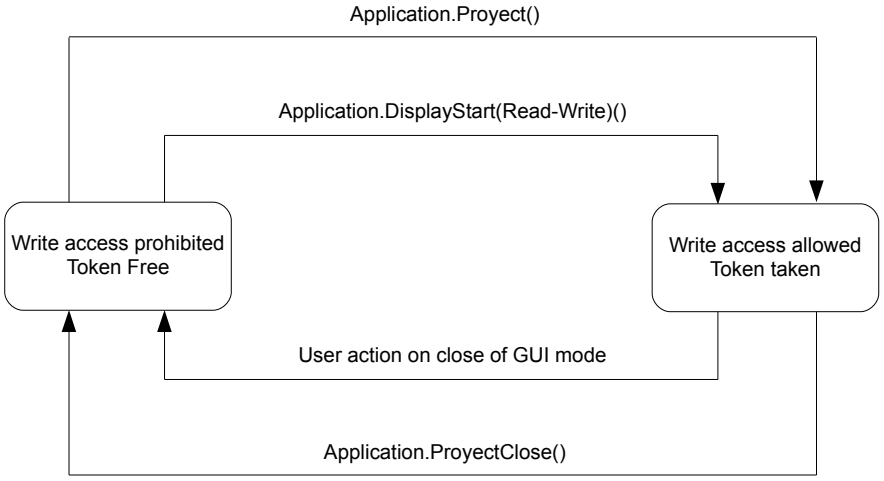

Figura C.3: Permisos de acceso escritura-lectura del UPS

- Current Signature. Se corresponde con el proyecto una vez que ha sido abierto por Unity Pro y cargado en la memoria del ordenador. La primera vez que se abre el proyecto, la *"Current Signature"* es idéntica a la *"File Signature"* pero se actualiza cada vez que se modifica el proyecto.

- Target Signature. Incrustada en el código binario del proyecto que se carga en el PLC. Se modifica en cada transferencia del proyecto.

C.1.3. Notificaciones

Una notificación corresponde a un evento generado por Unity Pro Server y dirigido a las aplicaciones cliente cuando ciertas operaciones se han disparado o ejecutado. Los mecanismos de notificación son:

1. UPS notifica a sus clientes mediante eventos.

2. UPS notifica a sus clientes asíncronamente.

3. UPS notifica a sus clientes y devuelve la firma del proyecto.

Notificaciones por eventos

El evento sólo se envía cuando la modificación es seguido por:

- Un salvado del proyecto.
- Una compilación del proyecto.
- Una transferencia al PLC.

Otras tres situaciones en las que se envía una notificación :

- Cuando un acceso de escritura es liberado, los clientes son alertados.
- Cuando Unity Pro Server inicia en modo GUI.
- Cuando Unity Pro Server finaliza en modo GUI.

Notificaciones asíncronas

Después del envío de la notificación, Unity Pro Server no espera a que el cliente procese el evento antes de continuar. Ciertos eventos se generan en el inicio y el final de una operación.

El cliente no interrumpe la acción en curso y la notificación se procesa posteriormente.

Notificaciones con retorno de firma

Después de que se lance un evento como resultado de una modificación del proyecto (guardar, compilar, transferir), la notificación devuelve la firma actual del proyecto para que el cliente pueda identificar las partes del proyecto que han cambiado y volver a sincronizar.

Sincronización

Existen dos mecanismos de sincronización diferente, pero ambos se basan en la firma:

- Sincronización estática. Basada en el afirma y requiere una fase de aprendizaje
- Sincronización dinámica. Basada en la firma y las notificaciones. Requiere una sesión del UPS en una aplicación Unity Pro abierta. La conexión entre el cliente y el servidor se debe mantener y no cerrar nunca.

Gestión de errores

Unity Pro Server dispone de tres mecanismos de gestión de errores:

- Retorno estándar de errores. Un método devuelve un valor de error (Hresult) y un objeto error (ISupportErrorInfo).
- Parámetros de salida. Con un numero de error y una cadena de caracteres.
- El error se visualiza en la ventana de salida del Unity Pro.

Para el retorno estándar utiliza ISupportErrorInfo y ErrorInfo. La descripción del error devuelto está en el idioma instalado en Unity Pro. En la mayoría de los casos, los servicios del UPS devuelven un HRESULT basado en este mecanismo. En el caso de usar parámetros de salida, uno de los parámetros es un VARIANT conteniendo una tabla de códigos de error y una tabla de mensajes de error.

Los errores mostrados en la ventana de salida del Unity Pro (GUI) están disponibles en el objeto OutputWindow con el servicio GetContentsAsString (). Este servicio devuelve una tabla BSTR en un VARIANT. Cada BSTR corresponde a una línea que aparece en la ventana de salida.

Modelo de objetos

El cierre de una sesión Unity Pro Server depende de dos condiciones:

1. Todos los objetos debe ser liberados.
2. No hay ningún cliente conectado.

El diagrama del modelo de objetos del UPS puede observarse en la figura C.4.

Cabe destacar que en la nomenclatura empleada en el modelo las colecciones se diferencian porque su nombre terminan en *"s"*. Por ejemplo, EFBs es una colección de objetos de la clase EFB.

C.2. Unity Studio Manager Server (USMS)

El Unity Studio Manager Server (USMS) permite el acceso a la información compartida en el ámbito de multi-aplicación.

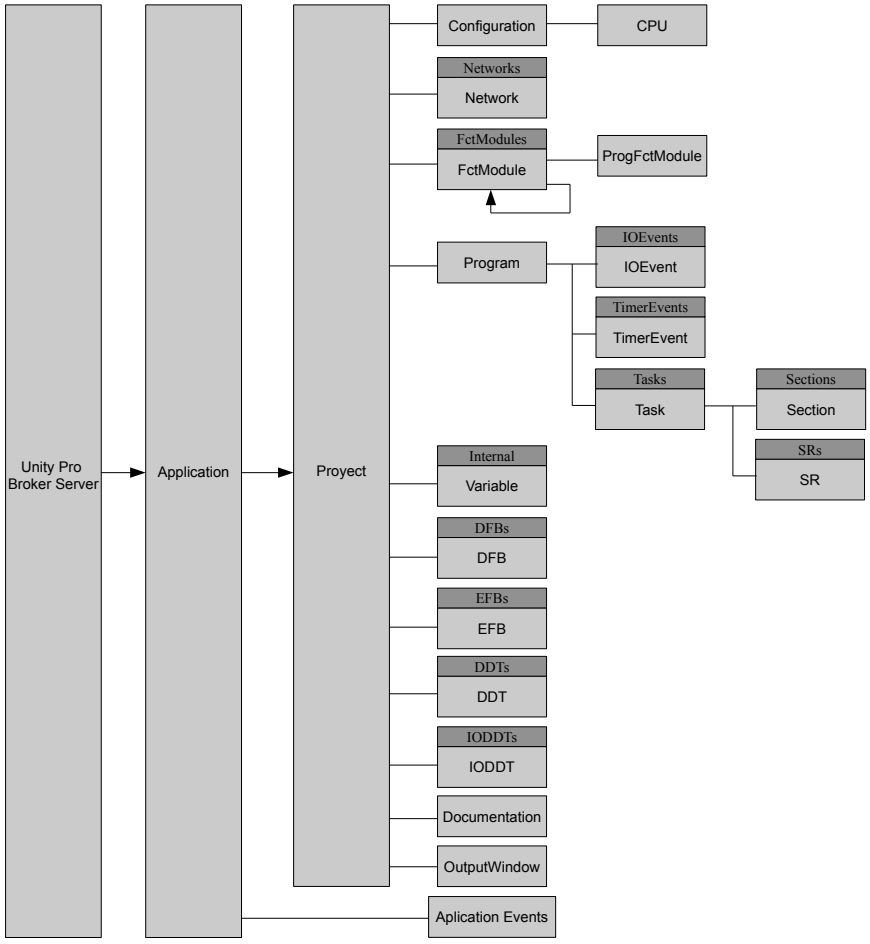

Figura C.4: Modelo de objetos del UPS

Es una herramienta en *"tiempo de construcción"* lo que permite el acceso a nivel de la planta de:

- Información topológica.
- Información de la funcionalidad de las entidades.
- Información de datos globales.
- Información de la aplicación.

La información manejada por USMS puede agruparse en las siguientes categorías principales:

- Estructura funcional. Diseño y seguimiento de navegación en la estructura funcional de la aplicación distribuida.
- Gestión de la comunicación. Gestión variable de la red.
- Gestión de aplicaciones. Gestión del repositorio para todas las partes de la aplicación distribuida y controles de coherencia.
- Información topológica. Diseño topológico de la red y modificaciones. Aplicación, nombre y direcciones asignadas.

C.2.1. Gestión de aplicaciones

Unity Studio Manager permite que varios clientes puedan trabajar en la misma planta física. Abrir una planta implica cargar el proyecto de la planta en la memoria. Las modificaciones sólo son vistas por el cliente que las realiza.

Después de una operación de guardado, todos los cambios se registran en el archivo de proyecto. Cada cliente que trabaja en una planta dada, lo hace sobre una copia propia de la base de datos y se ocupa de un modelo de objetos independiente. Si un cliente cambia el modelo de objetos, los cambios sólo se realizan en la planta de trabajo del cliente.

Los cambios no son visibles para cualquier otro cliente antes que la planta se guarde. Cuando la planta se guarda, la planta de trabajo se copia en el lugar de

almacenamiento de la planta. Después, se notifica a todos los clientes que trabajan con esa planta que se ha guardado.

Cada cliente puede solicitar individualmente al USMS actualizar a la última versión, en cuyo caso se copia la planta almacenada en la planta de trabajo de dicho cliente. Todo el modelo de objetos a excepción del objeto planta en sí mismo, deja de ser válido y debe ser creado de nuevo.

Una instancia del Unity Studio Manager Server puede servir para varios proyectos de plantas. Sólo puede haber un cliente trabajando en modo escritura con el proyecto de la planta. Si el cliente solicita el acceso de escritura y otro usuario ya ha abierto la planta con derechos de acceso para escritura, esta última petición será rechazada. Cuando el testigo de escritura es liberado, no se notifica a los otros clientes.

C.2.2. Notificaciones

A fin de no inundar el cliente con mensajes (si se realizan múltiples cambio), los eventos no se envían más que una vez en un intervalo de tiempo mínimo. Unity Studio Manager Server recopila todos los cambios realizados durante este intervalo de tiempo y envía los eventos adecuados después de que el tiempo mínimo que haya transcurrido.

Cada evento indica al cliente la lista de todos los cambios que se hicieron desde el evento anterior. Cada cambio consta de la siguiente información:

- El interfaz COM de los objetos modificados.
- El tipo de modificación (crear, borrar, modificar).

USMS también proporciona servicios para detener y reanudar el envío de eventos con el fin de permitir a un cliente hacer un gran número de cambios sin la necesidad de gestionar todos los eventos correspondientes. Después de reanudar el envío de un evento USMS utiliza otro evento para indicar que es necesaria una actualización completa.

C.2.3. Modelo de objetos

Cada colección ofrece iteradores estándar (a través de la propiedad _NewEnum) que se pueden utilizar para recorrer todos los elementos de la colección. Las colecciones en el modelo de objetos Unity Studio Manager Server (ver figuras C.5 y C.6) pueden cambiar durante su recorrido.

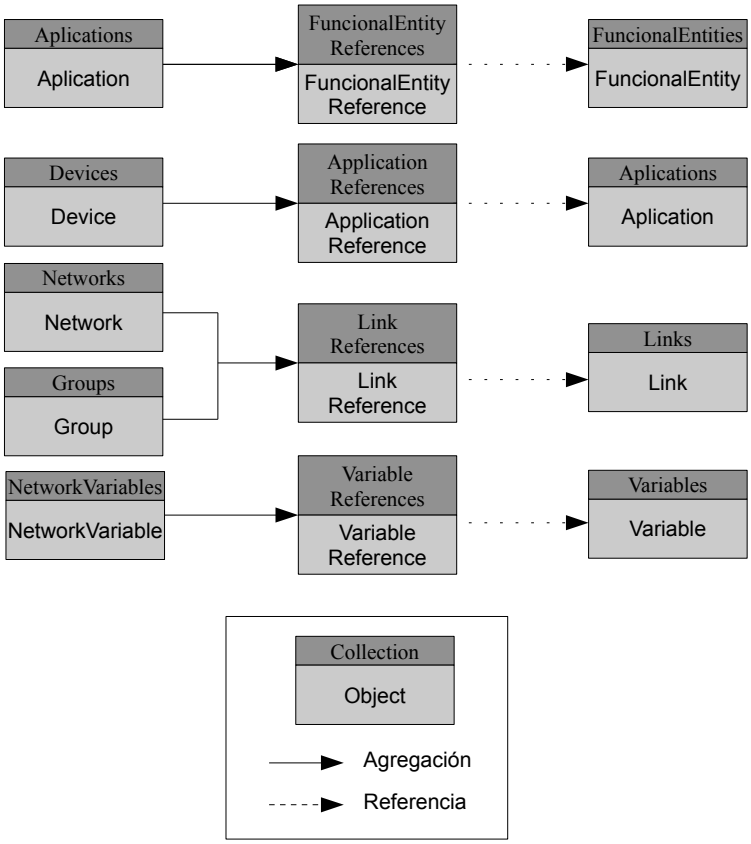

Figura C.5: Modelo de objetos del USMS

C.3. Unity Library Server (ULS)

El Unity Library Server (ULS) permite a las aplicaciones cliente poder leer los elementos libset.

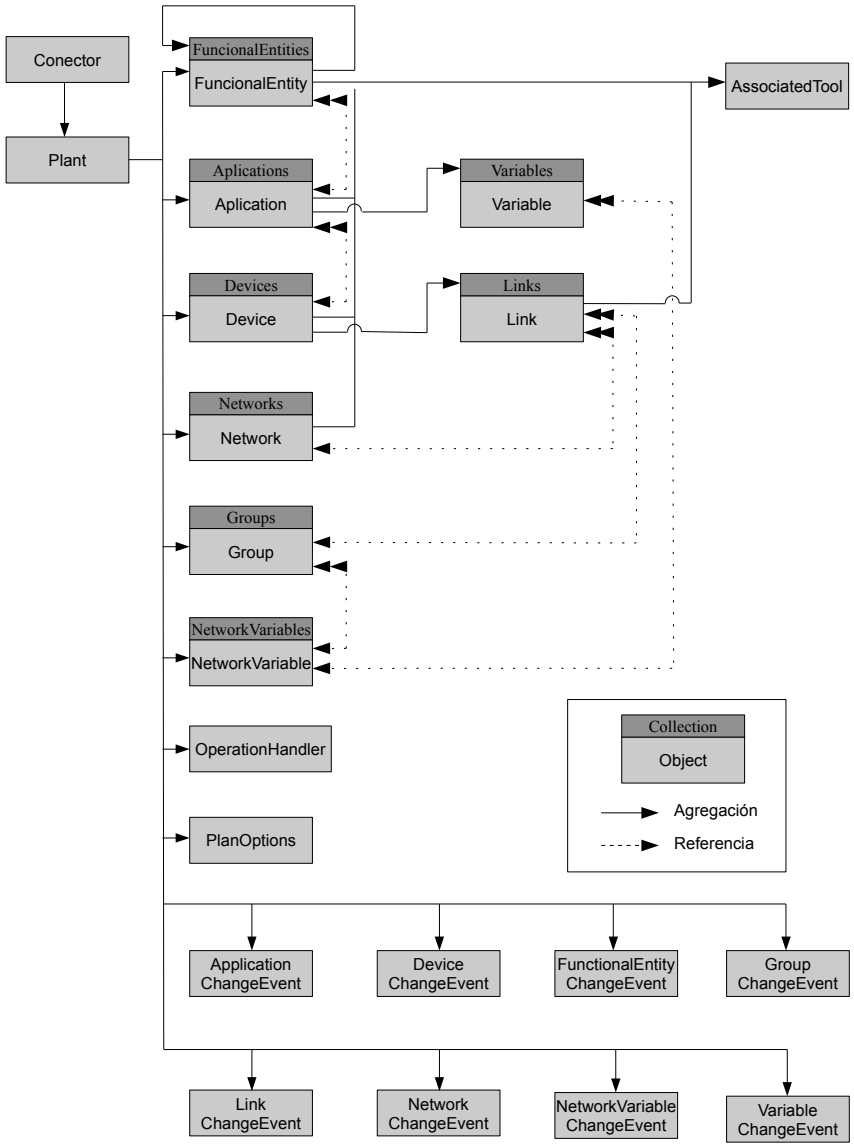

Figura C.6: Modelo de objetos del USMS

Los clientes pueden acceder a los siguientes elementos:

- Estructuras de datos definidas por el usuario (DDT).
- Bloques funcionales definidos por el usuario (DFB).
- Bloques funcionales elementales (EFB).

Una instancia del ULS puede servir a varios libset y a varios clientes simultáneamente. Como el Unity Pro Server, Unity Library Server copia los objetos usados por los clientes en su memoria.

C.3.1. Modelo de objetos

El modelo de objetos que se implementa en el ULS se puede observar en la figura C.7.

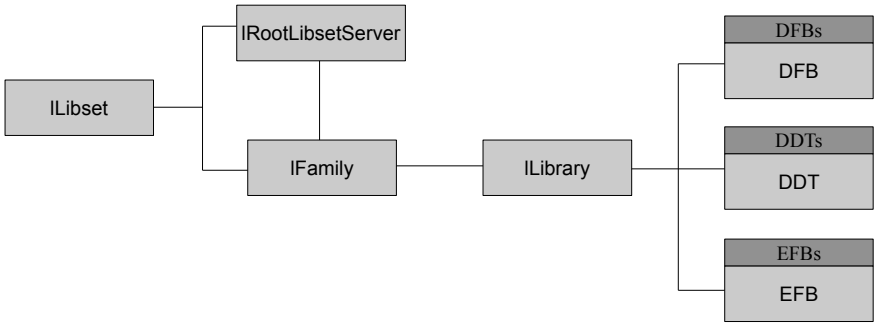

Figura C.7: Modelo de datos del ULS

El acceso a los diversos elementos de un libset puede llevarse a cabo de acuerdo a la tabla C.1:

Por otro lado, es posible acceder a la información de un elemento determinado del libset tal y como se muestra en la tabla C.2:

También es posible acceder a un elemento del libset directamente como se puede observar en la tabla C.3:

Paso	Acción
1	Utilizando el objeto *IRootLibsetServer* con el método *OpenApplication*. Resultado: devuelve un objeto *libset*.
2	Utilizando el objeto *ILibset*. Resultado: devuelve los diversos elementos del *libset* (biblioteca, familias, etc).
3	El uso de objetos *Library*. Resultado: devuelve varias colecciones de familias.
4	El uso de objetos *Family*. Resultado: devuelve varios elementos de la colección familia.
5	Utilizando objetos *DDT*, *DFB* y *EFB*.

Cuadro C.1: Acceso del ULS al libset

Paso	Acción
1	Utilizando el objeto *IRootLibsetServer* con el método *OpenApplication*. Resultado: devuelve un objeto *libset*.
2	El método *ListAttributesSupported* del objeto *ILibset* produce una lista de diferentes tipos de información para un tipo de elemento. Resultado: devuelve un objeto *libset*.
3	Utilizando el método *ListAttributesValues* del objeto *ILibset*. Resultado: devuelve información sobre los elementos *libset* según el tipo.

Cuadro C.2: Acceso a la información de un elemento libset

Paso	Acción
1	Utilizando el objeto *IRootLibsetServer* con el método *OpenApplication*. Resultado: devuelve un objeto *libset*.
2	Utilizando el método *GetElement* del objeto *ILibset*. Resultado: devuelve el elemento *libset* acuerdo con el tipo (Objeto tipo *DDT*, *DFB* o *EFB*).

Cuadro C.3: Acceso a un elemento libset

C.4. OFS Server

OFS (OPC Factory Server) es un software servidor que proporciona acceso en tiempo real a los datos sobre plataformas PLC Premium/Quantum mediante aplicaciones cliente. A tal efecto, el servidor OFS ofrece a las aplicaciones cliente un conjunto de servicios. OPC (OLE for Process Control) es un estándar basado en tecnología Microsoft OLE/COM para promover interoperabilidad entre los sistemas de automatización y las aplicaciones de control. OFS (OPC Factory Server) es el nombre del producto Schneider que admite el estándar OPC para los protocolos Modbus y XWay. Las diversas bases de datos del PLC son gestionadas por una única instancia del OFS Server, lo cual es posible gracias a la herramienta de configuración OFS. Una instancia del OFS Server puede servir a varios clientes que quieren acceder a las diferentes bases de datos del PLC.

C.5. Relación entre servidores

C.5.1. Unity Manager Studio Server (UMSS)

Unity Manager Studio Server (UMSS) es cliente de Unity Library Server para acceder a los nombres de los DDT desde las variables que instáncian su posible creación. UMSS es cliente de Unity Pro Server cuando es necesario:

- Crear o abrir un proyecto Unity.
- Leer o definir las propiedades de un proyecto Unity.
- Leer o instanciar datos globales.
- Leer o definir los módulos funcionales.
- Leer o definir las redes de comunicaciones.

C.5.2. Unity Pro Server

Aunque Unity Pro Server no está conectado con el resto de servidores si depende del contenido de los libset, para crear instancias de DDT, DFB y/o EFB, y de las bases de datos del sistema de seguridad cuando se inician sesiones con perfiles.

C.5.3. OFS Server

OFS Server es un cliente del Unity Pro Server para lectura. Esta relación sólo existe si se define un alias con la herramienta de configuración del OFS Server, refiriéndose al archivo .STU.

Por último, destacar la relación con el PLC, considerando al OFS SERVER como cliente del PLC para lectura o escritura de los valores de las variables.

Apéndice D

Resultados experimentales (II)

Lo que se gana por la fuerza, dura poco.
Lo que se gana por la razón y el dialogo,
dura para siempre.
Hironori Oh Tsuka

D.1. Introducción

En este apendice se presenta una ampliación del ejemplo mostrado en el capítulo 5 de la aplicación de MIOOP en la programación de un sistema de control de eventos discretos usando SimPLC++.

A diferencia del capítulo 5, donde se muestran varios trozos de código significativos de las subestaciones programadas siguiendo el paradigma de programación estructurado y el OO, así como los tiempos de ejecución de cada estación. En este apendice se muestra toda la programación orientada a objetos de la estación de

montaje de rodamientos (estación 2) así como su traducción a lenguaje IL estándar con la norma IEC 61131, traducción que en el capítulo 5 no se muestra. Así mismo, se presenta también un trozo de código del *"transfer"* encargado de ejecutar cada una de las estaciones. Los métodos que se usan en el mencionado código del *"transfer"* servirán al lector para poder ver el tipo de traducción a lenguaje IL que realiza SimPLC++, tanto del polimorfismo como de la sobrecarga de métodos, siguiendo los principios presentados en el capítulo 3 (ver secciones 3.21 y 3.25 respectivamente).

A continuación, se hace una breve descripción del sistema y de la estación de montaje de rodamientos a modo de recordatorio.

D.1.1. Descripción del sistema

El proceso que se pretende automatizar se corresponde con la célula de fabricación fexible FMS 200 del proveedor multinacional con sede en Japón SMC, especialista en el desarrollo de componentes neumáticos.

Una imagen de esta estación se puede apreciar en la figura D.1. En la figura D.2 se muestra el aspecto completo de la célula FMS 200 cuyo cometido consiste en la producción de piezas por ensamblado de los componentes mostrados en la figura D.3. Entre los componentes mostrados aparecen enmarcados en una elipse la base y el rodamiento que la estación 2 debe insertar en ella. Para llevar a cabo este proceso, la citada estación cuenta con los componentes descritos en el apartado D.2 los cuales deben ser comandados de la manera indicada en la sección D.3.

D.2. Componentes del proceso

La estación 2 está dividida en cinco secciones claramente diferenciadas, a saber: alimentación, trasvase, medición, expulsión y evacuación de rodamiento. Los procesos de los que se compone la estación 2 son los que se detallan a continuación en las siguientes subsecciones.

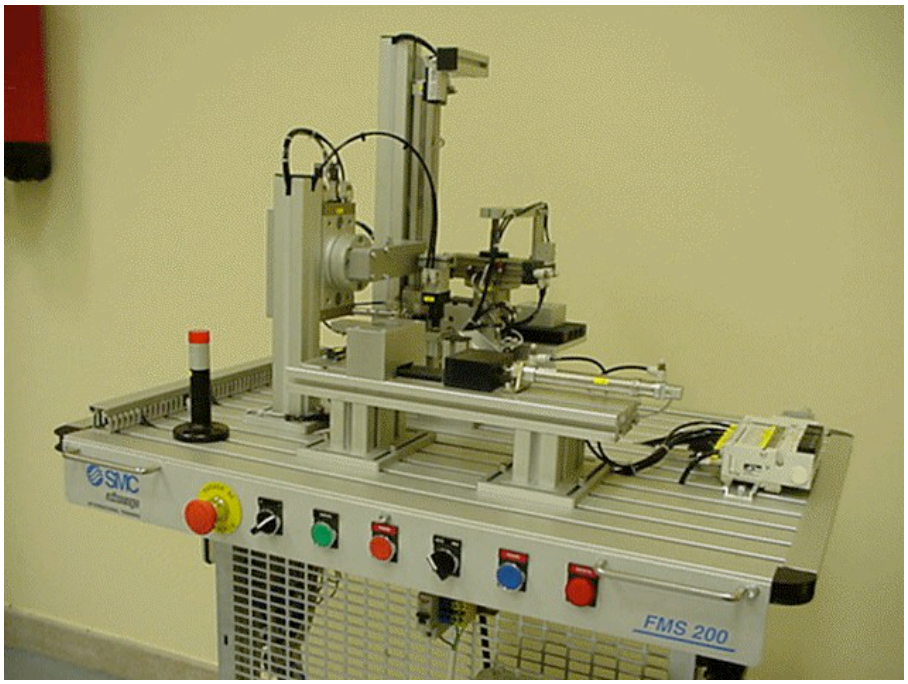

Figura D.1: Vista de la Estación de Inserción de Rodamiento

Figura D.2: Vista de la Célula de Fabricación Flexible FMS 200

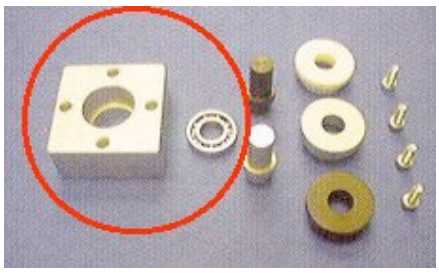

Figura D.3: Vista de los Componentes a Ensamblar por la FMS 200

D.2.1. Alimentación de rodamiento

Como se aprecia en la figura D.4, la sección de alimentación está formada por un cilindro de doble efecto comandado por una electroválvula monoestable. Para determinar cuándo el cilindro está completamente recogido, se dispone de un sensor de posición de tipo magnético.

Por último, el alimentador se completa con un sensor de posición de tipo electromecánico empleado para detectar cuándo se terminan los rodamientos en el alimentador de petaca.

D.2.2. Trasvase de rodamiento

Como se aprecia en la figura D.5, la sección de trasvase está formada por un cilindro de doble efecto comandado por una electroválvula biestable. Este cilindro se emplea para pasar el rodamiento desde la plataforma del alimentador, hacia la plataforma de medida del medidor. Para ello, el trasvasador dispone de una pinza neumática que en realidad es un cilindro de doble efecto gobernado por una electroválvula monoestable. El cilindro de trasvase, que es giratorio, puede estar en tres posiciones posibles (sobre la plataforma de alimentación, sobre la plataforma de medida o en el medio de estas dos posiciones). Para determinar en qué posición se encuentra exactamente se dispone de tres sensores de posición de tipo magnético.

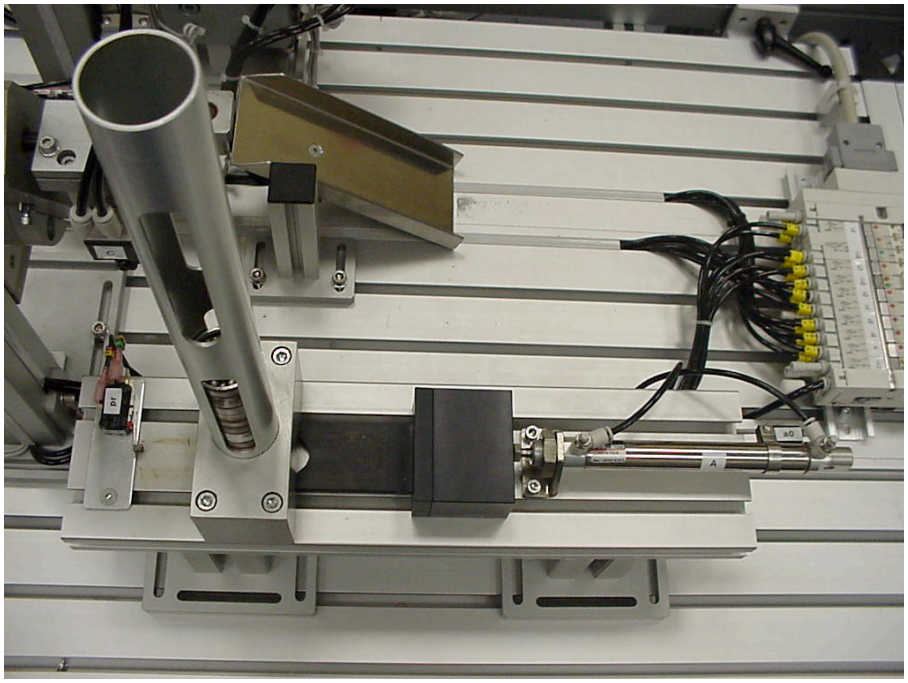

Figura D.4: Vista de los Componentes de la Sección de Alimentación de Rodamiento

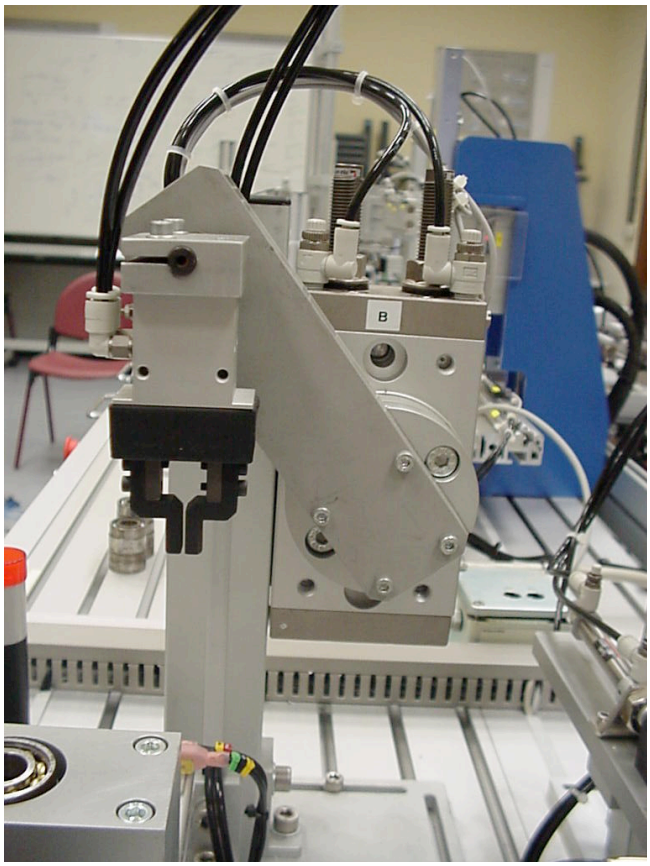

Figura D.5: Vista de los Componentes de la Sección de Medición de Rodamiento

D.2.3. Medición de rodamiento

La sección de medición está diseñada para determinar la altura del rodamiento con el objeto de asegurarse que el rodamiento insertado en la base es de la altura adecuada. Para ello, se dispone de varios componentes como se aprecia en la figura D.6. La plataforma sobre la que el trasvasador deja los rodamientos que provienen de la sección de alimentación, está acoplada a un cilindro sin vástago de doble efecto el cual se acciona por medio de una electroválvula biestable. Al ser accionado este cilindro, la plataforma se elevará hasta llegar a un punto en el que se encuentra un sensor palpador analógico, el cual facilita una señal proporcional a la medida del rodamiento. El cilindro dispone de dos finales de carrera de tipo magnéticos para detectar cuándo el mismo llega a la parte superior e inferior de su carrera.

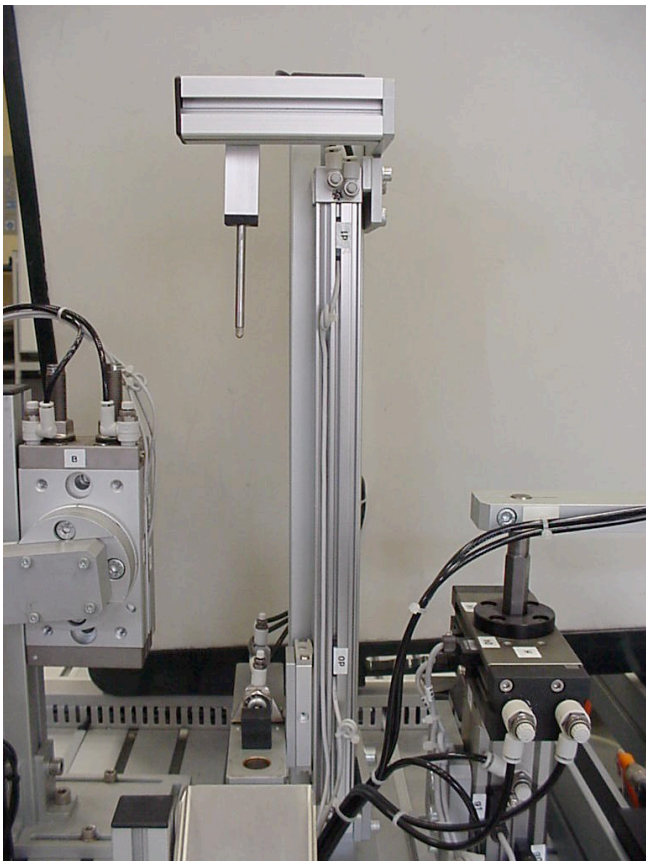

Figura D.6: Vista de los Componentes de la Sección de Trasvase de Rodamiento

Para asegurarse de que el rodamiento se encuentra en la posición exacta en la que el medidor podrá llevar a cabo su labor correctamente, la plataforma cuenta con un cilindro centrador de tipo simple efecto comandado por una electroválvula monoestable.

D.2.4. Expulsión de rodamiento

Una vez que el rodamiento ha sido medido, si la medida del mismo no coincide con la medida seleccionada por el operador a través del selector de dos posiciones correspondiente del panel de mando, el rodamiento debe ser expulsado. Para ello, la estación está provista de un cilindro de doble efecto gobernado por una electroválvula monoestable, tal y como se muestra en la figura D.7.

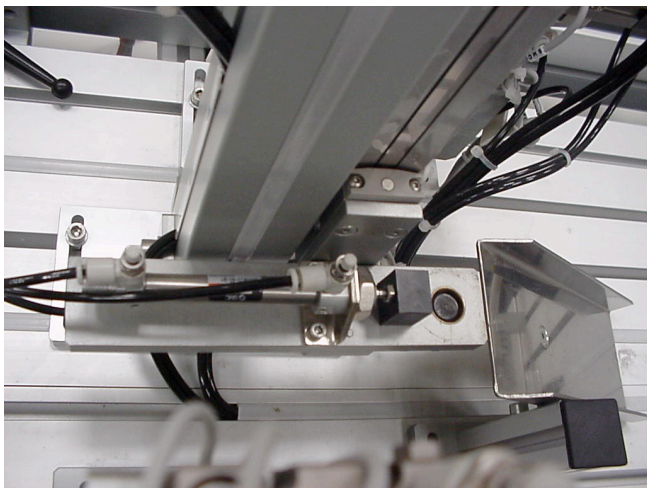

Figura D.7: Vista de los Componentes de la Sección de Expulsión de Rodamiento

D.2.5. Evacuación del rodamiento

Si la medida del rodamiento es la adecuada, entonces el rodamiento debe ser insertado en la base que está situada en la posición adecuada sobre el sistema de transferencia denominado *"transfer"*. Para ello, la sección de inserción de rodamiento está formada, como se aprecia en la figura D.8, por un cilindro giratorio de simple efecto gobernado por una electroválvula monoestable. Este cilindro lleva

adosada una pinza que no es más que un cilindro de simple efecto comandado por una electroválvula monoestable.

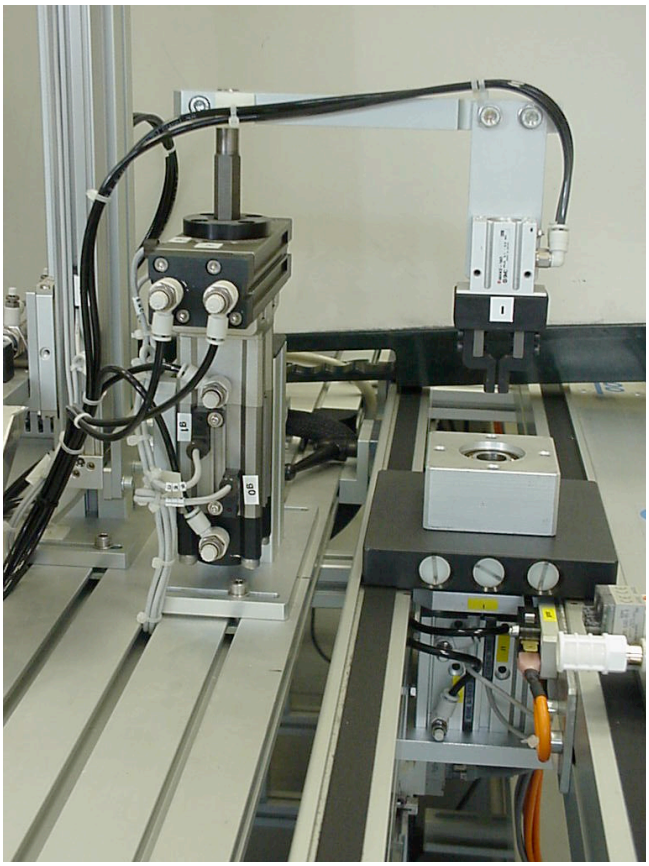

Figura D.8: Vista de los Componentes de la Sección de Expulsión de Rodamiento

Estos dos cilindros están montados sobre un tercer cilindro de simple efecto accionado por una electroválvula monoestable. Su cometido es el de elevar verticalmente a los otros dos cilindros lo suficiente como para permitir la correcta inserción del rodamiento en la base. El conjunto se complementa con dos pares de sensores de posición de tipo magnético que permiten detectar cuándo el cilindro giratorio se encuentra en la posición del medidor o sobre la base que está situada en el *"transfer"*, y para detectar cuándo el cilindro elevador está arriba o abajo.

D.2.6. El panel de mando

Cada una de las estaciones de la FMS 200 viene equipada con un sencillo pupitre de mando como el mostrado en la figura D.9.

Figura D.9: Esquema del Panel de Mando de la Estación de Inserción de Rodamiento

En él se pueden apreciar los siguientes componentes:

- Un interruptor de tipo seta de emergencia.
- Un pulsador normalmente abierto de color verde etiquetado con el nombre *"Marcha"*.
- Un pulsador normalmente abierto de color rojo etiquetado con el nombre *"Parada"*.
- Un pulsador normalmente abierto de color azul etiquetado con el nombre *"Rearme"*.

- Un selector de dos posiciones que permite escoger entre dos posibles modos de funcionamiento: manual y automático.
- Un selector de dos posiciones que permite al operador indicar el tipo de rodamiento (alto o bajo) que desea sea insertado en la base.
- Una lámpara de color rojo para señalizar cualquier posible problema en el sistema que requiera de un rearme para continuar con la producción.

D.3. Modos de funcionamiento

En el manual que acompaña a la estación de inserción de rodamiento de la célula FMS 200 se detallan una serie de posibles modos de funcionamiento que son los que se han decidido implementar y que son los que se describen a continuación.

D.3.1. Alimentador de rodamiento

Una vez que comienza el ciclo de producción, independientemente de que sea en modo manual o automático, el primer paso a realizar es la alimentación de un rodamiento. Para ello, se accionará el cilindro de doble efecto etiquetado con la letra *"A"* hasta que el sensor de posición *"PR"* detecte la presencia del rodamiento.

Si transcurrido un tiempo determinado tras activar el cilindro alimentador, no se activa la señal del sensor *"PR"*, se deberá señalizar un error de producción y se deberá bloquear el proceso productivo. Para desbloquearlo será necesario que el operador subsane las causas que provocaron el error y que posteriormente accione el pulsador *"Rearme"*. Ésto provocará que el sistema vuelva a condiciones iniciales permitiéndose de nuevo la entrada en producción cuando el operador lo solicite.

El cilindro alimentador no se recogerá hasta que el sistema de trasvase no haya pasado el rodamiento alimentado a la plataforma de medición. Se dice que el sistema de alimentación está en condiciones iniciales cuando el cilindro alimentador está recogido.

D.3.2. Trasvase de rodamiento

Una vez que el proceso de alimentación ha finalizado, se procederá a pasar el rodamiento correspondiente a la plataforma de medida para determinar su altura.

Para ello, se accionará el cilindro rotativo *"B"* hasta la posición de alimentación. Llegado a esta posición, se accionará la pinza *"C"* y se girará el cilindro rotativo hasta que alcance la plataforma de medida. Una vez en esta posición, se liberará la pinza para que suelte el rodamiento sobre la citada plataforma.

Se dice que el sistema de trasvase está en condiciones iniciales si el cilindro rotativo se encuentra en la posición intermedia entre la plataforma de alimentación y la de medida.

D.3.3. Medición de altura de rodamiento

Una vez que el rodamiento se encuentra en la plataforma de medida, se procede a accionar el cilindro centrador *"F"* para asegurarse de que el rodamiento queda en la posición adecuada para ser medido correctamente. Acto seguido, se accionará el cilindro *"D"* para elevar la plataforma de medida hasta el tope. Una vez llegado al tope y transcurrido un tiempo prudencial para que el sensor palpador pueda efectuar la correcta medida, se accionará de nuevo el cilindro *"D"* para bajar la plataforma de medida. Cuando ésta llega a su posición más baja se desaccionará el cilindro centrador.

Se dice que el sistema de medida está en condiciones iniciales si la plataforma de medida está en su posición más baja y el centrador no está accionado.

D.3.4. Expulsión de rodamiento

Tras llevar a cabo la medida del rodamiento y si esta no coincide con la medida seleccionada por el operador, se procede a su expulsión. Para ello, se acciona el cilindro *"E"*.

El sistema de expulsión está en condiciones iniciales cuando no está activado el cilindro *"E"*.

D.3.5. Evacuación de rodamiento

Si tras llevar a cabo la medida del rodamiento, ésta coincide con la seleccionada por el operador, se procederá a insertar el rodamiento en la base que se encuentra a la espera en el sistema de transferencia. Para ello, se accionará el cilindro rotatorio

"H" para posicionar el sistema de inserción sobre la plataforma de medida. Una vez en este punto, se accionará el cilindro elevador *"G"* para bajar el sistema de inserción y permitir que la pinza *"I"* se sitúe en el interior del rodamiento, momento en el cual se accionará la citada pinza. Acto seguido, se accionará de nuevo el cilindro elevador *"G"* para elevar el rodamiento a la plataforma de medida, y se rotará el cilindro *"H"* para llevar el rodamiento hasta la posición en la que se encuentra la base. A continuación, se bajará el sistema de inserción y se liberará la pinza, quedando el rodamiento alojado en el interior de la base. Por último, se retirará la pinza del interior del rodamiento elevando el cilindro *"G"*.

Se dice que el sistema de inserción está en condiciones iniciales si la pinza está desaccionada, el cilindro elevador está en su posición más alta y el cilindro rotativo se encuentra en la vertical sobre el sistema de transferencia.

D.3.6. Transición entre distintos modos de funcionamiento

La primera vez que la parte de mando sea alimentada con energía eléctrica y habilitada, ésto es, que el operador seleccione el modo *"RUN"* en el equipo de control que finalmente implemente la parte de mando, se deberá pulsar el pulsador *"Rearme"* para que el equipo de control lleve a cabo las acciones necesarias sobre los componentes de la estación que permita la entrada en producción cuando el operador lo indique. Cuando esas acciones se hayan completado, se puede decir que el sistema se encuentra en condiciones iniciales. El sistema permanecerá entonces a la espera de que el operador seleccione el modo de producción que desee: Automático o Manual.

No será posible transitar directamente entre los dos modos de producción, manual y automático. Para permitir el cambio de modo de producción de automático a manual o viceversa, el operador deberá pulsar previamente el pulsador de *"Parada"*. Para permitir el cambio del modo de producción, el operador deberá pulsar el pulsador *"Rearme"*. Estas dos pulsaciones tienen el efecto de llevar al sistema a una situación de espera en condiciones iniciales. Será en este momento cuando el operador podrá seleccionar el modo de funcionamiento que desee mediante el selector de dos posiciones del panel de mando, y pulsar a continuación el botón *"Marcha"* para que el sistema entre en el modo de producción seleccionado.

D.3.7. Panel de mando

Para comandar los distintos modos de funcionamiento del sistema automatizado y supervisar el correcto funcionamiento del mismo, se dispone de un panel de mando con la arquitectura representada en la figura D.9, y cuyo modo de funcionamiento es el que se indica a continuación.

- *Emergencia*. Mediante la seta de emergencia se cortará la alimentación eléctrica tanto a la parte operativa como a la parte de mando. Por este motivo, este modo de funcionamiento no se va a tener en cuenta a la hora de confeccionar el modelo de la lógica de control, ya que al quedarse sin energía la parte de mando tras pulsar la seta, no tiene sentido intentar llevar a cabo ningún tipo de control en este punto.

- *Inicialización*. Para llevar a cabo la inicialización del sistema, el operador cuenta con el pulsador *"Rearme"*. Cuando se den las condiciones oportunas, ésto es que el sistema se encuentre en una situación de error, que se produzca un arranque en frío o que el operador pulse el botón *"Rearme"* para finalizar el modo de producción manual, el equipo de control llevará a cabo las acciones oportunas que conduzcan a los componentes de la parte operativa a condiciones iniciales.

- *Automático*. Una vez que la parte operativa está en condiciones iniciales, el operador podrá seleccionar, si lo desea, el modo de funcionamiento automático mediante el selector de dos posiciones del panel de mando. En este modo de funcionamiento se llevarán a cabo las acciones necesarias para garantizar que se inserta en la base un rodamiento del tipo adecuado, según el operador lo haya indicado por medio del selector de dos posiciones *"Alto/Bajo"* del panel de mando. Una vez seleccionado el modo automático, para que el proceso comience a producir será necesario que el operador oprima el pulsador *"Marcha"*.

- *Manual*. El operador podrá seleccionar el modo de funcionamiento manual cuando lo desee, sin embargo, dado que el sistema no podrá llegar a él si no está en condiciones iniciales, si el sistema se hallaba en el modo de producción automático, será necesario que el operador previamente pulse el pulsador *"Parada"* para finalizarlo y permitir así el cambio de modo.
  Cuando el sistema esté en condiciones de entrar en modo manual, el operador

deberá oprimir el pulsador *"Marcha"* para hacerlo efectivo. En este modo de funcionamiento se llevarán a cabo las mismas acciones que en el modo automático, pero el operador deberá actuar sobre el pulsador *"Marcha"* para permitir la evolución de una acción a la siguiente.

- *Parada*. El operador podrá emplear el pulsador de parada en cualquier momento que lo desee mientras el sistema esté en producción automática. En ese caso, el sistema finalizará el ciclo de producción en que se encuentre y se detendrá en condiciones iniciales a la espera de que el operador seleccione el modo de funcionamiento deseado y vuelva a pulsar *"Marcha"*.

- *Error*. Si eventualmente se produjese un error, el sistema dejará de estar en producción y se señalizará la situación anómala por medio de la lámpara de color rojo situada a tal efecto en el panel de mando. El sistema permanecerá en este estado hasta que el operador solucione (manualmente) la causa del error y oprima el pulsador *"Rearme"*. En este momento el sistema volverá a condiciones iniciales.

D.4. Especificación de requisitos

De la descripción de los componentes del sistema y del modo de funcionamiento glosado en el apartado anterior, se deduce que va a haber un único perfíl de usuario que maneje el sistema: *"el operador de producción"* que se encarga de emplear el sistema automatizado para producir;

Asi mismo, del enunciado proporcionado por el manual de la estación de inserción de rodamiento se pueden establecer un conjunto de posibles usos del sistema automatizado. Para ello, se ha empleado la guía GEMMA y se han seguido los pasos del método establecido en la especificación de requisitos de la Métodología orientada a objetos MLAV [Gon02]. Como resultado se han detectado los siguientes modos de marcha y parada del sistema:

- *A1*. Parada en estado inicial.

- *F1*. Producción normal.

- *F5*. Marchas de verificación en orden.

- *A2*. Demanda de parada a fin de ciclo
- *A6*. Inicialización de parte operativa
- *D2*. Diagnóstico y/o tratamiento de fallos.

Estos modos de marcha y parada aparecen representados en el GDMMA [Gon02] de la figura D.10, y en realidad representan los distintos modos de uso del sistema automatizado los cuales pueden ser representados de manera normalizada empleando diagramas de notación UML.

D.5. Tabla de E/S

Entradas En la tabla D.1 se muestra el mapeo de señales de entrada.

Etiqueta	**Variable**	**Descripción**
MARCHA	%I1.0	Pulsador de puesta en marcha.
PARADA	%I1.1	Pulsador de parada.
AUTO-MAN	%I1.2	Selector de modo de producción. Automática o manual.
REARME	%I1.3	Pulsador de rearme de la estación.
a0	%I1.4	Detección de inicio de carrera del alimentador.
a1	%I1.5	Detección de presencia de rodamiento.
b0	%I1.6	Detección de posición inicial del manipulador de trasvase.
b1	%I1.7	Detección de posición central del manipulador trasvase.
b2	%I1.8	Detección de posición final del manipulador trasvase.
d0	%I1.9	Detección de posición abajo del elevador.
d1	%I1.10	Detección de posición arriba del elevador.
g0	%I1.11	Detección del manipulador de inserción abajo.
g1	%I1.12	Detección del manipulador de inserción arriba.
h0	%I1.13	Detección de posición inicial de giro del manipulador de inserción.
h1	%I1.14	Detección de posición final de giro del manipulador de inserción.
	%I2.0	Selector de tipo de rodamiento.
	%IW0.8	Medida de la altura del rodamiento.

Cuadro D.1: Tabla de salidas de la estación de rodamientos

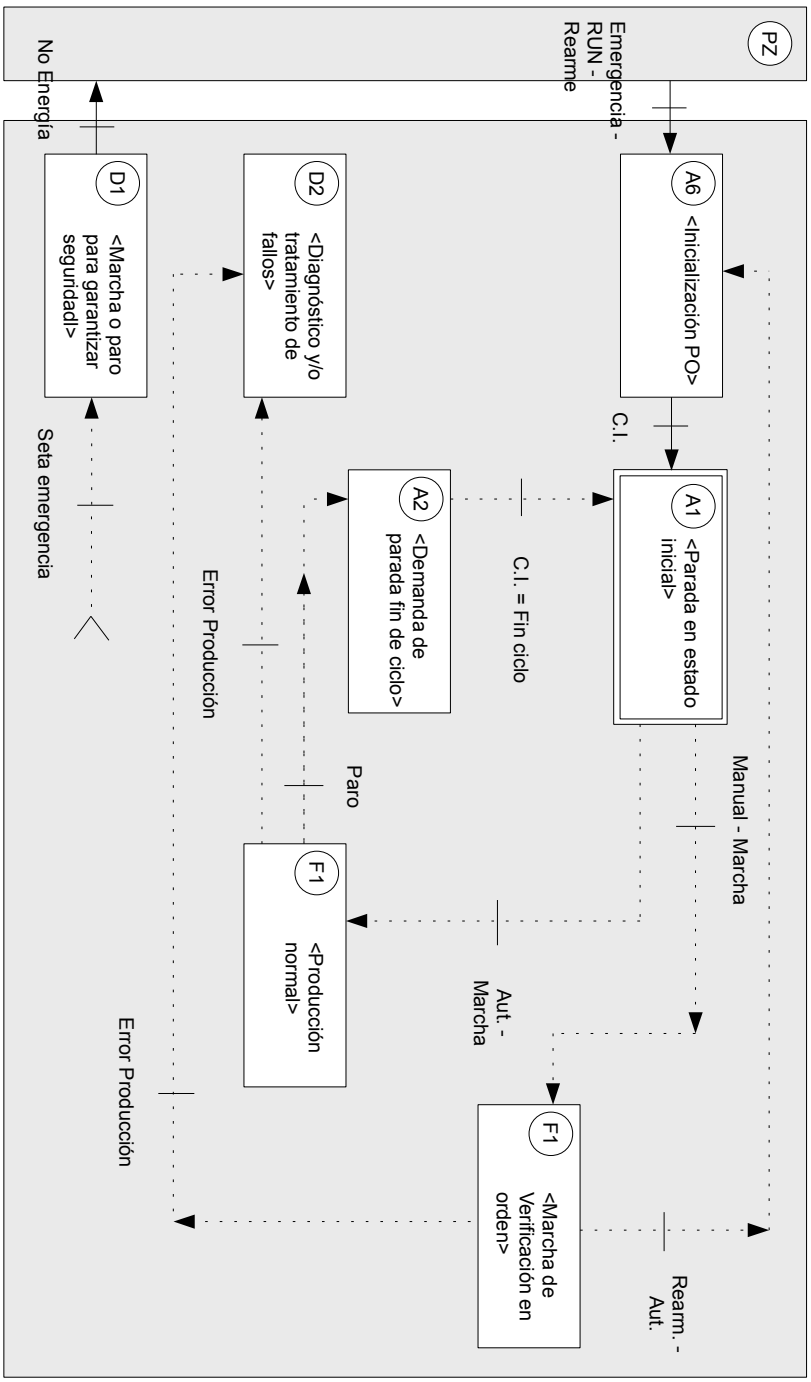

Figura D.10: DMMA para la Estación de Inserción de Rodamientos

Salidas En la tabla D.2 se muestra el mapeo de señales de salida.

Etiqueta	Variable	Descripción
DEFECTO	%Q3.0	Piloto luminoso de defecto.
A+	%Q3..1	Avance del alimentador.
B+	%Q3..2	Avance del manipulador de trasvase.
B-	%Q3..3	Retroceso del manipulador de trasvase.
C+	%Q3..4	Apertura de la pinza del manipulador de trasvase.
D+	%Q3..5	Subida del elevador.
D-	%Q3..6	Bajada del elevador.
E+	%Q3..7	Avance del expulsor de rechazo.
F+	%Q3..8	Avance del centrador.
G+	%Q3..9	Bajada del manipulador inserción.
H+	%Q3..10	Avance de giro del manipulador de inserción.
I+	%Q3..11	Apertura de la pinza del manipulador de inserción.
FM	%Q3..12	Piloto luminoso de falta material.

Cuadro D.2: Tabla de salidas de la estación de rodamientos

D.6. Diagrama de clases

La estación de insercción de rodamientos se compone de 14 clases y su diagrama de clases se muestra en la figura D.11.

D.7. Programación de la estación

En la siguiente sección se detalla la implementación de cada una de las clases que componen la estación así como los métodos de las mismas. Además, por cada definición de clase y de cada método, se adjunta la traducción a código IL[1] que realiza SimPLC++ tras ejecutar el módulo traductor (ver sección 4.4).

Como último detalle de esta sección, se muestran las llamadas a los métodos virtuales y sobrecargados que realiza el *"transfer"*, así como su traducción a código

[1]En la traducción de los métodos a código IL, se omitirá el control de las variables ON y ENO, así como algunas de las variables intermedias de estado de las transiciones-etapas para no presentar códigos IL excesivamente largos.

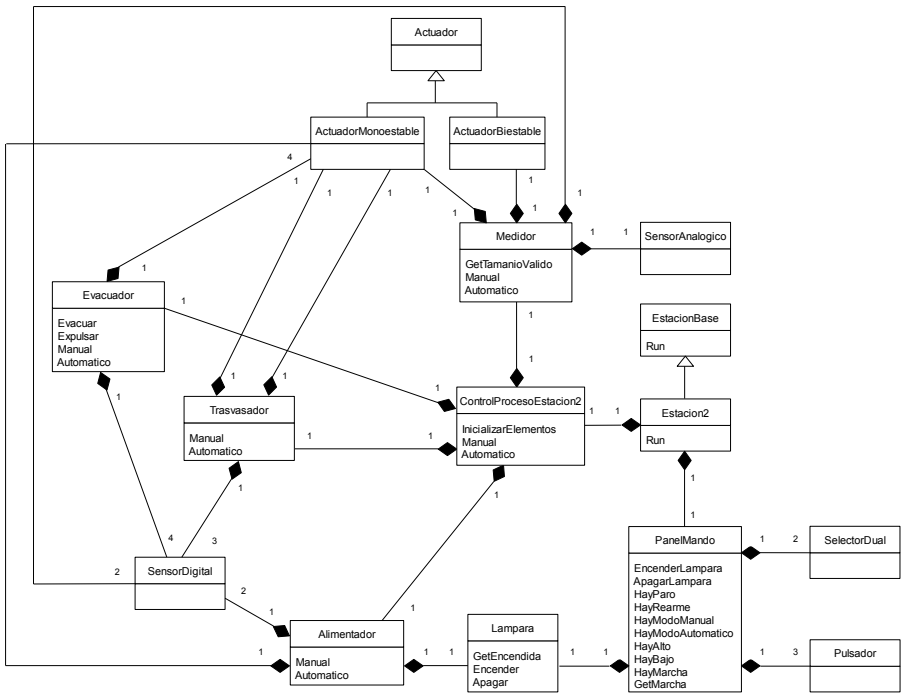

Figura D.11: Diagrama de clases de la estación de inserción de rodamientos

Algoritmo D.1 Clase *"SensorDigital"*

```
CLASS SensorDigital ( )
  PUBLIC activado : BOOL;
END_CLASS
```

```
TYPE SensorDigital:
   STRUCT
      activado : BOOL;
   END_STRUCT;
END_TYPE
```

Figura D.12: Traducción de la clase *"SensorDigital"* a IL

IL (ver apartado D.7.17).

D.7.1. Clase SensorDigital

Esta clase se corresponde con un sensor de tipo digital. Posee un único atributo de tipo boolean que almacena el estado del sensor. En el algoritmo D.1 se muestra la estructura de la clase y en la figura D.12 la estructura que implementa la clase.

D.7.2. Clase SensorAnalogico

Esta clase se corresponde con un sensor de tipo analógico. Posee un único atributo de tipo entero que almacena el valor numérico del sensor. En el algoritmo D.2 se muestra la estructura de la clase y en la figura D.13 la estructura que implementa la clase.

Algoritmo D.2 Clase *"SensorAnalogico"*

```
CLASS SensorAnalogico ( )
  PUBLIC valor : INT;
END_CLASS
```

```
TYPE SensorAnalogico:
   STRUCT
      valor : INT;
   END_STRUCT;
END_TYPE
```

Figura D.13: Traducción de la clase *"SensorAnalogico"* a IL

Algoritmo D.3 Clase *"SensorDual"*

```
CLASS SensorDual ( )
  PUBLIC valor1 : BOOL;
  PUBLIC valor2 : BOOL;
END_CLASS
```

D.7.3. Clase SensorDual

Esta clase se corresponde con un sensor digital que admite dos valores. Su función es la de poder elegir el tamaño del rodamiento. Posee dos atributos de tipo boolean que almacenan que tipo de medida está activada. En el algoritmo D.3 se muestra la estructura de la clase y en la figura D.14 la estructura que implementa la clase.

D.7.4. Clase Pulsador

Esta clase se corresponde con un pulsador que sirve para poner en marcha cada uno de los componentes de la estación. Posee un único atributo de tipo boolean que almacena si el objeto esta activado. En el algoritmo D.4 se muestra la estructura de la clase y en la figura D.15 la estructura que implementa la clase.

```
TYPE SensorDual:
   STRUCT
      valor1 : BOOL;
      valor2 : BOOL;
   END_STRUCT;
END TYPE
```

Figura D.14: Traducción de la clase *"SensorDual"* a IL

Algoritmo D.4 Clase *"Pulsador"*

```
CLASS Pulsador ( )
  PUBLIC accionado : BOOL;
END_CLASS
```

```
TYPE Pulsador:
   STRUCT
      accionado : BOOL;
   END_STRUCT;
END_TYPE
```

Figura D.15: Traducción de la clase *"Pulsador"* a IL

D.7.5. Clase ActuadorMonoestable

Esta clase se corresponde con un actuador monoestable neumático. Posee un único atributo de tipo boolean que indica si el actuador esta extendido o no. Por defecto el valor es falso, es decir, sin extensión. En el algoritmo D.5 se muestra la estructura de la clase y en la figura D.16 la estructura que implementa la clase.

D.7.6. Clase ActuadorBiestable

Esta clase se corresponde con un actuador de tipo biestable neumático. Posee dos atributos de tipo boolean que sirven para extender el actuador en cada sentido. En el algoritmo D.6 se muestra la estructura de la clase y en la figura D.17 la estructura que implementa la clase.

Algoritmo D.5 Clase *"ActuadorMonoestable"*

```
CLASS ActuadorMonoestable ( )
  PUBLIC accionado : BOOL := false;
END_CLASS
```

```
TYPE ActuadorMonoestable:
    STRUCT
        activado : BOOL;
    END_STRUCT;
END_TYPE
```

Figura D.16: Traducción de la clase *"ActuadorMonoestable"* a IL

Algoritmo D.6 Clase *"ActuadorBiestable"*

```
CLASS ActuadorBiestable ( )
  PUBLIC accionado1 : BOOL;
  PUBLIC accionado2 : BOOL;
END_CLASS
```

```
TYPE ActuadorBiestable:
    STRUCT
        activado1 : BOOL;
        activado1 : BOOL;
    END_STRUCT;
END_TYPE
```

Figura D.17: Traducción de la clase *"ActuadorBiestable"* a IL

Algoritmo D.7 Clase *"Lampara"*

```
CLASS Lampara ( )
  PRIVATE encendida : BOOL := false;
  PUBLIC METHOD GetEncendida ( ) : BOOL;
  PUBLIC METHOD Encender ( ) : VOID;
  PUBLIC METHOD Apagar ( ) : VOID;
END_CLASS
```

```
TYPE Lampara:
   STRUCT
      encendida : BOOL;
   END_STRUCT;
END_TYPE
```

Figura D.18: Traducción de la clase *"Lampara"* a IL

D.7.7. Clase Lampara

Esta clase se corresponde con un elmento de tipo lampara que sirve como piloto indicativo. Posee un único atributo de tipo boolean que indica si la lampara está encendida y contiene 3 métodos. Uno devuelve el estado de la lampara y los otros dos sierven para cambiar el estado del atributo de la clase (encender y apagar la lampara). En el algoritmo D.7 se muestra la estructura de la clase y en la figura D.18 la estructura que implementa la clase.

Método GetEncendida

Este método devuelve el valor del atributo de la clase (ver algoritmo D.8). En la figura D.19 se muestra la traducción a código IL del método.

Algoritmo D.8 Método *"GetEncendida"* de la clase *"Lampara"*

```
METHOD Lampara::GetEncendida ( )
  RETURN encendida;
END_METHOD
```

```
FUNCTION_BLOCK Lampara_GetEncendida
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : Lampara;
END_VAR

LD _This.encendida
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.19: Traducción del método *"GetEncendida"* de la clase *"Lampara"* a IL

Algoritmo D.9 Método *"Encender"* de la clase *"Lampara"*

```
METHOD Lampara::Encender ( )
  encendida:=true;
END_METHOD
```

Método Encender

Este método pone a true el valor del atributo de la clase (ver algoritmo D.9). En la figura D.20 se muestra la traducción a código IL del método.

Método Apagar

Este método pone a false el valor del atributo de la clase (ver algoritmo D.10). En la figura D.21 se muestra la traducción a código IL del método.

```
FUNCTION_BLOCK Lampara_Encender
VAR_IN_OUT
        _This : Lampara;
END_VAR

LD TRUE
ST _This.encendida
END_FUNCTION_BLOCK
```

Figura D.20: Traducción del método *"Encender"* de la clase *"Lampara"* a IL

Algoritmo D.10 Método *"Apagar"* de la clase *"Lampara"*

```
METHOD Lampara::Apagar ( )
   encendida:=false;
END_METHOD
```

```
FUNCTION_BLOCK Lampara_Apagar
VAR_IN_OUT
        _This : Lampara;
END_VAR

LD FALSE
ST _This.encendida
END_FUNCTION_BLOCK
```

Figura D.21: Traducción del método *"Apagar"* de la clase *"Lampara"* a IL

D.7.8. Clase ObjetoComplejoBase

Esta clase se corresponde con un elmento básico del que heredan todas las demás clases complejas. Posee tres atributos de tipo boolean que indican, respectivamente, si el proceso ha finalizado, si el objeto está en condiciones iniciales y si se ha producido un fallo. Posee además 6 métodos, a saber:

- *InicializarElementos* pone todas los atributos de la clase en su estado inicial.
- *GetFallo* devuelve el estado del atributo *"fallo"*.
- *ResetFallo* pone a false el atributo *"fallo"*.
- *GetCondicionesIniciales* decuelve el estado del atributo *"condicionesIniciales"*.
- *ResetCondicionesIniciales* pone a false el atributo *"condicionesIniciales"*.
- *GetFinProceso* decuelve el estado del atributo *"finProceso"*.
- *ResetFinProceso* pone a false el atributo *"finProceso"*.

En el algoritmo D.11 se muestra la estructura de la clase y en la figura D.22 la estructura que implementa la clase.

Algoritmo D.11 Clase *"ElementoBaseComplejo"*

```
CLASS ObjetoComplejoBase ( )
  PROTECTED finProceso : BOOL;
  PROTECTED condicionesIniciales : BOOL;
  PROTECTED fallo : BOOL;
  PUBLIC METHOD GetFallo ( ) : BOOL;
  PUBLIC METHOD ResetFallo ( ) : BOOL;
  VIRTUAL METHOD GetCondicionesIniciales ( ) : BOOL;
  VIRTUAL METHOD ResetCondicionesIniciales ( ) : BOOL;
  VIRTUAL METHOD GetFinProceso ( ) : BOOL;
  VIRTUAL METHOD ResetFinProceso ( ) : BOOL;
  VIRTUAL METHOD InicializarElementos ( ) : void;
  VIRTUAL METHOD Manual ( ) : void;
  VIRTUAL METHOD Automatico ( ) : void;
END_CLASS
```

```
TYPE ObjetoComplejoBase:
   STRUCT
      vPointer : *VOID;
      finProceso : BOOL;
      condicionesIniciales : BOOL;
      fallo : BOOL;
   END_STRUCT;
END_TYPE
```

Figura D.22: Traducción de la clase "*ElementoBaseComplejo*" a IL

Algoritmo D.12 Métodos de la clase *"ElementoBaseComplejo"*

```
METHOD ObjetoComplejoBase::ObjetoComplejoBase ( )
  THIS.vPointer:=&vPointer+20;
END_METHOD

METHOD ObjetoComplejoBase::GetFallo ( )
  RETURN encendida;
END_METHOD

METHOD ObjetoComplejoBase::ResetFallo ( )
   fallo:=false;
END_METHOD

METHOD ObjetoComplejoBase::GetCondicionesIniciales ( )
  RETURN condicionesIniciales;
END_METHOD

METHOD ObjetoComplejoBase::ResetCondicionesIniciales ( )
   condicionesIniciales:=false;
END_METHOD

METHOD ObjetoComplejoBase::GetFinProceso ( )
  RETURN finProceso;
END_METHOD

METHOD ObjetoComplejoBase::ResetFinProceso ( )
   finProceso:=false;
END_METHOD
```

Métodos de la clase ObjetoComplejoBase

En la figura D.23 se muestra la traducción de todos los métodos de la clase *"ElementoBaseComplejo"* (ver figura D.23). Los métodos get devuelven el valor del atributo al que hacen referencia y los métodos reset ponen a false dichos atributos.

D.7.9. Clase Alimentador

La función de esta clase es la de alimentar la estación de rodamientos. Se trata de una clase elaborada que hereda de *"ObjetoComplejoBase"* y está formada por un actuador genérico monoestable, un sensor de posición magnético, un sensor de

```
FUNCTION_BLOCK ObjetoComplejoBase_ObjetoComplejoBase
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR
VAR
        aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 20
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK


FUNCTION_BLOCK ObjetoComplejoBase_GetFallo
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR

LD _This.fallo
ST _Retorno
END_FUNCTION_BLOCK


FUNCTION_BLOCK ObjetoComplejoBase_ResetFallo
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR

LD FALSE
ST _This.fallo
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK ObjetoComplejoBase_ResetCondicionesIniciales
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR

LD FALSE
ST _This.condicionesIniciales
END_FUNCTION_BLOCK


FUNCTION_BLOCK ObjetoComplejoBase_GetFinProceso
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR

LD _This.finProceso
ST _Retorno
END_FUNCTION_BLOCK


FUNCTION_BLOCK ObjetoComplejoBase_ResetFinProceso
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR

LD FALSE
ST _This.finProceso
END_FUNCTION_BLOCK


FUNCTION_BLOCK ObjetoComplejoBase_GetCondicionesIniciales
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : ObjetoComplejoBase;
END_VAR

LD _This.condicionesIniciales
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.23: Traducción de la los métodos de la clase *"ElementoBaseComplejo"* a IL

Algoritmo D.13 Clase *"Alimentador"*

```
CLASS Alimentador (ObjetoComplejoBase)
  PRIVATE petacaVacia : BOOL;
  PRIVATE Lampara : Lampara;
  PRIVATE cilMono_alimentador : ActuadorMonoestable;
  PRIVATE sDigital_material : SensorDigital;
  PRIVATE sDigital_alimentador : SensorDigital;
  PRIVATE T0 : TON;
  PRIVATE T1 : TON;
  PUBLIC METHOD InicializarElementos ( ) : void;
  PUBLIC METHOD Manual ( ) : void;
  PUBLIC METHOD Automatico ( ) : void;
END_CLASS
```

Algoritmo D.14 Constructor de la clase *"Alimentador"*

```
METHOD Alimentador::Alimentador ( )
  THIS.vPointer:=&vPointer+28;
END_METHOD
```

posición digital electromecánico y una lampara que indica al operario la falta de material o si se ha producido un error. Posee un atributo booleano que indica si se ha vaciado el alimentador de rodamientos y dos contadores de tiempo. En el algoritmo D.13 se muestra la estructura de la clase y en la figura D.24 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.14). En la figura D.25 se muestra la traducción a código IL del método constructor.

Método InicializarElementos

Este método coloca todos los elementos del objeto en su posición inicial (ver algoritmo D.26). En las figuras D.27 y D.28 se muestra la traducción a código IL del método.

```
TYPE Alimentador:
   STRUCT
      vPointer : *VOID;
      petacaVacia : BOOL;
      Lampara : Lampara;
      cilMono_alimentador : ActuadorMonoestable;
      sDigital_material : SensorDigital;
      sDigital_alimentador : SensorDigital;
      T0 : TON;
      T1 : TON;
      _ObjetoComplejoBase : ObjetoComplejoBase;
   END_STRUCT;
END_TYPE
```

Figura D.24: Traducción de la clase *"Alimentador"* a IL

```
FUNCTION_BLOCK Alimentador_Alimentador
VAR_IN_OUT
        _This : Alimentador;
END_VAR
VAR
      aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 23
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.25: Traducción del a IL del constructor de la clase *"Alimentador"*

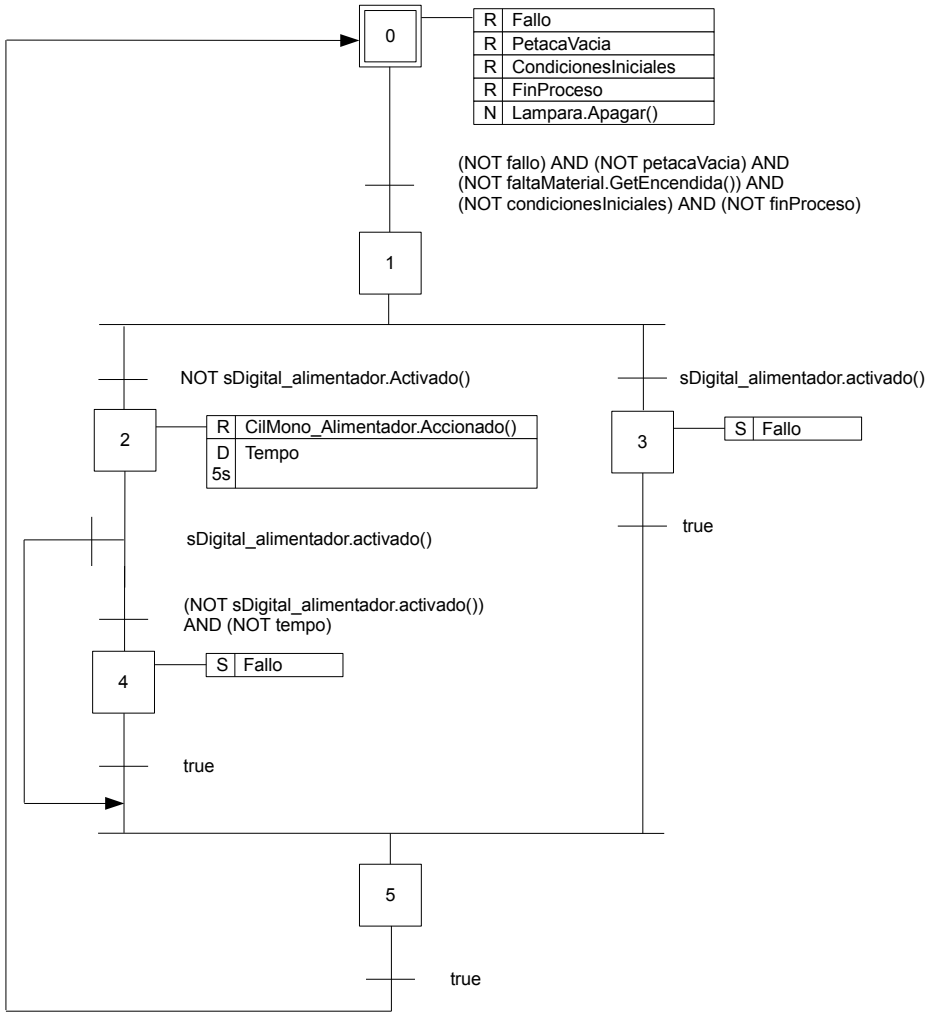

Figura D.26: Método *"InicializarElementos"* de la clase *"Alimentador"*

```
FUNCTION_BLOCK Alimentador_InicializarElementos
VAR_IN_OUT
        _THIS : Alimentador;
END_VAR

VAR (* Variables locales *)
        tempo : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUX_7 : BOOL;
        _AUX_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E3 : TSFCTYPE;
        E6 : TSFCTYPE;
        apagarLampara : BOOL := FALSE;
        T0 : TON;
        Lampara_GetEncendida : Lampara_GetEncendida;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN _THIS.ObjetoComplejoBase.fallo
ST _AUX_2
LDN _THIS.petacaVacia
AND _AUX_2
ST _AUX_4
(* Llamada a metodo*)
CAL Lampara_GetEncendida(_THIS := _THIS.faltaMaterial)
LDN GetEncendida_Lampara_0._Retorno
AND _AUX_4
ST _AUX_6
LDN _THIS.condicionesIniciales
AND _AUX_6
ST _AUX_8
LDN _THIS.ObjetoComplejoBase.finProceso
AND _AUX_8
ST _AUXT_1
LDN _THIS.sDigital_alimentador.activado
ST _AUXT_11
LDN _THIS.sDigital_alimentador.activado
ST _AUXT_13
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E3.X
ST E3.C
LD E6.X
ST E6.C
LD TRUE
AND E6.C
S E0.X
R E6.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_11
ANDN _THIS.sDigital_alimentador.activado
AND E1.C
S E2.X
R E1.X
LD _AUXT_13
ANDN _THIS.sDigital_alimentador.activado
AND E2.C
S E4.X
R E2.X
LD E4.C
```

Figura D.27: Traducción del a IL del método *"InicializarElementos"* de la clase *"Alimentador"* - parte 1

```
AND TRUE
OR ( E2.C
AND _THIS.sDigital_alimentador.activado
)
S E5.X
R E4.X
LD _THIS.sDigital_alimentador.activado
ANDN _AUXT_11
AND E1.C
S E3.X
R E1.X
LD E5.C
AND TRUE
OR ( E3.C
AND TRUE
)
S E6.X
R E5.X
R E3.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.petacaVacia
LDN apagarLampara
JMPC _SALTO_0
_SALTO_0:
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E2.X
R _THIS.cilMono_alimentador.accionado
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E2.X , PT := T#5s)
LD T0.Q
ST tempo
LD E4.X
S _THIS.ObjetoComplejoBase.fallo
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
LD E0.X
ST apagarLampara
END_FUNCTION_BLOCK
```

Figura D.28: Traducción del a IL del método *"InicializarElementos"* de la clase *"Alimentador"* - parte 2

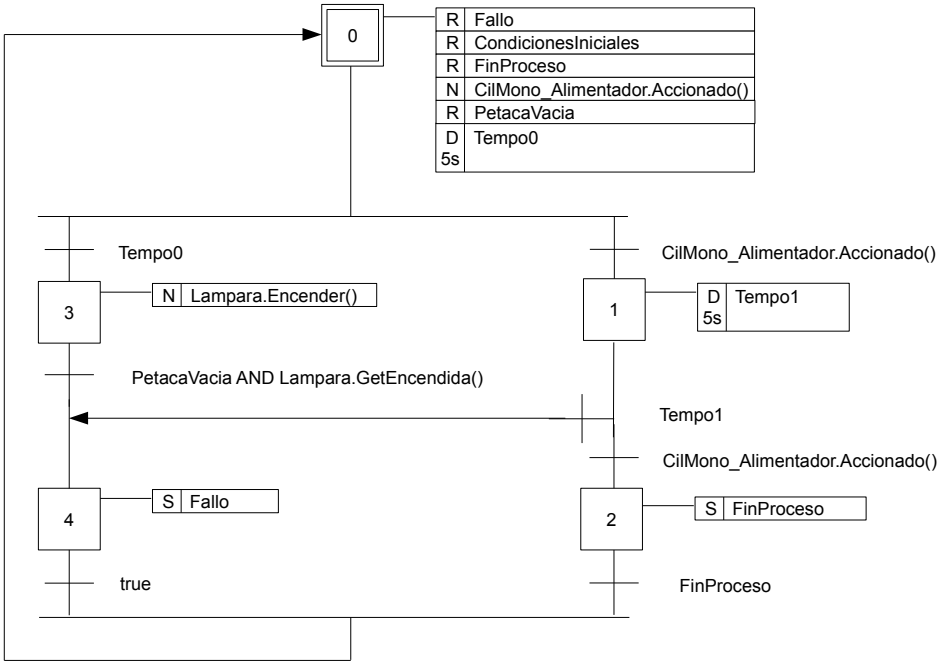

Figura D.29: Método *"Manual"* de la clase *"Alimentador"*

Método Manual (ejecución manual)

Este método ejecuta la alimentación de rodamientos de forma manual (ver figura D.29). En las figuras D.30 y D.31 se muestra la traducción del método a código IL.

Método Automatico (ejecución automática)

Este método ejecuta la alimentación de rodamientos de forma automática (ver figura D.32). En las figuras D.33 y D.34 se muestra la traducción a código IL del método.

D.7.10. Clase Trasvasador

La función de esta clase es la de transportar los rodamientos de la zona de alimentación a la zona de medición. Se trata de una clase elaborada que hereda de *"ObjetoComplejoBase"* y está formada por un actuador genérico biestable, un actuador genérico monoestable y cinco sensores de posición de tipo magnético para

```
FUNCTION_BLOCK Alimentador_Manual
VAR_IN_OUT
        _THIS : Alimentador;
END_VAR
VAR_INPUT
        marcha : Pulsador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUXT_5 : BOOL;
        _AUX_6 : BOOL;
        _AUX_7 : BOOL;
        _AUXT_8 : BOOL;
        _AUX_9 : BOOL;
        _AUXT_10 : BOOL;
        _AUX_11 : BOOL;
        _AUX_12 : BOOL;
        _AUX_13 : BOOL;
        _AUX_14 : BOOL;
        _AUXT_15 : BOOL;
        _AUX_16 : BOOL;
        _AUXT_17 : BOOL;
        _AUX_18 : BOOL;
        _AUX_19 : BOOL;
        _AUX_20 : BOOL;
        _AUX_21 : BOOL;
        E0 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E5 : TSFCTYPE;
        apagarLampara : BOOL := FALSE;
        T0 : TON;
        encenderLampara : BOOL := FALSE;
        T1 : TON;
        Lampara_GetEncendida : Lampara_GetEncendida;
END_VAR
```

```
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUX_2
LDN _THIS.sDigital_material.activado
AND _AUX_2
ST _AUXT_1
(* Llamada a metodo*)
CAL Lampara_GetEncendida(_THIS := _THIS.faltaMaterial)
LD _THIS.petacaVacia
AND GetEncendida_Lampara_0._Retorno
AND marcha.accionado
ST _AUXT_5
LD _THIS.sDigital_material.activado
AND marcha.accionado
ST _AUXT_8
LDN _THIS.sDigital_alimentador.activado
ST _AUX_11
LDN tempo1
AND _AUX_11
AND marcha.accionado
ST _AUXT_10
LD _THIS.sDigital_alimentador.activado
AND marcha.accionado
ST _AUXT_15
LDN _THIS.sDigital_alimentador.activado
ST _AUX_18
LDN tempo1
AND _AUX_18
AND marcha.accionado
ST _AUXT_17
LD E0.X
ST E0.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E4.C
AND TRUE
```

Figura D.30: Traducción del a IL del método *"Manual"* de la clase *"Alimentador"* - parte 1

```
S E0.X
R E4.X
LD _AUXT_1
ANDN _AUXT_8
AND E0.C
S E3.X
R E0.X
LD _AUXT_5
AND E3.C
OR ( E1.C
AND _AUXT_17
)
S E4.X
R E3.X
R E1.X
LD _AUXT_8
ANDN _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_15
ANDN _AUXT_17
AND E1.C
S E2.X
R E1.X
LD E2.C
AND _THIS.ObjetoComplejoBase.finProceso
R E2.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.petacaVacia
```

```
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LDN apagarLampara
JMPC _SALTO_0
_SALTO_0:
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E3.X
S _THIS.petacaVacia
LDN encenderLampara
JMPC _SALTO_1
_SALTO_1:
LD E4.X
S _THIS.ObjetoComplejoBase.fallo
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E2.X
S _THIS.ObjetoComplejoBase.finProceso
LD E0.X
ST apagarLampara
LD E0.X
ST _THIS.cilMono_alimentador.accionado
LD E3.X
ST encenderLampara
END_FUNCTION_BLOCK
```

Figura D.31: Traducción del a IL del método *"Manual"* de la clase *"Alimentador"* - parte 2

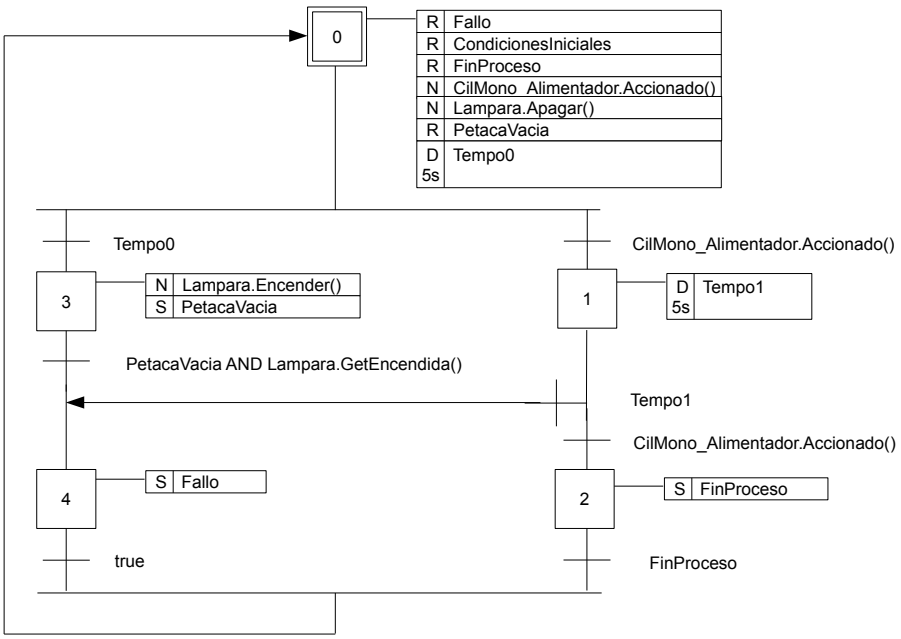

Figura D.32: Método *"Automatico"* de la clase *"Alimentador"*

detectar las distintas situaciones de los cilindros. Además, posee cuatro atributos de tipo contador de tiempo. Como servicios proporciona los métodos de inicialización de la clase así cono los de ejecución manual y automática. En el algoritmo D.15 se muestra la estructura de la clase y en la figura D.35 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.16). En la figura D.36 se muestra la traducción a código IL del método constructor.

Método InicializarElementos

Este método coloca todos los elementos del objeto en su posición inicial (ver figura D.37). En las figuras D.38, D.39, D.40 y D.41 se muestra la traducción a código IL del método.

```
FUNCTION_BLOCK Alimentador_Automatico
VAR_IN_OUT
        _THIS : Alimentador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUXT_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUX_13 : BOOL;
        _AUX_14 : BOOL;
        E0 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        apagarLampara : BOOL := FALSE;
        T0 : TON;
        encenderLampara : BOOL := FALSE;
        T1 : TON;
        Lampara_GetEncendida : Lampara_GetEncendida;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUX_2
LDN _THIS.sDigital_material.activado
AND _AUX_2
ST _AUXT_1
(* Llamada a metodo*)
CAL Lampara_GetEncendida(_THIS := _THIS.faltaMaterial)
LD _THIS.petacaVacia
AND GetEncendida_Lampara_0._Retorno
ST _AUXT_5
LDN tempo1
ST _AUX_8
LDN _THIS.sDigital_alimentador.activado
AND _AUX_8
ST _AUXT_7
LDN tempo1
ST _AUX_12
LDN _THIS.sDigital_alimentador.activado
AND _AUX_12
ST _AUXT_11
LD E0.X
ST E0.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E4.C
AND TRUE
OR ( E5.C
AND TRUE
)
S E0.X
R E4.X
R E5.X
LD _AUXT_1
ANDN _THIS.sDigital_material.activado
AND E0.C
S E3.X
R E0.X
LD _AUXT_5
AND E3.C
OR ( E1.C
AND _AUXT_11
)
S E4.X
R E3.X
```

Figura D.33: Traducción del a IL del método *"Automatico"* de la clase *"Alimentador"* - parte 1

```
R E1.X
LD _THIS.sDigital_material.activado
ANDN _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _THIS.sDigital_alimentador.activado
ANDN _AUXT_11
AND E1.C
S E2.X
R E1.X
LD E2.C
AND _THIS.finAlimentacion
R E2.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.petacaVacia
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LDN apagarLampara
JMPC _SALTO_0
_SALTO_0:
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E3.X
S _THIS.petacaVacia
LDN encenderLampara
JMPC _SALTO_1
_SALTO_1:
LD E4.X
S _THIS.ObjetoComplejoBase.fallo
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E2.X
S _THIS.ObjetoComplejoBase.finProceso
LD E0.X
ST apagarLampara
LD E0.X
ST _THIS.cilMono_alimentador.accionado
LD E3.X
ST encenderLampara
END_FUNCTION_BLOCK
```

Figura D.34: Traducción del a IL del método *"Automatico"* de la clase *"Alimentador"* - parte 2

Algoritmo D.15 Clase *"Trasvasador"*

```
CLASS Transvasador (ObjetoComplejoBase)
  PRIVATE cilMono_pinza : ActuadorMonoestable;
  PRIVATE sDigital_alimentador : SensorDigital;
  PRIVATE sDigital_centro : SensorDigital;
  PRIVATE sDigital_medidor : SensorDigital;
  PRIVATE cilBi_brazo : ActuadorBiestable;
  PRIVATE T0 : TON;
  PRIVATE T1 : TON;
  PRIVATE T2 : TON;
  PRIVATE T3 : TON;
  PRIVATE T4 : TON;
  PUBLIC METHOD InicializarElementos ( ) : void;
  PUBLIC METHOD Manual ( ) : void;
  PUBLIC METHOD Automatico ( ) : void;
END_CLASS
```

```
TYPE Trasvasador:
   STRUCT
      vPointer : *VOID;
      cilMono_pinza : ActuadorMonoestable;
      sDigital_alimentador : SensorDigital;
      sDigital_centro : SensorDigital;
      sDigital_medidor : SensorDigital;
      cilBi_brazo : ActuadorBiestable;
      T0 : TON;
      T1 : TON;
      T2 : TON;
      T3 : TON;
      T4 : TON;
      _ObjetoComplejoBase : ObjetoComplejoBase;
   END_STRUCT;
END_TYPE
```

Figura D.35: Traducción de la clase *"Trasvasador"* a IL

Algoritmo D.16 Constructor de la clase *"Trasvasador"*

```
METHOD Trasvasador::Trasvasador ( )
  THIS.vPointer:=&vPointer+31;
END_METHOD
```

```
FUNCTION_BLOCK Trasvasador_Trasvasador
VAR_IN_OUT
        _This : Trasvasador;
END_VAR
VAR
       aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 31
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.36: Traducción del a IL del constructor de la clase *"Trasvasador"*

Método Manual (ejecución manual)

Este método ejecuta el trasvase de rodamientos de forma manual (ver figura D.42). En las figuras D.43, D.44 y D.45 se muestra la traducción del método a código IL.

Método Automatico (ejecución automática)

Este método ejecuta el trasvase de rodamientos de forma automática (ver figura D.46). En las figuras D.47, D.48 y D.49 se muestra la traducción a código IL del método.

D.7.11. Clase Medidor

La función de esta clase es la de medir la altura de los rodamientos que sucesivamente se sitúan en la plataforma de medición. Se trata de una clase elaborada que hereda de *"ObjetoComplejoBase"* y está formada por un actuador genérico biestable, un actuador genérico monoestable, dos sensores magnéticos para detección de la posición del cilindro y un sensor analógico de tipo palpador para llevar a cabo

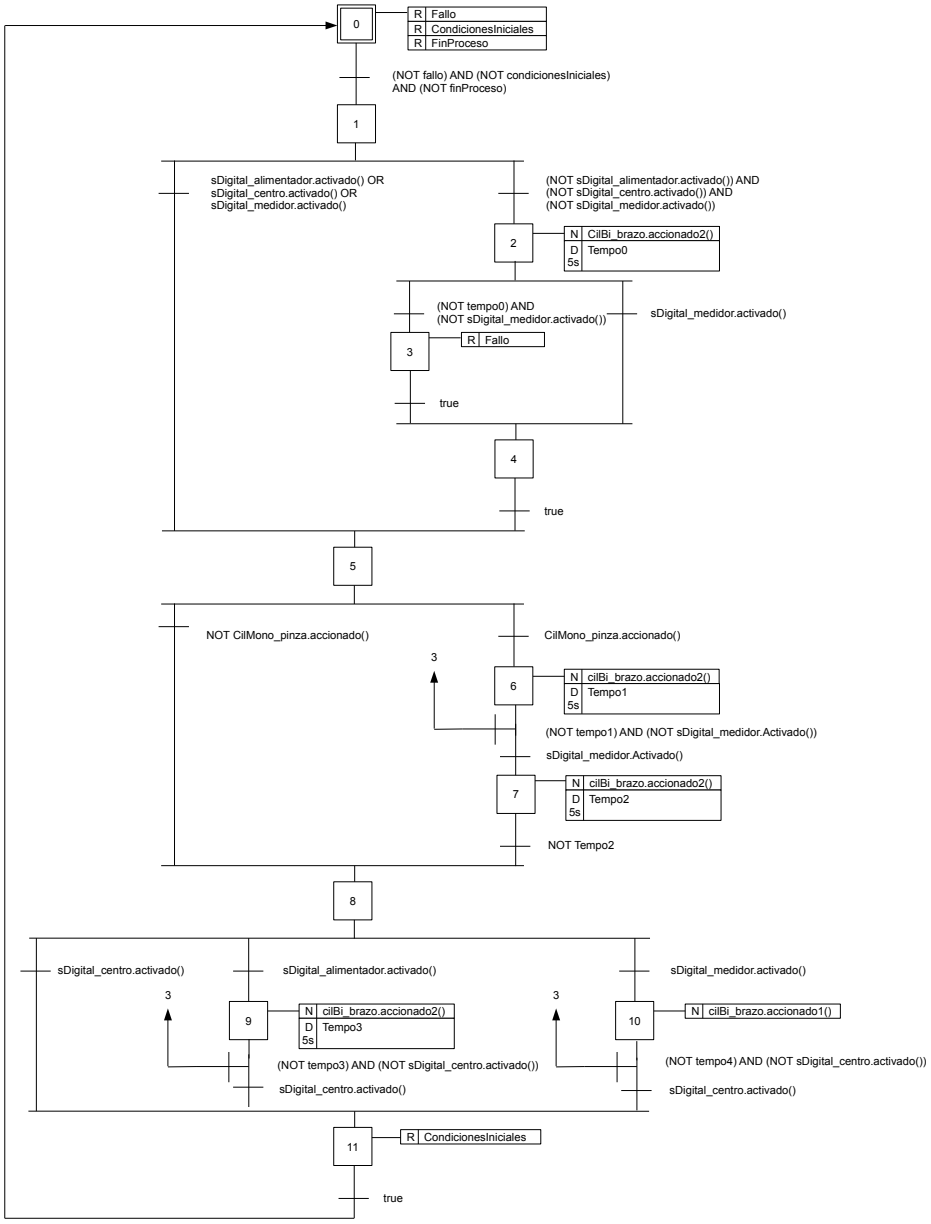

Figura D.37: Método *"InicializarElementos"* de la clase *"Trasvasador"*

```
FUNCTION_BLOCK Trasvasador_InicializarElementos
VAR_IN_OUT
        _THIS : Transvasador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUX_9 : BOOL;
        _AUXT_10 : BOOL;
        _AUX_11 : BOOL;
        _AUX_12 : BOOL;
        _AUX_13 : BOOL;
        _AUX_14 : BOOL;
        _AUX_15 : BOOL;
        _AUXT_16 : BOOL;
        _AUX_17 : BOOL;
        _AUX_18 : BOOL;
        _AUX_19 : BOOL;
        _AUXT_20 : BOOL;
        _AUX_21 : BOOL;
        _AUXT_22 : BOOL;
        _AUX_23 : BOOL;
        _AUX_24 : BOOL;
        _AUX_25 : BOOL;
        _AUXT_26 : BOOL;
        _AUX_27 : BOOL;
        _AUXT_28 : BOOL;
        _AUX_29 : BOOL;
        _AUX_30 : BOOL;
        _AUX_31 : BOOL;
        _AUXT_32 : BOOL;
        _AUX_33 : BOOL;
        _AUX_34 : BOOL;
        _AUX_35 : BOOL;
        _AUXT_36 : BOOL;
        _AUX_37 : BOOL;
        _AUX_38 : BOOL;
        _AUX_39 : BOOL;
        _AUXT_40 : BOOL;
        _AUX_41 : BOOL;
        _AUX_42 : BOOL;
        _AUX_43 : BOOL;
        _AUXT_44 : BOOL;
        _AUX_45 : BOOL;
        _AUX_46 : BOOL;
        _AUX_47 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        E9 : TSFCTYPE;
        E10 : TSFCTYPE;
        E12 : TSFCTYPE;
        E11 : TSFCTYPE;
        E13 : TSFCTYPE;
        E14 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        T3 : TON;
        T4 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN _THIS.ObjetoComplejoBase.fallo
ST _AUX_2
LDN _THIS.ObjetoComplejoBase.condicionesIniciales
AND _AUX_2
ST _AUX_4
```

Figura D.38: Traducción del a IL del método *"InicializarElementos"* de la clase *"Trasvasador"* - parte 1

```
LDN _THIS.ObjetoComplejoBase.finProceso
AND _AUX_4
ST _AUXT_1
LD _THIS.sDigital_alimentador.activado
OR _THIS.sDigital_centro.activado
OR _THIS.sDigital_medidor.activado
ST _AUXT_7
LDN _THIS.sDigital_alimentador.activado
ST _AUX_11
LDN _THIS.sDigital_centro.activado
AND _AUX_11
ST _AUX_13
LDN _THIS.sDigital_medidor.activado
AND _AUX_13
ST _AUXT_10
LDN tempo0
ST _AUX_17
LDN _THIS.sDigital_medidor.activado
AND _AUX_17
ST _AUXT_16
LDN _THIS.cilMono_pinza.accionado
ST _AUXT_20
LDN tempo1
ST _AUX_23
LDN _THIS.sDigital_medidor.activado
AND _AUX_23
ST _AUXT_22
LDN tempo2
ST _AUXT_26
LDN tempo1
ST _AUX_29
LDN _THIS.sDigital_medidor.activado
AND _AUX_29
ST _AUXT_28
LDN tempo3
ST _AUX_33
LDN _THIS.sDigital_centro.activado
AND _AUX_33
ST _AUXT_32
LDN tempo3
ST _AUX_37
LDN _THIS.sDigital_centro.activado
AND _AUX_37
ST _AUXT_36
LDN tempo4
ST _AUX_41
LDN _THIS.sDigital_centro.activado
AND _AUX_41
ST _AUXT_40
LDN tempo4
ST _AUX_45
LDN _THIS.sDigital_centro.activado
AND _AUX_45
ST _AUXT_44
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E9.X
ST E9.C
LD E10.X
ST E10.C
LD E12.X
ST E12.C
LD E11.X
ST E11.C
LD E13.X
ST E13.C
LD E14.X
ST E14.C
LD TRUE
AND E14.C
S E0.X
R E14.X
LD _AUXT_1
AND E0.C
```

Figura D.39: Traducción del a IL del método *"InicializarElementos"* de la clase *"Trasvasador"* - parte 2

```
S E1.X
R E0.X
LD _AUXT_10
ANDN _AUXT_7
AND E1.C
S E2.X
R E1.X
LD _AUXT_16
ANDN _THIS.sDigital_medidor.activado
AND E2.C
OR ( E6.C
AND _AUXT_28
)
OR ( E10.C
AND _AUXT_36
)
OR ( E11.C
AND _AUXT_44
)
S E3.X
R E2.X
R E6.X
R E10.X
R E11.X
LD E3.C
AND TRUE
OR ( E2.C
AND _THIS.sDigital_medidor.activado
)
S E4.X
R E3.X
LD E1.C
AND _AUXT_7
OR ( E4.C
AND TRUE
)
S E5.X
R E4.X
LD _THIS.cilMono_pinza.accionado
ANDN _AUXT_20
AND E5.C
S E6.X
R E5.X
LD _THIS.sDigital_medidor.activado
ANDN _AUXT_28
AND E6.C
S E7.X
R E6.X
LD E7.C
AND _AUXT_26
S E8.X
R E7.X
LD E5.C
AND _AUXT_20
OR ( E8.C
AND TRUE
)
S E9.X
R E8.X
LD _THIS.sDigital_alimentador.activado
ANDN _THIS.sDigital_centro.activado
ANDN _THIS.sDigital_medidor.activado
AND E9.C
S E10.X
R E9.X
LD E10.C
AND _THIS.sDigital_centro.activado
S E12.X
LD _THIS.sDigital_medidor.activado
ANDN _THIS.sDigital_centro.activado
ANDN _THIS.sDigital_alimentador.activado
AND E9.C
S E11.X
R E9.X
LD E11.C
AND _THIS.sDigital_centro.activado
S E13.X
LD E9.C
AND _THIS.sDigital_centro.activado
OR ( E12.C
AND TRUE
)
OR ( E13.C
AND TRUE
)
S E14.X
R E12.X
R E13.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
```

Figura D.40: Traducción del a IL del método *"InicializarElementos"* de la clase *"Trasvasador"* - parte 3

```
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E2.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E6.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E7.X
R _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E7.X , PT := T#5s)
LD T2.Q
ST tempo2
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E10.X , PT := T#5s)
LD T3.Q
ST tempo3
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E11.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E14.X
S _THIS.ObjetoComplejoBase.condicionesIniciales
LD E11.X
ST _THIS.cilBi_brazo.accionado1
LD E2.X
OR E6.X
OR E10.X
ST _THIS.cilBi_brazo.accionado2
END_FUNCTION_BLOCK
```

Figura D.41: Traducción del a IL del método *"InicializarElementos"* de la clase *"Trasvasador"* - parte 4

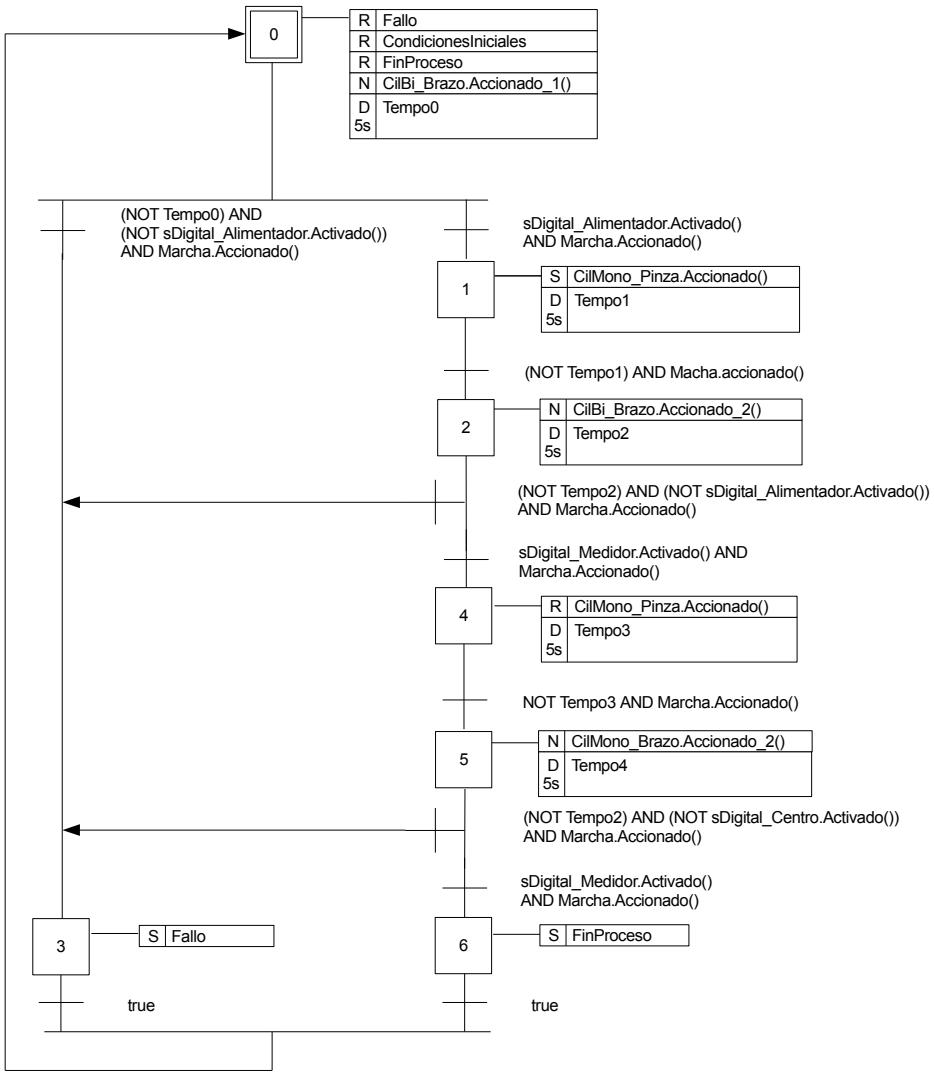

Figura D.42: Método *"Manual"* de la clase *"Trasvasador"*

```
FUNCTION_BLOCK Trasvasador_Manual
VAR_IN_OUT
        _THIS : Transvasador;
        marcha : Pulsador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUXT_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUXT_6 : BOOL;
        _AUX_7 : BOOL;
        _AUX_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        _AUX_15 : BOOL;
        _AUXT_16 : BOOL;
        _AUX_17 : BOOL;
        _AUX_18 : BOOL;
        _AUX_19 : BOOL;
        _AUX_20 : BOOL;
        _AUXT_21 : BOOL;
        _AUX_22 : BOOL;
        _AUXT_23 : BOOL;
        _AUX_24 : BOOL;
        _AUX_25 : BOOL;
        _AUX_26 : BOOL;
        _AUX_27 : BOOL;
        _AUXT_28 : BOOL;
        _AUX_29 : BOOL;
        _AUX_30 : BOOL;
        _AUX_31 : BOOL;
        _AUX_32 : BOOL;
        _AUXT_33 : BOOL;
        _AUX_34 : BOOL;
        _AUX_35 : BOOL;
        _AUX_36 : BOOL;
        _AUX_37 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        T3 : TON;
        T4 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LD _THIS.sDigital_alimentador.activado
AND marcha.accionado
ST _AUXT_1
LDN tempo1
AND marcha.accionado
ST _AUXT_3
LDN tempo2
ST _AUX_7
LDN _THIS.sDigital_medidor.activado
AND _AUX_7
AND marcha.accionado
ST _AUXT_6
LD _THIS.sDigital_medidor.activado
AND marcha.accionado
ST _AUXT_11
LDN tempo3
AND marcha.accionado
ST _AUXT_13
LDN tempo4
ST _AUX_17
LDN _THIS.sDigital_centro.activado
```

Figura D.43: Traducción del a IL del método *"Manual"* de la clase *"Trasvasador"* - parte 1

```
AND _AUX_17
AND marcha.accionado
ST _AUXT_16
LD _THIS.sDigital_centro.activado
AND marcha.accionado
ST _AUXT_21
LDN tempo4
ST _AUX_24
LDN _THIS.sDigital_centro.activado
AND _AUX_24
AND marcha.accionado
ST _AUXT_23
LDN tempo2
ST _AUX_29
LDN _THIS.sDigital_medidor.activado
AND _AUX_29
AND marcha.accionado
ST _AUXT_28
LDN tempo0
ST _AUX_34
LDN _THIS.sDigital_alimentador.activado
AND _AUX_34
AND marcha.accionado
ST _AUXT_33
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD TRUE
AND E8.C
S E0.X
R E8.X
LD _AUXT_1
ANDN _AUXT_33
AND E0.C
S E1.X
R E0.X
LD _AUXT_3
AND E1.C
S E2.X
R E1.X
LD _AUXT_11
ANDN _AUXT_28
AND E2.C
S E3.X
R E2.X
LD _AUXT_13
AND E3.C
S E4.X
R E3.X
LD _AUXT_21
ANDN _AUXT_23
AND E4.C
S E5.X
R E4.X
LD E5.C
AND TRUE
S E6.X
R E5.X
LD E6.C
AND TRUE
S E7.X
R E6.X
LD E7.C
AND TRUE
OR ( E4.C
AND _AUXT_23
)
OR ( E2.C
AND _AUXT_28
)
OR ( E0.C
AND _AUXT_33
)
S E8.X
R E7.X
```

Figura D.44: Traducción del a IL del método *"Manual"* de la clase *"Trasvasador"* - parte 2

```
R E4.X
R E2.X
R E0.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E1.X
S _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E2.X , PT := T#5s)
LD T2.Q
ST tempo2
LD E3.X
R _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E3.X , PT := T#5s)
LD T3.Q
ST tempo3
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E4.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E5.X
S _THIS.ObjetoComplejoBase.finProceso
LD E8.X
S _THIS.ObjetoComplejoBase.fallo
LD E0.X
OR E4.X
ST _THIS.cilBi_brazo.accionado1
LD E2.X
ST _THIS.cilBi_brazo.accionado2
END_FUNCTION_BLOCK
```

Figura D.45: Traducción del a IL del método *"Manual"* de la clase *"Trasvasador"* - parte 3

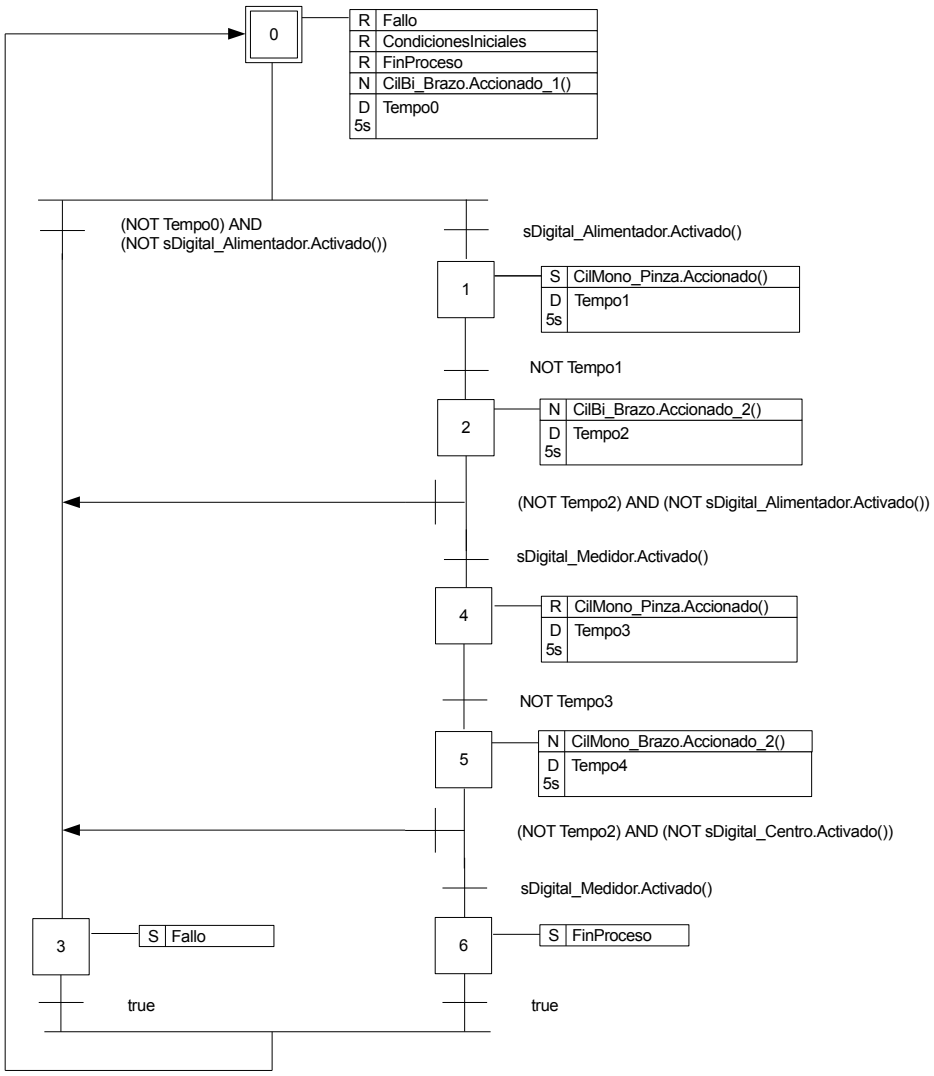

Figura D.46: Método *"Automatico"* de la clase *"Trasvasador"*

```
FUNCTION_BLOCK Trasvasador_Automatico
VAR_IN_OUT
        _THIS : Transvasador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUXT_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        _AUX_15 : BOOL;
        _AUX_16 : BOOL;
        _AUXT_17 : BOOL;
        _AUX_18 : BOOL;
        _AUX_19 : BOOL;
        _AUX_20 : BOOL;
        _AUXT_21 : BOOL;
        _AUX_22 : BOOL;
        _AUX_23 : BOOL;
        _AUX_24 : BOOL;
        E0 : TSFCTYPE;
        E3 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        T3 : TON;
        T4 : TON;
END_VAR
LD FALSE
ST ENO
LDN EN
JMPC Fin_Transvasar_Transvasador_1
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUX_2
LDN _THIS.sDigital_alimentador.activado
AND _AUX_2
ST _AUXT_1
LDN tempo1
ST _AUXT_5
LDN tempo2
ST _AUX_8
LDN _THIS.sDigital_medidor.activado
AND _AUX_8
ST _AUXT_7
LDN tempo3
ST _AUXT_11
LDN tempo4
ST _AUX_14
LDN _THIS.sDigital_centro.activado
AND _AUX_14
ST _AUXT_13
LDN tempo4
ST _AUX_18
LDN _THIS.sDigital_centro.activado
AND _AUX_18
ST _AUXT_17
LDN tempo2
ST _AUX_22
LDN _THIS.sDigital_medidor.activado
AND _AUX_22
ST _AUXT_21
LD E0.X
ST E0.C
```

Figura D.47: Traducción del a IL del método *"Automatico"* de la clase *"Trasvasador"* - parte 1

```
LD E3.X
ST E3.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E3.C
AND TRUE
OR ( E8.C
AND TRUE
)
S E0.X
R E3.X
R E8.X
LD _AUXT_1
ANDN _THIS.sDigital_alimentador.activado
AND E0.C
OR ( E5.C
AND _AUXT_17
)
OR ( E2.C
AND _AUXT_21
)
S E3.X
R E0.X
R E5.X
R E2.X
LD _THIS.sDigital_alimentador.activado
ANDN _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_5
AND E1.C
S E2.X
R E1.X
LD _THIS.sDigital_medidor.activado
ANDN _AUXT_21
AND E2.C
S E4.X
R E2.X
LD _AUXT_11
AND E4.C
S E5.X
R E4.X
LD _THIS.sDigital_centro.activado
ANDN _AUXT_17
AND E5.C
S E6.X
R E5.X
LD E6.C
AND TRUE
S E7.X
R E6.X
LD E7.C
AND TRUE
S E8.X
R E7.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.condicionesIniciales
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
LD E1.X
S _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E2.X , PT := T#5s)
LD T2.Q
ST tempo2
```

Figura D.48: Traducción del a IL del método *"Automatico"* de la clase *"Trasvasador"* - parte 2

```
LD E4.X
R _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E4.X , PT := T#5s)
LD T3.Q
ST tempo3
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E5.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E6.X
S _THIS.ObjetoComplejoBase.finProceso
LD E0.X
ST _THIS.cilBi_brazo.accionado1
LD E2.X
OR E5.X
ST _THIS.cilBi_brazo.accionado2
END_FUNCTION_BLOCK
```

Figura D.49: Traducción del a IL del método *"Automatico"* de la clase *"Trasvasador"* - parte 3

la medida de altura del rodamiento, además de cuatro contadores de tiempo y un atributo boolean que indica si la medición es correcta. Posee cuatro servicios, uno para la inicialización de la clase, dos para la ejecución de la medición (ejecución automática y manual) y un último servicio que devuelve si el tamaño del rodamiento es correcto. En el algoritmo D.17 se muestra la estructura de la clase y en la figura D.50 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.18). En la figura D.51 se muestra la traducción a código IL del método constructor.

Método InicializarElementos

Este método coloca todos los elementos del objeto en su posición inicial (ver figura D.52). En las figuras D.53 y D.54 se muestra la traducción a código IL del método.

Métodos de la clase GetTamanioValido

El método GetTamanioValido devuelve el valor del atributo privado *"tamanioValido"* (ver algoritmo D.19). En la figura D.55 se muestra la traducción a código

Algoritmo D.17 Clase *"Medidor"*

```
CLASS Medidor (ObjetoComplejoBase)
  PRIVATE tamanioValido : BOOL;
  PRIVATE referencia : INT;
  PRIVATE cilBi_elevador : ActuadorBiestable;
  PRIVATE sDigital_abajo : SensorDigital;
  PRIVATE sDigital_arriba : SensorDigital;
  PRIVATE cilMono_centrador : ActuadorMonoestable;
  PRIVATE sAnalog_palpador : SensorAnalogico;
  PRIVATE T0 : TON;
  PRIVATE T1 : TON;
  PRIVATE T2 : TON;
  PRIVATE T3 : TON;
  PRIVATE T4 : TON;
  PUBLIC METHOD InicializarElementos ( ) : void;
  PUBLIC METHOD GetTamanioValido ( ) : BOOL;
  PUBLIC METHOD Manual ( ) : void;
  PUBLIC METHOD Automatico ( ) : void;
END_CLASS
```

```
TYPE Medidor:
   STRUCT
      vPointer : *VOID;
      tamanioValido : BOOL;
      referencia : INT;
      cilBi_elevador : ActuadorBiestable;
      sDigital_abajo : SensorDigital;
      sDigital_arriba : SensorDigital;
      cilMono_centrador : ActuadorMonoestable;
      sAnalog_palpador : SensorAnalogico;
      T0 : TON;
      T1 : TON;
      T2 : TON;
      T3 : TON;
      T4 : TON;
      _ObjetoComplejoBase : ObjetoComplejoBase;
   END_STRUCT;
END TYPE
```

Figura D.50: Traducción de la clase *"Medidor"* a IL

Algoritmo D.18 Constructor de la clase *"Medidor"*

```
METHOD Medidor::Medidor ( )
  THIS.vPointer:=&vPointer+34;
END_METHOD
```

```
FUNCTION_BLOCK Medidor_Medidor
VAR_IN_OUT
        _This : Medidor;
END_VAR
VAR
       aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 34
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.51: Traducción del a IL del constructor de la clase *"Medidor"*

IL de dicho método.

Método Manual (ejecución manual)

Este método ejecuta el trasvase de rodamientos de forma manual (ver figura D.56). En las figuras D.57 y D.58 se muestra la traducción del método código IL.

La tarea *"Medición"* asociada a la etapa 2 contiene la línea de codigo en ST:

```
tamanioValido:=(referencia=sAnalog_palpador.valor);
```

Método Automatico (ejecución automática)

Este método ejecuta el trasvase de rodamientos de forma automática (ver figura

Algoritmo D.19 Método *"GetTamanioValido"* de la clase *"Medidor"*

```
METHOD MEdidor::GetTamanioValido ( )
  RETURN tamanioValido;
END_METHOD
```

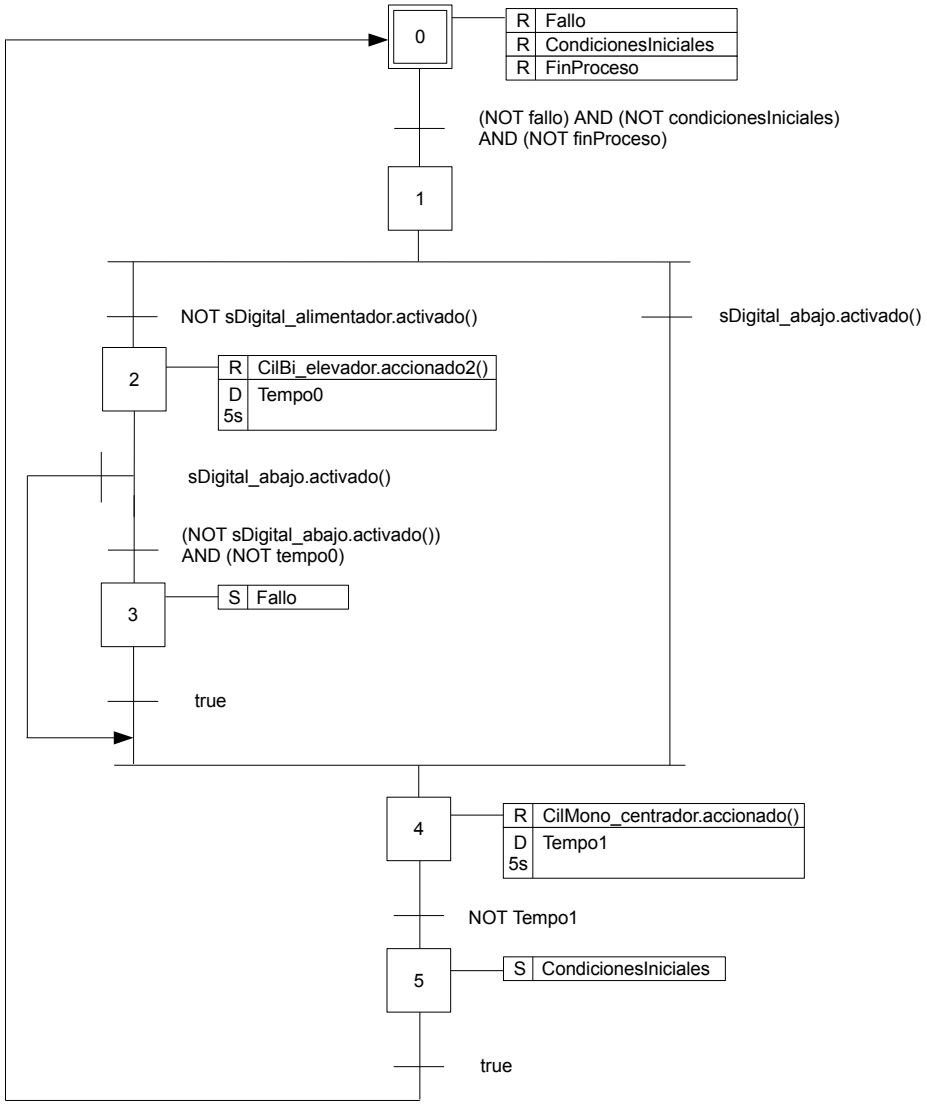

Figura D.52: Método *"InicializarElementos"* de la clase *"Medidor"*

```
FUNCTION_BLOCK Medidor_InicializarElementos
VAR_IN_OUT
        _THIS : Medidor;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUX_11 : BOOL;
        _AUXT_12 : BOOL;
        _AUX_13 : BOOL;
        _AUX_14 : BOOL;
        _AUX_15 : BOOL;
        _AUXT_16 : BOOL;
        _AUX_17 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN _THIS.ObjetoComplejoBase.fallo
ST _AUX_2
LDN _THIS.ObjetoComplejoBase.condicionesIniciales
AND _AUX_2
ST _AUX_4
LDN _THIS.ObjetoComplejoBase.finProceso
AND _AUX_4
ST _AUXT_1
LDN _THIS.sDigital_arriba.activado
ST _AUX_8
LDN _THIS.sDigital_abajo.activado
AND _AUX_8
OR _THIS.sDigital_arriba.activado
ST _AUXT_7
LDN tempo0
ST _AUX_13
LDN _THIS.sDigital_abajo.activado
AND _AUX_13
ST _AUXT_12
LDN tempo1
ST _AUXT_16
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD TRUE
AND E6.C
S E0.X
R E6.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_7
ANDN _THIS.sDigital_abajo.activado
AND E1.C
S E2.X
R E1.X
LD _AUXT_12
ANDN _THIS.sDigital_abajo.activado
```

Figura D.53: Traducción del a IL del método *"InicializarElementos"* de la clase *"Medidor"* - parte 1

```
AND E2.C
S E3.X
R E2.X
LD E3.C
AND TRUE
OR ( E2.C
AND _THIS.sDigital_abajo.activado
)
S E4.X
R E3.X
LD E1.C
AND _THIS.sDigital_abajo.activado
OR ( E4.C
AND TRUE
)
S E5.X
R E4.X
LD _AUXT_16
AND E5.C
S E6.X
R E5.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E2.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
LD E5.X
R _THIS.cilMono_centrador.accionado
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E5.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E6.X
S _THIS.ObjetoComplejoBase.condicionesIniciales
LD E2.X
ST _THIS.cilBi_elevador.accionado2
END_FUNCTION_BLOCK
```

Figura D.54: Traducción del a IL del método *"InicializarElementos"* de la clase *"Medidor"* - parte 2

```
FUNCTION_BLOCK Medidor_GetTamanioValido
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : Medidor;
END_VAR

LD _This.tamanioValido
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.55: Traducción del método *"GetTamanioValido"* de la clase *"Medidor"* a IL

D.59). En las figuras D.60 y D.61 se muestra la traducción a código IL del método.

La tarea *"Medición"* asociada a la etapa 2 contiene la línea de codigo en ST:

```
tamanioValido:=(referencia=sAnalog_palpador.valor);
```

D.7.12. Clase Evacuador

La función de esta clase es la de transportar el rodamiento a la zona del transfer en el caso de que el rodamiento sea correcto o expulsarlo en caso contrario. Se trata de una clase elaborada que hereda de *"ObjetoComplejoBase"* y está formada por cuatro actuadores genéricos monoestables, cuatro sensores magnéticos de proximidad para detectar la posición de los cilindros necesarios y cinco contadores de tiempo. Posee además 5 servicios que se encargan de inicializar la clase, expulsar el rodamiento (tanto de forma manual como automática) y de ejecutar lo operatividad de la propia clase (de forma automática o manual). En el algoritmo D.20 se muestra la estructura de la clase y en la figura D.62 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.21). En la figura D.63 se muestra la traducción a código IL del método constructor.

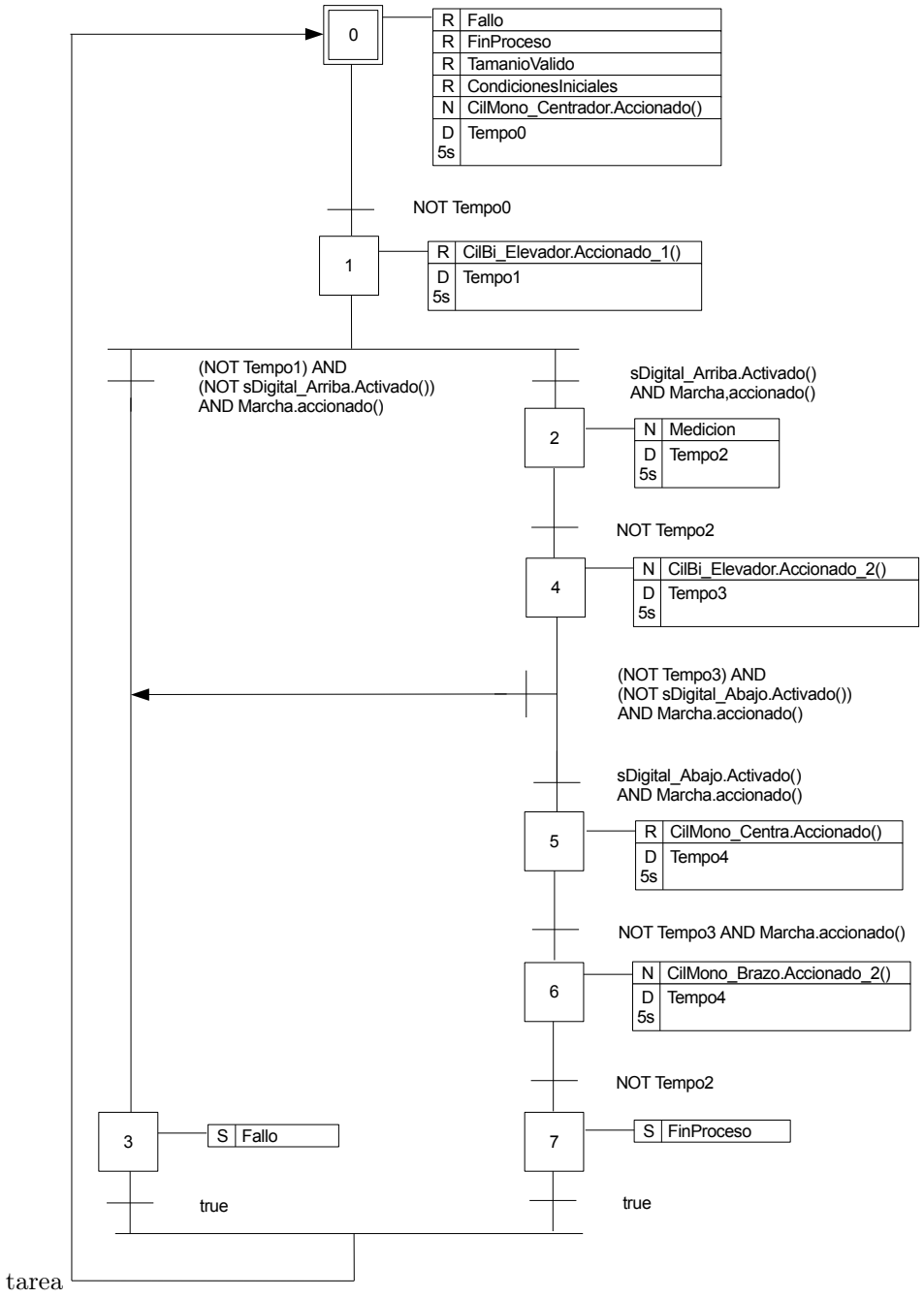

Figura D.56: Método *"Manual"* de la clase *"Medidor"*

```
FUNCTION_BLOCK Medidor_Manual
VAR_IN_OUT
        _THIS : Medidor;
        marcha : Pulsador;
END_VAR
VAR_INPUT
        referencia : INT;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUXT_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUX_7 : BOOL;
        _AUX_8 : BOOL;
        _AUXT_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        _AUX_15 : BOOL;
        _AUX_16 : BOOL;
        _AUX_17 : BOOL;
        _AUXT_18 : BOOL;
        _AUX_19 : BOOL;
        _AUXT_20 : BOOL;
        _AUX_21 : BOOL;
        _AUX_22 : BOOL;
        _AUXT_23 : BOOL;
        _AUX_24 : BOOL;
        _AUX_25 : BOOL;
        _AUX_26 : BOOL;
        _AUX_27 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E3 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        medicion : BOOL := FALSE;
        _AUX_28 : BOOL;
        T3 : TON;
        T4 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
AND marcha.accionado
ST _AUXT_1
LDN tempo1
ST _AUX_5
LDN _THIS.sDigital_arriba.activado
AND _AUX_5
AND marcha.accionado
ST _AUXT_4
LD _THIS.sDigital_arriba.activado
AND marcha.accionado
ST _AUXT_9
LDN tempo2
ST _AUXT_11
LDN tempo3
ST _AUX_14
LDN _THIS.sDigital_abajo.activado
AND _AUX_14
AND marcha.accionado
ST _AUXT_13
LD _THIS.sDigital_abajo.activado
AND marcha.accionado
ST _AUXT_18
LDN tempo4
AND marcha.accionado
ST _AUXT_20
LDN tempo3
```

Figura D.57: Traducción del a IL del método *"Manual"* de la clase *"Medidor"* - parte 1

```
ST _AUX_24
LDN _THIS.sDigital_abajo.activado
AND _AUX_24
AND marcha.accionado
ST _AUXT_23
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E3.X
ST E3.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E3.C
AND TRUE
OR ( E7.C
AND TRUE
)
S E0.X
R E3.X
R E7.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_4
ANDN _AUXT_9
AND E1.C
OR ( E4.C
AND _AUXT_23
)
S E3.X
R E1.X
R E4.X
LD _AUXT_9
ANDN _AUXT_4
AND E1.C
S E2.X
R E1.X
LD _AUXT_11
AND E2.C
S E4.X
R E2.X
LD _AUXT_18
ANDN _AUXT_23
AND E4.C
S E5.X
R E4.X
LD _AUXT_20
AND E5.C
S E6.X
R E5.X
LD E6.C
AND TRUE
S E7.X
R E6.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.tamanioValido
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
S _THIS.cilMono_centrador.accionado
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E2.X , PT := T#5s)
LD T2.Q
ST tempo2
LDN medicion
JMPC _SALTO_0
LD referencia
EQ _THIS.sAnalog_palpador.valor
ST _THIS.tamanioValido
_SALTO_0:
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E4.X , PT := T#5s)
LD T3.Q
ST tempo3
LD E5.X
R _THIS.cilMono_centrador.accionado
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E5.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E6.X
S _THIS.ObjetoComplejoBase.finProceso
LD E1.X
ST _THIS.cilBi_elevador.accionado1
LD E4.X
ST _THIS.cilBi_elevador.accionado2
LD E2.X
ST medicion
END_FUNCTION_BLOCK
```

Figura D.58: Traducción del a IL del método *"Manual"* de la clase *"Medidor"* - parte 2

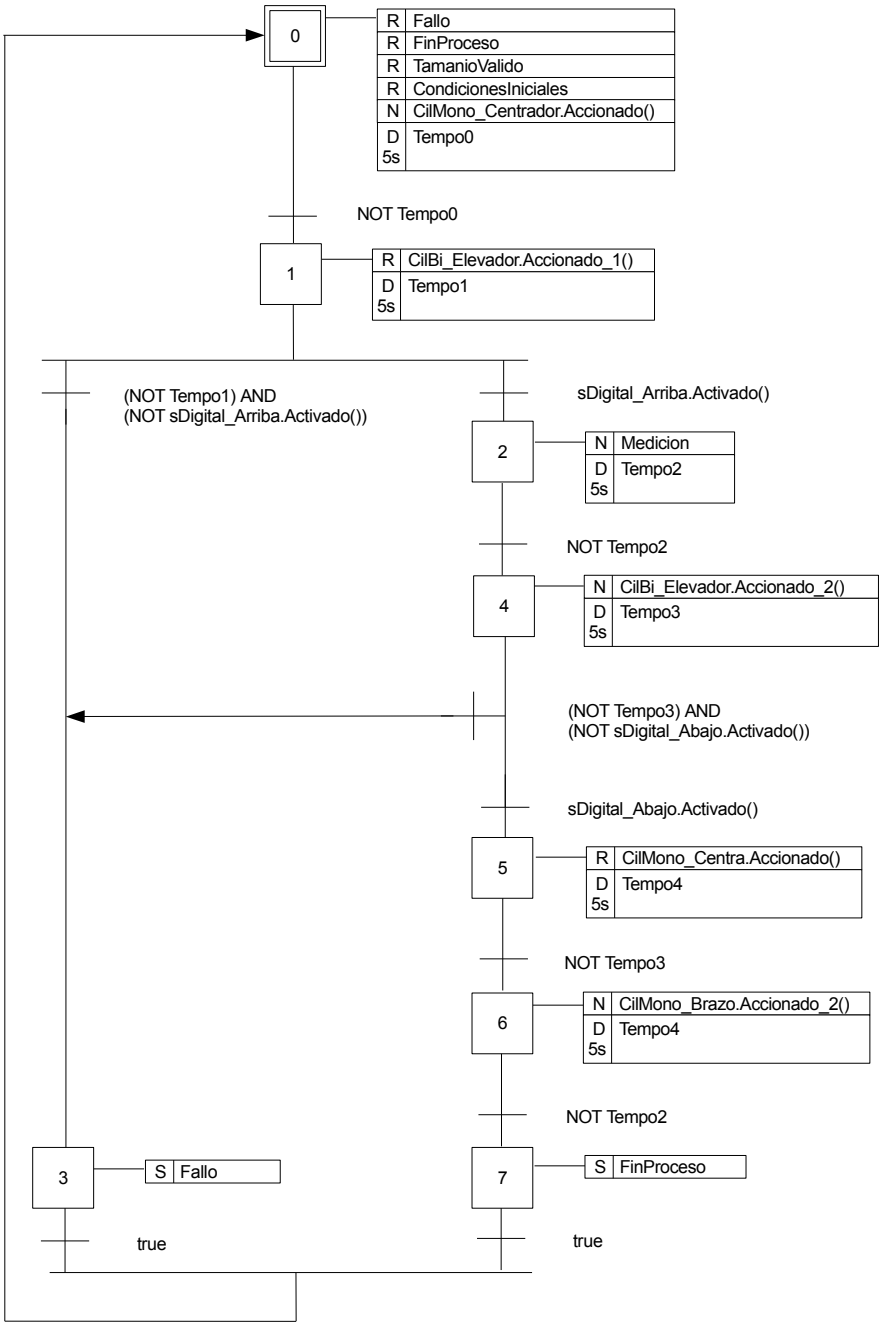

Figura D.59: Método *"Automatico"* de la clase *"Medidor"*

```
FUNCTION_BLOCK Medidor_Automatico
VAR_IN_OUT
        _THIS : Medidor;
END_VAR
VAR_INPUT
        referencia : INT;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUXT_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUXT_9 : BOOL;
        _AUX_10 : BOOL;
        _AUX_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        _AUXT_15 : BOOL;
        _AUX_16 : BOOL;
        _AUX_17 : BOOL;
        _AUX_18 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E3 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        medicion : BOOL := FALSE;
        _AUX_19 : BOOL;
        T2 : TON;
        T3 : TON;
        T4 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUXT_1
LDN tempo1
ST _AUX_4
LDN _THIS.sDigital_arriba.activado
AND _AUX_4
ST _AUXT_3
LDN tempo2
ST _AUXT_7
LDN tempo3
ST _AUX_10
LDN _THIS.sDigital_abajo.activado
AND _AUX_10
ST _AUXT_9
LDN tempo4
ST _AUXT_13
LDN tempo3
ST _AUX_16
LDN _THIS.sDigital_abajo.activado
AND _AUX_16
ST _AUXT_15
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E3.X
ST E3.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
```

Figura D.60: Traducción del a IL del método *"Automatico"* de la clase *"Medidor"* - parte 1

```
LD E3.C
AND TRUE
OR ( E7.C
AND TRUE
)
S E0.X
R E3.X
R E7.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_3
ANDN _THIS.sDigital_arriba.activado
AND E1.C
OR ( E4.C
AND _AUXT_15
)
S E3.X
R E1.X
R E4.X
LD _THIS.sDigital_arriba.activado
ANDN _AUXT_3
AND E1.C
S E2.X
R E1.X
LD _AUXT_7
AND E2.C
S E4.X
R E2.X
LD _THIS.sDigital_abajo.activado
ANDN _AUXT_15
AND E4.C
S E5.X
R E4.X
LD _AUXT_13
AND E5.C
S E6.X
R E5.X
LD E6.C
AND TRUE
S E7.X
R E6.X
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.tamanioValido
LD E0.X
S _THIS.cilMono_centrador.accionado
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E1.X
S _THIS.cilBi_elevador.accionado1
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
LDN medicion
JMPC _SALTO_0
LD referencia
EQ _THIS.sAnalog_palpador.valor
ST _THIS.tamanioValido
_SALTO_0:
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E2.X , PT := T#5s)
LD T2.Q
ST tempo2
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E4.X , PT := T#5s)
LD T3.Q
ST tempo3
LD E5.X
R _THIS.cilMono_centrador.accionado
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E5.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E6.X
S _THIS.ObjetoComplejoBase.finProceso
LD E4.X
ST _THIS.cilBi_elevador.accionado2
LD E2.X
ST medicion
END_FUNCTION_BLOCK
```

Figura D.61: Traducción del a IL del método *"Automatico"* de la clase *"Medidor"* - parte 2

Algoritmo D.20 Clase Evacuador

```
CLASS Evacuador (ObjetoComplejoBase)
  PRIVATE sDigital_medidor : SensorDigital;
  PRIVATE sDigital_transporte : SensorDigital;
  PRIVATE sDigital_abajo : SensorDigital;
  PRIVATE sDigital_arriba : SensorDigital;
  PRIVATE cilMono_expulsor : ActuadorMonoestable;
  PRIVATE cilMono_pinza : ActuadorMonoestable;
  PRIVATE cilMono_brazo : ActuadorMonoestable;
  PRIVATE cilMono_elevador : ActuadorMonoestable;
  PRIVATE T0 : TON;
  PRIVATE T1 : TON;
  PRIVATE T2 : TON;
  PRIVATE T3 : TON;
  PRIVATE T4 : TON;
  PRIVATE T5 : TON;
  PUBLIC METHOD InicializarElementos ( ) : void;
  PUBLIC METHOD ExpulsarManual (marcha : & Pulsador)
                : ActuadorBiestable;
  PUBLIC METHOD ExpulsarAutomatico ( ) : void;
  PUBLIC METHOD Manual ( ) : void;
  PUBLIC METHOD Automatico ( ) : void;
END_CLASS
```

Algoritmo D.21 Constructor de la clase *"Evacuador"*

```
METHOD Evacuador::Evacuador ( )
  THIS.vPointer:=&vPointer+37;
END_METHOD
```

```
TYPE Evacuador:
   STRUCT
     vPointer : *VOID;
      sDigital_medidor : SensorDigital;
      sDigital_transporte : SensorDigital;
      sDigital_abajo : SensorDigital;
      sDigital_arriba : SensorDigital;
      cilMono_expulsor : ActuadorMonoestable;
      cilMono_pinza : ActuadorMonoestable;
      cilMono_brazo : ActuadorMonoestable;
      cilMono_elevador : ActuadorMonoestable;
      T0 : TON;
      T1 : TON;
      T2 : TON;
      T3 : TON;
      T4 : TON;
      T5 : TON;
      _ObjetoComplejoBase : ObjetoComplejoBase;
   END_STRUCT;
END_TYPE
```

Figura D.62: Traducción de la clase *"Evacuador"* a IL

```
FUNCTION_BLOCK Evacuador_Evacuador
VAR_IN_OUT
        _This : Evacuador;
END_VAR
VAR
      aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 37
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.63: Traducción del a IL del constructor de la clase *"Evacuador"*

Método InicializarElementos

Este método coloca todos los elementos del objeto en su posición inicial (ver figura D.64). En las figuras D.65, D.66 y D.67 se muestra la traducción a código IL del método.

Método ExpulsarManual

Este método permite accionar el cilindro expulsador de forma manual, logrando empujar el rodamiento hacia una rampa donde los rodamientos inadecuados serán retirados por el operador (ver figura D.68). En la figura D.69 se muestra la traducción a código IL del método.

Método ExpulsarAutomatico

Este método permite accionar el cilindro expulsador de forma automática, logrando empujar el rodamiento hacia una rampa donde los rodamientos inadecuados serán retirados por el operador (ver figura D.70). En la figura D.71 se muestra la traducción a código IL del método.

Método Manual (ejecución manual)

Este método ejecuta la evacuación de rodamientos de forma manual (ver figura D.72). En las figuras D.73, D.74 y D.75 se muestra la traducción del método código IL.

Método Automatico (ejecución automática)

Este método ejecuta la evacuación de rodamientos de forma automática (ver figura D.76). En las figuras D.97, D.78 y D.79 se muestra la traducción del método código IL.

D.7.13. Clase PanelMando

La función de esta clase es la de simular el panel de control que tiene el operario de la estación. Está formado por tres pulsadores, dos selectores de dos posiciones y una lamapra de aviso de falta de material. Posee varios servicios entre los que se cuenta los que permiten manejar las lamparas de aviso (para encender y apagarlas), los que permiten parar y rearmar la estación, los que permiten comprobar el tipo

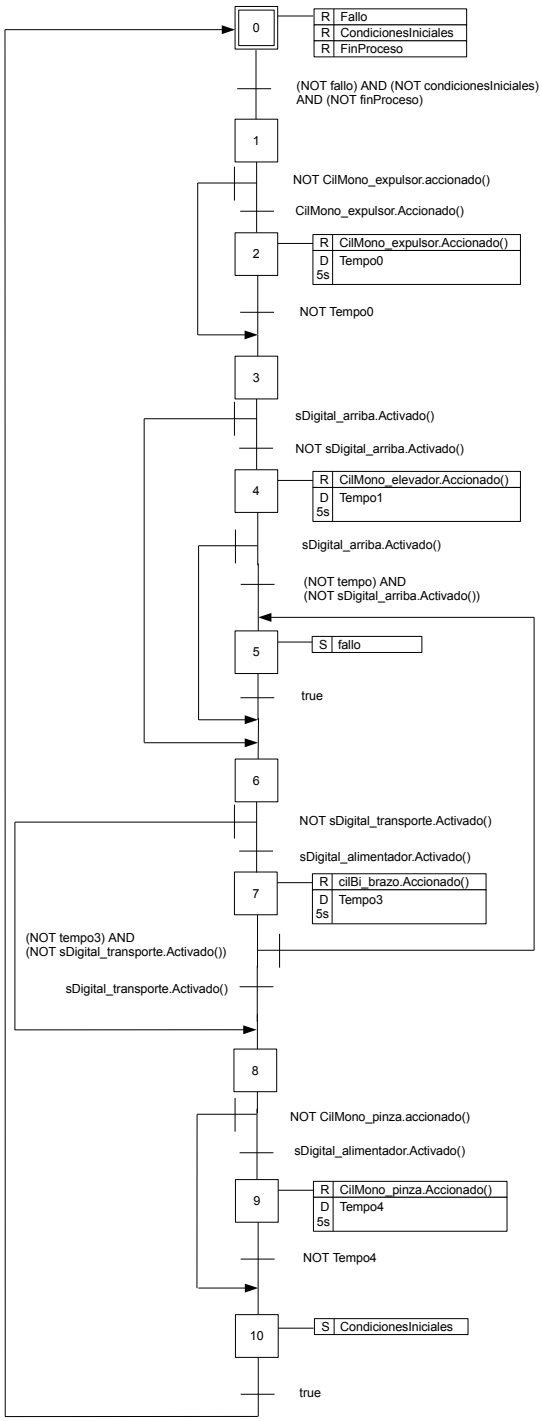

Figura D.64: Método *"InicializarElementos"* de la clase *"Evacuador"*

```
FUNCTION_BLOCK Evacuador_InicializarElementos
VAR_IN_OUT
        _THIS : Evacuador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUXT_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        _AUX_15 : BOOL;
        _AUX_16 : BOOL;
        _AUXT_17 : BOOL;
        _AUX_18 : BOOL;
        _AUXT_19 : BOOL;
        _AUX_20 : BOOL;
        _AUX_21 : BOOL;
        _AUX_22 : BOOL;
        _AUXT_23 : BOOL;
        _AUX_24 : BOOL;
        _AUX_25 : BOOL;
        _AUX_26 : BOOL;
        _AUXT_27 : BOOL;
        _AUX_28 : BOOL;
        _AUXT_29 : BOOL;
        _AUX_30 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        E9 : TSFCTYPE;
        E10 : TSFCTYPE;
        E11 : TSFCTYPE;
        E12 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        T3 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN _THIS.ObjetoComplejoBase.fallo
ST _AUX_2
LDN _THIS.ObjetoComplejoBase.condicionesIniciales
AND _AUX_2
ST _AUX_4
LDN _THIS.ObjetoComplejoBase.finProceso
AND _AUX_4
ST _AUXT_1
LDN _THIS.cilMono_expulsor.accionado
ST _AUXT_7
LDN tempo0
ST _AUXT_9
LDN _THIS.sDigital_arriba.activado
ST _AUXT_11
LDN tempo1
ST _AUX_14
LDN _THIS.sDigital_arriba.activado
AND _AUX_14
ST _AUXT_13
LDN _THIS.sDigital_transporte.activado
ST _AUXT_17
LDN tempo2
ST _AUX_20
LDN _THIS.sDigital_transporte.activado
AND _AUX_20
ST _AUXT_19
LDN tempo2
```

Figura D.65: Traducción del a IL del método *"InicializarElementos"* de la clase *"Evacuador"* - parte 1

```
ST _AUX_24
LDN _THIS.sDigital_transporte.activado
AND _AUX_24
ST _AUXT_23
LDN _THIS.cilMono_pinza.accionado
ST _AUXT_27
LDN tempo3
ST _AUXT_29
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E9.X
ST E9.C
LD E10.X
ST E10.C
LD E11.X
ST E11.C
LD E12.X
ST E12.C
LD FALSE
AND E12.C
S E0.X
R E12.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _THIS.cilMono_expulsor.accionado
ANDN _AUXT_7
AND E1.C
S E2.X
R E1.X
LD E1.C
AND _AUXT_7
OR ( E2.C
AND _AUXT_9
)
S E3.X
R E2.X
LD _AUXT_11
ANDN _THIS.sDigital_arriba.activado
AND E3.C
S E4.X
R E3.X
LD _AUXT_13
ANDN _THIS.sDigital_arriba.activado
AND E4.C
OR ( E8.C
AND _AUXT_23
)
S E5.X
R E4.X
R E8.X
LD E5.C
AND TRUE
OR ( E4.C
AND _THIS.sDigital_arriba.activado
)
S E6.X
R E5.X
LD E3.C
AND _THIS.sDigital_arriba.activado
OR ( E6.C
AND TRUE
)
S E7.X
R E6.X
LD _THIS.sDigital_transporte.activado
ANDN _AUXT_17
AND E7.C
S E8.X
R E7.X
LD E8.C
AND _THIS.sDigital_transporte.activad
S E9.X
```

Figura D.66: Traducción del a IL del método *"InicializarElementos"* de la clase *"Evacuador"* - parte 2

```
LD E7.C
AND _AUXT_17
OR ( E9.C
AND TRUE
)
S E10.X
R E9.X
LD _THIS.cilMono_pinza.accionado
ANDN _AUXT_27
AND E10.C
S E11.X
R E10.X
LD E10.C
AND _AUXT_27
OR ( E11.C
AND _AUXT_29
)
S E12.X
R E11.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E2.X
R _THIS.cilMono_expulsor.accionado
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E2.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E4.X
R _THIS.cilMono_elevador.accionado
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E4.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E5.X
S _THIS.ObjetoComplejoBase.fallo
LD E8.X
R _THIS.cilMono_brazo.accionado
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E8.X , PT := T#5s)
LD T2.Q
ST tempo2
LD E11.X
R _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E11.X , PT := T#5s)
LD T3.Q
ST tempo3
LD E12.X
S _THIS.ObjetoComplejoBase.condicionesIniciales
END_FUNCTION_BLOCK
```

Figura D.67: Traducción del a IL del método *"InicializarElementos"* de la clase *"Evacuador"* - parte 3

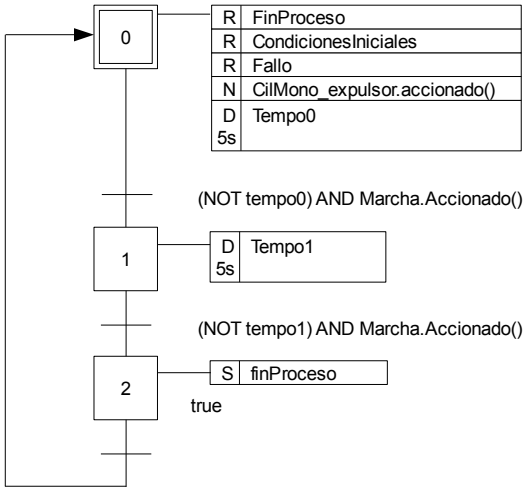

Figura D.68: Método *"ExpulsarManual"* de la clase *"Evacuador"*

```
FUNCTION_BLOCK Evacuador_ExpulsarManual
VAR_OUTPUT
        _Retorno : ActuadorBiestable;
END_VAR
VAR_IN_OUT
        _This : Evacuador;
        marcha : Pulsador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUXT_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
AND marcha.accionado
ST _AUXT_1
LDN tempo1
AND marcha.accionado
ST _AUXT_4
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD TRUE
AND E2.C
S E0.X
R E2.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_4
AND E1.C
S E2.X
R E1.X
LD E0.X
R _This.ObjetoComplejoBase.finProceso
LD E0.X
R _This.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E2.X
S _THIS.ObjetoComplejoBase.finProceso
LD E0.X
ST _THIS.cilMono_expulsor.accionado
END_FUNCTION_BLOCK
```

Figura D.69: Traducción del a IL del método *"ExpulsarManual"* de la clase *"Evacuador"*

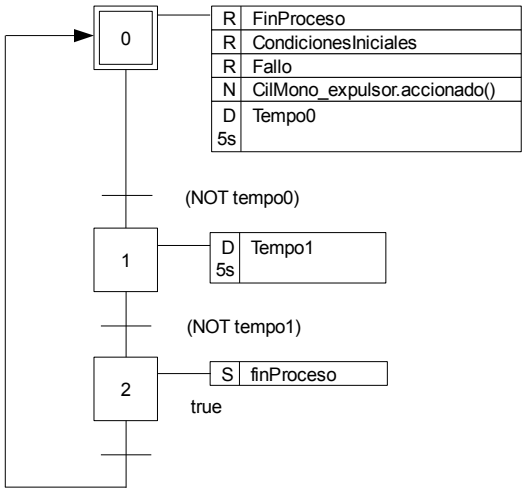

Figura D.70: Método *"ExpulsarAutomatico"* de la clase *"Evacuador"*

```
FUNCTION_BLOCK Evacuador_ExpulsarAutomatico
VAR_IN_OUT
        _THIS : Evacuador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUXT_3 : BOOL;
        _AUX_4 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUXT_1
LDN tempo1
ST _AUXT_3
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD TRUE
AND E2.C
S E0.X
R E2.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
LD _AUXT_3
AND E1.C
S E2.X
R E1.X
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E1.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E2.X
S _THIS.ObjetoComplejoBase.finProceso
LD E0.X
ST _THIS.cilMono_expulsor.accionado
END_FUNCTION_BLOCK
```

Figura D.71: Traducción del a IL del método *"ExpulsarAutomatico"* de la clase *"Evacuador"*

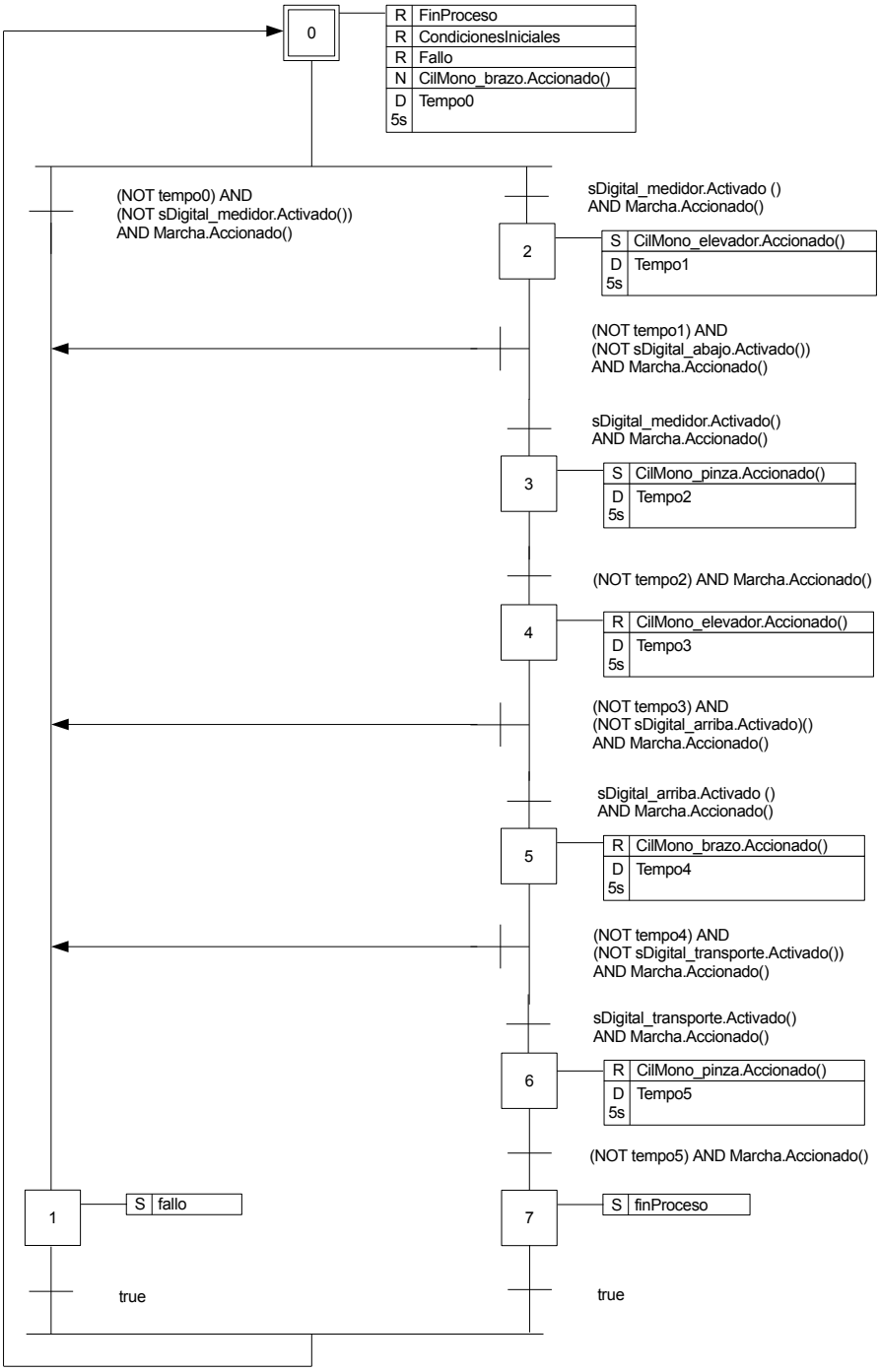

Figura D.72: Método *"Manual"* de la clase *"Evacuador"*

```
FUNCTION_BLOCK Evacuador_Manual
VAR_IN_OUT
        _THIS : Evacuador;
        marcha : Pulsador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        tempo5 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUXT_6 : BOOL;
        _AUX_7 : BOOL;
        _AUXT_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUX_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        _AUXT_15 : BOOL;
        _AUX_16 : BOOL;
        _AUX_17 : BOOL;
        _AUXT_18 : BOOL;
        _AUX_19 : BOOL;
        _AUX_20 : BOOL;
        _AUX_21 : BOOL;
        _AUX_22 : BOOL;
        _AUXT_23 : BOOL;
        _AUX_24 : BOOL;
        _AUXT_25 : BOOL;
        _AUX_26 : BOOL;
        _AUX_27 : BOOL;
        _AUX_28 : BOOL;
        _AUX_29 : BOOL;
        _AUXT_30 : BOOL;
        _AUX_31 : BOOL;
        _AUXT_32 : BOOL;
        _AUX_33 : BOOL;
        _AUX_34 : BOOL;
        _AUXT_35 : BOOL;
        _AUX_36 : BOOL;
        _AUX_37 : BOOL;
        _AUX_38 : BOOL;
        _AUX_39 : BOOL;
        _AUXT_40 : BOOL;
        _AUX_41 : BOOL;
        _AUX_42 : BOOL;
        _AUX_43 : BOOL;
        _AUX_44 : BOOL;
        _AUXT_45 : BOOL;
        _AUX_46 : BOOL;
        _AUX_47 : BOOL;
        _AUX_48 : BOOL;
        _AUX_49 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        E9 : TSFCTYPE;
        E10 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        T3 : TON;
        T4 : TON;
        T5 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUX_2
LDN _THIS.sDigital_medidor.activado
AND _AUX_2
AND marcha.accionado
```

Figura D.73: Traducción del a IL del método *"Manual"* de la clase *"Evacuador"* - parte 1

```
ST _AUXT_1
LD _THIS.sDigital_medidor.activado
AND marcha.accionado
ST _AUXT_6
LDN tempo1
ST _AUX_9
LDN _THIS.sDigital_abajo.activado
AND _AUX_9
AND marcha.accionado
ST _AUXT_8
LD _THIS.sDigital_abajo.activado
AND marcha.accionado
ST _AUXT_13
LDN tempo2
AND marcha.accionado
ST _AUXT_15
LDN tempo3
ST _AUX_19
LDN _THIS.sDigital_arriba.activado
AND _AUX_19
AND marcha.accionado
ST _AUXT_18
LD _THIS.sDigital_arriba.activado
AND marcha.accionado
ST _AUXT_23
LDN tempo4
ST _AUX_26
LDN _THIS.sDigital_transporte.activado
AND _AUX_26
AND marcha.accionado
ST _AUXT_25
LD _THIS.sDigital_transporte.activado
AND marcha.accionado
ST _AUXT_30
LDN tempo5
AND marcha.accionado
ST _AUXT_32
LDN tempo4
ST _AUX_36
LDN _THIS.sDigital_transporte.activado
AND _AUX_36
AND marcha.accionado
ST _AUXT_35
LDN tempo3
ST _AUX_41
LDN _THIS.sDigital_arriba.activado
AND _AUX_41
AND marcha.accionado
ST _AUXT_40
LDN tempo1
ST _AUX_46
LDN _THIS.sDigital_abajo.activado
AND _AUX_46
AND marcha.accionado
ST _AUXT_45
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E9.X
ST E9.C
LD E10.X
ST E10.C
LD E1.C
AND TRUE
OR ( E10.C
AND TRUE
)
S E0.X
R E1.X
R E10.X
LD _AUXT_1
ANDN _AUXT_6
AND E0.C
OR ( E5.C
AND _AUXT_35
```

Figura D.74: Traducción del a IL del método *"Manual"* de la clase *"Evacuador"* - parte 2

```
)
OR ( E4.C
AND _AUXT_40
)
OR ( E2.C
AND _AUXT_45
)
S E1.X
R E0.X
R E5.X
R E4.X
R E2.X
LD _AUXT_6
ANDN _AUXT_1
AND E0.C
S E2.X
R E0.X
LD _AUXT_13
ANDN _AUXT_45
AND E2.C
S E3.X
R E2.X
LD _AUXT_15
AND E3.C
S E4.X
R E3.X
LD _AUXT_23
ANDN _AUXT_40
AND E4.C
S E5.X
R E4.X
LD _AUXT_30
ANDN _AUXT_35
AND E5.C
S E6.X
R E5.X
LD _AUXT_32
AND E6.C
S E7.X
R E6.X
LD E7.C
AND TRUE
S E8.X
R E7.X
LD E8.C
AND TRUE
S E9.X
R E8.X
LD E9.C
AND TRUE
S E10.X
R E9.X
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
S _THIS.cilMono_brazo.accionado
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E1.X
S _THIS.ObjetoComplejoBase.fallo
LD E2.X
S _THIS.cilMono_elevador.accionado
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E2.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E3.X
S _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E3.X , PT := T#5s)
LD T2.Q
ST tempo2
LD E4.X
R _THIS.cilMono_elevador.accionado
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E4.X , PT := T#5s)
LD T3.Q
ST tempo3
LD E5.X
R _THIS.cilMono_brazo.accionado
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E5.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E6.X
R _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T5 *)
CAL T5(IN := E6.X , PT := T#5s)
LD T5.Q
ST tempo5
LD E7.X
S _THIS.ObjetoComplejoBase.finProceso
END_FUNCTION_BLOCK
```

Figura D.75: Traducción del a IL del método *"Manual"* de la clase *"Evacuador"* - parte 3

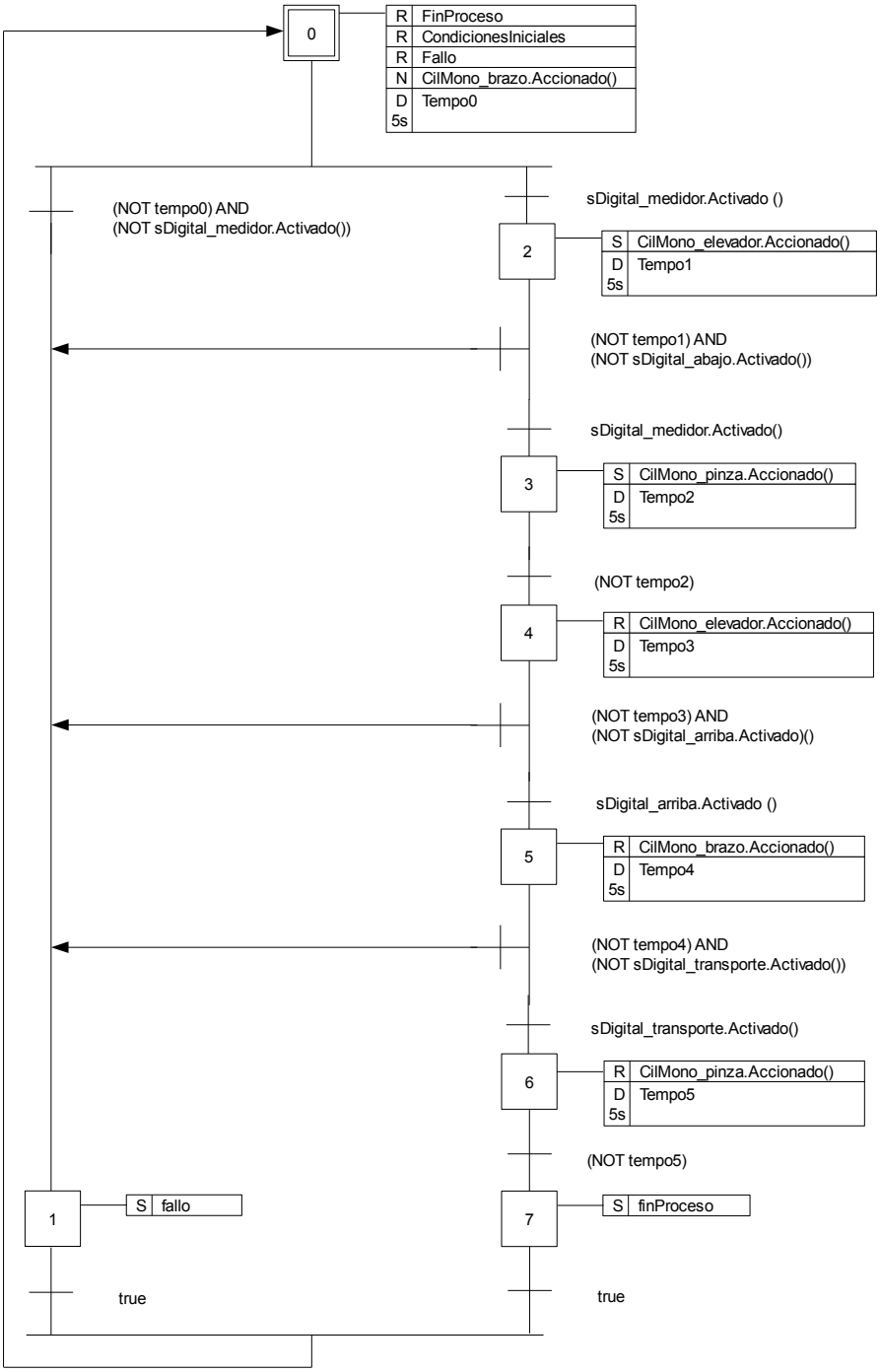

Figura D.76: Método *"Automatico"* de la clase *"Evacuador"*

```
FUNCTION_BLOCK Evacuador_Automatico
VAR_IN_OUT
        _THIS : Evacuador;
END_VAR

VAR (* Variables locales *)
        tempo0 : BOOL;
        tempo1 : BOOL;
        tempo2 : BOOL;
        tempo3 : BOOL;
        tempo4 : BOOL;
        tempo5 : BOOL;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUXT_5 : BOOL;
        _AUX_6 : BOOL;
        _AUX_7 : BOOL;
        _AUX_8 : BOOL;
        _AUXT_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUX_13 : BOOL;
        _AUX_14 : BOOL;
        _AUXT_15 : BOOL;
        _AUX_16 : BOOL;
        _AUX_17 : BOOL;
        _AUX_18 : BOOL;
        _AUXT_19 : BOOL;
        _AUX_20 : BOOL;
        _AUXT_21 : BOOL;
        _AUX_22 : BOOL;
        _AUX_23 : BOOL;
        _AUX_24 : BOOL;
        _AUXT_25 : BOOL;
        _AUX_26 : BOOL;
        _AUX_27 : BOOL;
        _AUX_28 : BOOL;
        _AUXT_29 : BOOL;
        _AUX_30 : BOOL;
        _AUX_31 : BOOL;
        _AUX_32 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        E9 : TSFCTYPE;
        E10 : TSFCTYPE;
        T0 : TON;
        T1 : TON;
        T2 : TON;
        T3 : TON;
        T4 : TON;
        T5 : TON;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN tempo0
ST _AUX_2
LDN _THIS.sDigital_medidor.activado
AND _AUX_2
ST _AUXT_1
LDN tempo1
ST _AUX_6
LDN _THIS.sDigital_abajo.activado
AND _AUX_6
ST _AUXT_5
LDN tempo2
ST _AUXT_9
LDN tempo3
ST _AUX_12
LDN _THIS.sDigital_arriba.activado
AND _AUX_12
ST _AUXT_11
LDN tempo4
ST _AUX_16
LDN _THIS.sDigital_transporte.activado
AND _AUX_16
ST _AUXT_15
LDN tempo5
```

Figura D.77: Traducción del a IL del método *"Automatico"* de la clase *"Evacuador"* - parte 1

```
ST _AUXT_19
LDN tempo4
ST _AUX_22
LDN _THIS.sDigital_transporte.activado
AND _AUX_22
ST _AUXT_21
LDN tempo3
ST _AUX_26
LDN _THIS.sDigital_arriba.activado
AND _AUX_26
ST _AUXT_25
LDN tempo1
ST _AUX_30
LDN _THIS.sDigital_abajo.activado
AND _AUX_30
ST _AUXT_29
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E9.X
ST E9.C
LD E10.X
ST E10.C
LD E1.C
AND TRUE
OR ( E10.C
AND TRUE
)
S E0.X
R E1.X
R E10.X
LD _AUXT_1
ANDN _THIS.sDigital_medidor.activado
AND E0.C
OR ( E5.C
AND _AUXT_21
)
OR ( E4.C
AND _AUXT_25
)
OR ( E2.C
AND _AUXT_29
)
S E1.X
R E0.X
R E5.X
R E4.X
R E2.X
LD _THIS.sDigital_medidor.activado
ANDN _AUXT_1
AND E0.C
S E2.X
R E0.X
LD _THIS.sDigital_abajo.activado
ANDN _AUXT_29
AND E2.C
S E3.X
R E2.X
LD _AUXT_9
AND E3.C
S E4.X
R E3.X
LD _THIS.sDigital_arriba.activado
ANDN _AUXT_25
AND E4.C
S E5.X
R E4.X
LD _THIS.sDigital_transporte.activado
ANDN _AUXT_21
AND E5.C
S E6.X
R E5.X
LD _AUXT_19
AND E6.C
S E7.X
```

Figura D.78: Traducción del a IL del método *"Automatico"* de la clase *"Evacuador"* - parte 2

```
R E6.X
LD E7.C
AND TRUE
S E8.X
R E7.X
LD E8.C
AND TRUE
S E9.X
R E8.X
LD E9.C
AND TRUE
S E10.X
R E9.X
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
S _THIS.cilMono_brazo.accionado
(* Llamada a bloque funcional 'T0 *)
CAL T0(IN := E0.X , PT := T#5s)
LD T0.Q
ST tempo0
LD E1.X
S _THIS.ObjetoComplejoBase.fallo
LD E2.X
S _THIS.cilMono_elevador.accionado
(* Llamada a bloque funcional 'T1 *)
CAL T1(IN := E2.X , PT := T#5s)
LD T1.Q
ST tempo1
LD E3.X
S _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T2 *)
CAL T2(IN := E3.X , PT := T#5s)
LD T2.Q
ST tempo2
LD E4.X
R _THIS.cilMono_elevador.accionado
(* Llamada a bloque funcional 'T3 *)
CAL T3(IN := E4.X , PT := T#5s)
LD T3.Q
ST tempo3
LD E5.X
R _THIS.cilMono_brazo.accionado
(* Llamada a bloque funcional 'T4 *)
CAL T4(IN := E5.X , PT := T#5s)
LD T4.Q
ST tempo4
LD E6.X
R _THIS.cilMono_pinza.accionado
(* Llamada a bloque funcional 'T5 *)
CAL T5(IN := E6.X , PT := T#5s)
LD T5.Q
ST tempo5
LD E7.X
S _THIS.ObjetoComplejoBase.finProceso
END_FUNCTION_BLOCK
```

Figura D.79: Traducción del a IL del método *"Automatico"* de la clase *"Evacuador"* - parte 3

Algoritmo D.22 Clase *"PanelMando"*

```
CLASS PanelMando ( )
  PRIVATE Defecto : Lampara;
  PRIVATE Rearme : Pulsador;
  PRIVATE Marcha : Pulsador;
  PRIVATE Paro : Pulsador;
  PRIVATE Modo : SelectorDual;
  PRIVATE Medida : SelectorDual;
  PUBLIC METHOD EncenderLampara ( ) : void;
  PUBLIC METHOD ApagarLampara ( ) : void;
  PUBLIC METHOD HayParo ( ) : BOOL;
  PUBLIC METHOD HayRearme ( ) : BOOL;
  PUBLIC METHOD HayModoManual ( ) : BOOL;
  PUBLIC METHOD HayModoAutomatico ( ) : BOOL;
  PUBLIC METHOD HayAlto ( ) : BOOL;
  PUBLIC METHOD HayBajo ( ) : BOOL;
  PUBLIC METHOD HayMarcha ( ) : BOOL;
  PUBLIC METHOD GetMarcha ( ) : Pulsador;
END_CLASS
```

Algoritmo D.23 Método *"EncenderLampara"* de la clase *"PanelMando"*

```
METHOD PanelMando::EncenderLampara ( )
   defecto.Encender(true);
END_METHOD
```

de ejecución solicitado por el operario (manual o automático), los que comprueban el tamaño del rodamiento (alto o bajo) y los que comprueban si la estación está en marcha. En el algoritmo D.22 se muestra la estructura de la clase y en la figura D.80 la estructura que implementa la clase.

Método EncenderLampara

Este método pone a true el atributo privado *"encendida"* de la clase agregada *"defecto"* de tipo *"Lampara"* por medio de la invocación del método de dicha clase *"Encender"* (ver algoritmo D.23). En la figura D.81 se muestra la traducción a código IL del método.

```
TYPE PanelMando:
    STRUCT
        defecto : Lampara;
        rearme : Pulsador;
        marcha : Pulsador;
        paro : Pulsador;
        modo : SelectorDual;
        medida : SelectorDual;
    END_STRUCT;
END_TYPE
```

Figura D.80: Traducción de la clase *"PanelMando"* a IL

```
FUNCTION_BLOCK PanelMando_EncenderLampara
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR
VAR (* Variables locales *)
        Lampara_Encender : Lampara_Encender;
END_VAR

(* Llamada a metodo*)
CAL Lampara_Encender(_THIS := _THIS.defecto)
END_FUNCTION_BLOCK
```

Figura D.81: Traducción del método *"EncenderLampara"* de la clase *"PanelMando"* a IL

Algoritmo D.24 Método *"ApagarLampara"* de la clase *"PanelMando"*

```
METHOD PanelMando::ApagarLampara ( )
   defecto.Apagar(true);
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_ApagarLampara
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR
VAR (* Variables locales *)
        Lampara_Apagar : Lampara_Apagar;
END_VAR

(* Llamada a metodo*)
CAL Lampara_Apagar(_THIS := _THIS.defecto)
END_FUNCTION_BLOCK
```

Figura D.82: Traducción del método *"ApagarLampara"* de la clase *"PanelMando"* a IL

Método ApagarLampara

Este método pone a false el atributo privado *"encendida"* de la clase agregada *"defecto"* de tipo *"Lampara"* por medio de la invocación del método de dicha clase *"Apagar"* (ver algoritmo D.24). En la figura D.82 se muestra la traducción a código IL del método.

Método HayParo

Este método devuelve el valor del atributo privado *"accionado"* de la clase *"paro"* de tipo *"Pulsador"* (ver algoritmo D.25). En la figura D.83 se muestra la traducción a código IL del método.

Algoritmo D.25 Método *"HayParo"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayParo ( )
   HayParo:=paro.accionado;
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_HayParo
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.paro.accionado
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.83: Traducción del método *"HayParo"* de la clase *"PanelMando"* a IL

Algoritmo D.26 Método *"HayRearme"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayRearme ( )
  HayRearme:=rearme.accionado;
END_METHOD
```

Método HayRearme

Este método devuelve el valor del atributo privado *"accionado"* de la clase *"rearme"* de tipo *"Pulsador"* (ver algoritmo D.26). En la figura D.84 se muestra la traducción a código IL del método.

Método HayModoManual

Este método devuelve el valor del atributo privado *"valor1"* de la clase *"modo"* de tipo *"SensorDual"* (ver algoritmo D.27). En la figura D.85 se muestra la traducción a código IL del método.

Algoritmo D.27 Método *"HayModoManual"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayModoManual ( )
  HayModoManual:=modo.valor1;
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_HayRearme
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.rearme.accionado
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.84: Traducción del método *"HayRearme"* de la clase *"PanelMando"* a IL

```
FUNCTION_BLOCK PanelMando_HayModoManual
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.modo.valor1
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.85: Traducción del método *"HayModoManual"* de la clase *"PanelMando"* a IL

Algoritmo D.28 Método *"HayModoAutomatico"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayModoAutomatico ( )
  HayModoAutomatico:=modo.valor2;
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_HayModoAutomatico
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.modo.valor2
ST _Retorno
END FUNCTION BLOCK
```

Figura D.86: Traducción del método *"HayModoAutomatico"* de la clase *"PanelMando"* a IL

Método HayModoAutomatico

Este método devuelve el valor del atributo privado *"valor2"* de la clase *"modo"* de tipo *"SensorDual"* (ver algoritmo D.28). En la figura D.86 se muestra la traducción a código IL del método.

Método HayAlto

Este método devuelve el valor del atributo privado *"valor2"* de la clase *"medida"* de tipo *"SensorDual"* (ver algoritmo D.29). En la figura D.87 se muestra la traducción a código IL del método.

Algoritmo D.29 Método *"HayAlto"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayAlto ( )
  HayAlto:=medida.valor2;
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_HayAlto
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.medida.valor2
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.87: Traducción del método *"HayAlto"* de la clase *"PanelMando"* a IL

Algoritmo D.30 Método *"HayBajo"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayBajo ( )
  HayBajo:=medida.valor2;
END_METHOD
```

Método HayBajo

Este método devuelve el valor del atributo privado *"valor1"* de la clase *"medida"* de tipo *"SensorDual"* (ver algoritmo D.30). En la figura D.88 se muestra la traducción a código IL del método.

Método HayMarcha

Este método devuelve el valor del atributo privado *"accionado"* de la clase *"marcha"* de tipo *"Pulsador""* (ver algoritmo D.31). En la figura D.89 se muestra la traducción a código IL del método.

Algoritmo D.31 Método *"HayMarcha"* de la clase *"PanelMando"*

```
METHOD PanelMando::HayMarcha ( )
  HayMarcha:=marcha.accionado;
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_HayBajo
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.medida.valor1
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.88: Traducción del método *"HayBajo"* de la clase *"PanelMando"* a IL

```
FUNCTION_BLOCK PanelMando_HayMarcha
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.marcha.accionado
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.89: Traducción del método *"HayMarcha"* de la clase *"PanelMando"* a IL

Algoritmo D.32 Método *"GetMarcha"* de la clase *"PanelMando"*

```
METHOD PanelMando::GetMarcha ( )
   GetMarcha:=marcha;
END_METHOD
```

```
FUNCTION_BLOCK PanelMando_GetMarcha
VAR_OUTPUT
        _Retorno : Pulsador;
END_VAR
VAR_IN_OUT
        _THIS : PanelMando;
END_VAR

VAR (* Variables locales *)
END_VAR
LD _THIS.marcha
ST _Retorno
END_FUNCTION_BLOCK
```

Figura D.90: Traducción del método *"GetMarcha"* de la clase *"PanelMando"* a IL

Método GetMarcha

Este método devuelve una referencia a la clase *"marcha"* de tipo *"Pulsador""* (ver algoritmo D.32). En la figura D.90 se muestra la traducción a código IL del método.

D.7.14. Clase ControlProcesoEstacion2

Esta clase engloba toda la lógica de negocio de la estación 2 tanto para los procesos automáticos como manuales. Se trata de una clase elaborada que hereda de *"ObjetoComplejoBase"*. En el algoritmo D.33 se muestra la estructura de la clase y en la figura D.91 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.34). En la figura D.92 se muestra la traducción a código IL del método constructor.

Algoritmo D.33 Clase *"ControlProcesoEstacion2"*

```
CLASS ControlProcesoEstacion2 (ObjetoComplejoBase)
  PRIVATE Alimentador : Alimentador;
  PRIVATE Transvasador : Transvasador;
  PRIVATE Medidor : Medidor;
  PRIVATE medida : INT;
  PRIVATE Evacuador : Evacuador;
  PUBLIC METHOD InicializarElementos ( ) : void;
  PUBLIC METHOD Manual ( ) : void;
  PUBLIC METHOD Automatico ( ) : void;
END_CLASS
```

```
TYPE ControlProcesoEstacion2:
   STRUCT
      vPointer : *VOID;
      Alimentador : Alimentador;
      Transvasador : Transvasador;
      Medidor : Medidor;
      Evacuador : Evacuador;
      _ObjetoComplejoBase : ObjetoComplejoBase;
   END_STRUCT;
END_TYPE
```

Figura D.91: Traducción de la clase *"ControlProcesoEstacion2"* a IL

Algoritmo D.34 Constructor de la clase "*ControlProcesoEstacion2*"

```
METHOD ControlProcesoEstacion2::
       ControlProcesoEstacion2 ( )
  THIS.vPointer:=&vPointer+40;
END_METHOD
```

```
FUNCTION_BLOCK ControlProcesoEstacion2_ControlProcesoEstacion2
VAR_IN_OUT
        _This : ControlProcesoEstacion2;
END_VAR
VAR
      aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 40
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.92: Traducción del a IL del constructor de la clase "*ControlProcesoEstacion2*"

Método InicializarElementos

Este método coloca todos los elementos del objeto en su posición inicial (ver figura D.93). En las figuras D.94 y D.95 se muestra la traducción a código IL del método.

Método Manual (ejecución manual)

Este método ejecuta de forma manual el proceso de producción de la estación de montaje de rodamientos (ver figura D.96). En las figuras D.97, D.98 y D.99 se muestra la traducción del método código IL.

Método Automatico (ejecución automática)

Este método ejecuta de forma manual el proceso de producción de la estación de montaje de rodamientos (ver figura D.100). En las figuras D.97, D.102 y D.103 se muestra la traducción del método código IL.

D.7.15. Interfaz EstacionBase

Este interfaz se corresponde con un tipo de estación básico del que heredan el resto de estaciones. Posee un único método virtual *"Run"* que sirve para hacer la ligadura dinámica desde el *"transfer"*.

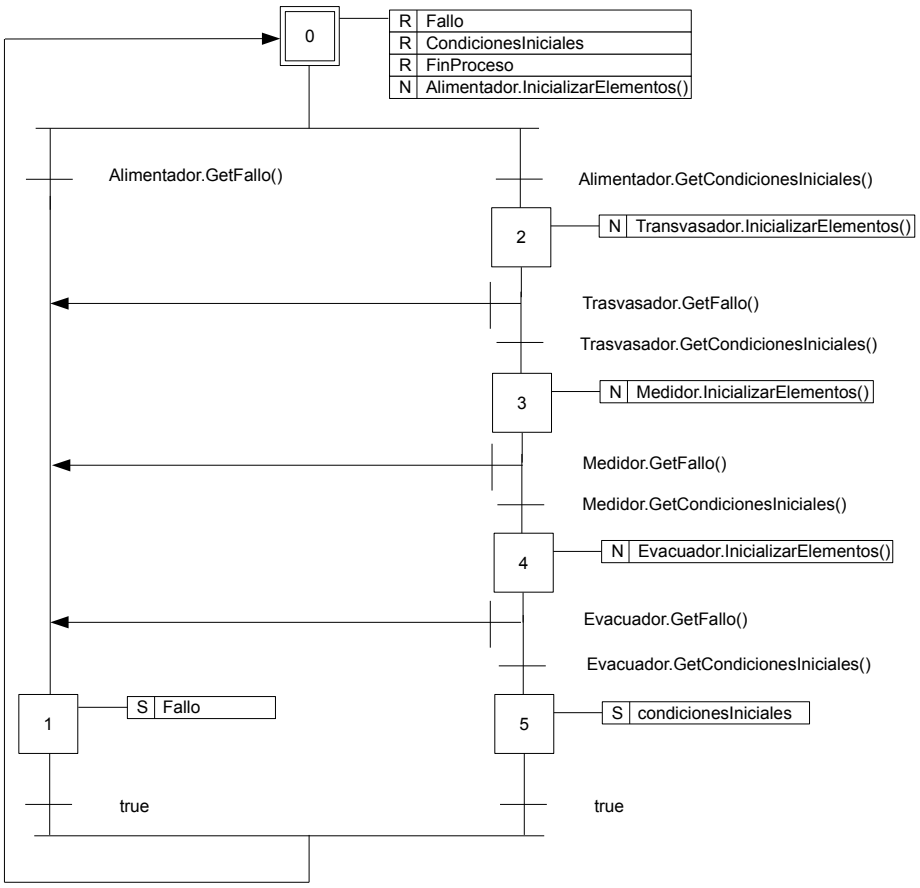

Figura D.93: Método *"InicializarElementos"* de la clase *"ControlProceso"*

```
FUNCTION_BLOCK ControlProceso_InicializarElementos
VAR_IN_OUT
        _THIS : ControlProceso;
END_VAR

VAR (* Variables locales *)
        _AUXT_0 : BOOL := TRUE;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E3 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E6 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        CI_Alimentador : BOOL := FALSE;
        CI_Transvasador : BOOL := FALSE;
        CI_Medidor : BOOL := FALSE;
        CI_Evacuador : BOOL := FALSE;
        Alimentador_GetFallo : Alimentador_GetFallo;
        Alimentador_GetCondicionesIniciales : Alimentador_GetCondicionesIniciales;
        Evacuador_GetFallo : Evacuador_GetFallo;
        Medidor_GetFallo : Medidor_GetFallo;
        Trasvasador_GetFallo : Trasvasador_GetFallo;
        Trasvasador_GetCondicionesIniciales : Trasvasador_GetCondicionesIniciales;
        Medidor_GetCondicionesIniciales : Medidor_GetCondicionesIniciales;
        Evacuador_GetCondicionesIniciales : Evacuador_GetCondicionesIniciales;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E3.X
ST E3.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E6.X
ST E6.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E1.C
AND FALSE
OR ( E8.C
AND TRUE
)
S E0.X
R E1.X
R E8.X
(* Llamada a metodo*)
CAL Alimentador_GetFallo(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Alimentador_GetCondicionesIniciales(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Trasvasador_GetFallo(_THIS := _THIS.transvasador)
LD Alimentador_GetFallo._Retorno
ANDN Alimentador_GetCondicionesIniciales._Retorno
AND E0.C
OR ( E4.C
AND Evacuador_GetFallo._Retorno
)
OR ( E3.C
AND Medidor_GetFallo._Retorno
)
OR ( E2.C
AND Trasvasador_GetFallo._Retorno
)
S E1.X
R E0.X
R E4.X
R E3.X
R E2.X
(* Llamada a metodo*)
CAL Alimentador_GetCondicionesIniciales(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Alimentador_GetFallo(_THIS := _THIS.alimentador)
```

Figura D.94: Traducción del a IL del método *"InicializarElementos"* de la clase *"ControlProceso"* - parte 1

```
LD Alimentador_GetCondicionesIniciales._Retorno
ANDN Alimentador_GetFallo._Retorno
AND E0.C
S E2.X
R E0.X
(* Llamada a metodo*)
CAL Trasvasador_GetCondicionesIniciales(_THIS := _THIS.transvasador)
(* Llamada a metodo*)
CAL Trasvasador_GetFallo(_THIS := _THIS.transvasador)
LD Trasvasador_GetCondicionesIniciales._Retorno
ANDN Trasvasador_GetFallo._Retorno
AND E2.C
S E3.X
R E2.X
(* Llamada a metodo*)
CAL Medidor_GetCondicionesIniciales(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD Medidor_GetCondicionesIniciales._Retorno
ANDN Medidor_GetFallo._Retorno
AND E3.C
S E4.X
R E3.X
(* Llamada a metodo*)
CAL Evacuador_GetCondicionesIniciales(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
LD Evacuador_GetCondicionesIniciales._Retorno
ANDN Evacuador_GetFallo._Retorno
AND E4.C
S E5.X
R E4.X
LD E5.C
AND TRUE
S E6.X
R E5.X
LD E6.C
AND TRUE
S E7.X
R E6.X
LD E7.C
AND TRUE
S E8.X
R E7.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LDN CI_Alimentador
JMPC _SALTO_0
_SALTO_0:
LD E1.X
S _THIS.ObjetoComplejoBase.fallo
LDN CI_Transvasador
JMPC _SALTO_1
_SALTO_1:
LDN CI_Medidor
JMPC _SALTO_2
_SALTO_2:
LDN CI_Evacuador
JMPC _SALTO_3
_SALTO_3:
LD E5.X
S _THIS.ObjetoComplejoBase.condicionesIniciales
LD E0.X
ST CI_Alimentador
LD E4.X
ST CI_Evacuador
LD E3.X
ST CI_Medidor
LD E2.X
ST CI_Transvasador
END_FUNCTION_BLOCK
```

Figura D.95: Traducción del a IL del método *"InicializarElementos"* de la clase *"ControlProceso"* - parte 2

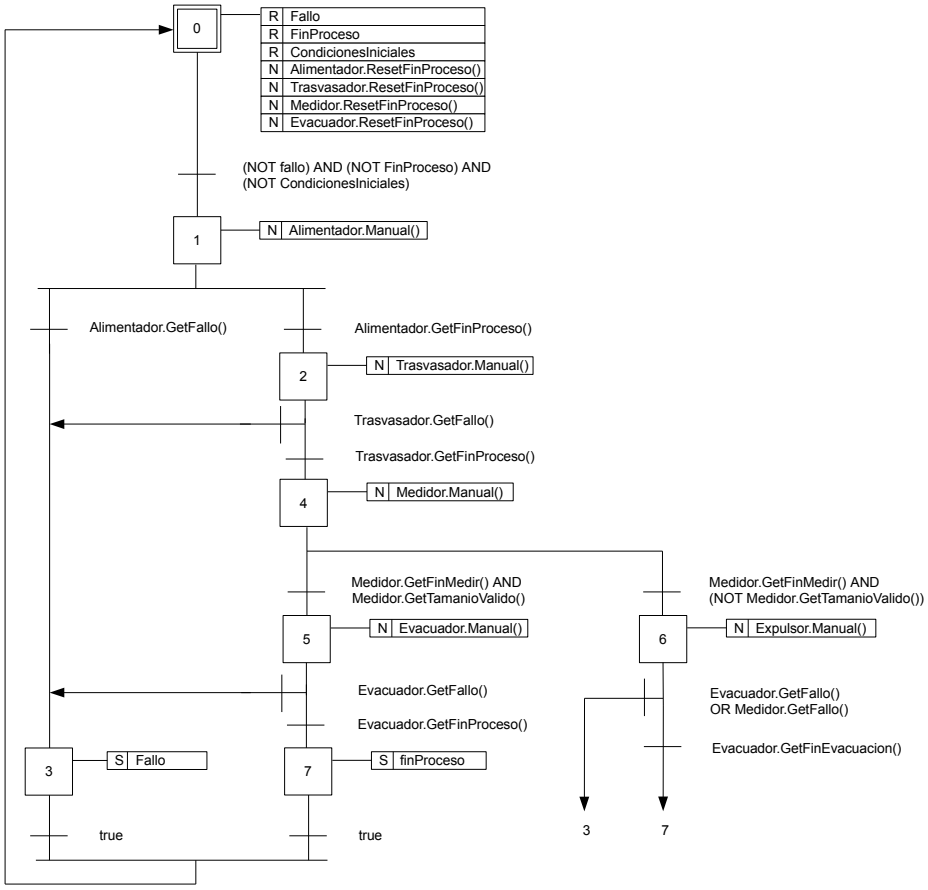

Figura D.96: Método *"Manual"* de la clase *"ControlProceso"*

```
FUNCTION_BLOCK ControlProceso_Manual
VAR_IN_OUT
        _THIS : ControlProceso;
        marcha : Pulsador;
END_VAR
VAR_INPUT
        medida : INT;
END_VAR

VAR (* Variables locales *)
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUXT_9 : BOOL;
        _AUX_10 : BOOL;
        _AUX_11 : BOOL;
        _AUXT_12 : BOOL;
        _AUX_13 : BOOL;
        _AUXT_14 : BOOL;
        _AUX_15 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E3 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E7 : TSFCTYPE;
        E8 : TSFCTYPE;
        E6 : TSFCTYPE;
        E9 : TSFCTYPE;
        E10 : TSFCTYPE;
        E11 : TSFCTYPE;
        reset : BOOL := FALSE;
        alimentacion : BOOL := FALSE;
        transvase : BOOL := FALSE;
        medicion : BOOL := FALSE;
        evacuacion : BOOL := FALSE;
        expulsion : BOOL := FALSE;
        Medidor_GetFinProceso : Medidor_GetFinProceso;
        Medidor_GetTamanioValido : Medidor_GetTamanioValido;
        Evacuador_GetFallo : Evacuador_GetFallo;
        Medidor_GetFallo : Medidor_GetFallo;
        Alimentador_GetFallo : Alimentador_GetFallo;
        Alimentador_GetFinProceso : Alimentador_GetFinProceso;
        Trasvasador_GetFallo : Trasvasador_GetFallo;
        Trasvasador_GetFinProceso : Trasvasador_GetFinProceso;
        Evacuador_GetFinEvacuacion : Evacuador_GetFinEvacuacion;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN _THIS.ObjetoComplejoBase.fallo
ST _AUX_2
LDN _THIS.ObjetoComplejoBase.finProceso
AND _AUX_2
ST _AUX_4
LDN _THIS.ObjetoComplejoBase.condicionesIniciales
AND _AUX_4
ST _AUXT_1
(* Llamada a metodo*)
CAL Medidor_GetFinProceso(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Medidor_GetTamanioValido(_THIS := _THIS.medidor)
LD Medidor_GetFinProceso._Retorno
AND Medidor_GetTamanioValido._Retorno
ST _AUXT_7
(* Llamada a metodo*)
CAL Medidor_GetTamanioValido(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Medidor_GetFinProceso(_THIS := _THIS.medidor)
LDN Medidor_GetTamanioValido._Retorno
AND Medidor_GetFinProceso._Retorno
ST _AUXT_9
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD Evacuador_GetFallo._Retorno
OR Medidor_GetFallo._Retorno
ST _AUXT_12
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
```

Figura D.97: Traducción del a IL del método *"Manual"* de la clase *"ControlProceso"* - parte 1

```
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD Evacuador_GetFallo._Retorno
OR Medidor_GetFallo._Retorno
ST _AUXT_14
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E3.X
ST E3.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E7.X
ST E7.C
LD E8.X
ST E8.C
LD E6.X
ST E6.C
LD E9.X
ST E9.C
LD E10.X
ST E10.C
LD E11.X
ST E11.C
LD E3.C
AND FALSE
OR ( E11.C
AND TRUE
)
S E0.X
R E3.X
R E11.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
(* Llamada a metodo*)
CAL Alimentador_GetFallo(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Alimentador_GetFinAlimentacion(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Trasvasador_GetFallo(_THIS := _THIS.transvasador)
LD Alimentador_GetFallo._Retorno
ANDN Alimentador_GetFinAlimentacion._Retorno
AND E1.C
OR ( E5.C
AND Evacuador_GetFallo._Retorno
)
OR ( E6.C
AND _AUXT_14
)
OR ( E4.C
AND Medidor_GetFallo._Retorno
)
OR ( E2.C
AND Trasvasador_GetFallo._Retorno
)
S E3.X
R E1.X
R E5.X
R E6.X
R E4.X
R E2.X
(* Llamada a metodo*)
CAL Alimentador_GetFinAlimentacion(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Alimentador_GetFallo(_THIS := _THIS.alimentador)
LD Alimentador_GetFinAlimentacion._Retorno
ANDN Alimentador_GetFallo._Retorno
AND E1.C
S E2.X
R E1.X
(* Llamada a metodo*)
CAL Transvasador_GetFinTransvasar(_THIS := _THIS.transvasador)
(* Llamada a metodo*)
CAL Trasvasador_GetFallo(_THIS := _THIS.transvasador)
LD Transvasador_GetFinTransvasar._Retorno
ANDN Trasvasador_GetFallo._Retorno
AND E2.C
S E4.X
R E2.X
(* Llamada a metodo*)
```

Figura D.98: Traducción del a IL del método *"Manual"* de la clase *"ControlProceso"* - parte 2

```
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD _AUXT_7
ANDN _AUXT_9
ANDN Medidor_GetFallo._Retorno
AND E4.C
S E5.X
R E4.X
(* Llamada a metodo*)
CAL Evacuador_GetFinEvacuacion(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Evacuador_GetFinEvacuacion(_THIS := _THIS.evacuador)
LD Evacuador_GetFinEvacuacion._Retorno
ANDN Evacuador_GetFallo._Retorno
AND E5.C
OR ( E6.C
AND Evacuador_GetFinEvacuacion._Retorno
)
S E7.X
R E5.X
R E6.X
LD E7.C
AND FALSE
S E8.X
R E7.X
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD _AUXT_9
ANDN _AUXT_7
ANDN Medidor_GetFallo._Retorno
AND E4.C
S E6.X
R E4.X
S E9.X
LD E8.C
AND TRUE
OR ( E9.C
AND TRUE
)
S E10.X
R E8.X
R E9.X
LD E10.C
AND TRUE
S E11.X
R E10.X
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LDN reset
JMPC _SALTO_0
_SALTO_0:
LDN alimentacion
JMPC _SALTO_1
_SALTO_1:
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
LDN transvase
JMPC _SALTO_2
_SALTO_2:
LDN medicion
JMPC _SALTO_3
_SALTO_3:
LDN evacuacion
JMPC _SALTO_4
_SALTO_4:
LD E7.X
S _THIS.ObjetoComplejoBase.finProceso
LDN expulsion
JMPC _SALTO_5
_SALTO_5:
LD E1.X
ST alimentacion
LD E5.X
ST evacuacion
LD E6.X
ST expulsion
LD E4.X
ST medicion
LD E0.X
ST reset
LD E2.X
ST transvase
END_FUNCTION_BLOCK
```

Figura D.99: Traducción del a IL del método *"Manual"* de la clase *"ControlProceso"* - parte 3

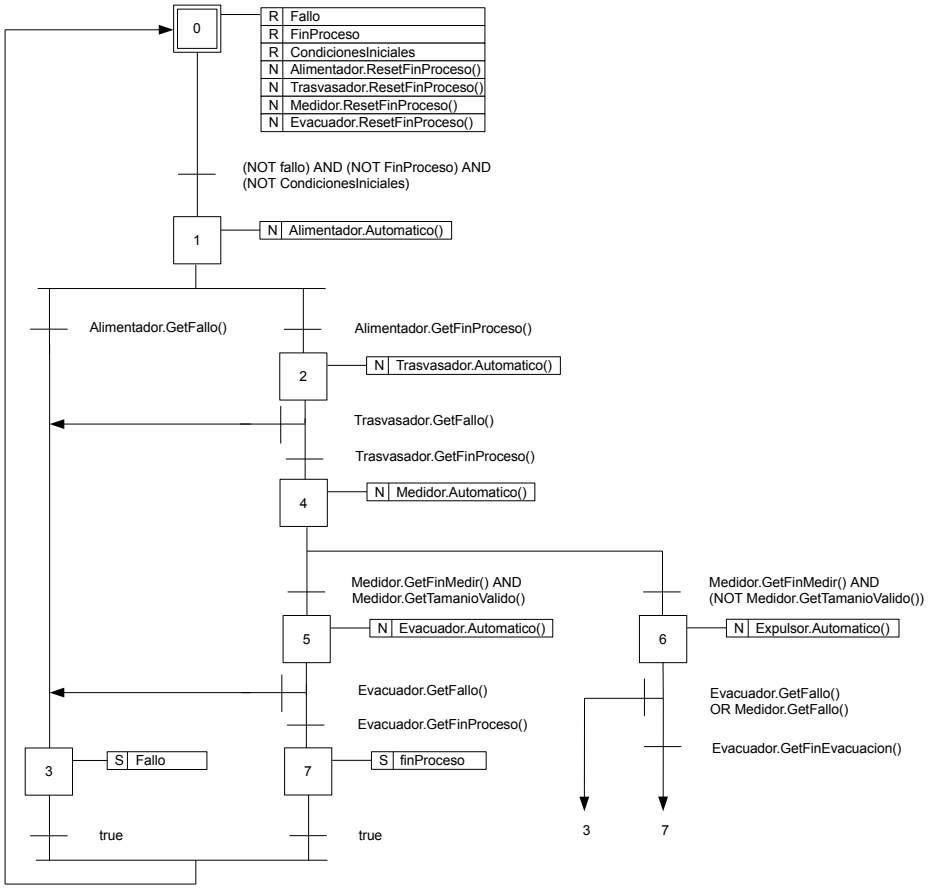

Figura D.100: Método *"Automatico"* de la clase *"ControlProceso"*

```
FUNCTION_BLOCK ControlProceso_Automatico
VAR_IN_OUT
        _THIS : ControlProceso;
END_VAR
VAR_INPUT
        medida : INT;
END_VAR

VAR (* Variables locales *)
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUX_4 : BOOL;
        _AUX_5 : BOOL;
        _AUX_6 : BOOL;
        _AUXT_7 : BOOL;
        _AUX_8 : BOOL;
        _AUXT_9 : BOOL;
        _AUX_10 : BOOL;
        _AUX_11 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E3 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E5 : TSFCTYPE;
        E7 : TSFCTYPE;
        E6 : TSFCTYPE;
        E8 : TSFCTYPE;
        E9 : TSFCTYPE;
        E10 : TSFCTYPE;
        E11 : TSFCTYPE;
        reset : BOOL := FALSE;
        alimentacion : BOOL := FALSE;
        transvase : BOOL := FALSE;
        medicion : BOOL := FALSE;
        evacuacion : BOOL := FALSE;
        expulsion : BOOL := FALSE;
        Medidor_GetFinProceso : Medidor_GetProceso;
        Medidor_GetTamanioValido : Medidor_GetTamanioValido;
        Alimentador_GetFallo : Alimentador_GetFallo;
        Alimentador_GetFinAlimentacion : Alimentador_GetFinAlimentacion;
        Evacuador_GetFallo : Evacuador_GetFallo;
        Medidor_GetFallo : Medidor_GetFallo;
        Transvasador_GetFallo : Transvasador_GetFallo;
        Transvasador_GetFinProceso : Transvasador_GetFinProceso;
        Evacuador_GetFinProceso : Evacuador_GetFinProceso;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
LDN _THIS.ObjetoComplejoBase.fallo
ST _AUX_2
LDN _THIS.ObjetoComplejoBase.finProceso
AND _AUX_2
ST _AUX_4
LDN _THIS.ObjetoComplejoBase.condicionesIniciales
AND _AUX_4
ST _AUXT_1
(* Llamada a metodo*)
CAL Medidor_GetFinProceso(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Medidor_GetTamanioValido(_THIS := _THIS.medidor)
LD Medidor_GetFinProceso._Retorno
AND Medidor_GetTamanioValido._Retorno
ST _AUXT_7
(* Llamada a metodo*)
CAL Medidor_GetTamanioValido(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Medidor_GetFinProceso(_THIS := _THIS.medidor)
LDN Medidor_GetTamanioValido._Retorno
AND Medidor_GetFinProceso._Retorno
ST _AUXT_9
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E3.X
ST E3.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E5.X
ST E5.C
LD E7.X
ST E7.C
LD E6.X
```

Figura D.101: Traducción del a IL del método *"Automatico"* de la clase *"Control-Proceso"* - parte 1

```
ST E6.C
LD E8.X
ST E8.C
LD E9.X
ST E9.C
LD E10.X
ST E10.C
LD E11.X
ST E11.C
LD _THIS.ObjetoComplejoBase.finProceso
AND E11.C
S E0.X
R E11.X
LD _AUXT_1
AND E0.C
S E1.X
R E0.X
(* Llamada a metodo*)
CAL Alimentador_GetFallo(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Alimentador_GetFinAlimentacion(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Evacuador_GetFallo(_THIS := _THIS.evacuador)
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
(* Llamada a metodo*)
CAL Transvasador_GetFallo(_THIS := _THIS.transvasador)
LD Alimentador_GetFallo._Retorno
ANDN Alimentador_GetFinAlimentacion._Retorno
AND E1.C
OR ( E5.C
AND Evacuador_GetFallo._Retorno
)
OR ( E6.C
AND Evacuador_GetFallo._Retorno
)
OR ( E4.C
AND Medidor_GetFallo._Retorno
)
OR ( E2.C
AND Transvasador_GetFallo._Retorno
)
S E3.X
R E1.X
R E5.X
R E6.X
R E4.X
R E2.X
(* Llamada a metodo*)
CAL Alimentador_GetFinAlimentacion(_THIS := _THIS.alimentador)
(* Llamada a metodo*)
CAL Alimentador_GetFallo(_THIS := _THIS.alimentador)
LD Alimentador_GetFinAlimentacion._Retorno
ANDN Alimentador_GetFallo._Retorno
AND E1.C
S E2.X
R E1.X
(* Llamada a metodo*)
CAL Transvasador_GetFinTransvasar(_THIS := _THIS.transvasador)
(* Llamada a metodo*)
CAL Transvasador_GetFallo(_THIS := _THIS.transvasador)
LD Transvasador_GetFinTransvasar._Retorno
ANDN Transvasador_GetFallo._Retorno
AND E2.C
S E4.X
R E2.X
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD _AUXT_7
ANDN _AUXT_9
ANDN Medidor_GetFallo._Retorno
AND E4.C
S E5.X
R E4.X
(* Llamada a metodo*)
CAL Evacuador_GetFinEvacuacion(_THIS := _THIS.evacuador)
LD E5.C
AND Evacuador_GetFinEvacuacion._Retorno
S E7.X
(* Llamada a metodo*)
CAL Medidor_GetFallo(_THIS := _THIS.medidor)
LD _AUXT_9
ANDN _AUXT_7
ANDN Medidor_GetFallo._Retorno
AND E4.C
S E6.X
R E4.X
(* Llamada a metodo*)
```

Figura D.102: Traducción del a IL del método *"Automatico"* de la clase *"Control-Proceso"* - parte 2

```
CAL Evacuador_GetFinEvacuacion(_THIS := _THIS.evacuador)
LD E6.C
AND Evacuador_GetFinEvacuacion._Retorno
S E8.X
LD E7.C
AND TRUE
OR ( E8.C
AND TRUE
)
S E9.X
R E7.X
R E8.X
LD E9.C
AND TRUE
S E10.X
R E9.X
LD E3.C
AND FALSE
OR ( E10.C
AND TRUE
)
S E11.X
R E3.X
R E10.X
LD E0.X
R _THIS.ObjetoComplejoBase.finProceso
LD E0.X
R _THIS.ObjetoComplejoBase.fallo
LD E0.X
R _THIS.ObjetoComplejoBase.condicionesIniciales
LDN reset
JMPC _SALTO_0
_SALTO_0:
LDN alimentacion
JMPC _SALTO_1
_SALTO_1:
LD E3.X
S _THIS.ObjetoComplejoBase.fallo
LDN transvase
JMPC _SALTO_2
_SALTO_2:
LDN medicion
JMPC _SALTO_3
_SALTO_3:
LDN evacuacion
JMPC _SALTO_4
_SALTO_4:
LDN expulsion
JMPC _SALTO_5
_SALTO_5:
LD E11.X
S _THIS.ObjetoComplejoBase.finProceso
LD E1.X
ST alimentacion
LD E5.X
ST evacuacion
LD E6.X
ST expulsion
LD E4.X
ST medicion
LD E0.X
ST reset
LD E2.X
ST transvase
END_FUNCTION_BLOCK
```

Figura D.103: Traducción del a IL del método *"Automatico"* de la clase *"Control-Proceso"* - parte 3

Algoritmo D.35 Interfaz *"EstacionBase"*

```
INTERFAZ EstacionBase ( )
  METHOD Run ( ) : void;
  METHOD GetCondicionesIniciales ( ) : BOOL;
  METHOD ResetCondicionesIniciales ( ) : BOOL;
  METHOD GetFinProceso ( ) : BOOL;
  METHOD ResetFinProceso ( ) : BOOL;
END_INTERFAZ
```

```
TYPE EstacionBase:
  STRUCT
    vPointer : *VOID;
  END_STRUCT;
END_TYPE
```

Figura D.104: Traducción del interfaz "*EstacionBase"* a IL

En el algoritmo D.35 se muestra la estructura de la clase y en la figura D.104 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.36). En la figura D.105 se muestra la traducción a código IL del método constructor.

D.7.16. Clase Estacion2

Se trata de una abstracción de toda la estación de ensamblaje de rodamientos y garantiza los distintos modos de funcionamiento según GEMMA (ver apartado D.4). Posee dos atributos privados que se corresponden con el panel de mandos

Algoritmo D.36 Constructor del interfaz "*EstacionBase*"

```
METHOD EstacionBase::EstacionBase ( )
  THIS.vPointer:=&vPointer+24;
END_METHOD
```

```
FUNCTION_BLOCK EstacionBase_EstacionBase
VAR_IN_OUT
        _This : EstacionBase;
END_VAR
VAR
      aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 24
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.105: Traducción del a IL del constructor del interfaz "*EstacionBase*"

Algoritmo D.37 Clase *"Estacion2"*

```
CLASS Estacion2 (EstacionBase)
  PRIVATE Proceso : ControlProcesoEstacion2;
  PRIVATE Botonera : PanelMando;
  PUBLIC METHOD Run ( ) : void;
  PUBLIC METHOD GetCondicionesIniciales ( ) : BOOL;
  PUBLIC METHOD ResetCondicionesIniciales ( ) : BOOL;
  PUBLIC METHOD GetFinProceso ( ) : BOOL;
  PUBLIC METHOD ResetFinProceso ( ) : BOOL;
  PUBLIC METHOD InicializarElementos ( ) : void;
END_CLASS
```

(clase *"PanelMando"*) y con la lógica de funcionamiento propiamente dicha de la estación (clase *"ControlProceso"*). Respecto a los servicios que ofrece, esta clase sólo proporciona un único método virtual *"Run"* que es común a todas las estaciones y que será invocado por el transfer. En el algoritmo D.37 se muestra la estructura de la clase y en la figura D.106 la estructura que implementa la clase.

Constructor

El constructor es generado en tiempo de compilación e inicializa el puntero *"vPointer"* (ver algoritmo D.38). En la figura D.107 se muestra la traducción a código IL del método constructor.

```
TYPE Estacion2:
   STRUCT
      vPointer : *VOID;
      Proceso : ControlProcesoEstacion2;
      Panel : PanelMando;
      _EstacionBase : EstacionBase;
   END_STRUCT;
END_TYPE
```

Figura D.106: Traducción de la clase *"Estacion2"* a IL

Algoritmo D.38 Constructor de la clase "*Estacion2*"

```
METHOD Estacion2::Estacion2 ( )
  THIS.vPointer:=&vPointer+43;
END_METHOD
```

```
FUNCTION_BLOCK Estacion2_Estacion2
VAR_IN_OUT
        _This : Estacion2;
END_VAR
VAR
      aux_vPointer : *VOID;
END_VAR

LD vPointer
ADD 43
ST aux_vPointer
MOVE @THIS.vPointer, aux_vPointer
END_FUNCTION_BLOCK
```

Figura D.107: Traducción a IL del constructor de la clase "*Estacion2*"

Algoritmo D.39 Métodos varios de la clase "Estacion2"

```
METHOD Estacion2::GetFallo ( )
  RETURN encendida;
END_METHOD

METHOD Estacion2::ResetFallo ( )
   fallo:=false;
END_METHOD

METHOD Estacion2::GetCondicionesIniciales ( )
  RETURN condicionesIniciales;
END_METHOD

METHOD Estacion2::ResetCondicionesIniciales ( )
   condicionesIniciales:=false;
END_METHOD

METHOD Estacion2::GetFinProceso ( )
  RETURN finProceso;
END_METHOD

METHOD Estacion2::ResetFinProceso ( )
   finProceso:=false;
END_METHOD
```

Método Run

Este método virtual desencadena la ejecución de la estación de acuerdo a los distintos modos de funcionamiento que esta soporta. En la figura D.108 se puede ver el código del método. En la figuras D.109, D.110 y D.111 se muestra la traducción a código IL del método run.

Métodos varios de la clase Estacion2

En la figura D.112 se muestra la traducción de todos los métodos (menos del constructor y del método *"Run"*) de la clase *"Estacion2"* (ver figura D.112). Los métodos get devuelven el valor del atributo al que hacen referencia y los métodos reset ponen a false dichos atributos.

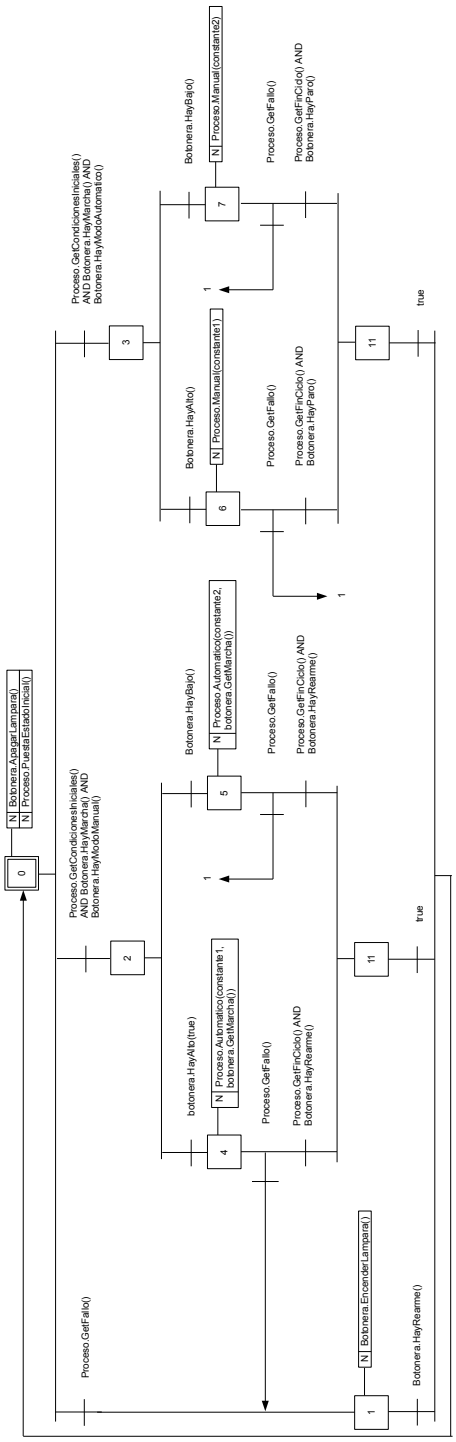

Figura D.108: Método "*Run*" de la clase "*Estacion2*"

```
FUNCTION_BLOCK Estacion2_Run
VAR_IN_OUT
        _THIS : Estacion2;
END_VAR
VAR (* Variables locales *)
        alturaRodamiento : INT := 0;
        constante1 : INT := 0;
        constante2 : INT := 1;
        _AUXT_0 : BOOL := TRUE;
        _AUXT_1 : BOOL;
        _AUX_2 : BOOL;
        _AUX_3 : BOOL;
        _AUXT_4 : BOOL;
        _AUX_5 : BOOL;
        _AUXT_6 : BOOL;
        _AUX_7 : BOOL;
        _AUXT_8 : BOOL;
        _AUX_9 : BOOL;
        _AUX_10 : BOOL;
        _AUXT_11 : BOOL;
        _AUX_12 : BOOL;
        _AUXT_13 : BOOL;
        _AUX_14 : BOOL;
        E0 : TSFCTYPE;
        E1 : TSFCTYPE;
        E2 : TSFCTYPE;
        E4 : TSFCTYPE;
        E8 : TSFCTYPE;
        E5 : TSFCTYPE;
        E9 : TSFCTYPE;
        E12 : TSFCTYPE;
        E3 : TSFCTYPE;
        E6 : TSFCTYPE;
        E10 : TSFCTYPE;
        E7 : TSFCTYPE;
        E11 : TSFCTYPE;
        E13 : TSFCTYPE;
        A6 : BOOL := FALSE;
        EncenderLampara : BOOL := FALSE;
        F5_Alto : BOOL := FALSE;
        F5_Bajo : BOOL := FALSE;
        F1_Alto : BOOL := FALSE;
        F1_Bajo : BOOL := FALSE;
        ControlProceso_GetCondicionesIniciales : ControlProceso_GetCondicionesIniciales;
        PanelMando_HayMarcha : PanelMando_HayMarcha;
        PanelMando_HayModoManual : PanelMando_HayModoManual;
        ControlProceso_GetFinCiclo : ControlProceso_GetFinCiclo;
        PanelMando_HayRearme : PanelMando_HayRearme;
        PanelMando_HayModoAutomatico : PanelMando_HayModoAutomatico;
        PanelMando_HayParo : PanelMando_HayParo;
        ControlProceso_GetFallo : ControlProceso_GetFallo;
        PanelMando_HayAlto : PanelMando_HayAlto;
        PanelMando_HayBajo : PanelMando_HayBajo;
END_VAR
LD _AUXT_0
S E0.X
LD FALSE
ST _AUXT_0
(* Llamada a metodo*)
CAL ControlProceso_GetCondicionesIniciales(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL PanelMando_HayMarcha(_THIS := _THIS.botonera)
(* Llamada a metodo*)
CAL PanelMando_HayModoManual(_THIS := _THIS.botonera)
LD ControlProceso_GetCondicionesIniciales._Retorno
AND PanelMando_HayMarcha._Retorno
AND PanelMando_HayModoManual._Retorno
ST _AUXT_1
(* Llamada a metodo*)
CAL ControlProceso_GetFinCiclo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL PanelMando_HayRearme(_THIS := _THIS.botonera)
LD ControlProceso_GetFinCiclo._Retorno
AND PanelMando_HayRearme._Retorno
ST _AUXT_4
(* Llamada a metodo*)
CAL ControlProceso_GetFinCiclo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL PanelMando_HayRearme(_THIS := _THIS.botonera)
LD ControlProceso_GetFinCiclo._Retorno
AND PanelMando_HayRearme._Retorno
ST _AUXT_6
(* Llamada a metodo*)
CAL ControlProceso_GetCondicionesIniciales(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL PanelMando_HayMarcha(_THIS := _THIS.botonera)
(* Llamada a metodo*)
CAL PanelMando_HayModoAutomatico(_THIS := _THIS.botonera)
LD ControlProceso_GetCondicionesIniciales._Retorno
AND PanelMando_HayMarcha._Retorno
```

Figura D.109: Traducción del método *"Run"* de la clase *"Estacion2"* - parte 1

D.7.17. Clase Transfer

El *"transfer"* es el encargado de llevar la base hasta la estación e indicarle a ésta que comience el proceso de insercción de rodamientos. Si el rodamiento tiene una altura incorrecta se procede a su expulsión, y en caso contrario se realizará la evacuación o inserción del rodamiento en la base que está a la espera en el sistema de transferencia *"transfer"*. Una vez que la estación termina su ejecución, enviá una señal al *"transfer"* sobre el estado del proceso indicando si ha habido un error o si la inserción del rodamiento se efectuo de forma correcta.

El comienzo de la ejecución de las estaciones se hace por medio del método *"Run"*. Este método es virtual y común en todas las estaciones por lo que el *"transfer"* invoca la ejecución del método *"Run"* del interfaz *"EstacionBase"*. Es el propio sistema el encargado de decidir a que estación debe invocar en cada caso.

Tanto la estación de inserción de rodamientos como el resto de estaciones se presentan como atributos privados de la propia clase *"Transfer"*. Para poder acceder de forma sencilla a los objetos estación, la clase *"Transfer"* posee un atributo privado llamado *"Vector_ Estacion"* que es un array de objetos que contiene cada

```
AND PanelMando_HayModoAutomatico._Retorno
ST _AUXT_8
(* Llamada a metodo*)
CAL ControlProceso_GetFinCiclo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL PanelMando_HayParo(_THIS := _THIS.botonera)
LD ControlProceso_GetFinCiclo._Retorno
AND PanelMando_HayParo._Retorno
ST _AUXT_11
(* Llamada a metodo*)
CAL ControlProceso_GetFinCiclo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL PanelMando_HayParo(_THIS := _THIS.botonera)
LD ControlProceso_GetFinCiclo._Retorno
AND PanelMando_HayParo._Retorno
ST _AUXT_13
LD E0.X
ST E0.C
LD E1.X
ST E1.C
LD E2.X
ST E2.C
LD E4.X
ST E4.C
LD E8.X
ST E8.C
LD E5.X
ST E5.C
LD E9.X
ST E9.C
LD E12.X
ST E12.C
LD E3.X
ST E3.C
LD E6.X
ST E6.C
LD E10.X
ST E10.C
LD E7.X
ST E7.C
LD E11.X
ST E11.C
LD E13.X
ST E13.C
(* Llamada a metodo*)
CAL PanelMando_HayRearme(_THIS := _THIS.botonera)
LD E1.C
AND PanelMando_HayRearme._Retorno
OR ( E12.C
AND TRUE
)
OR ( E13.C
AND TRUE
)
S E0.X
R E1.X
R E12.X
R E13.X
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
LD ControlProceso_GetFallo._Retorno
ANDN _AUXT_1
ANDN _AUXT_8
AND E0.C
OR ( E4.C
AND ControlProceso_GetFallo._Retorno
)
OR ( E5.C
AND ControlProceso_GetFallo._Retorno
)
OR ( E6.C
AND ControlProceso_GetFallo._Retorno
)
OR ( E7.C
AND ControlProceso_GetFallo._Retorno
)
S E1.X
R E0.X
R E4.X
R E5.X
R E6.X
R E7.X
```

Figura D.110: Traducción del método *"Run"* de la clase *"Estacion2"* - parte 2

```
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
LD _AUXT_1
ANDN ControlProceso_GetFallo._Retorno
ANDN _AUXT_8
AND E0.C
S E2.X
R E0.X
(* Llamada a metodo*)
CAL PanelMando_HayAlto(_THIS := _THIS.botonera)
(* Llamada a metodo*)
CAL PanelMando_HayBajo(_THIS := _THIS.botonera)
LD PanelMando_HayAlto._Retorno
ANDN PanelMando_HayBajo._Retorno
AND E2.C
S E4.X
R E2.X
LD E4.C
AND _AUXT_4
S E8.X
(* Llamada a metodo*)
CAL PanelMando_HayBajo(_THIS := _THIS.botonera)
(* Llamada a metodo*)
CAL PanelMando_HayAlto(_THIS := _THIS.botonera)
LD PanelMando_HayBajo._Retorno
ANDN PanelMando_HayAlto._Retorno
AND E2.C
S E5.X
R E2.X
LD E5.C
AND _AUXT_6
S E9.X
LD E8.C
AND TRUE
OR ( E9.C
AND TRUE
)
S E12.X
R E8.X
R E9.X
(* Llamada a metodo*)
CAL ControlProceso_GetFallo(_THIS := _THIS.proceso)
LD _AUXT_8
ANDN ControlProceso_GetFallo._Retorno
ANDN _AUXT_1
AND E0.C
S E3.X
R E0.X
(* Llamada a metodo*)
CAL PanelMando_HayAlto(_THIS := _THIS.botonera)
(* Llamada a metodo*)
CAL PanelMando_HayBajo(_THIS := _THIS.botonera)
LD PanelMando_HayAlto._Retorno
ANDN PanelMando_HayBajo._Retorno
AND E3.C
S E6.X
R E3.X
LD E6.C
AND _AUXT_11
S E10.X
(* Llamada a metodo*)
CAL PanelMando_HayBajo(_THIS := _THIS.botonera)
(* Llamada a metodo*)
CAL PanelMando_HayAlto(_THIS := _THIS.botonera)
LD PanelMando_HayBajo._Retorno
ANDN PanelMando_HayAlto._Retorno
AND E3.C
S E7.X
R E3.X
LD E7.C
AND _AUXT_13
S E11.X
LD E10.C
AND TRUE
OR ( E11.C
AND TRUE
)
S E13.X
R E10.X
R E11.X
LDN A6
JMPC _SALTO_0
_SALTO_0:
LDN EncenderLampara
JMPC _SALTO_1
_SALTO_1:
LDN F5_Alto
JMPC _SALTO_2
_SALTO_2:
LDN F5_Bajo
JMPC _SALTO_3
_SALTO_3:
LDN F1_Alto
JMPC _SALTO_4
_SALTO_4:
LDN F1_Bajo
JMPC _SALTO_5
_SALTO_5:
LD E0.X
ST A6
LD E1.X
ST EncenderLampara
LD E6.X
ST F1_Alto
LD E7.X
ST F1_Bajo
LD E4.X
ST F5_Alto
LD E5.X
ST F5_Bajo
END_FUNCTION_BLOCK
```

Figura D.111: Traducción del método *"Run"* de la clase *"Estacion2"* - parte 3

```
FUNCTION_BLOCK Estacion2_GetFallo
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : Estacion2;
END_VAR

LD _This.ElementoComplejoBase.fallo
ST _Retorno
END_FUNCTION_BLOCK


FUNCTION_BLOCK Estacion2_ResetFallo
VAR_IN_OUT
        _This : Estacion2;
END_VAR

LD FALSE
ST _This.ElementoComplejoBase.fallo
END_FUNCTION_BLOCK


FUNCTION_BLOCK Estacion2_GetCondicionesIniciales
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : Estacion2;
END_VAR

LD _This.ElementoComplejoBase.condicionesIniciales
ST _Retorno
END_FUNCTION_BLOCK


FUNCTION_BLOCK Estacion2_ResetCondicionesIniciales
VAR_IN_OUT
        _This : Estacion2;
END_VAR

LD FALSE
ST _This.ElementoComplejoBase.condicionesIniciales
END_FUNCTION_BLOCK


FUNCTION_BLOCK Estacion2_GetFinProceso
VAR_OUTPUT
        _Retorno : BOOL;
END_VAR
VAR_IN_OUT
        _This : Estacion2;
END_VAR

LD _This.ElementoComplejoBase.finProceso
ST _Retorno
END_FUNCTION_BLOCK


FUNCTION_BLOCK Estacion2_ResetFinProceso
VAR_IN_OUT
        _This : Estacion2;
END_VAR

LD FALSE
ST _This.ElementoComplejoBase.finProceso
END_FUNCTION_BLOCK
```

Figura D.112: Traducción de varios métodos de la clase *"Estacion2"* a IL

Algoritmo D.40 Método *"InicializarVectorEstaciones"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::InicializarVectorEstaciones ( )
   Vector_Estacion(1):=Estacion1;
   Vector_Estacion(2):=Estacion2;
   Vector_Estacion(3):=Estacion3;
   Vector_Estacion(4):=Estacion4;
   Vector_Estacion(5):=Estacion5;
   Vector_Estacion(6):=Estacion6;
END_METHOD
```

uno de los objetos que instancian las distintas clases estación. La inicialización del vector de estaciones (*"Vector_Estacion"*) se realia en el método *"InicializarVectorEstaciones"* que es invocado en el contructor de la propia clase (ver algoritmo D.40).

Para que el *"transfer"* sepa cuando el palet llega a cada estación, se mapean los sensores de presencia de cada estación. Para facilitar el manejo de dichos sensores y poder mezclar su uso con el del vector *"Vector_Estacion"*, la clase *"Transfer"* posee un segundo array como atributo privado que contiene el mapeo de los sensores de presencia (*"Vector_pp"*). Cada posición del vector *"Vector_pp"* contiene el valor del sensor de cada estación haciendo una correspondencia con el número de cada estación, siendo la posición 1 del vector el sensor de la estación 1 (estación de bases), la posición 2 el sensor de la estación 2 (estación de rodamientos) y asi sucesivamente. Para acceder y usar este vector, la clase *"Transfer"* proporciona los servicios *"GetPresenciaPalet"* y *"SetPresenciaPalet"*.

En la figura D.113 se muestra las ramas concernientes al proceso que permite ejecutar las distintas estaciones.

Las etapas 18 y 19, así como las transiciones asociadas a la etapa 21, tienen la misión de simular un bucle *"FOR"* para la ejecución de las estaciones usando el polimorfismo.

La etapa 18 tiene una acción denominada *"InicializarBucle"* que contiene el código ST:

```
i:=0;
```

Figura D.113: Proceso de ejecución de las estaciones en el transfer

Algoritmo D.41 Método *"GetFinProceso"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::GetFinProceso (i:BYTE)
  RETURN Vector_Estacion[i].GetFinProceso;
END_METHOD
```

Algoritmo D.42 Traducción a código estructurado en lenguaje ST del método *"GetFinProceso"* perteneciente a la clase *"Transfer"*

```
FUNCTION_BLOCK Transfer_GetFinProceso
VAR_IN_OUT
   _This : Transfer;
END_VAR
VAR_IN
   i : BYTE;
END_VAR
  RETURN (*_This.Vector_Estacion[i].vPointer)+3 ( );
END_FUNCTION_BLOCK
```

La etapa 19 tiene una acción denominada *"InicializarBucle"* que contiene el código ST:

```
i++;
```

La transición asociada a la etapa 18 cotiene una llamada al método *"GetPresenciaPalet"* perteneciente a la clase *"Transfer"* (ver algoritmo D.41). La traducción de este método a código ST normal se muestra en el algoritmo D.42 y su traducción a código IL se puede observar en la figura D.114. La suma +3 al vector *"Vector_Estacion"* se realiza para acceder al tercer método virtual de las clase estación.

En la etapa 20 se hace una llamada a al método *"Transmitir"* que coloca un bit en la posición de memoria compartida por todos los PLCs de las estaciones para comenzar la ejecución de éstas. Por otro lado, en la misma etapa 20 se hace una invocación del método *"Run"* de la clase *"Transfer"*. En dicho método (ver algoritmo D.43) se hacen las invocaciones a los métodos *"Run"* particulares de cada estación por medio del polimorfismo. La traducción de este método a código ST normal se muestra en el algoritmo D.43 y su traducción a código IL se puede observar en la figura D.44.

```
FUNCTION_BLOCK Transfer_GetFinProceso
VAR IN
        i : BYTE;
END_VAR
VAR_IN_OUT
         _This : Transfer;
END_VAR
VAR
       aux_vPointer : *VOID;
END_VAR

LD _This.Vector_Estacion[i].vPointer
ADD 3
ST aux_vPointer
CAL @aux_vPointer [ i ] ( )
END_FUNCTION_BLOCK
```

Figura D.114: Traducción a lenguaje IL del método *"GetFinProceso"* de la clase *"Transfer"*

Algoritmo D.43 Método *"Run"* perteneciente a la clase *"Transfer"*

```
METHOD Transfer::Run (i:BYTE)
  Vector_Estacion[i].Run();
END_METHOD
```

Algoritmo D.44 Traducción a código estructurado en lenguaje ST del método *"Run"* perteneciente a la clase *"Transfer"*

```
FUNCTION_BLOCK Transfer_Run
VAR_IN_OUT
  _This : Transfer;
END_VAR
VAR_IN
  i : BYTE;
END_VAR
  (*_This.Vector_Estacion[i].vPointer) ( );
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK Transfer_Run
VAR IN
        i : BYTE;
END_VAR
VAR_IN_OUT
         _This : Transfer;
END_VAR

CAL @_This.Vector_Estacion[i].vPointer ( )
END_FUNCTION_BLOCK
```

Figura D.115: Traducción a lenguaje IL del método *"Run"* de la clase *"Transfer"*

Glosario

ACM Association for Computing Machinery (Asociación para la Maquinaria de Computación) 11

AD Conversión analógica-digital 53

ADEPA Agence Nationale pour le Développment de la Production Automatisée (Agencia nacional por el Desesarrollo de la Producción Automatizada) 58

ADN Ácido Desoxirribonucleico (Deoxyribonucleic Acid) 31

AFCET Association Française pour la Cybernétique Economique et Technique (Asociación Francesa para la Economía, la Cibernética y la Técnica) 57

Algol Algorithmic Language (Lenguaje de Algoritmo) 10

ANSI American National Standards Institute (Instituto de Estandares Nacional Americano) 183

AO Architecture Object (Objetos de Arquitectura) 393

AOO Análisis Orientado a Objetos (Object Oriented Analysis) 367

API Interface de Programación de Aplicaciones (Application Program Interface) 393

Basic Beginners All Purpose Symbolic Instruction Code (Código de Instrucciones de Símbolos para Principantes para todos los Propósitos) 11

BCPL	Combined Programming Language (Lenguaje de Programación Básico Combinado) 12
CAD	Computer Aided Design (Diseño Asistido por Ordenador) 419
CASE	Computer Aided Software Engineering (Ingeniería de Software Asistida por Computadora) 69
CIM	Computer Integrated Manufacturing (Fabricación Integrada por Computador) 7
CLOS	Common Lisp Object System (Sistema de Objetos Común de Lisp) 29
CORBA	Common Object Request Broker Architecture (Arquitectura de Requerimiento de Intermediación de Objetos Comunes) 383
COSTS	Commercial Off The Shelf (Adjetivo que describe productos SW ó HW que están disponibles en el mercado y que pueden usarse en la construcción de sistemas propios) 40
CRC	Class, Responsability and Collaboration (Clase, Responsabilidad y Colaboración) 35
DA	Conversión digital-analógica 53
DCOM	Distributed Component Object Model (Modelo de Objetos de Componentes Distribuido) 382
DCS	Distributed Control System (Sistemas de Control Distribuido) 53
DDC	Digital Direct Control (Sistemas de Control Digital Directo) 53
DDT	Estructuras de Datos Definidas por el usuario 431
DFB	Bloques Funcionales Definidos por el usuario 431
DOO	Diseño Orientado a Objetos (Object Oriented Design) 367
EFB	Bloques Funcionales Elementales 431
FB	Functional Block (Bloque Funcional) 66
FBA	Function Block Adapter (Bloque Funcional Adaptado) 69

FBD Function Block Diagram (Diagrama de Bloques Funcionales) 200

FBD++ Function Block Diagram object oriented(Diagrama de Bloques Funcionales orientadado a objetos) 200

Fortran Formula Translating System (Formula de Sistema de Traducción) 10

GDMMA Graphe Descriptif des Modes de Marches et dÁrrètes (Gráfico Descriptivo de los Modos de Marcha y Parada) 450

GEMMA Guide d´Etude des Modes de Marches et d´Arrèts (Guía de Estudio de los Modos de Marcha y Parada) 449

GENIA Grupo de Entornos Integrados de Automatización 199

GNU GNU Lesser General Public License (GNU Licencia General Pública Reducida) 38

GRAFCET Graphe Fontcionnel de Commande, Etapes et Trasitions (Gráfico Funcional de Mando, Etapa y Transición) 55

GUI Graphical User Interface (Interfaz Gráfica de Usuario) 45

HMI Human MAchine Interface (Interfaz Hombre Máquina) 65

HMS Holonic Manufacturing Systems (Sistemas de Manufactura Holónicos) 75

HOOD Hierarchic Object Oriented Design (Diseño Orientado a Objetos Jerárquico) 46

IA Inteligancia Artificial (Artificial Intelligence) 37

ICL Idiomatic Control Language (Lenguaje de Control Idiomático) 70

IDL Interface Definition Language (Lenguaje de Definición de Interfaces) 384

IDN Integrated Design Notation (Notación de Diseño Integrada) 71

IEC International Electrotechnical Commision (Comosión Electrotécnica Internacional) 8

IEEE Institute of Electrical and Electronics Engineers (Ingenieros del Instituto Eléctrico y Electrónico) 39

IFIP International Federation for Information Processing (Federación Internacional para el Procesamiento de Información) 12

IL Instruction List (Lista de Instrucciones) 200

IL++ Instruction List object oriented (Lista de Instrucciones orientada a objetos) 200

IMS Intelligent Manufacturing Systems (Sistemas de Manufactura Inteligentes) 76

INDO Interoperable Network Distributed Object (Objetos Distribuidos de Red Interoperables) 385

ISA Ingeniería de Sistemas y Automatización 408

JDK Java Development Kit (Kit de Desarrollo de Java) 29

LAV Laboratorio de Automatización Virtual, Conjunto de herramientas software para la automatización de preocesos (Virtual Automation Laboratory. Set of software tools to help developing automation projects) 201

LD Ladder Diagram (Diagrama de Escalera) 200

LD++ Ladder Diagram object oriented (Diagrama de Escalera orientado a objetos) 200

LOO Lenguaje Orientado a Objeto (Object Oriented Language) 29

MAP Manufacturing Automation Protocol (Protocolo de Automatización de Manufactura) 397

middleware Software de intermediación entre una aplicación servidor y una red hererogenea de clientes 64

MIOOP Modification of the IEC 61131 standard for Object Oriented Programming (Modificación del estándar IEC 61131 para la Programación Orientada a Objetos) 81

MIT Massachusetts Institute of Technology (Instituto Tecnológico de Massachusetts) 11

MLAV Metodología LAV (LAV Methodology) 76

Modicon Modular Digital Controler (Controlador Digital Modular) 6

OD Objeto Distribuido (Distributed Object) 385

OFS OPC Factory Server 433

OLE Object Linking and Embedding (Objetos Enlazados y Empotrados) 413

OMG Object Management Group (Grupo de Gestión de Objetos) 47

OMT Object Modeling Technique (Técnicas de Modelado Orientadas a Objetos) 46

OO Object Oriented (Orientación a Objetos) 13

OODLE Object Oriented Design Learning Environment (Lenguaje de Diseño Orientado a Objetos) 46

OOPSLA Object Oriented Programming, Systems, Languages and Applications (Programación, Sistemas, Lenguajes y Apliaciones Orientadas a Objetos) 30

OOSD Object Oriented System Development (Diseño orientado a objetos estructurado) 46

OOSE Object Oriented Software Engineering (Modelo de Ingeniería del Software Orientada a Objetos) 46

OPC OLE for Process Control (OLE para el Control de Procesos) 413

OSACA Open System Architecture for Controls within Automation Systems (Arquitecturas Abiertas de Control para Sistemas Automatizados)392

PDA Personal Data Administrator (Administrador de Datos Personal) . . 4

PLC Programmable Logic Controller (Controlador Lógico Programable) 5

POO Paradigma Orientado a Objetos (Object Oriented Paradigm) 14

POU	Program Organization Unit (Unidad de Organización de Programa) 66
RAD	Rapid Application Development (Desarrollo Rápido de Aplicaciones) 41
RAE	Real Academia Española 1
RCM	Remote Connections Manager (Gestor de Conexiones Remotas) . 385
RdP	Redes de Petri (Petri Networks) 5
RMI	Remote Method Invocation (Invocación Remota de Métodos) . . . 382
ROOM	Real Time Object Oriented Modeling (Modelado Orientado a Objetos en Tiempo Real) 67
RTS	Real Time Systems (Sistemas en Tiempo Real) 67
SCADA	Supervisory Control and Data Acquisition (Supervisión de Control y Adquisición de Datos) 390
SFC	Sequential Function Chart (Gráfico Funcional Secuencial) 58
SFC++	Sequential Function Chart object oriented (Gráfico Funcional Secuencial orientado a objetos) 200
SimPLC	Simulador de PLC Orientado a Objetos. Herramienta que permite la programación de sistemas control usando los lenguajes definidos en la norma IEC 61131-3 y sus versiones OO. Software tool to programme control systems using IEC 61131-3 norm languages and their OO versions. 197
SOMA	Service Oriented Modeling (Modelado Orientado a Servicios) 46
ST	Structured Text (Texto Estructurado) 200
ST++	Structured Text object oriented (Texto Estructurado orientado a objetos) 200
T4G	Técnicas de cuarta generación 41
TCP/IP	Transmission Control Protocol/Intetrnet Protocol (Protocolo de Control de Transmisiones/Protocolo de Internet) 7

TMT Time To Market (Tiempo que tarda un producto desde que se concibe hasta que se vende) 39

TOO Tecnología Orientada a Objetos (Object Oriented Technology) . . . 14

UDE Unity Developer's Edition (Edición para Desarrolladores de Unity) 201

ULS Unity Library Server 429

UML Unified Modeling Language (Lenguaje Unificado de Modelado) . . . 47

UML-RT UML-Real Time 68

UMSS Unity Manager Studio Server 433

Univac Universal Automatic Computer (Computadora Automática Universal) 9

UPB Unity Pro Broker 421

UPS Unity Pro Server 421

USMS Unity Studio Manager Server 425

VEE Virtual Engineering Environmment (Entorno Virtual de Ingeniería) 202

VM Virtual Machine (Máquina Virtual) 382

VMT Virtual Method Table (Tabla de Métodos Virtuales) 160

VPointer Virtual method Pointer (Puntero a métodos Virtuales) 160

XML Extensible Markup Language (Lenguaje de Marcas Extensible) . . 201

Bibliografía

[AA92] M. Alfonseca and A. Alcalá. *Programación orientada a objetos. Teoría y técnicas OOP para el desarrollo de software*. Ediciones Anaya Multimedia, S.A., 1992.

[AB07] Alessi M. Aiello, G. and M. Bruccoleri. An agile methodology for manufacturing control systems development. In *5th IEEE International Conference*, volume 2, pages 817–822. Industrial Informatics, 2007.

[Abb83] R.J Abbott. Program design by informal english descriptions. *Communications of the ACM*, 26(11):882–894, 1983.

[AFC77] AFCET. Normalisation de la représentation du cahier des charges d'un automatisme logique. Technical report, August 1977.

[AH92] R. David Alla and H. *Petri Nets and Grafcet: Tools for modelling discrete events systems*. PrenticeHal, Inc., 1992.

[And95] D.E. Anderson. Extensible programming : Beyond reusable objetcs. *Behavior Resarch Methods, Instruments and Computers*, 27(2):131–133, 1995.

[AT91] S.M. Alessi and S. R. Trollip. *Computer-Based Instruction. Method and Development*. Prentice Hall College Div; 2 Sub edition, 1991.

[Bac54] John Backus. The ibm 701 speedcoding system. In *Journal of the ACM (JACM)*, volume 1, pages 4–6, 1954.

[BC08] F. Basile and P. Chiacchio. A two-stage modelling architecture for distributed control of real-time industrial : Application of uml and petri net. *Computer Standards and Interfaces*, 31(3):528–538, 2008.

[Ben93] S. Bennett. *A History of Control Engineering 1930-1955 (I E E Control Engineering Series)*. Institution of Engineering and Technology, 1993.

[BG95] Monari P. Bonfatti, F. and G. Gadda. Bridging structural and software design of plc-based system families. In *ICECCS 95*, pages 377–384. IEEE, ed., 1995.

[BJ98] Rumbaugh K. Booch, G. and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 1998.

[BJ99] Rumbaugh K. Booch, G. and I. Jacobson. *The Unificed Software Development Process*. Addison-Wesley, 1999.

[Blu92] A. Blum. *Neural Networks in C++ : An Objetc-Oriented framework for building Connectionist Systems*. John Wiley and Sons, 1992.

[BN07] Fletcher M. Brennan, W. and D.H. Norrie. Reconfiguring real-time holonic manufacturing systems. In *12th International Workshop on Database and Expert Systems Applications*, Munich, Germany, 2007.

[Boe88] B. Boehm. A spiral model for software development and enhacement. *Computer*, 21(5):61–72, 1988.

[Boo47] G. Boole. *The mathematical analysis of logic : being an essay towards a calculus of deductive easoning*. 1847.

[Boo84] Grandy Booch. Object-oriented development. *IEEE transactions on software engineering*, 12(2):211–221, 1984.

[Boo91] G. Booch. *Object oriented design with applications*. Addison-Wesley Professional; 2 edition, 1991.

[Boo94] Grandy Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, benjamin cummings edition, 1994.

[Bro95] F. Brooks. *The Mytical Man-Month*. Addison-Wesley Professional; 2 edition, 1995.

[Bro97] D. Brown. *An Introduction to Object-Oriented Analysis : Objects in Plain English*. John Wiley and Sons, Inc., 1997.

[BS05] Giese H. Burmester, S. and W. Sch´afer. Model driven architecture for hard real-time systems. In *Proceedings of the European Conference on Model Driven Architecture Foundations and Applications*, pages 25–40, Springer, Berlin, 2005.

[BS06] Fantuzzi C. Bonfe, M. and C. Secchi. Behavioural inheritance in object-oriented models for mechatronic systems. *International Journal of Manufacturing Research*, 1(4):421–441, 2006.

[bV90] G. booch and M. Vilot. The design of the c++ booch components. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–11, Ottawa, Canada, 1990.

[bV93] G. booch and M. Vilot. Simplifying the c++ booch components. *The C++ report*, pages 59–89, 1993.

[Car91a] T.A. Cargill. The case againts multiple inheritance in c++. *Computer Systems*, 4(1):101–109, 1991.

[Car91b] M. Caroll. Using multiple inheritance to implement abstract data types. *The C++ Report*, 3(4):34–43, 1991.

[Cho9a] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959a.

[Cho9b] N. Chomsky. A note on phrase structure grammars. *Information and Control*, 2(4):393–395, 1959b.

[CJ94] Colgate R.J. Hunter .J.C. Capper, N.P. and M.F. James. The impact of object-oriented technology on software quality: Three case histories. *IBM Systems Journal*, 33:131–157, 1994.

[CK07] F. Chiron and K. Kouiss. Design of iec 61131-3 fb using sysml. In *Proceedings of the 15th Mediterranean Conference on Control and Automation*, Athens, Greece, 2007. Mediterranean Conference on Control and Automation(MED07).

[Coa92] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–158, 1992.

[Col93] D. Coleman. *Object-Oriented development : The fusion method*. Prentice Hall, 1993.

[Cot94] A. Cota. Software engineering. *Avance solutions*, pages 5–13, 1994.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall PTR; 2 edition, 1990.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall, 1991.

[DB94] O. Duran and A. Batocchio. A high-level object-oriented programmable controller programming interface. In *ISIE 94*, pages 226–230. I.E.S. IEEE, ed., 1994.

[Dee00] M. Deen. A computational model for holonic manufacturing systems. In *Holonic manufacturing systems (HMS). International symposium*, 2000.

[DeM82] Tom DeMarco. *Controlling software projects*. Englewood Cliffs, NJ: Prentice Hall, 1982.

[Des92] P. Desfray. *Ingenerie des objetcts : Approche clase-relation application a C++*. Editions Masson, 1992.

[DF06] Krause J. Diedrich, C. and A. Franke. Uml based software development under safety constraints. In *Sicherheit (Dittmann, J. (ED.))*, pages 361–368, Magdeburg, 2006.

[DH72] Dijkstra E. Dahl, O. and C.A.R. Hoare. *Structured programming*. Academic Press, 1972.

[Die08] S. Diehm. Anforderungen an die modellierung von funktionsbausteinen mit uml aus sicht eines systemanbieters. In *Automation and Embedded Systems*, Oldenbourg Industrieverlag, 2008.

[Dij68] E. Dijkstra. Go to statement considered harmful. *Go To Statement Considered Harmful*, 11(3):147–148, 1968.

[DJ67] Nygaard Dahl and Ole Johan. Simula 67. In *IFIP TC 2 Working Conference on Simulation Languages*, Lysebu, Oslo, 1967.

[Dou99] B.P. Douglas. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[DS04] Kleanthis C. Thramboudilis Doukas and George S. An iec-compliant field device model for distributed control applications. *2 IEE international conference on industrial informatics*, page 6, 2004.

[DW97] C.M. Davidson and J.Mc. Whinne. Engineering the control software develpment. In *5th International conference on FACTORY*, pages 247–250, 1997.

[Dye92] M. Dyer. *The Cleanroom Approach to Quality Software Development*. Wiley, 1992.

[EO07] Marcos M. Estevez, E. and D. Orive. Automatic generation of plc automation projects from component based models. In *International Journal of Advanced Manufacturing Technology*, volume 6, pages 527–540, Springer, Berlin, 2007.

[EW92] Kurtz B.D. Embley, D.W. and S.N. Woodfield. *Object-oriented systems analysis : A model-driven approach*. Yourdon Press, 1992.

[FG04] G'ohner P. Fischer, K. and F. Gutbrodt. Conceptual design of an engineering model for product and plant automation. In *Integration of Software Specification Techniques for Applications in Engineering*, pages 301–321, Springer, Berlin, 2004.

[Fre87] P. Freeman. A perspective on reusability. *IEEE Tutorial: Software Reusability (ed. P. Freeman), IEEE Computer Society Press*, pages 2–8, 1987.

[Fri09] A. Friedrich. *Anwendbarkeit von Methoden und Werkzeugen des konventionellen Softwareengineering zur Modellierung und Programmierung von Steuerungssystemen*. PhD thesis, Universidad Kassel, 2009.

[FV03] L. Ferrarini and C. Veber. Design and lmplementation of machining centers control functions with object-oriented techniques. In *Proceedings of the 2003 IEEUASME lntemational Conference on Advanced Intelligent Mechatronics (AIM 2003)*, volume 2, pages 1037–1042, 2003.

[GB50] D.P. Campbell G.S. Brown. Its growth and promise in process-control problem. Mechanical engineering. *Instrument engineering*, 72(2):124, 1950.

[Geh05] M. Gehrke. *Entwurf mechatronischer Systeme auf Basis von Funktionshierarchien und Systemstrukturen*. PhD thesis, Universidad Paderborn, 2005.

[GH] H. Giese and S. Henkler. A survey of approaches for the visual model-driven development of next generation software-intensive systems. In *Journal of Visual Languages and Computing*, number 6.

[Gil88] T. Gilb. *Principles of software project management*. Addison-Wesley Professional, 1988.

[Gon02] V.M. Gonzalez. *Metodología de analisis y modelado de sistemas de eventos discretos mediante tecnicas orientadas a objetos. Aplicación a la generación de la lógica de control basada en IEC 61131-3 (MLAV)*. PhD thesis, Oviedo, 2002.

[Gra95] I. Graham. *Migrating to Object Thechnology*. Addison Wesley Longman; Har/Dsk edition, 1995.

[Gra00] I. Graham. *Objetc Oriented Methods : Principles and Practice*. Addison-Wesley Professional; 3 edition, 2000.

[Gro09] ARC Advisory Group. Programmable logic controllers worldwide outlook. five year market analysis and technology forecast through 2013. Technical report, 2009.

[Gru99] G. Grube. Perspectives of cacsd : Embedding the control system design process into a virtual engineering environment. In *IEEE International Symposium on Computer Aided Control System Design*, pages 297–302, Kohala Coast, HI, 1999.

[GS05] Joy B. Gosling, K. and G. Steel. *The Java Language Specification*. Addison Wesley; 3 edition, 2005.

[GW05] A. Gemino and Y. Wand. Complexity and clarity in conecptual modeling: comparison of mandatory and optional properties. In *Data and Knowledge Engineering*, number 3, pages 301–326, Elsevier, Munchen, 2005.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of computer programming*, pages 231–274, 1987.

[Hev03] T. Heverhagen. Verification of function block adapters through model-checking. *CODEN ATRTE9*, 51(4):153–63, 2003.

[HP98] D. Harel and M. Politi. *Modeling reactive systems with Statecharts : the Statemate approach*. McGraw-Hill, 1998.

[HT01] T. Heverhagen and R. Tracht. Integrating uml-realtime and iec 61131-3 with function block adapters. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 395–402, 2001.

[IEC88] IEC. Preparation of function charts for control systems. Technical report, International electronic commision, 1988.

[IEC93] IEC. Programmable controllers - part 3: Programming languages. technical report. Technical report, 1993.

[IEC00a] IEC. Programmable controllers - part 7: Fuzzy control programing. technical report. Technical report, International Electrotechnical Commission, 2000.

[IEC00b] IEC. Programmable controllers - part 8: Guidelines for the application and implementation of programming languages. technical report. Technical report, International Electrotechnical Commission, 2000.

[IEC00c] IEC. Fieldbus standard for use in industrial control systems - part 2: Physical layer specification and service definition. technical report. Technical report, International Electrotechnical Commission, August 2000.

[IEC00d] IEC. Digital data communications for measurement and control - fieldbus for use in industrial control systems - part 3: Data link service definition. technical report. Technical report, International Electrotechnical Commission, January 2000.

[IEC00e] IEC. Digital data communications for measurement and control - fieldbus for use in industrial control systems - part 6: Application layer protocol specification. technical report. Technical report, International Electrotechnical Commission, January 2000.

[IEC00f] IEC. Digital data communications for measurement and control- fieldbus for use in industrial control systems - part 4: Data link protocol specification. technical report. Technical report, International Electrotechnical Commission, January 2000.

[IEC00g] IEC. Programmable controllers - part 5: Communications. technical report. Technical report, International Electrotechnical Commission, January 2000.

[IEC00h] IEC. Function blocks for industrial-process measurement and control systems, part 1: 'architecture'. technical report. Technical report, International Electrotechnical Commission, September 2000.

[IEC01] IEC. Function blocks for industrial-process measurement and control systems - part 2: Software tools requirements. standard. Technical report, International Electrotechnical Commission, May 2001.

[IEC02] IEC. Function blocks for industrial-process measurement and control systems - part 4: Rules for compliance profiles. standard. Technical report, International Electrotechnical Commission, July 2002.

[IEC2a] IEC. Programmable controllers - part 1 : General information. technical report. Technical report, 1992a.

[IEC2b] IEC. Programmable controllers - part 2 : Equipment requirements and test. technical report. Technical report, 1992b.

[IEE90] IEE. Ieee guide to the posix open system environment (ose). Technical report, IEEE, 1990.

[JA98] Ohman M. Johansson, S. and Karl-Erik Arzen. Implementation aspects of the plc standard iec 1131-3. *Control Engineering practice*, 6:547–555, 1998.

[JO92] Christerson M. Jonsson P. Jacobson, I. and G. Overgaard. *Object-oriented software engineering : A use case drive approach*. Workingham : Addison-Wesley, 1992.

[JR99] Booch G. Jacobson, I. and J. Rumbaug. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[KA00] Nickel U. Niere J. Kohler, H.J. and Zundorf A. Integrating uml diagrams for production control systems. In *International Conference on Software Engineering, Proceedings of the 22nd international conference on Software engineering*, pages 241–251, Limerick, Ireland, 2000.

[Kay96] A. C. Kay. *he early history of Smalltalk*. ACM New York, NY, USA, history of programming languages—ii edition, 1996.

[kG97] Kim J. Watts M. kasabov, K. and A. Gray. A fuzzy neural network architecture for adaptive learning and knowledge acquisition. *Inf. Sci.*, pages 155–175, 1997.

[KH94] J. Kerr and R. Hunter. *Inside Rad: How to Build Fully Functional Computer Systems in 90 Days or Less (Systems Design and Implementation)*. McGraw-Hill Companies, 1994.

[KN00] R. Kremer and D. Norrie. Human-holon interface architecture. In *Holonic manufacturing systems (HMS). International symposium*, 2000.

[KO04] S. Kajihara and M. Ono. Development and products of the object-oriented engineering tool for the integrated controller based on iec 61131-3. In *SICE Annual Conference in Sapporo*, volume 3, pages 1952–1956, 2004.

[Koe67] A. Koestler. *The ghost in the machine*. Penguin Group, 1967.

[Koo94] R.W. Koontz. Lessons from the experts. In *Conference on Object-Oriented Software Development at St. Thomas University*, volume 4, St. Paul, MN, 1994.

[Kru92] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[KS09] S. Kain and F. Schiller. Supporting the operation phase of manufacturing systems by synchronous and forward simulation. In *Proceedings of 3rd International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV 2009)*, Munich, Germany, 2009.

[Kuo96] C. B. Kuo. *Sistemas de Control automático*. Prentice Hall, 1996.

[KVH04] U. Katzke and B. Vogel-Heuser. Analysis and state of the art of modules in industrial automation. *Automation Technology in Practice international*, 2(1):23–31, 2004.

[KVH05] U. Katzke and B. Vogel-Heuser. Design and application of an engineering model for distributed process automation. In *Proceedings of the American Control Conference*, pages 2960–2965, Minneapolis, 2005.

[Lew94] G. Lewis. What is software engineering? *DataPro (4015)*, pages 1–10, 1994.

[Lew95] Robert Lewis. *Programming Industrial Control Systems using IEC 1131-3*. London, Herts, U.K, 1995.

[Lew01] Robert Lewis. *Modelling control systems using IEC 61499. Applying function blocks to distributed systems.* The Institution of Electrical Engineers, Herts, UK, 2001.

[Lic04] T. Licht. *Ein Verfahren zur zeitlichen Analyse von UML-Modellen beim Entwurf von Automatisierungs systemen*. PhD thesis, Universidad Ilmenau, 2004.

[Mar95] J. M. Marques. *Jerarquias de Herencia en el Diseno de Software Orientado a Objetos*. PhD thesis, Valladolid, 1995.

[MC99] Ferrarini L. Matfezzoni, C. and E. Carpanzano. Objed-oriented models for advanced automation engineering. *Control Engineering Practice*, 7(8):957–968, 1999.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 1960.

[McD93] P. McDermid, J. y Rook. Software development process model. *Software Engineers Reference Book, CRC Press*, pages 5/26–15/28, 1993.

[McI76] M. D. McIlroy. Mass-produced software components. In *NATO Conference on Software Engineering*, pages 88–98. In J.M. Buxton, P. Naur, and B. Randell, editors, Software Engineering Concepts and Techniques, 1976.

[MK90] J. McGregor and T. Korson. Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, 33(9):40–60, 1990.

[MM95] Mili F. Mili, H. and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[MO94] J. Martin and J.J. Odell. *Object-oriented analysis and design*. Prentice Hall, 1994.

[Mon92] Puhr G. I. Monarchi, D. E. A research typology for object-oriented analysis and design. *Communicatios of the ACM*, 35(9):35–47, 1992.

[Moo89] D. Moon. The common lisp object-oriented programming language standard. *Acm Press Frontier Series. Object-oriented concepts, databases, and applications*, pages 48–78, 1989.

[Mor02] S. Moreno. *Le GRAFCET : Conception - Implantation dans les automates programmables Industriels : Bac STI - STS - IUT - IUFM - IUP ecoles d'ingenieurs*. Editions Casteilla, 2002.

[MR06] A. Morillas Raya. *Introducción al análisis de datos difusos*. 2006.

[MS03] Astrom K. Boyd S. Brockett R. Murray, R. and G. Stein. Future directions in control in an information-rich world. *IEEE Control Systems Magazine*, pages 20–33, 2003.

[Mur89] T. Murata. Petri nets: properties, analysis and applications. *Proceedings of IEEE 77*, 77(4):541–580, 1989.

[Nes99] N. Nessar. Cj international welcomes you to the soft logic open world. In *IEEE Colloquium on The Application of IEC 61131 to Industrial Control : Improve your bottom line through high valvue Industrial control systems*, pages 3/1–3/5, 1999.

[NX00] Brennan D.H. Norrie, Z.Z. and Yuefei Xu. A multi-level reconfiguration control for holonic plc. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 3, pages 1762–1767, Nashville, TN, 2000.

[Oga03] K. Ogata. *Ingeniería de Control moderna*. Prentice Hall, 2003.

[OJDN68] Bjorn Myhrhaug Ole-Johan Dahl and Kristen Nygaard. *SIMULA 67: common base language*. Norwegian Computing Center, 1968.

[OMG03] OMG. Unified modeling language specification. Technical report, 18 of March 2003.

[OSA96] OSACA. Open system architecture for controls within automation systems. Technical report, ESPRIT III Proyects EP6379 / EP9115, 1996.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, 1972.

[Pat08] S Patig. A practical guide to testing the under standability of notations. In *Proceedings of the 5th on Asia-Pacific Conference on Conceptual Modeling*, pages 49–58, Wollongong, 2008.

[Pet62] C. A Petri. *Kommunication mit Automaten*. PhD thesis, Universidad de Bonn, 1962.

[PF96] Calvo-Manzano J. Cervera .J. Piattini, M.G. and L. Fernandez. *Analysis and detailed design of computer management applications*. Ra-Ma, 1996.

[Pos01] E. Post. Advantages of using the object-oriented paradigm for designing and developing software. In *JADE Convention 2001*, Christchurch, New Zealand, 2001.

[PR94] A.J. Peralta and H. Rodriguez. *Enginyeria del software. Programacio orientada a objetes*. Editions UPC, 1994.

[Pre96] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill; 4th edition, 1996.

[RHF05] D.N. Ramos-Hernandez and P.J. Fleming. A novel object-oriented environment for distributed process control systems. *Control Engineering Practice*, 13:257–264, 2005.

[Rit78] D. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey, 1978.

[RL90] Blaha-M. Premerlani W. Eddy P. Rumbaugh, J. and J. Lorensen. *Object-oriented modelling and design*. Prentice-Hall, 1990.

[Rob92] P. Robinson. *Object-oriented design*. Chapman and Hall, 1992.

[Roy70] W. Royce. Managing the development of large software systems: Concepts and techniques. *Proceedings WESCON*, 1970.

[Sak92] M. Sakkinen. A critique of the inheritance principles of c++. *Computing Systems*, 5(1):69–110, 1992.

[Sav90] D. Savic. *Object-Oriented programming with Smalltalk-V.* Englewood Cliffs, NJ : Prentice Hall, Inc., 1990.

[Sch99] S.R. Schach. Classical and object-oriented software engineering with uml and java. In *4th edition, WCB*, McGraw Hill, 1999.

[Sel98] B. Selic. Using uml for modeling complex real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474, pages 250–260, 1998.

[SF07] Bonfe-M. Secchi, C. and C. Fantuzzi. On the use of uml for modeling mechatronic systems. *IEEE Transactions on Automation Science and Engineering*, 1(4):105–113, 2007.

[SM88] S. Shlaer and S.J. Mellor. *Object-Oriented systems analysis : Modeling the world in data.* Prentice Hall, 1988.

[Sou01] M. Sousa. Linux-based for industrial control. In *Embedded Linux Journal*, volume 3, 2001.

[Spi92] Spivey. *The Z Notation: A Reference Manual (Prentice-Hall International Series in Computer Science).* Prentice Hall, 1992.

[SS00] C. Shaeffer and J. Sievering. Holonic production and material flow in industry. In *Holonic manufacturing systems (HMS). International symposium*, 2000.

[Ste90] G.L. Jr. Steel. *Coomon Lisp, The Language.* Digital Press; 2 edition, 1990.

[Str84] B. Stroustrup. The c++ reference manual. Technical report, AT and T Bell labs computer science, January 1984.

[Str86] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1986.

[Str87] B. Stroustrup. Multiple inheritance for c++. In *EUUG Conference*, volume 2, 1987.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE software*, 5(3):10–20, 1988.

[Str89] B. Stroustrup. The evolution of c++: 1985-1989. *Computer Systems*, 2(3):13–52, 1989.

[Str90] B. Stroustrup. On language wars. *Hotline on Objetc-Oriented Technology*, 1(3), 1990.

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Professional, 1994.

[Tay97] D. Taylor. *Object-Oriented techonolgy.* Addison-Wesley Professional; 2nd edition, 1997.

[Thi01] L. Thiele. Discrete event system - introduction. Technical report, Swiss federal institute of technology, 2001.

[Thr02] K.C. Thramboudilis. Development of distributed industrial control applications : The corfu framework. In *4 IEEE international workshop on factoru communication systems*, Sweden, 2002.

[TT06] K.C. Thramboudilis and C. Tranoris. A tool supported engineering process for developing control applications. *Computers in Industry archive*, 57(5):462–472, 2006.

[VHK05] Witsch D. Vogel-Heuser, B. and U. Katzke. Automatic code generation from a uml model to iec 61131-3 and system configuration tools. In *International Conference on Control and Automation, ICCA '05.*, volume 2, pages 1034–1039, Budapest, 2005.

[VHK11] Braun S. Vogel-Heuser, B. and B. Kormann. Implementation and evaluation of uml as modeling notation in object oriented software engineering for machine and plant automation. In *18th IFAC World Congress*, pages 9151–9157, Milano, Italy, 2011.

[Wal91] J. Waldo. Controversy : The case for multiple inheritance in c++. *Computer Systems*, 4(2):157–171, 1991.

[Wal93] J. Waldo. *The Evolution of C ++: Language Design in the Marketplace of Ideas*. The MIT Press, 1993.

[WBW90] Wilkerson B. Wirfs-Brock, R. and L. Wierner. *Designing Objetc-Oriented Software*. Prentice Hall; 1st International edition, 1990.

[Wer09] Bernhard Werner. Object-oriented extensions for iec 61131-3. *IEEE Industrial Electronics*, 3(4):36–39, 2009.

[Wir76] N. Wirth. *Algorithms + data structures = programs*. Englewood Cliffs, New Jersey : Prentice-Hall, 1976.

[WK10] Ricken M. Witsch, D. and B. Kormann. Plc state charts: An approach to integrate uml state charts in open loop control engineering. In *Proceedings of the 8th IEEE Industrial Conference on Industrial Informatics*, pages 915–920, Osaka, 2010.

[WM99] Pircher P.A. Warsserman, A.I. and R.J. Muller. The object-oriented structured design notation for software design representation. *IEEE computer*, pages 50–62, 1999.

[WVH09] D. Witsch and B. Vogel-Heuser. Close integration between uml and iec 61131-3: New possibilities through object-oriented extensions. In *Proceedings of the 14th IEEE International Conference Emerging Technologies and Factory Automation*, pages 1–6, Mallorca, 2009.

[Zua00] C. Zuazo. Concept and implementation of a holonic machining centre. In *Holonic manufacturing systems (HMS). International symposium*, 2000.