Desarrollo de Sistemas Software Industriales Dirigido por Modelos: Aplicación a OPC UA e IEC 61131-3

Tesis Doctoral

José Miguel Gutiérrez Guerrero

Noviembre 2018

UNIVERSIDAD DE GRANADA

Programa oficial de Doctorado en Tecnologías de la información y la Comunicación

Directores: Juan Antonio Holgado Terriza, Miguel Damas Hermoso

Editor: Universidad de Granada. Tesis Doctorales
Autor: José Miguel Gutiérrez Guerrero
ISBN: 978-84-1306-049-1
URI: http://hdl.handle.net/10481/54427

Desarrollo de Sistemas Software Industriales Dirigido por Modelos: Aplicación a OPC UA e IEC 61131-3

Programa oficial de Doctorado en Tecnologías de la información y la Comunicación

Tesis Doctoral

por

José Miguel Gutiérrez Guerrero

Noviembre 2018

Universidad de Granada

Departamento de Lenguajes y Sistemas Informáticos

Grupo de Investigación de Sistemas Concurrentes

Director: Prof. Dr. Juan Antonio Holgado Terriza

Director: Prof. Dr. Miguel Damas Hermoso

ÍNDICE GENERAL

AGRADECIMIENTOS

Este trabajo no podría haberse realizado sin la ayuda y la colaboración de muchas personas, las cuales me han acompañado en este largo camino. En primer lugar me gustaría agradecer todo su apoyo a mi familia, a mis padres, por el esfuerzo que han echo y siguen haciendo, para que sus hijos tengan acceso a unos estudios universitarios, a mi mujer y mis hijos, los cuales han demostrado una paciencia infinita y que han vivido todo el desarrollo de esta tesis muy de cerca día tras día. También me gustaría agradecer el apoyo a mi compañero de trabajo Julio López, amigo de juventud y compañero de carrera, por sus discusiones sobre como llevar a la practica las ideas de esta tesis, a Jesús Muros y Sandra Rodríguez porque su ayuda especialmente en los primeros años de desarrollo de esta tesis, a Juan Antonio porque siempre ha estado ahí en los momentos mas difíciles de este trabajo y a Miguel por su aportaciones en la elaboración de este trabajo. También dejo un hueco para todos los compañeros de Abbott con los que trabajo día a día, por el interés mostrado durante estos años en este trabajo, y para todos los alumnos con los que he tenido la suerte de trabajar en este bonito mundo de los sistema industriales.

José Miguel Gutiérrez Guerrero
Noviembre 2018

1

INTRODUCCIÓN

"The mind, once stretched by a new idea, never returns to its original dimensions."
Ralph Waldo Emerson 1803-1882

1.1. SISTEMAS INDUSTRIALES AUTOMÁTICOS

El crecimiento tanto en tamaño como en complejidad de los procesos productivos en el marco de la industria ha desencadenado la necesidad de crear sistemas, formados por elementos electrónicos y con capacidad de cómputo, que controlen la maquinaria y el proceso industrial que se lleva a cabo. Estos nuevos sistemas son capaces de operar y controlar todo el proceso productivo por sí mismo, pero a su vez necesitan la supervisión y monitorización de los operarios, para garantizar el correcto funcionamiento de todo el sistema industrial.

Figura 1.1: Sala de control para de una central nuclear.

Conceptualmente un sistema industrial automatizado puede ser entendido como un sistema complejo que engloba un conjunto de elementos: la instru-

mentación industrial (sensores y actuadores), los dispositivos de control y supervisión, los sistemas de interconexión de todos estos elementos, los sistemas de transmisión/recopilación de datos y, por último, los sistemas software de tiempo real para supervisar y controlar todo el proceso productivo. Sobre este conjunto de elementos o sistemas debemos añadir otro conjunto de sistemas complementarios fundamentalmente de software para optimizar y explotar las capacidades del sistema industrial, tales como los sistemas software de supervisión, control y adquisición de datos, los sistemas de ejecución de recetas, los sistemas de historización de datos y los sistemas de ejecución de órdenes de producción, entre otros, que se ejecutan en ordenadores utilizando sistemas operativos de propósito general.

La incorporación de sistemas o elementos software como parte de un sistema industrial han experimentado igualmente una evolución, convirtiéndose en elementos fundamentales e independientes con valor propio, que participan en los procesos de transformación como las propias máquinas. Este nuevo escenario dista bastante de las primeras industrias que utilizaban únicamente máquinas simples que sustituían el esfuerzo humano y que eran manejadas por operarios. Actualmente, además de los sistemas mecánicos, hidráulicos y neumáticos tradicionales tenemos que sumarles los sistemas eléctricos y los sistemas software. Estos últimos están siendo una parte más importante del proceso productivo y su porcentaje en términos de coste, sobre el precio final de una maquina industrial ha pasado del 20 % al 40 %, y esta tendencia sigue al alza [1]. Por tanto, el desarrollo software de un sistema industrial supone una parte importante de su coste tanto en tiempo como en dinero. Además, un despliegue correcto en términos de funcionamiento, eficiencia y usabilidad, puede hacer que el sistema industrial sea aceptado y considerado un éxito o por el contrario un fracaso.

Tradicionalmente la programación y el desarrollo de software industrial no ha sido una preocupación importante en los primeros sistemas industriales al considerarse una tarea inexistente o de poca envergadura [2]. Sin embargo, a medida que el papel del software ha ido creciendo en importancia, el desarrollo del software se ha convertido en una tarea tan importante como la instalación mecánica y eléctrica.

En todo sistema industrial siempre se buscan 3 objetivos, reducir costes, au-

Figura 1.2: Sistemas complejo de tuberías de una refinería

mentar la calidad del producto y aumentar la productividad. La incorporación de sistemas software en los sistemas industriales ha permitido una mayor flexibilidad en la construcción de sistemas que cumplan dichos objetivos, ya que de esta forma es posible automatizar tareas que se realizaban de forma manual, de forma mucho más rápida, y una relación coste-eficiencia mucho más alta que cuando se diseñan equipos dedicados. La forma de abordar un sistema software industrial puede variar en función de varios factores:

- *El tamaño del proceso industrial*: podemos encontrarnos con procesos industriales en los que tengamos que gestionar decenas, cientos o miles de equipos, los cueles necesiten cientos, miles, decenas de miles o cientos de miles de señales para su funcionamiento. Es obvio que cuanto mayor sea el número de señales o datos a gestionar mayor será el esfuerzo a realizar en términos de programación/parametrización. Sin embargo, existen soluciones como los sistemas DCS (Distribute Control System) que pueden facilitarnos las tareas de programación y parametrización de señales, sobre todo en grandes sistemas industriales, aunque en sistemas de tamaños medios los costes de estos sistemas pueden suponer una desventaja a la hora de abordarlos y ser difíciles de justificar.

- *El nivel de automatización*: No es lo mismo desarrollar un sistema industrial en el que solo intervenga sistemas como SCADA (Supervisory Control And Data Acquisition) y PLC (Programmable Logic Controller), donde únicamente se realizan tareas de supervisión y monitorización, que un siste-

ma industrial en el que además se debe de planificar y gestionar ordenes de producción de forma automática, además de las tareas de supervisión y monitorización correspondientes. En este último caso intervienen elementos software adicionales como los sistemas BATCH o de ejecución por lotes y los sistemas MES (Manufacturing Executin System) para gestionar y lanzar ordenes de producción.

- *El nivel de integración*: El nivel de integración/colaboración entre los diferentes elementos del sistema industrial va a permitir gestionar y coordinar mejor los procesos industriales que se llevan a cabo en la planta. Sin embargo, puede haber casos en los que dicha integración no es factible porque cada máquina tenga un protocolo de comunicaciones distinto o porque directamente no exista forma de comunicarla.

En definitiva, los sistemas industriales requiere la intervención de elementos hardware (sensores, actuadores y dispositivos de control), redes de comunicaciones (buses de tiempo real, redes de comunicaciones de propósito general) y sistemas software que se pueden ejecutar en diferentes dispositivos (dispositivos industriales de control y ordenadores). Estos sistemas tienen que organizarse en arquitecturas que nos permitan interconectar todos estos elementos de forma segura, estructurándolos en distintos niveles de forma lógica en función de su papel dentro del sistema industrial.

En estas arquitecturas existen dos elementos, bajo nuestro punto de vista que son cruciales para el correcto funcionamiento del sistema software. El primero es el dispositivo industrial, PLCs, RTUs o controlador distribuido, que se encarga de enviar órdenes a los actuadores así como recibir información de los sensores. En este elemento suele residir toda la lógica y funcionalidad que hace que todo el sistema industrial o máquina, funcione de forma automática sin apenas intervención humana. El segundo de estos elementos son los servidores OPC (OLE for Process Control) . Estos elementos se encargan de realizar el intercambio de información entre los dispositivos industriales y los sistemas software que monitorizan y supervisan un proceso industrial. Antes de la aparición de este elemento, este intercambio era una tarea compleja en la que en la mayoría de los casos se necesitan drivers desarrollados específicamente para comunicarse con el o los dispositivos industriales.

Con la aparición de servidores OPC, se han ido definiendo estándares en el sector que han ido evolucionando con el paso del tiempo hasta la última especificación denominada OPC Arquitectura Unificada u OPC UA [3]. Esta última especificación del estándar tiene entre otras ventajas que proporciona un entorno multiplataforma, que ofrece un modelo de comunicación no dependiente de la tecnología subyacente así como un modelo de información que permite crear estructuras de datos y relaciones entre dichos datos. Así, el modelo de comunicaciones ofrece la posibilidad de utilizar un protocolo binario, basado en un identificador de recursos uniforme o URI, que puede ser utilizado incluso entre clientes y servidores que se encuentren en diferentes subredes.

Dicho el modelo de información nos permite modelar y organizar, tanto los datos que maneja el dispositivo industrial como su comportamiento con un enfoque orientado a objetos. Esto nos permite construir estructuras complejas de objetos, tanto desde el punto de vista sintáctico como semántico que se puede representar en un modelo abstracto del sistema. Esto supone un salto cualitativo y cuantitativo, en comparación con el modelo de información que ofrecía el OPC clásico, orientado únicamente en la organización de un conjunto de tipos de datos simples.

Sin embargo, el modelo de información de OPC UA presenta algunos desafíos importantes a los ingenieros de sistemas industriales. El primero de ellos, es el escaso soporte por parte de productos software comerciales, para crear y alojar modelos dentro de un servidor OPC UA; hoy en día la mayoría de los productos comerciales existentes están más centrados en explotar el modelo de comunicaciones de este estándar, que el modelo de información.

En segundo lugar, el modelo de información que presenta OPC UA es complejo porque contiene un conjunto amplio de conceptos y relaciones para la construcción de modelos de sistemas industriales, con lo que la construcción de un modelo puede ser una tarea laboriosa al no incluir una metodología de cómo deben construirse dichos modelos. Es decir, proporciona una infraestructura importante de elementos sintácticos que pueden ser utilizados para el modelado como ocurre en UML [4] , pero sin el apoyo documental necesario que aportan las metodologías que utilizan UML[5].

Los servidores OPC se encargan de conectar los dispositivos industriales con

los sistemas de más alto nivel, encargados de la supervisión del proceso industrial. Actúan por tanto como puentes entre los dispositivos industriales de control y los sistemas de alto nivel, estableciendo un intercambio de datos y ordenes entre ambos. Dentro del conjunto de dispositivos industriales se encuentran los PLCs o RTUs. Dichos dispositivos tienen alojados un conjunto de instrucciones o programa, el cual recoge toda la información de los diferentes sensores del sistema, toman ciertas decisiones en base a los datos capturados de dichos sensores y envían las ordenes necesarias a los diferentes controladores.

El desarrollo de estos programas, que controlan los procesos industriales, se ha tenido que desarrollar utilizando la plataforma de programación que el propio fabricante del dispositivo industrial proporciona. Sin embargo, con la liberación de la norma IEC 61131, concretamente en su parte tercera (IEC 61131-3), se recogen todos los aspectos y características que debe tener una plataforma de programación para PLCs. Esta norma pretende unificar las diferentes plataformas de programación para dispositivos industriales, mediante un conjunto de especificaciones y características, que deben cumplir dichas plataformas, permitiendo la reutilización de los programas desarrollados, simplemente compilando y descargando el código en dichos dispositivos. Sin embargo, en realidad los fabricantes de PLCs no suelen seguir todas las especificaciones y características, recogidas en la norma IEC 61131-3, por lo que los programas no pueden reutilizarse en distintas plataformas, requiriendo re-programaciones parciales o en algunos casos totales de los mismos. Otro problema que presenta dicha norma es que están muy centrada en paradigmas como la programación estructurada, basada en funciones o bloques de función, por lo que paradigmas más ampliamente utilizados en otro ámbitos como POO, MDD o SOA [1], son difícilmente aplicables, sobre todo en lo que a plataformas y framework de programación se refiere, ya que por lo general no tienen capacidad para soportar estos paradigmas.

Con estas limitaciones, sobre todo en el ámbito de la programación de dispositivos industriales, desarrollar un sistema software industrial requiere parametrizar, desplegar y programar varios sistemas software, que además deben de estar interconectados entre sí, para intercambiar información e interactuar. Además, la cantidad de información y señales que se manejan en estos entornos puede llegar a ser un problema, en términos de tamaño (cientos de miles de señales), organi-

zación y configuración (escalado y calibración E/S). Por ello intentar abordar el despliegue de un sistema software industrial, utilizando listados de señales mantenidos manualmente o mediante técnicas de programación tradicionales, como la programación estructurada, puede ser una tarea bastante ardua y compleja, y difícilmente mantenible.

En este escenario, se hacen necesarias la utilización de técnicas de desarrollo de software en los diferentes elementos de un sistema industrial, que nos faciliten todo el proceso de construcción, implantación y despliegue que garanticen propiedades flexibilidad, escalabilidad y mantenibilidad. Además, en muchos casos, pueden llegar a ser críticos tanto desde el punto de vista de la seguridad de los operarios que trabajan en la planta o desde el punto de vista económico. Por ese motivo el desarrollo de estos sistemas sin utilizar técnicas de ingeniería de software, que nos ayuden durante todo el proceso de desarrollo y despliegue, puede ser algo inviable. Afortunadamente la ingeniería del software es un paradigma ampliamente desarrollado y utilizado en las tecnologías de información [6], por lo que parece razonable trasladar estos paradigmas a los entornos industriales mediante herramientas, entornos de programación y framework que nos den soporte a dichas técnicas de ingeniería del software.

Entre los muchos de los paradigmas existentes en la ingeniería del software, hay un área dedicada al desarrollo de software dirigido por modelos, denominada Ingeniería dirigida por Modelos (o MDE, Model Driven Engineering, en ingles). Dentro de esta área, podemos destacar MDD (Model Driver Developement), que se basa en la utilización de modelos para todo el proceso de desarrollo de un sistema software, con objeto de industrializar el software.

Aunque la utilización de modelos no es algo nuevo, y está presente en disciplinas como la ingeniería civil, la arquitectura, la electrónica o la aeronáutica, se usa para la construcción de sistemas muy complejos, como por ejemplo, la construcción de aviones o cohetes en el caso de la aeronáutica. Generalmente, en todas ellas se incluyen notaciones, lenguajes y estructuras bien definidas con las que se pueden construir abstracciones de alto nivel. Existen además procesos y formalizaciones bien definidas con la capacidad de transformar esas abstracciones en resultados finales; en el caso de la aeronáutica en construcciones de aviones, de una forma fiable y estructurada.

Del mismo modo, podemos aplicar técnicas de modelado a nivel software. La utilización de modelos para el desarrollo de sistemas software presenta las siguientes ventajas:

- Documentan y recogen los aspectos del sistema software.
- Nos permite razonar sobre lo que se va a construir.
- Nos facilita la comunicación de ideas.
- Definen un lenguaje común que puede ser utilizado por el equipo para discutir todos los aspectos del sistema.
- Facilita la construcción del sistema.
- Podemos generar código a partir de modelos.

La utilización de MDD en entornos industriales puede ser muy interesante, ya que nos permite conceptualizar el sistema que queremos construir en base a un modelo abstracto, que debe satisfacer las necesidades del cliente. A partir de dicho modelo abstracto podemos llevar a cabo un proceso de desarrollo y despliegue para obtener los correspondientes elementos software integrantes del sistema. Si la construcción de dichos modelo se realiza en base a un lenguaje de modelado que regule desde el punto de vista sintáctico y semántico las normas o reglas de modelado que se deben satisfacer, esto nos permite homogeneizar y estandarizar el proceso de construcción de dichos sistemas software.

Pero para poder utilizar MDD en sistemas industriales en particular o en cualquier entorno en general, necesitamos herramientas y mecanismos que den soporte y obtener soluciones software finales, generalmente código, que puedan ser desplegados en plataformas software industriales. Algunos autores todavía van más lejos [7] y afirman que para que se pueda adoptar MDD como paradigma se debe de:

- Completar el marco teórico y disponer de herramientas usables y robustas.
- Disponer de personal capacitado y con conocimientos suficientes para utilizar este paradigma.

- Concienciar a las empresas sobre las ventajas que ofrece.

Por ese motivo en este trabajo de tesis estamos interesados en explorar como pueden aplicarse estas técnicas para facilitar el proceso de construcción de sistemas software en el contexto de la industria manteniendo las propiedades que se requieren en este dominio de aplicaciones restrictivo, conservador, exigente y crítico.

1.2. OBJETIVOS DE LA TESIS

El objetivo principal de esta tesis se centra en estudiar las posibilidades que ofrecen las herramientas y tecnologías de la Ingeniería Dirigida por Modelos en la construcción de software para sistemas industriales flexibles, adaptables, confiables y que sean fácilmente mantenibles.

Los objetivos específicos planteados en este trabajo de tesis son los siguientes:

- Explorar las capacidades de modelado que tiene OPC UA para estudiar el impacto que podría tener en el desarrollo de aplicaciones en el entorno industrial, y a partir del estudio identificar los elementos sintácticos y semánticos más adecuados para plantear una estrategia de desarrollo de aplicaciones orientadas a entornos industriales.

- Explorar las capacidades y posibilidades que ofrece el estándar IEC-61131-3 para el desarrollo de aplicaciones en dispositivos de control en la red de automatización de plantas industriales con un enfoque dirigido a objetos y en un mayor nivel con enfoques orientados a modelos, estudiando las estrategias de desarrollo comúnmente aceptadas con objeto de generalizarlo de una forma más abstracta.

- Estudiar la posible conceptualización de sistemas industriales con un enfoque orientado a modelos con objeto de establecer una estrategia de desarrollo que facilite tanto la identificación del problema como el planteamiento de una solución siguiendo, un método estructurado adaptable a las necesidades de los ingenieros que tienen que desarrollar aplicaciones en entornos industriales.

- Establecer un lenguaje de modelado que simplifique el proceso de desarrollo de aplicaciones industriales dentro del contexto de la Ingeniería Dirigida por Modelos, y que permita generar código para el despliegue de los componentes software necesarios para un funcionamiento en un entorno industrial.

- Establecer una metodología de desarrollo de sistemas industriales que tenga presente el proceso de desarrollo completo de una aplicación en el ámbito industrial teniendo en cuenta etapas de especificación, análisis, diseño, implementación o pruebas entre otros, utilizando un enfoque hacia buenas prácticas de ingeniería del software aplicado a sistemas industriales.

1.3. ESTRUCTURA DE LA TESIS

En el capítulo 2, se realiza un estudio general del concepto de sistema industrial, su evolución histórica y como han ido apareciendo nuevos elementos y sistemas dentro de los procesos productivos. También se exploran todos los elementos que pueden llegar a formar parte de un sistema industrial y las arquitecturas que se suelen utilizar para construir estructurar dichos elementos.

El capítulo 3 recoge un estudio detallado del estándar OPC gestionado y controlado por la Fundación OPC [8]. Se verán las diferentes especificaciones liberadas por esta fundación, centrándose sobre todo en la ultima especificación: OPC UA, haciendo un recorrido por las principales características de este novedoso estándar.

Seguidamente, en el capítulo 4 se explora la norma IEC 611313, prestando especial atención a la especificación tercera como norma para la programación de PLCs, analizando las capacidades y características de la misma.

En el capítulo 5 se introduce el paradigma de MDE (Ingeniería Dirigido por Modelos) y MDD (Desarrollo Dirigido por Modelos) para posteriormente presentar iMMAS como un lenguaje de modelado que facilita la utilización de MDD en el proceso de desarrollo de un sistema software industrial. Además también se presenta MOSIE como un perfil con un conjunto de paquetes que recogen conceptos especialmente diseñados para crear modelos independientes de la plataformas para entornos industriales.

El capítulo 6 se describen las reglas de transformaciones realizadas, entre los

conceptos de iMMAS/MOSIE y el modelo de información de OPC UA, facilitando la creación y el despliegue de modelos basados en iMMAS/MOSIE en servidores OPC UA.

En el capítulo 7 se realiza una descripción de las reglas de transformación realizadas entre iMMAS/MOSIE y la especificación IEC 61131-3. Además se recoge en un anexo todas las transformaciones especificas realizadas para la plataforma de programación de Siemens TIA Portal, ya que como se verá en este capítulo esta plataforma contempla parcialmente la especificación IEC 61131-3.

En el capítulo 8 se describe la metodología MIAS para el desarrollo de software en entornos industriales. Dicha metodología se ha creado basándose en la GAMP 5 (Good Automated Manufacturing Practice). Posteriormente dicha metodología será aplicada a un caso de estudio real en el capitulo 9 utilizando como sistema una cadena de montaje de piezas.

Por último en el capítulo 10 se presentan las conclusiones y los trabajos futuros derivados de esta tesis.

2

SISTEMAS INDUSTRIALES

"Quality means doing it right when no one is looking."
Henry Ford 1863-1947

2.1. SISTEMAS DE AUTOMATIZACIÓN INDUSTRIAL (IAS)

El concepto de sistemas industrial nació con la primera revolución industrial a mediados de la segunda mitad del siglo XVIII. Puede definirse como aquel sistema capaz de elaborar bienes de consumo a partir de materias primas [9] [10]. La transformación de estas materias primas en productos elaborados, recibe el nombre de proceso productivo o proceso industrial. En un proceso productivo participan principalmente tres actores:

- La maquinaria industrial.
- Las personas que operan esas máquinas (operarios).
- Un espacio donde desarrollar la actividad productiva.

En el nacimiento de la industria bastaba con estos elementos para elaborar coches, ropa, barcos, farolas, medicamentos y, en general, cualquier bien de consumo que necesitase ser procesado o refinado. Estos primeros entornos industriales se caracterizaban por presentar una fuerte interacción hombre máquina, ya que todos los trabajos se hacían de forma manual y era el operario quien controlaba y supervisaba el trabajo que realizaba la maquinaria. Por ello la maquinaria industrial, entendida como un conjunto de elementos que trabaja de forma integrada para realizar una tarea concreta, se consideraba una herramienta más,

Figura 2.1: Cadena de montaje Automóviles, 1820-1840.

como un martillo o una sierra, solo que su manejo presentaba cierto grado de complejidad [11].

Estos arcaicos sistemas industriales pretendían alcanzar fundamentalmente tres objetivos, reducir los costes de los bienes elaborados, aumentar su calidad y la productividad. Esto ha implicado que los sistemas industriales estén en una constante evolución desde su nacimiento, tratando de incorporar los avances técnicos que van surgiendo en otras áreas de ingeniería para optimizar el cumplimiento de dichos objetivos [12]. Ahora bien, dicha evolución se caracteriza por ser mucho más lenta que en otros ámbitos, como puede ser el militar o el de la electrónica de consumo, ya que en el ámbito industrial se buscan soluciones robustas y fiables, capaces de soportar procesos continuos que estén funcionando 24 horas al día, los 365 días del año. Por tanto, cualquier nuevo elemento o innovación que se realice debe ser capaz de trabajar en estas condiciones con garantías, ya que un elemento o sistema nuevo que funcione mal y produzca un defecto en el bien de consumo que se está elaborando, puede producir grandes pérdidas económicas y dañar la imagen de la empresa o la marca que hay detrás de ese producto.

La automatización de sistemas industriales contempla todas las etapas de un proceso productivo, la logística, el almacenamiento y la elaboración de bienes de consumo a partir de la transformación de las materias primas. Su aparición se debe a la evolución de la tecnología eléctrica, electrónica y sobre todo de la utilización de la informática como parte de un sistema industrial[2].

Un sistema industrial puede considerarse como un sistema complejo en el

que conviven de forma coordinada un conjunto de sistemas mecánicos, neumáticos o hidráulicos, todos ellos controlados por sistemas electrónicos. El trabajo conjunto de todos estos sistemas posibilita la transformación de las materias primas en bienes de consumo. Dado que dichos sistemas son susceptibles de presentar fallos o funcionamientos anómalos, si dichos fallos no son detectados rápidamente, pueden ocasionar grandes pérdidas económicas, tanto porque interrumpen el proceso productivo o por generar bienes con defectos. Si dichos bienes defectuosos no son detectados por los sistemas de control de calidad, pueden incluso ser perjudiciales para los clientes finales, con las consiguientes consecuencias negativas para la empresa como pérdidas económicas, desprestigio de las marcas, demandas millonarias por parte de los consumidores, retiradas de productos del mercado, etc.

De estos niveles de exigencia, en los que los fallos de los sistemas son inadmisibles, se deriva el carácter conservador y reticente a la hora de aplicar las nuevas tecnologías a los procesos industriales. Sin embargo, como ya se ha explicado, para poder alcanzar las cada vez mayores exigencias de los mercados, se hace necesaria una evolución de estos sistemas industriales, aplicando avances en aquellos elementos del sistema industrial que aporten un mayor rendimiento y rapidez (maquinas más rápidas, eficientes, seguras y respetuosas con el medio ambiente), así como nuevos elementos eléctricos-electrónicos más flexibles, fiables y con mayores capacidades de auto-diagnostico, integración y rendimiento.

Uno de los elementos que revolucionó los sistemas industriales a finales de los años 60 fue la adopción de los PLCs (Controladores Lógicos Programables) [2]. Este elemento se diseñó para sustituir los sistemas de control basados en relés e interruptores por un elemento electrónico más eficiente capaz de ejecutar programas. Los nuevos PLCs eran capaces de controlar o monitorizar el proceso productivo, o parte del mismo, sin necesidad de diseñar complejos circuitos electrónicos. Cualquier cambio en el proceso productivo implicaba únicamente adaptar el programa existente sin tener que rediseñar el circuito electrónico o tener que añadir más elementos electrónicos (relés o pulsadores).

La inclusión de los PLCs como elementos de control supuso un primer paso de una serie de avances simultáneos, como los sistemas HMI (Human Machine Interface) para la visualización de los procesos. Pronto aparecieron sistemas de

visualización basados en ordenadores mediante la utilización de software SCADA (Supervisory Control and Data Acquisition) [13]. La forma de interconectar los elementos de campo, sensores y actuadores, con los dispositivos industriales también evolucionó, pasando de sistemas cableados directamente a cada uno de dichos dispositivos hacia la utilización de buses de campo en los que todos los elementos comparten un mismo medio físico. Actualmente, se están utilizando estándares como TCP/IP para dar soporte a las comunicaciones en los sistemas industriales, materializándose así buses de campo como PROFINET, ETHERCAT (Industrial Ethetnet)[14].

En este escenario de evolución de los sistemas industriales, la introducción de ordenadores ha supuesto integrar las tecnologías de la información como parte del proceso productivo, las cuales se suma a los sistemas ya presentes en el dominio industrial, como los sistemas mecánicos, hidráulicos-neumáticos y electroelectrónicos. Esto ha hecho que el grado de complejidad de un sistema industrial aumente añadiendo una posible fuente más de fallos, aunque, por otro lado, ha supuesto un avance, ya que se pueden sustituir los sistemas HMI basados en botoneras o displays por sistemas software mucho más flexibles y escalables. Por ello actualmente las tecnologías de la información están ampliamente aceptadas en los entornos industriales, utilizando no solo sistemas software para la monitorización de los procesos industriales, sino que además existen numerosos sistemas TIC que hoy en día forman parte integral de un sistema industrial, como los sistemas de bases de datos para almacenar información del procesos productivo (sistemas históricos de datos), sistemas de ejecución de recetas (sistemas BATCH) [15], sistemas de ejecución y planificación de órdenes de planta (sistemas MES, Manufacturing Execution System)[16].

En definitiva se ha pasado de sistemas completamente manuales supervisados y controlados por operarios a pie de máquina, a sistemas en los que el control se realiza mediante elementos electrónicos que ejecutan un programa y la supervisión se realiza utilizando sistemas informáticos que utiliza un software específico para ello. De este modo, los operarios han pasado a interactuar con ordenadores en salas de control y no con máquinas.

Además, gracias a la utilización de ordenadores, podemos disponer de una serie de herramientas software, que nos permite planificar y organizar nuestros

procesos productivos de forma más eficiente y rápida (sistemas MES y BATCH), apoyando otras tareas como el mantenimiento de los equipos, por ejemplo, gracias al registro de las horas de trabajo de los mismos de forma automática. También se facilitan las tareas relativas al control de calidad, ya que podemos realizar la definición de los márgenes de trabajo de un proceso y su seguimiento de forma automática, supervisando si el proceso se encuentra dentro de especificaciones o fuera de ellas.

Todas estas posibilidades están ya siendo explotadas en mayor o menor medida en los sistemas software que dan soporte a los procesos industriales. Sin embargo, se están planteando nuevas formas de entender los procesos productivos, como la industria 4.0 o el Smart Factory, abriendo la puerta a la aplicación de nuevos paradigmas como el Big Data o el Internet de las Cosas (IoT) [17][18]. Sin embargo, estos nuevos enfoques están todavía en fase de desarrollo o de conceptualización, y por ahora no están teniendo un impacto importante en los entornos industriales reales, dado el carácter conservador del dominio en el que nos estamos moviendo.

En cambio, otros conceptos como la virtualización [19] y el Cloud Computing si se están asentando dentro de los sistemas industriales[20] [21], formando ya parte de ellos. Así, por ejemplo, el Software SCADA ya puede ser desplegado en entornos virtuales o entornos de nube [22] [23]. Estos nuevos avances de las TIC, facilitaran la evolución de los sistemas industriales mejorando el propósito principal de estos, que cómo indicábamos anteriormente, consiste en reducir los costes de los bienes elaborados, aumentando su calidad y cantidad.

2.2. ELEMENTOS DE UN SISTEMA DE CONTROL INDUSTRIAL

Los sistemas de Automatización industrial están formados por una serie de elementos que realizan o introducen cambios en el proceso productivo y por elementos que miden una serie de magnitudes físicas (temperatura, presión, caudal) del proceso industrial. A los primeros se les denomina actuadores y a los segundos sensores.

El control de un sistema industrial se puede realizar de dos formas:

- *Manual:* En este caso es el operario quien se encarga de realizar las tareas

de supervisión y control del sistema manualmente. Para ello utiliza las lecturas obtenidas de los sensores mediante algún tipo de interface HMI, display, reloj, o indicador luminoso. En función de la información que recibe el operario puede tomar ciertas decisiones que se traducen en operaciones para los diferentes actuadores del proceso industrial.

- *Automático:* En el caso del control automático existe algún tipo de sistema que toma las decisiones de forma autónoma sin la intervención del operario, basándose también en las lecturas de los distintos sensores que tenga el sistema. Decimos que existe algún tipo de sistema porque el control lo pueden realizar de forma autónoma tantos sistemas neumáticos, hidráulicos o eléctrico/electrónicos. Si bien es cierto que generalmente el control que se realiza con sistemas que no sean electrónicos suele ser muy sencillo y para aplicaciones muy específicas, en casi todos los procesos industriales controlados de forma automática se utilizan sistemas eléctricos/electrónicos.

La Real Academia de las Ciencias Exactas, Físicas y Naturales define la *automática* como el conjunto de métodos y procedimientos para la substitución del operario en tareas físicas y mentales previamente programada[24]. Por tanto, la aplicación de la *automática* al control de un proceso industrial se define como *automatización*. Por otra parte, podemos definir como proceso, a aquella parte del sistema industrial que a partir de la entrada de materias primas, energía e información, genera una salida, un producto manufacturado que se consigue mediante una transformación sujeta a perturbaciones del entorno.

Los procesos industriales pueden ser clasificados en:

- *Continuos:* Son aquellos procesos que mantiene una salida continua de producto o material. Como ejemplo de estos procesos tenemos la generación de electricidad y la destilación del petróleo.

- *Discretos:* Son aquellos que generan productos en forma de unidades como, por ejemplo, la industria del automóvil, la fabricación de electrodomésticos y maquinaria pesada.

- *Batch:* Son procesos en los que la fabricación del producto se realiza en forma de cantidades o lotes de material. Ejemplos de este tipo de procesos son los que se realizan en la industria alimentaria, farmacéutica y cosmética.

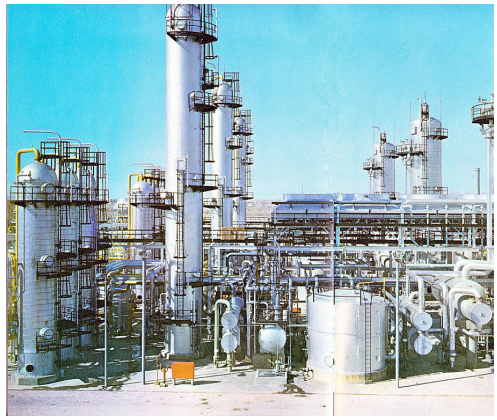


Figura 2.2: Parte de una Refinería de Petroleo.

2.2.1. SENSORES Y ACTUADORES

Los sensores y los actuadores son los elementos más próximos al proceso industrial y son los encargados de realizar las transformaciones de las materias primas para obtener el producto final. Dichas transformaciones se consiguen mediante la supervisión y regulación del proceso productivo y en caso necesario la introducción de perturbaciones sobre el mismo, de acuerdo al resultado que se desee conseguir; por ejemplo, aumentando la temperatura de cocción en el proceso de fermentación de la cerveza.

Está claro que para introducir estas perturbaciones necesitamos elementos que actúen durante el proceso de transformación. Dado que estas perturbaciones pueden hacer que el resultado final no sea el esperado, haciendo que el producto final tenga deficiencias o que no tenga la calidad esperada, es necesario aplicar estas perturbaciones supervisando en todo momento que está sucediendo en dicho proceso de trasformación. Son los actuadores los elementos que introducen estas perturbaciones, y los sensores los elementos que supervisan lo que está sucediendo.

Tanto actuadores como sensores tiene una parte eléctrica/electrónica que se encarga de hacer llegar las órdenes a los componentes que ejecutan estas órdenes en el caso de los actuadores, y de transformar las medidas de la magnitud física en señales eléctricas en el caso de los sensores. Dichos componentes son los que hacen que tanto actuadores como sensores se puedan conectar mediante algún tipo de cableado a los dispositivos industriales de nivel superior sirviendo como

interfaz de comunicaciones. Debido a la importancia que tienen estos elementos para un sistema industrial es interesante ver de qué tipos disponemos y de cómo podemos interactuar con ellos.

ACTUADORES

Estos elementos están diseñados y pensados para tareas muy concretas, como, por ejemplo, bombas neumáticas/hidráulicas positivas, bombas de vació neumático, motores eléctricos, válvulas de paso, válvulas modulantes etc.

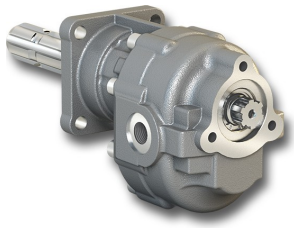


Figura 2.3: Motor Hidráulico.

Son elementos dotados con un comportamiento activo en el sistema, es decir están ”vivos” ya que pueden hacer cosas tan simples como parar y arrancar una cinta transportadora o algo más complejo como regular el paso de un fluido en función de una consigna.

En el caso de los actuadores podemos encontrarnos con una gran variedad de tipos de dispositivos, pero por lo general su control y manejo es más sencillo que el de los sensores. Por ejemplo, una gran numero de actuadores trabaja con dos estados marcho/paro por lo que con una señal que le indique estos dos estados podemos manejar estos actuadores 0-24V (0-1).

Existen otro tipo de actuadores que se controlan utilizando valores analógicos (válvulas modulantes, motores de velocidad variable, bombas, etc.). Para este tipo se utilizan señales analógicas con valores eléctricos de 0-5V, 0-10V, 0-20 mA y 4-20 mA. También pueden ser controlados mediante el envío de órdenes utilizando protocolos o comandos específicos, utilizando algún interfaz de comunicaciones, puerto serie, conexión de red, o en general buses de campo. También es posible controlarlos con una electrónica especialmente diseñada para este propósito. Podemos clasificar los actuadores en:

- *Actuadores digitales o todo/nada:* Son los más comunes en los sistemas de

control. En muchos casos, se incorporan dispositivos que permite que estos controladores sean controlados por equipos externos con órdenes marcha/paro. Los elementos más comunes son los relés y los contactores.

- *Actuadores analógicos o variables:* Para determinadas aplicaciones como dosificaciones de líquidos, movimientos de cintas transportadoras y en general para cualquier actuación en la que se necesite regular la velocidad del actuador, se necesitan actuadores que respondan a salidas típicas de control con valores variables.

SENSORES

Los elementos que se encargan de medir y monitorizar el proceso productivo reciben el nombre de sensores o instrumentación de campo. Estos elementos "vigilan" dicho proceso, midiendo magnitudes físicas críticas que nos indican si se están realizando bien las transformaciones de las materias primas. Son los "ojos" de todo el proceso de transformación, informando en todo momento de lo que está ocurriendo. Están pensados y diseñados para medir valores de temperatura, presión, caudal, humedad, densidad de un fluido, etc.

Son tan importantes como los actuadores, ya que si realizan mediciones incorrectas durante el proceso de fabricación pueden hacer que los actuadores trabajen de forma inadecuada, pudiendo tener gran impacto en el producto final. Los operarios podrían tomar decisiones erróneas, desechando incluso producciones, si dichos elementos no miden variables dentro los parámetros correctos, cuando en realidad si han estado dentro de los parámetros, por ejemplo. Por tanto, el simple hecho de que un elemento de medida no esté funcionando correctamente puede hacernos tomar decisiones incorrectas.

Existen una gran cantidad de tipos de sensores para realizar medidas de todo tipo y por tanto, se pueden clasificar de muchas formas distintas:

- **Según el tipo de salida:**
  - *Analógicos:* Transforman la medida de la magnitud física en una salida con valores en niveles dentro de un rango. Por ejemplo, un sensor de temperatura proporciona valores de 0º a +150º con salida 4-20ma.
  - *Binarios:* Son sensores con dos valores 0,1, como, por ejemplo un detector de nivel vacío/lleno.

  - *Digitales:* Utilizan algún protocolo de comunicaciones específico como I2C, SPI, UART, etc., para obtener una medida de la magnitud física.

- **Según su estructura interna:**

  - *Pasivos:* Son sensores que no necesitan alimentación eléctrica. Normalmente son resistencia que varían con el cambio de alguna magnitud, como la temperatura o la luz.
  - *Activos:* Debido a que utilizan circuitos electrónicos para realizar las transformaciones de los valores medidos, estos últimos suelen necesitar una fuente de alimentación externa.

- **Según la naturaleza de la medida realizada:** esta clasificación es muy amplia pero podemos hacer 3 grupos, donde aunque no estén todos los tipos de sensores se recogen en ellos la gran mayoría:

  - *Pasivos Mecánicos:* Son utilizados para detectar accionamientos mecánicos, contactos, aceleración, etc.
  - *Ambientales:* Medidas de temperatura, humedad, pluviometría, velocidad del viento, etc.
  - *Químicos:* Niveles de CO2, niveles de oxígeno, PH, contaminación en el aire, etc.

Una vez que hemos agrupado los sensores en tipos podemos analizar cómo pueden y en qué tipos de señales eléctricas transforman las medidas que realizan:

- Salida analógica que varía entre 0 y 10Vcc.
- Salida analógica de bucle de corriente 0 y 20mA.
- Salida analógica de bucle de corriente 4 y 20mA.
- Valores de resistencia variable.
- Niveles lógicos 0 o 1.
- Salida de dos niveles 0V o 24V (cerrado/abierto).

- Codificaciones digitales especiales del fabricante, para ello es necesario consultar la información que proporciona dicho fabricante.

Se pueden analizar más características de los sensores, como son el rango y la resolución de medida, precisión, etc. pero desde el punto de vista de un sistema de control industrial los aspectos más importantes son el tipo de sensor y la forma en cómo envía la información. El resto de características dependen más de la aplicación que se le quiera dar al sensor dentro del proceso industrial y de las necesidades que este proceso requiera.

Tanto para los actuadores como los sensores tenemos una taxonomía común en función del tipo señal que utilizan digital o analógica. Los actuadores/sensores digitales son los más sencillos y fáciles de manejar porque solo tienen dos estados y con ellos se pueden realizar una gran cantidad de operaciones dentro de un proceso productivo. Por otra parte, son los que mejor se comportan frente a ruidos electromagnéticos y condiciones extremas de temperatura y humedad. El segundo tipo de actuadores/sensores en general se utilizan para controlar y regular las tareas más delicadas que necesitan cierto grado de precisión tanto en la medida como en la actuación. Este tipo de elementos suelen ser los más delicados y los que mayor mantenimiento (calibraciones, revisiones) presentan. Además son elementos sensibles a las perturbaciones electromagnéticas y pueden presentar comportamientos no deseados con temperaturas elevadas o ambientes con mucha humedad, por lo que se suelen proteger con encapsulados acordes a los ambientes en donde van a desarrollar su trabajo.

2.2.2. DISPOSITIVOS DE CONTROL

Todos los actuadores y sensores de un sistema industrial deben de estar gestionados por un sistema de control que procese la información recibida de los sensores, tome ciertas decisiones y genere un conjunto de órdenes para los diferentes actuadores. Estos sistemas de control han ido evolucionando con el paso del tiempo existiendo un gran número de distinto de dispositivos de control. Estos dispositivos son de gran importancia, ya que de sus características y prestaciones depende en gran medida el funcionamiento del sistema industrial. Los tres tipos de dispositivos industriales más utilizados para realizar tareas de control son: PLCs (Programmable Logic Controller, Controlador Lógico Programable), RTUs

(Remote Terminal Unit, Unidad Terminal Remota) y los sistemas basados en controladores distribuidos o DSC (Distributed Control System). Además de estos dispositivos también podemos encontrar IEDs (Intelligent Electronic Device, Dispositivo Electrónico Inteligente) y los PACs (Programmable Automation Controller, Controlador Automático Programable) y los sistemas de gestión y control para los diferentes tipos de robots industriales existentes hoy en día.

CONTROLADORES LÓGICOS PROGRAMABLES (PLCS)

Los controladores lógicos programables aparecieron como resultado de la búsqueda de un elemento Eléctrico/electrónico, con capacidad para ejecutar programas, que sustituyera a los sistemas en electrónica dedicada o especifica para una aplicación concreta. Fue la empresa General Motors [2] la que impulsó esta iniciativa y la compañía Modicon (Modular Digital Controller) la que desarrollo el primer PLC. La aparición de este nuevo dispositivo electrónico aportó flexibilidad y escalabilidad a los sistemas industriales, ya que por un lado podían ser utilizados para un gran número de aplicaciones industriales, y no solo para un propósito concreto como ocurría con la electrónica dedicada, y por otro lado, cualquier cambio en un sistema industrial se podía gestionar tan solo reescribiendo el programa de control que ejecutaba dicho dispositivo.

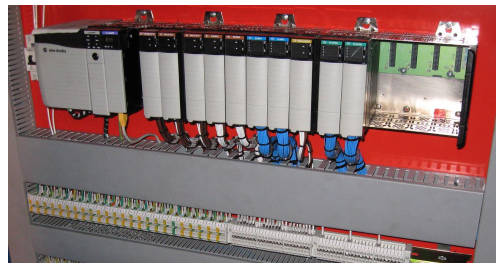


Figura 2.4: Elementos de un PLC.

Un controlador lógico programable está formado por una CPU, como se puede ver en la figura 2.4 y un sistema para recoger las entradas y salidas, tanto analógicas como digitales, de los sensores y actuadores del sistema industrial. La electrónica encargada de recoger las entradas y las salidas suele ser fácilmente ampliable. El sistema para recoger las entradas y salidas está compuesto por una serie de tarjetas que se añaden a la CPU. Dichas tarjetas pueden conectarse a diferentes buses de campo, proporcionando interconexión entre los distintos dis-

positivos y PLCs. Normalmente un PLC hace de maestro del bus y este arbitra dicho bus, leyendo información de los dispositivos del bus o enviando órdenes a los diferentes dispositivos que se conectan a dicho bus. En un bus de campo pueden convivir varios dispositivos y PLCs, aunque por defecto solo puede existir un maestro del bus, normalmente un PLC, mientras que el resto de dispositivos y PLCs mantienen un papel de esclavos dentro de dicho bus como se puede ver en la figura 2.5.

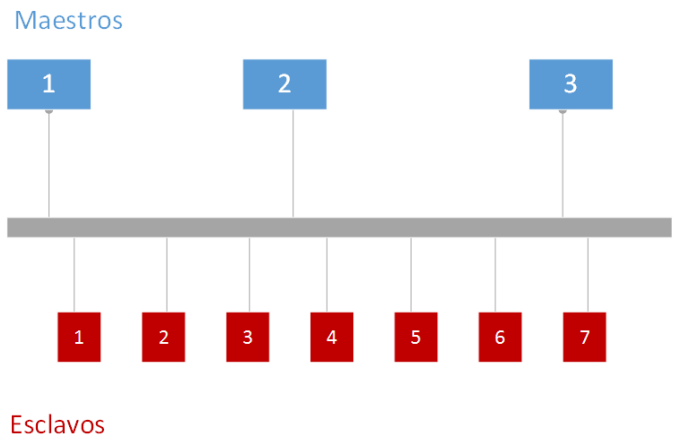


Figura 2.5: Dispositivos industriales conectados a un bus de campo.

Básicamente los PLCs ejecutan los programas de forma cíclica. Cada ciclo comienza con una serie de operaciones internas de diagnostico y control de la memoria, esta primera parte de la ejecución del ciclo es muy rápida, a continuación se realiza una lectura de las señales de entrada (señales recibidas de los sensores), las cuales se convierte a señales binarios o digitales, para posteriormente ser enviadas a la CPU y almacenadas como datos en la memoria. Después se ejecutan las instrucciones albergadas en el programa de la CPU de forma secuencial. Después de la ejecución de la última parte del programa se suelen generar nuevas señales de salida (binaria, digital o analógica), las cuales se almacén también en memoria, para posteriormente ser convertidas de forma apropiada a señales que se puedan enviar a los diferentes actuadores. En la figura 2.6 podemos ver un ejemplo de la implementación de un ciclo

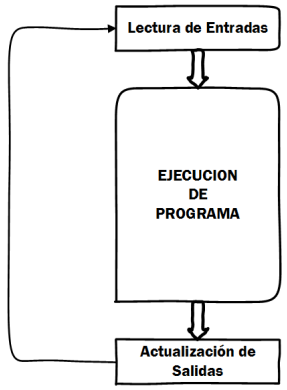


Figura 2.6: Ciclo de Ejecución de un PLC.

UNIDADES DE TERMINAL REMOTO (RTUS)

Una RTU es un equipo con múltiples capacidades de comunicación. Su capacidad de procesamiento no es excesivamente alta. Por tanto no es útil para el control de procesos que requieran tomar decisiones rápidas, pero es capaz de trabajar con varios protocolos a la vez, lo que le permite comunicarse con muchos dispositivos, incluso de múltiples fabricantes, para intercambiar información entre ellos.

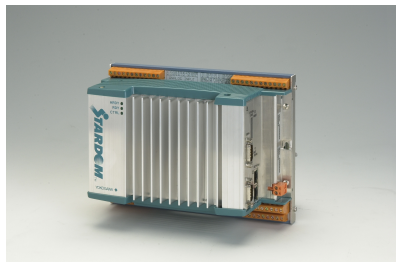


Figura 2.7: RTU Remote Terminal Unit.

En el diseño de un sistema industrial podríamos pensar en utilizar PLCs para realizar el control del proceso industrial y RTUs para recoger información del mismo. Sin embargo, la inserción de demasiados elementos en un sistema de control no es una buena práctica, ya que al aumentar el número de elementos la probabilidad de fallo también es mayor y, consecuentemente, aumentan también las necesidades de mantenimiento y, por tanto, el coste operativo. Este es uno de los principales motivos que ha provocado que tanto PLCs como RTUs sean cada vez más similares y estén evolucionando hacia un modelo que integra en un único

equipo sus respectivas características.

En la actualidad, los PLC son capaces de soportar cada vez más protocolos, incluso de alto nivel, lo que les permite comunicarse con más dispositivos industriales. Las RTU, por otra parte, han mejorado en su velocidad utilizando mejores procesadores, lo que les permite interactuar de forma más eficiente con el proceso industrial.

SISTEMAS DE CONTROL DISTRIBUIDO (DCS)

Los sistemas de control distribuido están basados en la utilización de diversos nodos distribuidos, cada uno de ellos con capacidad de proceso y conectado a un conjunto de sensores y actuadores. La característica principal de estos sistemas es que el control se lleva a cabo de forma coordinada entre todos los nodos que forma el sistema distribuido. En estos sistemas las redes de comunicaciones que interconectan todos estos nodos son cruciales, ya que un mal funcionamiento o una caída de la red pueden colapsar el sistema.

Los DCS fueron concebidos como una alternativa a los sistemas centralizados aportando una seria de ventajas como un menor tiempo de diseño, ya que es más fácil dividir un problema complejo en pequeñas soluciones (nodos) y coordinarlos, siguiendo una filosofía de divide y vencerás. Además también se simplifica el cableado de entradas y salidas, ya que podemos tener el nodo de control todo lo cerca que se necesite de los sensores/actuadores, y después conectar este nodo en red con el resto de nodos. La industria del automóvil es el sector donde más se utilizan este tipo de sistemas, ya que es un sector con grandes exigencias de adaptabilidad, escalabilidad y flexibilidad.

Hoy en día la línea que divide los sistemas de control distribuido y los sistemas basados en PLC es casi inexistente, debido a los avances tanto en el hardware como en el software. Por ejemplo podemos utilizar estándares de programación como el IEC 1131-3 [25], pensados originariamente para PLC en sistemas DCS. Además podemos utilizar PLC para crear sistemas DCS, haciendo que estos se comuniquen e intercambien datos y se coordinen para realizar ciertas tareas de control entre distintas partes de una fábrica.

Otro aspecto que ha ido evolucionado con los nuevos dispositivos de control han sido las características de seguridad que proporcionan. De hecho, además de mejorar las capacidades computacionales aumentando la velocidad, el poder de

procesamiento, o la memoria, los nuevos diseños también permiten incorporar mecanismos de seguridad implementados a bajo nivel, como las memorias criptográficas para proteger contraseñas.

2.2.3. SISTEMAS DE INTERACCIÓN HOMBRE MAQUINA HMI

Todo Sistema industrial por muy automatizado que este necesita ser supervisado y monitorizado por un ser humano. A veces esta interacción es algo tan simple como pulsar un botón de inicio y otras veces es necesaria una continua supervisión y parametrización del proceso industrial.

Los elementos que permiten realizar estas actuaciones, por parte de los operarios, en los sistemas industriales también han ido evolucionando con el paso del tiempo. En un principio consistían en palancas y botoneras, mientras que hoy en día se utilizan ordenadores con un software específico para realizar estas tareas. Estos sistemas permiten a un operario modificar el proceso industrial o simplemente ver el estado en que se encuentra dicho proceso. Reciben el nombre de sistemas HMI o interfaces hombre máquina (Human Machine Interface, en inglés) y pueden ser de dos tipos: tradicionales o basados en computadora. En el primer caso se utilizan elementos físicos (botones, palancas o indicadores luminosos) para ayudar al operario a monitorizar y supervisar el proceso industrial, mientras en el segundo caso se utiliza un software específico. Dentro de los HMI de tipo software podemos encontrar:

- **Sistemas Software a Medida.** Son sistemas que han sido desarrollados de acuerdo a las necesidades específicas de un determinado proceso industrial mediante algún lenguaje de programación que facilite el desarrollo de interfaces gráficos como Visual Basic o java. Estos sistemas fueron muy utilizados en las primeros HMI basados en PC en la década de los 80 y los 90.

- **Sistemas SCADA.** Los sistemas SCADA (Supervisory Control and Data Adquisición) son paquetes software de terceros, que facilita el desarrollo de un sistema HMI, proporcionando herramientas o elementos software ya elaborados y probados que permiten recoger, procesar y almacenar la información recibida del proceso industrial. También facilitan realizar la supervisión de todas las variables de control en una pantalla de ordenador, así

como interactuar con el proceso industrial mediante el envío de órdenes a los diferentes actuadores del mismo [13]. Con el paso del tiempo han aparecido multitud de soluciones SCADA comerciales de la mano de fabricantes como Siemens, General Electric, Wonderware y RockWell, entre otros.

Los sistemas HMI software más extendidos hoy en día son los basados en software SCADA comercial, debido fundamentalmente a que son sistemas ya probados y testados. Además tienen un fabricante detrás que da soporte y proporciona actualizaciones de dichos sistemas. La utilización de software SCADA como software HMI acorta los tiempos de desarrollo e integración, abaratando así los costes del sistema industrial. Generalmente el software SCADA ofrece entre otras las siguientes funcionalidades:

- Facilidad para realizar cálculos complejos e incluso programar algunas tareas y funcionalidades específicas.
- Gestión de alarmas e incidencias.
- Manejo eficiente de bases de datos para el almacenamiento de la información de la planta.
- Monitorización, supervisión y gestión de todos los sistemas remotos, pudiendo analizar en todo momento su estado, así como activarlos o desactivarlos.
- Un sistema HMI amigable para el usuario final.
- Capacidad para generar reportes e informes comparando datos.

Además de estas funcionalidades los sistemas SCADA suelen presentar características que ayudan al desarrollo/programación:

- Creación de pantallas gráficas con sinópticos de planta, facilitando la animación de los mismos.
- Amplias librerías de objetos para representar elementos como motores, válvulas, tanques, tuberías, etc...

- Acceso a drivers de comunicación con los dispositivos industriales integrados como: clientes OPC, drivers S7, Modbus, OMROM, Rockwell etc...

Los sistemas SCADA o software HMI están formados por varios módulos o programas independientes que proporcionan distintas funcionalidades, como el diseño y la configuración del sistema, runtime de ejecución, base de datos de variables de proceso a monitorizar y drivers de comunicaciones. Todos estos módulos unidos forman el sistema HMI software y hacen que todo funcione correctamente, aunque el usuario/operario solo percibe la visualización del proceso en una pantalla de ordenador [26].

Los sistemas HMI de hoy en día, sobre todo para procesos complejos, están basados en ordenadores o estaciones de trabajo que utilizan software SCADA. Estos ordenadores se conectan a los dispositivos industriales mediante algún tipo de conexionado o red de comunicaciones. La forma y el modo de conexión entre estos ordenadores y los dispositivos industriales también han experimentado una evolución con el paso del tiempo, dando lugar a diferentes tipos de arquitecturas jerárquicas dentro de los sistemas industriales en función de cómo se interconectan los sistemas software HMI y los dispositivos industriales. En muchos casos, los sistemas HMI software han originado problemas por la complejidad que supone llevar las señales que gestiona el dispositivo industrial a un software alojado en un PC tanto para su lectura (Supervisión y Monitorización) como para su escritura (Control y Actuación).

2.2.4. SISTEMAS DE INTERCONEXIÓN

Las primeras formas de interconectar un actuador o un sensor con el dispositivo industrial consistían en cablear dicho elemento directamente a una entrada o salida de dicho dispositivo. Aunque para ciertas aplicaciones industriales de pequeña escala o en sistemas donde se requiere el control centralizado de máquinas aisladas podía ser interesante este tipo de conexionado simple, no ocurría así en grandes instalaciones debido a que el aumento de cables para conexionar cada actuador o sensor hacían bastante difícil su mantenimiento. En esta línea podemos distinguir tres formas de realizar estas interconexiones:

- Cableado clásico: Consiste en conectar hilo a hilo los diferentes elementos de nuestro sistema industrial.

- Sistema pre-cableado: En esta configuración se utilizan unos módulos adicionales con una alta densidad de entradas y salidas, las cuales se conectan al dispositivo de control mediante una manguera de cables y no a las tarjetas de entradas y salidas como en el cableado clásico.

- Entradas y salidas distribuidas o periferia descentralizada: Esta forma de realizar el interconexionado entre elementos industriales reduce la cantidad de cables, sobre todo en longitud. Para ello se coloca una caja cerca del proceso a controlar con el cableado necesario para conectar tanto sensores como actuadores. Posteriormente, se comunica dicha caja de conexionado con el dispositivo de control mediante un módulo de comunicaciones. De esta manera se consigue que los cables de los sensores/actuadores sean más cortos, lo que disminuye las posibles perturbaciones y evita las caídas de tensión.

REDES DE COMUNICACIONES DE TIEMPO REAL, BUSES DE CAMPO

A finales de los 80 y sobre todo durante los 90 aparecen en el mercado nuevas opciones de comunicación en el marco de los sistemas industriales, concretamente los buses de campo [14]. Estos buses permiten conectar sensores y actuadores al dispositivo de control industrial con un solo medio de comunicaciones. Las modificaciones y ampliaciones de estas instalaciones se pueden realizar fácilmente sólo con ampliar el cable del bus y conectar los nuevos componentes, con lo que se consigue una mayor escalabilidad y un abaratamiento de los costes frente a las formas de cableado existentes hasta el momento.

Esta aproximación se inspiró en las redes de comunicaciones de los sistemas informáticos en los que existía un medio común, generalmente un cable de cobre, que era compartido por todos los elementos que se conectaban a él para intercambiar información utilizando algún tipo de protocolo para gestionar dichos intercambios. A estos nuevos sistemas de comunicaciones se le añadieron restricciones de tiempo real. Es decir, estos buses debían de asegurar que la información llegaba a su destino en plazos de tiempo estricto. Para garantizar estas restricciones en casi todos los protocolos industriales se utiliza una arquitectura con esquema Maestro/Esclavo, en el que hay un elemento que hace de maestro del bus arbitrando sobre los diferentes esclavos, estableciendo los turnos para realizar los

intercambios de información con un protocolo de comunicaciones determinista y de tiempo real.

Este tipo de comunicación permite ir más allá que la simple conexión con actuadores o sensores de tipo digital o de tipo analógico, ya que nos permite conectar dispositivos más complejos como controladores de robots, terminales de visualización, ordenadores industriales o controladores PID, entre otros. El intercambio de información entre los dispositivos se realiza utilizando pequeños paquetes de información con unas velocidades de intercambio del orden de Kbps o Mbps.

Este tipo de comunicación presenta varias ventajas respecto a los sistemas cableados. La primera ventaja tiene que ver con la reducción de costes en las instalaciones, ya que el cableado es más sencillo y no es necesario llevar todos los cables al dispositivo industrial, tan solo necesitamos ampliar el bus. En segundo lugar, son sistemas fácilmente ampliables, ya que para añadir un elemento solo hay que conectarlo al bus e informar al maestro de ese nuevo elemento. Por último, ofrecen una mayor precisión, posibilitando la realización de diagnósticos de los instrumentos de campo así como su calibración de forma remota, facilitando incluso la conectividad entre componentes de distintos fabricantes.

Pero no todo son ventajas, también presenta algunos inconvenientes. El primero es que los elementos, actuadores/sensores, tienen que soportar dicho protocolo por lo que la electrónica se hace más compleja. En segundo lugar, fue necesario estandarizar los protocolos de comunicación y que éstos no fuesen propietarios de un único fabricante para no generar un problema de interoperabilidad entre dispositivos electrónicos y así poder abaratar realmente los costes de las instalaciones.

Afortunadamente hoy en día existen estándares para buses de campo ampliamente aceptados por casi cualquier fabricante como, por ejemplo, el industrial Ethernet que ha dado origen a protocolos como Profinet o Ethercat, o protocolos como Profibus o Modbus que están tan ampliamente extendidos que cualquier fabricante tiene dispositivos que los soportan.

El medio físico para conectar los dispositivos también ha ido evolucionando, siendo posible utilizar además del cobre, fibra óptica y medios inalámbricos. Estos últimos todavía no han conseguido una aceptación tan amplia como los

medios tradicionales basados en cobre o en fibra óptica. No obstante, en aplicaciones industriales en espacios abiertos y en las que no existen interferencias electro-magnéticas, como por ejemplo, en la industria de las refinerías de petróleo o en los sistemas de regadío automáticos, se están utilizando buses inalámbricos con los que se obtiene mejoras tanto en el coste de la instalación, al no tener que cablear los elementos, como en el rendimiento y la tolerancia a fallos, generando redes de sensores con varios caminos para transmitir la información [27].

Como buses de campo mas utilizados podemos destacar: Modbus [28], AS-i [29], Profibus (Process Field Bus) [30] e Industrial Ethernet [31]. De todos estos tipos de buses de campo Industrial Ethernet es el que esta teniendo una mayor presencia en la industria, esto ha llevado a que otros tipos de buses de campo hayan evolucionado hacia la utilización de Industrial Ethernet, dando lugar a toda una nueva generación de protocolos industrial: EtherCAT, EtherNet/IP, Profinet, Powerlink, SERCOS III, CC-Link IE, y Modbus TCP.

REDES DE COMUNICACIONES DE PROPÓSITO GENERAL

Las redes de propósito general en los sistemas industriales juegan un papel distinto al de los buses de campo, ya que se utilizan como medio de interconexión entre sistemas TIC que son utilizados dentro de los sistemas industriales, los cuales desarrollan alguna tarea o papel dentro del sistema industrial como, por ejemplo, los sistemas SCADA para la monitorización y la supervisión del proceso industrial, o los sistemas BATCH para la ejecución de recetas. Este tipo de redes de comunicaciones se utilizan en los entornos industriales para interconectar elementos y sistemas en los que solo se necesita realizar intercambios de información y no se requieren características adicionales como el cumplimiento de restricciones de tiempo real, siendo ampliamente utilizadas, por ejemplo, para interconectar dispositivos industriales, como PLCs, con sistemas de monitorización basados en SCADAS.

El estándar más ampliamente difundido a la hora de interconectar dispositivos en el mundo de las TIC es el Ethernet al que también se le conoce como IEEE 802.3. Este estándar utiliza CSMA/CD (Acceso múltiple por Detección de Portadora con Detector de Colisiones) para arbitrar el acceso al medio de los dispositivos a la hora de transmitir. La utilización de esta técnica mejora el rendimiento

de la red, ya que se dispone de la capacidad de detectar errores como resultado de la transmisión de información por parte de varios elementos de la red a la vez. Este estándar especifica desde las características de los cables que se deben de utilizar para establecer las conexiones hasta los formatos de las tramas de datos a transmitir. Podemos encontrar numerosa literatura sobre el funcionamiento de este estándar y sus especificaciones [31][32].

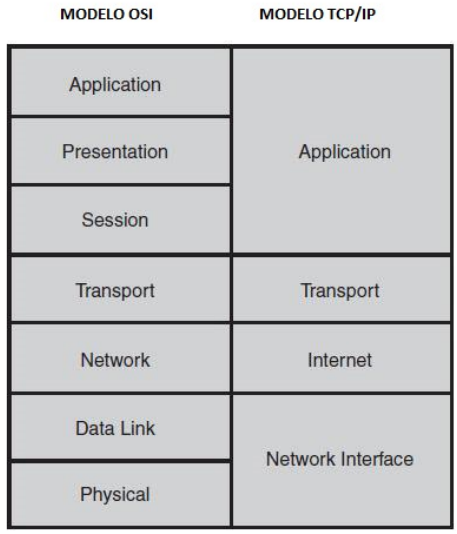


Figura 2.8: Correspondencia entre el modelo ISO y el TCP/IP.

Sobre este estándar se asienta el protocolo TCP/IP, el cual está alineado con el modelo de comunicaciones de red definido por ISO (International Organization for Standardization)[33]. El modelo ISO describe una forma de organizar diferentes aspectos de una red de comunicaciones en una serie de capas. Dichas capas engloban especificaciones diferentes. Así los aspectos eléctricos y de acceso medio se recogen en la capa del medio físico mientras que la forma de realizar el direccionamiento de los elementos que están en la red se recogen en la capa de red. Además ISO también establece una forma de comunicar procesos que pertenecen a diferentes capas, siendo las capas inferiores las que prestan servicio a las capas superiores. La figura 2.8 muestra las siete capas del modelo de referencia OSI y su correspondencia general con las capas del conjunto de protocolos TCP/IP.

La utilización del estándar Ethernet y el conjunto de protocolos TCP/IP en entornos industriales ha facilitado por un lado la interconexión entre los distintos elementos que lo forman, y por otro lado el intercambio de información entre ellos. Además ha permitido que protocolos como HTTP se pueda utilizar también para acceder a dispositivos industriales y poder así ver las variables del proceso y parametrizarlos. Hoy en día existen sensores que se pueden parametrizar accediendo a un conjunto de páginas web alojadas en el propio dispositivo a partir de su dirección IP. También podemos encontrar sistemas que disponen de servicios FTP para acceder a la información que manejan.

Gracias a estas aportaciones, propiciadas por el uso de este estándar, la separación entre el mundo de las TIC y los sistemas industriales se ha reducido, llegando incluso a ser difícil diferenciarlos.

2.3. CONVERGENCIA ENTRE LAS TIC Y LOS IAS

La utilización de tecnologías y elementos propios del dominio de las TIC en entornos industriales han hecho que aparezcan sistemas software específicos para este tipo de entornos como los sistemas MES (Manufacturing Executing System, en inglés). Además otros sistemas como los ERP (Enterprise Resource Planning) o las bases de datos relacionales, que se utilizan fuera del contexto industrial, también están empezando a formar parte de los sistemas industriales, aunque de forma más indirecta.

Los sistemas MES se encargan de introducir las órdenes de fabricación en los sistemas de control que se encuentran más cerca del proceso industrial. Estas órdenes de fabricación derivan de pedidos introducidos y planificados en los sistemas de ERP de la compañía. Por lo que en un escenario ideal en el que un pedido se da de alta en el sistema de ERP, este sistema gestiona dicho pedido tanto a nivel de planificación como de reserva de materia prima necesaria para su fabricación y lo introduce en el sistema MES. Este sistema se encarga de lanzar las órdenes al sistema de control para que dicho pedido sea fabricado.

2.3.1. EL SISTEMA ERP

Un ERP es un conjunto de herramientas software que recogen todas las necesidades de una compañía, gestión de pedidos, control de inventario, facturación,

contabilidad, nominas, etc. Toda la información que manejan estas herramientas se almacena en una base de datos relacional, que permite tener estos datos organizados y accesibles a los usuarios, pudiendo realizar consultas y análisis de información en cualquier momento.

Un ERP ofrece un gran número de funcionalidades por lo que este tipo de software suele dividirse en módulos que por lo general coinciden con los departamentos de una empresa, compras, facturación/contabilidad, materiales, planificación, etc. Cada fabricante de software propone su propio modelo de gestión en base a módulos estándares.

2.3.2. EL SISTEMA MES

Los sistemas MES son sistemas que unen los sistemas ERP con los sistemas de control industrial, llevando toda la gestión y planificación que se realiza en un sistema ERP al proceso productivo. Básicamente cogen los pedidos que están en el ERP, los trasforman en órdenes de fabricación, para posteriormente enviarlas al sistema de control, utilizando información online del sistema industrial para identificar en todo momento los recursos disponibles, adaptando la producción a los requerimientos de la compañía.

Un sistemas MES funciona como una extensión de un sistema ERP, pero a diferencia de este último, este interactúa con el sistema de control industrial. De este modo un ERP indica qué se va a fabricar y el sistema MES ejecuta esta fabricación, realizando algunas tareas adicionales como la conexión o desconexión de equipos, gestión de flujos de materiales, gestión de prioridades en ordenes de trabajo, o la gestión de parámetros de calidad y personal.

Muy ligado a un sistema MES podemos encontrar sistemas BATCH o sistemas de ejecución por lotes, los cuales se utilizan en procesos productivos de tipo Batch, que se caracterizan por realizar fabricaciones en grupos de unidades o lotes de material. Son estos sistemas los que controla el inicio y le fin de la producción así como todos los pasos intermedios necesarios. Estos sistemas son ejecutores de pasos o recetas, que variaran en función del tipo de producto que se desee fabricar.

La utilización de los sistemas MES ayuda a reducir tanto el tiempo de entrega como los costes de fabricación, gracias a la reducción de los tiempos de espera

entre la ejecución de órdenes de producción y los tiempos de entrega, facilitando la gestión del inventario de materias primas, así como la trazabilidad del propio proceso productivo. También reduce las transacciones en papel, ya que todo está gestionado por sistemas informáticos y toda la información sobre unidades fabricadas, parámetros de proceso, tiempos de fabricación y tolerancias se encuentra disponible en soporte electrónico.

2.3.3. NUEVOS SISTEMAS IT PARA LA INDUSTRIA 4.0

La utilización de las tecnologías de la información en los entornos industriales se ha generalizado desde el momento en que se empezaron a utilizar ordenadores como sistemas HMI y redes de comunicaciones como Ethernet, para realizar la comunicación y el intercambio de datos entre dispositivos de control industrial y ordenadores. Esto ha permitido que otros tipos de sistemas, que a priori no son propios de los entornos industriales, puedan formar parte de sistemas industriales, como por ejemplo los sistemas de gestión de bases de datos que dan soporte al almacenamiento de datos de proceso o a la gestión de maquinaria a través de interfaces web que facilitan su parametrización.

Pero la utilización de las TIC en el dominio industrial no se ha circunscrito solo a la utilización de protocolos o al desarrollo de software específico, sino que incluso paradigmas que no estaban pensados para ser aplicados en dominios industriales están empezando a tener cabida como parte de estos. En esta línea podemos destacar la utilizando de sistemas de Datawarehouse [34] o sistemas de Bussines Intelligent [35] con el fin de recoger toda la información de proceso y de diferentes fuentes de datos, sistemas de planificación, datos de proceso, sistemas de ERP o MES, y preparar informes que ayuden a analizar el rendimiento y la eficiencia del proceso productivo tanto de forma vertical como horizontal. Además con estos sistemas se pueden implementar el cálculo de indicadores como el OEE (Overall Equipment Effectiveness, en inglés), la disponibilidad de equipos o el análisis de paros de las distintas fases de un proceso industrial.

Las necesidades tanto de recopilar información de los procesos productivos, como de analizar esta información de forma conjunta, están siendo cada vez más importante, sobre todo en los nuevos paradigmas que se están desarrollando en el marco de la industria. Paradigmas como la industria 4.0 o la fábrica inteligente

pretenden, entre otros cosas, facilitar el análisis de la información, cruzando datos de mercado, datos de evoluciones de ventas de productos, datos de costes de materias primas y datos de procesos, para poder tomar decisiones sobre ciclos de fabricación en función por ejemplo de la reducción de costes o el aumento de los márgenes de beneficio lo máximo posible.

Todos estos volúmenes de datos, estructurados o no, que provienen de diferentes plataformas tecnológicas en formatos tan diversos como páginas web, bases de datos relacionales o ficheros de texto plano, pueden ser un claro escenario en el que aplicar técnicas de Big Data [36] o minería de datos [37]. Aplicando estas técnicas podemos conseguir analizar toda esta información de forma conjunta, consiguiendo uno de los desafíos que plantea la industria 4.0, consistente en planificar la producción de determinados bienes de consumo de acuerdo con la demanda del mercado, abaratando de este modo los costes de fabricación. Por ejemplo, se puede comprar materia prima en la época del año en la que los precios estén más bajos, previniendo futuras necesidades.

Ya existen trabajos [17] en los que se postula la utilización de sistemas de Big Data como parte de un sistema industrial y del potencial de su aplicación en entornos industriales. Además de los sistemas de Big Data existen otras tecnologías como la virtualización o el Cloud Computing que ya forman parte de los entornos software industrial, sobre todo los entornos virtualizados [19].

Estos entornos se están utilizando para dar soporte a los sistemas SCADA y HMI basados en software, sustituyendo Workstation físicos y servidores por máquinas virtuales, alojadas en clúster con varios hosts físicos, con las siguientes ventajas:

- **Robustez:** estos sistemas tienen una mayor tolerancia ante fallos que los sistemas tradicionales en los que una aplicación se alojaba en una máquina y, si esta caía, se caía el sistema.

- **Flexibilidad:** Es fácil aumentar la memoria, el tamaño de un disco duro o el número de procesadores en un entorno virtualizado.

- **Fácil Mantenimiento y escalabilidad:** El mantenimiento hardware y la ampliación del mismo se simplifica, ya que solo hay que preocuparse de los

hosts físicos existentes, los cuales son inferiores en número al de los sistemas tradicionales.

En este sentido podemos decir que la utilización de entornos virtuales proporciona una serie de ventajas en cualquier dominio de aplicación, y poco a poco se está extendiendo su utilización en el dominio industrial también.

En relación al Cloud Computing, aunque existen trabajos para llevar sistemas SCADA a sistemas de nube [21], quizás esta tecnología todavía esta siendo mas utilizada en el ámbito de las TIC, y sin apenas utilización como tecnología para sistemas software industriales. Pero estos sistemas ofrecen una gran ventaja, la infraestructura que los soporta no está a cargo de la empresa o fábrica que lo utiliza. Esto suele suponer por lo general una reducción tanto en coste en infraestructura, pagando tan solo por los servicios recibidos, y despreocupándose del funcionamiento de estos, así como de la evolución, actualización y el mantenimiento de los mismos. Por lo que su utilización como tecnología para dar soporte a entornos software industriales conseguiría abaratar costes, en infraestructura y coste derivados de la adquisición y mantenimiento del propio software.

Otro concepto que está empezando a asentarse en los sistemas industriales es el de IoT "Internet de las Cosas". El internet de las cosas es un concepto muy relacionado con el mundo de las TIC, en el que cada dispositivo está identificado de forma individual por una dirección URL y se puede interactuar con él utilizando Internet. En entornos industriales el concepto de IoT está derivando en IIoT (Industrial Internet of Things, en inglés). El concepto de IIoT engloba la digitalización de los sistemas industriales mediante la utilización de redes de sensores inteligentes, así como la coordinación y comunicación entre las máquinas que toman parte en los distintos procesos productivos de una fábrica.

Todas estas tecnologías aplicadas a los entornos industriales pretende conseguir una optimización de los proceso de transformación de materias primas en bienes de consumo, con el ya tradicional objetivo de reducir costes y aumentar la calidad de los productos fabricados.

Pero la aplicación de estas tecnológicas en un sistema industrial, introduce un mayor grado de complejidad, haciendo que la correcta organización e interconexión de los diferentes elementos que forman parte de estos sistemas sea crucial para el funcionamiento del mismo y en consecuencia para el rendimiento del

proceso productivo.

2.4. ARQUITECTURA DE SISTEMAS DE AUTOMATIZACIÓN INDUSTRIAL

La forma de organizar e interconectar todos los elementos que conforma un sistema industrial ha originado varios tipos de arquitecturas para entornos industriales. Estas arquitecturas han ido evolucionando con el paso del tiempo para satisfacer las cambiantes necesidades de los sistemas de control industrial, como la escalabilidad, versatilidad, flexibilidad y facilidad de diagnóstico y mantenimiento.

Los elementos de estas arquitecturas se han organizado en función del tipo de interconexionado existente ente ellos y de la solución adoptada. Además, la realización de estas interconexiones no siempre ha sido posible ya que las tecnologías de red y comunicación han ido evolucionando con el paso del tiempo tanto en aspectos software como hardware.

Dentro de estas arquitecturas uno de los aspectos más importantes que debe cubrir es el intercambio entre los dispositivos industriales y los sistemas HMI, sobre todo en sistemas basados en SCADA. Esto siempre ha planteado cierta dificultad tanto en la lectura de la información como en el envío de órdenes, especialmente cuando se integran sistemas software con dispositivos industriales de diferentes fabricantes. También hay que tener en cuenta que con la evolución de estas arquitecturas, aspectos como la seguridad y la movilidad han ido tomando cada vez más relevancia, y han influido tanto en la forma de organizar los elementos como en los sistemas utilizados para interconectarlos.

Una de las formas que se pueden utilizar para organizar estos elementos se basa en la utilización de arquitecturas jerárquica, que quizás han sido las más aceptadas y utilizadas en los sistemas industriales, existiendo trabajos que estudian la forma de crear sistemas jerárquicos flexibles [38] [39], así como diferentes tipos de propuestas y como han evolucionado estos sistemas durante los años [40].

2.4.1. MODELO ARQUITECTÓNICO DE UN SOLO NIVEL

Las arquitecturas de un solo nivel se caracterizan porque tanto los dispositivos industriales como los sistemas SCADAs (PCs) se encuentran interconectados

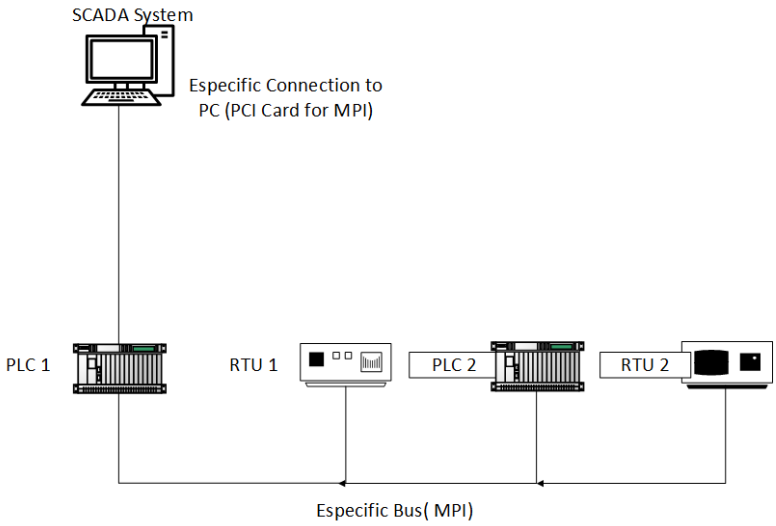


Figura 2.9: Arquitectura de un nivel.

al mismo medio, llegando incluso a compartir este con sensores y actuadores.

En estas primeras arquitecturas no se utilizaban estándares de comunicaciones ni protocolos de redes. Simplemente se utilizaban soluciones propietarias de fabricantes, las cuales podían emplear buses y protocolos de tiempo real, aprovechando los múltiples puertos hardware que ofrece un ordenador serie o paralelo, o instalando tarjetas especiales de comunicaciones que se configuran como si fuera una tarjeta más dentro del ordenador.

Esto originaba sistemas de comunicación punto a punto que por lo general empleaban cables especiales y software especial para realizar el intercambio de datos entre todos los elementos del bus. Así, podíamos encontrar protocolos como MPI o Modbus en estas arquitecturas.

Estas primeras arquitecturas presentaban varios inconvenientes como la poca escalabilidad y flexibilidad que ofrecían, ya que la introducción de un nuevo elemento requería parar todo el sistema para luego reconfigurarlo antes de volver a arrancarlo. Por tanto, las reconfiguraciones de los sistemas industriales solían ser bastante costosas y complejas. Por otra parte, este tipo de arquitectura favorece la incorporación de elementos del mismo fabricante al tener una interoperabilidad escasa.

2.4.2. MODELO ARQUITECTÓNICO DE DOS NIVELES

Las arquitecturas de un solo nivel se sustentan en la utilización de protocolos propietarios o pensados para otros ámbitos, como el de los sistemas de tiempo real (buses de campo). Esta forma de interconectar los diferentes elementos de un sistema industrial fue remplazada por sistemas de interconexión que utilizaban protocolos abiertos, como Token bus, Token Ring o Ethernet.

Esta evolución tecnológica proporciona una mayor flexibilidad y capacidad de integración entre sistemas, incluso entre elementos de distintos fabricantes, ya que el medio de interconexión no es propietario y cualquier fabricante lo puede utilizar sin ningún tipo de restricción. Todos los fabricantes disponen de tarjetas de expansión o simplemente han dotado a sus dispositivos industriales de conexiones Ethernet, por lo que basta con conectar dicho dispositivo a una red Ethernet y dotarlo de una dirección IP para poder acceder al dispositivo.

La utilización de redes Ethernet originó una nueva forma de organizar los sistemas industriales en la que los elementos de campo, actuadores y sensores, se desligan de los elementos de supervisión como los sistemas SCADA. De esta forma, los dispositivos industriales se encargan de hacer de puente entre el mundo de tiempo real y el mundo de las comunicaciones de propósito general. A pesar de utilizar protocolos y estándares abiertos para realizar el intercambio de información entre los elementos que se encuentran en la red de propósito general, son necesarios drivers o software propietario del fabricante.

Estos “drivers” de comunicación sobre TPC/IP proporcionan el acceso a la información del dispositivo industrial. De este modo los sistemas SCADA o HMI pueden comunicarse con los dispositivos industriales, accediendo así a los datos de proceso tanto para su lectura como para su escritura.

Otra forma de comunicarse con un dispositivo industrial en este escenario podría ser la creación de un programa que se comunicase con dicho dispositivo, utilizando las especificaciones del fabricante. Pero esta forma de comunicación puede suponer un esfuerzo de desarrollo bastante elevado, ya que para cada tipo de dispositivo industrial se debe de desarrollar un driver de comunicaciones específico y, en muchas ocasiones, los fabricantes no proporcionan las especificaciones para intercambiar información con sus dispositivos industriales. Otra opción consiste en utilizar los drivers propios del fabricante, pero puede que és-

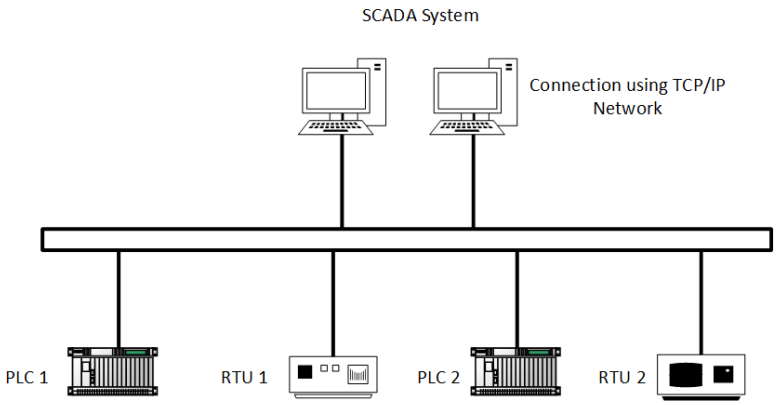


Figura 2.10: Arquitectura de dos niveles.

tos no sean capaces de satisfacer todas nuestras necesidades de comunicación como, por ejemplo, con sistemas superiores como ejecutores de recetas (BATCH Execution System) o sistemas MES (Manufacturing Execution System) o inclusos ERPs (Enterprise Resource Plannig) corporativos. De cualquier forma nos encontraremos con muchas dificultades y en la gran mayoría de los casos tendremos que realizar desarrollos específicos [41]. La comunicación entre sistemas de distintos fabricantes en la gran mayoría de los casos presenta muchos problemas [26]. Un PLC de un fabricante no se podrá comunicar con un SCADA de otro, a menos que este tenga implementado un driver para comunicarse con el PLC. Por lo que aunque estemos utilizando estándares de comunicaciones no propietarios seguimos teniendo problemas para realizar el intercambio de datos, ya que no disponemos de estándares software abiertos que nos faciliten tanto el intercambio de información como la integración de los distintos elementos que forma un sistema industrial.

2.4.3. MODELO ARQUITECTÓNICO DE TRES NIVELES

A raíz de las dificultades comentadas en la sección anterior los principales vendedores y fabricantes de sistemas HMI y software SCADA, Fisher-Rosemount, Rockwell Software, Opto 22, Intellution and Intuitive Technology, crearon un consorcio en 1995 para intentar dar solución a estos problemas de comunicación, integración y sobre todo de acceso a la información de proceso, ya que para el éxito de sus productos todos estos aspectos eran esenciales. El objetivo de esta inicia-

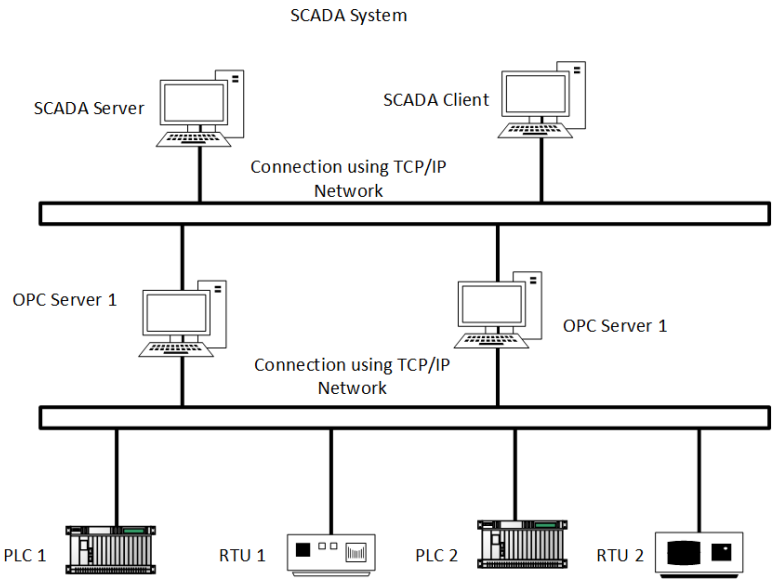


Figura 2.11: Arquitectura de tres niveles.

tiva fue definir un estándar que proporcionase un acceso uniforme a los datos de proceso, basándose en ciertas características que ofrecían los sistemas operativos Microsoft Windows.

El resultado de esta iniciativa fue la creación de las especificaciones OPC Data Access (OPC DA), las cuales vieron la luz en un periodo de tiempo relativamente corto, en Agosto de 1996. Como consecuencia se creó una fundación sin ánimo de lucro denominado OPC Foundation para mantener este estándar[8]. Esta fundación consiguió adoptar y definir su estándar más rápido que otras organizaciones, debido sobre todo a que se redujeron las principales características de indefinición con respectos a otras API. Concretamente se adoptó una arquitectura cliente-servidor utilizando la tecnología COM (Común Microsoft Object) y DCOM(Distribution Común Microsoft Object) de Windows [42] [43] que pronto fue soportado por los principales sistemas software industriales como los sistemas SCADA, HMI, los sistemas de gestión de procesos, los sistemas de control distribuido (DCS), los sistemas de control basados en PC o sof-PLCs y los sistemas (MES).

La incorporación de los servidores OPC en el entorno industrial ha favorecido la estandarización de las arquitecturas de tres niveles en las que los servidores

OPC hacen de puente entre los sistemas SCADA y los dispositivos industriales. Esto permite que los sistemas SCADA pueden acceder a la información de proceso utilizando los servidores OPC sin necesidad de utilizar drivers específicos ni desarrollos a medida para acceder a los dispositivos industriales. Al definir un único punto de acceso a la información de proceso se desacopla la administración y manejo de los dispositivos industriales de los sistemas software que gestionan los datos del proceso industrial. De esta forma, cualquier fabricante de dispositivos industriales puede crear una interfaz para comunicarse con un servidor OPC, garantizando por un lado la interoperabilidad de los servidores OPC sobre cualquier tipo de bus de campo y proporcionando transparencia con respecto a los sistemas que necesitan acceder a la información de proceso. En definitiva, esto ofrece sistemas más flexibles, interoperables y escalables, tanto desde el punto de vista de la forma de realizar las interconexiones entre los distintos elementos (son sistemas abiertos), como del software utilizado para realizar el intercambio de datos (basados en un estándar abierto).

2.4.4. MODELO ARQUITECTÓNICO DE CUATRO NIVELES

El número de elementos o sistemas que despeñan alguna tarea dentro de un sistema industrial puede ser elevado, sobre todo desde la introducción de sistemas TIC, por lo que la organización de estos elementos para conseguir sistemas seguros con un buen rendimiento es crucial.

En la figura 2.12, podemos ver una propuesta de arquitectura para organizar los distintos elementos que pueden llegar a formar un sistema industrial. Estos se organizan conceptualmente mediante la utilización de distintas subredes, las cuales están interconectadas mediante cortafuegos. Esta forma de organizar los elementos de un sistema industrial nos proporciona un sistema parecido al de las capas de una cebolla, las cuales tendremos que ir quitando hasta llegar al centro de la misma, donde está la información y los sistemas más sensibles y que interactúan con el proceso. Este enfoque es el que ofrece más garantías desde un punto de vista de seguridad, ya que protegemos los sistemas de control y monitorización en la parte central de la cebolla.

Con este enfoque los sistemas que realizan las mismas funciones se encuentran en la misma subred. Así tenemos en primer lugar los buses de campo conec-

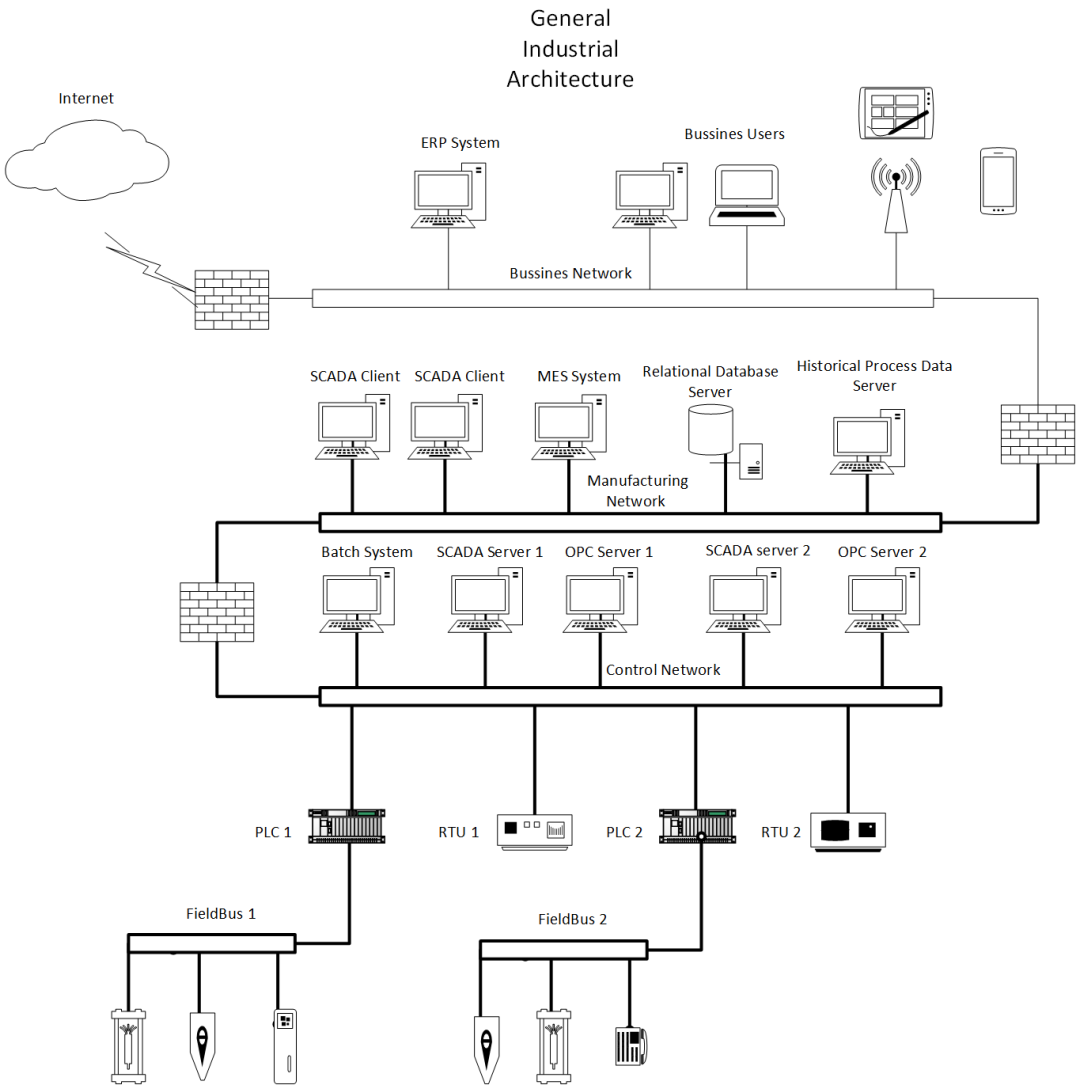


Figura 2.12: Arquitectura de cuatro niveles.

tados a los distintos dispositivos industriales. Este primer nivel es el único donde tenemos sistemas de tiempo real estricto y, es el que interactúa directamente con la maquinaria, sistemas neumáticos, hidráulicos y eléctricos del proceso industrial.

En el siguiente nivel, que denominamos red de control, nos encontramos con los dispositivos industriales y los servidores industriales. Toda la información de proceso se accede a través de los servidores OPC, los cuales brindan esta información a los niveles superiores, mediante la utilización de clientes OPC. El siguiente nivel, que recibe el nombre de red de Manufacturing, incluye los sistemas SCADA, los sistemas MES, los históricos de datos de planta y los sistemas BATCH. Estos sistemas utilizan los servidores OPC para enviar y recibir información de los dispositivos industriales.

Por último, nos encontramos con la red de Negocio, donde tenemos alojados sistemas como ERPs, sistemas de BI, servidores Web, dispositivos móviles y demás sistemas corporativos. En esta red se incluyen todos los sistemas TIC que la compañía utiliza y que sustentan todos los aspectos de negocio. Estos sistemas suelen interactuar con los sistemas de la red de Manufacturing, para desempeñar tareas como la planificación de órdenes de producción y recogida de información de proceso por ejemplo.

2.4.5. MODELO ARQUITECTÓNICO DE N NIVELES

La penetración de tecnologías como el Big Data, Datawarehouse e IoT en los entornos industriales, así como la aparición de paradigmas como industria 4.0, están cambiando la forma de entender el proceso productivo y planteando una nueva forma de organizar los elementos que componen un sistema industrial, por lo que es necesario una nueva arquitectura que de soporte a todos estos nuevos requerimientos:

- Acceso transparente y universal a toda la información de proceso para poder planificar las distintas producciones.
- La información de proceso debe poder ser accesible por cualquier operador o técnico en cualquier sitio y en cualquier momento.
- La información debe de poder analizarse fácilmente, para optimizar el pro-

ceso productivo y poder adaptarlo a las necesidades de los clientes finales.

- La seguridad debe poder garantizarse a dos niveles: intercambio seguro de datos y acceso seguro a los mismos.
- Soporte para la utilización de infraestructuras de nube.
- Los diferentes sistemas industriales, MES, BATCH y SCADAS, deben compartir información para coordinar sus acciones. Disponer de un conjunto de servicios "industriales" nos ayudaría a tener un medio común para interactuar con cualquier tipo de software. Esto facilita el acceso a los procesos productivos por parte de los sistemas de negocio.
- Los nuevos sistemas HMI deben de ayudar a los operarios a supervisar los procesos industriales. Por esta razón estos sistemas HMI debe de ser ubicuos, móviles y deben de tener un interfaz gráfico amigable para mostrar información contextual.
- Soporte para nuevos tipos de aplicaciones corporativas que ayudaran a incrementar la productividad uniendo los sistemas industriales y los sistemas TIC.
- Soporte para sistemas predictivos que analizaran la información utilizando técnicas de Biga Data
- Soporte para la gestión de sistemas descentralizados y virtualizados.

En la figura 2.13 podemos ver una posible arquitectura con la que se puede dar soporte a todas estas nuevas características. Esta arquitectura la denominamos de N niveles porque, ya que a diferencia de las anteriores, no está organizada en un número finito de niveles que separen conceptualmente los sistemas. Además, en este caso no todos los niveles tienen por qué estar en la propia infraestructura de la fábrica, sino que pueden encontrarse en ubicaciones geográficas distintas. La inclusión de Internet como parte de un sistema industrial facilita el acceso a plataformas de Cloud Computing, Big Data e IoT como parte de los sistemas industriales [44].

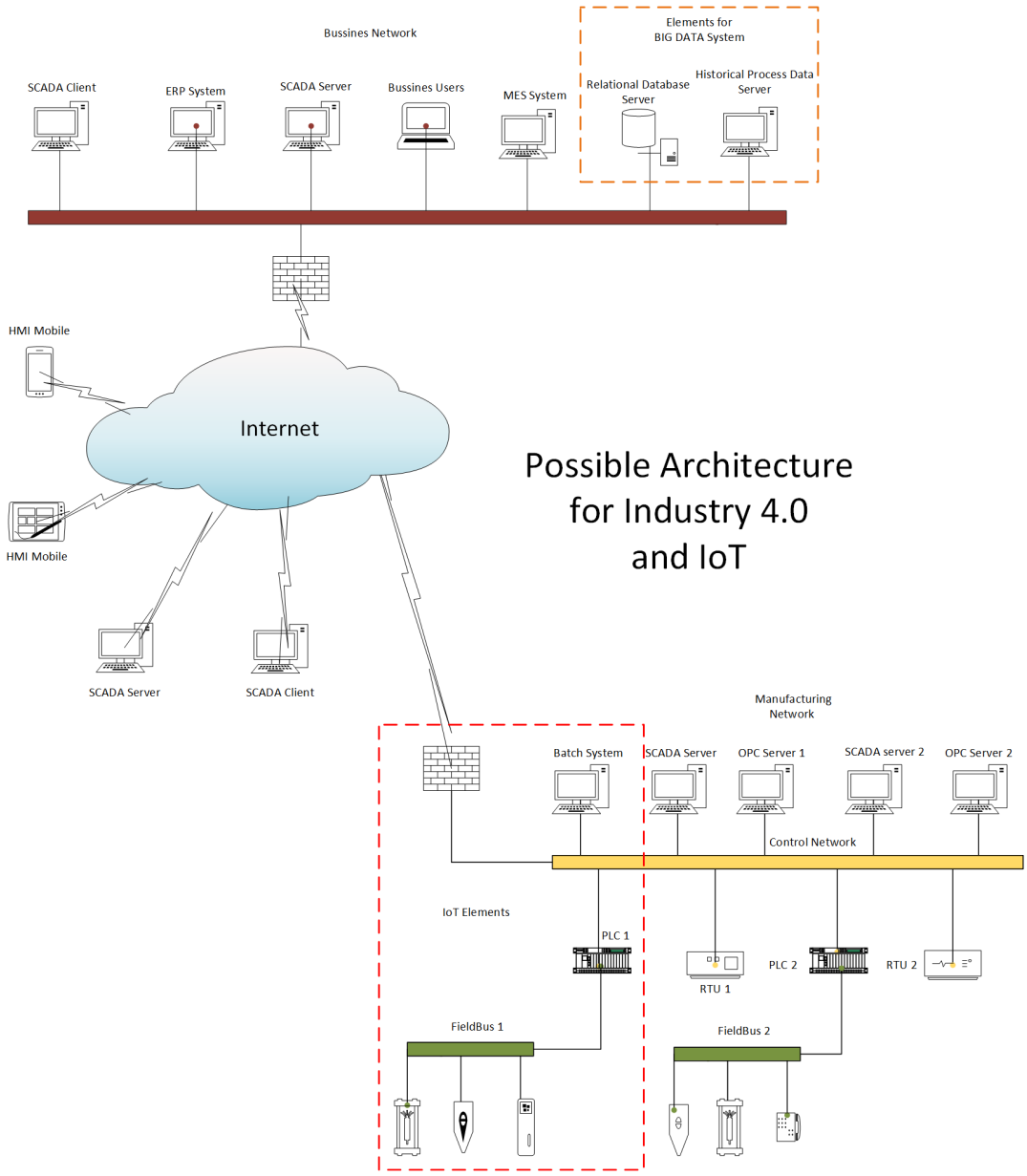


Figura 2.13: Arquitectura de n niveles.

La utilización de Internet en los sistemas industriales también presenta ciertas cuestiones que se deben resolverse. Al basarse las comunicaciones por Internet en TCP/IP se simplifica el diseño del sistema industrial si Ethernet se aplica en los niveles de red del sistema industrial; en caso contrario, habrá que agregar routes o nodos adicionales que se encarguen de hacer de puente entre las distintas subredes. El acceso a Internet de los distintos elementos de esta arquitectura, también supone considerar la seguridad como un aspecto fundamental, especialmente si utilizamos medios inalámbricos para el intercambio de información. Debemos garantizar la seguridad de nuestros sistemas con mecanismos que encripten los intercambios de datos, no permitiendo el acceso no deseado a los diferentes sistemas software de nuestro entorno industrial. Incluso puede ser necesario definir redes virtuales privadas seguras para la interconexión de los elementos de sistema industrial que tengan que utilizar Internet cuando se encuentran en ubicaciones geográficas diferentes. Otros aspectos como la interoperabilidad, la disponibilidad y reusabilidad también deben de ser tenidos en cuenta, consiguiendo así sistemas industriales universales, con un alto grado de robustez, fáciles de mantener y escalar.

2.5. CONCLUSIONES

Los procesos industriales han experimentado una evolución a largo del tiempo, buscando cubrir principalmente tres objetivos aumentar la productividad, aumentar la calidad y reducir costes. Esta evolución se ha visto reflejada en todos los aspectos que forman parte de un sistema industrial, introduciendo nuevos sistemas y mejoras en cada uno de los elementos que forman parte de este tipo de sistemas. Los sistemas software en el contexto de los sistemas industriales han ido adquiriendo un papel cada vez más relevante a medida que los sistemas industriales han tenido que ir haciendo frente a procesos productivos más automatizados y especializados. Estos nuevos proceso productivos necesitan sistemas software más flexibles, capaces de adaptarse a la heterogeneidad de una fabrica, facilitando la gestión y la integración de todos los elementos del sistema industrial.

Todos estos elementos deben estar organizados de alguna forma mediante la definición de algún tipo de arquitectura. Las arquitecturas jerárquicas son la for-

ma más aceptada a la hora de organizar los sistemas software industriales[45], y en especial para cubrir las necesidades que la Industria 4.0 va a ir demandando. Una buena arquitectura jerarquizada puede ayudarnos a cubrir aspectos como la seguridad, la interoperabilidad y la escalabilidad de un sistema software industrial.

3

EL ESTÁNDAR OPC

"With the right infrastructure, data can be converted into knowledge, often in surprising ways."
Michael Nielsen 1974

3.1. INTRODUCCIÓN

La introducción de las TIC y utilización de ordenadores PCs en entornos industriales ha facilitado la utilización de aplicaciones software en el proceso de monitorización y control de los procesos productivos. Estos nuevos sistemas de control software empezaron a utilizarse a principios de los 90[2]. En estos sistemas los ordenadores PCs empezaron a ser utilizados como sistemas HMI flexibles con la capacidad monitorizar y supervisar el proceso industrial mediante la utilización de software desarrollado para este propósito sobre plataformas Windows.

Para poder realizar las tareas de supervisión y monitorización estos sistemas software debían poder comunicarse con los dispositivos industriales, tanto para acceder a la información que estos manejan (lectura) como para enviarles órdenes y parámetros (escritura). Por este motivo, la capacidad de intercambio de información entre los sistemas software y dispositivos industriales fue y sigue siendo uno de los aspectos más significativos en un sistema industrial, y supuso un verdadero desafío en los primeros diseños de sistemas industriales.

Aunque inicialmente el propio fabricante se encargaba de desarrollar todo el software específico necesario, desde los sistemas SCADAs hasta los drivers para comunicarse con los dispositivos industriales, su aplicación real en la Industria presentaba serias dificultades al ser incapaces de atender la heterogeneidad de los distintos fabricantes de dispositivos industriales, así como no tener la flexi-

bilidad suficiente para satisfacer todas las necesidades que se iban demandando en el nivel de supervisión y control[26]. Toda esta problemática ha hecho que los desarrolladores software de estos sistemas inviertan mucho tiempo y esfuerzo en estandarizar el acceso a la información del proceso, ya que por lo general esta información se encuentra disponible en arquitecturas heterogéneas que engloban desde distintos buses de campo, distintos protocolos de y distintos tipos de interfaces de comunicación. Los vendedores de sistemas HMI y SCADA siempre han tenido que enfrentarse a este problema de acceso a la información de proceso. Por este motivo los principales fabricantes de estos sistemas software crearon una iniciativa en 1995 para intentar dar solución a esta problemática. La meta de esta iniciativa fue definir un estándar abierto que proporcionase un acceso común y normalizado a los datos de proceso basado en sistemas Windows.

3.2. LA FUNDACIÓN OPC

Las tareas principales de esta fundación que nació en 1996 [8] se centran en desarrollar el estándar OPC, creando nuevas especificaciones que se adaptan a las necesidades de los sistemas software industriales y, por otra parte, velar por que los productos que se desarrollan sigan las especificaciones del estándar.

Hoy en día la fundación OPC tiene unos 450 miembros, alrededor de unos 2.500 productos registrados y unos 1.500 productos de distintos fabricantes habilitados. Para el registro y aprobación de estos productos la fundación OPC dispone de un mecanismo de verificación para garantizar aquellos que adoptan su estándar tienen cierta calidad. Para este propósito se creó el programa de cumplimiento de OPC.

El programa de cumplimiento de la fundación OPC, define dos niveles de certificación. El primer nivel combina la propia certificación y la interoperabilidad de los productos en diferentes workshops. La fundación OPC ofrece una serie de herramientas de test para el cumplimento de sus estándares más relevantes. Estas herramientas se pueden usar para testear y enviar los resultados, encriptados, a la fundación OPC. Los talleres de interoperabilidad se celebran anualmente en Europa, Norte América y Japón y en estos talleres los diferentes vendedores pueden testear la interoperabilidad de sus productos OPC. Los productos que pasen el auto-test de certificación puede usar el logotipo *Self-Test* de la fundación indi-

cando que han pasado el nivel básico de cumplimiento de OPC.

El segundo nivel para la certificación de un producto se realiza mediante un test de Certificación independiente, lo que se denomina *Certification Test Labs*. Esta tercera parte de la acreditación se realiza con un test de laboratorio más amplio, en el que se realizan pruebas de estrés comportamiento, carga, interoperabilidad (comportamiento en varios entornos) y usabilidad. Los productos que pasen esta tercera parte de la certificación pueden utilizar el logo de OPC, indicando el alto nivel de cumplimiento de los estándares OPC.

La OPC Foundation ha desarrollado varias especificaciones desde el año 1996, siendo la última, la OPC Unified Architecture, la que intenta dar respuesta a las nuevas necesidades de los sistemas software industriales.

3.3. OPC CLÁSICO

De acuerdo con los diferentes requerimientos en el marco de las aplicaciones industriales de finales de los años 90, la fundación OPC desarrolló tres especificaciones: Data Access (DA), Alarm & Events (A&E) y Historical Data Access (HDA)[46]. El acceso a los datos de proceso está descrito en la especificación DA, la especificación A&E describe las interfaces para la información basada en eventos y el reconocimiento de alarmas y, por último, la especificación HDA describe las funciones necesarias para almacenar datos. Todas estas especificaciones ofrecen interfaz para interactuar con los dispositivos industriales a través del conjunto de señales de un sistema industrial.

OPC utiliza una arquitectura cliente-servidor para el intercambio de información. El servidor OPC se encarga de encapsular las diferentes fuentes de datos de los distintos dispositivos industriales, haciendo esta información accesible mediante su propio interface. Por otra parte, los clientes se conectan al servidor OPC para poder acceder a los datos y consumirlos. En la figura 3.1 se puede ver la arquitectura cliente y servidor típica de OPC.

Las interfaces clásicas de OPC están basadas en la tecnología COM y DCOM de Microsoft[47]. COM y DCOM proporcionan un mecanismo transparente para el cliente a la hora de invocar a objetos-COM, que se están ejecutando en el servidor en el contexto del mismo proceso o en otro proceso o nodo de la red. Utilizando esta tecnología, podemos acceder a los procesos y servicios de todos los

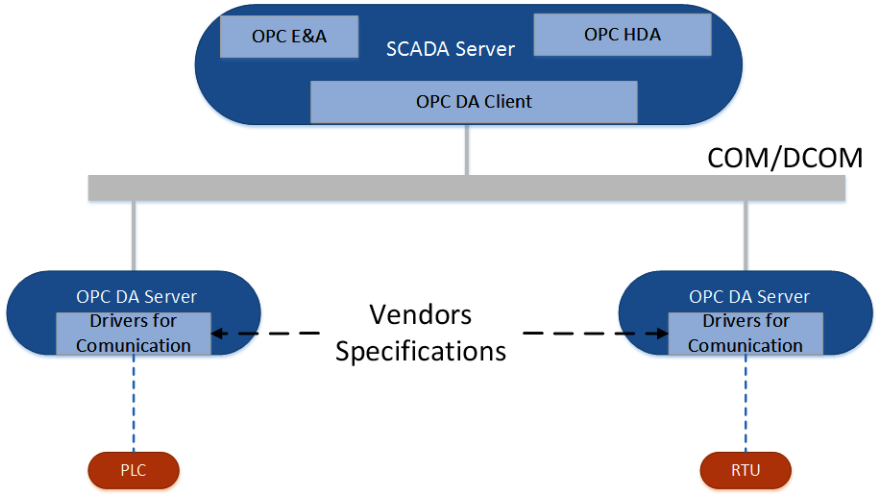


Figura 3.1: Arquitectura Cliente Servidor de un sistema OPC [3].

ordenadores PCs, que tengan sistema operativo Windows, reduciendo así el tiempo de desarrollo de las especificaciones para productos comerciales que sigan el estándar OPC. Esta ventaja fue importante para el desarrollo de los estándares de la OPC Foundation.

Las dos principales desventajas que presenta este enfoque son la dependencia de las plataformas Windows y los problemas DCOM, ya que DCOM es difícil de configurar, cuando se utilizan comunicaciones remotas con OPC. Además, no se pueden configurar *timeouts* de reenvió o detección de errores, ni realizar comunicaciones entre elementos que se encuentran en diferentes subredes, lo que hace que no se pueda utilizar en entornos en los que Internet sea parte de ellos.

3.3.1. OPC PARA EL ACCESO DIRECTO A LOS DATOS (DA)

El interfaz OPC Data Access permite tanto la lectura como escritura de variables que contiene los datos actuales de proceso. El principal uso de esta especificación consiste en mover los datos en tiempo *"real"* desde los dispositivos industriales como PLCs, a los sistemas HMI, SCADA u otros sistemas de visualización clientes. OPC DA es quizás el interfaz más importante de OPC, ya que es el más extendido y soportado [48] [42] [41].

En esta especificación, los clientes explícitos OPC DA seleccionan las variables (OPC items) que quieren leer, escribir o monitorizar del servidor. Los clientes

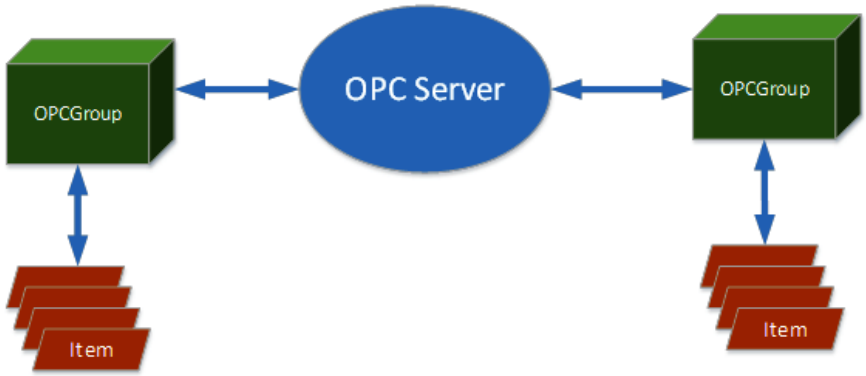


Figura 3.2: Estructura de Objetos de un servidor para la especificación OPC DA [3].

OPC establecen una conexión con el servidor creando un objeto OPCServer. Los objetos servidor ofrecen métodos para navegar a través de la jerarquía de datos, creadas en el espacio de direcciones del servidor, para encontrar items o propiedades de estos como, por ejemplo, el tipo de dato.

Para acceder a los datos, los clientes cuentan con la posibilidad de crear grupos de items OPC con una configuración común. De este modo se actualizará todo el grupo de items mediante el objeto *OPCGroup*. En la siguiente figura 3.2 se muestra un ejemplo de dos grupos con un conjunto de items que un cliente puede crear en el servidor.

Los grupos de items de un servidor OPC pueden ser leídos o modificados por un cliente según las necesidades del cliente. Pero la manera estándar para realizar una lectura de un dato desde un cliente consiste en monitorizar los cambios de valor de un ítem o un grupo de items en el servidor. Es el cliente el que define una velocidad de actualización, o tiempo de escaneo del grupo de items La velocidad de actualización es utilizada en el servidor como ciclo para comprobar los valores que han cambiado. Después de cada ciclo el servidor envía solo los datos que han cambiado al cliente.

OPC proporciona datos de forma instantánea que no pueden ser accesibles de forma permanente, por ejemplo, cuando la comunicación con los dispositivos se interrumpe temporalmente. Para detectar estos problemas en el intercambio de información, la especificación OPC proporciona un timestamp y un valor de calidad de los datos que se están manejado en cada instante de tiempo. Este valor de calidad especifica si el dato es exacto (good), no disponible (bad) o desconoci-

do (uncertain).

3.3.2. OPC PARA EL ACCESO A DATOS HISTÓRICOS (HDA)

OPC Historical Access proporciona acceso a los datos almacenados, desde un simple sistema de data logging hasta un complejo sistema SCADA. Esta especificación permite acceder a archivos históricos y recuperarlos de una manera uniforme.

Los clientes OPC pueden conectarse al servidor HDA, mediante la creación de un objeto OPCHDAServer. Este objeto ofrece todos los interfaces y métodos para leer o actualizar el historial de datos. Si fuese necesario además se podría crear un segundo OPCHDABrowser para navegar por el espacio de direcciones del servidor HDA.

La principal funcionalidad que ofrece esta especificación es la de acceder a los datos historizados que se han recogido de los dispositivos industriales. Para ello contempla tres mecanismos diferentes. El primer mecanismo consiste en leer los datos en bruto del sistema de almacenamiento, siendo el cliente el que define una o más variables y el dominio de tiempo que quiere leer. El servidor devuelve todos los datos archivados para ese rango de tiempo específico para un máximo de valores definidos por el cliente. El segundo mecanismo consiste en leer valores de una o más variables con marcas de tiempo específicas. La tercera forma de acceso a los datos consiste en la utilización de cálculos agregados para uno o más datos historizados, dentro de un domino de tiempo específico de una o más variables. Los valores siempre tienen asociado la calidad y la marca de tiempo. Adicionalmente a los métodos de lectura, OPC HDA también define métodos de inserción, actualización y borrado de datos históricos en la base de datos.

3.3.3. OPC PARA EVENTOS Y ALARMAS (E&A)

Esta especificación permite la recepción de eventos o notificaciones y alarmas desde el servidor al cliente. Para esto el cliente se puede suscribir a los eventos o las alarmas configuradas en el servidor mediante la creación de objetos *OPCEventServer* y después generando objetos *OPCEventSubscription* (Como se puede observar en la figura 3.3).

Figura 3.3: Objetos de un servidor para la especificación E&A [3].

3.3.4. OTROS INTERFACES DE OPC

La fundación OPC define otras especificaciones, aparte de las vistos, con las que pretende cubrir ciertas necesidades más específicas. Las básicas, como *OPC Overview* y *OPC Common* define interfaces y comportamientos que son comunes a todas las especificaciones OPC basadas en COM y DCOM. En la figura 3.4 se pueden ver todas estas especificaciones clásicas de OPC.

OPC Security especifica cómo debe de controlarse el acceso de los clientes a los servidores para proteger la información sensible y no permitir accesos no deseados a los datos y parámetros de proceso. *OPC Complex Data*, *OPC Batch* y *OPC Data eXchange* (DX) son extensiones de OPC DA. *OPC Complex Data* define como deben ser descrito y enviados los valores de datos con estructuras complejas. *OPC DX* especifica el intercambio de datos entre servidores DA mediante la definición del comportamiento de la parte cliente y de la configuración de los interfaces cliente dentro del propio servidor. *OPC Batch* es una extensión de DA para las necesidades especiales de procesos de producción por lotes. *OPC Command* define un mecanismo para ejecutar programas vía OPC. Esta especificación nunca ha sido liberada ya que antes de ser terminada se liberó el estándar OPC UA, el cual recoge completamente dicha especificación.

3.3.5. OPC XML-DA

OPC XML-DA fue la primera especificación de la fundación OPC que pretendía ser multiplaforma al no estar basada en COM/DCOM. Esta especificación se basa en tecnología HTTP/SOAP y Servicios Web [49]. La idea detrás de esta especificación fue la de reducir al mínimo el conjunto de métodos para el intercambio de información de forma similar al estándar OPC DA.

Así, solo 8 métodos hicieron falta para cubrir las principales características de la especificación OPC DA:

- *GetStatus*, para verificar el estado del servidor.

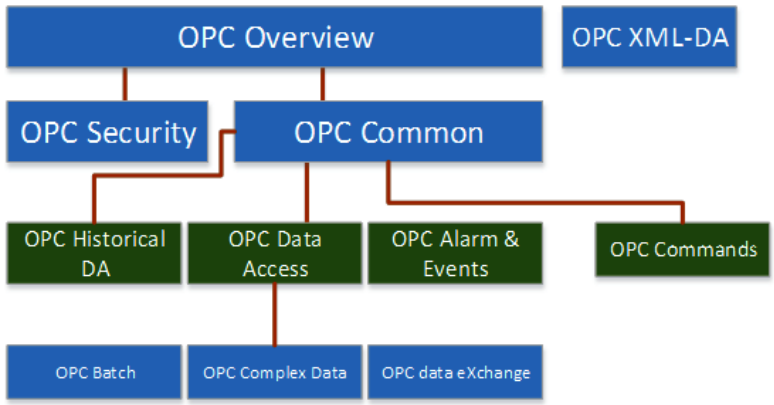


Figura 3.4: Organización de las especificaciones del OPC clásico [3].

- *Read*, para leer uno a mas valores de items
- *Write*, para escribir uno o más valores de items
- *Browse y GetPropieties*, para obtener información sobre los items disponibles.
- *Subscribe*, para crear un suscripción a una lista de items
- *SubscriptionPolledRefresh* para intercambiar los valores que han cambiado en una subscripción.
- *SubcriptionCancel* para cancelar una subscripción.

OPC XML-DA fue diseñado para proporcionar el acceso a los datos de proceso utilizando Internet y facilitando así la integración con los sistemas de negocio y el mundo empresarial. Además presenta una característica bastante interesante desde el punto de vista de la interoperabilidad, la posibilidad de implementarse en plataformas embebidas independientes sin necesidad de utilizar tecnología Microsoft. Sin embargo, su implantación no tuvo gran éxito por su alto consumo de recursos y limitado rendimiento, ya que el intercambio de información entre cliente y servidor se realiza mediante mensajes XML, que requieren mayor procesamiento que los mensajes o paquetes binarios.

3.3.6. HASTA DONDE LLEGA OPC CLÁSICO

El conjunto de estándares que forman el OPC clásico fueron diseñados y pensados para facilitar un único punto de acceso a los datos de proceso con un conjunto de características y funcionalidades bien definidas, las cuales en sus inicios solucionó de forma exitosa la problemática que presentaban el intercambio de información entre los sistemas de supervisión y monitorización y los dispositivos industriales.

Como se ha comentado el estándar OPC clásico fue desarrollad sobre las tecnologías COM y DCOM. Inicialmente esto facilitó su adopción y redujo la complejidad a la hora de desarrollar productos software, pero con el paso del tiempo la utilización de la tecnología COM y DCOM supuso más un inconveniente que una ventaja. En primer lugar porque la utilización de esta tecnología obliga a utilizar plataformas basadas en sistemas operativos de Microsoft, por lo que el grado de interoperabilidad se reduce al no poder utilizar otras plataformas basadas por ejemplo en Linux, Android o MAC Os X. Esto además se acentuó con la aparición de dispositivos como Tablets o SmartPhones, en los que predominan sistemas operativos como Android o IOS y en los que se hace una tarea imposible desplegar sistemas HMI que utilicen OPC DA [50]. Por otra parte, la utilización de DCOM y COM nos obliga a trabajar con sistemas que se encuentren siempre en la misma subred, ya que este no es capaz de interconectar sistemas que se encuentran en diferentes subredes. Suele ser complejo de configurar y en cada intercambio de datos suele cambiar de puerto de comunicaciones, lo que acentúa aún más su complejidad a la hora de configurar firewall o routers. Por lo tanto, su utilización en sistemas que se apoyen en Internet es casi inviable [51].

La seguridad es otro aspecto que no se gestiona bien en COM y DCOM al ser una tecnología bastante antigua y por existir un gran número de malware que utiliza COM/DCOM [52] para infectar o atacar un ordenador. Ninguno de los estándares del OPC clásico ofrecen mecanismos de seguridad adicionales más allá de la autentificación en la parte servidor mediante usuario y contraseña, no ofrece encriptación de datos, ni ningún otro mecanismo basado en la utilización de certificados digitales, por ejemplo.

Sin embargo, el estándar OPC ha sido ampliamente aceptado, haciendo que hoy por hoy OPC se utilice como el interfaz estándar para el intercambio de infor-

mación entre los diferentes niveles de la pirámide jerárquica de automatización. Incluso se utiliza en muchas áreas para las que no ha sido diseñado, como la conexión con ERP o entornos científicos como MATLAB.

Por otro lado las especificaciones de OPC clásicas solo contempla el manejo de datos o información simple, es decir, solo son capaces de gestionar los tipos de datos tradicionales, enteros, flotantes o cadenas de caracteres. Pero estos tipos de datos no son suficientes para cubrir las necesidades que están apareciendo de la mano de los nuevos paradigmas como la industria 4.0 [17] e IIoT [53] en los que se necesitan estructurar y organizar la información de los diferentes procesos industriales, con el objetivo de crear plataformas o sistemas con una mayor nivel de abstracción. Estos sistemas nos aíslan de los detalles del protocolo de comunicaciones del dispositivo industrial, y además nos proporcionan una forma de organizar y modelar los propios sistemas industriales [54].

3.4. OPC UA

3.4.1. MOTIVACIÓN DE OPC UA

La nueva especificación de la fundación OPC nació con el objetivo de reemplazar todas las especificaciones existentes basadas en COM sin sacrificar ninguna de las características ni rendimiento que poseían los sistemas basados en OPC clásico. Esta especificación recibió el nombre de arquitectura unifica, más comúnmente conocida como OPC UA [3]. Este nuevo estándar pretende ofrecer un sistema/plataforma independiente del sistema operativo Windows (tal y como ocurría en versiones precedentes) y con un metamodelo rico y extensible capaz de describir y modelar sistemas con cualquier grado de complejidad.

El amplio rango de aplicaciones donde se utiliza OPC [55] requiere que este sea fácilmente escalable para abordar sistemas embebidos, SCADA y DCS, e incluso sistemas MES y ERP [56]. Los requerimientos más importantes de OPC UA se pueden ver en la tabla 3.1.

Estos requerimientos pueden ser agrupados, como se ve en la tabla 3.1, en dos grandes grupos: requerimientos de intercomunicación y requerimientos para el modelado de datos.

El estándar OPC está siendo utilizado como un sistema software clave en el intercambio de datos entre las diferentes partes software de un sistema indus-

Comunicación entre sistemas distribuidos	Modelo de datos
Confiabilidad por: Robustez y tolerancia a fallos Redundancia	Modelo común para todos los datos OPC.
Plataforma independiente	Orientado a Objetos
Escalabilidad	Tipo de sistema extensible
Alto Rendimiento	Meta información
Internet y Cortafuegos	Datos y Métodos Complejos
Seguridad y Control de acceso	Escalabilidad dese modelos simples a modelos complejos
Interoperabilidad	Basado en modelo abstracto Basado en otros estándares y modelos.

Tabla 3.1: Principales características y funcionalidades de OPC UA [3].

trial [57]. Por lo tanto, la fiabilidad de las comunicaciones entre estos sistemas es esencial. Gracias a la evolución de las redes de comunicaciones, los esfuerzos para conseguir sistemas de intercomunicaciones fiables se han centrado en aspectos como la robustez y la tolerancia a fallos, mediante la redundancia y la alta disponibilidad de todos los sistemas software que forman parte en el intercambio de información.

Otro aspecto que se debe de tener en cuenta es el alto rendimiento en entornos de intranet o redes locales, en cuanto al intercambio de información se refiere. Además también se debe de considerar la comunicación en Internet a través de cortafuegos, la cual necesita de requerimientos adicionales de seguridad y control de acceso [58].

Por otro lado, las especificaciones del OPC clásicas presentan una capacidad

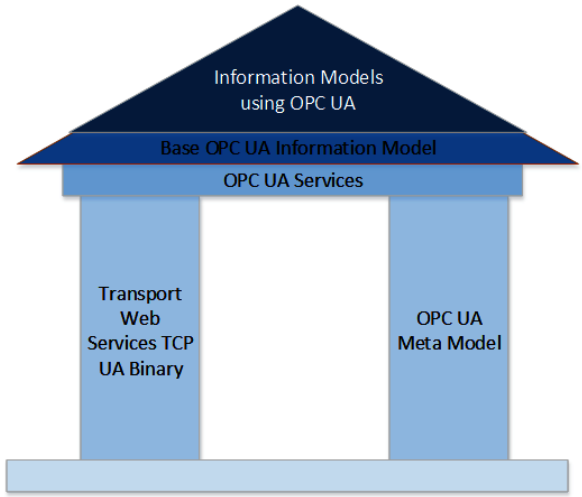


Figura 3.5: Organización en capas de OPC UA [3].

de modelado muy limitada orientada a una estructuración simple de datos. Se hace, por tanto, necesario mejorar esta capacidad, proporcionando un modelo común orientado a objetos para todos los datos que puede manejar un servidor OPC [59]. Este modelo debe incluir un sistema extensible que permita describir también estructuras de datos más complejas. Por ese motivo, la capacidad de crear modelos con cierto grado de abstracción en OPC UA supone un salto importante respecto a las especificaciones del OPC clásico. No sólo proporciona capacidad de modelado sino también da soporte a modelos simples con conceptos simples. Por esta razón se requiere tener un modelo base simple y abstracto, pero extensible para poder escalar este modelo simple a un modelo complejo.

En un principio se plantearon unos 40 requerimientos y casos de uso para la nueva generación del estándar OPC. Estos requerimientos no fueron solo creados por miembros de la fundación OPC, sino que otras organizaciones como IEC [60] e ISA (Instrument, System and Automation)[61], interesadas en la utilización de OPC como mecanismo de transporte e intercambio de datos se involucraron desde el principio en el proceso de diseño. En este grupo la fundación OPC definió “COMO” describir y transportar los datos y las organizaciones colaboradores “QUE” datos querían describir y transportar dependiendo de su modelo de información.

Otro objetivo importante, que se tuvo en cuenta durante el diseño de esta es-

pecificación fue, la de permitir una fácil migración desde los sistemas de OPC clásico hacia OPC UA para proteger las inversiones realizados con anterioridad[3].

Para alcanzar los objetivos definidos, el estándar OPC UA está organizado en diferentes capas, como se puede ver en la figura 3.5, siendo los pilares principales de OPC UA el mecanismos de transporte de datos y el modelo de información.

La capa de trasporte define diferentes mecanismos de comunicación optimizados para diferentes casos de uso. La primera versión de OPC UA fue definida como una optimización binaria del protocolo TCP, con el objetivo de conseguir un sistema de comunicaciones de alto rendimiento en redes locales o LAN. Además también se contemplaron estándares de Internet como servicios Web, XML y HTTP para las comunicaciones entre distintas subredes y firewall. El modelo abstracto de comunicación no depende de ningún protocolo específico, permitiendo de esta manera añadir nuevos protocolos en el futuro.

El meta-modelo de información define las reglas y las bases sobre los que construir los modelos que se alojaran y podrán ser explorados dentro de un servidor OPC UA. OPC UA define también los puntos de entrada en el espacio de direcciones y los tipos básicos para construir una jerarquía de elementos tipo. Estos conceptos y elementos básicos pueden ser extendidos, creando modelos conceptuales más abstractos. Adicionalmente, se definen algunas mejoras conceptuales, como la descripción de máquinas de estados utilizando diferentes modelos de información.

Los servicios UA pueden verse como interfaces entre distintos servidores, los cuales proporcionan un punto de acceso para la gestión y visualización de los modelos alojados en dichos servidores. Estas interfaces permiten a los clientes poder acceder al modelo de información almacenado en el servidor OPC UA. Estos servicios no son más que una serie de funcionalidades que los servidores ofrecen a sus clientes y se definen de una manera abstracta, utilizando el modelo de transporte para intercambiar datos entre los clientes y el servidor.

OPC UA cubre con su conjunto de especificaciones todas las características del OPC clásico que se mencionaron en la sección anterior. En el nuevo estándar, el modelo de información no se limita a una extensión de datos analógicos y digitales como ocurre en OPC DA, sino que ofrece un modelo de información mucho más completo. De hecho otras organizaciones pueden construir su mo-

delo sobre el modelo de información de OPC UA, exponiendo su especificación de información vía OPC UA. Ejemplos de estos modelos sobre OPC UA son Field Device Integration (FDI) [62], Electronic Device Description Language (EDDL) [63] y Field Device Tool (FDT) [64].

3.4.2. LAS ESPECIFICACIONES DE OPC UA

Las especificaciones de OPC UA están divididas en diferentes bloques siguiendo las recomendaciones de la norma IEC. Es por ello que el estándar OPC UA también es conocido como la norma IEC 62541. La figura 3.6 muestra todas las especificaciones del estándar UA, organizadas en grupos según su importancia y tipología. Las dos primeras especificaciones [65] [66] no se pueden considerar estándares, ya que la primera de ella *"Concepts"* dan una visión general sobre OPC UA y la segunda describe los requerimientos y el modelo de seguridad para OPC UA.

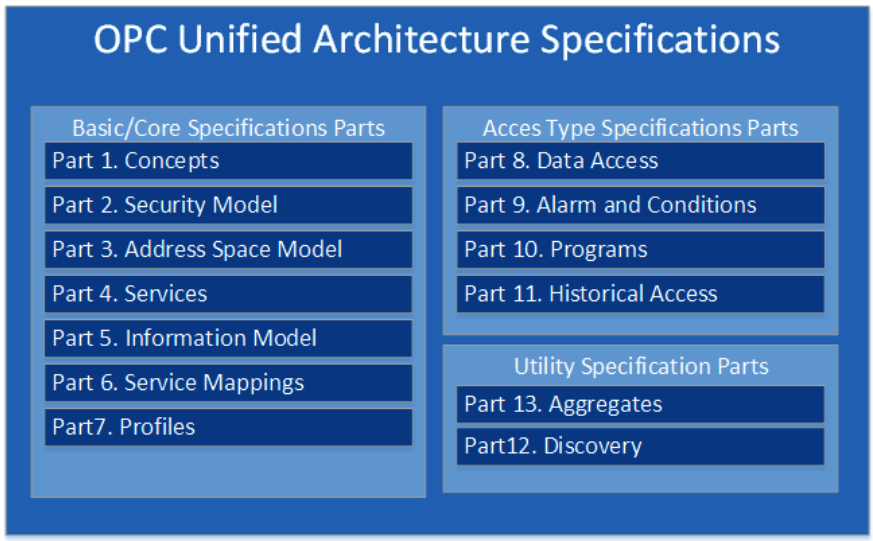


Figura 3.6: Grupos de las diferentes especificaciones de OPC UA [3].

Las especificaciones más importantes del estándar son la 3 [67] y 4 [68], ya que son los documentos clave para el diseño y desarrollo de aplicaciones acordes al estándar UA. El modelo del espacio de direcciones, parte 3, especifica cómo construir los elementos que se instanciarán y serán expuestos en el espacio de direcciones de un servidor OPC UA. Además también especifica cómo construir

nuevos tipos de objetos que podrán ser utilizados dentro del modelo de información de OPC UA, con los que construir modelos acordes a estos nuevos tipos de objetos.

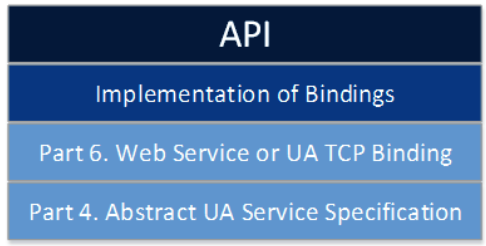


Figura 3.7: APIs de comunicación que pueden utilizar los diferentes servicios UA [3].

La abstracción de los servicios definida en la parte 4, especifica cómo se realiza la interacción entre las aplicaciones UA cliente y servidor. Los clientes utilizan servicios para encontrar la información proporcionada por el servidor. Estos servicios son abstracciones desde el punto de vista de la información a intercambiar entre las aplicaciones UA, pero no especifican como representar la conexión y tampoco una API concreta para las aplicaciones. En la figura 3.7 se muestra la arquitectura en capas para la comunicación de OPC UA para estos servicios.

La especificación de los diferentes mensajes que utiliza cada servicio, así como los mecanismos de seguridad a utilizar por cada uno de estos está definido en la parte 6 [69]. Toda estas especificaciones son necesarias para los desarrolladores y programadores que vayan a utilizar el stack de OPC UA para desarrollar un servidor o un cliente OPC UA.

El modelo de información básico está especificado en la parte 5 [70], este proporciona un framework en el que poder crear modelos que puedan ser utilizados en OPC UA. En él se define lo siguiente:

- Los puntos de entrada que los clientes utilizarán para navegar por las instancias y tipos de objetos de un servidor OPC UA.
- Los tipos básicos para construir jerarquías de objetos.
- La construcción de extensiones de tipos de objetos y tipos de datos.

Los perfiles se definen utilizando subconjuntos de características de OPC UA en la parte 7 [71]. Una aplicación UA puede contemplar solo un subconjunto de

las características del estándar UA, así esta puede solo utilizar un subconjunto de servicios, o utilizar lo ciertas funcionalidades de esos servicios. Las especificaciones definen subconjuntos a dos niveles. En el primer nivel están las unidades conformadas (CU), las cuales definen un pequeño conjunto de funcionalidades que siempre se utilizan de forma conjunta y pueden ser comprobadas y verificada como una unidad mediante la herramienta *Compliance Test Tools* (de la fundación OPC). El segundo nivel está compuesto por perfiles, los cuales no son más que una lista de unidades conformadas UC. Un perfil debe estar implementado completamente y será verificado durante la certificación de productos OPC UA. La lista de perfiles soportados y utilizados se intercambiará durante el establecimiento de la conexión entre el cliente y el servidor, y permitirá a las aplicaciones determinar si necesitan algunas características de las soportadas durante la comunicación establecida.

El modelo de información DA (*Data Access*), parte 8 [72], define cómo representar y utilizar los datos de proceso y sus características especiales como, por ejemplo, las unidades de ingeniería.

El modelo AC (*Alarms and Conditions*), parte 9 [73], específica las alarmas de proceso y las condiciones de monitorización que definen el estado de las máquinas y los tipos de eventos.

El modelo de información de programas, en la parte 10 [74], define los estados maquinas básicos para ejecutar, monitorizar y manipular programas.

El modelo de información HA (*Historical Access*), en la parte 11 [75], especifica como representar la información y la configuración de los datos y eventos históricos. Las agregaciones son utilizadas para realizar cálculos con los datos en bruto de acuerdo a la especificación en la parte 13 [76]. Las agregaciones son utilizadas para el acceso a datos históricos si bien se puede utilizar para la monitorización de valores de datos actuales. Y por último la parte 12 [77] define cómo encontrar el servidor en la red y cómo un cliente puede tener el acceso a la información pudiendo establecer una conexión con un servidor seguro.

3.4.3. EL MODELO DE COMUNICACIONES

Una de las características más interesantes que ofrece OPC UA es el modelo de transporte. Dicho modelo se basa en dos tipos de protocolos de transporte: UA

TCP y SOAP/HTTP. Estos dos protocolos se utilizan para establecer la conexión a nivel de aplicación entre las partes clientes y el servidor, y gracias a ellos se consigue la independencia del sistema operativo al no utilizar COM/DCOM como en el caso del OPC Clásico. También definen un canal seguro cuando se crea una sesión para cifrar las comunicaciones entre el cliente y el servidor.

El protocolo UA TCP.

El protocolo UA TCP proporciona una de las formas más fáciles y simples de realizar la comunicación entre elementos interconectados en una red , pero desde el punto de vista de la capa de transporte. Para conseguir que este protocolo funcione sobre TCP se necesitan tomar ciertas decisiones de diseño:

- Ajustar el tamaños de los buffer de envío y recepción de datos mediante su configuración en el nivel de aplicación.
- Establecer los diferentes puntos de acceso (endpoints) de los servidores OPC UA que pueden compartir dirección y puerto de acceso.
- Determinar los tipos de errores y los mecanismos de recuperación ante errores.

Un mensaje UA TCP está formado por una cabecera y un cuerpo como se indica en la figura 3.8. La cabecera del mensaje contiene información sobre el tipo de mensaje y la longitud del mismo. También puede cifrarse los mensajes UA TCP mediante una capa de seguridad al abrir un canal UA-Secure. El cuerpo del mensaje contiene el mensaje codificado de forma segura, el cual se reenvía a la capa superior. Además el cuerpo puede también albergar mensajes de conexión específicos de UA TCP utilizados para establecer una conexión de socket o para intercambiar información de error de conexión.

Hay definidos tres tipos de mensajes UA TCP los cuales se presentan brevemente a continuación:

- **Mensaje Hello:** Los mensajes Hello los envían los clientes al servidor para establecer una conexión mediante socket en un punto/puerto específico proporcionado por el servidor. Además de establecer el puerto de conexión, es necesario acordar tanto el tamaño de los buffer de envío y recepción de

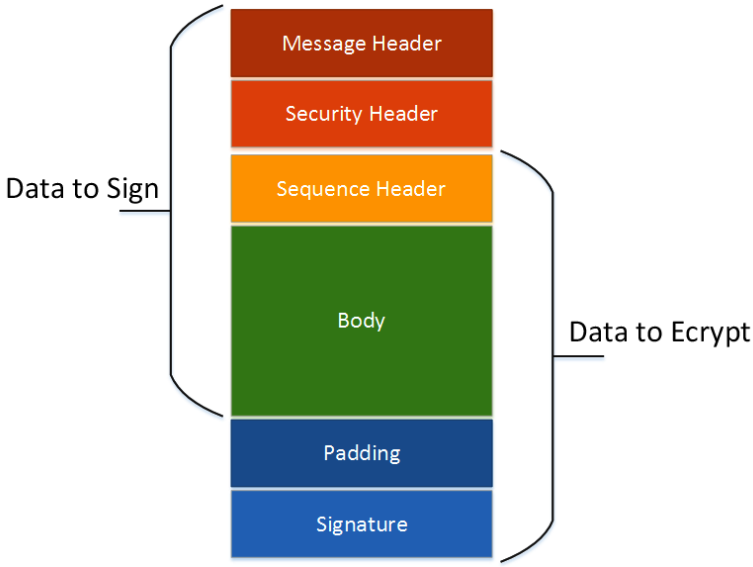


Figura 3.8: Estructura de mensaje UA TCP [69].

datos, así como el tamaño máximo del bloque del mensaje que se van a utilizar.

- **Mensajes de Reconocimiento:** Como respuesta a los mensajes de establecimiento de conexión (mensajes Hello), el servidor envían un mensaje de reconocimiento confirmando los tamaños de los buffers y del cuerpo del mensaje o proponiendo unos tamaños nuevos. Llegar a un acuerdo en estos tamaños es muy importante, tanto desde el punto de vista de la seguridad como de la disponibilidad del sistema. Tanto clientes como servidores conocen de ese modo la capacidad del otro de resistir ciertos ataques como los desbordamientos de buffer o la denegación del servicio.

- **Mensajes de Error:** Los mensajes de error proporcionan información sobre los errores que se producen. Estos mensajes se generan cuando se producen problemas relacionados con las conexiones entre los clientes y los servidores como por ejemplo, si un mensaje no puede ser procesado porque tiene un tamaño muy grande o porque el servidor esta sobrecargado.

El modelo de comunicaciones UA TCP define también los mecanismos de recuperación necesarios para hacer que las sesiones (sesiones OPC UA), que se establecen entre cliente y servidor sobrevivan a las interrupciones de red. Cuando

un cliente pierde la conexión de su socket, automáticamente se crea un nuevo socket al que se le asigna el mismo canal seguro, creado previamente en la primera autentificación del servidor. Las peticiones pendientes, así como las respuestas del servidor se almacenan en los diferentes buffers hasta que el nuevo socket esté disponible. Sin embargo, si este mecanismo de recuperación fallase (después de intentar reconectarnos un cierto número de veces), se genera un mensaje de error que se envía a la capa de aplicación.

SOAP/HTTP.

SOAP/HTTP (SOAP sobre HTTP) es un esquema de comunicación ampliamente aceptado en entornos de servicios web [49], porque es simple y utiliza un sistema de representación de datos basado en XML. Puede superar fácilmente los firewall, ya que SOAP utiliza los puertos estándares que utiliza el protocolo HTTP, por lo que no es necesario abrir ningún puerto adicional en los cortafuegos. Esto hace que las aplicaciones OPC UA puedan realizar una comunicación segura sobre Internet de la misma manera en la que los navegadores web se conectan con los servidores web [78].

SOAP es un protocolo de red de propósito general para el intercambio de datos entre sistema, que utiliza XML para la representación de datos y HTTP o TCP para el transporte de esos datos. Hay que tener en cuenta que el estándar OPC UA solo contempla HTTP como tecnología. Los mensajes SOAP están estructurados en una cabecera y un cuerpo como se indica en la figura 3.9.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2001/12/soap-envelope">
    <s:Header>
    </s:Header>
    <s:Body>
    <!-- Either XML encoded
         or UA Binary encoded service message -->
    </s:Body>
</s:Envelope>
```

Figura 3.9: Estructura de mensaje SOAP [69].

La cabecera de un mensaje SOAP contiene la información de direccionamiento y ruteo, mientras que el cuerpo encierra la carga útil que debe de transportarse

(información). El estándar OPC no define cabeceras específicas por lo que utiliza las cabeceras definidas en la especificación WS-Addressing[79].

Aunque el mensaje SOAP es una estructura de datos basada en XML, este es capaz de transportar un mensaje de servicio XML dentro del cuerpo de un mensaje binario OPC UA. Las solicitudes de servicio OPC UA integradas en las solicitudes SOAP se encapsulan en peticiones HTTP con el comando POST, y las respuestas OPC UA a través de respuestas HTTP.

También se puede utilizar HTTPS que utiliza TLS para proteger los intercambios de datos HTTP. Sin embargo, cuando ya se aplica uno de los canales seguros, previamente creados, esto puede suponer una sobrecarga innecesaria al utilizar HTTPS sobre una canal seguro ya, pero puede que en algunos escenarios en los que no se necesitan todos los requisitos de OPC UA, pueda tener sentido no utilizar la seguridad UA y utilizar HTTP sobre TLS en su lugar.

3.4.4. EL MODELO DE INFORMACIÓN

El OPC clásico maneja los datos de proceso en crudo y la única meta-información adicional que incluye sobre los datos que utiliza, a parte del tipo de dato, es el nombre de la señal y las unidades de ingeniería. Así, por ejemplo, en el caso de un sensor de temperatura dispondríamos de un valor analógico del sensor, el nombre de la señal en el servidor y la unidad de medida en grados centígrados o Fahrenheit.

El nuevo estándar OPC UA ofrece un modelo de datos mucho más rico, pudiendo crear modelos de información mucho más complejos en los que alojar meta-información de todos los datos almacenados en el servidor OPC UA. Los clientes OPC UA pueden acceder a dicha información para conocer los datos manejados por los dispositivos industriales, obtener información semántica de los datos y relacionarlos con los dispositivos físicos subyacentes de donde proceden como, por ejemplo un sensor de temperatura o una válvula [3].

Las especificaciones básicas del estándar OPC UA solo proporcionan la infraestructura del modelo de información [67] [70], pero no especifica cómo se deben de desarrollar o construir dichos modelos de información [80]. La información puede ser modelada por los diferentes ingenieros, programadores o usuarios de forma diferente para adaptarla a sus necesidades, por lo que puede que estos mo-

delos sean difíciles de interpretar para las aplicaciones cliente, especialmente si estas son utilizadas por terceras personas que no han participado en el proceso de modelado [81] [82].

Para evitar esta situación, se pueden utilizar modelos de información específicos propuestos por la fundación OPC que incluyen determinados modelos para la exposición de la información de dispositivos y de tipos de dispositivos en OPC UA [83] [84]. La ventaja de este enfoque es que el cliente podría acceder a los datos de dichos dispositivos utilizando un mismo modelo de información, aunque los datos provengan de dispositivos industriales de distintos fabricantes. Dichos modelos pueden luego extenderse para adaptarse a las necesidades específicas de la planta [85]. Así, por ejemplo, podemos extender este mismo modelo a otros escenarios en los que se necesite proporcionar datos de proceso organizados a otros sistemas que los consuman como, por ejemplo, a sistemas MES o ERP contemplados en la norma ISA 95 [86].

La transparencia con los datos también está asegurada, ya que el cliente OPC-UA puede acceder y conocer los tipos de datos almacenados en el servidor OPC UA antes de consumirlos.

Las características y principios básicos que se tienen en cuenta a la hora de crear modelos en OPC UA son los siguientes:

- Utilización de técnicas orientadas a objetos que incluyen la utilización de jerarquías de tipos y herencia.
- Acceso a la información de la definición de tipos como si fueran instancias.
- Organización del modelo en una red de nodos en forma de malla, posibilitando varias formas de conexión ente nodos. Esta característica facilita el acceso a la misma información de distintas formas simplemente cambiando la jerarquía entre nodos.
- Extensibilidad de la jerarquía de nodos y los tipos de referencias entre nodos.
- No hay limitaciones en como modelar la información. Se pueden crear modelos apropiados para cualquier tipo de necesidad, aplicación o escenario.

- Los modelos de información de OPC UA están siempre alojados en el servidor OPC-UA, y pueden ser consultados/modificados por los clientes OPC-UA

Como vamos a ver a continuación, todo el modelo de información que ofrece OPC UA se construye en base a dos elementos básicos: *nodos y referencias entre nodos*.

ELEMENTOS BÁSICOS DEL MODELO DE INFORMACIÓN DE OPC UA: NODOS

Los nodos los podemos clasificar en dos grandes grupos: nodos que representan la definición de tipos y nodos que son instanciaciones de esos tipos. Todos los nodos están representados mediante una clase de nodo o *NodeClass*, la cual especifica de qué tipo es el nodo en cuestión. El tipo de un nodo puede variar dependiendo de la funcionalidad o del propósito específico del mismo.

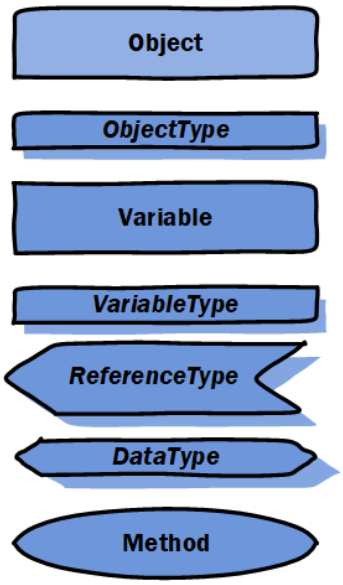


Figura 3.10: Diferentes tipos de nodos representables en OPC UA y su notación gráfica [70].

Todos los nodos se definen en un espacio de direcciones de un servidor OPC UA para que puedan ser accesibles desde una aplicación cliente. Así podemos tener nodos que representen tipos de datos *(DataType)* o tipos de objetos *(ObjectType)*, nodos que sean instancias de estos objetos (Object), nodos contenedores u organizativos o nodos que simplemente sean métodos *(Method)* o variables

(Variable). En la figura 3.10 podemos ver un resumen de los principales nodos que contempla el estándar UA.

Todos estos tipos de nodos poseen ciertos atributos, que describen las propiedades de dicho nodo. El número y tipo de estos atributos están definidos por defecto y variarán en función del *NodeClass* del cual el nodo sea instancia o extienda. Aunque los atributos de un nodo vienen determinados por su *NodeClass*, existen varios atributos comunes que siempre deben de estar presentes en un nodo independientemente de la clase del nodo o del tipo de este.

Un resumen de estos atributos comunes se puede ver en la tabla 3.2. De entre todos estos atributos, quizás el más importante sea *NodeID*, ya que este es el único que identifica unívocamente a un nodo. Cuando un cliente navega o consulta el espacio de direcciones, el servidor devuelve el conjunto de *NodeID*s que ha solicitado el cliente como consecuencia de esa operación de navegación o de esa consulta. Este conjunto de *NodeID*s son utilizados por el cliente para direccionar estos nodos y poder así acceder a los servicios que proporcionan los servidores.

Clases predefinidas para la instanciación de nodos:
El modelo de información de OPC UA proporciona un conjunto de tipos de nodos predefinidos como los que se muestra en la figura 3.10 para la construcción de modelos. Este conjunto de nodos engloba todas las clases (definidas por *Type*) a partir de las cuales se pueden definir nodos que engloban las instancias de las clases (el resto). Ambos tipos de nodos se modelan dentro del espacio de direcciones de OPC UA para que posteriormente puedan ser accedidos por los clientes OPC-UA.

De entre todas estas *NodeClasses*, quizás las más importantes en OPC UA o por los menos las que más versatilidad nos pueden ofrecer, son *Object*, *Variable* y *Method*. Conceptualmente los objetos se caracterizan en OPC-UA por tener variables y métodos como en programación orientada a objetos (OOP, Object Oriented Programming en inglés) y, además, pueden disparar eventos. En la figura 3.11 se muestra esquemáticamente la representación de un objeto de OPC UA.

Atributo	Tipo de dato	Descripción
NodeId	NodeId	Identifica de forma única un nodo en un servidor OPC UA y se utiliza para direccionar el nodo por parte de los servicios que ofrece un servidor OPC UA.
NodeClass	NodeClass	Identifica la clase de nodo a la que pertenece un nodo dado.
BrowseName	QualifiedName	Identifica el nodo cuando se navega por el servidor OPC UA.
DisplayName	LocalizedText	Contiene el nombre del nodo que se debe utilizar en una interfaz de usuario.
Description	LocalizedText	Es opcional y contiene una descripción del nodo.
WriteMask	UInt32	Es opcional y especifica qué Atributos del Nodo pueden ser modificados por un cliente OPC UA.
UserWriteMask	UInt32	Es opcional y especifica qué Atributos del Nodo pueden ser conectado modificados por el usuario que está en ese momento al servidor.

Tabla 3.2: Principales atributos que tiene definidos los nodos representados por NodeClass en OPC UA [3].

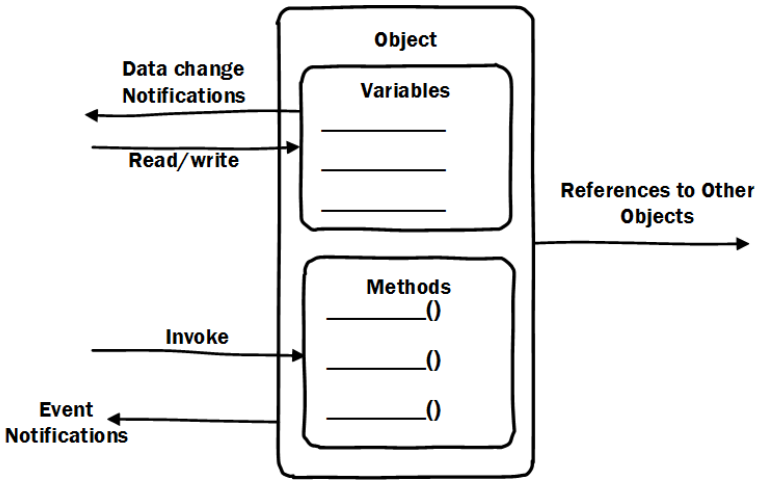


Figura 3.11: Representación de un nodo Object con sus atributos (variables en OPC UA) y operaciones (métodos en OPC UCA). [3]

Un objeto es una entidad software que nos permite encapsular un concepto,

cosa o idea en base al conjunto de datos relacionado que lo caracteriza y al conjunto de tareas que determina las funciones o actividades que se pueden llevar a cabo sobre el propio objeto. Los objetos nos permiten organizar y estructurar de forma lógica el espacio de direcciones de OPC UA, ya que un objeto puede incluir variables, métodos u otros objetos. Al igual que en OOP los métodos de un objeto se invocan siempre en el contexto del objeto. El objeto además puede ser por si mismo un notificador de eventos *EventNotifier*, de forma que cualquier cliente puede suscribirse, recibiendo así un evento por ejemplo, cada vez que se produzca un cambio.

Los nodos del tipo *Variable* representan un valor, cuyo tipo dependerá del tipo asociado a la variable definida. Los clientes pueden leer y escribir el valor de una variable, así como subscribirse para recibir los cambios de dicho valor. Una variable se utiliza, por ejemplo, para representar las medidas que se obtienen de magnitudes del proceso productivo, como por ejemplo la temperatura o la presión.

En general se puede utilizar para exponer cualquier dato que se encuentre asociado a objetos u otro tipo de nodos. La clase de nodo *Method* representa un método, es decir, un comportamiento ejecutable que puede ser llamado o invocado desde un cliente y que devuelve un resultado. Cada Método puede tener argumentos de entrada y argumentos de salida para indicar el resultado que devuelven al cliente.

Un método está pensado para ser ejecutado siguiendo un esquema petición-respuesta, de modo que el cliente utiliza el servicio de llamadas para invocar el método y como respuesta el cliente recibe el resultado. Mientras no recibe la respuesta el cliente se mantiene bloqueado. Ejemplos de métodos pueden ser, abrir y cerrar una válvula o parar/arrancar un motor, es decir, métodos que se ejecutan de forma atómica (con un coste de tiempo cuasi cero) o que estén asociados a una acción especial como un disparador bien definido en el servidor. No existe una forma estándar a la hora de implementar un método en un servidor OPC UA.

Cuando el servidor necesita manejar procesos con tiempos de ejecución muy largos, que son lanzados y controlados por un cliente, se debe utilizar programas de OPC UA (hay una especificación OPC UA para ello [74]) que nos permite estructurar mejor la lógica que se tiene que ejecutar, así como la sincronización de

la respuesta con el cliente.

Clases para la definición de tipos de Nodos

Además del conjunto de tipos de nodos predefinidos, OPC UA también contempla la posibilidad de crear nuevos tipos de clases de nodos, pudiendo así crear nuevos tipos de instanciaciones de nodos a partir de estos nuevos tipos de clases. Esto nos permite extender el modelo base con la definición de nuestros propios tipos de nodos adecuándolos a nuestras necesidades, haciendo que estos se correspondan mejor a elementos industriales concretos, y facilitando por consiguiente la creación de modelos específicos para aplicaciones industriales concretas [4]. El concepto base para definir el tipo de dato es *DataTypes*. Como ocurre con todos los conceptos de OPC UA, exceptuando las referencias, se representan mediante nodos dentro del espacio de direcciones. En la figura 3.12 se puede ver de forma jerarquizada los *DataTypes* más importantes de OPC UA.

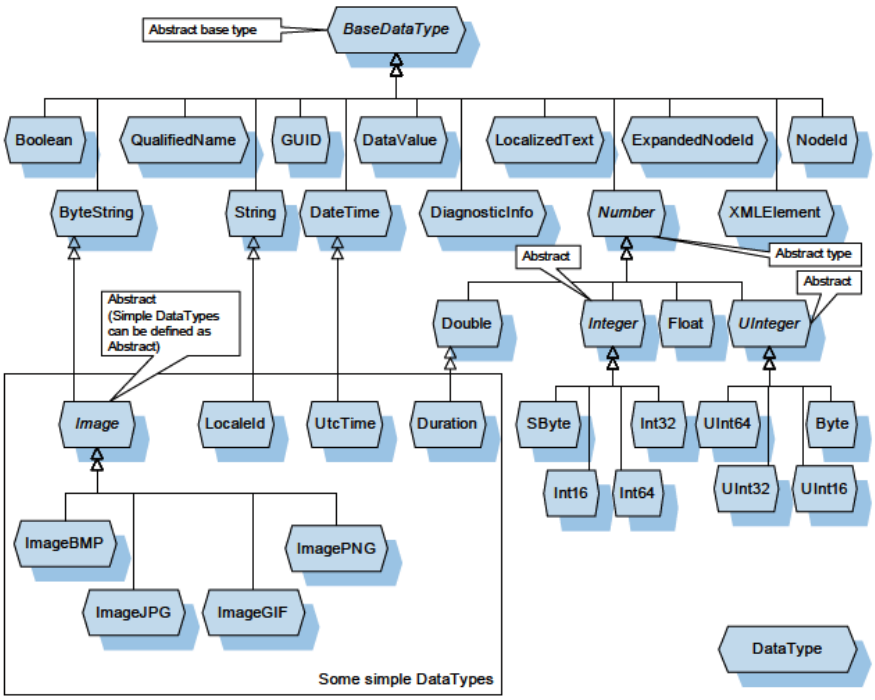


Figura 3.12: Parte de la jerarquía de DataTypes existentes en OPC UA [3].

Además de los tipos de datos predefinidos de OPC UA String, Integer o Float,

también se pueden definir tipos de clases de nodos del resto de clases de OPC UA como *ObjectType* o *VariableType*". Así, *ObjecType* nos permite definir nuevos tipos de objetos simples o complejos a partir de los cuales podremos crear nodos objetos instanciados (*Objects*). Los consumidores de estos nuevos tipos de objetos se pueden olvidar del funcionamiento interno de los mismos, solo deben de preocuparse de como invocar sus métodos o como leer sus variables. Por otra parte, con *VariableType* podemos definir variables simples o complejas donde se expone el valor de los objetos. En la figura 3.13 se pueden ver con detalle los atributos que caracterizan al nodo genérico *BaseNode*, así como los tipos de clases de nodos y de tipos de nodos que se definen por herencia a partir nodo *BaseNode*.

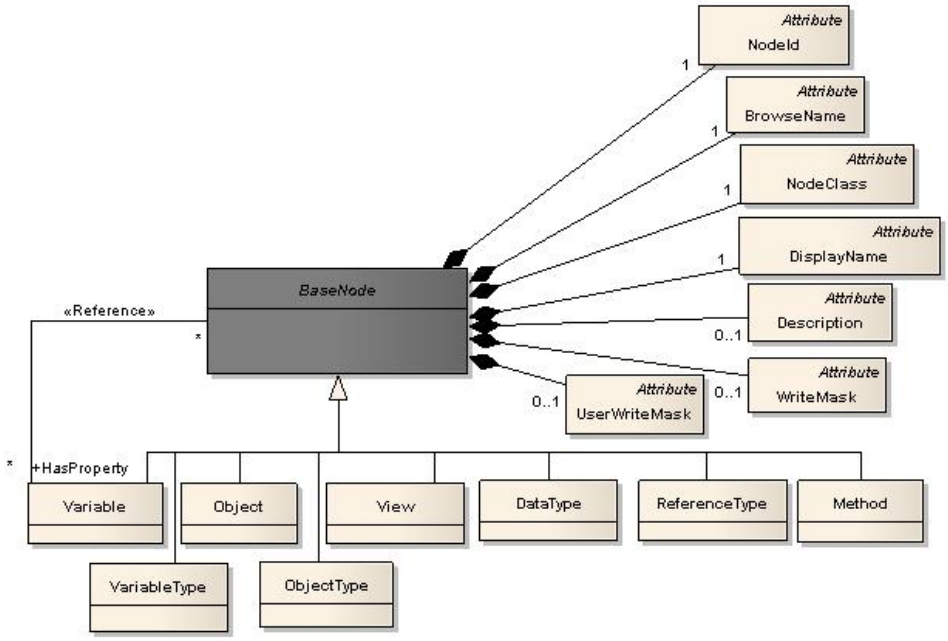


Figura 3.13: Modelo de información de los nodos base de OPC UA [67].

ELEMENTOS BÁSICOS DEL MODELO DE INFORMACIÓN DE OPC UA: REFERENCIAS

Una referencia describe la relación entre exactamente dos nodos. Por lo tanto una *referencia* puede identificarse de forma única reconociendo el nodo fuente y el nodo destino, el tipo de referencia (semántica de la referencia) y la navegabilidad o dirección hacia donde fluyen los datos. En la figura 3.14 podemos ver un ejemplo de varios nodos referenciados entre sí.

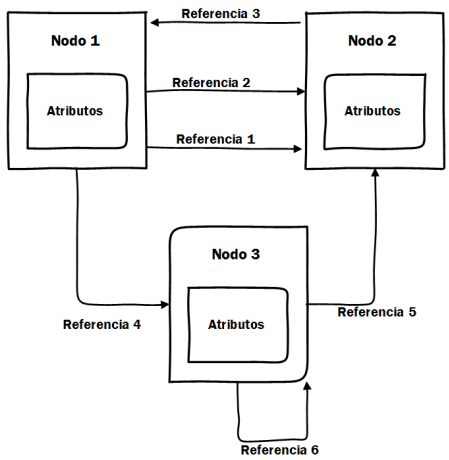


Figura 3.14: Ejemplo de nodos y de referencias entre nodos suponiendo origen la izquierda y destino la derecha[3].

Una de las características que diferencia a las referencias de los nodos, es que estas no puede ser accedidas directamente, solo se puede acceder a ella de forma indirecta navegando entre nodos y teniendo en cuenta las referencias que hay entre dichos nodos. Las referencias no se representan como nodos y no pueden contener atributos ni propiedades. En la práctica, un servidor puede exponer las referencias sólo en una dirección, pudiendo estar relacionados nodos que se encuentran en el mismo servidor o nodos que se encuentran en otro servidor OPC UA. Podemos pensar en el concepto de referencia como si fuesen un puntero que sale de un nodo y que apunta a otro nodo almacenando el *NodeID* de este último. En la figura 3.15 se puede ver un ejemplo de cómo se realiza una referencia entre dos nodos.

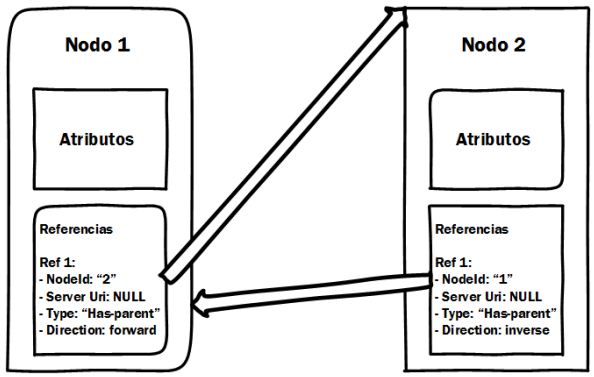


Figura 3.15: Ejemplo de referencias como punteros a nodos [3].

Las referencias pueden ser simétricas o asimétricas (no simétricas). Una referencia no simétrica es aquella que tiene significado en una sola dirección como, por ejemplo, *"has-parent"* en una dirección y *"is-child-of"* en otra dirección. En cambio una referencia "no simétrica" tiene el mismo significado en ambas direcciones. Para las referencias simétricas la dirección de la referencia carece de importancia, mientras que para las no simétricas la dirección de la referencia si es importante. Aunque las referencias conectan dos Nodos, los servidores OPC UA pueden exponer sólo una dirección de la referencia, por ejemplo, sólo el puntero del Nodo 1 al Nodo 2 en la figura 3.15 y no la dirección inversa. Si sólo se expone una dirección, la referencia se denomina unidireccional, y bidireccional cuando se exponen ambas direcciones. Las referencias pueden relacionar nodos que no existan en el propio servidor o en otro servidor en el momento de ser consultadas. Por ello los clientes deben de soportar escenarios de funcionamiento en los que una navegación entre nodos siguiendo una referencia no encuentre el nodo destino de dicha referencia. Las referencias por lo general no suelen estar ordenadas, por lo que preguntar a un nodo por sus referencias puede tener resultados distintos en lo que se refiera al orden del conjunto de las mismas. Sin embargo existen referencias que definen un orden entre tipos de referencias, como por ejemplo *HasOrderedComponent*. En este caso el servidor tiene siempre que devolver el conjunto de referencias solicitadas en el mismo orden.

Referencias predefinidas en OPC UA

Como se puede observar en la figura 3.16, OPC UA ofrece ya un conjunto predefinido de referencias que podemos utilizar para organizar los elementos de nuestro modelo. Además lo podemos hacer de forma jerárquica o no, según sea el tipo de referencia. Los *ReferenceTypes* que se expresan utilizando letras en cursiva son abstractos y solo se usan para propósitos organizativos. Como podemos observar existen principalmente dos subtipos de referencias: referencias jerárquicas y referencias no jerárquicas. Entre estas referencias podemos encontrar *Organizes* y *HasChild* como las dos referencias principales dentro del grupo de las referencias jerárquicas y *HasTypeDefinition* en el grupo de las referencias no jerárquicas.

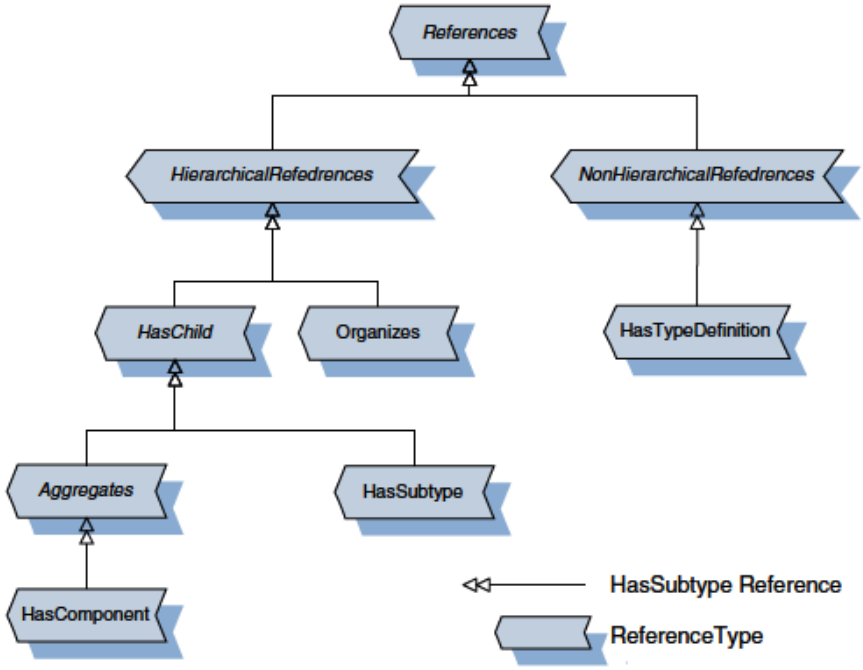


Figura 3.16: Jerarquía de referencias de OPC UA [3].

Entre los tipos de referencias predefinidas en OPC UA podemos distinguir los siguientes tipos:

- *Organizes* se utiliza cuando dos tipos de nodos deben estar conectados de forma jerárquica, pero sin ninguna definición semántica.
- *HasChild* se utiliza cuando dos nodos deben estar conectados de forma jerárquica y hay además una definición semántica, la cual consiste en una relación Padre-Hijo entre dichos nodos.
- *HasTypeDefinition* se utiliza para hacer referencia al tipo de una instancia, por ejemplo, al tipo de objeto o una variable. Obviamente este tipo de referencia no puede ser jerárquica, ya que un objeto o una variable se pueden encontrar, por ejemplo, en cualquier posición dentro de una relación jerárquica y será complicado relacionar este objeto con su tipo mediante una relación de tipo jerárquico.

Como subtipo de estas referencias tenemos también las siguientes referencias:

- *HasComponent* se utiliza para indicar que un nodo puede formar parte de otro.

- *HasProperty* se utiliza para indicar que un nodo puede ser una propiedad de otro.

- *HasSubType* esta referencia indica que un nodo es subtipo de otro.

- *HasEventSource* se utiliza para indicar que un nodo está subscrito a los eventos generados por el otro.

Estas referencias posen también una notación gráfica concreta, que se puede ver en la siguiente figura:

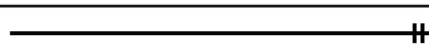
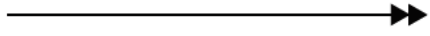
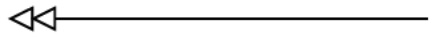
Reference	Notation	InverseName
HasComponent		ComponentOf
HasProperty		PropertyOf
HasTypeDefinition		Defines
HasSubtype		SubtypeOf
HasEventSource		EventSourceOf
Organizes		OrganizesBy

Figura 3.17: Referencias en OPC UA y su notación gráfica [3].

Aunque las referencias no son nodos y por lo tanto no tienen atributos (sólo en nodos), todos los tipos de referencias (*ReferenceTypes*) se exponen como nodos. Esto permite a los clientes obtener información sobre las referencias utilizadas dentro del modelo o modelos que aloja un servidor OPC UA. Por ello, aunque no tenga atributos explícitamente, al ser expuestas como nodos estas acaban teniendo una serie de atributos. En la tabla 3.3 podemos ver un resumen de los atributos que describen la definición de un tipo de referencia.

Atributo	Tipo de dato	Descripción
Contiene todos los atributos comunes definidos en la Tabla 3.2		
IsAbstract	Boolean	Especifica si ReferenceType se puede utilizar para referencias o sólo se utiliza para fines organizativos en la jerarquía ReferenceType.
Symmetric	Boolean	Indica si la Referencia es simétrica, es decir, si el significado es el mismo en ambas direcciones.
InverseName	LocalizedText	Este atributo opcional especifica la semántica de la Referencia en sentido inverso. Sólo se puede aplicar para referencias no simétricas y debe proporcionarse si dicho ReferenceType no es abstracto.

Tabla 3.3: Atributos adicionales para ReferenceTypes [3].

Además de estos atributos, los tipos de referencias tienen los atributos comunes definidos para un nodo 3.3. Algunos de estos atributos presentan restricciones adicionales cuando se utilizan como parte de los atributos de una referencia. Por ejemplo el atributo *BrowseName* de un *ReferenceType* debe ser único en un servidor OPC UA para evitar confusiones con otros ReferenceTypes con semántica diferente pero con el mismo nombre.

Definición de Tipos de Referencias

Para describir el significado de una referencia o crear nuevos tipos de referencias se puede utilizar la clase de nodo (*nodeClass*) *RefenceTypes*. Las especificaciones de OPC UA predefinen ya de por si un conjunto de *ReferenceTypes* o tipos de referencias. Algunas de ellas son fundamentales como, por ejemplo, para describir una jerarquía. Pero el concepto de tipo de referencia es extensible, por lo que dentro de un servidor OPC UA se pueden definir tipos de referencias propios, al igual que ocurría con los tipos de objetos. La organización de los tipos de referencias se gestiona jerárquicamente.

EJEMPLO DE UTILIZACIÓN DEL MODELO DE INFORMACIÓN.

Se han presentado los conceptos más importantes del modelo de información de OPC UA en base a nodos. Todos estos nodos se sustentan a partir del concepto clase de nodo definido por *NodeClass*. La descripción de dichos nodos junto con

sus atributos forma el denominado metamodelo de OPC UA. La descripción sintáctica de todos los tipos de nodos que conforman el metamodelo de OPC UA y sus restricciones se administran con el Modelo de Espacio de direcciones o *Address Space Model*. Dicho modelo establece el dominio específico con la definición de tipos y restricciones, a partir de los cuales podremos instanciar cada uno de los tipos de objetos y de datos en el Modelo de Información.

Por último, tendremos un nivel más que se corresponde con la concretización de cada objeto o dato que se alojan en el propio servidor con los datos específicos. Los distintos niveles de abstracción con los que se trabaja en OPC-UA se resumen muy bien en la figura 3.18, donde podemos ver como se organizan todos estos objectos. Si hacemos una equivalencia con el lenguaje UML, podríamos decir que el modelo de espacio de direcciones se corresponde con el documento de superestructura de UML que define los conceptos definidos en los diagramas de representación (diagrama de estados, diagrama de clases). El modelo de información hace referencia a los diagramas de clases, de secuencia, o cualquier otro, específico del problema que se va a construir, y por último, con el nivel de datos tendremos las instanciaciones concretas de cada uno de los conceptos definidos en el modelo de información.

El espacio de direcciones de OPC UA, junto con el modelo de información, proporciona una forma estándar de crear y definir nodos, los cuales estarán basados en los mismos conceptos organizados de OPC UA. Todas las aplicaciones clientes OPC UA que se conecten a un servidor OPC UA podrán utilizar estos objetos, navegar por ellos y aprovechar las funcionalidades que estos ofrecen.

Los modelos creados a partir del meta modelo de información de OPC UA y que definan nuevos tipos de objetos y conceptos, pueden ser vistos a su vez también como meta modelos pero con un nivel de abstracción menor, ya que en estos modelos se definen nuevos tipos de objetos, además de sus relaciones, pero de una manera más específica y acorde a las necesidades de la aplicación o proceso industrial que se pretende representar. Para ver las características y capacidades del proceso de modelado en OPC UA vamos construir dos ejemplos de aplicación, los cuales van a modelar el mismo proceso industrial pero con enfoques diferentes.

Para crear un modelo dentro de un servidor OPC UA, podemos proceder de

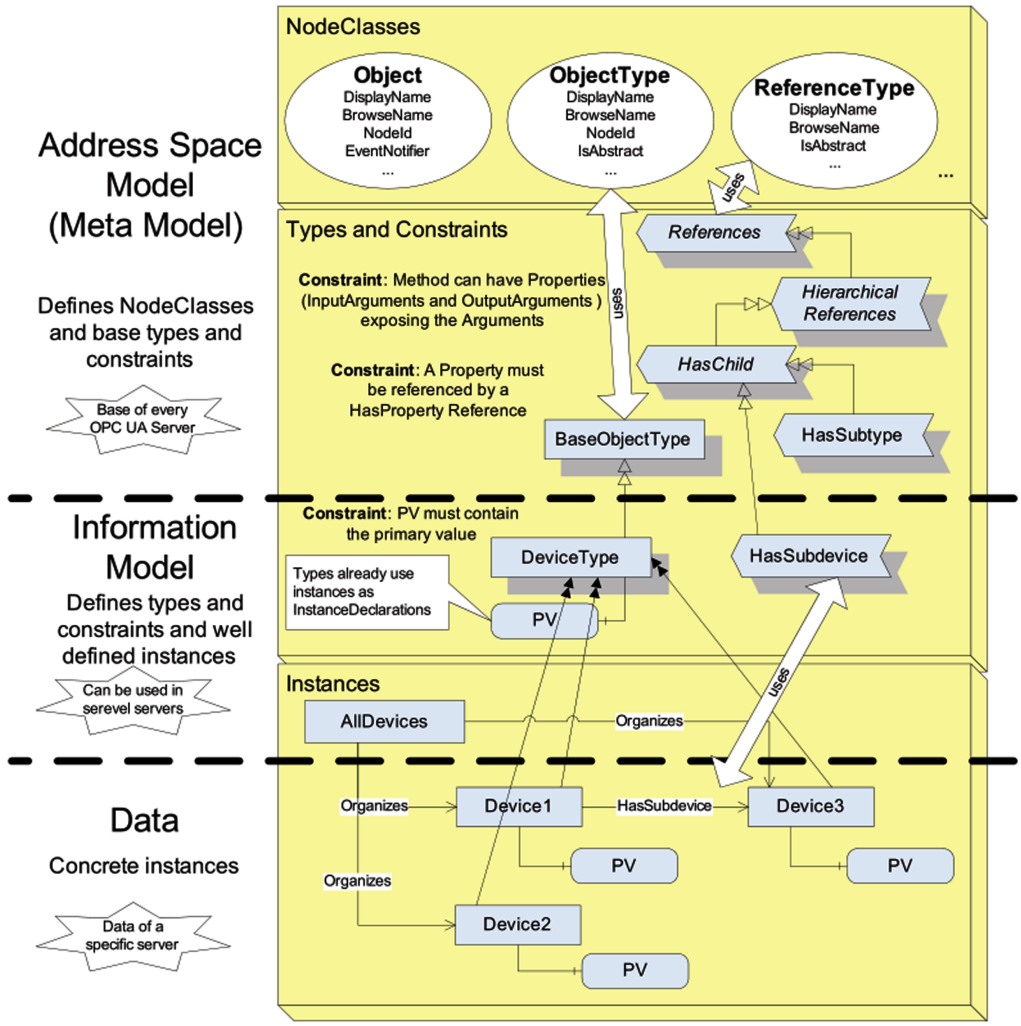


Figura 3.18: Representación del Modelo de espacio de direcciones (AddressSpace), Modelo de información y Datos [3].

dos formas. La primera consistiría en utilizar los conceptos ya predefinidos en el modelo de información de OPC UA como, por ejemplo, *Object*, y crear a partir de ellos todos los nodos que necesitemos para cubrir las necesidades y funcionalidades específicas que deseemos mostrar a las aplicaciones clientes. La segunda aproximación que podríamos utilizar tendría un paso previo que consistiría en crear un tipo de clase de nodo (objetos fundamentalmente) dentro del modelo de información de OPC UA. El tipo de clase de nodo como *ObjectType* nos permitiría determinar que variables y métodos caracterizan dicho tipo de objeto. Posteriormente podríamos instanciar objetos a partir de la definición del *ObjectType*, los cuales se adaptaran mejor a las necesidades específicas del sistema final, ya que han sido concebidos específicamente para un sistema o aplicación concreta.

Ambas aproximaciones tienen sus ventajas y sus inconvenientes. La elección de una u otra vendrá dada por las necesidades y características del sistema a modelar, así como las restricciones de tiempo de desarrollo y funcionalidad que se tenga. Por ejemplo, la primera aproximación permite desplegar modelos dentro de un servidor OPC UA de forma muy rápida al utilizar conceptos ya predefinidos en el propio modelo de información de OPC UA, por lo que los tiempos se reducen. Sin embargo, puede que no cubramos todas las especificaciones funcionales requeridas, cuestión que se resuelve mejor con la segunda aproximación, ya que estaríamos utilizando conceptos u objetos definidos específicamente por el usuario. A cambio tendría que invertir un esfuerzo extra para su definición

A continuación, vamos a realizar un modelo utilizando ambas aproximaciones. Para ello se va a modelar un pequeño sistemas industrial consistente en un transportador de rodillos controlado por un motor (Start/Stop), una fotocélula que detecta el paso de objetos y una barrera de Stop que bloqueará y limitará el paso de los mismos. En la figura 3.19 se puede ver todos estos elementos.

Modelado utilizando los conceptos predefinidos de OPC UA

En este primer ejemplo de modelado se ha seguido la estrategia propuesta en la primera aproximación y como se puede ver en la figura 3.20 se ha utilizado el concepto ya existente dentro del espacio de direcciones de *Object*, con el que se han modelado todos los nodos propuestos en el ejemplo. En concreto se ha creado una jerarquía de instancias de *BaseObjectType*, en el que el elemento raíz

Figura 3.19: Elementos del proceso industrial a modelar en OPC UA. Sistema creado utilizando Factory I/O [87].

es la instancia *CintaRodillos*, esta instancia no posee una variable *Estado* y tres métodos. Los métodos *Start* y *Stop* se utilizaran para arrancar y parar el motor de la cinta transportadora, mientras que el método *getStatus* servirá para devolver el estado almacenado en la variable *Estado*. Dicha variable nos va servir para indicar el estado de la cinta, si está en marcha o parado.

Este elemento tiene dos componentes más asociados mediante la relación *HasComponent* que son *DetectorPaso* y *BarreraStop*. El primero es una representación de la fotocélula que detecta el paso de objetos como cajas o pales, por lo que solo tiene una variable estado, y un método que nos devuelve el estado almacenado en estado (detección/no detección). Por otra parte, el objeto *BarreraStop* también posee una variable estado junto con el método *getStatus* que nos indicara si el elemento está abierto (paso libre) o cerrado (paso bloqueado). Además, posee dos métodos *Open* y *Close* que se utilizaran para subir o bajar la barrera de bloque del sistema. En ninguno de estos nodos se han añadido atributos adicionales que no sean los propios de la clase *BaseObjectType*.

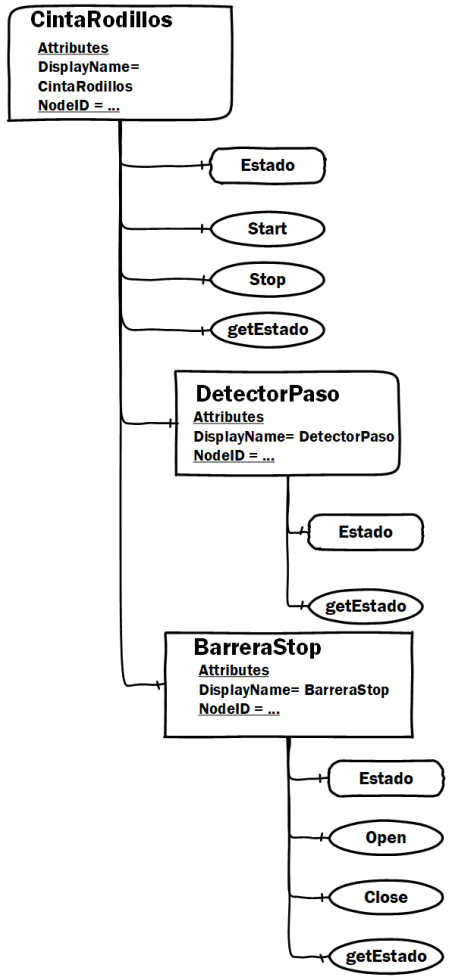


Figura 3.20: Ejemplo de modelado de un sistema industrial utilizando Objectos predefinidos en OPC UA.

Modelado utilizando un metamodelo específico en OPC UA

En esta segunda aproximación primero hay que crear un modelo en el que se organizarán los conceptos necesarios para modelar los nodos de nuestro sistema industrial. Este modelo definirá los nuevos tipos de objetos que podremos instanciar para un caso particular concreto. Dicho modelo se apoyará en conceptos ya existentes del metamodelo que ofrece OPC UA, como pueden ser en nuestro caso *BaseObjectType*. En la figura 3.21 podemos ver tanto el metamodelo definido, así como la instanciación a partir de este metamodelo.

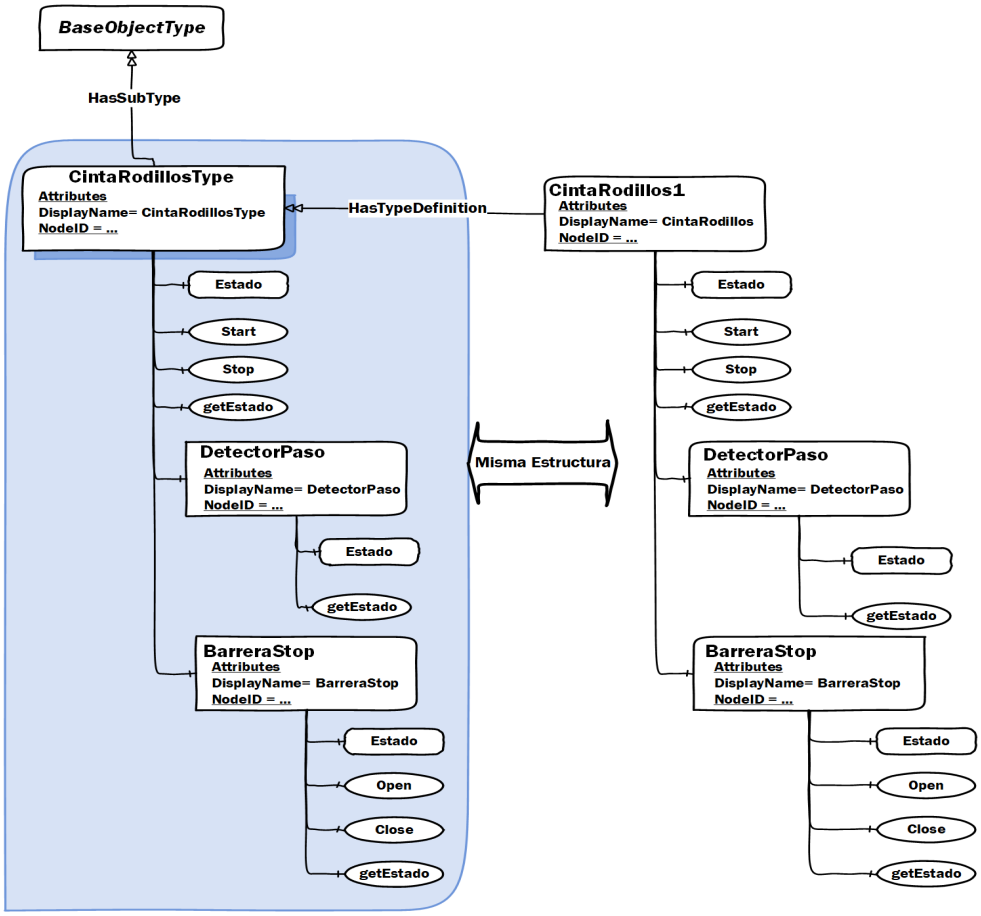


Figura 3.21: Ejemplo de modelado de un sistema industrial utilizando Objetos predefinidos en OPC UA.

El tipo de objeto *CintaRodillosType* se ha construido utilizando la misma estructura que en el ejemplo anterior. Dicho tipo de objeto está relacionada con la clase de nodo *BaseObjectType* mediante la relación *HasTypeDefinition*. De este modo heredamos todas las propiedades de esta clase de nodo, haciendo que nuestro nuevo tipo sea un tipo de objeto nuevo dentro del modelo de información de OPC UA. La estructura de este nuevo tipo *CintaRodillosType* es la misma que se ha explicado en el apartado anterior. Hay un objeto principal con dos componentes de tipo objeto, uno para el detector y otro para la barrera de stop.

Ahora tenemos que realizar un segundo paso, no basta con crear el nuevo tipo de dato *CintaRodillosType*, ya que hasta ahora solo hemos definido un tipo de objeto. Tenemos que instanciar un objeto a partir del nuevo tipo creado. Para

nuestro ejemplo hemos instanciado un nuevo objeto *CintaRodillo1*, el cual hereda toda la estructura de elementos del tipo *CintaRodillosType* como se puede ver en la figura 3.21. De este modo hemos conseguido modelar nuestro pequeño sistema industrial y cubrir todas sus funcionalidades. El resultado final es muy parecido al del enfoque anterior.

BUENAS PRÁCTICAS DE MODELADO EN OPC UA.

Como ya se ha comentado las dos aproximaciones anteriores tiene sus ventajas y sus inconvenientes y en este sentido lo más importante es saber cuándo utilizar una u otra. Para ello debemos de tener en cuenta varias factores como son: el proceso industrial que pretendemos modelar, la complejidad del mismo, la homogeneidad/heterogeneidad de sus elementos y las necesidades de escalabilidad y crecimiento del mismo.

En un proceso industrial en el que suele haber muchos dispositivos iguales como, por ejemplo, varias cintas de transporte de rodillos como las del ejemplo anterior, la segunda aproximación puede ser muy interesante, ya que tan solo tenemos que ir instanciando los nodos de los tipos recién definidos, creando las relaciones pertinentes entre ellos. Pero si nos enfrentamos a un sistema industrial en el que todos los nodos son distintos, quizás no merezca la pena crear tipos de objetos específicos para luego instanciar una instancia de cada tipo de objeto; en este caso el primer enfoque sería más acertado.

En cuanto a la escalabilidad quizás el segundo enfoque proporcione un modelo más escalable siempre que haya elementos en el proceso industrial que tengan las mismas características. Por ejemplo, si montamos una cinta de rodillos nueva con los mismos componentes tan solo tenemos que instanciar un nuevo objeto, y no tres con sus respectivas relaciones como en el primer enfoque.

Un análisis previo del sistema industrial así como de sus futuras necesidades serán clave para construir un modelo en OPC UA, independientemente de la estrategia que sigamos. Seguir un proceso sistemático para la identificación de los elementos del sistema industrial así como sus interrelaciones facilitará la obtención de un modelo valido que se ajuste a las necesidades del proceso industrial. Para que el modelo sea lo suficientemente estable será importante tener en cuenta a todos los ingenieros que intervienen en el diseño, despliegue y puesta en marcha del sistema industrial, son los ingenieros mecánicos, ingenieros de procesos,

ingenieros eléctricos o programadores.

3.4.5. MODELADO DEL COMPORTAMIENTO EN OPC UA: PROGRAMAS

El modelo de información que ofrece OPC UA nos proporciona una manera estructurada de organizar los datos/señales de un sistema industrial mediante la utilización de modelos basados en los conceptos que ofrece el propio modelo de información de OPC UA, o creando nuevos tipos de objetos que se adapten específicamente a nuestras necesidades. Pero OPC UA además de este modelo de información también ofrece un modelo para crear máquinas de estados, con el que podemos recoger el comportamiento de los diferentes nodos que forman parte del modelo de información. Para ello OPC UA proporciona en su especificación número 10 [74] el concepto de programa.

Genéricamente una máquina de estados finita[88] puede ser vista como una forma de expresar el comportamiento de un sistema en base a la transición a través de varios estados, siendo cada estado una representación de la situación temporal en la que se encuentra el sistema en un momento dado. Estos estados se encuentran interconectados mediante transiciones. Las transiciones representan las condiciones o circunstancias que hacen que la máquina de estados transite de un estado a otro. En la figura 3.22 podemos ver una representación para una máquina de estados finita.

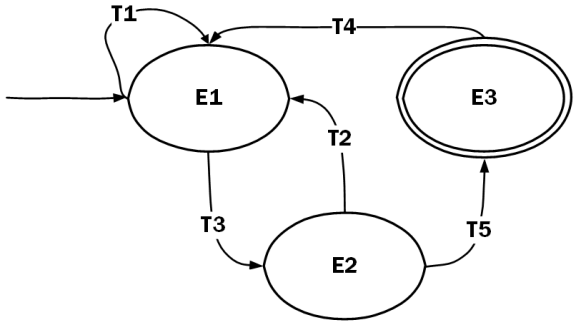


Figura 3.22: Ejemplo de una máquina de estados finita.

En el capítulo 5 de las especificaciones de OPC UA se recoge en un anexo[70] del modelo de información para crear máquinas de estados. Este modelo se utiliza en la especificación 10 de Programas [74] y en la especificación 9 de Alarmas & Condiciones [73]. Según este anexo una máquina de estados finta en OPC UA

está formada por los siguientes elementos:

- **Estados:** Según el modelo de información de OPC UA los estados son instancias del tipo *StateType*. Este tipo de objeto está recogido por el modelo de información para máquinas de estados. Este posee una propiedad *StateNumber* que se utiliza para indicar el número de estado, el cual identifica el estado dentro de la máquina de estados a la que pertenece. Una característica muy interesante que recoge el estándar OPC UA en su modelo de estados es que los estados pueden contener una submáquina de estados. Es decir, permite la jerarquización de estados lo que permite simplificar el conjunto de transiciones en una máquina de estado más compleja.

- **Causas(Métodos):** Las causas dentro de OPC UA son representadas por los métodos que producen que la máquina de estados finita cambie su estado actual.

- **Efectos (Eventos):** Los efectos dentro de OPC UA son representados por eventos. Estos ocurren si la transición es usada para cambiar el estado de una máquina de estados finita.

- **Transiciones:** Al igual que ocurre con los estados las transiciones también tienen un tipo de objeto del cual se instancian; en este caso del tipo *TransitionType*. Este tipo de objeto también posee una propiedad, *TransitionNumber*, que nos sirve para representar el número de transición dentro de la máquina de estados a la que pertenece. Pero a diferencia de los estados, las transiciones tienen contenido semántico, los cuales se representan mediante referencias. Estas referencias identifican los estados involucrados en cada transición. Los diferentes tipos de referencias que nos podemos encontrar son:

  - *FromState* (Estado Origen): Nos sirve para especificar el estado origen que provoca la transición. Según el modelo de información de OPC UA por cada transición solo puede existir una referencia de este tipo.
  - *ToState* (Estado Destino): Indica el estado destino de la transición. Según el modelo de información de OPC UA por cada transición solo puede existir una referencia de este tipo.

- *HasCause*: Indica la causa (método) que genera que se ejecute dicha transición.
- *HasEffect*: Indica el efecto (evento) que se genera cuando se ejecute la transición.

Según la especificación de OPC UA una transición puede albergar una o más causas asociadas. Por el contrario no es obligatorio que una transición tenga asociadas efectos. Esto se debe a que una transición puede desencadenarse debido a alguna condición o lógica dentro del servidor que no tiene por qué estar recogida en el espacio de direcciones de OPC UA.

Si revisamos el anexo recogido en la especificación 5[70], toda máquina de estados es una instancia del tipo de objeto *FiniteStateMachineType*, el cual a su vez es un subtipo de *StateMachineType*. Este último subtipo posee dos variables, *CurrentState* para representar el estado en el que se encuentra la máquina de estados y *LastTransition* que representa ultima transición ejecutada, la organización de todos estos objectos se puede ver en la figura 3.23.

Por lo que una Máquina de Estado Finita dentro de su modelo de información debe tener como mínimo uno o más estados. Además puede tener alguno de estos elementos: una o más transiciones, una o más causas (métodos) y uno o más efectos (eventos). Así la máquina de estados finita más simple que podemos modelar según el modelo de información de OPC UA, corresponde a una máquina con un único estado. El grado de complejidad aumentara conforme aumente el número de estados y transiciones, así como el de submáquinas de estados recogidas dentro de estados principales.

BUENAS PRÁCTICAS DE MODELADO DEL COMPORTAMIENTO EN OPC UA

Como ya vimos en el caso del modelo de información, podemos realizar el modelado de un sistema industrial siguiendo básicamente dos enfoques, utilizando los elementos que ofrece el propio modelo de OPC UA o creando nuestros propios tipos de objetos. En el caso de estar modelando el comportamiento de un sistema industrial con OPC UA y revisando el anexo B de la especificación 5[70] de OPC UA, podemos utilizar cuatro enfoques a la hora de crear máquinas de estados finitas:

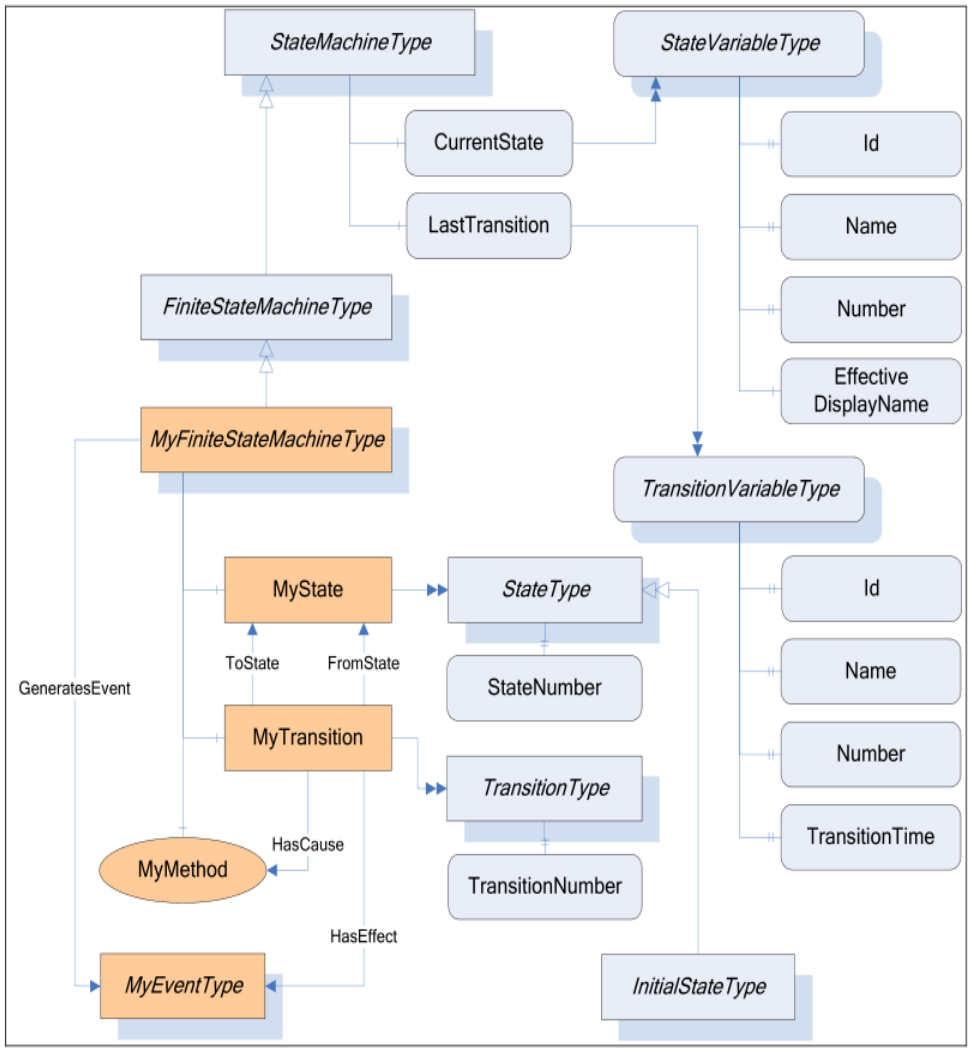


Figura 3.23: Componentes principales del modelo de información de OPC UA para el modelado de máquinas de estados[70].

- **Modelado de máquinas de estados por herencia**. Una máquina de estado se crea definiendo un nuevo tipo de máquina de estado por herencia de *StateMachineType* sobre la que se define los estados, transiciones, eventos y métodos basados en tipos ya definidos de OPC UA. Una vez creado el nuevo tipo de máquina de estado se puede instanciar un nodo con la máquina de estado concreta heredando todos los nodos definidos en el nuevo tipo de máquina de estados.

- **Modelado de máquinas de estados que contiene sub-máquinas de estados por herencia**. En este caso las máquinas de estados están formadas por estados que a su vez pueden contener máquinas de estados. Todos estos estados heredan del tipo *StateMachineType*.

- **Modelado de máquinas de estados como contenedores de nuevos tipos de objetos**. En este caso la máquina de estado se crea como una instancia de objeto (variable) contenida en la definición del tipo de objeto que la contiene. Dicha instancia de máquina de estado tiene que tener una definición que se obtiene por herencia de *StateMachineType* como en el enfoque 1.

- **Modelado de máquinas de estados con transiciones hacia estados de sub-máquinas de estados**. Este tipo de máquinas contemplan transiciones entre estados ”principales” y sub-estados de máquinas que están contenidas en estados ”principales”. Debido a la peculiaridad de este tipo de máquinas, es necesario tener en cuenta ciertos aspectos para que la maquina sea conforme y coherente al modelo de OPC UA. Al igual que en los casos anteriores los estados también descienden del tipo *StateMachineType*.

MODELADO DE MÁQUINAS DE ESTADO POR HERENCIA

El modelado de una máquina de estados se lleva a cabo a partir de la definición de un nuevo tipo de máquina de estado. Como se puede ver en el ejemplo de la figura 3.24, se ha definido un nueva máquina de estado denominada *MyStateMachineType* que hereda del tipo de clase de nodo *FiniteStateMachineType*. Esta nueva máquina de estados está formada por dos estados *state1* y *state2*. Además existe una transición *Transition1* entre *state1* y *state2* causada por el método *MyMethod* y el efecto de la transición es el evento *EventType1*. La variable *CurrentState* representa el estado en el que se encuentra la máquina de estados que

se hereda del tipo *StateMachineType*. Cada uno de las instancias de nodos, *State, Transition, Method* y *EventType* vienen definido por el tipo de clase de nodo correspondiente.

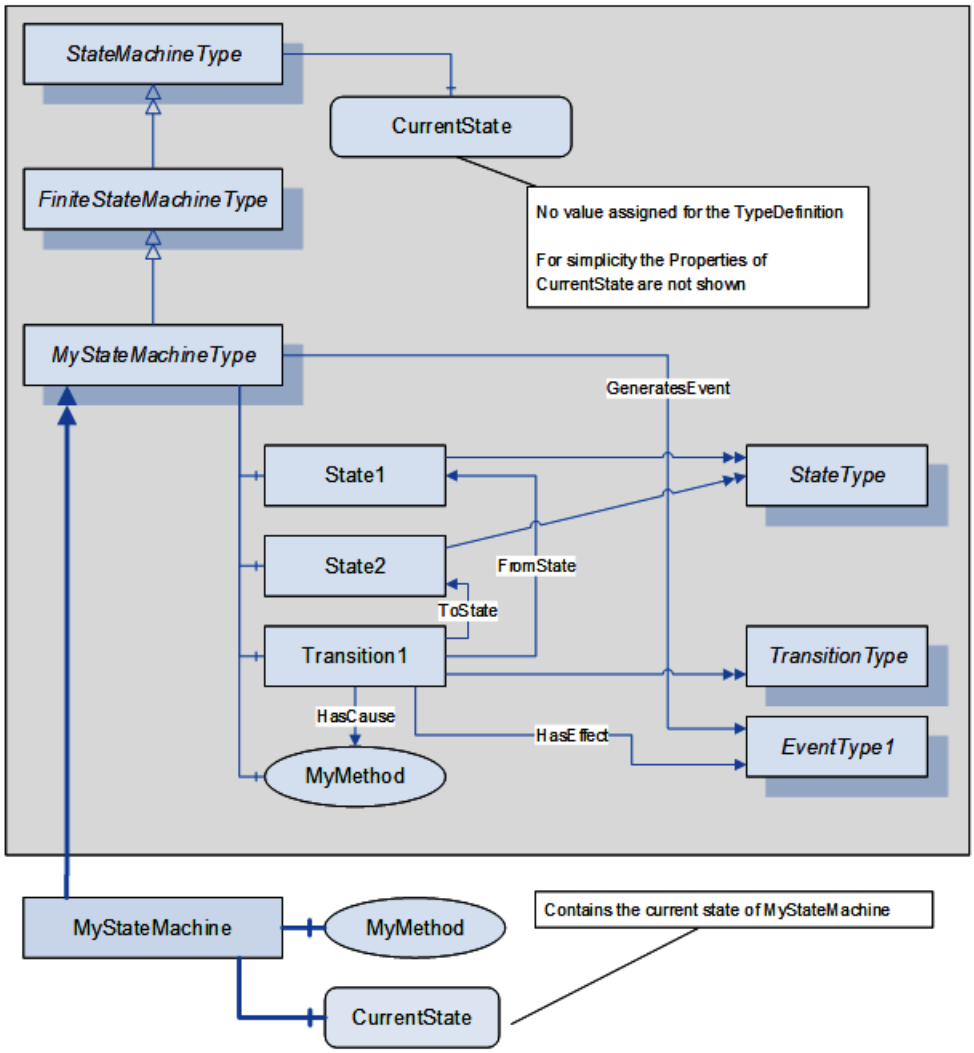


Figura 3.24: Ejemplo de modelado de estados a partir del concepto StateMachineType [70].

MODELADO DE MÁQUINAS DE ESTADOS QUE CONTIENE SUB-MÁQUINAS DE ESTADOS POR HERENCIA

Con este enfoque podemos modelar el comportamiento utilizando máquinas de estados jerárquicas que contienen máquinas de estados anidadas. En el ejemplo de la figura 3.25 se define un nuevo tipo de máquina de estado *MyStateMachi-*

neType que hereda de *StateMachineType*, y que contiene un objeto submáquina de estado *MySubMachine* instanciado a partir de otro tipo de máquina de estado *AnotherStateMachineType*. Como se puede ver en el diagrama esta sub-máquina de estados se ha asociado al estado *State1* de la máquina de estado principal.

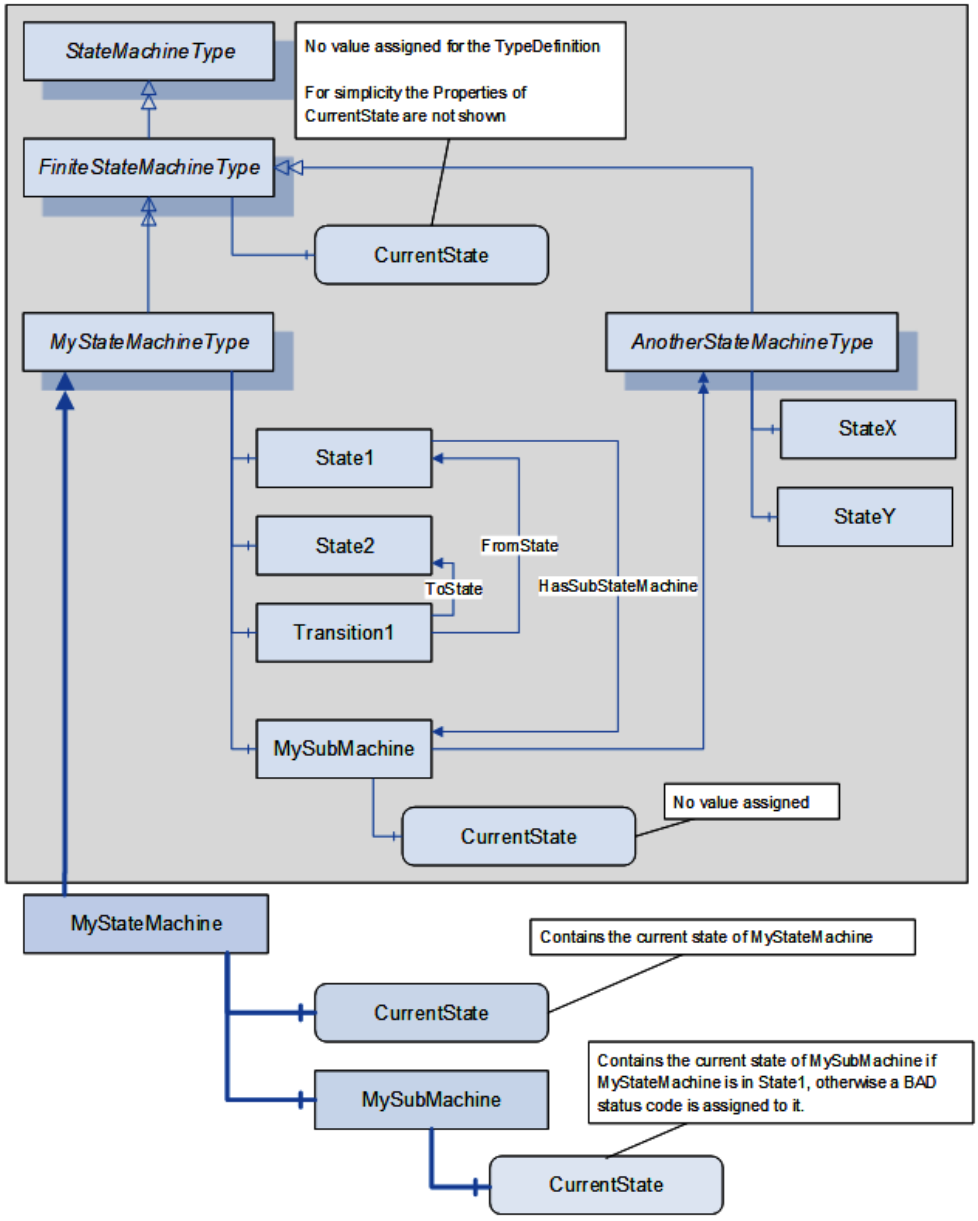


Figura 3.25: Ejemplo de modelado de estados con estados que contienen sub-máquinas [70].

La máquina de estado *MyStateMachineType* y la submáquina de estado *AnotherStateMachineType* tienen definidas como variables sus propios estados y transiciones. En este caso es significativo ver que la definición del tipo *MyStateMachineType* incluye como submáquina de estado un nodo instancia de máquina de estado. La sub-máquina de estados *AnotherStateMachine* solo se activará o podrá ser utilizada cuando *MyStateMachine* se encuentre en el estado *State1*. Esta a su vez está formada por dos estados *StateX* y *StateY*.

MODELADO DE MÁQUINAS DE ESTADOS COMO CONTENEDORES DE NUEVOS TIPOS DE OBJETOS

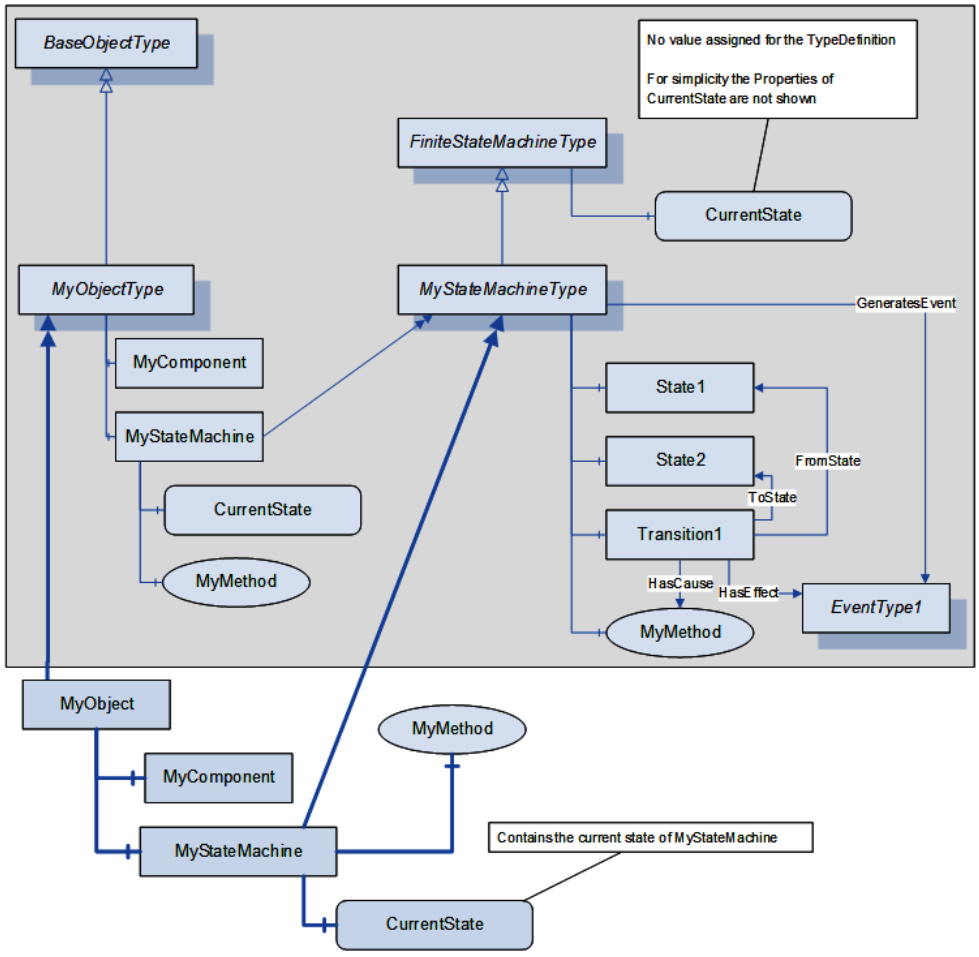


Figura 3.26: Ejemplo de modelado mediante conceptos específicos [70].

En la figura 3.26 podemos ver un ejemplo de modelado en el que se representa un objeto que desciende del tipo *MyObjectType* denominado *MyObject*. Este objeto contiene a su vez otro objeto denominado *MyStateMachine*, el cual es de tipo *MyStateMachineType*, por lo que este objeto puede albergar una máquina de estados. Como se puede ver el tipo *MyStateMachineType* desciende del tipo *FiniteStateMachineType*, y tiene que tener definido el conjunto de estados o transiciones.

MODELADO DE MÁQUINAS DE ESTADOS CON TRANSICIONES HACIA ESTADOS DE SUB-MÁQUINAS DE ESTADOS

Las máquinas de estados mostradas hasta ahora solo tenían transiciones entre estados en el mismo nivel, es decir, en la misma máquina de estados. Sin embargo, es posible y a menudo puede ser interesante tener transiciones entre los estados de la máquina principal y los estados de la sub-máquina contenidas dentro de los estados de la máquina principal.

Debido a que una sub-máquina de estados puede estar definida por otro tipo de *StateMachineType* y este tipo se puede usar en varios lugares, no es posible agregar una referencia bidireccional de uno de los estados compartidos de la submáquina de estados a otro de la máquina de estados contenedora. En este caso, es adecuado especificar que las referencias *FromState* o *ToState* son unidireccionales, es decir, solo apuntan desde la transición al estado y no se pueden navegar en sentido contrario. Si una transición apunta desde un estado de una sub-máquina *SubStateMachine* a un estado de otra submáquina, ambos, el estado de origen y la referencia hacia el estado, se manejan de manera unidireccional. Un Cliente deberá ser capaz de manejar la información de una máquina de estados si las referencias *ToState* y *FromState* se exponen únicamente como referencias de tipo *forward* y se omiten las referencias de tipo *inverse*.

En la figura 3.27 podemos ver un ejemplo de máquina de estados en la que desde un estado de la sub-máquina de estados se hace una transición a uno de los estados de la máquina principal.

En la figura 3.28 se puede ver una posible representación de la máquina de estados de la figura 3.27 dentro del espacio de direcciones de un servidor OPC UA. La transición *Transition1*, parte de la definición de *MyStateMachineType* y apunta al *StateX*, que está dentro de un nuevo tipo de máquina de estados *AnotherState-*

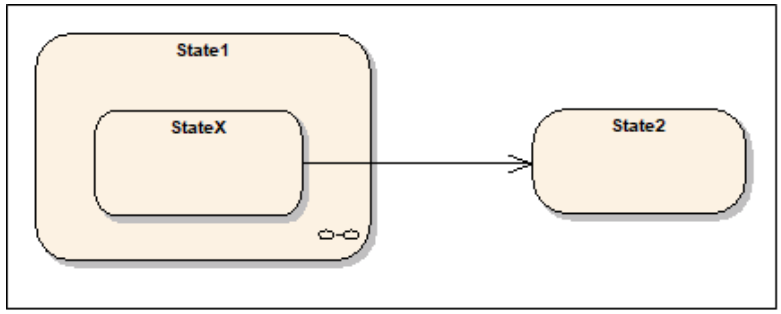


Figura 3.27: Ejemplo de máquina de estados con estados que contienen sub-máquinas [70].

MachineType. La referencia entre ambos estados se ha especificado solo de forma unidireccional, con lo que puede ser utilizada en cualquier parte de *MyStateMachineType* o de*AnotherStateMachineType*.

EJEMPLO DE MODELADO DEL COMPORTAMIENTO CON OPC UA

En el apartado anterior se han mostrado cuatro formas de modelar máquinas de estados utilizando los conceptos que ofrece el modelo de información de OPC UA. A continuación vamos a crear una máquina de estados que refleje el comportamiento del ejemplo del apartado 2.4. En este apartado se modeló un sistema industrial que consistía en una cinta de transporte mediante rodillos con un detector y una barrera de paso. Además en dicho apartado se proponía dos formas de realizar el modelado de este sistema:

- Modelado utilizando los conceptos (tipos) de OPC UA.
- Modelado utilizando la definición de tipos específicos en OPC UA.

En el caso del modelado del comportamiento, en las cuatro propuestas se ha tenido que crear un nuevo tipo de máquina de estados *MyStateMachine*, a partir del tipo ya existente *FiniteStateMachineType*. Podemos concluir, por tanto, que la mejor estrategia para modelar el comportamiento es la de crear una nueva máquina de estados, con los estados que se necesiten utilizando como base concepto *FiniteStateMachineType*.

Pero antes de crear nuestra máquina de estados debemos de tomar dos decisiones. La primera decisión es en relación a qué elementos del sistema industrial necesitan que se modele su comportamiento y en consecuencia necesitan una

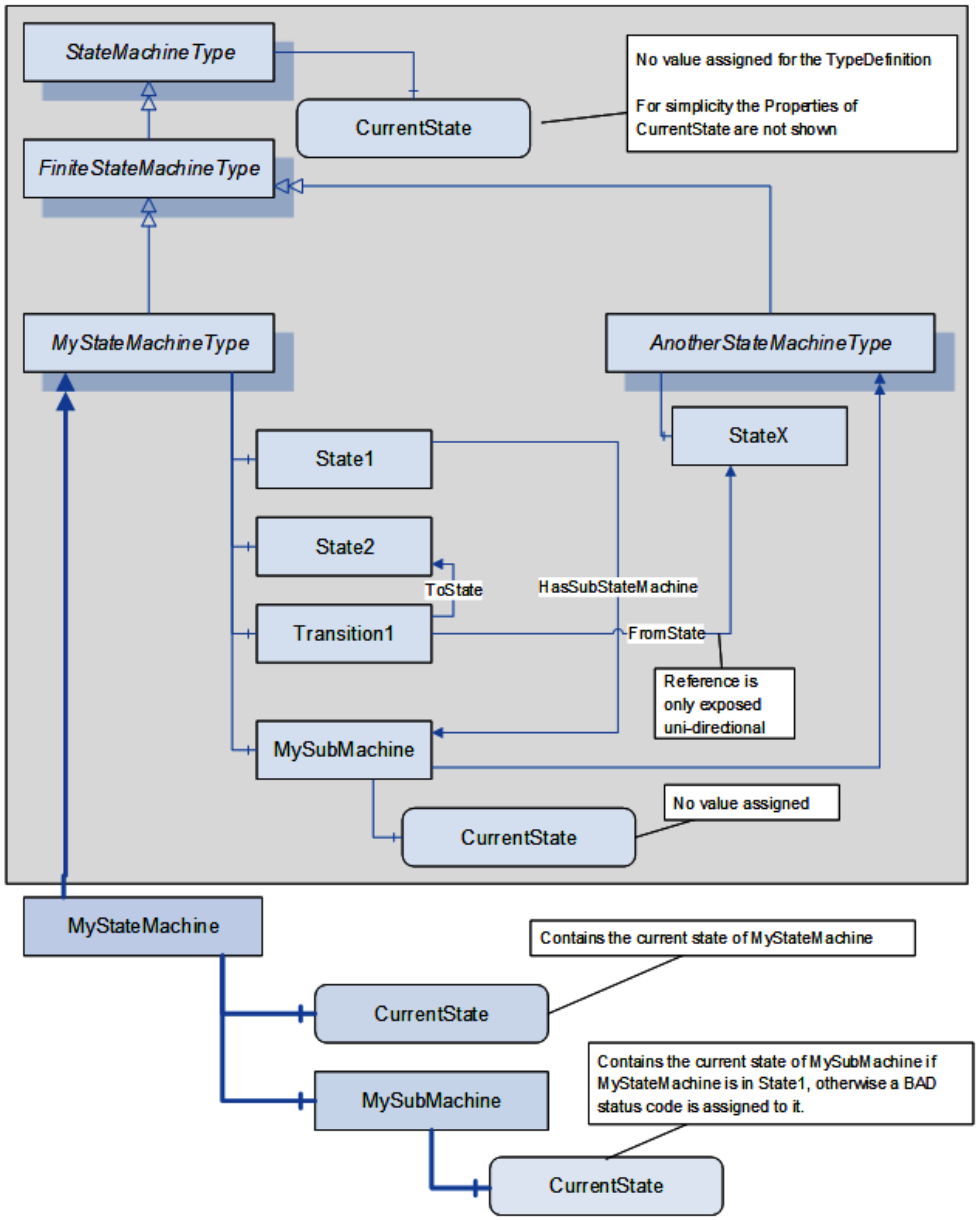


Figura 3.28: Ejemplo de modelado mediante conceptos específicos [70].

máquina de estados. La segunda decisión tiene que ver con determinar si vamos a asociar la máquina de estados a objetos del sistema o si, por el contrario, vamos a modelar el comportamiento de forma independiente al modelo de información.

Bajo nuestro punto de vista los elementos candidatos del sistema industrial que necesitan que su comportamiento sea modelado son aquellos que introducen perturbaciones en el sistema industrial. En el ejemplo anterior tan solo tenemos dos, el motor de la cinta transportadora y la barrera de paso. Así que los actuadores/reguladores son elementos candidatos a poseer una máquina de estados, descartando así otros elementos más pasivos como sensores e instrumentación en general, que tienen como único propósito medir y recoger cierta información del proceso industrial.

En cuanto a cómo construir la máquina de estados, partimos de un modelo de información ya creado en el apartado 3.4.4, en el que se recogen todos los nodos que lo caracterizan y las relaciones que hay entre ellos. Lo mejor es añadir a este modelo las máquinas de estados correspondientes a los objetos ya existentes en el modelo de información.

Esta estrategia puede ser interesante siempre que dispongamos de un modelo que recoja los objetos del sistema industrial y sus relaciones. A este modelo se le puede añadir el modelado del comportamiento de sus elementos, creando así un modelo completo el cual recoge tanto el modelo de información del sistema industrial, parte estática, como el modelo del comportamiento del mismo, parte dinámica.

En el caso de no disponer de un modelo de información previo, puede que para nuestra aplicación industrial solo sea necesario modelar el comportamiento. En este caso tenemos que crear una máquina de estados que no está asociada a ningún elemento del sistema industrial. En cuyo caso solo estamos recogiendo el comportamiento de nuestro sistema industrial mediante un modelo de máquina de estados que utiliza los conceptos que ofrece el estándar de OPC UA. Aunque esta aproximación no es completa, ya que no ofrece un modelo de todos los aspectos del sistema industrial, puede ser suficiente en casos en los que solo se quiere modelar el comportamiento dinámico. Para nuestro ejemplo, como partimos de un modelo previo, se ha creado el modelo de máquina de estados de la figura 3.29.

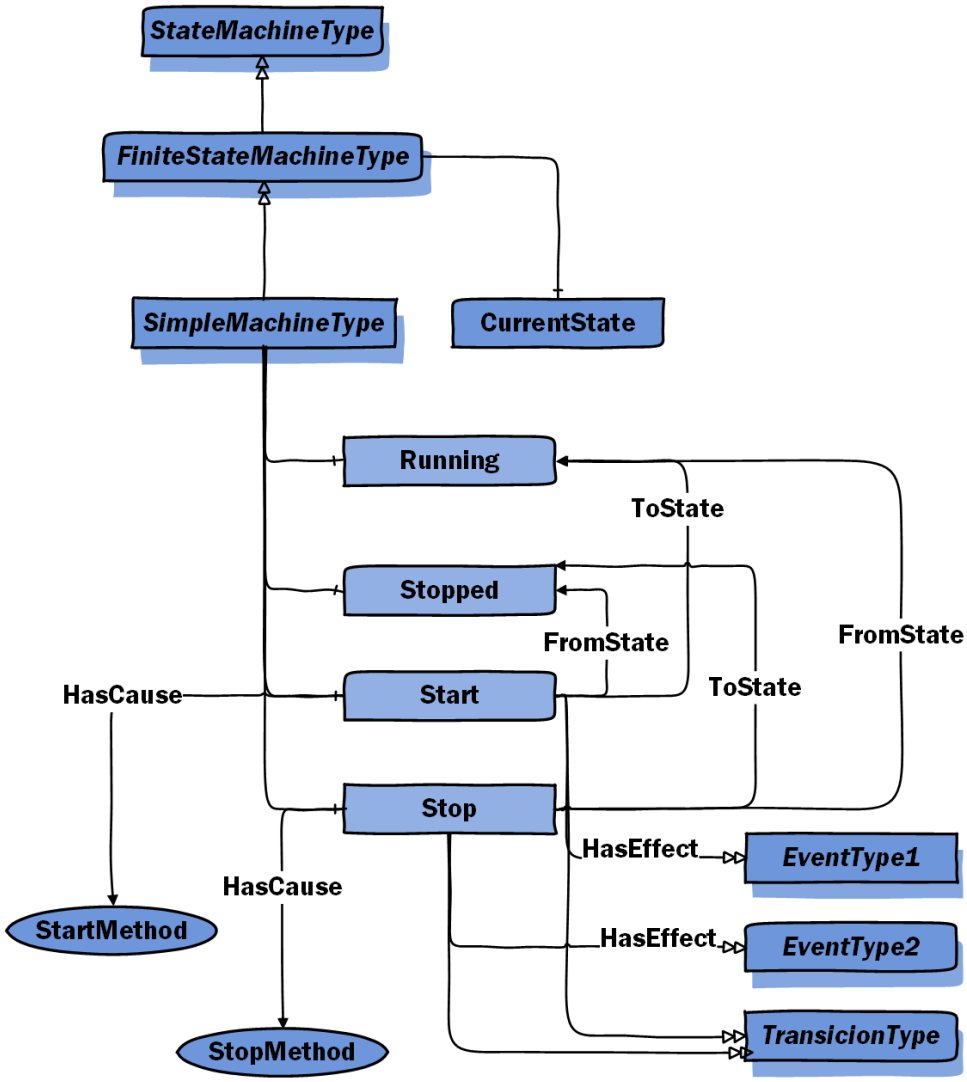


Figura 3.29: Ejemplo de modelado del comportamiento para el ejemplo del apartado 2.5.

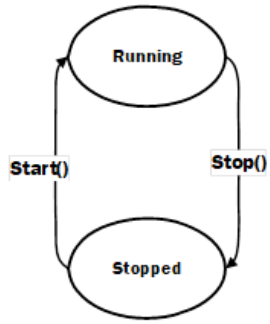


Figura 3.30: Máquina de estados abstracta para modelar el comportamiento de los actuadores del ejemplo 2.4

En dicha figura 3.29 podemos ver un nuevo tipo de máquina de estados *SimpleMachineType*, con dos estados *Running* y *Stopped* y dos transiciones entre estos estados, *Start* y *Stop*, los cuales tiene como causa dos métodos, *StarMethod* y *StopMethod*, y como efecto dos tipos de eventos distintos *EventType1* y*EventType2*. De forma abstracta esta máquina de estados corresponde con la representación de la figura 3.30.

Por último en la figura 3.31 tenemos el modelo completo, modelo de información más modelo del comportamiento. Para ello se ha creado previamente un tipo de objeto que recoge todos los aspectos del sistema industrial a modelar, es decir, comportamiento y elementos. A partir de la definición del nuevo tipo de objeto se ha instanciado el modelo definitivo. Como se puede ver en el metamodelo de información de la figura 3.31, se han añadido dos objetos de tipo *SimpleMachineType*, uno dentro del tipo de objeto *CintaRodillosType* y otro dentro del tipo *BarreraStop*. Adicionalmente se han eliminado los métodos *Start* y*Stop* de estos objetos ya que están recogidos dentro de la definición del tipo de máquina de estado *SimpleMachineType*.

Para crear el modelo definitivo se ha seguido el mismo procedimiento que se explicó en el apartado 2.4.Es decir, se ha creado un objeto *CintaRodillos* que extiende de un nuevo tipo de objeto *CintaRodillosType*, el cual recoge todos los nodos que caracterizan dicho tipo de objeto, incluido el modelado del comportamiento que utiliza el concepto de máquina de estados finita del estándar OPC UA según el anexo B de la especificación sobre el modelo de información [70].

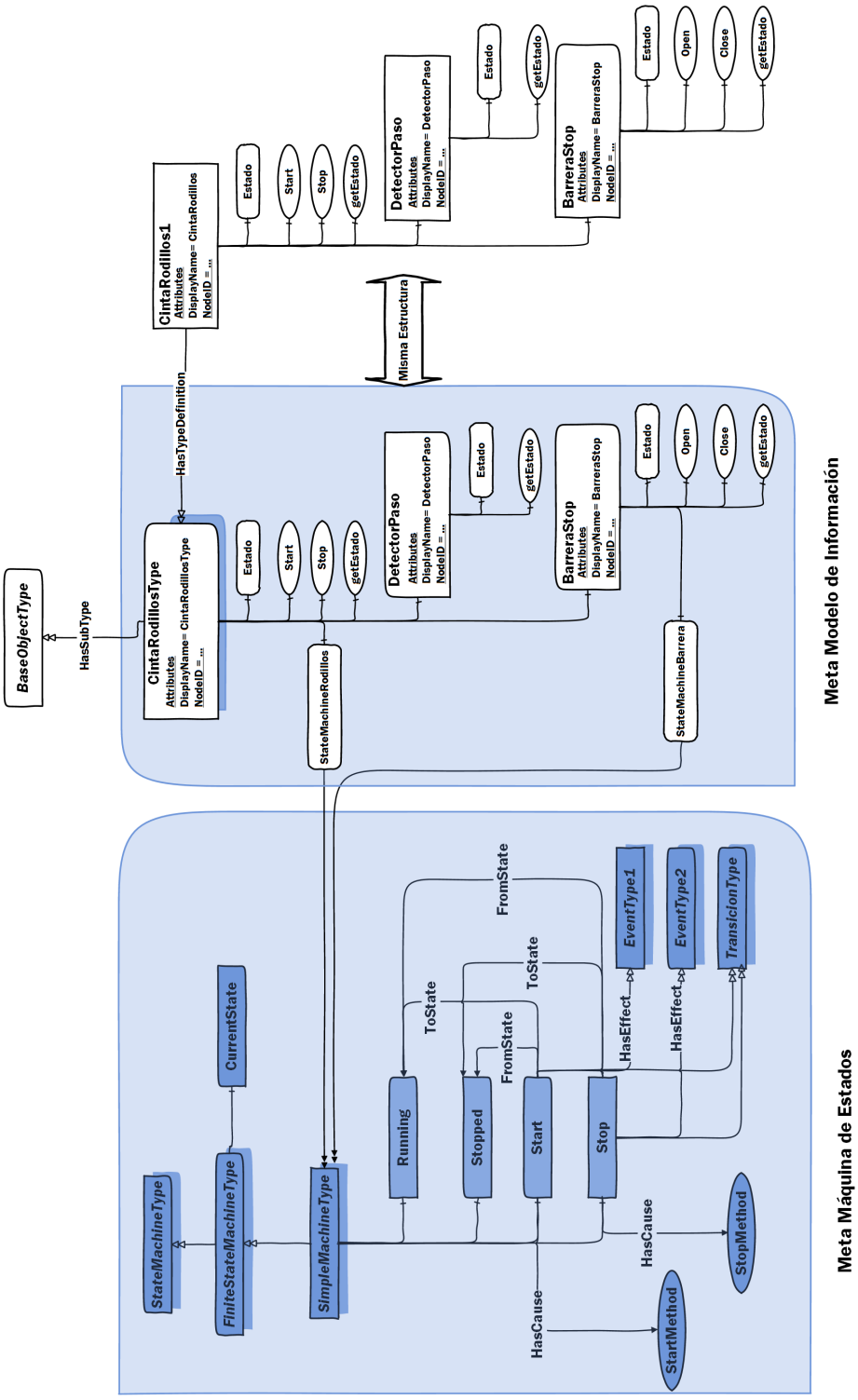


Figura 3.31: Modelado completo del ejemplo 2.5: comportamiento e información

3.5. CONCLUSIONES

El estándar OPC nació para cubrir la necesidades de intercambio de información entre los dispositivos industriales y los sistemas software que monitorizan y supervisan un proceso industrial [89] [90] [91].

Las primeras versiones de este estándar, denominado OPC Clásico [3], han ofrecido una forma homogénea y uniforme a la hora de acceder a la información que manejan los elementos electrónicos que controlan un proceso industrial [48].

Pero desde que este estándar vio la luz con su primera especificación de acceso a datos en los años 90 [46], las necesidades han ido cambiado en cuanto a la organización de la información [26] y el acceso a la misma con el paso del tiempo [17]. En respuesta a todas estas necesidades la fundación OPC liberó una nueva especificación del estándar OPC, denominada OPC Unified Architecture (OPC UA), con la idea de dar soporte a sistemas industriales multiplataforma, con un modelo de comunicación no dependiente de la tecnología subyacente y con un modelo de información que permite crear estructuras de datos complejas.

El nuevo modelo de comunicaciones de OPC UA permite integrar sistemas ya asentados en el mundo de las TIC en el entorno industrial como ocurre, con la computación en la nube (Cloud Computing) [20] [92], el IoT[53] o los sistemas virtualizados y remotos [23]. En consecuencia gracias a la utilización de este estándar podremos introducir Internet como una parte más de los sistemas software industriales [22], adquiriendo características adicionales en cuanto al intercambio seguro de datos en base a mecanismos de seguridad robustos, tanto en el cifrado de información como en el acceso a la misma[93][94].

El segundo aspecto que introduce el estándar OPC UA en los sistemas software industriales es un nuevo modelo de información, con el que podemos modelar y organizar, tanto los datos que maneja el dispositivo industrial como su comportamiento. Utilizando este modelo de información, podemos crear estructuras de objetos complejas jerárquicas o no jerárquicas. Dada la propia naturaleza de los sistemas industriales las estructuras jerárquicas son las que mejor se adaptan de forma natural a la hora de crear modelos para entornos industriales [40]. La flexibilidad del modelo de información de OPC UA nos permite organizar la información y funcionalidad de un proceso industrial, incluso definiendo nuestros propios tipos de objetos adaptados a las necesidades específicas de nuestros

sistema software industriales. Sin embargo, no proporciona ninguna guía de cómo se debe modelar un sistema industrial, ni cuál es la mejor estrategia a seguir durante todo el proceso de modelado. Tampoco explica como se puede aplicar los modelos creados a partir de su modelo de información a sistemas industriales reales, lo que complica su aplicación en dichos sistemas, por la dificultad que puede llegar a suponer para los ingenieros la creación de modelos conformes a las especificaciones del estándar, que puedan luego desplegarse en servidor OPC UA. En este caso es necesario disponer de metodología que nos sirva de guía y de ayuda en el proceso de desarrollo y despliegue. Por otro lado actualmente nos vamos a encontrar con ciertas dificultades a la hora de desplegar modelos en soluciones comerciales, ya que por ahora las soluciones existen solo soporta el modelo de comunicaciones del estándar, y en la mayoría de ellas nos encontraremos con que no contemplan, o lo hace parcialmente, el modelo de información de OPC UA. Pero aun así debido a la riqueza de dicho modelo, este estándar puede ser una pieza clave para la introducción de técnicas de ingeniería del software en los sistemas software industriales. En los siguientes capítulos se podrá estudiar con detalle la propuesta que se realiza en este trabajo de tesis, en la que se utilizará el modelo de información de este estándar como vía para la introducción del desarrollo dirigido por modelos en los entornos industriales.

4

PROGRAMACIÓN PARA DISPOSITIVOS INDUSTRIALES

"Technology, like art, is a soaring exercise of the human imagination."
Daniel Bell 1919-2011

4.1. INTRODUCCIÓN

Un programa está constituido por un conjunto de instrucciones que especifican cómo se ejecuta una determinada tarea o funcionalidad. Para ello, es necesario recoger un conjunto de datos como entrada, tomar una serie de decisiones y generar una respuesta en función de las decisiones que hayan sido tomadas por la lógica del programa. La programación o creación de programa como disciplina se ha circunscrito al mundo de la informática y de las tecnologías de la información, pero se ha extendido a otros dominios, como el de la automoción, la industria y las telecomunicaciones.

Actualmente los procesos productivos de los sistemas industriales presentan un alto grado de automatización, con una enorme cantidad de sensores, actuadores y maquinaria. Todos estos elementos están controlados por dispositivos electrónicos, como PLCs. Un sistema industrial puede estar formado por uno o varios PLCs, a veces incluso de distintos fabricantes. En estos escenarios, en los que conviven PLCs de distintos fabricantes, las tareas de programación e integración pueden ser bastante laboriosas y complejas, debido a que cada fabricante cuenta con una plataforma de programación propia, independiente y con características totalmente diferentes e incompatibles siguiendo sus propias especificaciones y directrices. Esta heterogeneidad en cuanto a dispositivos y plataformas de pro-

gramación puede suponer elevados costes y escasa flexibilidad [95] a la hora de automatizar un proceso productivo. Cuanto más grande es el sistema industrial mayores son estos costes, así como su grado de complejidad dificultando su mantenimiento [96].

Con el propósito de proporcionar un estándar para las diferentes plataformas de programación de un dispositivo industrial se desarrolló la norma IEC-61131 [60]. Actualmente esta norma es el principal estándar en el dominio de la automatización industrial, cuyo objetivo es el de proporcionar una serie de recomendaciones y características para todos los sistemas de programación de dispositivos industriales, facilitando así la integración de los sistemas de control de distintos fabricantes y estandarizando los lenguajes de programación de los dispositivos de control industrial [97].

4.2. EL ESTÁNDAR IEC 61131

La norma IEC 61131 pretende proporcionar un estándar que recoja todas las facetas de los autómatas programables, cubriendo aspectos como la forma del encapsulado y las características hardware, así como los lenguajes de programación de estos dispositivos. La norma IEC 61131 debe verse como una guía a tener en cuenta, tanto para el desarrollo de un entorno de programación para un PLC, como para programarlo. El estándar está formado por un conjunto de normas bastante flexibles que los fabricantes pueden asumir parcial o completamente. Cada norma pueden contemplar un número específico de detalles y aspectos, los cuales son difíciles de recoger completamente por un entorno de programación [98]. Dado que generalmente los fabricantes asumen las normas de forma parcial, deben incluir en su documentación justificación de las partes de la norma que cumplen y cuales no [99]. Para facilitar su adopción la IEC 61131 proporciona una serie de test basados en 62 tablas en las que se listan las principales características con los requisitos necesarios que los fabricantes deben cumplir. Los resultados de estos tests deben anotarse en estas tablas indicando las características que cumple y las que no [60].

El estándar está gestionado por un grupo de trabajo denominado SC65B-WG7 [100] (inicialmente: SC65A-WG6) acogido dentro de la organización internacional de estandarización IEC (International Electrotechinical Commission) [60] forma-

da por diferentes fabricantes de PLCs y software industrial. La norma IEC 61131 y en concreto la especificación IEC 61131-3 [99] se ha convertido en el único estándar reconocido mundialmente para entornos de programación de PLCs, debido principalmente a que el organismo que dirige esta norma está formado por los principales fabricantes de estos dispositivos; por lo que son los propios fabricantes los que aceptan este estándar, siendo una directriz que deben de cumplir sus plataformas de programación.

4.2.1. OBJETIVOS DEL ESTÁNDAR

Los sistemas industriales que utilizan PLCs como dispositivos de control se han ido incrementado con el paso del tiempo. Dichos dispositivos disponen de más interfaces de interconexión, propiciando su utilización en sistemas complejos con cientos de miles de señales. La utilización de PLCs en este tipo de entornos industriales, en los que tradicionalmente se han utilizado sistemas DCS, se ha traducido entre otras cosas, en un aumento de los costes en aspectos como:

- Formación de los programadores.
- Mayores tiempos de desarrollo, despliegue y puesta en marcha.
- Necesidad de equipos de desarrollo formados por varios ingenieros.

En este escenario, utilizar plataformas de desarrollo que sigan las directrices de algún estándar puede traducirse en una reducción de costes, ya que por un lado podemos reutilizar código entre sistemas y, por otro lado, podemos coordinar y cohesionar mejor los equipos de desarrollo. Además las inversiones realizadas en formación pueden ser validas no solo para un tipo de sistema si no para varios, ya que al seguir estos sistemas un estándar común la forma de programarlos puede ser muy similar y en algunos casos incluso la misma. A raíz de estas ventajas los sistemas de programación de PLC están siguiendo gradualmente una tendencia muy parecida a la del resto de los sistemas de desarrollo software convencionales, en las que la presión para reducir los costes mediante la utilización de estándares está, en cierto modo, dirigiendo su evolución y condicionando su utilización.

4.2.2. HISTORIA Y COMPONENTES DE LA NORMA IEC 61131

La norma IEC 61131 representa una combinación y continuación de diferentes estándares, que hacen referencia a otras 10 normas internacionales (IEC 50, IEC 559, IEC 617-12, IEC 617-13, IEC 848, ISO / AFNOR, ISO / CEI 646, ISO 8601, ISO 7185, ISO7498) [60] [61]. Todos estos estándares recogen especificaciones de la nomenclatura o las estructuras gráficas de los lenguajes de programación para PLCs. Además también se recogen aspectos de cómo debe ser el código y qué características deben tener los entornos de desarrollo y programación de autómatas industriales.

Se han desarrollado varias propuestas a lo largo del tiempo para establecer un estándar que recoja y unifique todos los aspectos necesarios a la hora de programar un PLC. La norma IEC 61131 es el primer estándar aceptado internacionalmente. En la tabla 4.1 se pueden ver algunos de los estándares precursores de esta norma.

La última actualización de la norma IEC 61131 pretende ser un estándar internacional que recoge un conjunto de 9 especificaciones (septiembre 2018), que se referencian con el nombre de la norma seguido del número de dicha especificación. Cada una de estas especificaciones recoge uno o varios aspectos:

- IEC 61131-1: Información general.
- IEC 61131-2: Requerimientos de equipos y testeo.
- IEC 61131-3: Lenguajes de programación.
- IEC 61131-4: Pautas para el usuario.
- IEC 61131-5: Comunicaciones.
- IEC 61131-6: Seguridad Funcional.
- IEC 61131-7: Lenguajes de programación para control difuso.
- IEC 61131-8: Directrices para la aplicación e implementación de lenguajes de programación para controladores programables.
- IEC 61131-9: Single-Drop interfaz de comunicaciones digital para pequeños sensores y actuadores (SDCI).

Año	Estándar Alemán	Estándar Internacional
1977	DIN 40 719-6 (Function block diagrams)	IEC 848.
1979		Start of the working group for the first IEC 61131 draft.
1982	VDI guideline 2880, sheet 4 PLC Programming languages	Completion of the first IEC 61131 draft; Splitting into 5 sub-workgroups.
1983	DIN 19239 PLC Programming	Christense Report (Allen Bradley) PLC Programming languages.
1985		First results of the IEC 65 A WG6 TF3
1990		IEC 61131 Parts 1 and 3 are mde standard
1992		International Standard IEC 61131-1,2
1994	DIN EN 661131 Part 3	
1995	DIN EN 661131 Parts 1 and 2	
1996	Additional guide to DIN EN 661131 (user's guide IEC 61131-4)	
1994 - 2001		Corrigendum to IEC 61131-3
1995,1996		Technical Reports type 2 and 3
1999-2001		Amendments

Tabla 4.1: Principales precursores de la norma IEC 61131-3 [99].

De forma resumida cada una de estas especificaciones recoge los siguientes aspectos:

Parte 1: Información general.

La parte 1 contiene definiciones generales y características funcionales típicas que distinguen un PLC de otros dispositivos industriales. Estos incluyen las propiedades que debe de tener un PLC estándar como, por ejemplo, el procesamiento cíclico del programa con una imagen almacenada de los valores de entrada y salida, o la división entre trabajar con el PLC y o con un interfaz hombre-máquina.

Parte 2: Requisitos y pruebas del equipo.

Esta parte define los aspectos eléctricos, mecánicos y funcionales que deben de cumplir un PLC, así como las correspondientes pruebas de calificación, y las con-

diciones ambientales (temperatura, humedad del aire, etc.) y las clases de tensión que deben de soportar los controladores y dispositivos programables.

Parte 3: Lenguajes de programación.
En esta parte se recogen las características y las especificaciones que debe de cumplir los lenguajes de programación para PLCs. Se especifica el modelo de software básico, así como los lenguajes de programación, los cuales deben de incluir las definiciones formales, descripciones léxicas, sintácticas y (parcialmente) semánticas.

Parte 4: Pautas del usuario.
La cuarta parte pretende ser una guía para ayudar al usuario/programador de un PLC en todas las fases del proyecto de automatización. La información está orientada de forma práctica y se tratan aspectos que abarcan desde el análisis de sistemas, la elección del dispositivo y el mantenimiento del mismo.

Parte 5: Comunicaciones.
Esta especificación recoge los aspectos referentes a la comunicación entre PLCs de diferentes fabricantes y de estos con otros dispositivos. En cooperación con ISO 9506 (Manufacturing Message Specification, MMS) [33] se definen clases de conformidad para permitir que los PLC se comuniquen, por ejemplo, a través de redes de interconexión. En esta especificación se cubren las funciones de selección de dispositivos, intercambio de datos, procesamiento de alarmas, control de acceso y administración de red.

Parte 6: Seguridad Funcional.
Esta parte de la especificación IEC 61131 define los requisitos de seguridad para los autómatas programables y los periféricos asociados a dicho autómata programable, que están destinados a utilizarse como subsistemas lógicos de un sistema electrónico o eléctrico/electrónico programable (E / E / PE).

Parte 7: Lenguajes de programación para control difuso.
El objetivo de esta especificación de la norma es proporcionar a los fabricantes y usuarios una forma común de entender e integrar las aplicaciones de control difuso, basadas en IEC 61131-3, y facilitar la portabilidad de los programas difusos entre diferentes fabricantes.

Parte 8: Directrices para la aplicación e implementación de lenguajes de programación para controladores programables: Esta especificación ofrece so-

luciones para aspectos que no están recogidos explícitamente en la norma. Incluye directrices de implementación, instrucciones de uso para usuarios finales, así como ayudas o guías a la hora de programar.

Parte 9: Single-Drop interfaz de comunicaciones digital para pequeños sensores y actuadores (SDCI).
Esta parte conocido como “IO-Link”, (actualmente CDIS) define un interfaz de comunicaciones para dispositivos muy específicos. Actualmente solo se dispone de un borrador.

Para nuestra investigación nos vamos a centrar en la parte 3 de la norma IEC 61131 "lenguajes de programación". Dentro de esta especificación de la norma se va hacer especial hincapié en los tipos de datos, así como en los elementos de programación dinámicos que hacen que el PLC realice tareas (funciones y programas).

4.3. LA NORMA IEC 61131-3

El estándar IEC 61131-3 armoniza la forma de diseñar y programar sistemas de control industrial mediante la estandarización y regularización de los interfaces/entornos de programación para autómatas programables. Un interfaz de programación estándar permite a personas con diferentes perfiles y habilidades crear diferentes elementos de un programa durante diferentes etapas del ciclo de vida del software: especificación, diseño, implementación, pruebas, instalación y mantenimiento. Esta especificación incluye la definición del lenguaje SFC (Sequential Function Chart), que se puede utilizar para estructurar y realizar la organización interna de un programa, y cuatro lenguajes de programación interoperables: Lista de instrucciones (IL), diagrama de contactos (LD), Diagrama de bloques funcionales (FBD) y Texto Estructurado (ST). Gracias a la utilización de estos lenguajes podemos realizar una descomposición en módulos lógicos, facilitando la modularización de los programas. Además también se pueden emplear algunas técnicas de ingeniera del software que nos ayuden a crear programas estructurados, aumentando así su reutilización, reduciendo errores y tiempos de desarrollo.

En los siguientes apartados vamos a realizar un pequeño recorrido por los aspectos más importantes de esta especificación del estándar IEC 61131, analizan-

do los elementos comunes que se pueden utilizar indistintamente del lenguaje de programación que se haya escogido.

4.3.1. TIPOS DE DATOS

Los lenguajes de programación de PLCs tradicionales contienen tipos de datos tales como reales, binarios, enteros, temporizadores y valores para contadores, que a menudo tienen formatos y codificaciones completamente incompatibles. Por ejemplo, la representación de coma flotante, para la cual se usan las palabras clave REAL o FLOAT, se implementa típicamente mediante palabras de datos de 32 bits y se emplean adicionalmente rangos de valores diferentes para la parte de la mantisa y del exponente. Por otro lado, la mayoría de los sistemas de programación tradicionales tienen un uso uniforme de BIT, BYTE, WORD y DWORD. Sin embargo, incluso para valores enteros simples, existen diferencias sutiles pero significativas entre las representaciones con signo o sin signo, en lo que al número de bits se refiere entre fabricantes de PLCs.

Por lo tanto, en la mayoría de los casos tener programas portables, desde el punto de vista de compatibilidad en el tipo de datos puede que no sea posible, ya que los tipos de datos pueden ser incompatibles entre PLCs de distintos fabricantes, y pueden llegar a requerir grandes adaptaciones en el código fuente de los programas, que además suelen ser propensas a errores.

Como consecuencia de estas discrepancias, la especificación IEC 61131-3 define los tipos de datos más utilizados a la hora de programar un PLC, de manera que su significado y uso dentro del mundo PLC sean uniformes. Esto es de particular interés para los fabricantes de maquinaria industrial, así como para los equipos de ingeniería que trabajan con varios PLC y sistemas de programación de diferentes fabricantes. Los tipos de datos uniformes son el primer paso hacia los programas de PLC portables.

TIPOS DE DATOS ELEMENTALES

En la IEC 61131-3 existe un conjunto de tipos de datos predefinidos y estandarizados, denominados tipos de datos elementales, que se resumen en la siguiente tabla 4.2.

Estos tipos elementales se caracterizan por el tamaño de su representación interna (número de bits) así como por el posible rango de valores que estos pueden

Boolean/strin	Signed Integer	Unsigned Integer	Floating point (Real)	Time, duration, date and character string
BOOL	INT	UINT	REAL	TIME
BYTE	SINT	USINT	LREAL	DATE
WORD	DINT	UDINT		TIME_OF_DAY
DWORD	LINT	ULINT		DATE_AND_TIME
LWORD				STRING

Tabla 4.2: Principales precursores de la norma IEC 61131-3 [99].

tener. Ambos valores están definidos por la Comisión Electrónica Internacional IEC [60]. Existen algunas excepciones a esta norma como, por ejemplo, el tamaño de la representación interna y el rango de tipos para datos de tipo fecha, hora y cadena que dependen de la implementación de la plataforma de programación. En la norma, no se definen tipos de datos BCD ni tipos de datos contador. El código BCD hoy en día ha perdido importancia con respecto a los primeros sistemas de programación de autómatas. En cuanto a los valores para tipos de datos contador suelen implementarse mediante enteros normales no requiriendo un formato específico, al menos para los bloques de función de contador estándar de IEC 61131-3.

TIPOS DE DATOS DERIVADOS (DEFINICIÓN DE TIPOS DE DATOS).

A partir de los tipos de datos elementos, los programadores de PLCs pueden definir sus propios tipos de datos, a estos nuevos tipos de datos se les suele denominar "tipos definidos por el usuario". Esta técnica permite a los programadores crear un conjunto de tipos de datos, los cuales se pueden adaptar mejor a las características de la aplicación que se esté desarrollando. Tales definiciones de tipos tienen carácter global dentro de un proyecto de programación de un PLC, lo que hace que se puedan utilizar por cualquier módulo, función o programa.

Los tipos de datos definidos, a los que se le debe de proporcionar un nombre, también reciben el nombre de tipos de datos derivados y se pueden utilizar para especificar el tipo de una variable en su declaración, de la misma forma que se utilizan los tipos de datos elementales.

Para realizar la definición de estos nuevos tipos la norma IEC 61131-3 solo

contempla que se haga de forma textual, (no sé contempla la representación gráfica). Las definiciones de tipo están enmarcadas por las palabras clave *TYPE* ... *END_TYPE*, como se muestra en el ejemplo:

```
TYPE
  (* direct derivation from IEC data type *)
  NewReal : LREAL;
  (* direct derivation from a user data type *)
  RealPoint : NewReal;
  (* derivation with new initial value *)
  InitReal : LREAL := 1.0;
  (* derivation with new initial value *)
  tActive : BOOL := TRUE;
END_TYPE
```

Código 4.1: Ejemplo de definición de tipos en ICE 61113-3 [101]

En el ejemplo anterior, el nuevo tipo de datos *NewReal* se define como una alternativa para el tipo de dato estándar *LREAL*. Después de esta declaración, *NewReal* y *LREAL* se pueden utilizar de forma equivalente para otras declaraciones de variables. Como muestra el ejemplo, un tipo de dato derivado puede a su vez servir como base para una derivación adicional. Por lo tanto *RealPoint* es también equivalente a *LREAL*. El tipo de dato derivado *InitReal* tiene un valor inicial diferente a *LREAL* 1,0 en lugar de 0,0. Además, *tActive* tiene el valor inicial *TRUE* en vez del valor inicial estándar *FALSE*.

Las definiciones de tipo son necesarias para crear nuevos tipos de datos con propiedades extendidas o diferentes que puedan ser reutilizadas. La utilización de tipos de datos facilita el desarrollo de programas, ya que estos nuevos tipos definidos recogen mejor las necesidades de la aplicación. Los usuarios y los fabricantes de PLC pueden crear o predefinir tipos de datos individuales con el objetivo de cubrir ciertas necesidades como:

- Que se pueden especificar los valores de inicialización (este aspecto no está recogido por la especificación).

- Definición de tipos de datos para rangos y enumerados.

- Arrays multidimensionales.
- Estructuras de datos complejas. Con las que agrupar variables de distintos tipos y crear a sí un modelo de datos.

Todas estas posibles aplicaciones de las definiciones de nuevos tipos de datos no son excluyentes, y se pueden combinar, ya que están soportadas por la especificación IEC 61131-3. Los conceptos de ”matriz” y ”estructura” también pueden ser utilizados para crear nuevos tipos de datos derivados, es decir, se pueden anidar, creando así matrices de tipos de datos definidos por el usuario, las cuales pueden ser multidimensionales o undimensionales. La propiedad *”range”*, rango de los valores que puede tomar una variable, se define en IEC 61131-3 sólo para el tipo de dato elemental *Integer* y sus derivaciones directas. Realizar una nueva definición de otro tipo de dato que no sea *Integer* para *range"* no está soportado por la especificación. Los tipos de datos enumerados no son tipos derivados en el sentido estricto, ya que estos no derivan de ningún tipo de dato elemental. Sin embargo, en IEC 61131-3 se definen como derivados, ya que los entornos de programación normalmente utilizan números enteros para implementar los tipos de datos enumerados.

```
TYPE
  Colour : (red, yellow, green); (* enumeration *)
  Sensor : INT (-56..128); (* range *)
  Measure : ARRAY [1..45] OF Sensor; (* array *)
  TestBench : (* structure *)
 STRUCT
   Place : UINT; (* elementary data type *)
   Light : Colour:= red; (* enumerated data type with initial value *)
   Meas1 : Measure; (* array type *)
   Meas2 : Measure; (* array type *)
   Meas3 : Measure; (* array type *)
  END_STRUCT;
END_TYPE
```

Código 4.2: Ejemplo de definición de tipos utilizando STRUCT en ICE 61113-3 [101]

En el ejemplo anterior, se muestra el uso de estas propiedades adicionales. La variable *Colour* recoge el rango de valores para un semáforo de tres colores, *Sensor* contiene el rango de temperatura admisible y el array *Measure* se puede utilizar para almacenar 45 mediciones de sensores con rangos de medida determinados por el tipo *Sensor*. *TestBench* es una estructura de datos compuesta de tipos de datos elementales y derivados. Los tres elementos entre paréntesis para el tipo de dato enumerado *Colour*, son introducidos por el programador como nombres, sin necesidad de información adicional, por lo tanto, pueden considerarse como cadenas de textos constantes. La plataforma de programación convierte automáticamente los tres valores rojo, amarillo y verde en un código adecuado.

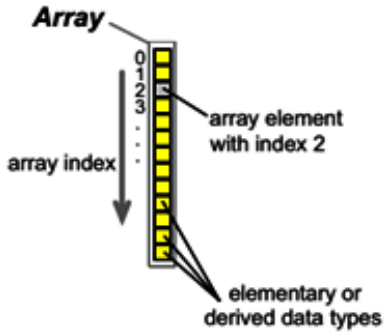


Figura 4.1: Ejemplo de una matriz unidimensional [99].

Estos valores están habitualmente correlacionados internamente de forma transparente para él programador a valores enteros, por ejemplo, comenzando en "1". En la declaración de intervalos para un tipo de dato, como en el caso de la variable *Sensor* en el ejemplo anterior, se informa de un error si se excede este intervalo durante la programación o en tiempo de ejecución. Los intervalos también se pueden usar en las sentencias CASE o IF en el lenguaje de texto estructurado o ST. Por otro lado, en el programa podemos utilizar directamente los nombres de los valores de *Colour* como constantes. El uso de tipos de datos enumerados simplifica las comprobaciones de valores y hace que los programas sean más legibles.

Matrices

Las matrices son elementos que agrupan datos consecutivos del mismo tipo en alguna zona de memoria. Se puede acceder a un elemento de una matriz con la ayuda de un índice, dentro de los límites especificados para dicha matriz. Esto se ilustra en la figura 4.1. Un sistema de programación y de desarrollo para PLCs debe garantizar que se emitirá un mensaje de error en tiempo de ejecución si se intenta acceder a una matriz con un subíndice fuera de los límites permitidos.

La matriz mostrada en la figura 4.1 es unidimensional, es decir, tiene una sola dimensión y por tanto solo necesita un índice para acceder a los elementos de la misma. Este tipo de matrices recibe el nombre de arrays. También se pueden crear matrices multidimensionales declarando conjuntos adicionales de límites separados por comas, como se muestra en el ejemplo siguiente (*Meas_2Dim*). En este caso, los elementos se almacenan en la memoria dimensión por dimensión. Las dimensiones se especifican en orden de importancia.

```
TYPE
Meas\_1Dim : ARRAY [1..45] OF Sensor; (* 1-dimensional array *)
Meas\_2Dim : ARRAY [1..10,1..45] OF Sensor; (*2-dimensional array*)
END\_TYPE
```

Código 4.3: Definiciones de tipo de una matriz unidimensional y bidimensional para la adquisición de una (Meas_1Dim) o de diez mediciones (Meas_2Dim) registradas [101].

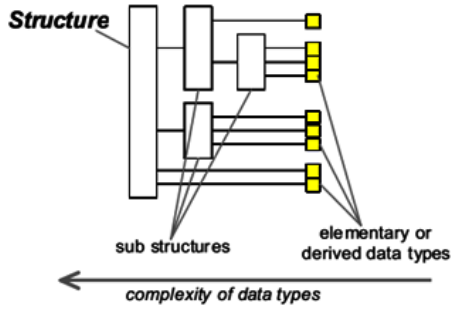


Figura 4.2: Representación interna de una estructura de datos [99].

Además de la definición de matrices como un tipo de datos, las matrices también se pueden definir directamente en la declaración de variables, como cual-

quier tipo de dato elemental o derivado definido por el usuario. La única restricción que contempla la norma es referente a la definición de matrices de bloques de función (FBs), la cual no están permitida en IEC 61131-3. Sin embargo, sería una extensión razonable de la norma a fin de permitir, por ejemplo, un acceso más fácil a temporizadores y contadores muy similares.

Estructuras de Datos

Con la ayuda de las palabras clave *STRUCT* y *END_STRUCT*, se puede construir estructuras de datos, de la misma forma que en lenguajes de programación de alto nivel, como en el caso de la construcción struct de C. Además estas estructuras de datos se pueden construir de forma jerárquica, pudiendo contener cualquier tipo de datos elementales o derivados. Los nombres de instancia de FBs tampoco están permitidos como elementos dentro de una estructura de datos. Si un sub-elemento es también una estructura, se crea una jerarquía de estructura (como se ilustra en la figura 4.2). De esta manera los programadores de PLCs pueden crear y adaptar de forma óptima estructuras de datos, agrupando varios tipos de datos para satisfacer mejor las necesidades de la aplicación o programa que se esté desarrollando. En el siguiente ejemplo se define la estructura *MotorState* creada para recoger las características de funcionamiento real de un motor. Para su definición se utilizan tipos de datos elementales y además incluye una variable *Revolutions* con una especificación de rango, así como un tipo enumerado *Level*.

```
TYPE
  LimitedRevol : UINT (0..230);
  TypLevel : (Idling, SpeedUp1, SpeedUp2, MaxPower);
  MotorState :
   STRUCT
    Revolutions : LimitedRevol; (* range *)
    Level : TypLevel; (* enumerated data type *)
    MaxReached : BOOL; (* elementary data type *)
    Failure : BOOL; (* elementary data type *)
    Brake : BYTE; (* elementary data type *)
  END_STRUCT;
END_TYPE
```

Código 4.4: Definición de tipo de una estructura compleja como un tipo de datos derivados [101].

```
TYPE
  MultiMotState : ARRAY [1..4] OF MotorState; (* further derivation *)
END_TYPE
VAR
(* case 1: *)
  Motor1 : MotorState; (* declaration *)
(* case 2: *)
  Motors : ARRAY [1..4] OF MotorState; (* array declaration *)
(* case 3: *)
  FourMotors : MultiMotState; (* declaration *)
END\_TYPE
```

Código 4.5: Uso de un tipo de datos derivado (arriba) y declaración de matriz (parte inferior) [101].

El nuevo tipo de datos *MotorState* puede ser utilizado para declarar variables de dicho tipo. Por ejemplo, para representar varios motores del mismo tipo se puede crear una matriz de motores (como una variable o también como un tipo de datos) de elementos de tipo *MotorState*.

Como se puede ver en el ejemplo anterior, la matriz con los cuatro elementos del tipo *MotorState* se puede crear mediante una declaración de una variable, en este caso *Motors* (caso 2). El caso 3 representa una alternativa al caso anterior ya que aquí se utiliza el tipo de dato *MultiMotState*. Este tipo de dato reserva para cada variable (como *FourMotors*) una matriz de cuatro elementos basados en el tipo de datos derivado *MotorState*.

En principio, la derivación de tipos de datos puede anidarse, es decir, se pueden derivar tipos de datos a partir de otros tipos de datos derivados. Sin embargo, la anidación es ilegal si produce recursión. Esto puede ocurrir, por ejemplo, cuando dentro de una estructura se define otra estructura que ya contiene esta estructura. En el siguiente ejemplo se muestra un caso de anidamiento ilegal.

```
TYPE
StructureA :
STRUCT
        Element1 : INT;
        Element2 : StructureB;(* legal: sub structure *)
        Element3 : BYTE;
END\_STRUCT;
StructureB :
STRUCT
        Element1 : LINT;
        Element2 : StructureA;(* illegal: recursive to StructureA *)
        Element3 : WORD;
END\_STRUCT;
END\_TYPE
```

Código 4.6: Anidamiento ilegal de tipos de datos derivados: StructureA tiene una subestructura que se contiene (definición de tipo recursivo) [101].

Tipos de datos genéricos

La norma IEC 61131-3 define un conjunto de tipos de dato especiales denominados tipos de datos genéricos para combinar jerárquicamente los tipos de datos elementales en grupos individuales. Estos tipos de datos comienzan con el prefijo *ANY*, por ejemplo todos los tipos de datos enteros (*INT*) se designan como *ANY_INT* de forma genérica. Muchas funciones estandarizadas que se recogen en la especificación IEC 61131-3 se pueden aplicar a más de un tipo de datos, por ejemplo la función *ADD* se puede utilizar con todos los tipos de enteros *INT* y además con el tipo de datos genérico *ANY_INT*. Los tipos de datos genéricos se utilizan para describir funciones estándares, con el fin de especificar cuáles de sus variables de entrada o salida permiten varios tipos de datos. Esto se conoce como sobrecarga de funciones.

El tipo de dato *ANY* forma la generalización más amplia de un tipo de datos dado. Así por ejemplo, la función estándar de multiplicación *MUL* admite el tipo de datos genérico *ANY_NUM* y el entorno de programación permite todos los tipos de datos para *MUL* que están incluidos en los tipos de datos *ANY_INT* y *ANY_REAL*, es decir, todos los enteros con y sin signos así como números de punto flotante.

Los tipos de datos definidos por el usuario (derivados) también contemplan el uso de tipos de datos *ANY*. Por otro lado los tipos *ANY* no se pueden utilizar en las POUs (Unidades Organizativas de Programa) definidas por el usuario a la hora de realizar la declaración de variables, ya que puede no estar permitido o al menos no está soportado por la norma.

4.3.2. VARIABLES

Las variables se declaran en la parte destinada para ello en una unidad organizativa. Las declaraciones de variables son independientes del lenguaje de programación escogido y, por tanto, son uniformes para todo el proyecto de programación de un PLC. Una declaración de una variable consiste esencialmente en un identificador (nombre de variable) y el tipo de dato de la variable. Para realizar las definiciones de tipos se pueden utilizar tanto los tipos simples como tipos derivados.

Representación interna de variables

En la programación convencional de un PLC (véase la norma DIN 19239 [102]) es normal acceder directamente a las direcciones de la memoria del PLC utilizando *operandos* como *M 3.1* o *IW 4*. Estas direcciones pueden estar en la memoria principal de la unidad central de procesamiento del PLC (CPU) o, por ejemplo, en los módulos de E/S (entradas y salidas). A estas direcciones se suele acceder utilizando los tipos de datos bit, byte, palabra o doble palabra. Las áreas de memoria a las cuales se puede acceder mediante direcciones físicas pueden albergar distintos tipos de datos dentro del programa de PLC como valores enteros -*BCD BYTE* y *WORD*-, números de coma flotante -*REAL* y *LREAL*- o contadores *INT*. Esto significa que las celdas de memoria tiene un formato de datos específico en función del dato que almacene (8, 16, 32 bits). Estos formatos de datos son en general incompatibles entre sí y los programadores tienen que recordar los distintos formatos de las direcciones correspondientes a las distintas variables utilizadas en el programa de PLC. Pueden producirse errores en los programas cuando se especifica una dirección de memoria incorrecta, o se utiliza una dirección con un formato de dato incorrecto.

En muchos sistemas de programación de PLCs se han introducido la utiliza-

ción de "símbolos", que se pueden utilizar de forma equivalente en lugar de las direcciones PLC absolutas para conseguir programas de PLCs más legibles. A cada dirección se le asigna un nombre simbólico único mediante una lista de asignaciones o una tabla de símbolos. Además de la utilización de simbólicos también se pueden utilizar lo que se denomina "variables sustitutivas", que van un paso más allá, ya que en lugar de las direcciones de hardware o símbolos, se define el uso de variables asociadas a estas direcciones de memoria o simbólicos. En el siguiente ejemplo se puede ver en la parte izquierda un ejemplo de utilización de simbólicos y en la parte derecha de "variables sustitutivas".

```
VAR
I3.4   = InpVar          InpVar AT %IX3.4 :BOOL;
M70.7  = FlagVar         FlagVar          :BOOL;
Q1.0   = OutVar          OutVar AT %QX1.0 :BOOL;
AT %MX70.6
END\_VAR

A InpVar                          LD InpVar
A FlagVar                         AND FlagVar
ON M70.6                          ORN %MX70.6
=OutVar                           ST OutVar
...                                  ...
```

Código 4.7: Ejemplo de utilización e introducción al concepto de variable según la norma IEC 61131-3. A la izquierda la utilización de variables de la norma DIN 19239, a la derecha la representación en IEC-61131-3 [101].

En el ejemplo anterior (derecha) se utiliza la palabra reservada *VAR* para la declaración de variables locales. El uso de la palabra clave *AT* así como las direcciones que comienzan con *%I* indican que se trata de una dirección de una entrada y *%Q* de una dirección de salida. Los símbolos o variables *InpVar* y *OutVar* en el ejemplo anterior se traducen directamente como direcciones de hardware *I 3.4* y *Q 1.0* del PLC. La dirección del PLC *M 70.6* se utiliza directamente sin símbolo ni nombre de variable. Este tipo de direcciones recibe el nombre de marcas de memoria. Todas las direcciones y variables de este ejemplo son booleanas, con

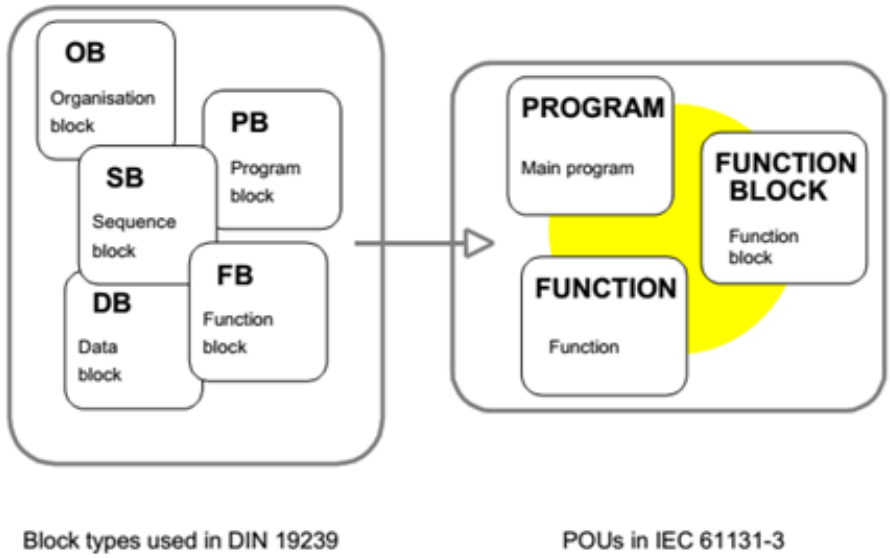


Figura 4.3: Evolución de los tipos de bloques, de la norma germana DIN 19239 a las POU de IEC 61131-3 [99].

valores binarios 0 y 1, por lo tanto, el tipo de dato "*BOOL*", abreviatura de *Boolean*, se especifica en la derecha en cada caso. La variable *FlagVar* se declara sin asignación directa a una dirección de PLC. El sistema de programación lo hace automáticamente al compilar el programa, encontrando y asignando una dirección de memoria libre, como por ejemplo *M 70.7*.

4.3.3. UNIDADES ORGANIZATIVAS DE PROGRAMA (POU)

Según la especificación IEC 61131-3, se denomina POU (Unidad Organizativa de Programa) a los bloques con los que se construyen los programas o los proyectos de un PLC. Las POU engloban a los bloques de programa, bloques de organización, bloques de secuencia y los bloques de funciones del mundo de programación convencional de PLCs. Uno de los objetivos de la especificación es unificar la gran variedad de tipos de bloques que albergan código, unificando y simplificando su uso.

Como se muestra en la figura 4.3, la especificación IEC 61131-3 reduce los diferentes tipos de bloques recogidos en la norma DIN 19239 [102], bloques de organización OB, bloques de secuencia SB, bloques de programa PB, bloques de

datos DB y bloques de función FB, a tres tipos básicos, programas, funciones y bloques de función. Los bloques de datos se sustituyen por bloques de función FB, los cuales tienen asociado una memoria donde se alojan los datos. Esta memoria recibe el nombre de *instancias* del FB. También podemos utilizar variables globales de elementos múltiples para sustituir los bloques de datos. Las principales características de estas unidades organizativas de programa son:

- Función (FUN). Es una POU a la que se le pueden pasar parámetros, pero no tiene variables estáticas (sin memoria). Cuando se invoca este tipo de POU con los mismos parámetros de entrada siempre se obtiene el mismo resultado.

- Bloque de función (FB). Es una POU a la que se le puede pasar parámetros de entrada y salida, y cuenta también con variables estáticas (con memoria). Un FB (por ejemplo un contador o un bloque temporizador) cuando se invoca con los mismos parámetros de entrada, dará valores que también dependen del estado de sus variables internas (*VAR*) y externas (*VAR_ EXTERNAL*), los cuales se conservan desde la ejecución anterior del Bloque de función a la siguiente.

- Programa (PROG). Este tipo de POU representa el "programa principal". Todas las variables del programa que están asignadas a direcciones físicas (por ejemplo a entradas y salidas de PLC) deben ser declaradas en este POU o en su defecto como un recurso configurable o como variables globales. En todos los demás aspectos se comporta como un FB.

PROG y FB pueden tener tanto parámetros de entrada como de salida. Las funciones FUN puede tener un único valor de salida o varios. Estas propiedades se limitaban anteriormente a "bloques funcionales". En IEC 61131-3 la palabra reservada *FUNCTION_BLOCK* con parámetros de entrada y salida corresponde con un bloque de función convencional. Los tipos de *POU PROGRAM* y *FUNCTION* no tienen correspondencias directas a bloques de normas anteriores, como por ejemplo a la norma DIN 19239 [102].

Un POU es una unidad encapsulada, que puede ser compilada independientemente. Sin embargo, el compilador necesita información sobre las interfaces de

llamada de las otras POUs que utiliza la POU que sé compila ("prototipos"). Las POUs compiladas se pueden vincular más tarde para crear un programa completo. Así, después de programar una POU (declaración), su nombre y su interfaz de llamada serán conocidos por las otras POUs del proyecto, es decir, el nombre de POU siempre será global. Esta independencia de las POUs facilita la modularización, así como la reutilización de las unidades de software ya implementadas y probadas, lo que en otros lenguajes se conoce como librerías.

Elementos de una Unidades Organizativas de programa

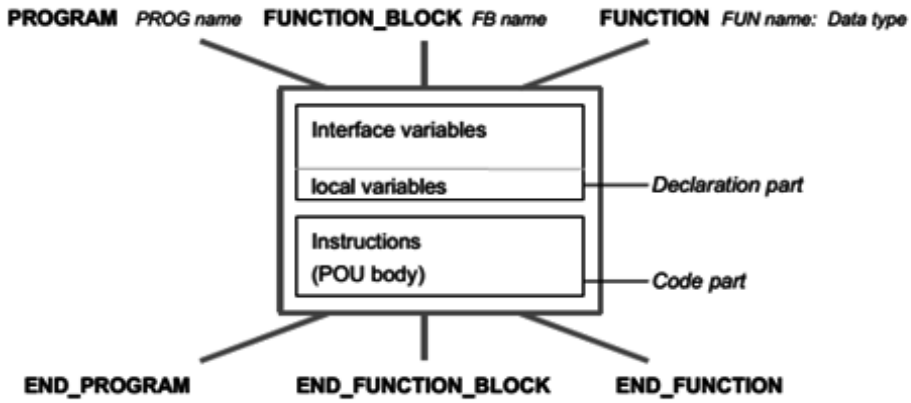


Figura 4.4: Estructura de un POU [99].

Una POU consta de los elementos ilustrados en la figura 4.4:

- Tipo de POU y nombre, en el caso de las funciones, también se especifica el tipo de datos.
- Parte de declaración, con declaraciones de variables
- Unidad de POU con instrucciones.

En la parte de declaración se definen todas las variables que se van a utilizar dentro de una POU. Aquí se hace una distinción entre variables visibles desde fuera de la POU (interfaz POU) y las variables locales de la POU. Dentro de la parte de código (cuerpo) de una POU, se aloja el circuito lógico o algoritmo que se

programa usando el lenguaje de programación elegido, por ejemplo IL o diagramas Ladder, por lo que las declaraciones y las instrucciones se pueden programar en forma gráfica o textual.

4.4. FUNCIONES Y BLOQUES DE FUNCIONES

4.4.1. FUNCIONES

La idea básica de una función (FUN), de acorde a la definición de la especificación IEC 61131-3, es que las instrucciones del cuerpo de la función utilicen los valores de las variables de entrada y den como resultado un valor de función no ambiguo, es decir, para el mismo conjunto de valores de entrada se produce siempre la misma salida, independientemente de la frecuencia o el momento en que se llame a la función. Esta definición de función coincide completamente con el concepto de función de lenguajes de programación estructurados como C o Pascal. A diferencia de los bloques de función o FBs, las funciones no tienen memoria. Las funciones se pueden usar como instrucciones en lenguajes como IL (lenguaje de instrucciones) o ST (Lenguaje Estructurado). En este sentido, las funciones se pueden ver como instrucciones específicas desarrolladas por el propio fabricante del autómata o por el programador como parte del conjunto de instrucciones para programar el PLC.

Al igual que los FBs, las funciones también son accesibles en todo el proyecto, es decir, conocidas por todas las POU de un proyecto de PLC. Con el propósito de simplificar y unificar la funcionalidad básica de un sistema PLC, la especificación IEC 61131-3 predefine un conjunto de funciones estándares, de uso frecuente, cuyas características y comportamiento en tiempo de ejecución e interfaces de llamada están también estandarizados.

TIPOS DE VARIABLES EN FUNCIONES Y EL VALOR DE LA FUNCIÓN.

Las funciones tienen varias restricciones en comparación con otros tipos de POU. Estas restricciones son necesarias para garantizar que las funciones sean verdaderamente independientes (sin efectos secundarios) y para permitir el uso de funciones dentro de expresiones. Como ya hemos comentado, las funciones tienen cualquier cantidad de parámetros de entrada y exactamente un valor de función (retorno) o parámetro de salida. El valor de la función puede ser de cualquier tipo de dato, incluidos los tipos de datos derivados. Por lo tanto, un valor

booleano simple o una palabra doble de punto flotante, es igual de válido, al de una matriz o una estructura de datos compleja formada por varios elementos.

Todos los lenguajes de programación recogidos por la especificación IEC 61131-3 usan el nombre de la función como una variable especial dentro del cuerpo de la propia función para asignar explícitamente el valor de retorno de la función. Como las funciones siempre devuelven el mismo resultado, cuando se proporcionan los mismos parámetros de entrada, no pueden almacenar resultados temporales, información de estado o datos internos entre sus invocaciones, es decir, operan "sin memoria".

Las funciones pueden usar variables locales para resultados intermedios, pero se perderán al terminar la función, por lo tanto, las variables locales no pueden declararse como variables de tipo persistente. Las funciones no pueden llamar a bloques funcionales, como temporizadores o contadores, además el uso de variables globales dentro de las funciones no está permitido.

La especificación no estipula cómo un sistema PLC debe tratar las funciones y los valores actuales de sus variables después de un corte de energía. La POU que llama a la función es por lo tanto responsable de realizar copias de seguridad de las variables cuando sea necesario. En cualquier caso, tiene sentido utilizar FB en lugar de funciones si se procesan datos importantes que deban de ser almacenados de forma persistente. Además, tanto en funciones como en FBs no está permitida la declaración de variables directamente relacionadas con direcciones de E/S físicas del autómata. En el siguiente ejemplo que muestra cómo se declara y se invoca a una función que realiza el cálculo de la raíz cuadrada de un número dado. Los parámetros que devuelve la función son el valor de la raíz cuadra y un indicador de error que indica que no se ha podido calcular la raíz cuadrada, por ejemplo en el caso números negativos).

```
FUNCTION SquareRoot : INT (* square root calculation *)
(* start of declaration part *)
VAR_INPUT                (* Input parameter *)
        VarIn : REAL;    (* input variable *)
END_VAR
VAR_TEMP                 (* temporary values *)
        Result : REAL;   (* local variable *)
END_VAR
VAR_OUTPUT               (* output parameter *)
        Error : BOOL;    (* flag for root from neg. number *)
END_VAR

(* start of instruction part *)
        LD VarIn          (* load input variable *)
        LT 0              (* negative number? *)
        JMPC M_error             (* error case *)
        LD VarIn          (* load input variable *)
        SQRT              (* calculate square root *)
        ST Result         (* result is ok *)
        LD FALSE          (* logical "0" for error flag: reset *)
        ST Error          (* reset error flag *)
        JMP M_end         (* done, jump to FUN end *)
        M_error:          (* handling of error "negative number" *)
        LD 0              (* zero, because of invalid result in *)
                                 (* case of error *)
        ST Result         (* reset result *)
        LD TRUE           (* logical "1" for error flag: set *)
        ST Error          (* set error flag *)
        M_end:
        LD Result         (* result will be in function value! *)
        RET
(* FUN end *)
END_FUNCTION
...                                          ...
```

Código 4.8: Declaración de una función que calcula la raíz cuadrada de un número en IL [101].

4.4.2. BLOQUES DE FUNCIÓN

Los bloques de funciones son los principales bloques de construcción para estructurar programas de PLC. Pueden ser utilizados por programas y además se pueden realizar llamadas a otros FBs o a funciones dentro de ellos. Asociado al concepto de FB, la especificación IE 61131-3 contempla el concepto de instancia de FB. Es el único tipo de POU que puede ser instanciado y por ello antes de profundizar en las características que presenta un FB, hay que tener claro el concepto de instancia.

¿QUÉ ES UNA INSTANCIA?

Cuando se declara una variable especificando su nombre y el tipo de dato de la misma, se está realizando una instanciación del tipo de dato especificado para la variable. En el siguiente ejemplo podemos ver como la variable *Valve* es una instancia del tipo **BOOL**:

```
(Nombre de la variable)          (Tipo de dato)
Valve:                           BOOL; (*Boolean variable *)

(Nombre de la instancia          (Tipo de FB definido por
de FB)                           el usuario)
Motor1:                          MotorType;(*FB instance *)
```

Código 4.9: Declaración de una variable como de un tipo predefinido de FB creado por el usuario [101].

Los bloques de funciones también se pueden instanciar como variables. En el ejemplo anterior la instancia de FB Motor1 se declara como una instancia del bloque de función definido por el usuario (tipo de FB) MotorType. Después de la instanciación se puede usar un FB como una instancia y llamarlo dentro de la POU en la que se declara, como si de una variable más se tratase.

Las declaraciones de FB se deben de hacer, al igual que ocurre con todas las variables, en la parte de declaraciones de cada POU. El sistema de programación puede generar automáticamente números internos absolutos para estas variables de FB al compilar la POU en código de máquina para el PLC. Con la ayuda de estos nombres de variables, el programador puede usar diferentes temporizadores,

contadores o bloques de función creados como tipos de FB, de forma transparente y sin la necesidad de verificar los conflictos de nombres. Mediante la creación de instancias, IEC 61131-3 unifica el uso de los FB propios de cada fabricante, temporizadores y contadores normalmente, y los FBs definidos por el usuario. Los nombres de instancia corresponden a los nombres simbólicos o a los denominados símbolos, utilizados por muchos sistemas de programación de autómatas.

BLOQUES DE FUNCIÓN

El término "bloque de función"se usa a menudo con dos significados ligeramente diferentes. Sirve para referirse al nombre de instancia de FB y también para hacer referencia al tipo de FB (nombre del propio FB). Para nosotros,"bloque de función" o "bloque funcional" significará tipo de FB, mientras que una instancia FB siempre se indicará explícitamente como un nombre de instancia.

En el siguiente ejemplo se puede ver una declaración tanto de variables de tipos de datos simples, como de tipos de FB, en este caso estándares (contadores y temporizadores):

```
VAR
  FillLevel : UINT;              (* unsigned integer variable *)
  EmStop : BOOL;                 (* Boolean variable *)
  Time9 : TON;                   (* timer of type on-delay *)
  Time13 : TON;                  (* timer of type on-delay *)
  CountDown : CTD;               (* down-counter *)
  GenCounter : CTUD;             (* up-down counter *)
END_VAR
```

Código 4.10: Declaración variables de bloques de función estandares [101].

Aunque en este ejemplo Time9 y Time13 se basan en el mismo tipo de FB para temporizadores, *"TON"*, son bloques de temporizador que se pueden llamar por separado como instancias independientes y representan dos temporizadores diferentes.

```
TYPE CTUD : (* data structure of an FB instance of FB type CTUD *)
        STRUCT
                (* inputs *)
                CU : BOOL;              (* count up *)
                CD : BOOL;              (* count down *)
                R : BOOL;               (* reset *)
                LD : BOOL;              (* load *)
                PV : INT;               (* preset value *)

                (* outputs *)
                QU : BOOL;              (* output up *)
                QD : BOOL;              (* output down *)

                CV : INT;               (* current value *)
        END_STRUCT;
END_TYPE
```

Código 4.11: Declaración alternativa utilizando estructuras para declarar bloques de función estándar [101].

La estructura de datos del ejemplo anterior contiene los parámetros formales (interfaz de llamada) y los valores de retorno del FB estándar CTUD o contador descendente y ascendente. Esta estructura representa la vista de la llamada o llamador del FB CTUD. Las variables locales o externas de la POU se mantienen ocultas. Esta estructura de datos es manejada automáticamente por el sistema de programación en tiempo de ejecución y es fácil de usar a la hora de asignarle valores a los parámetros de entrada del FB, como se muestra en el siguiente ejemplo utilizando IL (Lista de instrucciones):

```
        LD 34
        ST Counter.PV           (* preset count value *)
        LD % IX7.1
        ST Counter.CU           (* count up *)
        LD % M3.4
        ST Counter.R            (* reset counter *)
        CAL Counter             (* invocation of FB *)
                                (* with actual parameters *)
        LD Counter.CV           (* get current count value *)
```

Código 4.12: Parametrización e invocación de FB [101].

En este ejemplo la instancia del contador tiene asignados los parámetros *34* , *%IX7.1* y *% M3.4*, antes de que se llame al contador por medio de la instrucción *CAL*. El valor del contador actual se puede leer, mediante la instrucción *LD Counter.CV*. Como se ve en el ejemplo anterior, se accede a las entradas y salidas del FB utilizando el nombre de instancia de FB y un punto, al igual que en el caso de las estructuras de datos. Los parámetros de entrada o salida no utilizados tienen valores iniciales que se pueden definir dentro del propio FB.

INSTANCIACIÓN SIGNIFICA "MEMORIA"

Cuando se declaran varios nombres de variables para el mismo tipo de FB, se crea una "copia de datos FB" para cada instancia en la memoria del PLC. Estas copias contienen los valores de las variables locales (*VAR*) y las variables de entrada o salida (*VAR_INPUT*, *VAR_OUTPUT*), pero no los valores para *VAR_IN_OUT* (estos son solo punteros a las variables, no los valores en sí). Esto también ocurre en el caso de *VAR_EXTERNAL* (estas son variables globales).

Esto significa que la instancia puede almacenar valores de datos locales y parámetros de entrada y salida en varias invocaciones, es decir, tiene una especie de "memoria", la cual es importante para algunos tipos de FB, como los flip-flops o los contadores, ya que su comportamiento depende del estado actual de sus variables internas y de los valores de las entradas del FB, respectivamente. Todas las variables de esta memoria se almacenan en un área de memoria que está ligada a la instancia de FB,y por lo tanto debe ser estática. Particularmente en el caso de bloques de funciones que manejan áreas de datos grandes como tablas o matrices, esto puede conducir a requisitos de memoria estática grandes a la hora de trabajar con instancias de FBs. Para solucionar esta problemática, la especificación IEC 61131-3 contempla el tipo de variable *VAR_TEMP*. Este tipo de variables se pueden utilizar para definir variables cuyo valor no deba de ser mantenido entre diferentes llamadas del mismo FB. En este caso, el sistema de programación usa un área dinámica o pila para crear espacio de memoria que solo será válido mientras se ejecuta la instancia del FB. Por otro lado un gran número de parámetros de entrada y salida puede llevar a un gran consumo de memoria de la instancia de un FB, esto se puede solucionar parcialmente utilizando parámetros del tipo *VAR_IN_OUT* en lugar de *VAR_INPUT* y *VAR_OUTPUT*.

Cuando se trabaja con instancias de FB, hay que tener en cuenta los siguientes

aspectos:

- Los parámetros de entrada (parámetros formales) de una instancia de FB mantienen sus valores hasta la próxima invocación. Si el FB llamado cambia sus propias variables de entrada, estos valores serían incorrectos en la próxima llamada de la instancia de FB, y esto no sería detectado por la POU que está invocando dicho FB.

- Del mismo modo, los parámetros de salida (valores de retorno) de una instancia de FB mantienen sus valores entre llamadas. Permitir que la POU que invoca el FB altere estos valores, daría como resultado un comportamiento erróneo para el el FB invocado, ya que este podría hacer suposiciones incorrectas sobre el estado de sus propias salidas.

Al igual que las variables normales, las instancias de FB también pueden volverse retentivas utilizando la palabra clave *RETAIN*, es decir, mantienen su información de estado local y los valores de su interfaz de llamada durante los cortes de corriente del PLC.

RELACIÓN ENTRE INSTANCIAS DE FB Y BLOQUES DE DATOS

Antes de llamar a un FB convencional, que no tiene memoria de datos local (además de los parámetros formales), es una práctica común activar un bloque de datos que contenga, por ejemplo, parámetros o datos específicos de FB. Dentro del FB, el bloque de datos también puede servir como área de memoria de datos local. Esto significa que los programadores pueden usar un FB convencional con ”datos de instancia” individuales, pero deben garantizar la asignación inequívoca de los datos al propio FB. Estos datos también se conservan entre llamadas al FB, porque el bloque de datos es un ”área de memoria compartida” global, como se muestra en el siguiente ejemplo:

```
a) Conventional DB/FB pair
        JU DB 14 (* global DB *)
        JU FB 14 (* FB call *)
        ...
b) FB instance in IEC 61131-3
```

```
VAR_GLOBAL
        FB_14 : FB_Ex; (* global instance *)
END_VAR
        CAL FB_14 (* invocation of FB instance*)
        ...
```

Código 4.13: Ejemplo de utilización conjunta de DB/FB [101].

Este tipo de creación de instancias está restringido a bloques de función y no es aplicable a funciones. Los programas se instancian de forma similar y se llaman como si fuesen instancias, pero este tipo de instancia (más potente) son diferentes a la de los FBs, ya que genera la creación de programas en tiempo de ejecución por asociación con diferentes tareas.

Los bloques de funciones están sujetos a ciertas restricciones, que los hacen reutilizables en programas de PLC:

- La declaración de variables ”locales” en las que se hace una asignación directa de direcciones de hardware del PLC no está permitida en los bloques de funciones. Esto asegura que los FB son independientes del hardware específico. Sin embargo, el uso de direcciones de PLC como variables globales del tipo *VAR_EXTERNAL* si se permite.

- La declaración de las rutas de acceso del tipo de variable *VAR_ACCESS* o las variables globales con *VAR_GLOBAL* tampoco están permitida dentro de los FB. Se puede acceder a los datos globales y, por lo tanto, a las rutas de acceso indirecto a través de *VAR_EXTERNAL*.

- Los datos externos solo pueden ser pasados al FB por medio de la interfaz POU usando parámetros y variables externas. No hay ”herencia” como en algunos otros lenguajes de programación.

Como resultado de estas características, los bloques de función también se denominan bloques encapsulados, lo que indica que pueden ser utilizados universalmente y no tienen efectos secundarios no deseados durante la ejecución, una propiedad importante a la hora de utilizarlos en ciertas partes de un programa de PLC.

Los datos de trabajo de un FB son datos locales y por lo tanto, un FB no depende directamente de variables globales, E/S o rutas de comunicación de todo el sistema. Los FBs pueden manipular dichas áreas de datos solo de forma indirecta a través de su interfaz (bien documentada). Las principales propiedades de un FB son:

- Un bloque de funciones es una estructura de datos encapsulada e independiente con un algoritmo definido que trabaja con estos datos.
- El algoritmo está representado por la parte del código del FB. La estructura de datos se corresponde con la instancia de FB y se puede invocar, algo que no es posible con las estructuras de datos normales. De cada tipo de FB se puede derivar cualquier cantidad de instancias, cada una de ellas independiente de la otra. Cada instancia tiene un nombre único con su propia área de datos.

Gracias a estas características, la especificación IEC 61131-3 considera que los bloques de funciones están "orientados a objetos". Sin embargo, estas características no deben confundirse con las características de los modernos lenguajes de programación orientados a objetos OOP actuales como, por ejemplo C# o Java con su jerarquía de clases. Para resumir, los FB trabajan en su propia área de datos que contiene entradas, salidas y variables locales. En los sistemas de programación de autómatas previos, los FB generalmente trabajaban en áreas de datos globales con flags o marcas, memoria compartida, E/S y bloques de datos.

TIPOS DE VARIABLES EN FBS

Un bloque de función puede tener cualquier cantidad de parámetros de entrada y salida, incluso puede que no tenga ninguno y puede usar variables locales y externas. Además, como una alternativa para hacer que un FB tenga memoria ante cortes de corriente, las variables locales o de salida también pueden declararse como remanentes dentro de la parte de declaración del FB, a diferencia de la propia instancia de FB que puede declararse retentiva utilizando *RETAIN*. Los valores de los parámetros de entrada o entrada/salida no pueden declararse retentivos en la parte de declaración de FB, ya que la POU es la encargada de proporcionar estos valores y deben declararse como remanentes en la propia POU.

Para variables del tipo *VAR_IN_OUT* se debe tener en cuenta que los punteros a las variables se pueden declarar también persistentes en una instancia dada utilizando también la propiedad *RETAIN*. Sin embargo, los valores correspondientes pueden perderse si no se declaran también como remanentes en la POU que los invoca.

4.5. LENGUAJES DE PROGRAMACIÓN

La especificación IEC 61131-3 proporciona tres lenguajes de texto y tres lenguajes gráficos para escribir programas para autómatas. Los lenguajes textuales son:

- Listas de Instrucciones (IL)
- Texto estructurado (ST),
- Funciones Gráficas Secuencias (Versión en texto).

Y los lenguajes gráficos:

- Diagramas Ladder (LD)
- Diagramas de bloques de función (FBD)
- Funciones Gráficas Secuencias (SFC) (graphical version)

En lenguaje textual IL el código está formado por una secuencia de instrucciones. Dichas instrucciones consisten en un operador o una función más un número de operandos (parámetros). Un operador generalmente tiene un operando, aunque puede no tenerlo, y una función puede tener uno o más parámetros, aunque también podría darse el caso de que no tenga ninguno.

El código generado mediante el lenguaje textual ST consiste en una secuencia de expresiones y declaraciones. ST se puede considerar un lenguaje de alto nivel, el cual está formado por una combinación de palabras clave que controlan la ejecución del programa.

En el caso de los lenguajes gráficos se utilizan elementos gráficos para especificar el comportamiento deseado del autómata. Las líneas de conexión o los llamados conectores indican el flujo de datos entre las funciones y los bloques de

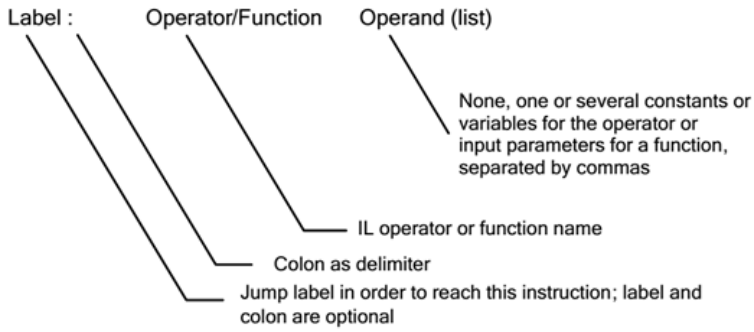


Figura 4.5: Estructura de una instrución en IL [99].

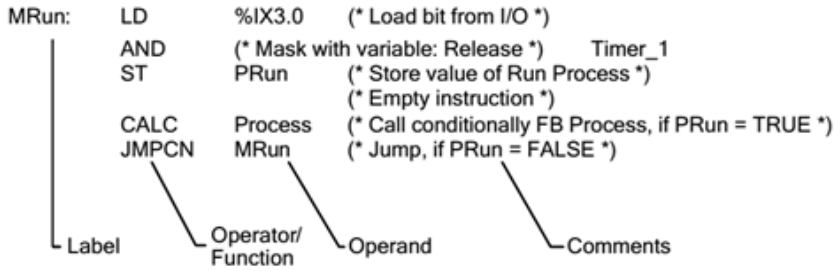


Figura 4.6: Varias Instrucciones IL [99].

función. En las siguientes secciones se discute la estructura fundamental de las construcciones de cada uno de estos lenguajes, tanto gráficos como textuales.

4.5.1. LENGUAJES DE PROGRAMACIÓN TEXTUALES

LISTA DE INSTRUCCIONES IL

IL es un lenguaje de programación de tipo ensamblador, universalmente utilizado y que se emplea a menudo como un lenguaje intermedio común, al cual se traducen los otros lenguajes textuales y gráficos. IL es un lenguaje organizado en líneas en el que una instrucción, que es un comando ejecutable para el PLC, se describe en exactamente una línea. También se permiten instrucciones vacías en forma de líneas en blanco. Una declaración en IL está formada por los elementos representados en la figura 4.5.

La etiqueta y el comentario de una línea IL son opcionales. A partir de la segunda edición de la especificación IEC 61131-3 los comentarios se permiten no

sólo al final de una línea,sino también en cualquier sitio en el que se pueda insertar un espacio en blanco, al igual que otros lenguajes de programación. Estos se delimitan utilizando un par de asteriscos y paréntesis (* *). Esta construcción se utiliza para comentarios informales sobre el contenido de una línea. Las etiquetas son necesarias para permitir que las instrucciones de salto indiquen a donde se tiene que dirigir la ejecución del programa, en función de las condiciones descritas por la instrucción de salto. Los operadores/funciones individuales especifican la operación deseada que se quiere ejecutar. No existe un formato definido (como el número de columna) para los operadores u operandos; ambas partes pueden estar separadas por cualquier número de espacios en blanco o pestañas (ver figura 4.6). El primer carácter de un operador puede comenzar en cualquier columna. El punto y coma ";"no está permitido en IL, ya sea como un carácter de inicio de comentario (como se utiliza en muchos ensambladores) o como un terminador de sentencia (como se utiliza en texto estructurado).

TEXTO ESTRUCTURADO ST

ST es un lenguaje de Alto Nivel ya que no utiliza operadores de bajo nivel o instrucciones máquina como ocurre en IL, sino que ofrece una amplia gama de instrucciones abstractas que describen una funcionalidad compleja de una manera muy comprimida. Las ventajas de la utilización de ST frente a IL son las siguientes:

- En pocas instrucciones se pueden programar acciones muy complejas.
- Creación de programas de forma estructurada.
- Gran expresividad y facilidad de programación.

Pero estas ventajas también vienen acompañadas de ciertos inconvenientes:

- La traducción al código máquina no es directa y además depende de cómo este escrito el programa ya que se realiza automáticamente mediante un compilador.
- El alto grado de abstracción puede conducir a una pérdida de eficiencia (los programas compilados son en general más largos y más lentos).

Un algoritmo ST se divide en varios bloques o partes denominadas declaraciones. Una sentencia se utiliza para calcular y asignar valores, para controlar el flujo del propio programa y para llamar o dejar un POU.

Declaraciones en ST

Un programa ST consiste en una serie de declaraciones separadas por punto y coma (;). Esto significa que una sentencia puede extenderse sobre varias líneas o varias sentencias pueden ser escritas en una sola línea. Los comentarios se enmarcan entre paréntesis y asteriscos "(* comentario *)", y pueden ser utilizados dentro de declaraciones en todos los lugares donde se permiten espacios:

```
A := B (* elongation *) + C (* temperature *);
```

Aunque el lenguaje está en formato libre, es recomendable que el usuario utilice las funciones de representación y comentario como una ayuda adicional a la documentación del programa para mejorar la legibilidad del mismo. La parte de una sentencia que combina varias variables y/o llamadas de función para producir un valor se denomina expresión.

4.5.2. LENGUAJES BASADOS EN GRÁFICOS

FUNCTION BLOCK DIAGRAM FBD

El lenguaje de diagrama de bloques de función (FBD) proviene originalmente del campo del procesamiento de señales, donde los datos en coma flotante o decimales son especialmente importantes. FBD se basa en la utilización de redes, elementos gráficos y conexiones LD (Leadder Diagram). La representación de una POU utilizando los lenguajes gráficos FBD o LD incluye las siguientes partes, que son comunes a los lenguajes textuales:

- Una parte principal y final de la POU.
- La parte de declaración.
- La parte de código.

La parte de la declaración puede ser gráfica o textual. La mayoría de los sistemas de programación soportan, al menos, una declaración textual. La parte de

código se organiza en redes,que son útiles para estructurar el flujo de control de una POU. Una red consiste en:

- Etiqueta de la red.
- Comentarios de la red.
- Gráfico de red.

Etiquetas de Red

Cada red (o grupo de redes) introducida por un salto desde otra red, tiene un prefijo en forma de un identificador alfanumérico definido por el usuario o un entero decimal sin signo 4.7. Esto es lo que se denomina etiqueta de red. Los sistemas de programación suelen numerar las redes consecutivamente. Esta numeración consecutiva de todas las redes se actualiza automáticamente cuando se inserta una nueva red. Esto hace que las redes sean más rápidas de encontrar y podemos ver esta numeración como los números de línea en lenguajes textuales.

Los elementos gráficos de una red FBD incluyen cajas rectangulares y declaraciones de flujo de control conectadas por líneas horizontales y verticales. Las entradas no conectadas de las cajas pueden tener variables o constantes asociadas a ellas. Las entradas/salidas también pueden permanecer abiertas o libres.

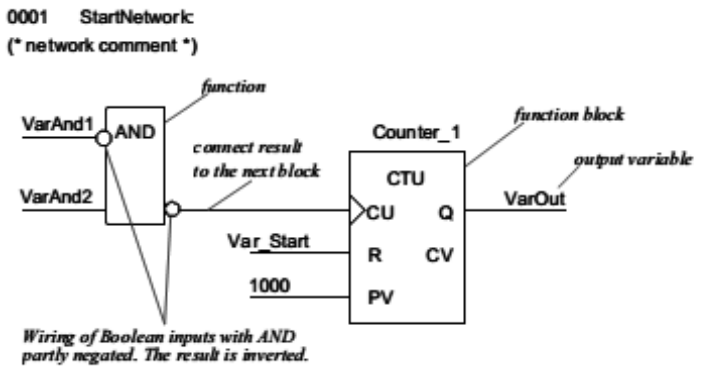


Figura 4.7: Elementos de red FBD [99].

DIAGRAMAS LADDER LD.

El lenguaje Diagrama Ladder (LD) proviene del campo de los sistemas de contactores electromecánicos y describe el flujo eléctrico a través de la red en una

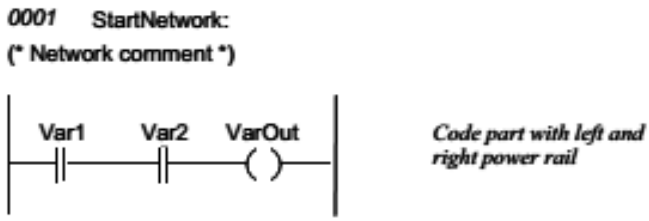


Figura 4.8: Ejemplo de red LD [99].

POU de izquierda a derecha. Este lenguaje de programación está diseñado principalmente para procesar señales booleanas (1 = TRUE o 0 = FALSE).

Elementos gráficos de LD

Al igual que en FBD, LD también utiliza redes y estas siempre se procesan en secuencia de arriba a abajo, a menos que se especifique lo contrario por el usuario. Una red LD está limitada por los llamados conectores eléctricos a izquierda y derecha. Desde el conector más a la izquierda, ”accionado” por el estado lógico 1 o ”paso de corriente”, la corriente alcanza todos los elementos conectados a este. Dependiendo de su estado lógico, los elementos permiten que la corriente pase a los siguientes elementos conectados, o por el contrario que se interrumpa el flujo de corriente.

DIAGRAMAS DE FUNCIONES SECUENCIALES SFC

El lenguaje SFC contemplado por la especificación IEC 61131-3 se definió para descomponer un programa complejo en unidades manejables más pequeñas, y para describir el flujo de control secuencial entre estas unidades. Utilizando SFC es posible diseñar procesos alternativos y paralelos. La temporización de la ejecución de estas unidades depende tanto de las condiciones estáticas (definidas por el programa) como de las condiciones dinámicas (comportamiento de las E/S).

La metodología de SFC se ha derivado de técnicas bien conocidas como las redes de Petri [103] o metodologías de secuenciación en cascada. El primer lenguaje en el mercado de la automatización que describe un sistema por varios estados y condiciones de transición fue Grafcet (Telemechanique, Francia)[104]. Este lenguaje se ha utilizado como base para una norma internacional (CEI 848) [60], que

posteriormente ha sido integrada en la especificación IEC 61131-3.

Los procesos con un comportamiento basados en transiciones de estados son especialmente adecuados para la estructuración de SFC. Por ejemplo en un proceso químico se añaden líquidos a un recipiente hasta que los sensores informen que el recipiente está lleno (paso 1). Un mezclador se activa después de cerrar la válvula de líquido (paso 2). Cuando se termina la mezcla requerida, el motor del mezclador se detiene y comienza el proceso de recipiente vacío"(paso 3).

SFC es principalmente un lenguaje gráfico, aunque también existe una descripción textual, principalmente para permitir el intercambio de información entre diferentes sistemas de programación. A la hora de desarrollar la estructura SFC de una POU, la versión gráfica es la mejor opción ya que podemos visualizar las dependencias e interdependencias de los diferentes pasos de forma más clara.

Las reglas para comentarios dentro del lenguaje SFC son las mismas que para los demás lenguajes de la especificación IEC 61131-3. En la forma textual, la combinación de paréntesis/asterisco (* comentario *) puede utilizarse donde quiera que se permitan espacios en blanco. No hay reglas para la forma gráfica, el formato de comentarios y el tipo son específicos de la implementación realizada por la plataforma de programación del fabricante. En la figura 4.9 podemos ver un ejemplo gráfico de SFC.

4.6. PARADIGMAS DE PROGRAMACIÓN PARA DISPOSITIVOS INDUSTRIALES

El desarrollo de lenguajes de programación en el dominio de las TIC ha sufrido un desarrollo continuo desde sus orígenes [105]. En la actualidad disponemos de paradigmas como la programación orientada a objetos [106], sistemas expertos [107], sistemas de objetos activos o agentes [108], arquitecturas orientadas a servicios SOA [49] o el desarrollo de software dirigido por modelos [109]. Todos estos avances se han intentado llevar tímidamente al mundo de los sistemas software industriales desde el dominio académico y de investigación [1], pero estas nuevas innovaciones no ha tenido un verdadero impacto en los sistemas industriales, fundamentalmente debido al carácter conservador tanto de los fabricantes de PLCs, como de los usuarios finales de estos, los cuales suelen ser reacios a utilizar innovaciones que no hayan sido suficientemente probadas o aceptadas

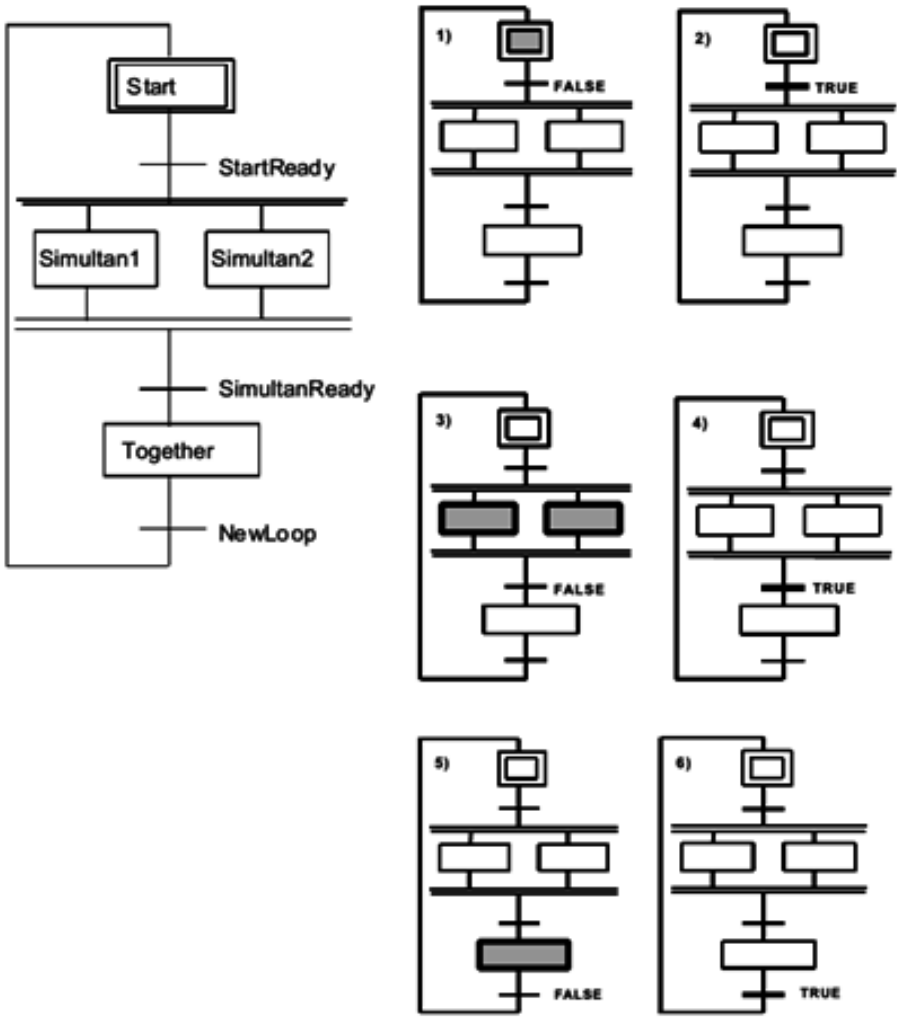


Figura 4.9: Ejemplo de diagramas de funciones secuenciales SFC [99].

en el entorno industrial.

La norma IEC 61131 proporciona una primera aproximación para estandarizar el mundo de la programación de PLCs [103]. La evolución de esta norma, que ha permitido recoger los nuevos paradigmas de programación que han aparecido o que están apareciendo en el mundo de las TIC, parece estancada [1]. Aunque en sus últimas especificaciones existen intentos por contemplar paradigmas como la programación orientada a objetos o arquitecturas orientadas a servicios [86], lo cierto es que la utilización de estos paradigmas por parte de los principales fabricantes de PLCs dista bastante de ser una realidad. La gran mayoría de fabricantes de PLCs que siguen la norma IEC 61131, y en concreto, su especificación número 3, proporcionan entornos de programación que siguen esquemas clásicos como la programación estructurada o la programación modular, dejando en una terreno más de investigación o académico paradigmas como la programación dirigida por objetos o el desarrollo de software dirigido por modelos.

4.6.1. PROGRAMACIÓN ESTRUCTURADA

La programación estructurada ha sido el paradigma tradicionalmente utilizado a la hora de realizar sistemas o programas secuenciales [103]. Sus orígenes se pueden remontar a los inicio de la informática, en los que se creaban programas utilizando instrucciones máquina o ensamblador. Estas instrucciones se ejecutan una tras otra dependiendo de la estructura utilizada a la hora de realizar el programa. Esta estructura puede ser de 3 tipos:

- Secuencial: Las acciones se realizan en el estricto orden en el que aparecen las instrucciones en el código del programa.

- Condicional: Las acciones se realizan en función de que se cumplan ciertas condiciones, cambiando el flujo de ejecución en función de estas. No deja de ser un flujo secuencial, solo que ciertas partes de este flujo puede que se ejecuten o no, o pueden que se ejecuten antes o después, todo depende de los condiciones y de los saltos que se programen.

- Iterativo: Las acciones se realizan de forma repetitiva si se cumplen ciertas condiciones. El flujo de este tipo de programas consiste en realizar una serie

de acciones también de forma secuencial pero varias veces hasta que se cumpla una condición.

En el paradigma de la programación estructurada se pueden combinar estas tres formas de estructurar el código de un programa e incluso podemos anidar estas estructuras unas dentro de las otras. La programación estructurada presenta varias ventajas:

- Los programas son fáciles de entender, ya que con una lectura secuencial del programa podemos entender que es lo que hace.
- Al disponer de programas que son legibles y fáciles de entender las pruebas y la depuración de los mismos son más sencillas.
- Los costes de mantenimiento son menores, debido también a que son programas sencillos y legibles.
- La documentación de este tipo de programas se reduce, ya que las instrucciones por si solas son auto-explicativas.

A pesar de las ventajas que presenta este paradigma, en ciertos casos el principal inconveniente que nos podemos encontrar es que se obtiene un único bloque de programación, lo cual para sistemas complejos puede llegar a ser demasiado complejo y extenso, con lo que su mantenimiento y gestión puede complicarse. Para resolver esta problemática se pueden utilizar de forma conjunta la programación estructurada y la programación modular, dando como resultados programas mixtos que utilizan ambos paradigmas. La combinación de ambos paradigmas de programación puede que sea la forma más extendida de programar un PLC, en la que los módulos se organizan mediante una estructura jerárquica en la que se pueden definir funciones, que a su vez pueden tener dentro otras funciones.

Esta forma de desarrollar un programa de PLC esta contemplada por la norma IEC 61131, y quizás sea la forma de programar un PLC que mejor se adapta a dicha norma, ya que esta especificación recoge conceptos como POUs, FCs y FBs, con los que podemos crear o encapsular conjuntos de instrucciones secuenciales dentro de módulos a modo de bloques de función, funciones o unidades organizativas.

4.6.2. PROGRAMACIÓN MODULAR

La programación modular se base en dividir el programa principal en subprogramas o módulos en los que se ejecutan un conjunto de instrucciones secuenciales, condicionales o iterativas. Este conjunto de instrucciones realizan una sub-tarea muy concreta, que forma parte de la tarea o funcionalidad del programa principal. Esta estrategia pretende hacer el programa más legible y simple utilizando el método de divide y vencerás. Históricamente la programación modular es una evolución de la programación estructurada con la que se pretende resolver sistemas complejos.

Las tareas más simples se han agrupado e implementado utilizando el concepto de función, subrutina o procedimiento. En el mundo de la programación de PLCs y en concreto en la especificación IEC-61131-3 se utiliza el concepto de función FC o bloque de función FB. La utilización de funciones o subrutinas que se invocan desde el programa principal (en este caso desde la POU principal) permite que el control del programa pase el control a la función o subrutina, y este no se devuelve hasta que la función o subrutina haya terminado de ejecutarse.

La principal ventaja de esta forma de programar reside en que para ejecutar una función solo nos tenemos que preocupar de sus datos de entrada y del resultado que esta devuelve, sin tener que entrar en el funcionamiento interno de la función.

Esta estrategia nos permite reutilizar una función todas las veces que sea necesario, sin tener que reescribir el código de la misma, ahorrando tiempo tanto en el desarrollo como en las pruebas de validación de la misma, ya que una vez que se ha comprobado que la función es correcta no nos tenemos que volver a preocupar más de la misma.

La programación modular por consiguiente fue el primer paradigma de programación que proporcionó una forma de reutilizar código, simplificando la creación de este, ya que se podían crear grupos de funciones (librerías), que podían ser utilizadas sin necesidad de conocer su funcionamiento interno. Esto facilitó que diferentes equipos de desarrolladores se pudiesen intercambiar estas librerías, aprovechando el trabajo ya realizado. Hoy en día quizás sea la técnica más utilizada por los equipos de ingeniería que se dedican a desarrollar sistemas softoware industriales, sobre todo a nivel de PLC [103].

4.6.3. PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos, como paradigma para el desarrollo de sistemas software industriales, ha sido explorada por varios autores [110] [111] [112]. Este paradigma puede ayudarnos en el proceso de desarrollo de software, sobre todo en las fases de diseño y modelado, ya que proporciona una descripción muy detalla de los componentes del sistema software, así como de las relaciones que hay entre ellos, con lo que obtenemos un modelo software conceptual que recoge todos los elementos físicos del sistema industrial.

El gran desafío que presenta este paradigma es el de llevar estos modelos conceptuales a un PLC. Nos vamos a encontrar con muchas limitaciones y restricciones, ya que aunque es cierto que en las últimas ediciones de la especificación IEC-61131-3 se han introducido ciertas características de este paradigma, las plataformas de programación de PLCs existentes aún no las soportan completamente y no está claro cómo se van a introducir características como el polimorfismo y la herencia en estos entornos de programación [1].

Por otro lado, en la norma IEC 61499 [113] se especifican y proporcionan ciertas características del paradigma POO a los bloques de función, dotándolos de un encapsulado de datos muy fuerte y de ciertos tipos de eventos que pueden ser vistos como los métodos de un objeto. De todos modos estas nuevas características son solo un conjunto parcial de POO, lo cual nos acerca a este paradigma pero de una forma muy laxa.

Otra posible manera de materializar los modelos conceptuales de POO en programas de PLC puede consistir en dividir el sistema en bloques pequeños, en los que se recogen tanto la parte eléctrica (física) como la parte software (lógica), de forma que se puedan desarrollar independientemente del programa principal; este enfoque recibe el nombre de "Mechatronic Object"[114]. Pero en este tipo de soluciones nos volvemos a encontrar con la dificultad de su despliegue en sistemas reales, sobre todo si no se utilizan las mismas plataformas hardware y software.

4.7. BUENAS PRÁCTICAS DE PROGRAMACIÓN PARA DISPOSITIVOS INDUSTRIALES

Hoy en día los entornos de programación para autómatas programables se rigen por la norma IEC 61131, siendo casi inexistente las plataformas que no se rigen por esta norma [115]. Por tanto, que a la hora de utilizar cualquier metodología de desarrollo de software, tendremos que evaluar si esta es compatible o se puede materializar utilizando esta norma, ya que de lo contrario no podremos desplegar nuestros programas en sistemas industriales reales, por lo menos en lo que a PLCs se refiere.

Como se ha comentado en la especificación 3 de esta norma se recogen las principales características que debe de cumplir un lenguaje de programación así como el modelo de ejecución de programas de un PLC. En esta norma además se definen ciertos principios de Ingeniería del Software para su aplicación en la programación de PLCs tales como la modularización, la utilización de estructuras y la reutilización de código. Además este estándar permite desarrollar aplicaciones software para sistemas industriales de forma jerárquica, que quizás sea una de las estrategias más utilizadas a la hora de desarrollar sistemas software en entornos industriales. Dicha especificación también nos proporciona conceptos genéricos como POUs, en los que podemos enmarcar tanto funciones, como bloques de función o programas. El concepto de POU nos ofrece la capacidad de reutilizar código, ya que una vez definida una función o un bloque de función, la podremos volver a utilizar tantas veces como queramos.

Todas estas características que nos facilita la norma IEC 61131 nos llevan a establecer una serie de directrices a la hora de programar un dispositivo industrial, sobre todo PLCs. La primera y más evidente es que la utilización de la norma de cualquier paradigma o metodología de desarrollo de software que se utilice, debe de poder desplegarse en un PLC utilizando los conceptos y características de dicha norma. Si dicha metodología o paradigma no es compatible o no puede ser desplegada utilizando los conceptos de esta norma, no se podrá utilizar en sistemas industriales reales, y por lo tanto su sentido y utilización se circunscribirá a un marco teórico o académico.

En segundo lugar debemos poder aprovechar al máximo las características que ofrece dicha norma, y que son la reutilización de código mediante la utiliza-

ción de POUs y el uso de estructuras para organizar los datos de un sistema industrial. Quizás esta última característica es más importante en sistemas industriales grandes, con miles o cientos de miles de señales, en los que una buena organización de estas señales es fundamental tanto a la hora de desarrollar un programa de calidad, como a la hora de reducir tiempos de desarrollo y costes. Este aspecto, quizás sea uno de los más complejos y de los que más trabajo requiere, ya que organizar una gran cantidades de señales, ya sean de entrada o salida, si no se sigue una metodología, puede llegar a ser una tarea muy laboriosa. Por ello, utilizar las características que proporciona dicha norma, de forma estructura y siguiendo una metodología puede hacer que se reduzcan tiempos de desarrollo, entregando soluciones software fiables, fáciles de mantener y escalables.

4.8. CONCLUSIONES

El desarrollo de software para dispositivos industriales ha planteado una serie de dificultades, ya que cada fabricante tenía su propia plataforma de programación con sus propias "reglas" y por tanto cada programa o desarrollo era diferente. Esta problemática se solventó parcialmente gracias a la aparición de la norma IEC 61131, la cual recoge en su especificación tercera, todos los aspectos de programación que debe de seguir una plataforma de programación para PLCs. Pero las plataformas de programación de PLCs rara vez siguen todas las especificaciones y características recogidas en la norma IEC 61131. Además, a esta limitación hay que añadirle el escaso soporte de estas plataformas para utilizar técnicas de ingeniería del software como POO, MDE o SOA [1], incluso cuando en las ultimas especificaciones de esta norma se recogen ciertos aspectos de algunos de estos paradigmas de programación como, por ejemplo, POO.

Las plataformas de programación comerciales de PLCs, de momento, no están adoptando estos aspectos. No obstante la utilización de técnicas de software para sistemas basados en PLCs hoy en día ha pasado del ámbito académico o de investigación [116], a ser una necesidad real a la hora de desplegar un sistema software industrial, debido a varias razones, como por ejemplo el nivel de criticidad de los sistemas software en entornos industriales, siendo este equiparable al de otros sistemas como los hidráulicos o los eléctricos. El desarrollo de estos sistemas se ha convertido en una tarea compleja, tanto por la enorme cantidad de

señales que pueden llegar a manejar, como por la dificultad a la hora de integrar y comunicar todos los elementos/partes físicas de nuestro sistema software. Por otra parte, un sistema de control industrial en muchos casos ya no es una simple rutina o subrutina software que procesa unas entradas y genera unas salidas. En su lugar tenemos programas con cientos de rutinas con lógica bastante compleja y que están monitorizando no solo entradas, sino que comprueban el estado de otros PLCs u otros equipos físicos como, por ejemplo, robots industriales.

Intentar abordar el desarrollo de sistemas software industriales de estas características puede ser una tarea extensa y laboriosa, por lo que aplicar técnicas de ingeniería de software que nos ayuden durante todo el proceso de desarrollo y despliegue se ha convertido en algo esencial. Hay que tener claro, que la utilización de la ingeniería de software, o de cualquier paradigma de programación, tiene que estar contemplado o recogido por la norma IEC 61131, o por lo menos que la norma lo soporte parcialmente, ya que de no ser así, la aplicación de nuestro paradigma de programación en un entorno real será prácticamente imposible.

En este trabajo se propone la utilización de MDE como paradigma de programación para sistemas software industriales. Este paradigma no está soportado de forma directa por la especificación IEC 61131-3, pero existen ciertas características de la especificación, como los tipos de datos definidos por el usuario y los bloques de función FBs, que se pueden utilizar realizando una buena correlación entre los conceptos de MDE y estas características de la especificación, que nos puede facilitar la utilización de este paradigma en el desarrollo software de sistemas basados en PLCs.

5

iMMAS (Industrial meta model for automation systesms)

"First solve the problem. Then write the code."
John Johnson

5.1. Introducción

Los sistemas software en el contexto de los sistemas industriales han ido adquiriendo un papel cada vez más relevante a medida que los sistemas industriales han tenido que ir haciendo frente a procesos productivos más automatizados y especializados.

La aparición de arquitecturas jerarquizadas en los sistemas industriales ha permitido organizar los elementos tanto hardware como software que forman parte de estos sistemas [117]. Desde el punto de vista software la descomposición del sistema industrial en un conjunto de sistemas software que se ejecutan sobre elementos hardware en una estructura jerarquizada facilita la gestión de aspectos como la seguridad, interoperabilidad y escalabilidad del sistema software industrial.

La creación y construcción de sistemas software en entornos industriales es, en general, una tarea compleja y laboriosa dado que requiere la construcción y puesta en marcha de muchos elementos software (dispositivos industriales, OPC, SCADA, etc.) que tienen que coordinarse entre sí para lograr que el proceso o procesos industriales funcionen correctamente [26].

En el caso del desarrollo de sistemas software para dispositivos industriales como PLCs y RTUs, los fabricantes de estos dispositivos han desarrollado sus pro-

pias herramientas y lenguajes de programación con su propia sintaxis y semántica, obteniendo como resultado programas ejecutables muy dependientes de la plataforma de programación y del fabricante. Aunque la norma IEC 61131 en su especificación tercera define los aspectos que debe tener una plataforma de programación para PLCs y RTUs, la aplicación parcial de la norma en algunas herramientas de programación y el escaso soporte de paradigmas como la programación orientada a objetos o el desarrollo dirigido por modelos han dificultado su difusión y utilización.

Por otra parte, la utilización del estándar OPC, materializado principalmente en servidores OPC, ha cubierto en gran medida las necesidades de intercambio de información entre los dispositivos industriales y los sistemas software que monitorizan y supervisan un proceso industrial [89] [90] [91]. La última especificación del estándar OPC, OPC UA, que no se encuentra soportada completamente por las distintas herramientas comerciales existentes actualmente, proporciona un modelo de comunicación no dependiente de la tecnología subyacente y un novedoso modelo de información que permite modelar y organizar los datos del proceso industrial que manejan los dispositivos como PLCs y RTUs. El modelo de información además permite especificar su comportamiento utilizando, o incluso definiendo, estructuras complejas de objetos jerárquicas y no jerárquicas. La incorporación de modelos simplifica la descripción de grandes conjuntos de datos, lo que facilita la comprensión de dichos datos por parte de los ingenieros y desarrolladores.

En definitiva el desarrollo del software en ámbitos industriales se ha dedicado fundamentalmente al diseño de un conjunto de aplicaciones software para los distintos elementos o partes del sistema industrial, como el software de control en los PLCs, la gestión de los datos de proceso en el OPC, la visualización de dichos datos en entornos SCADA, etc. Adicionalmente, y en función de la arquitectura utilizada en el sistema industrial, las aplicaciones de todos estos elementos tienen que integrarse estableciendo mecanismos para el intercambio de información, que permitan explotar y optimizar el proceso productivo que se monitoriza y se supervisa.

El desarrollador de software para entornos industriales tiene por tanto que realizar distintos tipos de actividades como por ejemplo diseñar el software de

control de un PLC, programar y organizar los datos/señales que va a manejar el PLC, y diseñar y programar los entornos de visualización que muestran dichos datos al operario que tiene que supervisar dicho proceso. El proceso de despliegue y configuración de cada uno de los elementos que entran en juego permitirá la integración de todos ellos en un único sistema final funcional y operativo [48] [55].

La pregunta que tratamos de responder con este trabajo de tesis es si es posible entender el sistema industrial de una forma global utilizando una descripción basada en modelos, que nos ayude a simplificar el proceso de construcción de sistemas industriales a través de la utilización de las herramientas y plataformas software disponibles en el ámbito industrial.

Para ello, no sólo nos interesa estudiar las posibilidades que ofrece una descripción del sistema industrial en base a modelos, sino investigar, si es posible aplicar de forma sistemática y organizada metodologías y técnicas de construcción de software de la Ingeniería del Software, concretamente en lo que respecta al desarrollo dirigido por modelos como paradigma que nos permita diseñar o configurar el software de los elementos que conforman el sistema industrial [118] [119].

En primer lugar, en este capítulo de tesis, estudiamos en qué consiste la ingeniería dirigida por modelos y las posibilidades que ofrece para el diseño y desarrollo de sistemas software aplicados en el dominio industrial. Posteriormente, se presenta la formalización del lenguaje de modelado que se ha creado específicamente para la descripción de sistemas industriales que hemos denominado iMMAS (Industrial MetaModel for Automation Systems). Dicho lenguaje nos va a permitir modelar y automatizar la construcción del software para sistemas industriales. Así mismo se ha creado un perfil denominado MOSIE (Model for structured industrial environments) a partir del lenguaje de modelado que ofrece iMMAS con la intención de ofrecer un conjunto de conceptos especialmente concebidos para modelar sistemas industriales.

5.2. EL DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

La ingeniera de software ha pretendido desde su aparición industrializar el proceso de construcción y desarrollo de software. Actualmente a la hora de desa-

rrollar un sistema software es necesario realizar muchas tareas de forma manual, por lo que el grado de industrialización en la industria del desarrollo de software sigue siendo escaso si lo comparamos con otras industrias como la automoción o la industria electrónica.

El desarrollo de un sistema software requiere de mucho esfuerzo por parte de los equipos de desarrollo, principalmente porque no es un proceso automático sino un proceso creativo que requiere de muchas horas de trabajo por parte de los programadores/desarrolladores. Por ello la ingeniería de software se ha ido apoyando en paradigmas y técnicas de desarrollo de software, con el objetivo de conseguir un mayor grado de automatización en el proceso de desarrollo de software, y facilitando así el desarrollo de software en su conjunto.

La abstracción ha sido una de las claves que ha permitido reducir la complejidad del proceso de desarrollo, ya que nos permite centrarnos en la descripción de los aspectos importantes del sistema. Así, los compiladores e intérpretes fueron las primeras herramientas software que facilitaron la construcción de sistemas software más complejos utilizando la capacidad de flexibilidad y abstracción que ofrecen los lenguajes de programación de alto nivel frente a los lenguajes de ensamblador y máquina. De esta forma se consiguió una mejora de la calidad del software, la productividad y el mantenimiento.

Dentro del contexto del desarrollo de software basado en código la programación orientada a objetos ha sido otro paradigma que ha tenido mucho éxito, ya que ofrece un grado de abstracción mayor que el que ofrece la programación estructurada, una mejor organización modular en torno al concepto de objeto y por lo tanto su aplicación proporciona sistemas software de mayor calidad. Pero aunque este paradigma ha supuesto una mejora significativa con respecto a la programación estructurada, algunos autores opinan que no proporciona una mejora del grado de automatización desde que se capturan los requerimientos del sistema, por el salto semántico que hay entre la etapa de análisis del problema y la etapa de diseño de la solución [109] [116] .

Frente a esta visión centrada en objetos ha aparecido en los últimos años [7] otra visión centrada en modelos cuyos objetivos siguen siendo los mismos, mejorar el grado de abstracción, flexibilidad, automatización y mantenibilidad, pero en este caso sustituyendo los lenguajes de programación por una descripción del

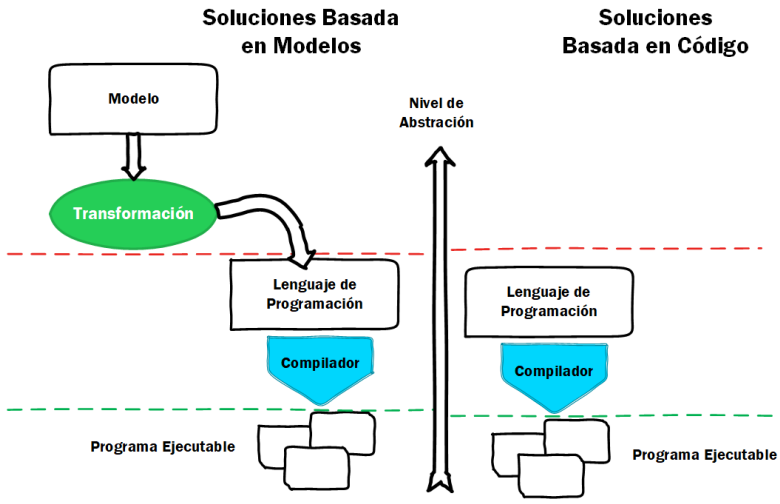


Figura 5.1: Niveles de abstracción, utilizando modelos vs Lenguajes de programación.

sistema en base a modelos.

La utilización de modelos presenta las siguientes ventajas:

- Documentan y recogen los aspectos del sistema software.
- Nos permite razonar sobre lo que se va a construir.
- Nos facilita la comunicación de ideas.
- Definen un lenguaje común que puede ser utilizado por el equipo para discutir todos los aspectos del sistema.
- Facilita la construcción del sistema pudiendo generar código a partir de modelos.

La utilización de modelos como elementos principales para el desarrollo del software constituye la base de un conjunto de paradigmas que ha recibido el nombre de "Ingeniería de Software dirigida por modelos" (MDE en inglés, Model Driven Engineering). MDE establece un conjunto de paradigmas para el desarrollo de sistemas software que gira en torno al concepto de abstracción como forma de abordar problemas complejos. Las abstracciones se capturan en base a modelos que nos permiten definir de forma parcial o total un aspecto o un conjunto de aspectos de un sistema a través de un lenguaje de modelado bien definido [120].

En el dominio del desarrollo de software un modelo nos puede ayudar, por ejemplo, a expresar los requisitos, la estructura o el comportamiento de un sistema software.

Dentro del marco MDE se define el Desarrollo Dirigido por Modelos (MDD en inglés, Model Driven Development) como el paradigma de desarrollo que utiliza modelos para el diseño de sistemas software con distintos niveles de abstracción. Contempla el desarrollo del software como la realización de secuencias sucesivas de transformaciones para, a partir de modelos más abstractos, ir generando modelos más concretos, hasta generar el código final de las aplicaciones que serán desplegados en las plataformas destino. Por tanto, MDD proporciona un enfoque descendente de construcción del software, en el que cada nivel ofrece un nivel de abstracción diferente. En los niveles más bajos la capacidad de abstracción se va perdiendo y se va aumentando el nivel de detalle hasta llegar al nivel concerniente al código. El código es considerado como un modelo más en este enfoque, ya que podemos considerar la representación del código fuente como un modelo o como un texto (fichero) que se compila para generar un programa ejecutable. En este sentido podemos hablar de transformaciones entre modelos (m2m) o entre modelos y texto (m2t).

La capacidad de realizar transformaciones entre modelos y entre modelos y código nos permite automatizar el proceso de construcción del software a través de modelos y, por tanto, automatizar el proceso de generación de código sin intervención de ningún programador o desarrollador. De esta forma, necesitaremos establecer las reglas de transformación entre los conceptos de un modelo y los conceptos del otro modelo, hasta llegar al código final. En consecuencia sería posible disminuir los tiempos y costes derivados del proceso de programación, ya que gran parte del código se podrá obtener directamente del modelo o modelos abstractos que definen el sistema software.

Existen varias aproximaciones para describir abstracciones de forma adecuada y precisa. Para ello podemos utilizar lenguajes de modelado de propósito general como, por ejemplo, UML o lenguajes específicos de dominio (DSL en inglés, domain-specific languages).

Si optamos por la primera opción, la utilización de patrones y factorías de software quizás sea la forma más acertada de dar soporte a MDD [116]. La des-

cripción de los modelos se realiza mediante el lenguaje de modelado UML utilizando diferentes tipos de diagramas de representación, que se definen formalmente mediante metamodelos en base al lenguaje MOF (Meta Object Facility) [121] especificado por OMG.

Por otra parte, la aplicación de DSL nos proporcionará los conceptos de un dominio de aplicación concreto con elementos pensados específicamente para dicho dominio [122]. Los lenguajes específicos de dominio vienen definido por:

- *Sintaxis abstracta:* Define los conceptos del lenguaje, las relaciones dichos conceptos y las reglas que indican si un modelo está bien formado.
- *Sintaxis concreta:* Establece la notación textual o gráfica a utilizar a la hora de construir un modelo.
- *Sintaxis Semántica:* Se define mediante la traducción de conceptos del modelo a conceptos de otro lenguaje destino cuya semántica se conoce.

La sintaxis abstracta de un DSL se puede describir mediante un metamodelo, que no es más que una representación formal del lenguaje, con la que podemos crear o instanciar modelos. Por tanto, el metamodelo se convierte en MDE en un elemento clave para capturar las características y propiedades de un lenguaje de modelado de forma unificada y completa con un contenido semántico bien formado, lo que nos permite definir cómo deben construirse los modelos.

Pero estas dos aproximaciones no son las únicas a la hora de afrontar el desarrollo dirigido por modelos. De hecho, en noviembre de 2000 apareció un enfoque complementario, denominado MDA (Model Driven Architecture en inglés) de la mano de Object Management Group (OMG)[123]. MDA solo es una visión particular de MDD, aunque generalmente se ha confundido de forma errónea con MDD por perseguir objetivos comunes en cuanto al modo en cómo debe realizarse el desarrollo de software dirigido por modelos. En la siguiente sección se profundiza en los aspectos esenciales de MDA para el MDD.

La utilización de paradigmas y técnicas de MDE en el desarrollo del software ofrece además una serie de ventajas adicionales como la posibilidad de realizar actualizaciones y evoluciones de un sistema software ya implantado, facilidad para los proceso de reingeniería o ingeniería inversa de sistemas software, o la posibilidad de confirmación dinámica de sistemas software en tiempo de ejecución.

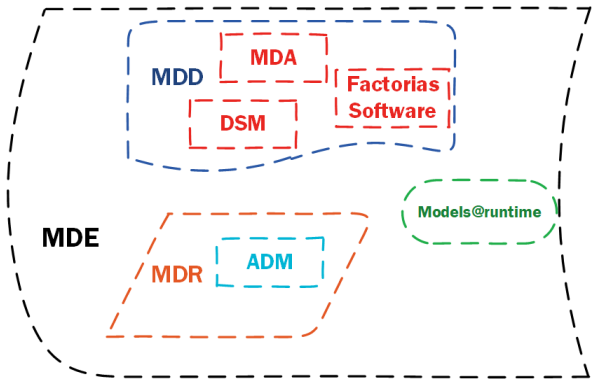


Figura 5.2: Paradigmas que forman MDE.

Así, la reingeniería del software (en inglés MDR, Model Driven Reverse Engineering) tiene como función principal la de reestructurar o rediseñar aplicaciones de software heredadas dentro de las empresas, que con los avances de tecnologías tanto de hardware y software se vuelven obsoletas, para lo cual es necesaria su operación sobre nuevas plataformas [109]. En este campo destaca la propuesta de estandarización de OMG ADM [124]. Por último, en el campo de los sistemas adaptativos, capaces de auto configurarse en tiempo de ejecución, tenemos Models@Runtime como paradigma principal. En la figura 5.2 podemos ver de forma general algunos de los paradigmas en el marco de MDE.

5.2.1. MDA

MDA es un propuesta arquitectónica de OMG [123] en la que se definen mecanismos, herramientas y notación para llevar a cabo una implementación del desarrollo de software dirigido por modelos. De forma general, MDA realiza una separación entre las especificaciones funcionales de los sistemas software, la tecnología de implementación y la plataforma de despliegue. Además ejemplifica la utilización de transformaciones entre modelos hasta obtener las implementaciones finales las cuales pueden ser desplegadas en las plataformas elegidas para ello.

MDA organiza todas las etapas de la ingeniería software en torno a modelos, que se convierten en el centro de todo el proceso de desarrollo del software. Dichos modelos se utilizan para representar todo el sistema software en su conjunto. La definición de modelos mediante lenguajes de modelado proporciona

un nivel de abstracción alto que nos aísla de los detalles de la tecnología tanto de desarrollo como de despliegue. Esto va a permitir evolucionar, modificar o añadir nuevas funcionalidades tan solo modificando el modelo o los modelos que definen nuestro sistema software. Si la descripción basada en modelos es independiente de la tecnología, los modelos pueden también perdurar más en el tiempo al no depender de si las tecnologías se quedan obsoletas o de si se producen cambios drásticos en ellas. Por tanto, podemos decir que la utilización de modelos protegen la inversión realizada en términos de tiempo y dinero, ya que no se empieza de nuevo en cada proyecto ni en las tareas de mantenimiento. Además facilitan la evolución de los sistemas para adaptarse a los cambios derivados de las necesidades que van surgiendo con el paso del tiempo.

La aplicación más ampliamente utilizada de MDA es el mapeo o transformación entre Modelos Independientes de Plataforma (PIM) y Modelos Específicos de Plataforma (PSM). Concretamente se construyen modelos de sistemas que cumplen todos los requisitos funcionales, pero que son completamente independientes de la plataforma, el lenguaje de programación y otros detalles de implementación. Estos modelos reciben el nombre de modelos PIM. A partir de los modelos PIM se pueden construir otros modelos, los cuales contemplan todas las dependencias y detalles necesarios para que el sistema software pueda ser implementado y desplegado utilizando una tecnología concreta. Estos nuevos modelos recibe el nombre de modelos PSM. Por último, a partir de los modelos PSM se puede generar el código de la aplicación final utilizando por ejemplo un compilador.

Todo este proceso se especifica de forma resumida en la figura 5.3. Las transformaciones entre modelos PIM y PSM se suelen realizar utilizando herramientas que traducen los conceptos de un modelo en otro de forma automática, por lo que podemos tener un modelo PIM en el que se captura toda la funcionalidad de un sistema, y transformarlo en un modelo PSM o en varios dependiendo de las necesidades que tengamos. Por ejemplo, podríamos generar a partir de un modelo PIM el modelo de base de datos (modelo relacional) y el modelo de clases de la aplicación que explotaría dicha base de datos. Si se necesitase desplegar el modelo en una nueva plataforma tan solo tendríamos que volver a generar el modelo PSM adaptada a la nueva plataforma, por ejemplo un servicio web. La existencia de los modelos PIM nos permite tener una independencia real con respecto a las

plataformas y, lo que es más importante aún, una rápida adaptación a la hora de desplegar nuestro sistema software en cualquier tipo de plataforma.

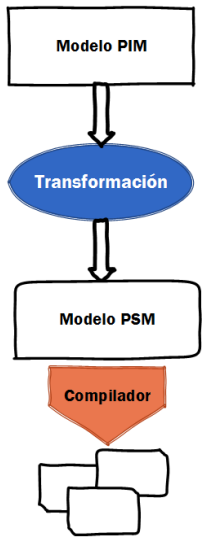


Figura 5.3: Proceso de transformación entre modelos PIM y PSM.

Para poder definir modelos PIM necesitamos un lenguaje y conceptos en los que apoyarnos. Este lenguaje junto con sus conceptos vendrá descrito mediante un metamodelo. El prefijo "meta" es una raíz griega que significa "más allá de". En ingles "meta" se utiliza para indicar que un concepto es una abstracción de otro. Este prefijo se utiliza en informática en términos como metaclase (clase que describe o define a otra clase), metadato (dato que describe otro dato) o metalenguaje (lenguaje que especifica otros lenguajes). Si hacemos la misma analogía con el termino metamodelo, podemos decir que es un modelo que define a otros modelos y, por tanto, el metamodelado sería el paradigma que estudia todos los aspectos relacionados con la creación de metamodelos.

ARQUITECTURA DE METAMODELADO DE CUATRO NIVELES.

Dentro del contexto de MDE un modelo viene descrito por medio de un lenguaje de modelado a partir de otro modelo (metamodelo), que describe la sintaxis abstracta del lenguaje de forma independiente a su notación. Dado que el metamodelo es un modelo a su vez, se necesita un lenguaje de metamodelado para la descripción del metamodelo como por ejemplo MOF. Dicho lenguaje de metamodelado podría representarse por un metametamodelo (un modelo de meta-

modelo), y así sucesivamente.

Para evitar el problema de definir de forma recursiva descripciones en metamodelos con respecto a sus instanciaciones modelos, se establece una arquitectura de metamodelado que especifica la relación que hay entre modelos y metamodelos. La organización OMG propone una arquitectura de metamodelado de cuatro niveles en MDA como la que se muestra en la figura 5.4.

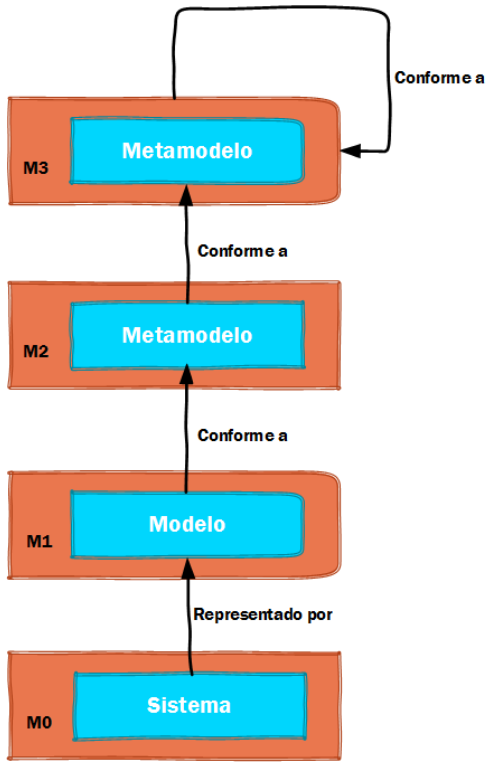


Figura 5.4: Arquitectura de metamodelado segun la OMG[123].

Cada nivel de esta arquitectura se explica a continuación:

- **M0** contiene los datos de la aplicación, como por ejemplo las instancias que forman un sistema orientado a objetos en tiempo de ejecución o las filas en tablas de bases de datos relacionales.
- **M1** contiene la aplicación, es decir, las clases de un sistema orientado a objetos o las definiciones de tablas de una base de datos relacional. Este es el nivel en el que tiene lugar el modelado de la aplicación (el tipo o nivel del modelo).

- **M2** contiene el metamodelo que captura el lenguaje, por ejemplo elementos UML como clase, atributo y operación. Este es el nivel en el que operan las herramientas (el metamodelo o nivel arquitectónico).

- **M3** contiene el meta-metamodelo que describe las propiedades de todos los metamodelos que se pueden definir.

Cada nivel en esta jerarquía representa una instancia de un nivel superior. Como se muestra en la figura **??**, los elementos en M0 son instancias de clases en M1, que pueden ser vistas como instancias de clases de metamodelo, y que a su vez pueden ser vistas como instancias de las clases del meta-metamodelo.

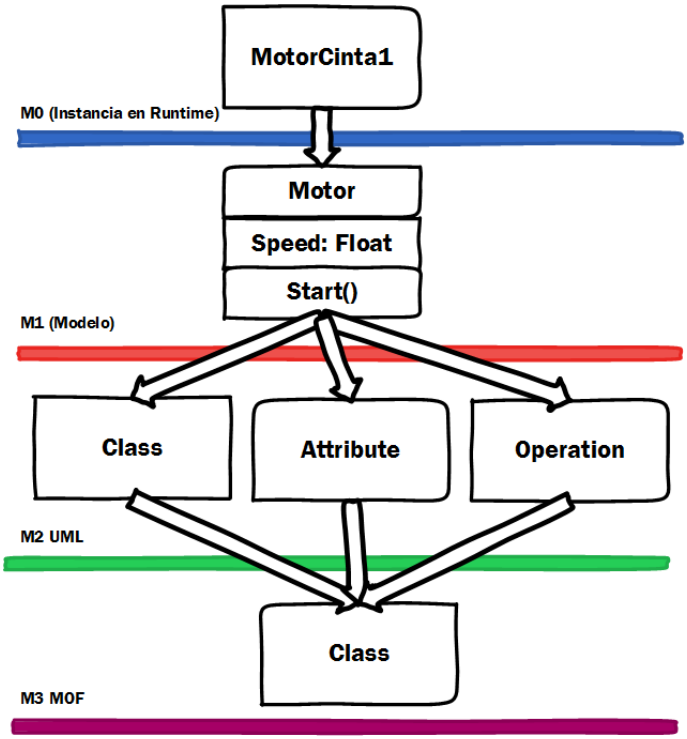


Figura 5.5: Agrupación de los elementos de un proceso de metamodelado, según el nivel al que pertenecen.

El factor unificador en esta arquitectura es el meta-metamodelo, ya que en él se define el conjunto de conceptos simples a partir de los cuales se puede definir metamodelos. De este modo cada modelo se especifica de acuerdo a su metamodelo o, lo que es lo mismo, mediante el lenguaje de modelado que define

este metamodelo. Así se establece una relación entre el metamodelo y el modelo instanciado conforme al metamodelo. Esto se suele ejemplificar mediante una relación de "conformidad" entre el modelo y el metamodelo.

En la figura 5.5 se muestra un ejemplo de conceptos en cada uno de los niveles de la arquitectura de metamodelado utilizando MOF como metalenguaje en el nivel M3. Dichos conceptos están enmarcados en el nivel al que pertenecen. Así, por ejemplo, el concepto *Class* definido por MOF está en el nivel M3, y puede ser utilizado por el nivel M2 de UML para crear un metamodelo en el nivel M2. A partir de este metamodelo un usuario puede definir clases que satisfagan sus necesidades instanciadas en el nivel M1; en nuestro caso hemos creado un clase *Motor*, modelo creado a partir del metamodelo que define UML. Por último, en el nivel M0 tenemos el objeto *MotorCinta* como una instancia de la clase *Motor*.

5.2.2. EL PROCESO DE METAMODELADO

La tarea de crear un metamodelo que defina un lenguaje de modelado no es trivial. Cuanto más complejo sea el lenguaje a definir, más difícil y compleja será la definición del metamodelo. Sin embargo, existe un método sistemático que se puede seguir a la hora de construir metamodelos basado en un proceso bien definido e iterativo [125]. Dicho proceso se puede resumir en los siguientes pasos básicos:

- **Definición de la sintaxis abstracta.** La sintaxis abstracta de un lenguaje describe el vocabulario, los conceptos del lenguaje y como se pueden combinar para construir modelos. La sintaxis abstracta trata únicamente la forma y la estructura de los conceptos del lenguaje sin tener en cuenta ningún aspecto de presentación o significado.

- **Definición de reglas de formación y meta-operaciones.** La definición de reglas de formación y meta-operaciones pertenecen al marco de la sintaxis abstracta, ya que además de definir conceptos se especifican también las relaciones existentes entre dichos conceptos, así como las reglas que indican como se pueden combinar dichos conceptos.

- **Definición de sintaxis concreta.** Todos los lenguajes proporcionan una notación que facilita la construcción y el desarrollo de modelos o programas

en dichos lenguajes. Esta notación se denomina sintaxis concreta para un lenguaje determinado. Tradicionalmente existen dos tipos de sintaxis concreta: sintaxis basada en texto y sintaxis visual. La sintaxis basada en texto permite construir programas o modelos utilizando texto estructurado en base a la utilización de forma conjunta de declaraciones y expresiones, por lo que es posible definir expresiones complejas. En cambio, la sintaxis visual utiliza elementos gráficos para representar el modelo o el programa, por lo que es posible expresar de forma sencilla una gran cantidad de detalles de forma intuitiva. Para definir una sintaxis concreta con cierto nivel de complejidad se podrían utilizar las dos formas, textual y visual, aprovechando las ventajas de cada una de ellas, por lo que podríamos tener un lenguaje con sintaxis visual para representar las vistas superiores del modelo y la sintaxis textual para capturar propiedades a un mayor nivel de detalle.

- **Definición de la semántica.** La sintaxis abstracta contiene cierta parte (información parcial) sobre los conceptos definidos con un significado concreto dentro del lenguaje. Pero es necesario introducir información adicional para capturar los aspectos semánticos del lenguaje y hacer un uso correcto de las construcciones del lenguaje. Por ello es crítico capturar de forma precisa la parte semántica del lenguaje.

- **Construcción de trasformaciones a otros lenguajes de modelado.** Un lenguaje no está aislado y suele convivir con otros lenguajes, por lo que es necesario establecer relaciones entre ellos. Estas relaciones vienen dadas a través de la traducción (conceptos en un lenguaje se traducen a conceptos en otro lenguaje). De este modo podemos establecer equivalencias semánticas entre distintos lenguajes (un lenguaje puede tener conceptos cuyo significado se corresponden con conceptos en otro lenguajes) o entre diferentes niveles de abstracciones (un lenguaje puede estar relacionado con otro lenguaje que está en un nivel diferente de abstracción). Capturar estas relaciones es una parte importante de la definición de un lenguaje, ya que sirve para establecer el lenguaje en el contexto del mundo que lo rodea.

- **Extensibilidad.** La extensibilidad no es un paso en sí, sino una característica que tenemos que tener en cuenta a la hora de definir un metamodelo,

ya que los lenguajes no contienen entidades estáticas sino que cambian y evolucionan con el tiempo. Por ejemplo, se pueden agregar nuevos conceptos que permitan expresar los patrones comunes de los modelos o códigos de forma más sucinta, mientras que los elementos no utilizados del lenguaje acabarán por desaparecer. La capacidad de ampliar un lenguaje de una manera precisa y bien administrada es vital para poder apoyar la adaptabilidad. Esto permite que el lenguaje se adapte a nuevos dominios de aplicación y evolucione para satisfacer nuevos requisitos.

El desarrollo dirigido por modelos puede ser utilizado para guiar todo el proceso de desarrollo del software siguiendo una determinada metodología. A diferencia de otras metodologías de desarrollo de software como las basadas en UML (UP, Harmony, etc.), además de establecer el conjunto sistemático de actividades o tareas a realizar para obtener una implementación del sistema, en este caso hay que considerar una tarea previa que consiste en la creación de metamodelos como base del lenguaje que se va a utilizar en el proceso de desarrollo.

Sin embargo la construcción de metamodelos sigue siendo un tema bastante discutido, ya que en los últimos años se han creado varios tipos de lenguaje de modelado. Pero no existen procedimientos sistemáticos y completamente definidos que nos guíen durante el proceso de creación de un metamodelo, como en el caso de una receta de cocina, si no que más bien depende de los conocimientos y la experiencia del ingeniero o grupo de ingenieros que pretenden crear metamodelos. En este escenario, tener claro el dominio de aplicación y el objetivo para el cual se pretende crear el metamodelo, es clave para la aplicación y utilización del metamodelo, y por consiguiente para el éxito del mismo.

5.3. iMMAS: INDUSTRIAL META MODEL FOR AUTOMATION SYSTEMS

iMMAS proporciona un lenguaje de modelado específico para el desarrollo de sistemas software en entornos industriales. Para ello, iMMAS define un lenguaje con una sintaxis y una semántica concebida para trabajar con los dispositivos que suelen estar presentes en la gran mayoría de los sistemas automáticos de control industrial IAS (Industrial Automation System, en ingles).

Además proporciona un nivel de abstracción que facilita la programación de los dispositivos industriales, así como la gestión de sistemas de automatización que tienen que manejar una gran cantidad de señales. Podemos construir modelos bien definidos que nos servirán para representar, estructurar y definir los distintos elementos que conforman los sistemas software dentro del marco del control y la automatización industrial.

iMMAS establece el lenguaje a través de la definición de un metamodelo sobre el metalenguaje MOF de OMG[123], aunque también podría ser trasladable al metalenguaje ECORE[126]. El metamodelo establece, por tanto, la sintaxis abstracta de los conceptos del lenguaje y sus relaciones, así como las reglas que se establecen cuando un modelo está bien formado en el dominio de los sistemas industriales.

A partir de la sintaxis abstracta del metamodelo se ha definido la sintaxis concreta de un sistema industrial utilizando la misma notación gráfica de UML. Esta notación nos permitirá elaborar los modelos que reflejan el diseño concreto de un sistema industrial en base a una serie de elementos sintácticos. Dichos elementos sintácticos se corresponden con la instanciación de los conceptos que aparecen en el metamodelo, y por tanto, es posible verificar si están bien construidos comprobando que se cumplen tanto desde el punto de vista sintáctico como semántico.

La utilización de iMMAS a la hora de desarrollar un sistema software industrial ofrece una serie de ventajas. Algunas de estas ventajas derivan de las características propias de la utilización de modelos y la ingeniería dirigida por modelos como paradigma de desarrollo de software como, por ejemplo, la ya mencionada, capacidad de abstracción y la automatización a la hora de generar código para sistemas específicos de un entorno industrial. En el caso de esta última, iMMAS está construido con el objetivo de generar/transformar los modelos generados a

partir de iMMAS en código/representaciones que se puedan desplegar en sistemas industriales como servidores OPC-UA o incluso en dispositivos industriales como PLCs. Por ello iMMAS, no sólo ofrece un lenguaje de modelado, sino también un marco de desarrollo para dirigir el proceso de desarrollo de software sobre los principales elementos software de un sistema industrial, unificando criterios (mismo conceptos), facilitando la extensibilidad y la adaptabilidad del sistema (ampliar el modelo), documentando el sistema (parte de la definición del sistema se encuentra en el propio modelo) y facilitando el mantenimiento y despliegue del sistema software (todos los elementos software comparten los mismos conceptos). Por tanto, una vez creado un modelo del sistema industrial, es posible generar código del modelo que puede ser desplegado sobre los dispositivos industriales como los PLCs, o sobre sistemas industriales de apoyo como los sistemas OPC. La figura 5.6 muestra esquemáticamente como el diseño de un modelo sirve de punto de partida para la generación de aplicaciones que pueden ser desplegados en dispositivos y sistemas industriales.

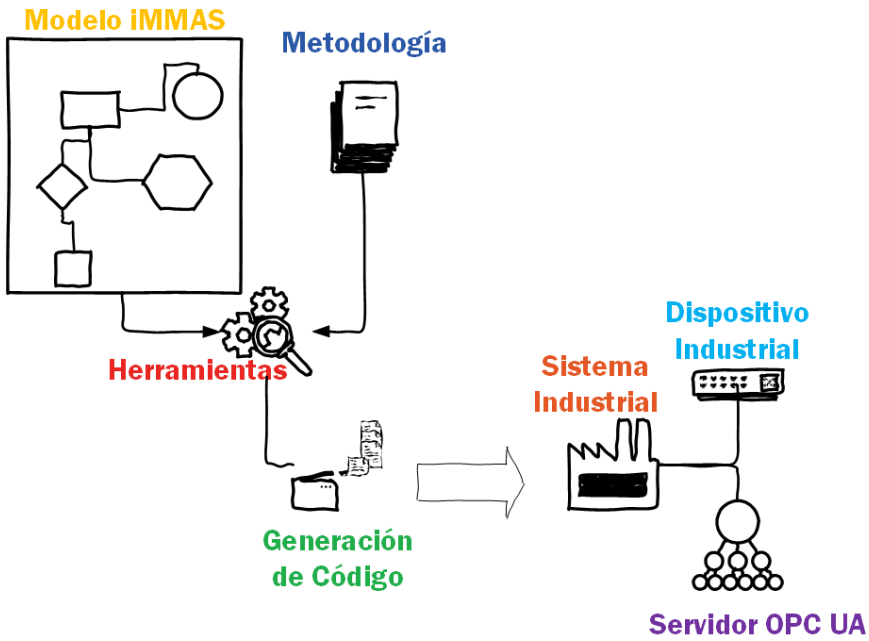


Figura 5.6: El metamodelo de iMMAS proporciona un lenguaje para diseñar Modelos que pueden desplegarse en servidores OPC UA y en dispositivos como PLCs.

5.4. FUNDAMENTOS DE iMMAS

iMMAS se ha concebido con el propósito de reducir el salto conceptual que hay entre la definición de un sistema industrial y su implementación. Aunque para el desarrollo de sistemas industriales los ingenieros siguen una serie de etapas en base a una guía metodológica, en muchos casos descrita formalmente y estandarizada para un dominio concreto de aplicaciones, suelen aparecer problemas entre la descripción del sistema desarrollado y la implementación, porque no manejan un mismo vocabulario o dicho vocabulario no es consistente. Por ejemplo, en la implementación de un componente del sistema se suele determinar el conjunto de señales que lo caracterizan, llevando en un listado de señales la identificación de cada señal, su comportamiento general, y la utilización que se hace en dicho sistema. Esto tiene fuertes implicaciones en el desarrollo del sistema que perjudica entre otros el mantenimiento y evolución de los sistemas industriales.

El enfoque que se ha seguido en esta tesis ha consistido en tratar de buscar la forma más natural de identificar los distintos elementos del sistema industrial, sus propiedades y relaciones, utilizando el mismo vocabulario que manejan los desarrolladores e ingenieros de sistemas. Pero además, se ha estudiado las consecuencias e impacto que tiene dicho enfoque sobre el diseño y la implementación de los distintos elementos del sistema, con objeto de encontrar una implementación correcta "por construcción" tanto para los dispositivos industriales, como para los sistemas de apoyo en distintos niveles de jerarquía del sistema industrial.

En este contexto, iMMAS proporciona por tanto un nuevo lenguaje de modelado específico para trabajar con sistemas industriales constituido por elementos sintácticos (construcciones y bloques) con una semántica bien definida para el dominio de sistemas industriales, y cuya sintaxis abstracta se ha representado mediante metamodelos. Cada uno de los elementos sintácticos que aparecen en el metamodelo es independiente de la representación utilizada para representarlos. Además se incluyen en el metamodelo una serie de propiedades para describir cada elemento sintáctico, las reglas de formación y operaciones que definen cada elemento, las relaciones que tienen que establecerse entre cada elemento, y las restricciones que deben satisfacerse tanto a nivel conceptual como a nivel global de sistema.

Para la descripción de todos los elementos sintácticos del metamodelo así co-

mo sus relaciones se ha utilizado al metalenguaje MOF, metalenguaje que se utiliza también como base para la definición del lenguaje de modelado UML[123]. Esto significa que el lenguaje de modelado iMMAS y UML se encuentra en el mismo nivel de abstracción. Por otra parte, la expresión de todas las restricciones que definen la validez de los modelos se representan en el lenguaje OCL (Object Constraint Language) [127].

Por otro lado, el lenguaje de modelado descrito mediante el metamodelo define cómo pueden describirse los modelos basados en iMMAS con el que podemos especificar sistemas industriales. A partir de dichos modelos podremos implementar programas que se ejecutaran, por ejemplo, sobre dispositivos industriales.

La relación que hay entre la implementación, el modelo, el metamodelo y el meta-metamodelo viene descrita por una arquitectura de cuatro niveles que se indica en la figura 5.7. Mediante dicha arquitectura se establece un marco de trabajo para el desarrollo de software de acuerdo al paradigma MDA (Model-Driven Architecture) de OMG [123]. Como se puede observar en el nivel M3 se describe el meta-lenguaje de modelado (MOF en nuestro caso) que viene descrito mediante un meta-metamodelo utilizando los propios elementos del lenguaje gracias a la definición circular de dichos conceptos. A partir del meta-metamodelo se define por instanciación, el metamodelo de iMMAS en el nivel M2 que incluye una descripción del lenguaje de modelado utilizado en este trabajo. A partir del nivel M2 podemos definir los modelos que especifican sistemas industriales por instanciación del metamodelo; todos estos modelos se corresponden con el nivel M1. Por último la descripción de los modelos de sistemas industriales puede ser implementada en el nivel M0 con un sistema de representación ligada a la plataforma concreta donde se implementa. En nuestro caso hemos trabajado con la posibilidad de definir modelos de implementación directamente sobre PLCs.

En la tabla 5.1 se realiza una comparación del contenido de cada uno de los niveles entre iMMAS y UML.

Para facilitar la construcción de modelos en iMMAS se ha desarrollado además una metodología que incluye las etapas que habría que seguir para diseñar el sistema y consolidarlo después de haber verificado su conformidad con el metamodelo. Para ello se especifican los métodos que hay que seguir, los principios

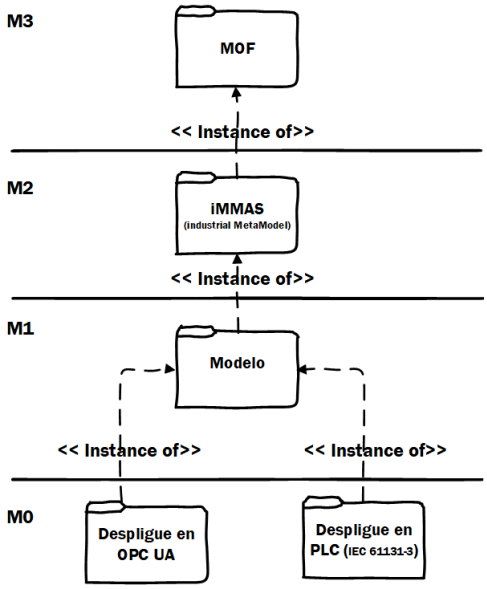


Figura 5.7: Representación esquemática de la arquitectura de cuatro niveles definida para iMMAS.

Niveles	UML	iMMAS
M3	MOF	MOF
M2	Lenguaje UML (metamodelos de diagrama de representación)	Metamodelo de iMMAS
M1	Modelos utilizando diagramas de representación	Perfiles de MOSIE
M0	Objetos, Instanciación	Implementación

Tabla 5.1: Tabla en la que se compara iMMAS/MOSIE con respecto a UML.

que hay que aplicar, y las heurísticas que hay que tener en cuenta para diseñar un modelo en IMMAS correcto. En el capítulo 8, se explica de forma detallada como llevarlo a cabo.

Por último indicar que el lenguaje definido por iMMAS es extensible, ya que admite ampliaciones o redefiniciones de los conceptos que alberga, pudiendo así añadir conceptos nuevos que podrán ser utilizados para crear modelos a partir de ellos. En nuestro caso hemos definido un perfil sobre el lenguaje de modelado, aprovechando la capacidad de extensibilidad del lenguaje, que incluye todos los conceptos que pueden ser reutilizables en modelos de sistemas industriales para simplificar el proceso de desarrollo.

5.5. LENGUAJE DE MODELADO IMMAS

El lenguaje de modelado iMMAS forma parte del nivel M2 en la arquitectura de cuatro niveles del paradigma MDE especificado anteriormente [109] como un metamodelo que utiliza el metalenguaje MOF. El metamodelo incluye el conjunto de abstracciones que define la sintaxis del modelo con el que se pueden construir modelos de sistemas industriales.

Durante el proceso de definición del lenguaje se evaluaron distintas alternativas. La más utilizada en el campo de DSL (Domain Specification Language) consiste en definir en primer lugar la sintaxis abstracta del lenguaje utilizando una descripción gráfica o textual con un lenguaje de metamodelado. Existen varias opciones como lenguajes de metamodelado, aunque suelen utilizarse el metalenguaje MOF estandarizado por OMG [123] y el metalenguaje ECORE[126] definido dentro de la infraestructura Eclipse [126]. Una vez creada la sintaxis abstracta se determina la semántica del lenguaje, sus restricciones y la sintaxis concreta para luego poder describir modelos sobre dicho lenguaje. Esta alternativa presenta dos ventajas importantes:

- Definición del lenguaje orientada específicamente al dominio de aplicaciones donde se va a aplicar.
- Descripción del lenguaje más sencilla, aunque debe comprobarse que el lenguaje no presente ambigüedades, sea suficientemente flexible y completo, incluyendo todos los elementos sintácticos para la definición de modelos.

Otra alternativa para la definición del lenguaje basada en las tecnologías de OMG consiste en utilizar UML como base para la definición del lenguaje. Como el lenguaje UML está sustentado por un conjunto de metamodelos descrito en el lenguaje MOF, en la definición del nuevo lenguaje es necesario ampliar o restringir la capacidad semántica de UML, así como su sintaxis abstracta y concreta. La ampliación o restricción puede realizarse sobre el lenguaje UML directamente modificando directamente los metamodelos que definen UML, y luego ir verificando además todas las restricciones definidas. Sin embargo, esto no suele ser sencillo, y cualquier modificación puede originar problemas de coherencia o inconsistencias.

La otra opción es aprovechar la capacidad de extensión de UML y definir un nuevo perfil. En este enfoque, un perfil establece un subconjunto de UML en base a definir nuevos estereotipos sobre los elementos sintácticos de UML, definir nuevos valores etiquetados, y establecer nuevas restricciones que modifiquen el comportamiento estándar de UML. Esta aproximación aprovecha completamente la capacidad de modelado de UML facilitando la creación de nuevos tipos de diagramas de representación o de nuevos tipos de construcciones basadas en elementos sintácticos de UML. Esta aproximación ha sido seguida por SysUML (Systems Modeling Language) o MARTE (model-driven development of Real Time and Embedded Systems), que son lenguajes de notación específicos para la programación de sistemas y de sistemas empotrados y de tiempo real respectivamente [128] [129]. Las ventajas y desventajas de esta aproximación son:

- Tomar como base UML permite trabajar sobre un conjunto de elementos sintácticos muy consolidados en la industria.
- Puede ser complicado cerrar la especificación completa del lenguaje que requiere tener un conocimiento profundo de UML para evitar inconsistencias.

iMMAS utiliza una aproximación hibrida entre las dos aproximaciones anteriores. Por un lado define un nuevo lenguaje de modelado como un metamodelo escrito en MOF, con el que se define el lenguaje base para la descripción de sistemas industriales. Dicho lenguaje se define tomando conceptos y abstracciones de UML y MOF, por lo que está influido en cierta manera por UML. Y por otro lado define también un perfil sobre el propio lenguaje de iMMAS, para simplificar la construcción de sistemas industriales al establecer conceptos y abstracciones basadas en el propio lenguaje de iMMAS utilizando su capacidad de extensión. De este modo podemos definir plantillas que facilitan la construcción de elementos concretos sobre iMMAS, adaptados al vocabulario empleado por los ingenieros que trabajan con sistemas industriales.

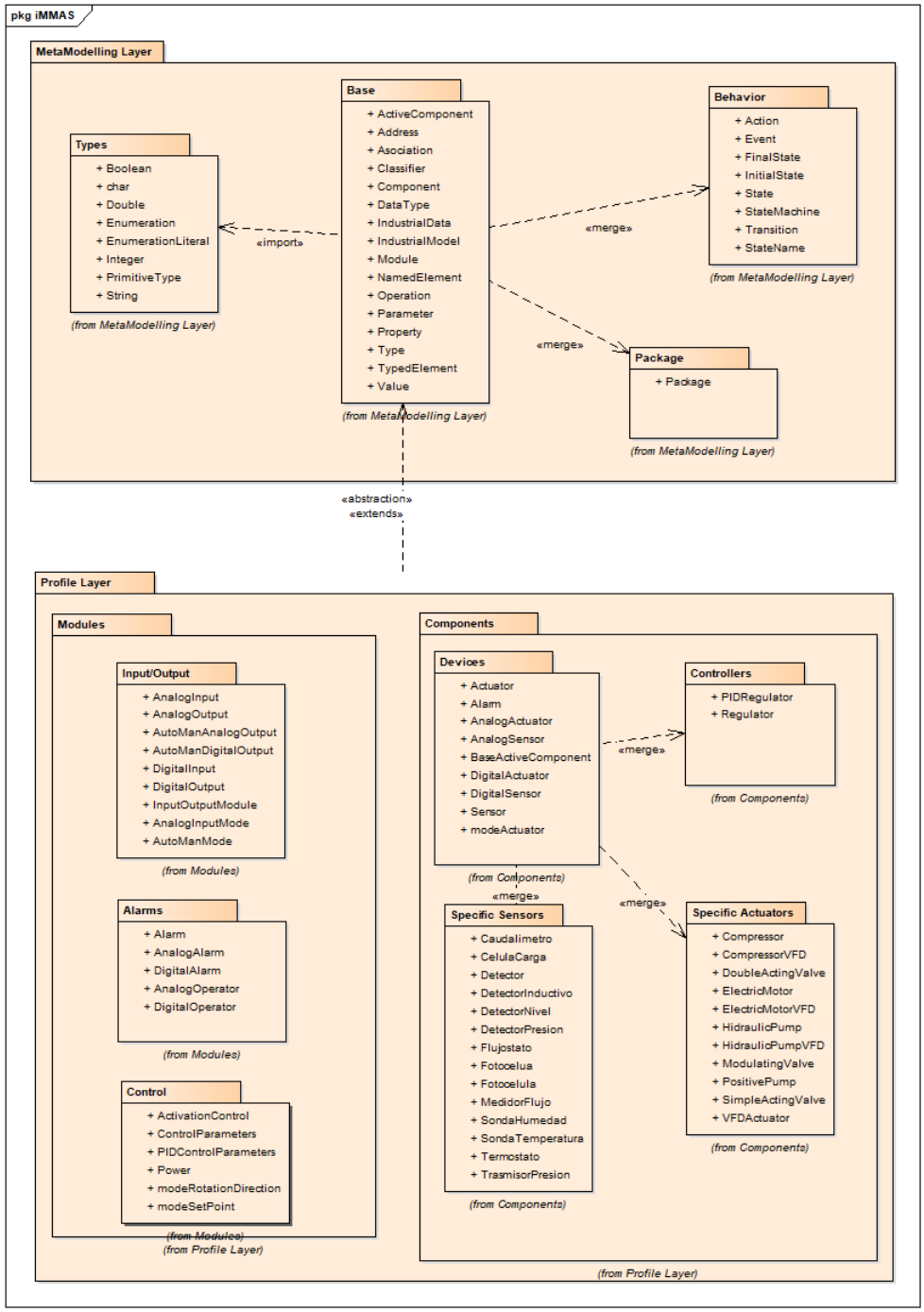


Figura 5.8: El lenguaje de modelado está sustentado por dos partes: el lenguaje de modelado base y el perfil de sistemas industriales.

En la figura 5.8 pueden verse las dos partes fundamentales del lenguaje iMMAS: el lenguaje de modelado base y el perfil de sistemas industriales. El lenguaje de modelado base incluye todos los elementos sintácticos necesarios que nos permiten definir el lenguaje y por tanto su sintaxis abstracta. Está constituido por cuatro paquetes: *types, base, packages, behavior*. Con dicho lenguaje ya se puede diseñar modelos que representan a sistemas industriales. Sin embargo, el apoyo del perfil de sistemas industriales ayuda a definir conceptos específicos de los sistemas industriales, incluidas sus propiedades y sus relaciones. De este modo, la utilización de conceptos del perfil simplifica aún más los modelos construidos sobre iMMAS.

5.5.1. LENGUAJE DE MODELADO BASE

El lenguaje de modelado base contiene todos los elementos sintácticos del lenguaje que requieren los sistemas industriales para describir dichos sistemas, incluyendo su sintaxis abstracta y concreta, la semántica asociada a cada elemento sintáctico, así como las interrelaciones y restricciones. Los metamodelos especificados se basan en conceptos de UML.

- Types: Recoge los tipos de datos utilizados por iMMAS.
- Base: Recoge los conceptos primitivos que caracterizan cualquier modelo que se describa en iMMAS.
- Package: Recoge los conceptos relacionados con las agrupaciones o paquetes.
- Behavior: Recoge los conceptos que permiten realizar modelos de comportamiento en iMMAS.

El paquete Base recoge los conceptos primitivos fundamentales sobre el que se asienta el lenguaje. Los paquetes, *Types* y *Packages* complementan al paquete *Base*, mientras que el paquete Behavior proporciona comportamiento a algunos de los conceptos primitivos incluidos en *Base*. Por esta razón todos los paquetes tienen una relación de dependencia de tipo ”merge” con los conceptos del paquete *Base*. A continuación se va a explicar el significado conceptual de cada uno de estos paquetes, enumerando cada uno de los elementos que lo componente.

EL PAQUETE BASE

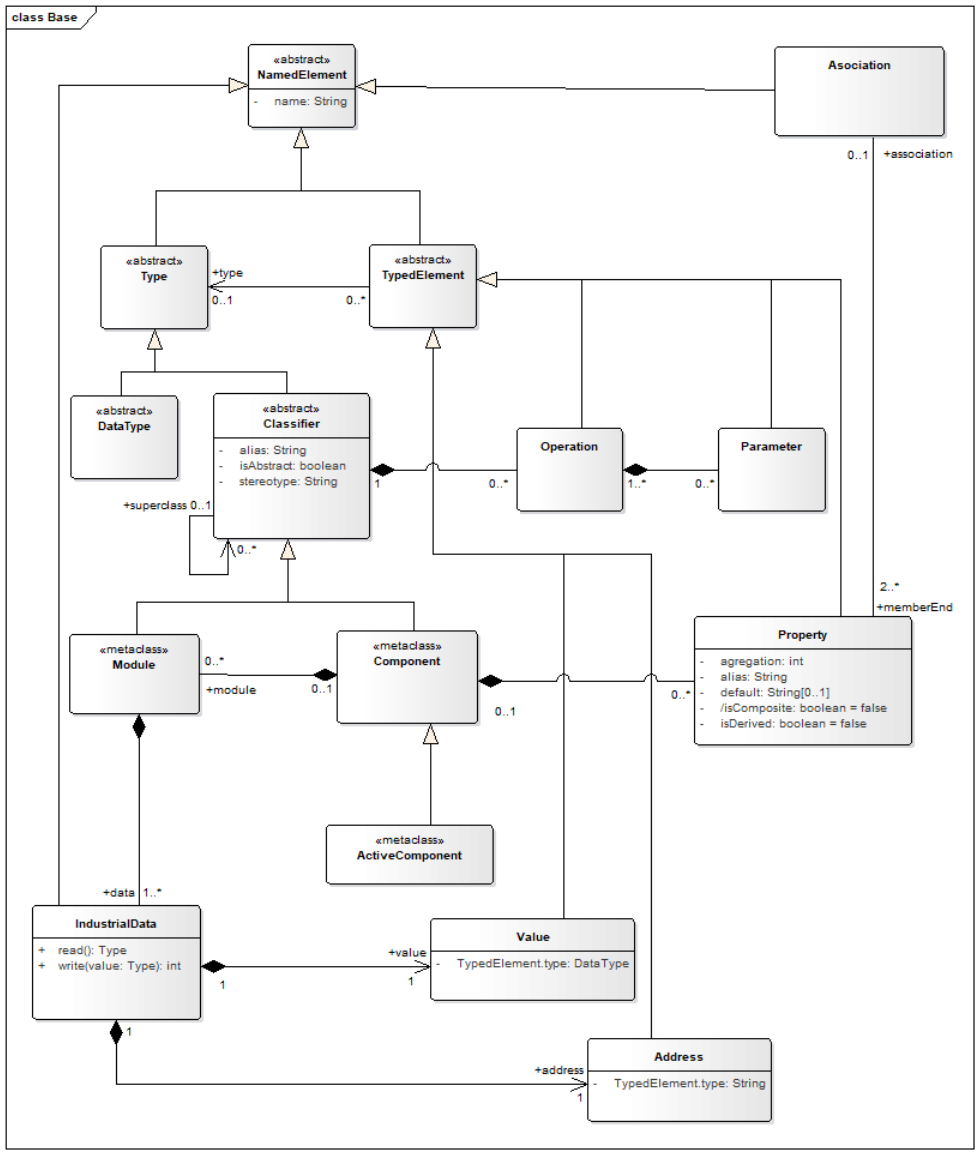


Figura 5.9: Elementos sintácticos y relaciones del paquete Base de iMMAS.

El módulo es una entidad del lenguaje que hace referencia al contenedor de datos industriales. Es el elemento sintáctico que nos permite agrupar datos industriales que deben estar relacionados entre sí. Los módulos son independientes y no pueden compartirse, por lo que es importante incluir todos aquellos da-

tos industriales para que el modulo sea lo más cohesivo posible. El módulo actúa como una estructura de datos, pero puede incluir operaciones para acceder o extraer la información encapsulada en datos industriales.

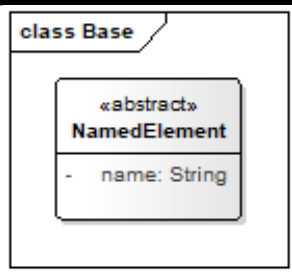
El componente por su parte hace referencia a las entidades individuales que puede haber en un sistema industrial. El componente es un contenedor de módulos por lo que incluye de este modo todos los datos persistentes que pueden definir al componente. Sin embargo, eso solo lo puede hacer a través de los módulos que son los encargados de llevar a cabo la gestión de los datos industriales. Un componente puede contener otros componentes más sencillos o estar relacionado con otros componentes del sistema industrial. Por último, también permite definir propiedades adicionales como atributos del componente, aunque no tendrán la propiedad de persistencia de los datos industriales.

Como ejemplo podríamos modelar en el nuevo lenguaje un actuador como un componente con un módulo para almacenar como dato industrial la señal con la que podemos actuar sobre el entorno.

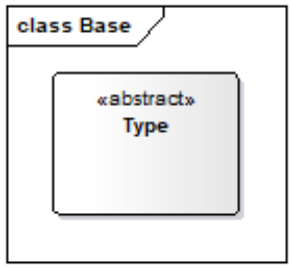
De forma general, el metamodelo viene definido por el conjunto de elementos sintácticos relacionados como se muestra en la figura 5.9.

A continuación se especifican las metaclases definidas en el paquete:

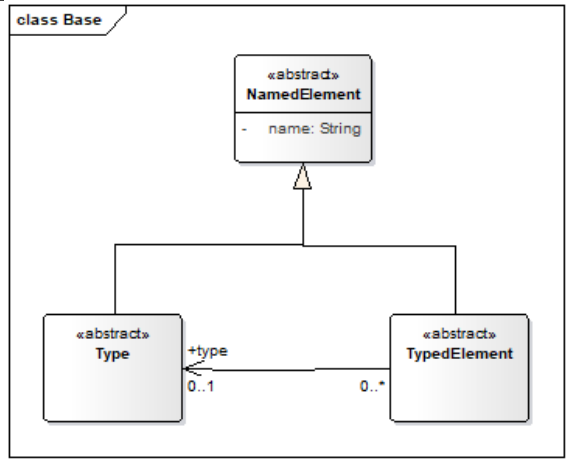
- **NameElement.**
  Contenido Semántico: El elemento nombrado es un concepto abstracto incluido habitualmente en los metamodelos para especificar que cada concepto o elemento del metamodelo debe tener un nombre determinado.
  Sintaxis Abstracta: Es una metaclase que viene definido únicamente por el atributo nombre (*name*) de tipo *String*. Es una clase abstracta.

Sintaxis Abstracta	Sintaxis Concreta
	No tiene sintaxis concreta, porque es una clase abstracta.

- **Type.**
  Contenido Semántico: El concepto tipo refleja las entidades que son susceptibles de ser etiquetadas con un tipo.
  Sintaxis Abstracta:: Es una metaclase que hereda de la metaclase abstracta *NamedElement*, y va a ser referenciada por las demás clases que necesitan definir un tipo.

Sintaxis Abstracta	Sintaxis Concreta
	No tiene sintaxis concreta, porque es una clase abstracta.

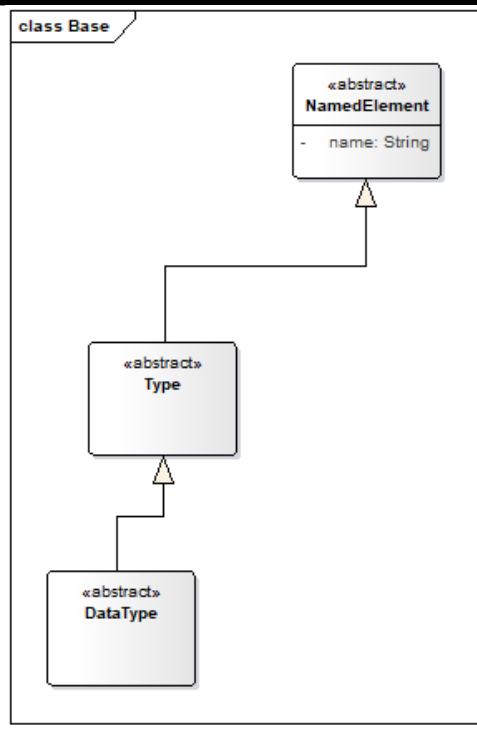
- **TypeElement.**
  Contenido Semántico: El concepto Elemento Tipado hace referencia a cualquier concepto o entidad que debe poseer un tipo.

Sintaxis Abstracta

Sintaxis Concreta
No tiene sintaxis concreta, porque es una clase abstracta.

Sintaxis Abstracta: Tiene las siguientes propiedades:

- Es una metaclase que hereda de la metaclase abstracta *NamedElement*.
- Tiene un metatributo heredado de *NamedElement* denominado *Name* de tipo *string*.
- Tiene un meta-tributo *type* que hace referencia a una instancia de la metaclase *Type*.

- **DataType.**
  Contenido Semántico: Hace referencia a los tipos de datos básicos del lenguaje.

Sintaxis Abstracta

Sintaxis Concreta
No tiene sintaxis concreta, porque es una clase abstracta.

Sintaxis Abstracta: Tiene las siguientes propiedades:

- Es una metaclase que hereda de la metaclase abstracta *Type*, y a su vez de la metaclase *NamedElement*.
- Tiene un metatributo heredado de *NamedElement* denominado *Name* de tipo *string*.

- **Classifier.**
  Contenido Semántico: El concepto Clasificador representa a los elementos principales en los que se estructura un modelo conforme al metamodelo. En nuestro caso, vendrá determinado por la posibilidad de agregar un alias (nombre alternativo más corto para hacer referencia a un objeto de tipo clasificador) y un estereotipo. El estereotipo hace referencia a un atributo que puede incluirse en los objetos de tipo clasificador para especificar una categoría especial en el momento de modelarlos. Esta última propiedad será importante, ya que nos permite extender el lenguaje y definir el perfil de sistemas industriales. Es una clase abstracta. En un Clasificador se puede aplicar la relación de herencia, permitiendo crear especializaciones de clasificadores. Un clasificador además puede contener también operaciones que manejan elementos del propio clasificador. Las operaciones pueden contener uno o más parámetros que deben tener un tipo especificado.

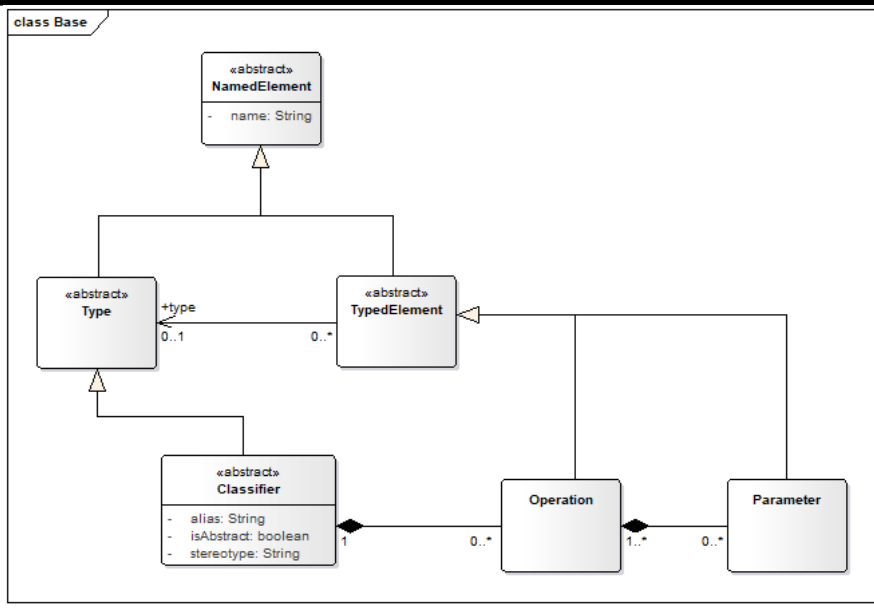
  Sintaxis Abstracta: La metaclase Clasificador tiene las siguientes propiedades:

  - Es una especialización de la metaclase *Type*, que a su vez hereda de la metaclase NamedElement, y por lo tanto hereda el metatributo name de tipo String.
  - La metaclase *Classifier* tiene como metaatributo la cadena alias y estereotipo. El estereotipo permite caracterizar taxonómicamente los clasificadores en base a una o varias categorías. El desarrollador puede utilizarlo para distinguir el concepto clasificador según su uso, su funcionalidad o cualquier otro criterio.
  - La metaclase *Classifier* está relacionada con la metaclase *Operation*, por lo que puede tener una o varias operaciones asociadas al clasificador. Dichas operaciones pueden tener uno o varios parámetros a

través de la metaclase *Parameter*.

– Un clasificador de la metaclase *Classifier* puede tener varios clasificadores de la metaclase *Classifier*.

Sintaxis Abstracta



Sintaxis Concreta

No tiene sintaxis concreta, porque es una clase abstracta.

- **Operation.**
  Contenido Semántico: Hace referencia a operaciones que pueden tener clasificadores como los Componentes y los Módulos, es decir, encapsula a una funcionalidad que puede resolverse mediante la ejecución de un algoritmo. Las operaciones requieren una serie de argumentos (parámetros), que permiten modificar la ejecución de una operación, y se obtiene un resultado que tendrá que tener definido un tipo.

  Sintaxis Abstracta: La metaclase Operation define las funcionalidades que puede tener un clasificador. Tiene las siguientes propiedades:

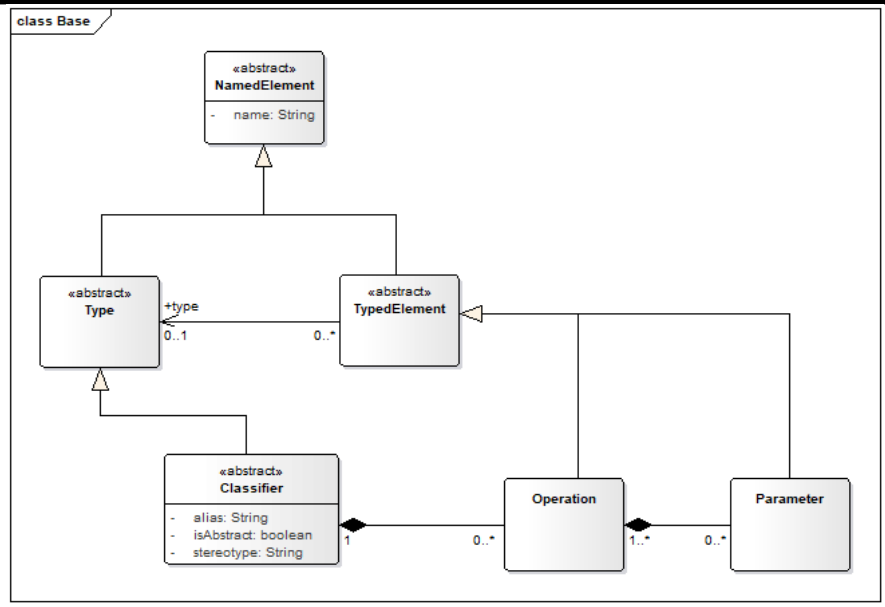
  – La metaclase *Operation* hereda de la metaclase *TypedElement*, y esta a

su vez de *NamedElement*.

– Las instancias de la metaclase *Operation* tienen un meta-atributo *type* de la metaclase *Type*, que hace referencia al tipo que devuelve el retorno de la operación.

– Cada instancia de la metaclase *Operation* puede tener asignadas una o más instancias de la metaclase *Parameter*, de modo que una operación puede tener uno o más parámetros. Cada parámetro al ser un objeto que hereda de la metaclase *TypedElement* tiene que tener definido un tipo dado por *type*.

Sintaxis Abstracta



Sintaxis Concreta

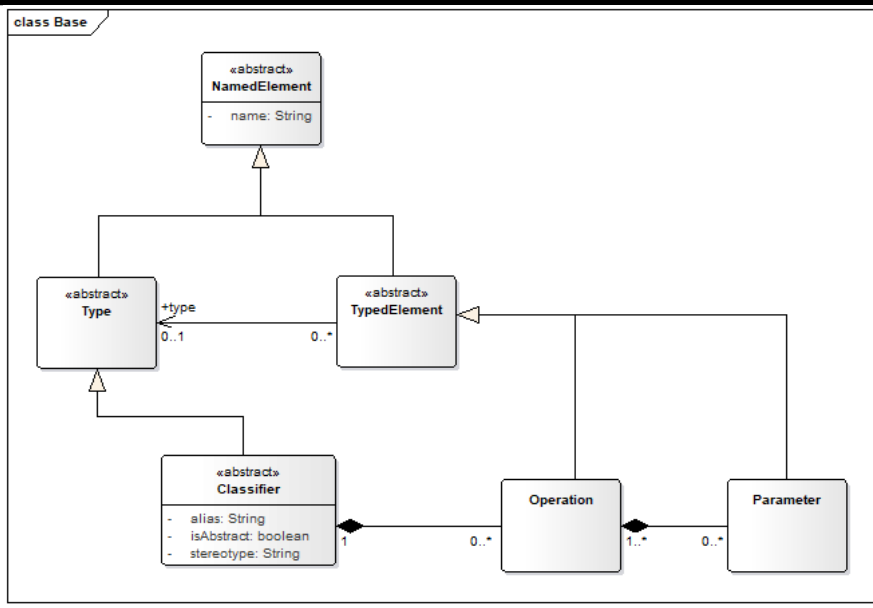
Las instancias de la metaclase *Operation* se modelan con los elementos típicos de una operación, es decir, nombre de la operación y entre paréntesis (el conjunto de parámetros que tiene que tener definido un tipo) seguido de dos puntos con el tipo de dato que se devuelve.

- **Parameter.**

  Contenido Semántico: Hace referencia a los argumentos que puede tener una operación definida en un modelo.

Sintaxis Abstracta: La instancia de la metaclase *Parameter* está relacionada con la instancia de la metaclase *Operation*, y tiene que tener definido un tipo *Type* por heredar de la metaclase *Type*.

Sintaxis Abstracta



Sintaxis Concreta

La instancia de la metaclase *Parameter* se visualizará en los modelos como un argumento de una operación, que tiene que tener definido un tipo para lo cual se utiliza el token de separación ":".

- **Component.**

  Contenido Semántico: El concepto *Component* es uno de elementos más importantes de iMMAS y hace referencia a elementos industriales independientes y autónomos que pueden contener a otros elementos más básicos. Puede representar tanto elementos físicos como motores, válvulas, dispositivos como elementos que regulan o controlan el el proceso o partes del proceso industrial.

  Un componente puede:

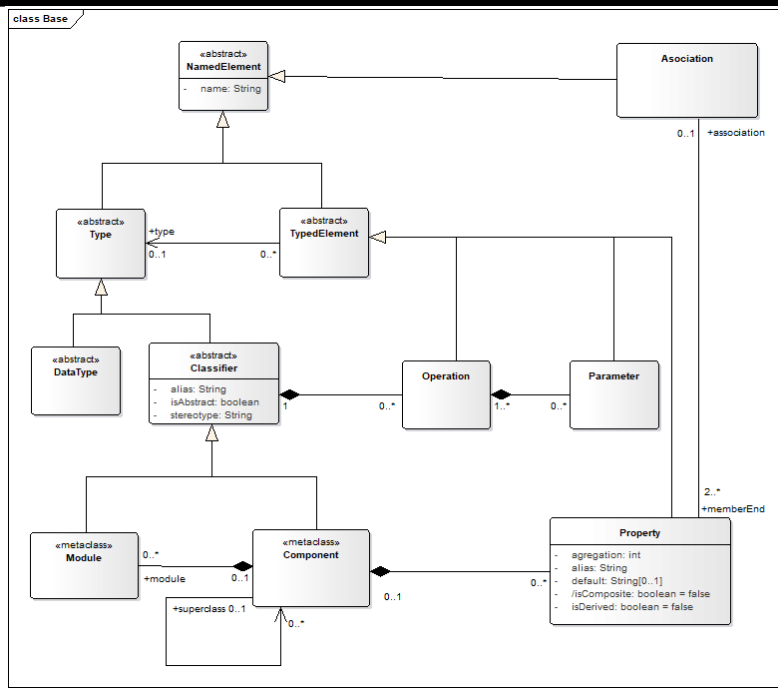
– Tener módulos que permiten caracterizar las propiedades del componente que se almacenan persistentemente.

– Tener a otros componentes asociados con relaciones de asociación (igual-a-igual), agregación (todo-parte compartido) o composición (todo-parte estricto). Para ello, hay que definir un calificador para conocer el rol con el que se relaciona el componente dependiente con dicho componente.

– Tener atributos que caracterizan al componente.

– Tener operaciones que definen una funcionalidad, para lo cual puede tener uno o varios parámetros que tienen que ser de un tipo determinado previamente definido.

Sintaxis Abstracta



Sintaxis Concreta

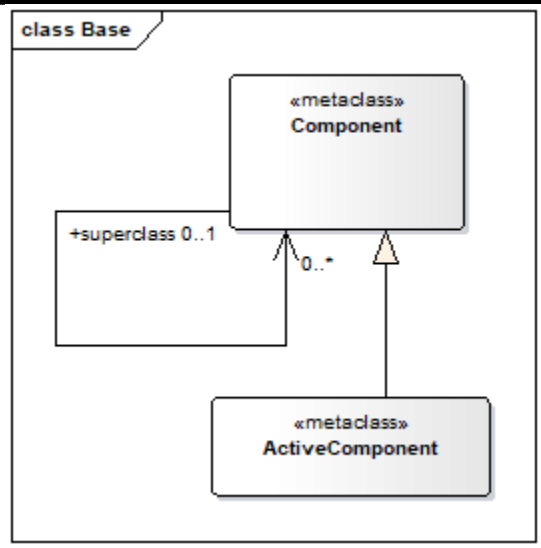
Se representa por una clase (estilo UML) que puede contener como atributos persistentes (metaclase *Module*) y atributos (metaclase *Properties*), asociaciones con un calificador (igual-a-igual, todo-parte compartida, todo-parte estricto) y operaciones.

Sintaxis Abstracta: La metaclase *Component* tiene las siguientes propiedades:

- La metaclase *Component* hereda de la metaclase *Classifier*, esta a su vez de la metaclase *Type*, y ésta de la metaclase *NamedElement*.
- Un componente de la metaclase *Component* tiene como meta-atributo la cadena nombre, alias y estereotipo. El estereotipo permite caracterizar taxonómicamente los componentes en base a una o varias categorías. El desarrollador puede utilizarlo para distinguir el concepto clasificador según su uso, su funcionalidad o cualquier otro criterio
- Un componente de la metaclase *Component* puede tener varios subcomponentes de la metaclase *Component*.
- Un componente de la metaclase *Component* al estar relacionada con las instancias de la metaclase *Properties*, puede tener uno o más atributos.
- Un componente puede estar asociado con uno o varios componente a través de un calificador definido por la metaclase *Properties* y la metaclase *Asociation*, que indica que la asociación puede ser de asociación (igual a igual), agregación (todo-parte compartido) o composición (todo-parte).
- Un componente de la metaclase *Component* puede estar relacionada con la metaclase *Operation*, por lo que puede tener una o varias operaciones asociadas al componente. Dichas operaciones pueden tener uno o varios parámetros a través de la metaclase *Parameter*. Tanto la operación como los parámetros tiene que tener definido un tipo *type* de la metaclase *Type*.

- **ActiveComponent.**
  Contenido Semántico: Un componente activo es un tipo de componente que tiene definido un comportamiento. Esto significa que tiene definido una máquina de estados de acuerdo al paquete *Behavior*.

Sintaxis Abstracta



Sintaxis Concreta

El componente activo se mostrará en los modelos como una clase de igual forma que un componente.

Sintaxis Abstracta: La metaclase *ActiveComponent* es una especialización de la metaclase *Component*.

- **Property.**
  Contenido Semántico: Un calificador es un atributo (o tupla de atributos) de la asociación cuyos valores sirven para particionar el conjunto de objetos enlazados a otro.

  Sintaxis Abstracta: La metaclase *Property* tiene las siguientes propiedades:

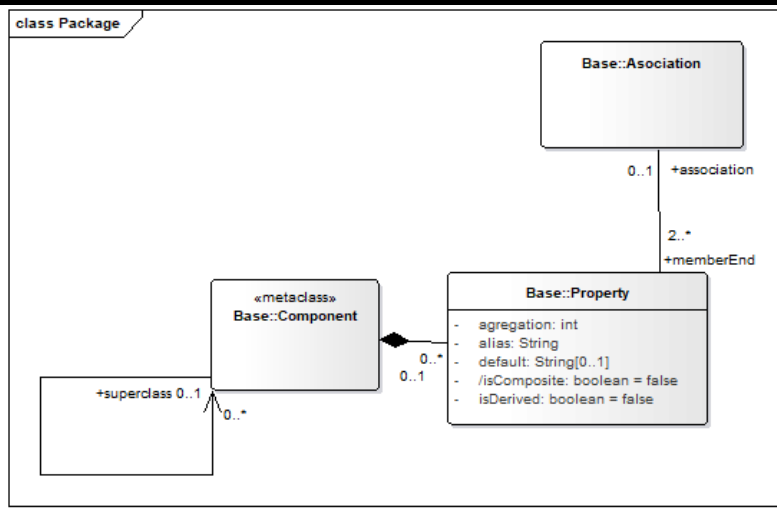
  – La metaclase *Property* hereda de la metaclase *TypedElement* por lo que tiene que tener definido un tipo *type* de la metaclase *Type*.

– La instancia de la metaclase *Property* tiene que tener definido el nombre, *isDerived*, *alias*, *isComposited*, *agregation*.

– Si la instancia de la metaclase *Property* no tiene instancia de la metaclase *Association,* entonces la instancia hace referencia a un atributo de la metaclase *Component*.

– Si la instancia de la metaclase *Property* tiene instancia de la metaclase *Association*, entonces *Property* representa el calificador de una asociación.

Sintaxis Abstracta



Sintaxis Concreta

Aparece como calificador en la asociación que puede representarse como un rombo negro (composición), rombo blanco (agregación), sin rombo (asociación) y un nombre opcional que refleja el calificador. Aparece como atributo dentro de la clase de tipo *Component*, especificando un tipo.

- **Asociation.**

  Contenido Semántico: La Asociación hace referencia a la relación de conexión que se establece entre componentes.

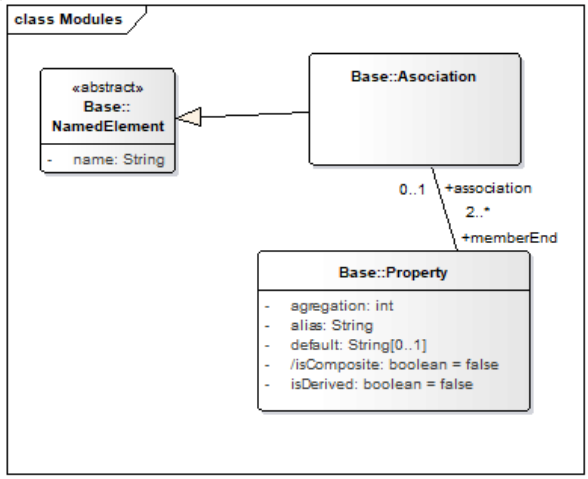
  Sintaxis Abstracta: La metaclase *Asociation* tiene las siguientes características:

– La metaclase *Asociation* hereda de la metaclase *NamedElement*.

– La metaclase *Asociation* tiene definida un nombre.

– La instancia de la metaclase *Asociation* se asocia a una o varias propiedades que se corresponde a los calificadores de las instancias componentes de la metaclase *Component* a los que se conecta.

Sintaxis Abstracta



Sintaxis Concreta

Se representa con una línea que une a dos componentes colocando en cada terminador un calificador con o sin rombo.

- **Module.**
  Contenido Semántico: El modulo es otro elemento importante para el modelado de sistemas industriales, ya que permite agrupar datos industriales. Representa a elemento software cohesivo autónomo que forma parte de un componente. Un módulo puede tener:

  – Uno o varios datos industriales que hacen referencia a unidades de memoria que contienen una dirección física y un dato. Los datos tienen que tener definido un tipo de dato.

  – Tener operaciones que definen una funcionalidad para lo cual puede

tener uno o varios parámetros que tienen que ser de un tipo determinado, previamente definido.

Restricciones:

- Un módulo no puede relacionarse con una relación de asociación con otro módulo.
- Los módulos son independientes y no pueden compartir su contenido.

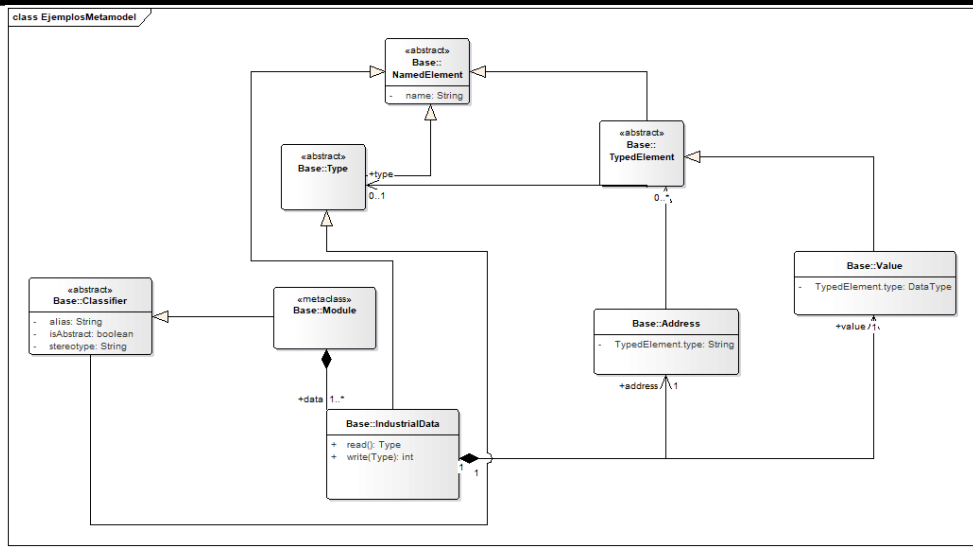
Sintaxis Abstracta: La metaclase Module tiene las siguientes propiedades:

- La metaclase *Module* hereda de la metaclase *Classifier*, ésta a su vez de *Type*, y ésta por último de *NamedElement*.
- Un módulo de la metaclase *Module* tiene un nombre, un alias y un estereotipo de tipo *String*.
- Un módulo de la metaclase *Module* está relacionada con la metaclase *IndustrialData*, por lo que un módulo puede tener una o más instancias de la metaclase *IndustrialData*. Esto es lo que denominamos un atributo persistente de módulo.
- El dato industrial se compone de un valor de la metaclase *Value*, y una dirección correspondiente a la metaclase *Address*. La instancia de la metaclase *Value* tiene que tener definido un tipo, al ser especialización de la metaclase *TypedElement*.
- Un módulo de la metaclase Module puede estar relacionada con la metaclase *Operation*, por lo que puede tener una o varias operaciones asociadas al modulo. Dichas operaciones pueden tener uno o varios parámetros a través de la metaclase *Parameter*. Tanto la operación como los parámetros tiene que tener definido un tipo *type* de la metaclase *Type*.

  Restricciones:

- Solo los módulos pueden definir atributos persistentes.

Sintaxis Abstracta



Sintaxis Concreta

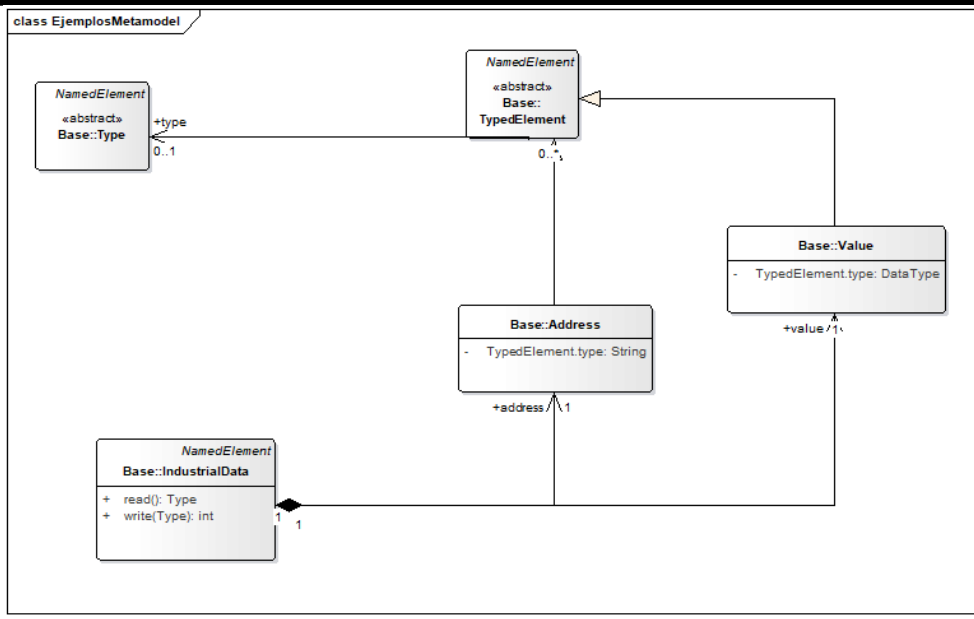
La instancia de la metaclase *Module* se modela con una clase tipo UML donde como atributos tenemos una instancia o más instancias de la metaclase *IndustrialData*, y como operaciones instancias de la metaclase *Operation*, que puede tener como argumentos instancias de la metaclase *Parameter*.

- **IndustrialData.**

  Contenido Semántico: Esta abstracción encapsula el dato persistente que se maneja en un entorno industrial. Para ello se requiere no sólo especificar un valor que tiene que tener definido un tipo, sino que además debe especificar la dirección de memoria que tiene dicho valor. Este modo de almacenar los datos industriales está influenciado por la forma en el que se almacenan los datos sobre dispositivos industriales como PLC, o en sistemas software SCADA u OPC UA.

  Sintaxis Abstracta: Este elemento sintáctico requiere de dos propiedades para su definición, la dirección donde se encuentra el dato y el valor del dato asignado a dicha dirección. Además de estas propiedades posee dos métodos, uno para la lectura y otro para la escritura de la propiedad donde se encuentra almacenado el valor de la dirección de memoria del PLC.

Sintaxis Abstracta



Sintaxis Concreta

Aparece el dato industrial como un atributo de la clase de tipo *Modulo* correspondiente, pero debe describirse como sigue: *nombreDatoIndustrial : (string,DataType).*

Restricciones

La lectura de un industrialData nos da el valor encapsulado en value.

Context IndustrialData::read():Type

post: result == self.value

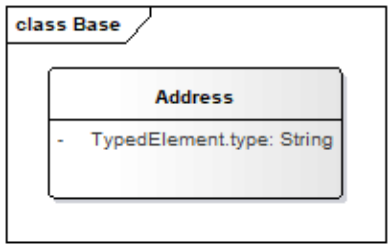
La escritura en un IndustrialData tiene que tener el mismo tipo que el valor proporcionado como parámetro y el valor de value tiene que ser igual que el valor aportado. context IndustrialData::write(value:Type):int

post self.value == value and self.value.OclIsType(Type)

- **Address.**

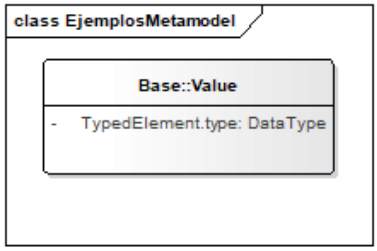
  Contenido Semántico: La metaclase Address hace referencia a la dirección de memoria que debe tener un industrialdata donde se almacena el IndustrialData cuyo valor viene dado por la metaclase *value. Sintaxis Abstracta:*

La metaclase Address forma parte de un *IndustrialData*.

Sintaxis Abstracta	Sintaxis Concreta
	Aparece como un atributo de *IndustrialData*.

- **Value.**

Contenido Semántico:: El concepto *value* hace referencia al valor que tiene un *industrialdata* como dato persistente del modelo. *Sintaxis Abstracta:* La metaclase *Value* forma parte del concepto de *industrialData*. El valor tiene definido un tipo de dato de tipo *DataType*.

Sintaxis Abstracta	Sintaxis Concreta
	Aparece como un atributo de *IndustrialData*, cuya sintaxis está descrita en el *IndustrialData*.

Restricciones:
El valor almacenado en la metaclase *Value* tiene que ser de tipo *DataType*:
Context Value inv IsDataType: self.TypedElement.type

EL PAQUETE TYPES

Este paquete incluye los tipos de datos que son admitidos por iMMAS. Al igual que ocurre en UML y en cualquier lenguaje se admiten los tipos primitivos de datos como, por ejemplo, los especificados en el estándar IEC 61131-3 (ver capítulo 4), y tipos enumerados con los cuales se pueden definir valores específicos. El

concepto *DataType* recoge los tipos que pueden tener cada uno de los datos manejados por un sistema industrial.

Por defecto, se han definido cuatro tipos primitivos a partir de los tipos primitivos que proporciona MOF:

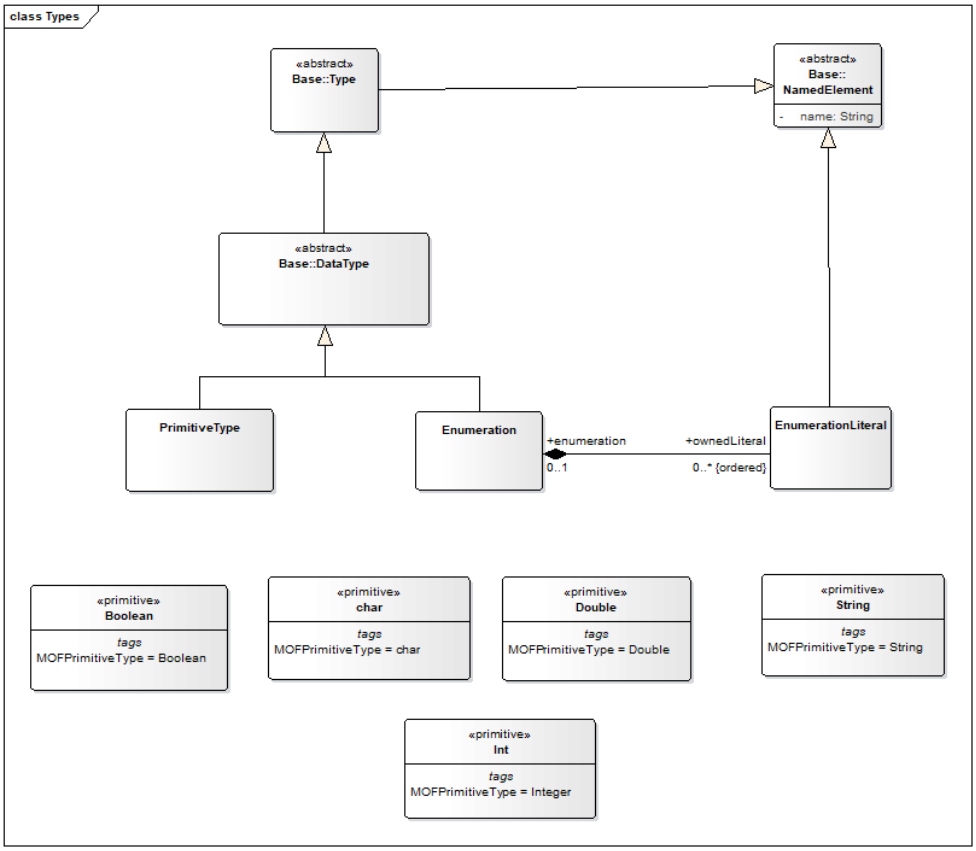


Figura 5.10: Tipos de datos definidos para iMMAS.

El metamodelo realizado admite la definición de nuevos tipos de datos. Por defecto, hemos utilizado los tipos de datos genéricos definidos en OPC UA, aunque es posible extenderlo al resto de tipos de datos específicos como *int8*,*int16*, *int32*.

EL PAQUETE PACKAGE

El paquete Package incluye abstracciones para definir los tipos de agrupaciones permitidos en iMMAS. El elemento sintáctico principal es *Package* que se define de un modo similar a como se realiza en UML. De modo que un paquete

puede contener distintos tipos de elementos especificados en otros paquetes del metamodelo (tipos de datos, modulos, componentes, etc.). Además los paquetes pueden también anidarse e incluir a su vez a otros paquetes.

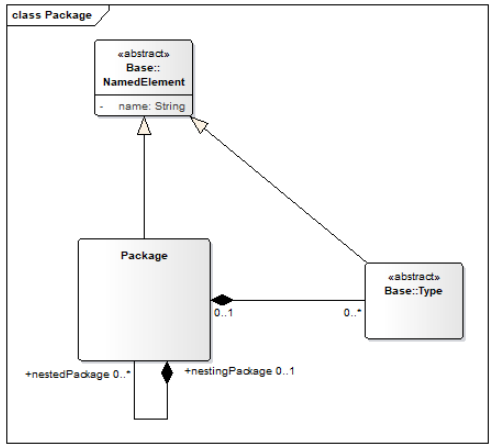


Figura 5.11: Tipos de datos definidos para iMMAS.

EL PAQUETE BEHAVIOR

Un sistema industrial está compuesto por conjuntos de datos que se encuentran organizados en iMMAS en base a componentes y módulos, que se actualizan porque o bien son manejados por algún dispositivo industrial o bien son actualizados por el propio usuario. Existe, sin embargo, la posibilidad de que algún elemento del sistema industrial tenga definido un comportamiento activo propio, de tal modo que puede actualizar o modificar su estado o el conjunto de datos industriales que controla. En iMMAS solo determinados componentes pueden tener un comportamiento activo, que hemos denominado componentes activos.

El comportamiento de un componente activo está determinado por las acciones que realiza y los estados por los que pasa a lo largo de la evolución temporal de dicho componente. De hecho, en estos casos la reactividad de un sistema depende de la acción que se va a realizar, los parámetros que parametrizan dicha acción y la historia pasada del sistema. Para modelar dicho comportamiento reactivo en iMMAS nos basamos en la utilización de la programación basada en estados, en concreto en los fundamentos de las máquinas de estados finitas. La ventaja de utilizar una descripción del comportamiento en máquina de estados es que dicha programación es compatible con la programación que se realiza en

los dispositivos industriales [103][88].

Una máquina de estados finita consta de una serie de estados y una función de transición que indica cual es el siguiente estado. Las máquinas de estados pueden presentarse en diferentes formas y variaciones. Para poder modelar el comportamiento basándonos en máquinas de estados, debemos de introducir en iMMAS los conceptos de estado y transición.

Estados y transiciones en iMMAS

Para modelar el comportamiento con iMMAS utilizando máquinas de estado se manejan dos elementos fundamentales a través de los cuales orbitan el resto de conceptos: estado y transición.

- Estado: Permite caracterizar la situación temporal en la que se encuentra el sistema en función del valor que tiene un conjunto de atributos, en función de la espera a que se produzca una transición, o bien en función del procesamiento que se está realizando. En general los estados en los que se encuentra el sistema deben ser excluyentes, ya que el sistema no puede encontrarse en más de un estado al mismo tiempo.

- Transición: Identifica cada uno de los cambios de estado que puede tener el sistema. Una transición se puede disparar cuando se encuentra en un estado de inicio, cundo sucede un evento, cuando se realiza una acción o se invoca un método u operación, y cuando se entra en un estado de terminación o fin.

A continuación se describen cada uno de los elementos sintácticos del lenguaje para la definición del comportamiento de un componente activo:

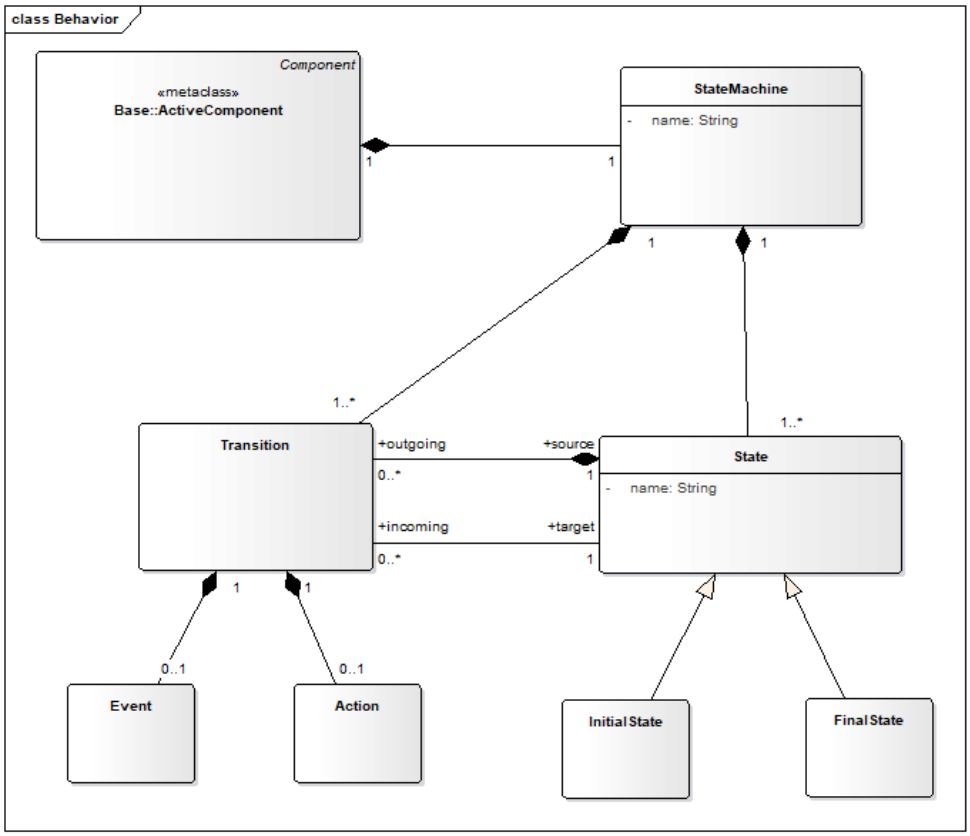


Figura 5.12: Elementos que describen el comportamiento en iMMAS.

- **StateMachine.**

  Contenido Semántico: Este elemento sintáctico hace referencia al concepto de máquina de estado como el artefacto software que incluye el cambio de estado que se produce durante la evolución del comportamiento de un componente activo.

  Sintaxis Abstracta: Este elemento sintáctico necesita tener definido un nombre que no puede repetirse durante la descripción de un sistema.

  Sintaxis Concreta: La sintaxis concreta que define una máquina de estado viene dado por la notación de una clase UML.

  Restricciones:

  La máquina de estado debe poseer al menos un pseudoestado de inicio, y una transición que cambie del pseudoestado de inicio a un estado *idle*.

  La máquina de estado puede poseer un pseudoestado de fin, aunque no es

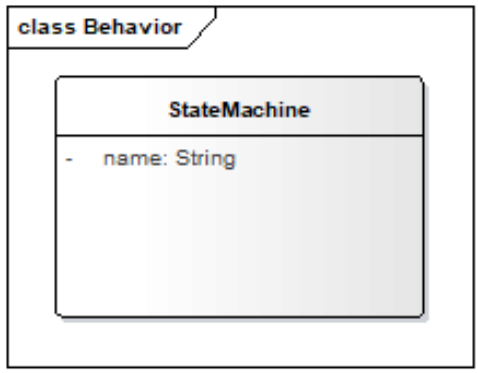


Figura 5.13: Sintaxis Concreta para StateMachine.

obligatorio.

- **State.**
  Contenido Semántico: El estado es el concepto fundamental que se utiliza para definir la situación temporal en la que se encuentra el sistema.
  Sintaxis Abstracta: El estado viene identificado por un nombre. Además, puede haber dos tipos de pseudoestados en una máquina de estados: *InitialState* para identificar el pseudoestado de inicio, y *FinalState* para identificar el pseudoestado de fin.

  Sintaxis Concreta:

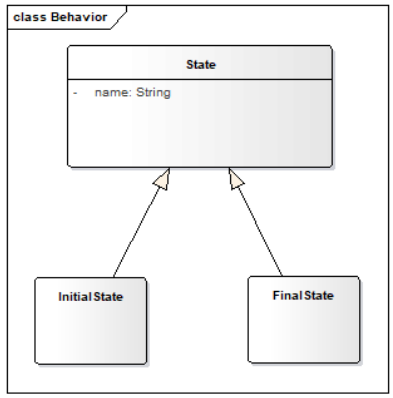


Figura 5.14: Sintaxis Concreta para State.

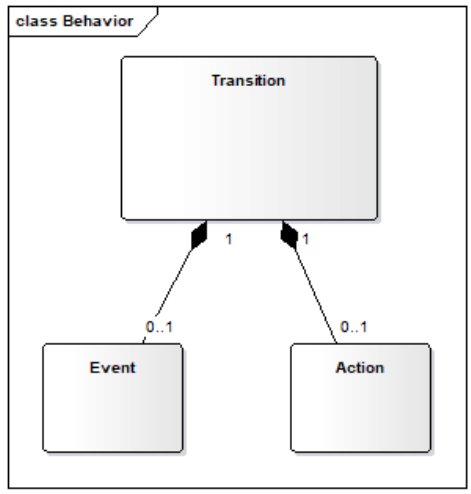


Figura 5.15: Sintaxis Concreta para Transition.

Se expresa mediante una elipse con el nombre del estado, que debe coincidir con uno de los identificados

Restricciones:
No puede haber dos estados con el mismo nombre.
Un pseudoestado de inicio no tiene nombre.
Context InitialState inv NoNameInitialState: self.Name.
Un pseudoestado de fin no tiene nombre.

- **Transition.**
  Contenido Semántico: Este elemento recoge el concepto de transición que especifica cómo se realiza el cambio de un estado al otro. Para caracterizar una transición que dispara el cambio de estado puede definirse un evento, una guarda y/o una acción. La característica principal de la transición es que se ejecuta de forma atómica y de forma instantánea (no tiene un consumo de tiempo específico).
  Sintaxis Abstracta: Este elemento no tiene ninguna propiedad ni método adicional.
  Sintaxis Concreta: La sintaxis concreta es un arco que se utiliza para conectar dos estados. La descripción de la transición se puede expresar mediante la siguiente sintaxis: evento [guarda] / acción, es decir, un evento o una

condición de guarda disparan la transición, y la acción determina lo que se ejecuta de forma atómica durante la transición de estado.

Restricciones:
Una transición puede contener un evento.
Context Transition inv Fullspec: self.event->notEmpty() implies self.event.size() == 1
Una transición puede contener opcionalmente una accion.
Context Transition inv Fullspec: self.action->notEmpty() implies self.action.size() == 1

- **Event.**
  Contenido Semántico: Este elemento sintáctico se utiliza para especificar la causa que genera el disparo de una transición. Puede identificar el nombre del evento o bien una guarda.
  Sintaxis Abstracta: Este elemento se incluye de forma opcional al definir una transición.

  Sintaxis Concreta:

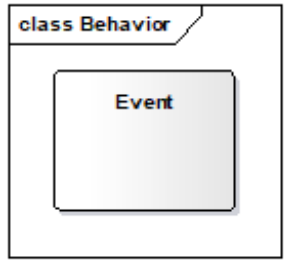


Figura 5.16: Sintaxis Concreta para Event.

La sintaxis concreta supone agregar el nombre del evento al definir la transición.

- **Action.**
  Contenido Semántico: Este elemento recoge el método o la operación que se dispara durante la transición entre dos estados.
  Sintaxis Abstracta: Este elemento se incluye de forma opcional al definir

una transición.
Sintaxis Concreta:

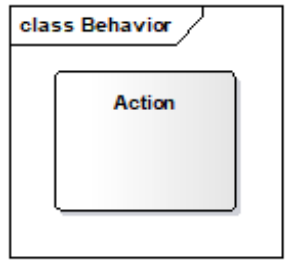


Figura 5.17: Sintaxis Concreta para Action.

Se incluye mediante la expresión del método.

- **InitialState.**
  Contenido Semántico: Este elemento hace referencia a un pseudoestado específico que aparece sólo una vez en cada máquina de estado para arrancarla.
  Sintaxis Abstracta: Este tipo de estado especial hereda las propiedades de estado pero no tiene definido ningún nombre.
  Sintaxis Concreta:

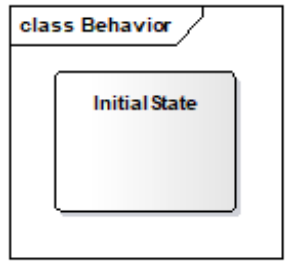


Figura 5.18: Sintaxis Concreta para InitialState.

Se define mediante un punto.

- **FinalState.**
  Contenido Semántico: Este elemento extiende del concepto State, y recoge el concepto de estado final dentro de una máquina de estados que indica que se ha completado la tarea a realizar.

Sintaxis Abstracta: Este elemento no tiene ninguna propiedad ni método adicional.
Sintaxis Concreta:

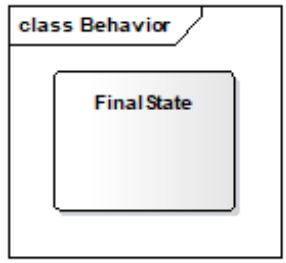


Figura 5.19: Sintaxis Concreta para FinalState.

Se define mediante un doble punto.

En la figura 5.20 se puede ver como se relacionan todos los elementos de este paquete.

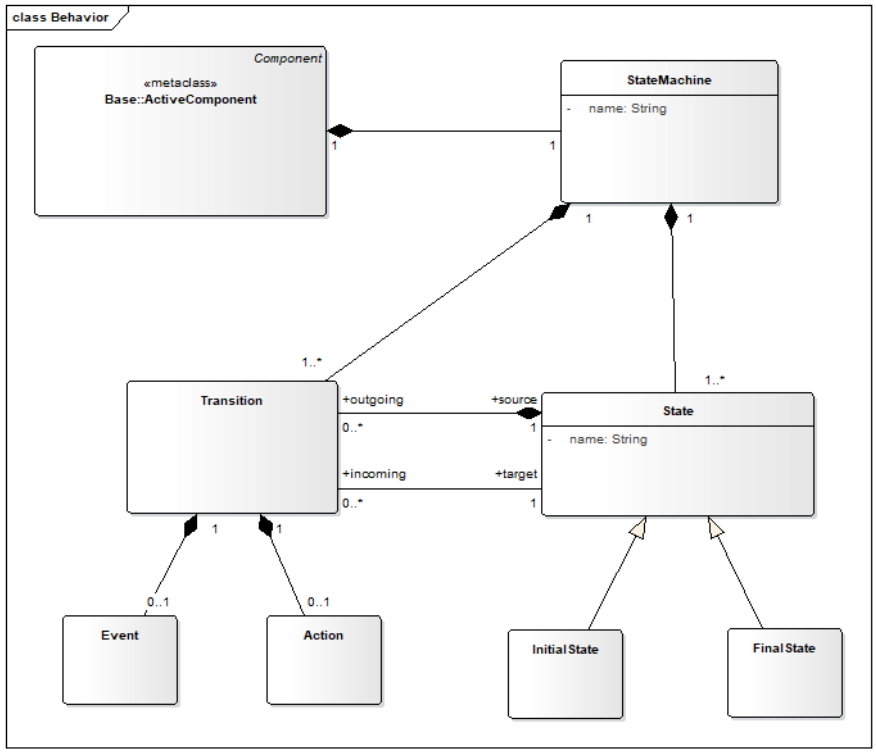


Figura 5.20: Elementos y relaciones del paquete behavior.

EJEMPLO DE MODELADO CON IMMAS

Con todos los conceptos de iMMAS base vamos a proceder a modelar un sistema industrial. El sistema escogido se puede ver en la figura 5.21. Este sistema está formado por dos tanques de almacenamiento de ingredientes líquidos, los cuales se llenan por la parte superior de los mismos utilizando la bomba P1 y las válvulas V11 y V21, y se vacían por la parte inferior utilizando la bomba P2 y las válvulas V12 y V22. Los elementos que forman todo el sistema completo son:

- 2 bombas de impulsión de velocidad regulable mediante variador de frecuencia.
- 4 válvulas de 2 posiciones y 2 vías.
- 4 detectores de nivel digitales.
- 2 tanques de acero inoxidable.

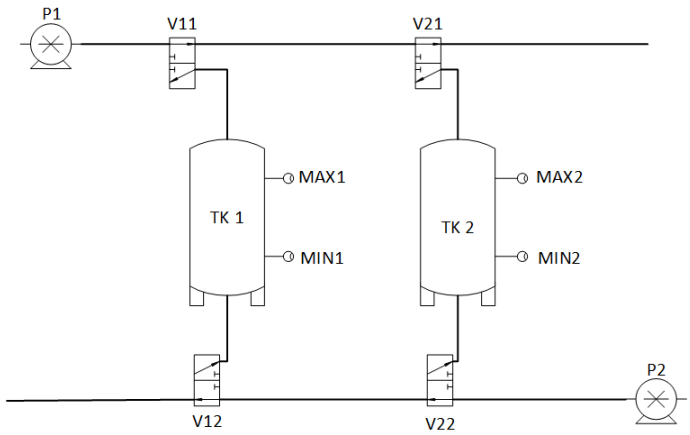


Figura 5.21: Diagrama de Flujo sistema de dosificación líquida.

En este sistema los tanques poseen dos secuencias, llenado y vaciado. Ambas secuencias no se pueden producir a la vez, de modo que no se puede estar llenando y vaciando un tanque al mismo tiempo. Además las operaciones de llenado y vaciado son parcialmente excluyentes, en el sentido de que si un tanque está siendo llenado, el otro tanque no puede llenarse a la vez y, si un tanque está vaciándose, el otro tanque no puede vaciarse a la vez. Sólo se puede producir simultáneamente una operación de llenado en un tanque y de vaciado en el otro

tanque, o viceversa. Las fases de llenado y vaciado se deben de realizar según la siguiente especificación:

- Llenado: Esta fase comienza cerrando la válvula V12 o V22 del tanque que no va a ser llenado. Una vez que se reciba confirmación de que la válvula está cerrada, se procederá a abrir la válvula V11 o V21 para el tanque que se vaya a llenar. Una vez que se reciba la confirmación de que la válvula está abierta se arrancará la bomba P1 con una consigna de velocidad en porcentaje. La bomba estará trabajando con dicha consigna hasta que el detector MAX1 o MAX2 del tanque que se esté llenando detecte nivel. En este momento se parará la bomba P1 para que no introduzca más líquido y cuando se reciba la confirmación de que la bomba esta parada se procederá a cerrar la válvula V11 o V21 dependiendo del tanque que se haya llenado. Por ejemplo, en el caso del tanque TK1, la secuencia sería:
  - Cierre de la válvula V21, apertura de la válvula V11 tras recibir la confirmación de cierre de la válvula V21.
  - Una vez que se ha recibido la confirmación de apertura de la válvula V11, se arranca la bomba P1 con una consigna de velocidad del 80 %.
  - Cuando el detector de nivel MAX1 detecte nivel, señal ON o True, se puede dar por terminado el proceso de llenado del tanque.
  - La secuencia de fin de llenado es la siguiente: parada de la bomba P1, cuando se tenga confirmación de bomba parada, se procederá a cerrar la válvula V11.
- Vaciado: Esta fase comienza cerrando la válvula V12 o V22 del tanque que no va a ser vaciado. Una vez que se reciba confirmación de que la válvula está cerrada, se procederá a abrir la válvula V12 o V22 para el tanque se vaya a vaciar. Una vez que se reciba la confirmación de que la válvula está abierta, se arrancará la bomba P2 a una consigna de velocidad en porcentaje. La bomba estará trabajando a esta consigna hasta que el detector MIN1 o MIN2 del tanque que se esté vaciando pierda el nivel. En este momento se parará la bomba P2 para que no extraiga más líquido, después se cerrará la válvula V12 o V22 dependiendo del tanque se haya vaciado. Por ejemplo en el caso del tanque TK2, la secuencia sería:

– Cierre de la válvula V12, y apertura de la válvula V22 tras recibir la confirmación de cierre de la válvula V12.
– Una vez que se ha recibido la confirmación de apertura de la válvula V22, se arranca la bomba P2 a una consigna de velocidad del 80 %.
– Cuando el detector de nivel MIN1 pierda el nivel, señal OFF o False, se puede dar por terminado el proceso de vaciado del tanque.
– La secuencia de fin de llenado es la siguiente: cerrar la válvula V22 y cuando se tenga confirmación de válvula cerrada se procederá a parar la bomba P2.

Para este sistema se ha realizado el siguiente modelo, utilizando iMMAS:

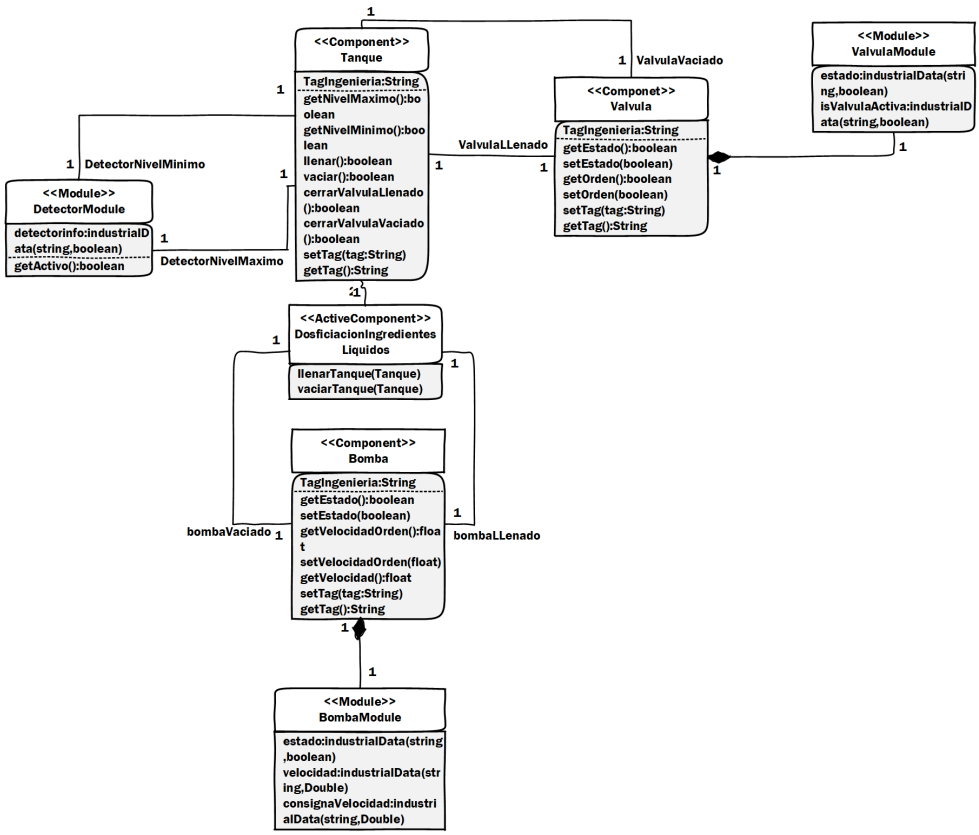


Figura 5.22: Modelado en iMMAS para el control de la dosificación líquida.

El elemento principal de dicho sistema es *DosificacionIngredientesLiquidos*, el cual se encarga de controlar el llenado y vaciado de los tanques a través de las

bombas. Por ese motivo se conecta por un lado con los dos tanques y por otro con las dos bombas, bomba de vaciado y bomba de llenado.

Para manejar el tanque se ha creado un elemento *Tanque* de tipo *Component* con una propiedad *TagIngenieria* para identificarlo, y con un conjunto de operaciones que nos permite trabajar tanto con la válvula que cierra o abre el llenado del tanque, como con los detectores que nos permitirá saber si hemos llegado al nivel mínimo o máximo.

Para que el tanque pueda controlar si se ha alcanzado el nivel mínimo o máximo de llenado se dispone de dos módulos *DetectorModule* que almacenarán de forma persistente las entradas que nos permite controlar si se ha detectado nivel mínimo o máximo en dicho tanque. Para ello, cada modulo *DetectorModule* tiene una variable *detectorinfo* de tipo *industrialData* que almacena un valor *booleano* que será *True* si ha detectado el nivel del líquido. La operación *getActivo* nos permitirá determinar si el detector está activo o no. El tanque podrá acceder al estado del detector mediante la invocación de las operaciones, *getNivelMaximo* y *getNivelMinimo*, que nos indican el estado de los detectores de nivel máximo y mínimo consultando los correspondientes *DetectorModule*. ara que se pueda realizar el llenado y vaciado del tanque se necesita abrir/cerrar dos válvulas. Para ello, en este caso hemos conectado el tanque a dos componentes de tipo *Valvula*, uno para el llenado y otro para el vaciado, en lugar de utilizar un módulo. Cada componente *Valvula* contiene a su vez un módulo *ValvualModule* para mantener persistentemente el valor del estado de la válvula a través de la variable estado de tipo IndustrialData booleano. El componente *Valvula* cuenta con seis operaciones, *setEstado* y *getEstado* para activar/desactivar y consultar el estado de la válvula, *setOrden* y *getOrden* para controlar la apertura/cierre de la válvula, y *getTag* y *setTag* para gestionar la propiedad TagIngenieria. El tanque puede controlar la válvula de llenado mediante las operaciones *llenar()* y *cerrarValvulaLlenado()*, y la válvula de vaciado mediante las operaciones *vaciar()* y *cerrarValvulaVaciado()*.

Para manejar las bombas se ha creado un elemento *Bomba* de tipo *Component* con una propiedad *TagIngenieria* para identificarlos, y con un conjunto de operaciones que nos permite trabajar con la bomba: *getVelocidad* para gestionar la velocidad a la que esta trabajando la bomba; *getEstado* y *setEstado* para determinar el estado; y *getVelocidadOrden* y *setVelocidadOrden* para establecer la

consigna de velocidad de la bomba. Los datos manejados por Bomba son gestionados por un módulo *BombaModule* que contiene tres *IndustrialData*: *estado* de tipo *booleano* para recoger el estado de la bomba, *velocidad* de tipo *Double* para recoger la velocidad real del elemento, y *consignavelocidad* de tipo *Double* para recoger la consigna de velocidad dada a la bomba.

Por último, dado que *DosificaciónIngredientesLiquidos* es de tipo *ActiveComponent* tiene asociado una máquina de estados que se muestra en la figura 5.23.

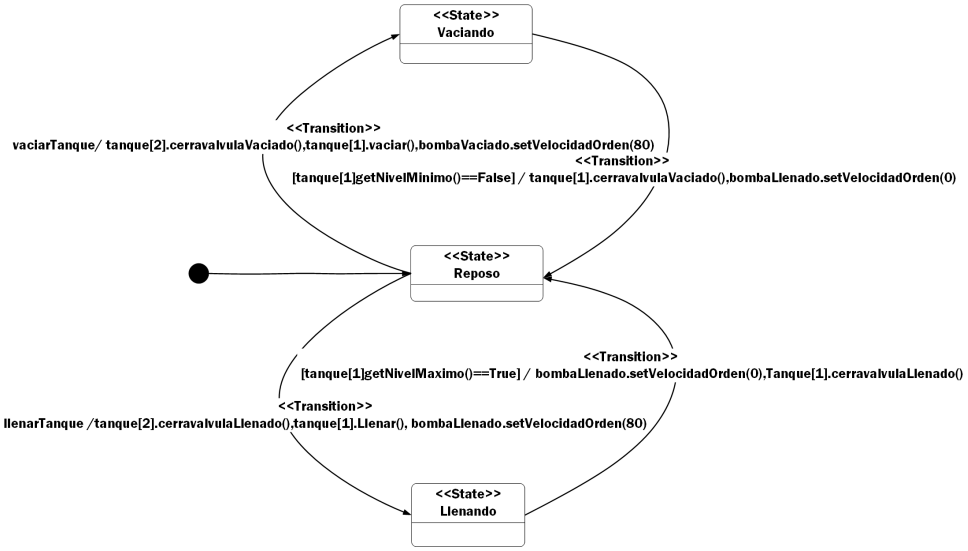


Figura 5.23: Maquina estados para modelar el comportamiento del sistema.

En esta máquina de estados se modelan las secuencias de llenado y vaciado del sistema para el tanque 1, siendo el modelo totalmente análogo para el caso del tanque 2. Se han definido tres estados: *Reposo* en el que el tanque no hace nada, *Vaciando* en el que el tanque se encuentra vaciando líquido y *Llenando* en el que el tanque se encuentra incorporando liquido. Desde el estado Reposo se puede pasar al estado *Llenando* siempre que se haya cerrado la válvula de llenado del tanque 2 *(tanque[2]. cerrarvalvulaLlenado())*, se haya abierto la válvula de llenado del tanque 1 *(tanque[1].Llenar())*, y se ha arrancado la bomba de llenado a un 80% *(bombaLlenado.setVelocidad(80))*. Desde el estado *Llenando* solo podemos pasar el estado de reposo cuando se cumplan las condiciones de llenado, es decir, el detector de nivel máximo debe de dar un valor *ON* indicando que el tanque esta lleno *([tanque[1]getNivelMaximo()==True])*, la bomba tiene que estar parada

(bombaLlenado.setVelocidadOrden(0)), y la válvula de llenado del tanque 1 este cerrada *(Tanque[1].cerravalvulaLlenado()*.

En el caso del vaciado, partimos también del estado de Reposo y pasamos a vaciar el tanque si la válvula de vaciado del tanque 2 esta cerrada *(tanque[2]. cerravalvulaVaciado())*, si la válvula de vaciado del tanque 1 esta abierta *(tanque[1].vaciar())* y si la bomba de vaciado está arrancada a un 80 % (*bombaVaciado.setVelocidadOrden(80)*). Desde el estado de Vaciando solo podemos volver al estado de reposo si el detector de mínimo del tanque 1 ha perdido señal *(tanque[1]getNivelMinimo()==False])*, se ha cerrado la válvula de vaciado *(tanque[1]. cerravalvulaVaciado())*, y se ha parado la bomba de vaciado *(bombaLlenado. setVelocidadOrden(0))*.

5.6. PERFIL PARA EL MODELADO DE ENTORNOS INDUSTRIALES ESTRUCTURADOS (MOSIE)

El lenguaje de modelado de iMMAS establece los elementos sintácticos y las relaciones que hay entre dichos elementos con una base semántica diseñada específicamente para el diseño de modelos en sistemas industriales. Además del lenguaje necesitamos definir abstracciones específicas que faciliten el modelado de sistemas industriales basados en los elementos sintácticos del lenguaje. Para ello se ha diseñado un nuevo perfil sobre iMMAS que incluye un conjunto estructurado de abstracciones utilizado directamente por los desarrolladores para el modelado de sistemas industriales. Este nuevo perfil denominado MOdelado de Sistemas Industriales Estructurado (MOSIE) establece unas directrices acerca de cómo deben organizarse los elementos del sistema industrial basándonos en el lenguaje iMMAS. El perfil aprovecha la capacidad de extensión del lenguaje gracias a los estereotipos definidos en los clasificadores del lenguaje: módulos y componentes. Por este motivo el perfil está estructurado en paquetes, en base a módulos y componentes específicos.

El lenguaje de iMMAS proporciona la flexibilidad para poder modelar un sistema industrial completo. Sin embargo, la aplicación del perfil MOSIE sobre los modelos proporcionará una serie de plantillas construidas en base a la experiencia en la programación de sistemas industriales que facilitan el modelado de los elementos de un sistema industrial.

El paquete *modules* contiene un conjunto de módulos específicos que almacenan persistentemente información que va a ser manejada por los componentes definidos en el sistema industrial. Este paquete contiene un conjunto de tipos de módulos agrupados en subpaquetes:

- Input/output. Incluye los distintos tipos de módulos que se utilizan habitualmente para la caracterización de las entradas/salidas manejadas por los dispositivos industriales. Dichos módulos tendrán que asociarse a los componentes que se definan en el sistema.
- Control. Incluye los módulos específicos para el almacenamiento persistente de la información manejada por reguladores y controladores. De igual modo estarán asociados a los componentes que se definan en el sistema.
- Alarms. Contiene los módulos específicos que nos permiten definir las alarmas del sistema que se disparan o se activan debido a alguna condición.

De igual modo, el paquete componentes se compone a su vez de un conjunto de subpaquetes anidados que incluye diferentes tipos de componentes con distinto nivel de abstracción:

- Devices: Recoge los conceptos básicos para poder modelar cualquier tipo de dispositivo, actuador o sensor.
- Actuators: Agrupa todos los conceptos para modelar un actuador industrial.
- Sensors: Agrupa todos los conceptos para modelar un sensor industrial.
- Controllers: Este paquete contiene todos los conceptos para modelar un regulador.

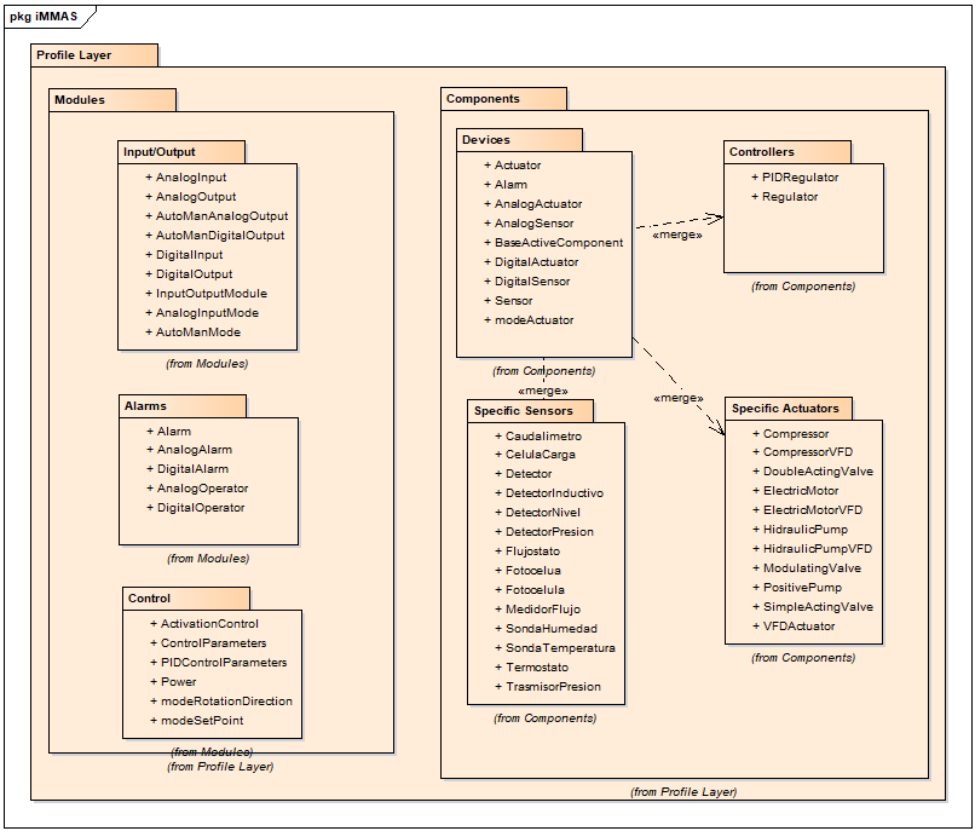


Figura 5.24: Conjunto de paquetes que forma MOSIE.

A continuación vamos a describir todos los paquetes/perfiles que se han creado dentro de MOSIE.

5.6.1. EL PAQUETE MODULES

El paquete Module contiene los diferentes tipos de módulos que puede haber en un sistema industrial mediante la definición de estereotipos. El estereotipo es un mecanismo de extensibilidad que proporciona iMMAS para categorizar elementos sintácticos que tienen una misma semántica en un contexto determinado. En este caso el estereotipo en el módulo define un conjunto de atributos y operaciones que caracterizan a un tipo de módulo. De este modo, cuando el desarrollador incluya el estereotipo en sus modelos está haciendo referencia a un tipo de módulo con un significado concreto. Por ejemplo, *DigitalInput* hace referencia a un tipo de módulo concreto que contiene una entrada de tipo digital,

cuyo valor se almacena persistentemente; si el desarrollador definiera un módulo de tipo *DigitalInput* estaría obligado a incluir como mínimo todos los atributos y operaciones que lo caracterizan.

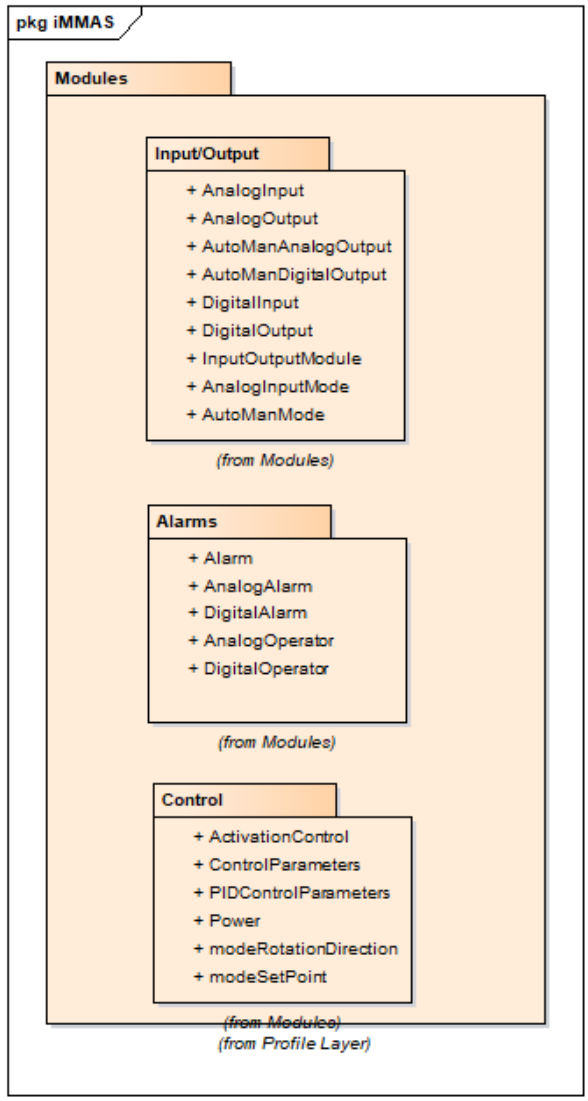


Figura 5.25: Sub-paquetes que conforman el paquete Modules.

Este paquete esta a su vez subdivido en 3 paquetes: Alarms, Input/outuput y Modules.

EL SUBPAQUETE INPUT-OUTPUT

Este paquete contiene los tipos de módulos que recogen los datos industriales o señales que hacen referencia a las entradas/salidas de un dispositivo industrial. Los datos que se almacenan de las entradas/salidas se pueden expresar con magnitudes físicas (especialmente en el caso de entradas/salidas analógicas) sin importar el modelo de dispositivo concreto utilizado o su fabricante. En la figura 5.26 se pueden ver todos los tipos de módulos definidos y de qué atributos y operaciones dependen.

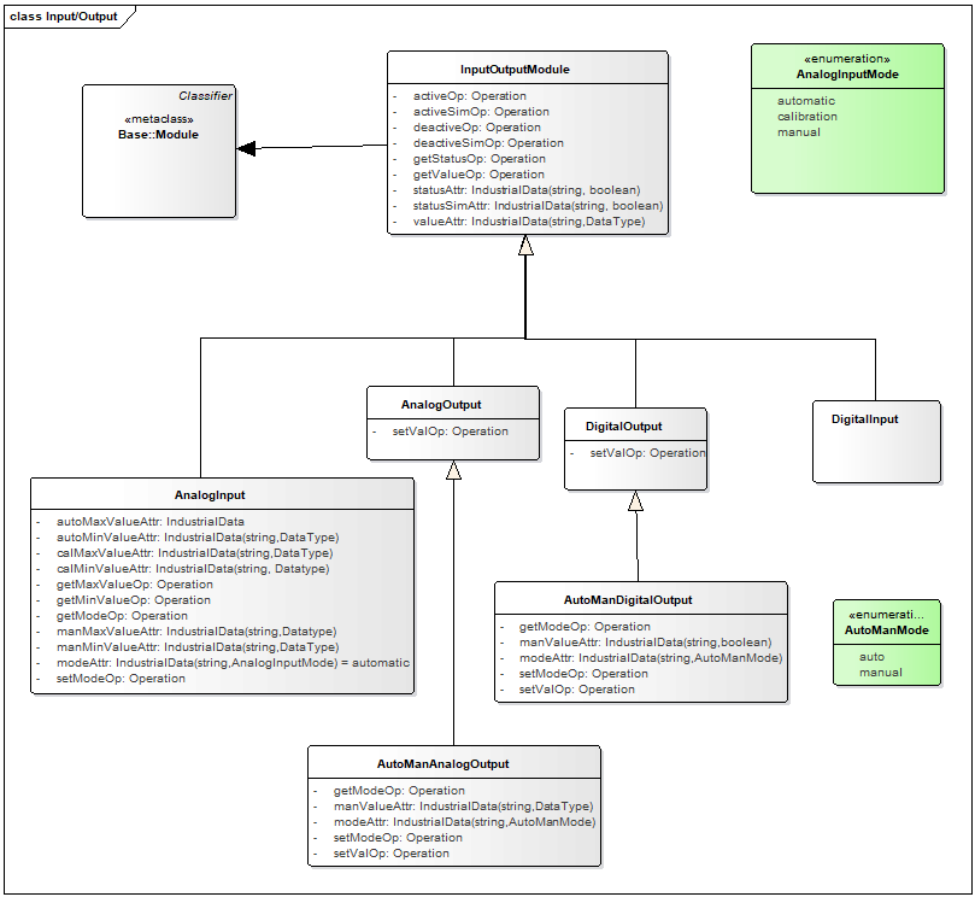


Figura 5.26: Tipos de módulos definidos en el paquete InputOuput.

A continuación se listan los tipos de módulos disponibles:

- **InputOutputModule.**
  Este tipo de modulo base contiene la definición de los atributos y opera-

ciones comunes a todos los módulos de entrada/salida. De forma general define tres atributos utilizando *IndustrialData* para el almacenamiento persistente: *statusAttr* que hace referencia al estado del elemento (activado/desactivado), *statusSimAttr* que hace referencia a si el elemento está en estado simulación o no, y por último *valueAttr* para almacenar el valor de la entrada o salida. Adicionalmente contiene las operaciones que permiten manejar sus atributos tanto para lectura como escritura, activación o desactivación.

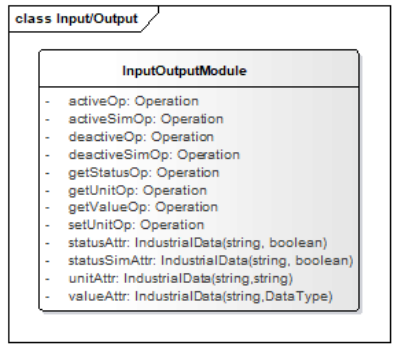


Figura 5.27: Representación del elemento InputOutputModule.

- **DigitalInput.**

  Este concepto pretende modelar las entradas digitales del dispositivo industrial. Dichas entradas digitales pueden servir para leer activaciones de elementos como detectores y actuadores todo/nada, así como sensores digitales. En este caso, con las operaciones y atributos de *InputOutputModule* se cubren todos los aspectos de este elemento.

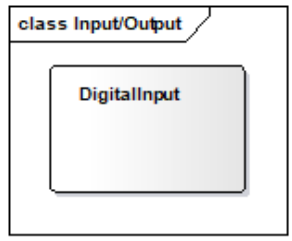


Figura 5.28: Representación del tipo de módulo digitalInput.

- **DigitalOuput.**
  Mediante este concepto se pretende modelar las diferentes salidas digitales del dispositivo industrial. Estas salidas digitales pueden activar elementos como válvulas o motores que solo tiene acciones todo/nada, es decir, *start* o *stop* en el caso de un motor u open/close en el caso de un válvula. Al igual que ocurría con *DigitalInput*, *DigitalOutput* extiende del concepto *InputOutputModule* pero en este caso al tratarse de una salida se ha introducido un método *setVal* que se encarga de escribir un valor en el atributo *value* de *DigitalOutput*.

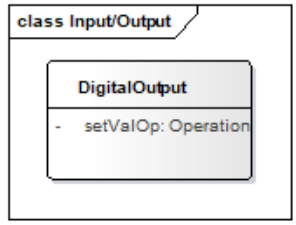


Figura 5.29: Representación del tipo de módulo digitalOutput.

- **AutoManDigitalOuput.**
  Con esta abstracción se modela un tipo particular de módulo que contiene salidas digitales que pueden operar en dos modos: automático cuando el propio dispositivo industrial gestiona el valor de salida; y manual cuando existe un sistema externo, por ejemplo, un SCADA que opera sobre dicha salida. Este elemento extiende de *DigitalOutput*, por lo que solo es necesario dotar de atributos y operaciones para gestionar los dos modos de funcionamiento de este elemento manual y automático.

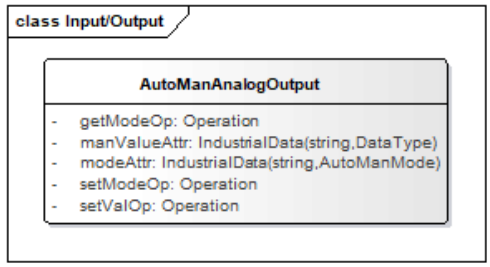


Figura 5.30: Representación del tipo de módulo digitalOutputAutoMant.

- **AnalogOuput.**

  Este estereotipo permite modelar los módulos con las diferentes salidas analógicas de un dispositivo industrial. Estas salidas analógicas se pueden utilizar para regular el funcionamiento de actuadores como bombas, válvulas modulantes y reguladores de presión, los cuales necesitan un valor analógico que les indique, por ejemplo, la velocidad de funcionamiento o el porcentaje de apertura. Al igual que ocurría con los elementos anteriores, extiende del concepto *InputOutputModule* pero en este caso al tratarse de una salida se ha introducido un método *setVal* que se encarga de escribir un valor en la propiedad *value*.

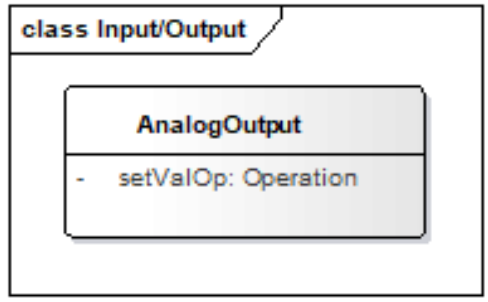


Figura 5.31: Representación del tipo de módulo AnalogOutput.

- **AutoManAnalogOutput.**

  Este estereotipo modela un tipo particular de módulo que contiene salidas analógicas que puedan operar en dos modos: automático cuando el propio dispositivo industrial gestiona el valor de salida; y manual cuando existe un sistema externo, por ejemplo, un SCADA que opera sobre dicha salida. Este elemento extiende de *AnalogOuput*, por lo que solo es necesario dotar de atributos y operaciones para gestionar los dos modos de funcionamiento de este elemento manual y automático.

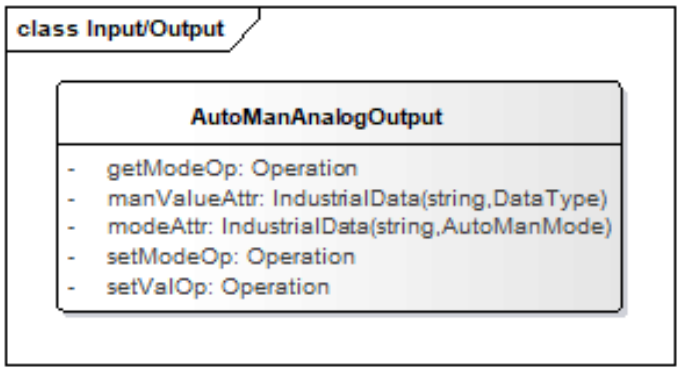


Figura 5.32: Representación del tipo de módulo analogOutputAutoMan.

- **AnalogInput.**
  Este es el tipo de módulo más complejo de este sub-paquete, ya que en una entrada analógica no sólo depende del valor que proporciona, sino también del intervalo de trabajo en el que se proporciona el valor. En el caso de entradas analógicas para realizar una medición de una magnitud física tenemos que hacer una conversión de la señal eléctrica a un señal digital a través de un conversor analógico-digital. Por lo tanto, es necesario incluir una serie de atributos y operaciones adicionales que nos permita comprender el intervalo de trabajo con el que se obtiene el valor analógico.

  Por este motivo se han incluido variables para almacenar los umbrales de calibrado que pueden ser definidos por el fabricante y/o determinados manualmente por el desarrollador/usuario. Así, *calMinValue* y *calMaxValue* determinan los umbrales de calibrado definidos por el fabricante del sensor analógico, mientras que *manCalMinValue* y*manCalMaxValue* especifica los umbrales de calibrado definidos manualmente. Mediante las operaciones *setMode* y *getMode* se puede cambiar el tipo de calibrado aplicado.

  Adicionalmente hay otro conjunto de variables, *autoMinValue* y *autoMaxValue*, para almacenar los umbrales de trabajo que alcanza el valor analógico durante su funcionamiento. Dicho umbral tendrá que ser un subconjunto del umbral de calibrado.

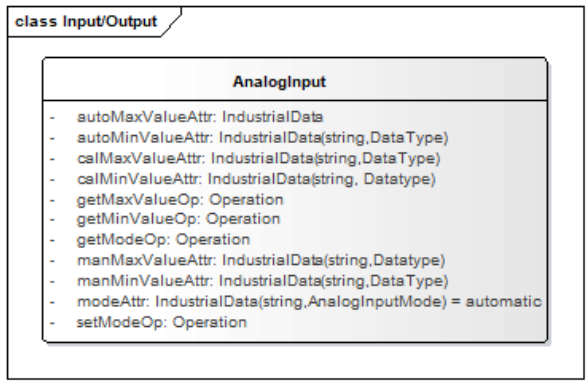


Figura 5.33: Representación del tipo de Módulo analogInput.

EL SUB-PAQUETE CONTROL

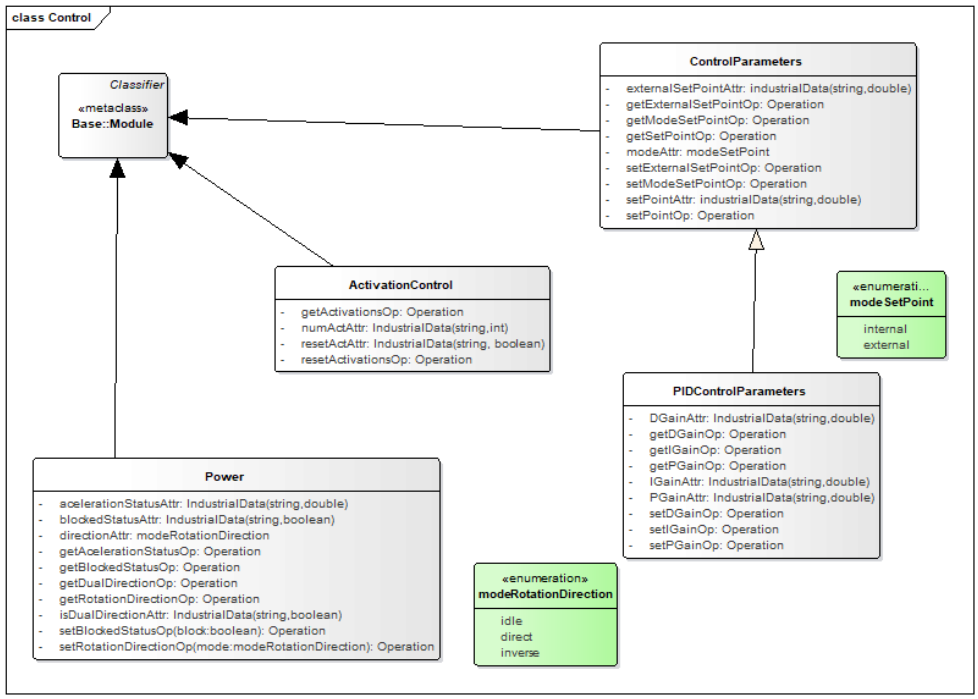


Figura 5.34: Elementos y relaciones del paquete Control.

El paquete control agrupa los tipos de módulos que han sido diseñado para almacenar los datos que nos permite crear reguladores y controladores. Los reguladores son empleados específicamente en problemas en los que el controlador tiene que trabajar con sistemas realimentados, mientras que los controladores es-

tán pensados tanto para modelar sistemas realimentados o sin realimentación en base a una simple activación o una consigna de funcionamiento. En un paquete posterior se tratarán los reguladores y controladores como componentes del sistema. Los tipos de módulos que se pueden emplear se encuentran en la figura 5.34.

- **ActivationControl**
  Este tipo de modulo ha sido creado para recoger el número de activaciones u horas de funcionamiento de un actuador, como una valvula o un motor. Para ello tiene dos atributos: *numActAttr* como un *IndustrialData* de tipo entero para contabilizar el número de activaciones/horas de funcionamiento, y *resetActAttr* como un *industrialData* de tipo *boolean* para poner a cero el valor del atributo *numActAttr*. Este tipo de módulo también posee las operaciones necesarias para poder gestionar estos atributos.

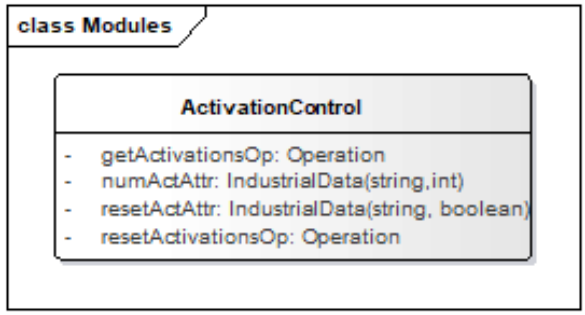


Figura 5.35: Representación del tipo de módulo ActivationControl.

- **ControlParameters.**
  Este tipo de módulo contiene los parámetros que debe de llevar un regulador simple, como por ejemplo, un controlador del tipo todo/nada. Contiene dos tipos de consigna: una interna y otra externa. La interna la gestiona directamente el controlador hardware o lógico que pongamos en el sistema industrial mediante el atributo *setPoint*, mientras que la externa puede ser gestionada por un sistema externo que puede ir cambiando en función de los criterios de ese sistema (sistemas predictivos) mediante el atributo externalSetPoint. El usuario puede elegir el tipo de consigna a aplicar mediante el atributo mode de tipo enumerado *modeSetPoint* que se gestiona con las correspondientes operaciones. Por último, la variable *outControl*

nos permitirá almacenar la señal de control que genera el controlador o regulador.

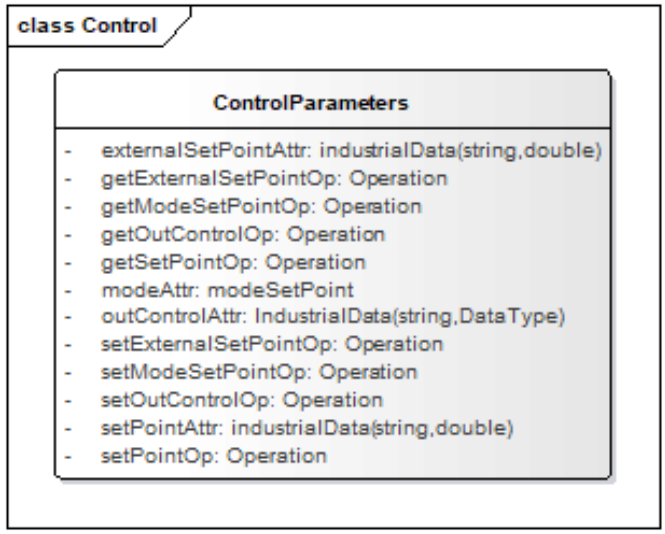


Figura 5.36: Representación del tipo de módulo ControlParametes.

- **PIDControlParameters.**

  Este tipo de módulo está pensado para almacenar los datos Kp, Kd, Ki de un controlador PID clásico, y sus variaciones PI o PD (dejando a cero la constante que no se emplea) para regular un proceso productivo. Incluye los atributos definidos en *ControlParameters*.

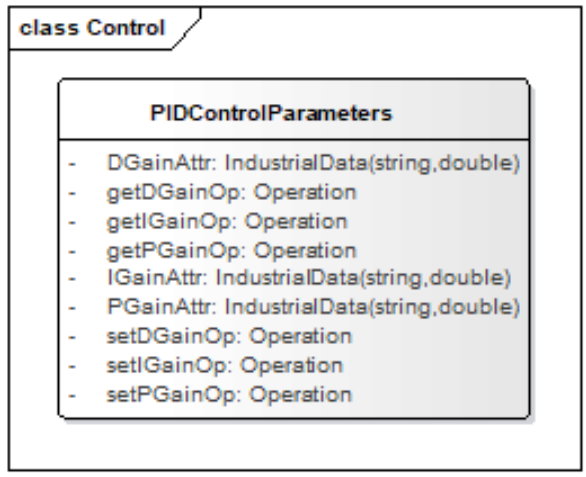


Figura 5.37: Representación del tipo de modulo PIDControlParameters.

- **Power.**
  Este tipo de módulo almacena la información que puede manejar un actuador/controlador de potencia de tipo analógico como un motor o un ventilador. Para ello incluye un conjunto de atributos específicos. El atributo *direction* de tipo *modeRotationDirection* que es un tipo enumerado *(idle, direct o inverse)* nos permite activar el funcionamiento en directo o en inverso (por ejemplo en un motor) siempre que admita el doble funcionamiento de acuerdo al valor booleano de *isDualDirection*. Por otra parte, el atributo *blockedStatus* nos permite saber si el actuador industrial está bloqueado o no, mientras que el atributo *accelerationStatus* si está acelerando y con que valor de aceleración.

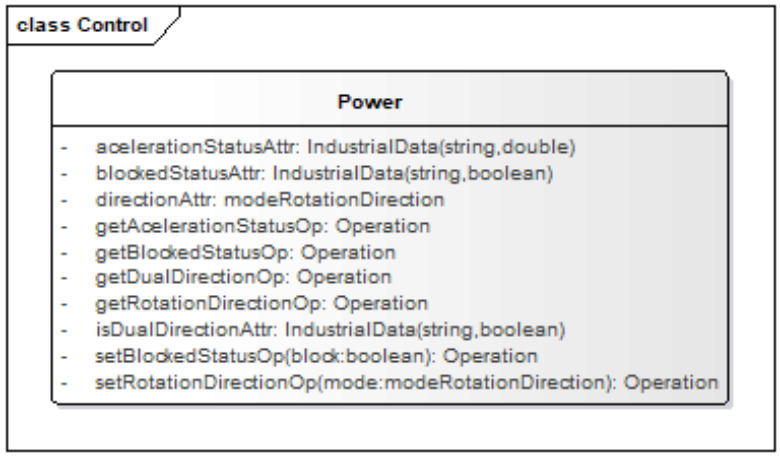


Figura 5.38: Representación del tipo de módulo Power.

EL SUB-PAQUETE ALARMS

En este paquete se recogen todos los modulos necesarios para poder dotar a un elemento de alarmas, las cuales se disparan o se activan debido a alguna condición. Los tipos de módulos que se han definido se pueden ver en la figura 5.39.

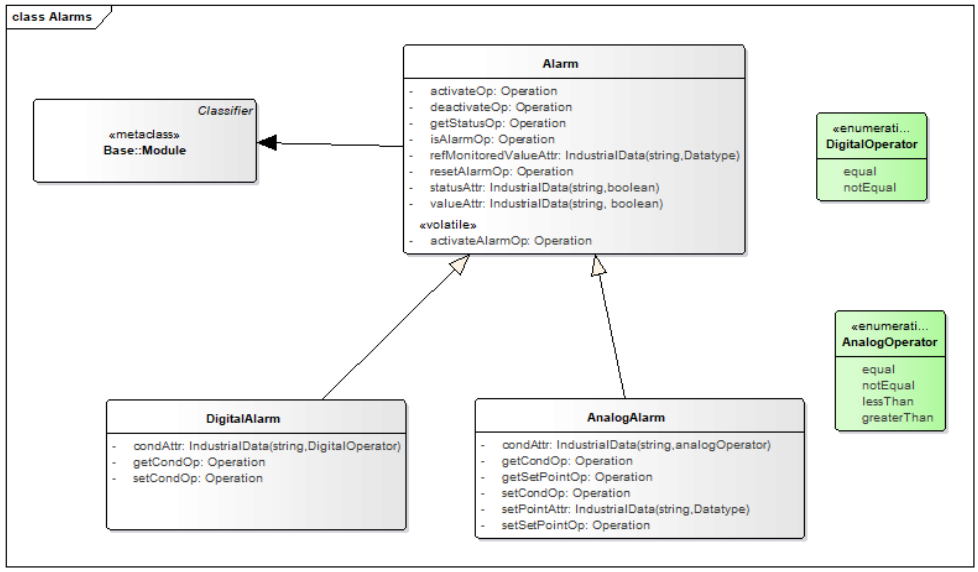


Figura 5.39: Tipos de módulos en el paquete Alarms.

- **Alarm.**
  Una alarma se puede producir por varias razones, como un mal funcionamiento, una avería eléctrica o mecánica, o porque se han infringido ciertas condiciones de trabajo. Por tanto, consideramos que cuando se dispara una alarma, ésta es critica y requiere atención inmediata. Si se necesitara establecer algún tipo de priorización o diferencia en importancia, tendría que ser gestionado por el desarrollador con otro nuevo concepto como prealarma o alerta. El módulo de tipo de alarma almacena los datos para la gestión de una alarma. Este incluye en nuestro caso tres atributos: status para indicar si la alarma está en funcionamiento o no, *value* para activar/desactivar la alarma, y *refMonitoredValue* para indicar el valor que se debe monitorizar para disparar la alarma. El método *activateAlarmOP* se ha declarado de tipo *«volatile»* para indicar que el valor de alarma, atributo *Value*, puede ser modificado desde un sistema externo, el cual dispararía la alarma. Si con la funcionalidad descrita no se cubren las necesidades específicas para determinados tipos de alarmas que podamos encontrar en algunas aplicaciones, siempre podemos realizar una extensión de este concepto añadiendo la funcionalidad necesaria para cubrir estas necesidades.

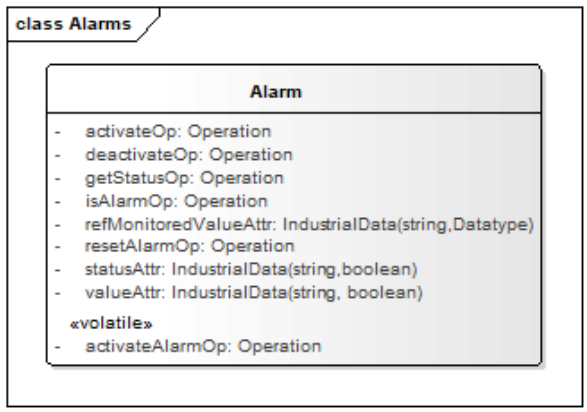


Figura 5.40: Representación del elemento Alarm.

- **DigitalAlarm**

  Este tipo de modulo extiende del concepto *Alarm* y nos sirve para modelar alarmas de tipo digital, en las que un valor digital *TRUE* o *FALSE* nos indica que el dispositivo o elemento está en alarma. A diferencia de *Alarm*, *DigitalAlarm* incluye un atributo para especificar la condición *cond* utilizada para comparar *refMonitoredValue* con valor a *TRUE*; es decir, si *cond = equal*, *refMonitoredValue equal TRUE* dispara la alarma cuando *refMonitoredValue := TRUE;* en cambio, si *cond =notEqual*, entonces *refMonitoredValue notEqual TRUE* dispara la alarma cuando *refMonitoredValue :=FALSE*.

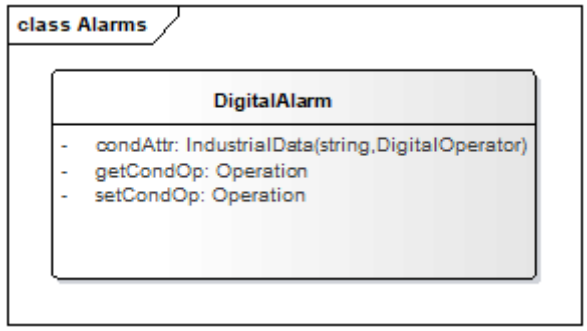


Figura 5.41: Representación del tipo de módulo DigitalAlarm.

- **AnalogAlarm**

  El tipo de módulo *AnalogAlarm* extiende del módulo de tipo *Alarm* al igual que *DigitalAlarm*. En este caso, como atributos nuevos tenemos *setPoint*

para definir el valor por defecto utilizado para la comparación, y *cond* para conocer los comparadores utilizados en un enumerado de tipo *AnalogOperator*. De este modo, si *cond* = *equal*, entonces se busca que *refMonitoredValue* = *setPoint* para disparar la alarma. Por último indicar que también se le han añadido los operaciones necesarios para manejar estos atributos.

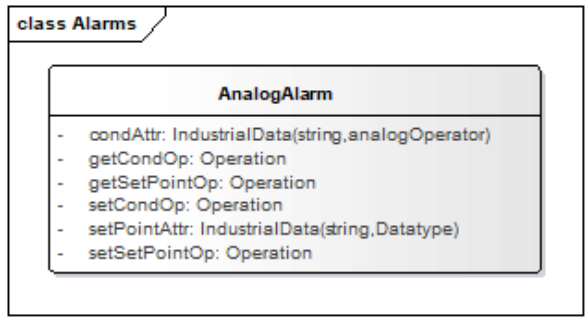


Figura 5.42: Representación del tipo de módulo AnalogAlarm.

5.6.2. EL PAQUETE COMPONENTS

Los elementos de este paquete se apoyan en las abstracciones de datos definidas como módulos en el paquete Module. Los componentes hacen referencia a elementos industriales físicos o lógicos que pueden identificarse como entidades individuales. Desde el punto de vista de iMMAS, el componente puede ser superclase de otros componentes, contener a uno o varios componentes, contener a uno o varios módulos, o tener una relación de asociación con otros componentes. Este paquete a su vez esta formado por 4 sub-paquetes: *Devices*, *Controllers*, *Sensors* y *Actuators*, como podemos ver en la figura 5.43 con conjuntos de componentes con características comunes.

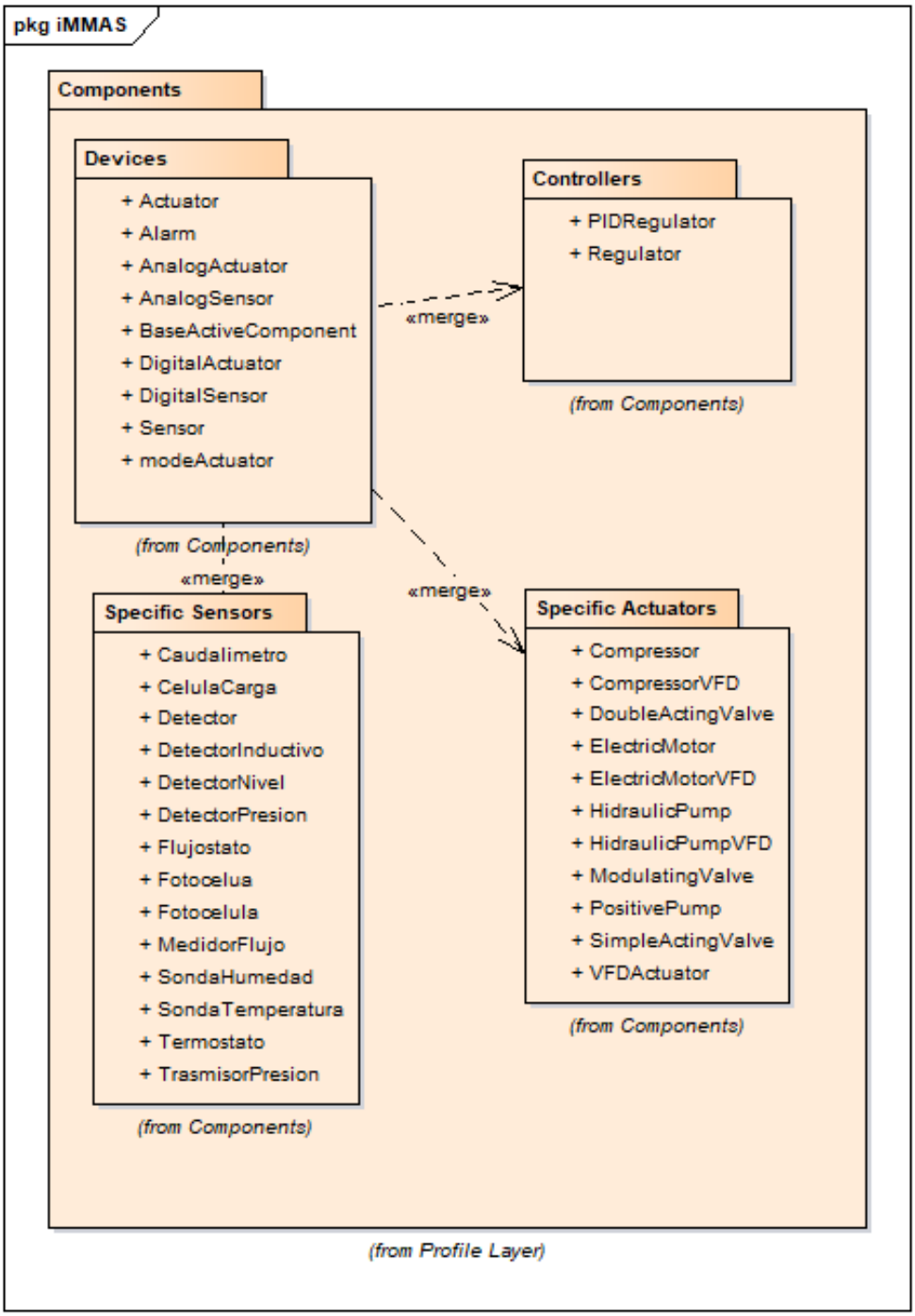


Figura 5.43: Elementos y sub-paquetes que conforma el paquete Components.

EL SUB-PAQUETE DEVICES

Los conceptos principales de este paquete son los actuadores y sensores. Podemos distinguir distintos tipos de actuadores y sensores básicamente en función de los módulos de entradas/salidas, de alarmas o de tipo control que incluyan. En la figura 5.44 se muestra los distintos tipos definidos.

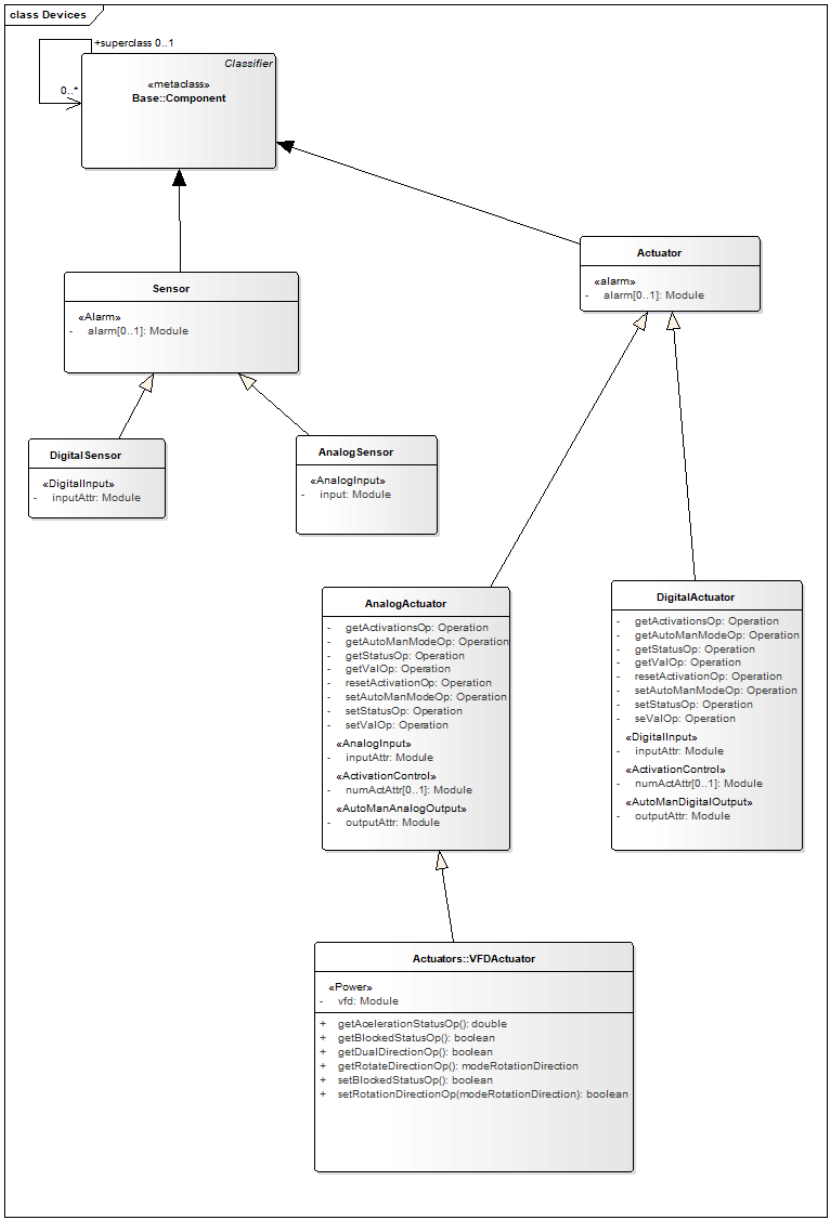


Figura 5.44: Componentes del paquete Device.

- **Sensor.**
  Esta abstracción recoge el concepto de sensor para así modelar cualquier tipo de instrumentación de un sistema industrial que capture información del entorno independientemente de sus características eléctricas, magnitud de medida o tipología. Es un tipo de componente dentro del contexto de iMMAS que incluye como atributo un módulo de tipo alarma *(Alarm[0..1]: Module)* que, cuando se activa, indica que el sensor está en avería o en alarma. Dependiendo del tipo de sensor tendrá asociado un módulo de datos de entrada como se puede ver más adelante.

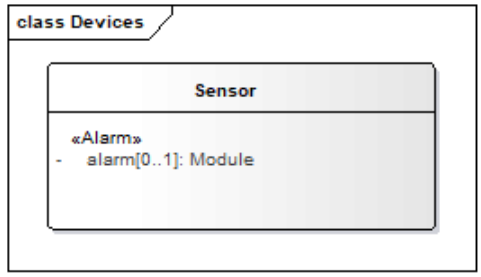


Figura 5.45: Representación del componente Sensor.

- **Actuator.**
  Este tipo de componente abstrae el concepto de actuador de un sistema industrial como aquel elemento industrial capaz de producir una acción que genera un efecto sobre un proceso automatizado independientemente de sus características eléctricas y su tipología. El Actuador en MOSIE es un tipo de componente que incluye adicionalmente como atributo un modulo de datos de tipo alarma para señalar cuando el Actuador se encuentra en avería o en alarma. Dependiendo del tipo de Actuador incluirá distintos módulo de salidas como se puede ver más adelante.

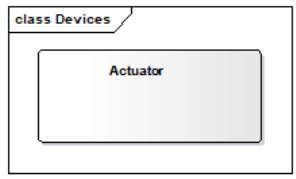


Figura 5.46: Representación del componente Actuator.

- **DigitalSensor.**

  Este componente es un tipo de sensor digital que representa instrumentación digital como transmisores de presión, detectores inductivos, detectores de nivel o presoestantos. Es importante destacar que este componente contiene un módulo de entrada digital *DigitalInput* para almacenar el valor capturado, pero con la capacidad adicional de estar en avería o alarma si se dispara.

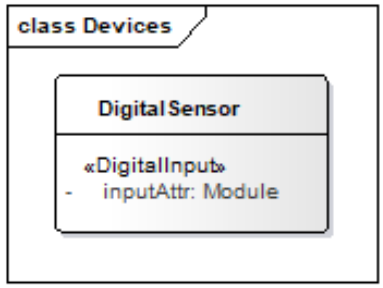


Figura 5.47: Representación del componente DigitalSensor.

- **AnalogSensor.**

  Este componente es un tipo de sensor analógico que representa instrumentación analógica como sondas de temperatura, células de carga o transmisores de presión que miden alguna magnitud física. Para ello requiere un módulo de entrada *AnalogInput* para almacenar la entrada, con la capacidad adicional de estar en alarma o avería.

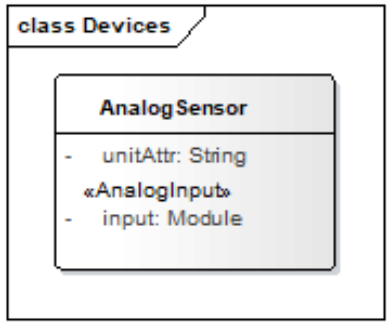


Figura 5.48: Representación del componente AnalogSensor.

- **DigitalActuator.**

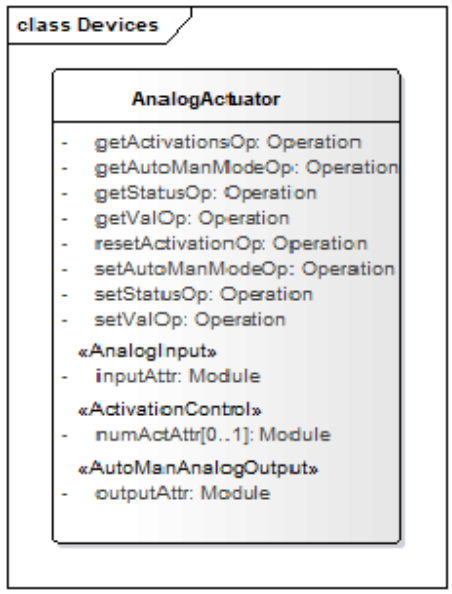


Figura 5.49: Representación del componente DigitalActuator.

El actuador digital es un tipo de actuador para modelar dispositivos como válvulas, cilindros neumáticos o motores marcha/paro. El actuador digital contiene un módulo de salida *AutoManDigitalOutput* para almacenar la orden o señal de salida con dos modos de funcionamiento automático y manual. En el modo automático el dispositivo industrial se encarga de accionar dicho componente, mientras que en el modo manual el componente es accionado por un sistema exterior, SCADA o HMI. El actuador además incluye un módulo de entrada *DigitalInput* para almacenar una entrada del proceso que permita confirmar si la orden ha realizado un accionamiento correcto. Por último, también se incluye el módulo de control *ActivationControl* para contar y resetear el número de activaciones u horas de funcionamiento con el objetivo de poder realizar mantenimientos preventivos del equipo o realizar seguimientos de sus ciclos de trabajo.

- **AnalogActuator.**

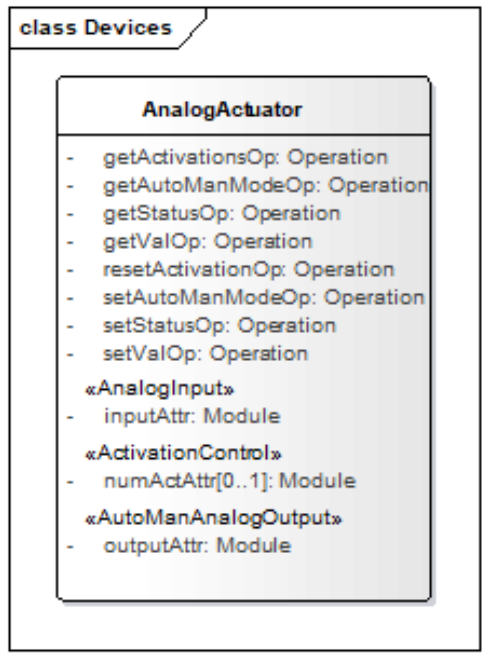


Figura 5.50: Representación del componente AnalogActuator.

AnalogActuator nos permite abstraer un actuador analógico como motores de corriente alterna controlados por variadores, válvulas modulantes o bombas hidráulicas. Todos estos elementos se caracterizan porque tienen la capacidad de regular su velocidad o su grado de apertura en el caso de las válvulas, en función de un valor eléctrico que puede ser proporcionado por varios tipos de señales (0-10 V, -10-10 V, 4-20 mA, etc.), o por tramas de datos en un bus de campo como Profibus, Profinet o Ethercat. Contiene un modulo de salida *AutoManAnalogOutput* en el que se establece la variable de salida o valor de trabajo, y un módulo de entrada *AnalogInput* para almacenar el valor real de funcionamiento del elemento, por ejemplo la velocidad real a la que está el motor trabajando. Por último, incluye un módulo de control *ActivationControl* para gestionar las horas de funcionamiento del sistema y planificar mantenimientos preventivos basados en dichas horas o lo que es lo mismo mantenimientos basados en condiciones.

- **VFDActuator.**

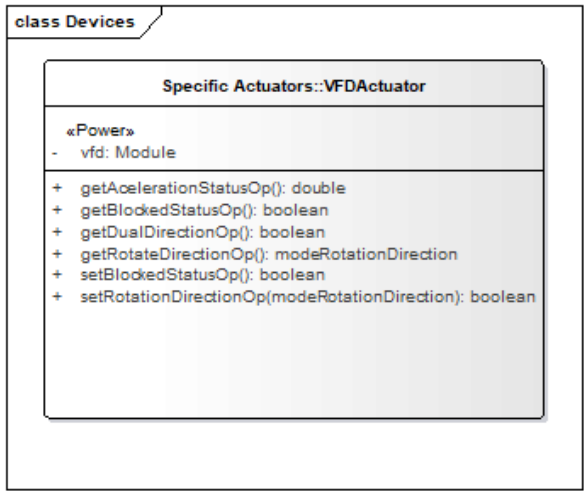


Figura 5.51: Representación del componente activo VFDActuator.

Este elemento representa una abstracción de actuador de tipo analógico más complejo, ya que está pensado para modelar actuadores en los que se conoce el modo de trabajo directo o inverso. Así podremos seleccionar, por ejemplo, el sentido de giro de un motor o si el actuador está acelerando o está bloqueado por la activación de alguna alarma de seguridad. Este componente extiende del concepto de *AnalogActuator*, por lo que directamente posee todos los atributos y operaciones de dicho componente. En este caso, el actuador incluye otro módulo de control de tipo *Power*, que nos proporciona la capacidad de controlar la dirección de funcionamiento en caso de que el dispositivo lo permita, pudiendo así activar el funcionamiento en directo o en inverso, o saber si está bloqueado o está acelerando.

EL SUB-PAQUETE CONTROLLERS

Este paquete Controllers incluye los componentes relacionados con el concepto de regulador o controlador industrial. Los mas extendidos y utilizados en el mundo de la industria son los controladores PID, aunque pueden extenderse a otro tipo de controladores como por ejemplo los basados en lógica difusa. En la figura 5.52 se muestra el conjunto de componentes que contiene dicho paquete.

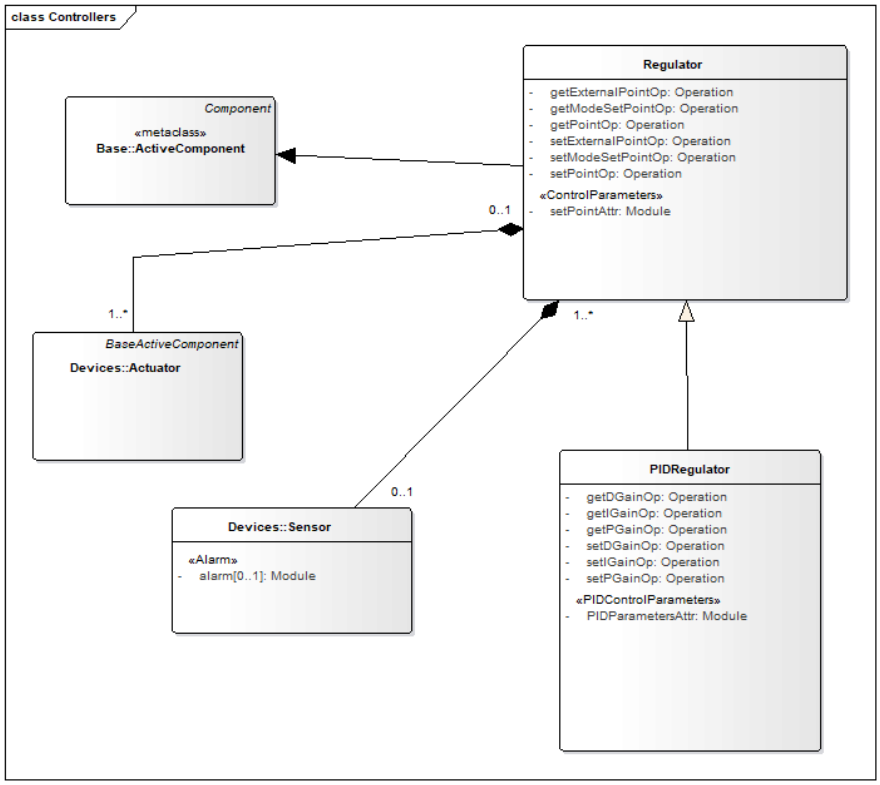


Figura 5.52: Conjunto de componentes del paquete Controller.

- **Regulator.**

  Este componente abstrae el concepto de un regulador simple. Está compuesto por un sensor para recoger el valor de la magnitud a controlar en el sistema controlado y un actuador que acciona ordenes hasta que se alcanza una consigna. Por este motivo, el componente regulador contiene a un componente Sensor y un componente Actuator, independientemente de la naturaleza digital o analógica del sensor o actuador. Además incluye un módulo de control ControlParameters para almacenar los parámetros que definen al regulador. El algoritmo, técnica de control o regulación utilizada para el proceso industrial escapa del alcance de esta abstracción, dejando en manos del ingeniero la técnica a emplear. El comportamiento del regulador se modela mediante la máquina de estados asociado que tendrá que definir el ingeniero.

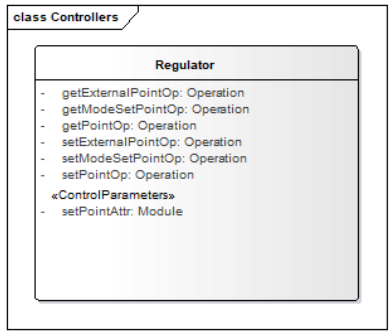


Figura 5.53: Representación del componente Regulator.

- **PIDRegulator.**
  Este componente nos permite modelar un controlador PID clásico para regular un proceso productivo, por lo que ha sido diseñado para recoger todas las características de un controlador PID, o variedades de estos como son PI o PD. Extiende el elemento Regulator y se ha agregado otro módulo de control PIDControlParameters para que se pueda almacenar el valor de las constantes de control Kp, Ki y Kd.

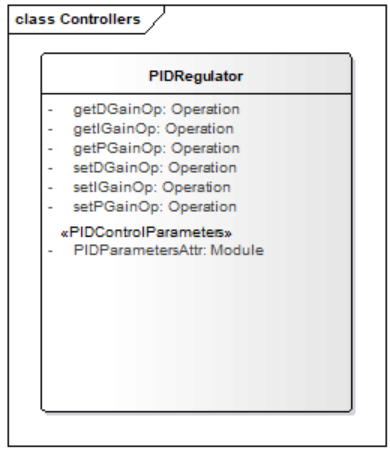


Figura 5.54: Representación del componente PIDRegulator.

EL SUB-PAQUETE SENSORS

El paquete Sensors de MOSIE incluye los sensores analógicos y sensores digitales más utilizados en entornos industriales. Los sensores digitales se pueden dividir a su vez en sensores que emiten una señal eléctrica de detección/no-detección

sin ninguna configuración adicional o sensores que necesitan configurar un valor umbral a partir del cual se dispara la señal de detección/no detección. En la figura 5.55 se muestra el conjunto de componentes de tipo sensor utilizados en MOSIE, aunque puede ser extendido para agregar nuevos dispositivos de sensorización.

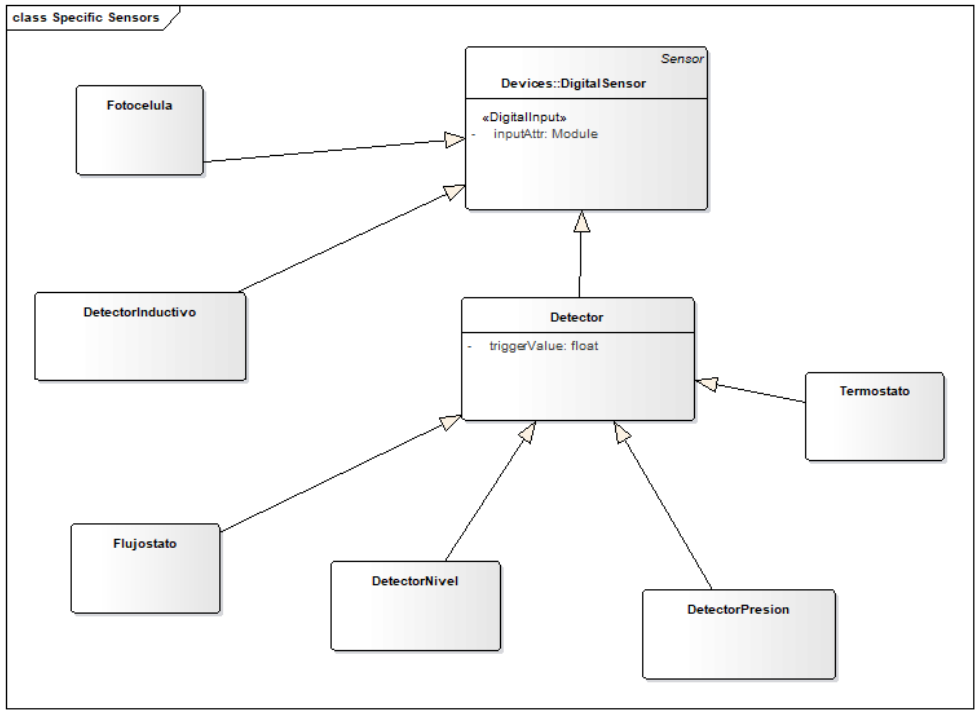


Figura 5.55: Sensores predefinidos a partir del concepto DigitalSensor.

Entre los sensores digitales podemos encontrar:

- Fotocélulas. No necesitan un valor de disparo, y generalmente basta con ajustar la sensibilidad del dispositivo.
- Detectores Inductivos. No necesitan ningún tipo de ajuste.
- Detectores de nivel: Pueden necesitar un ajuste del valor de disparo, depende del tipo de detector, y podemos tener detectores de nivel máximo o mínimo.
- Detectores de Presión: Al igual que los detectores de nivel pueden necesitar un nivel de disparo a partir del cual detectan presión o no.

- Termostato: Este tipo de sensores de temperatura pueden necesitar también un valor de disparo.

- Flujostato: Este tipo de sensores se suelen tarar con un valor de flujo (aire o algún tipo de gas) para disparar la detección o no del sensor.

Como se puede apreciar en la figura 5.55, los sensores digitales que no necesitan ningún tipo de configuración extienden directamente del concepto *DigitalSensor*. Por el contrario los sensores que necesitan algún tipo de tarado en función de alguna magnitud física como son los flujostatos, los detectores de nivel y presión o los termostatos, extienden del concepto detector. Este a su vez extiende del concepto *DigitalSensor* al que se le ha añadido una propiedad *triggerValue*. Este valor es solo informativo ya que el proceso de tarado de este tipo de sensores se hace en el propio dispositivo sin ningún tipo de interacción con sistemas externos, y generalmente esta información tampoco es accesible por ningún elemento industrial, como por ejemplo un PLC, ya que a estos elementos solo les llega la señal digital detección/no-detección.

En cuanto a los sensores analógicos nos podemos encontrar de acuerdo a la figura 5.56 con:

- Caudalímetros: Estos tipos de sensores se utilizan principalmente para medir el caudal de un fluido al pasar por una tubería devolviendo un valor analógico para esta medida.

- Sondas de Humedad: Miden el % de humedad en medios gaseosos o sólidos como por ejemplo la tierra.

- Trasmisores de Presión: Miden la presión de un gas o un fluido a su paso por una tubería.

- Sondas de temperatura: Miden la temperatura de un gas o un fluido en un recipiente cerrado, en una tubería o en el ambiente.

- Células de carga: Miden la presión ejercida sobre ellas de forma que se pueden utilizar como elementos para medir la presión ejercida en un punto. Como dicha presión puede ser transformada en peso también se pueden

utilizar para medir, por ejemplo, el peso de tanques de almacenaje de líquidos.

- Medidores de Flujo: Mide el flujo de un gas al pasar por una tubería.

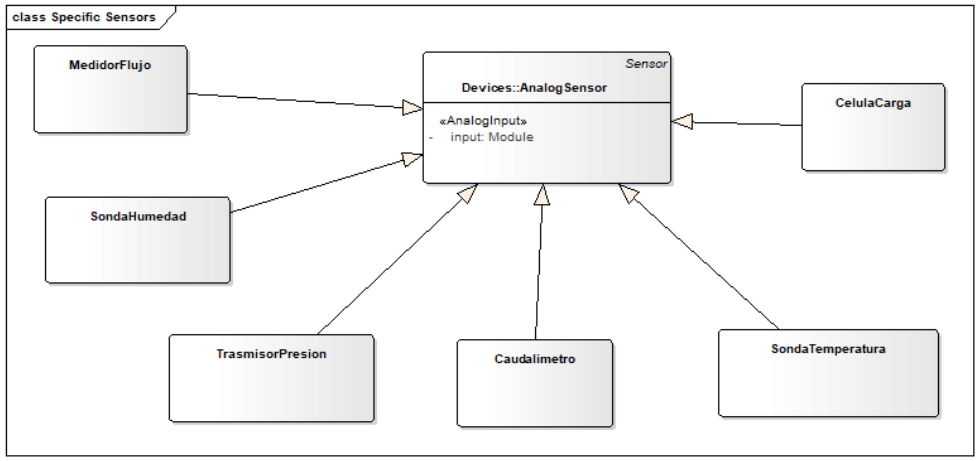


Figura 5.56: Sensores predefinidos a partir del concepto AnalogSensor.

EL SUB-PAQUETE ACTUATORS

Al igual que ocurre en el caso de los sensores, los actuadores los podemos dividir en digitales y analógicos dependiendo de las señales necesarias para enviar órdenes a estos elementos. Obviamente el conjunto de actuadores analógicos es más complejo que el conjunto de los digitales, y sus caracteristicas principales se encuentran recogidas en el concepto *AnalogActuator*. Los actuadores digitales que se han predefinido son:

- Válvula de simple efecto: Este actuador necesita solo una orden de apertura/cierre. Cuando la orden deja de aplicarse vuelve a su posición de reposo.

- Válvula de doble efecto: Este tipo de actuador necesita dos ordenes, una de apertura y otra de cierre. Mientras no ocurra alguna de ellas este elemento no realiza ninguna acción.

- Motor Eléctrico: Este actuador consiste básicamente en un motor eléctrico que solo tiene una orden de marcha y una orden de paro.

- Compresor: Los compresores son actuadores que "generan.ªire comprimido y lo introducen en una instalación neumática. Para estos elementos solo se dispone de una orden de marcha y otra de paro.

- Bomba Hidráulica: Este actuador bombea un fluido a través de una instalación hidráulica, y solo tiene orden de marcha o paro.

Como podemos ver en la figura 5.57 todos estos actuadores digitales extienden del concepto *digitalActuator*. No es necesario añadirles ningún atributo adicional excepto a *DoubleActingValve*. En este caso puede haber variantes formadas por dos *SimpleActivationValve*, aunque también pueden existir válvulas de 3 y 4 posiciones para las cuales tendríamos que utilizar 3 *SimpleActivationValve*.

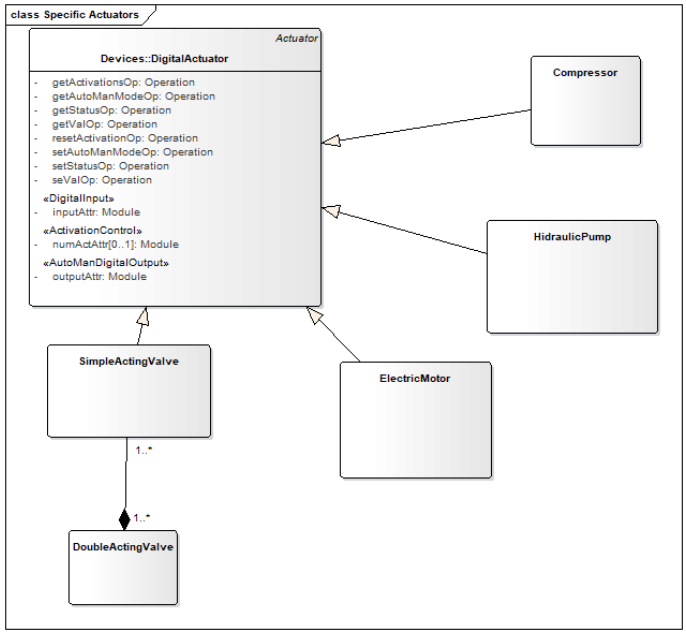


Figura 5.57: Actuadores predefinidos que extiende del concepto DigitalActuator.

Los actuadores analógicos que se han predefinido son:

- Válvula modulante: Este tipo de actuador analógico consiste básicamente en una válvula que varía su grado de apertura/cierre en función de un valor analógico.

- Compresor de velocidad variable: Esta actuador tiene las mismas características de un compresor "digitalçon arranque y paro, pero con la salvedad

de que se puede cambiar la velocidad del mismo, presión/caudal de aíre comprimido, en función de un valor analógico.

- Bomba Hidráulica de velocidad variable: Este actuador bombea un fluido a través de una instalación hidráulica, de forma que la velocidad de bombeo del fluido es regulada por un valor analógico.
- Motor Eléctrico de velocidad Variable (variadores de frecuencia): Este tipo de motor eléctrico es capaz de producir movimiento en función de un valor analógico. Normalmente este valor analógico, que representa la velocidad del motor, se envía a un variador de frecuencia y este regula la frecuencia eléctrica del motor y, por tanto, la velocidad del mismo.

Como podemos ver en el modelo 5.58 todos estos actuadores analógicos extienden del concepto *analogActuator* en el caso de las válvulas modulantes y los compresores, y de *VFDActuator* en el caso de los motores eléctricos y de las bombas hidráulicas, ya que estos últimos poseen, entre otras cosas, el estado de aceleración y el sentido de giro. Estos elementos generalmente llevarán asociado un variador de frecuencia que les proporcionará estas características.

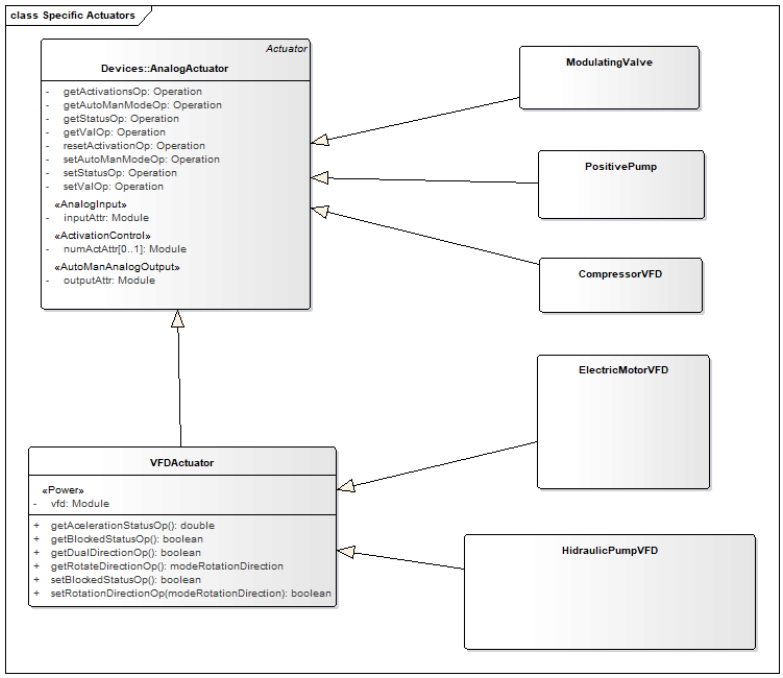


Figura 5.58: Sensores predefinidos a partir del concepto AnalogActuator.

EJEMPLO DE MODELADO CON MOSIE

A partir del perfil MOSIE se puede hacer una nueva remodelación del ejemplo de sistema de dosificación líquida que se vio anteriormente aprovechando las abstracciones que proporciona MOSIE. El nuevo modelo realizado para este sistema es el siguiente:

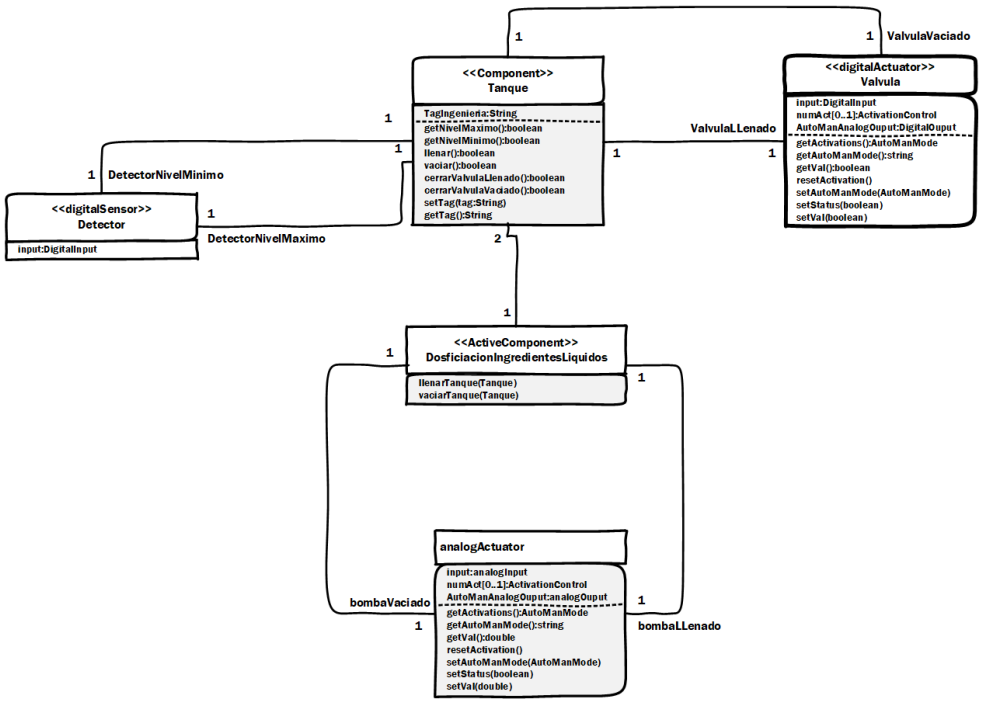


Figura 5.59: Modelado del sistema de dosificación líquida utilizando MOSIE

Como se puede ver en la figura 5.59 el modelo es muy similar al realizado con iMMAS. El componente principal *DosificacionIngredientesLiquidos* y el componente Tanque siguen siendo los mismos y apenas han sufrido cambios. Sin embargo tanto las bombas como los detectores se han podido modelar directamente con conceptos de MOSIE pensados para estos elementos como *analogActuator*, *digitalSensor* y *digitalActuator*. Por tanto, no se han tenido que construir módulos específicos para recoger las características de estos elementos, como en el ejemplo anterior, facilitando así las tareas de modelado y ofreciendo un nivel de abstracción mayor. Para el modelado del comportamiento del sistema podemos seguir utilizando las maquinas de estados que se construyeron anteriormente, ya que como se puede ver en la figura anterior *DosificaciónIngredientesLiquidos* si-

gue siendo un elemento de tipo *ActiveComponent*, y sigue siendo el elemento que gestiona todo el sistema de dosificación.

Empleando los conceptos de MOSIE hemos realizado un modelo mucho más simple y compacto que el realizado con los conceptos de iMMAS, y además con un nivel de abstracción mayor. En este caso no hemos tenido que crear nuevos conceptos, lo que nos permite reducir el tiempo de modelado además de no tener que crear módulos adicionales que alberguen datos de tipo *industrialData*. Aunque puede parecer menos flexible que un modelo diseñado ad-hoc al tener que utilizar conceptos predefinidos con una semántica determinada, el valor de MOSIE es justamente disponer de un conjunto de abstracciones organizados y bien documentados que pueden ser utilizados para la descripción de sistemas industriales. De este modo se facilita la comunicación entre los miembros del equipo de desarrollo y entre los ingenieros OT y los de IT de las fábricas, y se facilita el mantenimiento y evolución de los modelos.

En este sentido es el propio ingeniero el que puede decidir en que nivel desea trabajar, pero siempre que se pueda es recomendable trabajar con MOSIE por las ventajas que este conjunto de perfiles ofrece: mayor nivel de abstracción, documentación de dichas abstracciones, reducción de tiempos de modelado, y posibilidad de adaptar y extender conceptos ya existentes.

5.7. CONCLUSIONES

Para poder utilizar MDE como paradigma para el desarrollo de software en entornos industriales, es necesario poder crear modelos a partir de un metamodelo o lenguaje de modelado especialmente diseñado y pensado para el dominio de las aplicaciones industriales. iMMAS proporciona un vocabulario con conceptos especialmente diseñados y construidos para sistemas software industriales, con una semántica muy similar con la que se comunican los ingenieros en las plantas industriales. En un segundo nivel, MOSIE ofrece un perfil que agrupa los diferentes conceptos que normalmente puede encontrar un ingeniero en un sistema industrial. Dichos conceptos se han construido sobre el vocabulario y lenguaje de iMMAS, lo que simplifica el modelado de sistemas industriales.

La combinación iMMAS/MOSIE no pretende ofrecer una solución completa de la multitud de elementos y dispositivos industriales que puede encontrarse en

una planta y en diferentes ámbitos industriales, como puede ser la industria de la automoción o la petro-química. iMMAS/MOSIE no esta pensado para ser una aproximación de desarrollo dirigido por modelos cerrado, si no todo lo contrario, ofrece la suficiente flexibilidad para crear nuevos elementos, tanto por extensión de otros ya existentes como por construcción partiendo desde cero, siempre y cuando estos nuevos elementos sean conformes a las especificaciones de iMMAS. Es decir, iMMAS/MOSIE pretende proporcionar un lenguaje para sistemas software industriales, en constante evolución, con el objetivo de cubrir el mayor número de sistemas software industriales posible.

Los modelos diseñados con iMMAS definen lo que en MDA se conoce como PIM o modelos que describen sistemas industriales independientes de la plataforma. Estos modelos pueden desplegarse en plataformas industriales como en PLCs o en servidores OPC UA, lo que se denomina dentro del contexto de MDA como modelos específicos de la plataforma o PSMs.

En los próximos capítulos se detallará como a partir del modelo PIM desarrollado en iMMAS se pueden generar los modelos PSMs mediante un conjunto de transformaciones, que pueden desplegarse en dispositivos industriales PLCs y en sistemas OPC de intercambio de datos, en el contexto de las arquitecturas jerárquicas de los sistemas industriales. Posteriormente se detallará la nueva metodología propuesta sobre iMMAS, que nos permitirá el desarrollo sistemático de procesos industriales siguiendo un conjunto de etapas muy bien definidos.

6

DESPLIEGUE DE iMMAS EN SISTEMAS SOFTWARE INDUSTRIALES

"Software is a gas: this expands to fill its container."
Nathan Myhrvold.

6.1. INTRODUCCIÓN

La modelización de sistemas industriales nos permite conceptualizar dichos sistemas determinando los elementos esenciales que los componen, sus relaciones, y su comportamiento subyacente [130]. En este sentido, iMMAS ofrece un lenguaje de modelado que facilita su descripción y el perfil MOSIE una estructuración del conjunto de abstracciones que pueden ser utilizados para el desarrollo de modelos para sistemas industriales. Como resultado de dicha combinación iMMAS/MOSIE se obtienen modelos de sistemas industriales que representan de forma esquemática los elementos industriales que componen dichos sistema.

Estos modelos deben poder implementarse dentro de la arquitectura jerárquica que habitualmente tienen los sistemas industriales. El modelo diseñado con iMMAS no es ejecutable, pero si puede ser transformado a otros modelos que se pueden ejecutar en determinados elementos hardware y software de la arquitectura jerárquica del sistema industrial. Dentro de contexto de MDA, esto significa que los modelos más abstractos de iMMAS independientes de la plataforma (PIM, platform independent model) pueden transformarse en otros modelos específicos que dependen de una plataforma concreta (modelos PSM, platform

specific model). Al depender de la plataforma no son tan abstractos, y pueden implementarse o desplegarse directamente en dichas plataformas.

En este capítulo se estudiará cómo a partir de los modelos PIM desarrollados con iMMAS y MOSIE se pueden generar los modelos PSM para OPC UA, que puede desplegarse en un servidor OPC. Para ello, es necesario conocer cómo se llevan a cabo las transformaciones entre el modelo PIM y el modelo PSM para que pueda generarse automáticamente. En esta tesis, dichas transformaciones se han realizado buscando las correspondencias que hay entre los metamodelos del modelo PIM al modelo PSM que se quiere obtener. En este capitulo se realiza un análisis de cómo deben llevarse a cabo la correspondencia de conceptos y relaciones entre los dos lenguajes de modelado o metamodelos.

6.2. DESPLIEGUE DE iMMAS EN UN SERVIDOR OPC UA

Los servidores OPC dentro del contexto de la arquitectura jerarquizada de un sistema industrial facilitan el intercambio de información entre dispositivos industriales y los sistemas software que monitorizan y supervisan un proceso industrial. Por tanto, el despliegue de un servidor OPC requiere conocer muy bien los datos que se tienen que intercambiar entre dichos sistemas. El modelo PIM diseñado en iMMAS puede utilizarse para generar el modelo de información PSM que requieren los servidores OPC, para lo cual tenemos que transformar el modelo de iMMAS al modelo de información de OPC UA.

La correlación realizada entre iMMAS y OPC UA sigue un esquema jerárquico, lo que permite simplificar todo el proceso de transformación entre iMMAS y el modelo de información de OPC UA, simplificando el número y cantidad de relaciones entre los diferentes conceptos recogidos en dicha transformación. Esta organización jerárquica nos ofrece una serie de ventajas inherentes como son encapsulación, modularidad y cohesión[111]. La utilización de esquemas jerárquicos además son compatibles con los entornos industriales, en los que los elementos, conceptos o componentes de un sistema software industrial se encuentran organizados de forma jerárquica tal y como se recoge en estándares como la norma ISA 88, donde la organización depende de su nivel de abstracción *Enterprise*, *Site*, *Area*, *ProcessCell*, *Unit*, *EquipmentModule* y *Control Module* [15]. Por lo tanto, utilizar una estrategia jerarquizada a la hora de realizar la correspondencia

entre iMMAS y OPC UA, aparte de facilitar todo el proceso de transformación, se alinea con la forma de organizar y estructurar un sistema software industrial.

El estándar OPC UA ofrece un modelo de información con un vocabulario rico y una semántica bien definida que puede ser utilizado para crear modelos de intercambio de datos, lo que simplifica la gestión de dichos datos [131] [4] [132]. En el capítulo 3 se explicaron los aspectos más significativos del modelo de información de OPC UA. En la figura 6.1 se muestran los principales conceptos en los que se basa el metamodelo o lenguaje de modelado del modelo de información de OPC UA. El elemento sintáctico más importante es BaseNode, que representa al concepto de nodo, del que derivan el resto de conceptos de OPC UA como *Object*, *ObjectType*, *Variable*, *VariableType*, *Reference*, *ReferenceType*, entre otros. Por lo tanto, OPC UA estructura el modelo de información en torno al concepto de nodo, y modela los sistemas industriales en base a un conjunto de nodos y referencias entre nodos, lo que define el modelo de espacio de direcciones de OPC UA.

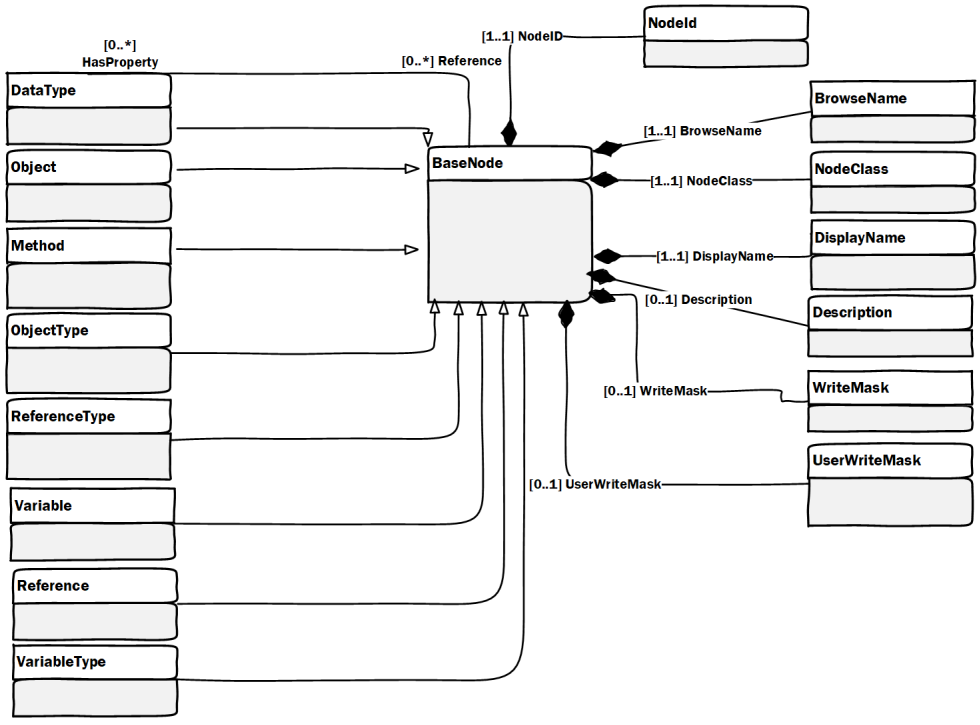


Figura 6.1: Conceptos Básicos del modelo de Información de OPC UA.

Para establecer la correlación entre iMMAS y el modelo de información de OPC UA, se han estudiado las posibles correspondencias que hay entre la sintaxis de iMMAS y la sintaxis del modelo de información de OPC UA. Ambos lenguajes tienen algunos conceptos similares al estar inspirados en lenguajes como UML [133], aunque también hay diferencias sutiles entre algunos de sus conceptos. En la figura 6.2 se pueden encontrar algunas equivalencias que hay entre iMMAS y OPC UA, que se detallará en la siguiente sección.

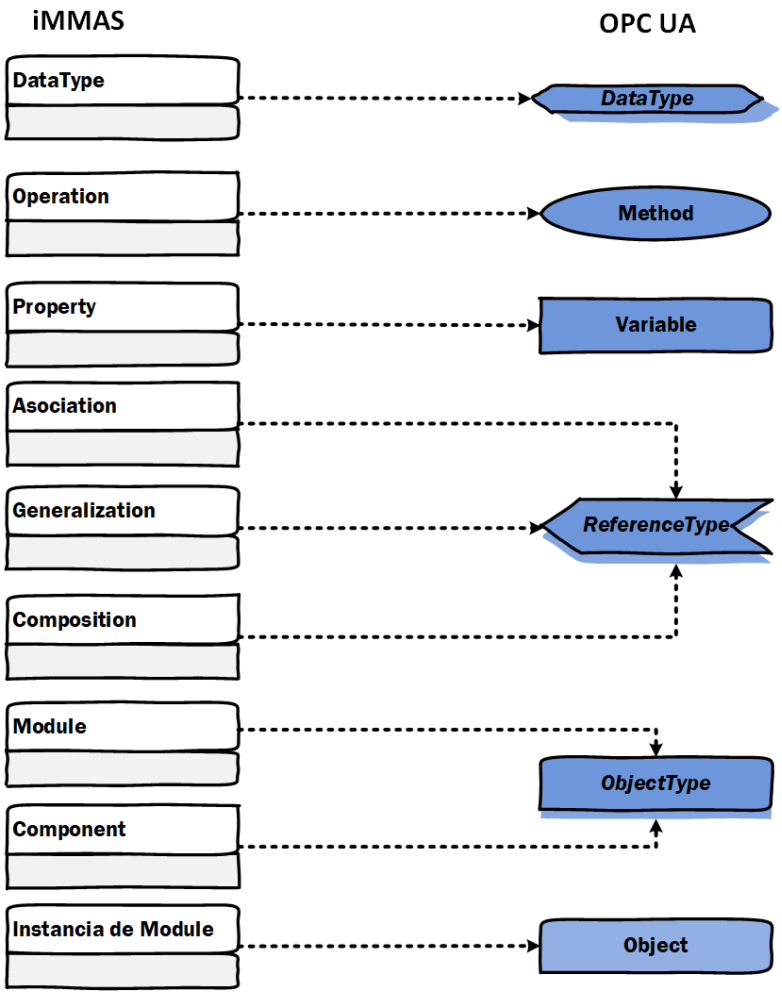


Figura 6.2: Equivalencias entre iMMAS y el modelo de información de OPC UA.

Al transformar los modelos iMMAS a OPC UA no sólo se obtiene como resultado un modelo equivalente PSM en OPC UA, sino que además se encaja en el

modelo del espacio de direcciones de OPC UA, lo que por un lado nos facilita su despliegue en servidores OPC UA y por otro lado lo hace accesible a cualquier cliente OPC UA. Es decir, se genera un modelo de información del servidor OPC UA ”por construcción”. Por tanto, esto significa que cualquier modelo PIM que sea correcto en iMMAS, realizando las transformaciones correspondientes a OPC UA, nos permitiría conseguir un modelo PSM correcto "por construccion.[en] OPC UA. En las siguientes secciones se estudiará con detalle cómo se llevó a cabo el proceso de transformación entre modelos y el despliegue sobre servidores OPC UA.

6.3. REGLAS DE TRANSFORMACIÓN DE IMMAS A OPC UA

Los cimientos conceptuales del lenguaje de modelado iMMAS se encuentran organizados en paquetes: *Type*, *Base*, *Packages* y *Behavior*. En cambio OPC UA tiene definido todos los conceptos dentro del modelo del espacio de direcciones, separando las definiciones de las instanciaciones. Las reglas de transformación entre modelos de iMMAS y OPC UA se han obtenido después de haber seguido una estrategia en dos etapas:

- Identificación directa de elementos sintácticos entre los dos lenguajes.
- Definición de los elementos sintacticos en OPC UA equivalentes a iMMAS

En la primera etapa se han identificado los conceptos y elementos sintáctico que son similares en cuanto a semántica en los dos lenguajes, tratando de determinar su correspondencia directa en el caso que sea posible, y evaluando posibles diferencias semánticas.

En la segunda etapa se han definido todos aquellos conceptos y elementos sintácticos de iMMAS en OPC UA dentro del contexto del modelo de información de OPC UA siguiendo una estrategia jerárquica. Es decir, primero se han definido los elementos sintácticos más básicos y luego se han definido los más complejos de mayor nivel de abstracción. Como consecuencia de la aplicación de las dos etapas se han obtenido las reglas de transformación que habrá que aplicar sobre los modelos PIM de iMMAS para transformarlos a modelos PSM de OPC UA.

La transformación de un modelo PIM a PSM se aplica en tres fases:

- Identificación de los elementos sintácticos de iMMAS contenidos en un modelo PIM.
- Transformación directa de cada uno de los elementos sintácticos de iMMAS a OPC UA.
- Modelado final de cada uno de los elementos sintácticos del modelo PIM identificado en el modelo PSM.

En la primera fase se deben identificar todos los elementos sintácticos contenidos en el modelo PIM, para luego buscar la transformación más adecuada en cada caso. Esto implica buscar los *componentes*, *componentes activos*, *módulos*, *industrialdata* que hay en el modelo PIM.

En la segunda fase se busca para cada elemento sintáctico del modelo PIM, cuál es la transformación que debe realizarse a OPC UA. Por ejemplo, si una clase Valvula de tipo Componente se ha identificado en el modelo PIM, se selecciona la regla de transformación del componente.

En la tercera fase se modela el elemento sintáctico del modelo PIM en el modelo PSM final teniendo en cuenta la regla de transformación más apropiada aplicada para dicho elemento sintáctico. Por ejemplo, se modela la Valvula en OPC UA teniendo en cuenta la regla de transformación del componente.

6.3.1. REGLAS DE TRANSFORMACIÓN DIRECTAS

Dentro del primer grupo podemos considerar los conceptos de iMMAS que tienen una definición equivalente directo en OPC-UA según se muestra en la tabla 6.1.

Concepto de iMMAS	Concepto de OPC UA	Correpondencia
DataType	BaseDataType	Directa
Operation	Method	Directa
Parameter	Argument	Directa
Property	Variable	Directa
Value	Variable	Directa
Address	Property de tipo String	Directa

Tabla 6.1: Resumen de las correspondencias básicas entre el paquete Base de iMMAS y OPC UA.

De los conceptos equivalentes de iMMAS con OPC UA es importante tener en

cuenta algunas diferencias semánticas entre algunos conceptos. El concepto *Property* de iMMAS hace referencia a atributos que pueden tener los componentes y que pueden variar a lo largo de la vida del sistema, y su correspondencia directa sería la Variable en OPC UA. El concepto *Value* de iMMAS de igual modo se corresponde con una *Variable* que tiene un valor de tipo *DataType*. En cambio el concepto *Address* equivale a una *Property* de tipo *string* en OPC UA. En OPC UA las propiedades *Property* reflejan los atributos de tipo constantes implícitas en el modelo, y que no cambian durante la evolución del sistema. Esto es justamente la semántica que tiene *Address*, dado que una vez que se define la referencia de memoria no cambia durante la evolución del sistema.

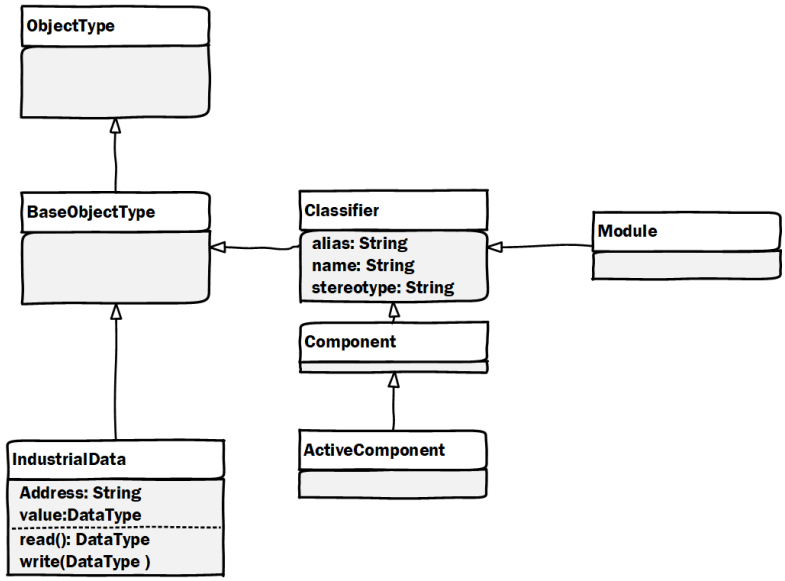


Figura 6.3: Correlación entre los conceptos de iMMAS y el modelo de información de OPC UA.

6.3.2. DEFINICIÓN DE ELEMENTOS SINTÁCTICOS DE iMMAS EN OPC UA

Para el conjunto de conceptos definidos en el paquete Base como Classifier, Module, *Component*, *ActiveComponent* e *IndustrialData* no existe una correspondencia directa con OPC UA por lo que se debe acudir a realizar definiciones en OPC UA para poder trabajar con el mismo elemento conceptual. Desde el punto de vista de OPC UA los conceptos anteriores de iMMAS se corresponden con nodos de tipo *ObjectType*, tipo de objeto en OPC UA. El modelo de información

de OPC UA proporciona el tipo de objeto *BaseObjectType* como especialización de *ObjectType* para definir nuevos tipos de objetos. En la figura 6.3 podemos ver cómo se han definido los conceptos elementales de iMMAS a partir de *BaseObjectType*. A partir del concepto de OPC UA *BaseObjectType* se ha creado el concepto de *Classifier* mediante herencia, por lo que hereda todos los atributos y operaciones definidas en el concepto *BaseObjectType* de OPC UA. Este proceso se repite recurrentemente para todos los conceptos que a su vez derivan de *Classifier*, como son *Component*, *Module* y *ActiveComponent* el cual a su vez es un concepto derivado de *Component*. El concepto *IndustrialData* se deriva directamente de *BaseObjectType* para la definición de un nuevo tipo de objeto en OPC UA. De forma más detallada los conceptos anteriores se definen explícitamente a continuación.

Classifier:

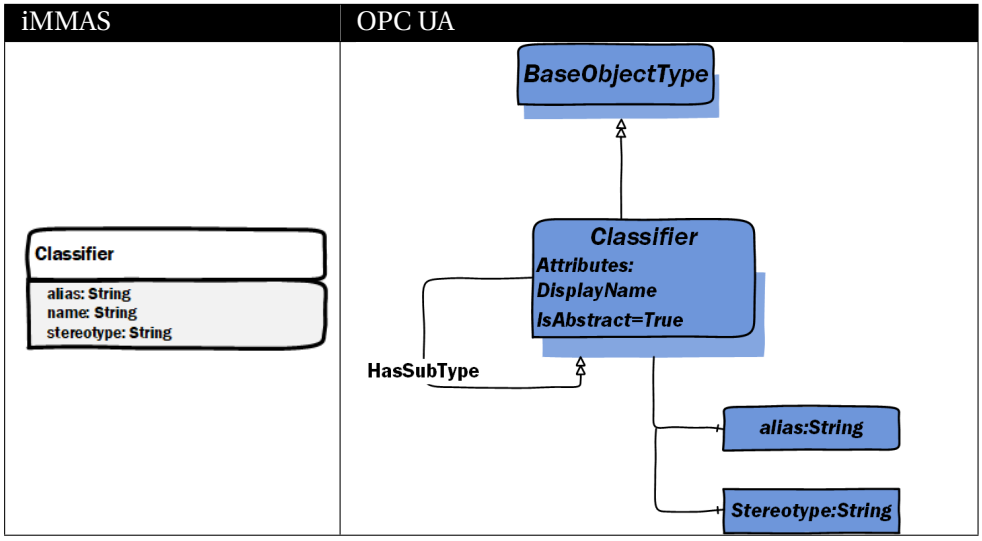


Tabla 6.2: Transformación realizada para Classifier.

El concepto abstracto *Classifier* se ha definido en OPC UA como un nuevo concepto *Classifier* de tipo *ObjectType* que es subtipo de *BaseObjectType*. Contiene como atributos dos variables *alias* y *stereotype*. Por otra parte tiene la propiedad *isAbstract* definida a true, ya que es un elemento abstracto que no se puede instanciar. Además contiene las propiedades que se heredan de *BaseObjectType*

como *Browsename* y *DisplayName*, por ser un nodo representable en OPC UA. *DisplayName* contiene el nombre del clasificador y se corresponde con el atributo name del *Classifier* de iMMAS. Un clasificador puede tener una relación de generalización con uno o más clasificadores (módulos y/o componentes) en iMMAS. Esta relación se corresponde en OPC UA a una relación de tipo *hasSubtype* entre los dos clasificadores.

IndustrialData:

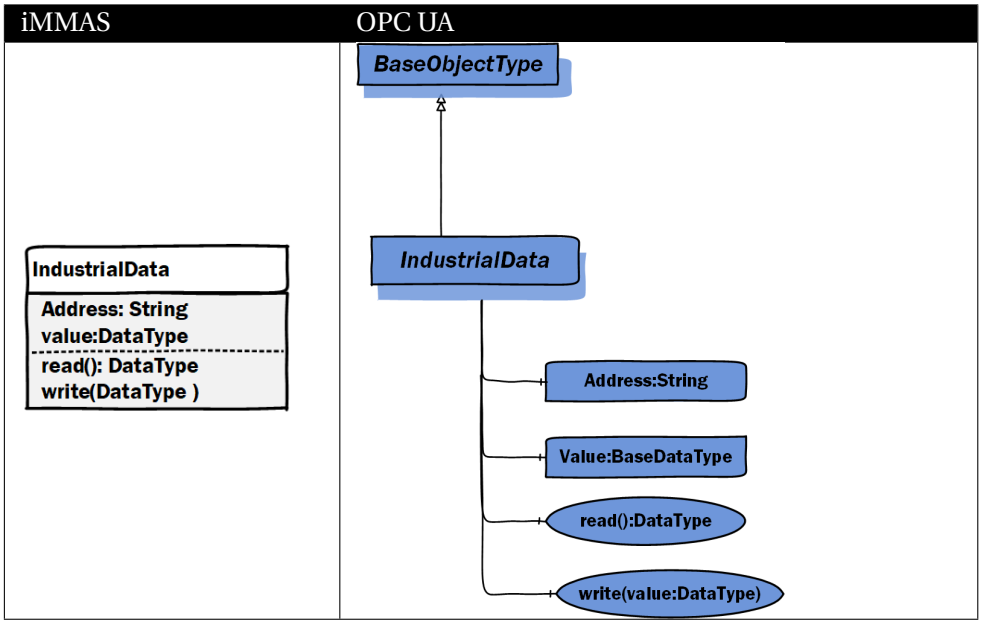


Tabla 6.3: Transformación realizada para IndustrialData.

IndustrialData es una abstracción de iMMAS con la que se encapsula el dato que se almacena persistententemente en el sistema, así como la dirección de memoria o la referencia en memoria en la que se encuentra. Para la transformación se puede definir en OPC UA un *ObjectType* para recoger el concepto de *IndustrialData* con la variable Value para representar el valor del Dato Industrial y la propiedad Address para representar la referencia en memoria que es inmutable. El valor Value viene definido por un tipo genérico *BaseDataType* que puede contener tipos de datos que puede manejar un dispositivo industrial como *Float, int16,int32, Double, Time, Counter, Char, Boolean, String*, mientras que *Address* viene definida por una cadena de tipo *String* que hace referencia a una dirección

fija. Por defecto tiene definido implicitamente dos operaciones o Method en OPC UA que nos permite leer el contenido de *Value*, o modificarlo.

Module:

Para el concepto Module en OPC UA se ha creado un *ObjectType* como subtipo del tipo base *Classifier* creado anteriormente, tal y como específica iMMAS, y se ha agregado una referencia de tipo *HasComponent* con la propiedad *IsOneWay* a true con el concepto *IndustrialData*. Dado que la referencia en OPC UA en un *ObjectType* implica una cardinalidad de al menos 1, se mantiene la semántica de iMMAS de que un módulo puede contener uno o más objetos de tipo *IndustrialData*.

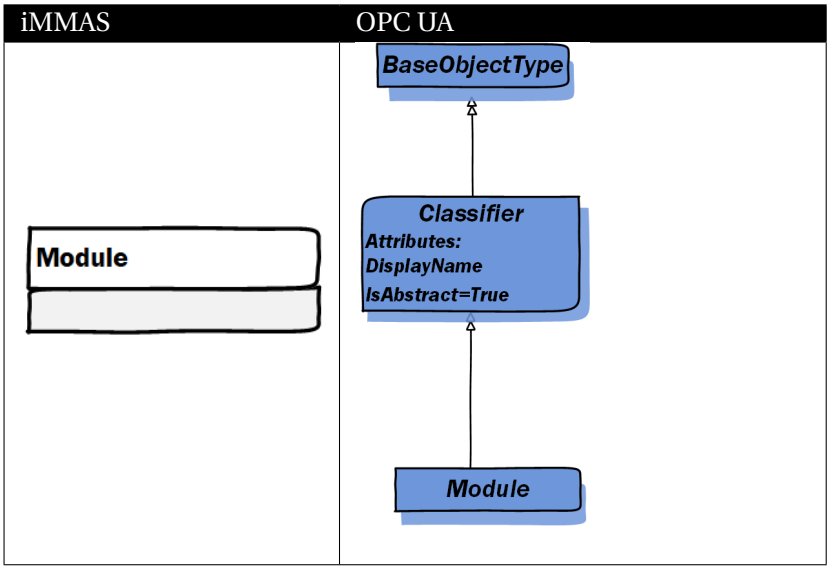


Tabla 6.4: Transformación realizada para Module.

Component:

Para el concepto *Component* de IMMAS de forma similar a Module se ha definido un *ObjectType Component* como subtipo de *Classifier* por lo que hereda sus variables y propiedades. En este caso concreto, la propiedad *isAbstract* es igual a false, ya que si son instanciables.

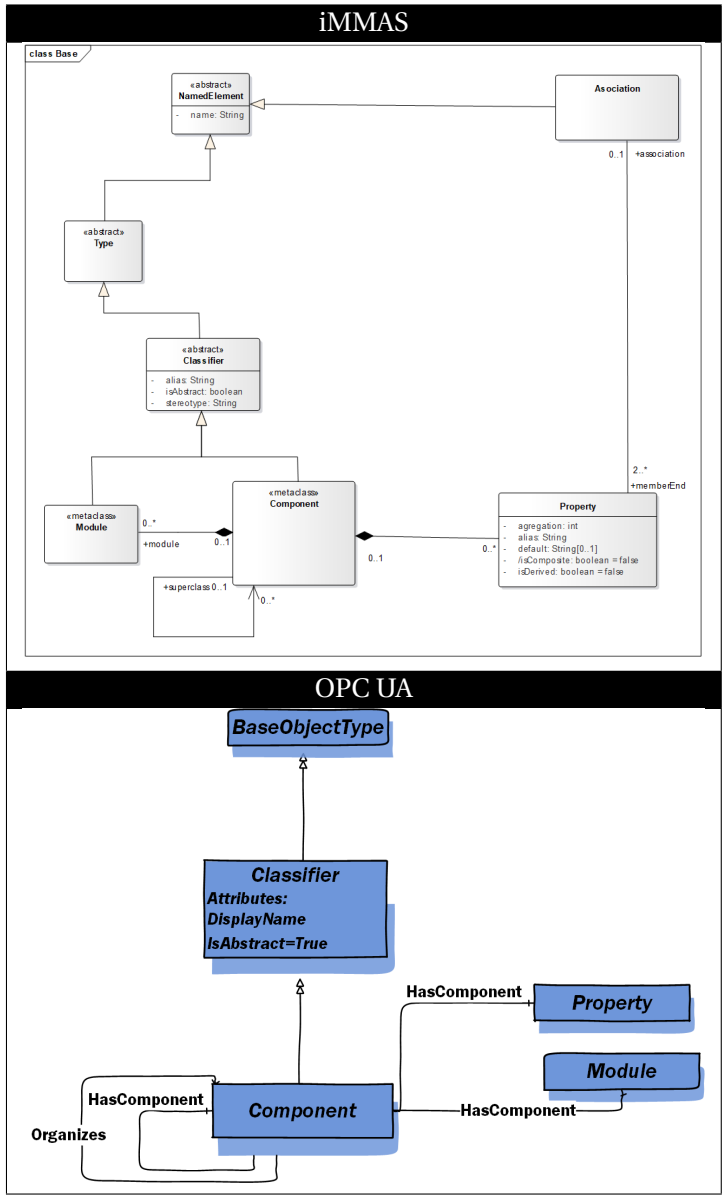


Tabla 6.5: Transformación realizada para Component.

Un componente puede contener uno o más módulos por lo que aparece una relación *ReferenceType* de tipo *HasComponent* de tipo *ReferenceType* con respecto al *ObjectType* Module definido anteriormente. En OPC UA se admite que no se especifique la cardinalidad especifica del *NodeClass* target cuando se hace una definición de *ObjectType*. En cambio en la instanciación de los *ObjectType* en *Ob-*

jects si tiene que estar definido. Por tanto, las relaciones hay que entender que tienen una semántica al menos 1.

Un componente puede contener una o más propiedades en iMMAS, por lo que en OPC UA esto aparece como una relación *ReferenceType* de tipo *hasComponent* con una variable *Property*. No confundir la variable *Property* con el tipo *Property*, ya que en OPC UA que tiene una semántica diferente tal y como se ha explicado anteriormente. Un componente puede tener una relación de asociación con uno o mas componentes en iMMAS. Estas relaciones pueden ser de tipo asociación, agregación o composición. En este caso, dichas relaciones se corresponden con *ReferenceType* de tipo *organizes*, *aggregates* y *hasComposition*, respectivamente. En la figura3.17 del capitulo 3 se muestran los tipos de relaciones que se admiten en la transformación de iMMAS a OPC UA.

ActiveComponent:

ActiveComponent es un concepto que deriva de Component en iMMAS, por lo que en OPC UA de igual modo este concepto tiene como tipo base Component.

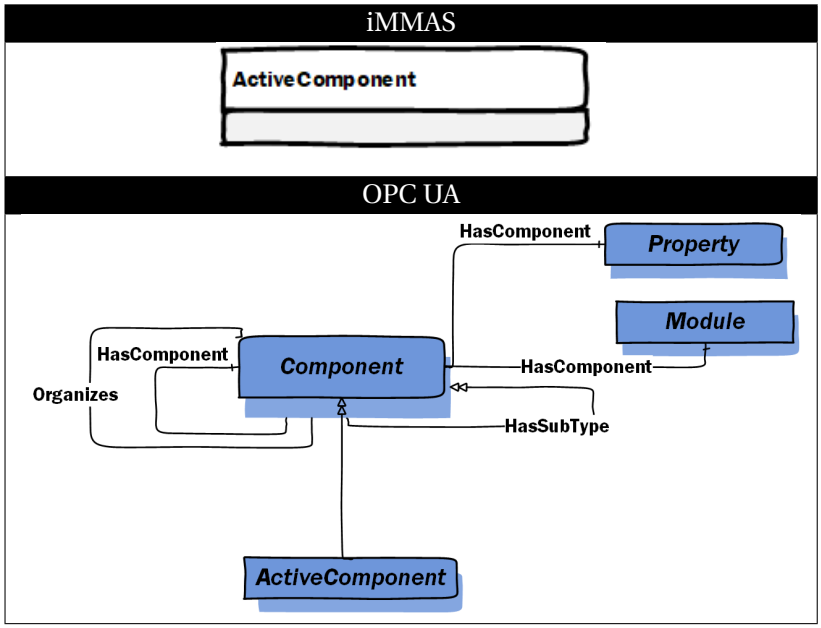


Tabla 6.6: Transformación realizada para ActiveComponent.

Package:

Este paquete está formado por un solo concepto, el concepto *Package*, el cual está pensado para agrupar conceptos similares dentro de él. En el modelo de información de OPC UA tenemos el concepto *FolderType*, el cual está pensado para organizar los elementos del espacio de direcciones de un servidor OPC UA de forma jerárquica. Los elementos pertenecientes a un paquete se incluyen utilizando la referencia *Organizes*.

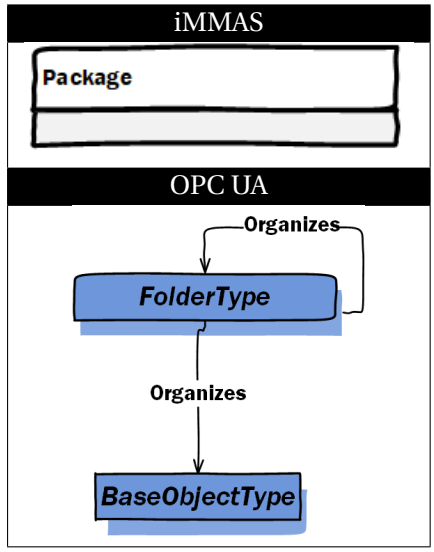


Tabla 6.7: Transformación realizada para Package.

DataType en OPC UA:

El paquete *Type* de iMMAS contempla tanto los tipos básicos de datos de iMMAS, *Double*, *int*, *boolean* y *String* como los tipos enumerados que se pueden definir dentro del contexto del concepto *DataType*. La transformación en OPC UA es inmediata ya que el modelo de información de OPC UA contempla estos tipos de datos y otros adicionales. En nuestro caso, solo incluimos la transformación directa de los cuatro tipos primitivos que iMMAS recoge como básicos, como se puede ver en la tabla 6.8

Concepto de iMMAS	Concepto de OPC UA	Correpondencia
Int	Integer	Directa
Double	Double	Directa
Boolean	Boolean	Directa
Char	String	Directa

Tabla 6.8: Resumen de las correspondencias básicas entre los tipos de iMMAS y OPC UA.

El paquete Behavior en OPC UA:

El paquete *Behavior* de iMMAS incluye los conceptos "básicos" para poder crear máquinas de estados que modelen el comportamiento de elementos industriales. Fundamentalmente los más importantes son los conceptos *StateMachine*, *State* y *Transition*. Estos pueden trasladarse directamente al modelo de información de OPC UA con los conceptos *FiniteStateMachineType*, *StateType* y *TransitionType*. En la tabla 6.9 se muestran las correspondencias directas entre las máquinas de estado de iMMAS y las de OPC UA.

Concepto de iMMAS	Concepto de OPC UA	Correpondencia
StateMachine	FiniteStateMachineType	Indirecta
State	StateType	Directa
Transition	TransitionType	Directa
Event	EventType	Directa
Action	Method	Directa

Tabla 6.9: Resumen de las correspondencias básicas entre las máquinas de estado entre iMMAS y OPC UA.

El único concepto que se debe definir en OPC UA de iMMAS es el concepto de *FinalState* para especificar si la máquina de estados termina y no se ejecuta más. Para ello, se ha definido un nuevo concepto *FinalState* de tipo *StateType* como se observa en la figura 6.10.

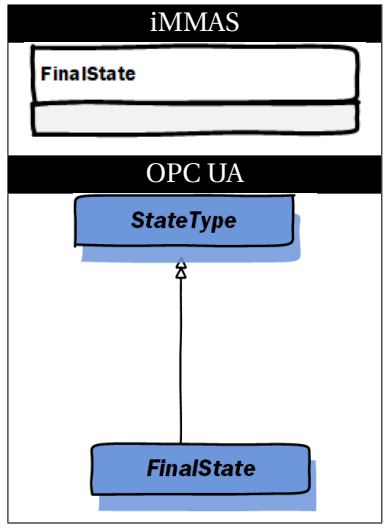


Tabla 6.10: Transformación realizada para FinalState.

EJEMPLO DE MODELADO CON LOS CONCEPTOS DE IMMAS EN OPC UA

A partir de las reglas de transformación entre iMMAS y OPC UA podemos transformar los modelos PIM diseñados en iMMAS en modelos PSM en OPC UA. Para ello tenemos que aplicar la transformación en tres fases como se ha indicado anteriormente.

Vamos a modelar un sistema industrial simple con iMMAS, para posteriormente realizar la transformación del modelo a OPC UA. Para este ejemplo se ha escogido el sistema de Cinta de Rodillos que se modeló en en el capítulo 3. En primer lugar se ha diseñado el modelo en iMMAS para dicho sistema tal como se muestra en la figura6.4.

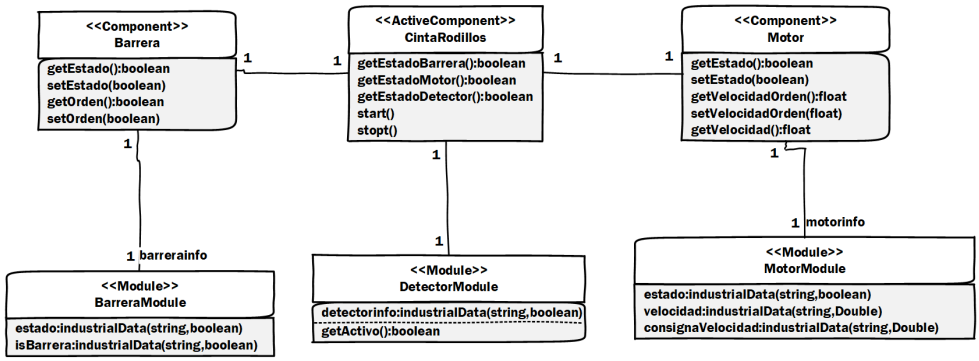


Figura 6.4: Modelo de iMMAS para el sistema industrial propuesto.

El elemento principal del modelo iMMAS es el componente activo *CintaRodillos*. *CintaRodillos* tiene un módulo *DetectorModule* para alojar un atributo *IndustrialData* cuyo valor registra el estado del detector *TRUE* o *FALSE*. Además *CintaRodillos* colabora con otros dos componentes que están relacionados con una relación de asociación, *Barrera* y *Motor*, que son dos componentes activos. El componente *Barrera* tiene un módulo *BarreraModule* con dos datos industriales para controlar el estado de la barrera y el funcionamiento de la barrera. En cambio, el componente *Motor* tiene un modulo con tres datos, uno para controlar el estado, otro para establecer la velocidad del *Motor*, y otro para establecer la consigna de velocidad. Como componente activo *CintaRodillos* tiene asociada una máquina de estado que denota su comportamiento. Cada uno de los componentes tienen las operaciones necesarias para poder acceder a cada elemento. La maquinas de estado para CintaRodillo se muestra en la figura 6.5.

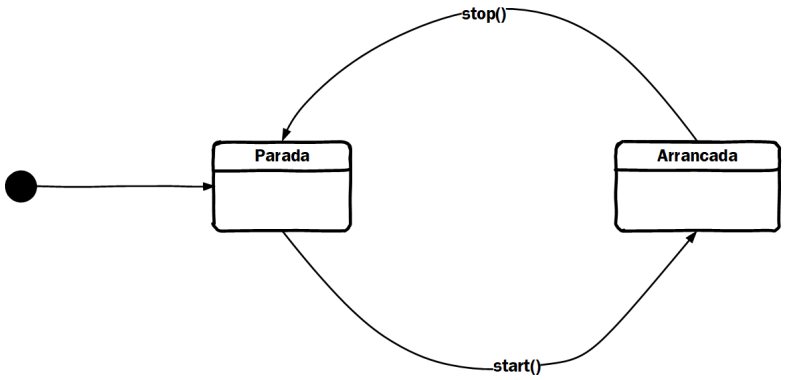


Figura 6.5: Modelo en iMMAS para el comportamiento de *CintaRodillos*.

El elemento *CintaRodillos* puede controlar el funcionamiento de todo el sistema mediante las operaciones, *start*/*stop*, para arrancar o parar todo el sistema de acuerdo a una secuencia o lógica especifica. Adicionalmente también posee otras operaciones que nos indican el estado de los elementos que forma el sistema, barrera, detector y motor. Una vez que hemos obtenido el modelo de acuerdo a iMMAS podemos obtener el modelo PSM correspondiente al modelo anterior aplicando las reglas de transformación específicada anteriormente utilizando una estrategia jerárquica. En la figura 6.6 se muestra el modelo PSM obtenido.

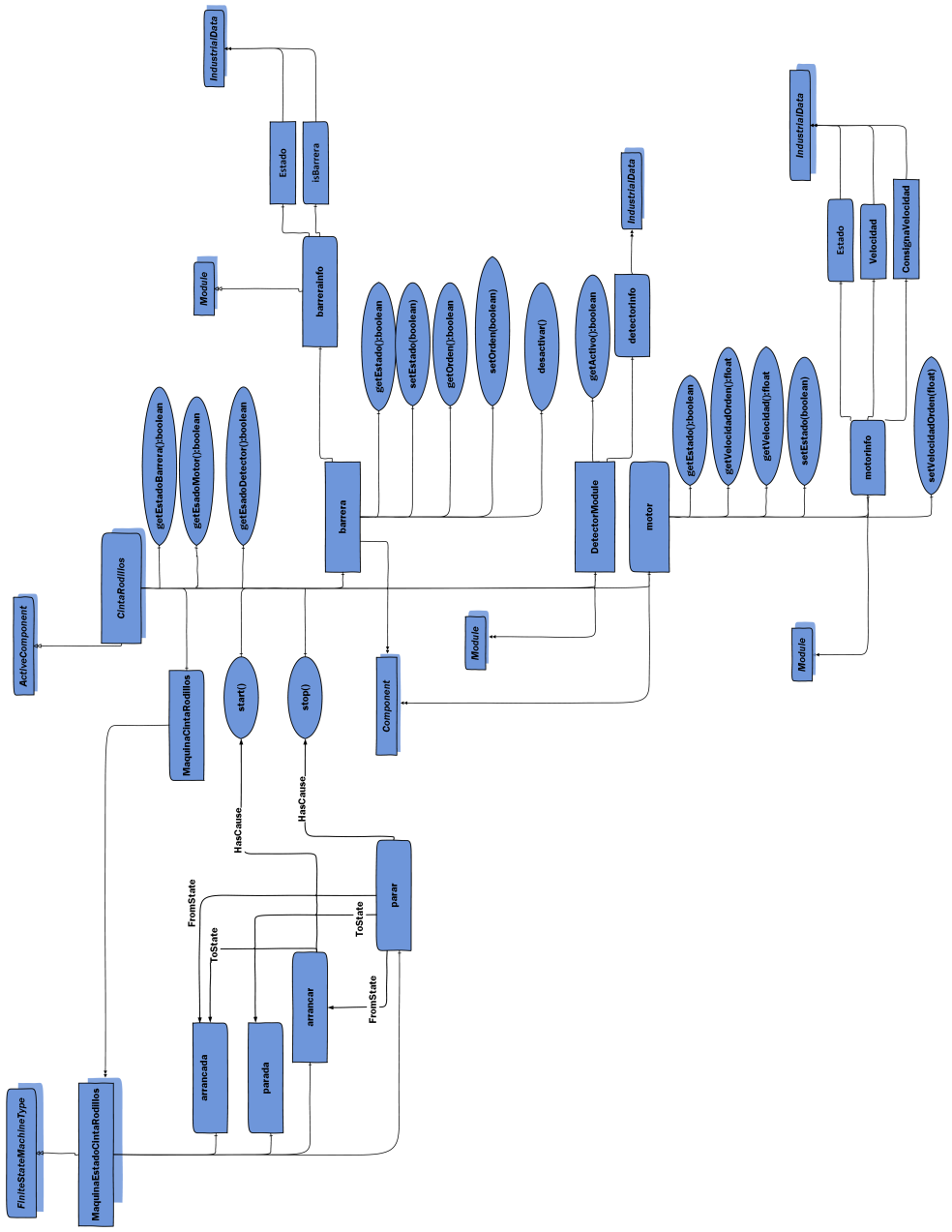


Figura 6.6: Transformación completa (modelo y máquina de estados) realizada para el modelado de IMMAS en OPC UA.

Para realizar la transformación del modelo PIM al modelo PSM en el modelo de información de OPC UA se han aplicado las tres fases de transformación señaladas anteriormente. Se han creado tres objetos de tipo Component para CintaRodillos, Barrera y Motor. El componente CintaRodillos tiene cinco operaciones, *getEstadoBarrera*, *getEstadoMotor*,*getEstadoDetector star* y *stop*. Para aplicar las reglas de transformación se utiliza una estrategia jerárquica como se ha indicado anteriormente.

6.4. TRANSFORMACION DE MOSIE A OPC UA

Del mismo modo que iMMAS puede transformarse a un modelo PSM en OPC UA buscando las correspondencias entre los dos lenguajes, se puede también transformar todos los conceptos y abstracciones definidos en MOSIE. La transformación de los elementos de MOSIE a conceptos de OPC UA se fundamentan en la reglas de transformación ya explicadas anteriormente con iMMAS, ya que el perfil contiene un conjunto de componentes y módulos predefinidos para el modelado de sistemas industriales. Por tanto, dado que los componentes y módulos se definieron como tipos de ObjectType, las nuevas abstracciones serán ObjectTypes de tipo componente y módulo. Para MOSIE se ha creado un nuevo espacio de direcciones, con la idea de diferenciar los conceptos de MOSIE de los conceptos de iMMAS.

6.4.1. EL PAQUETE MODULES EN OPC UA

El paquete Modules de MOSIE formado por los sub-paquetes, *Alarms*, *InputOutput* y *Control* contiene un conjunto de módulos predefinidos de propósito específicos que pueden simplificar el modelado de sistemas industriales basados en iMMAS. Por cada tipo de módulo se ha definido un estereotipo con un conjunto de definiciones de atributos y operaciones, de forma que sea conforme al lenguaje iMMAS. El proceso de transformación seguido para transformar cada módulo predefinido al modelo de información de OPC UA sigue el mismo enfoque jerárquico que el realizado anteriormente con los módulos en iMMAS. El módulo transformado en OPC UA vendrá definido por un ObjectType específico de tipo Modulo.

EL PAQUETE INPUT/OUPUT EN OPC UA

Todos las definiciones de módulos se ha realizado extendiendo de *InputOutputModel*. Al ser *InputOutputModule* la definición de un estereotipo sobre el elemento sintáctico *Module* del paquete base de iMMAS y además ser abstracto, el resto de conceptos derivados añaden definiciones sobre la definición base. Así, por ejemplo, el atributo *value* depende de un *IndustrialData* con un valor de tipo *DataType*, que se resuelve en la definición del módulo derivado. En OPC UA cada tipo de modulo se construye en un *objectType* agregando todos los elementos sintácticos que tiene su correspondiente versión iMMAS aplicando el enfoque jerárquico señalado anteriormente.

InputOutputModule:

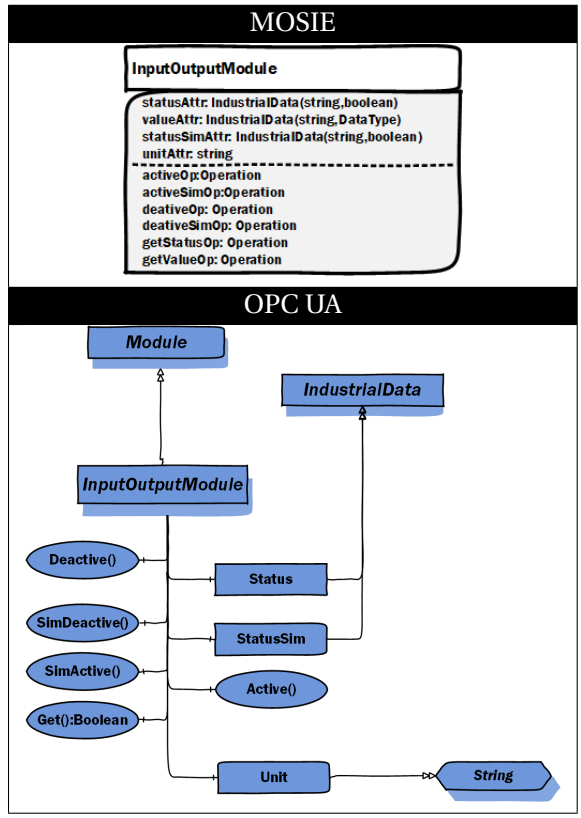


Tabla 6.11: Transformación realizada para InputOutputModel.

El elemento *InputOutModule* de iMMAS se transforma en un ObjectType de

OPC UA, así como cada uno de los atributos y operaciones que tiene definido el estereotipo, tal y como se observa en la tabla 6.11.

DigitalOutput y autoManDigitalOutput:

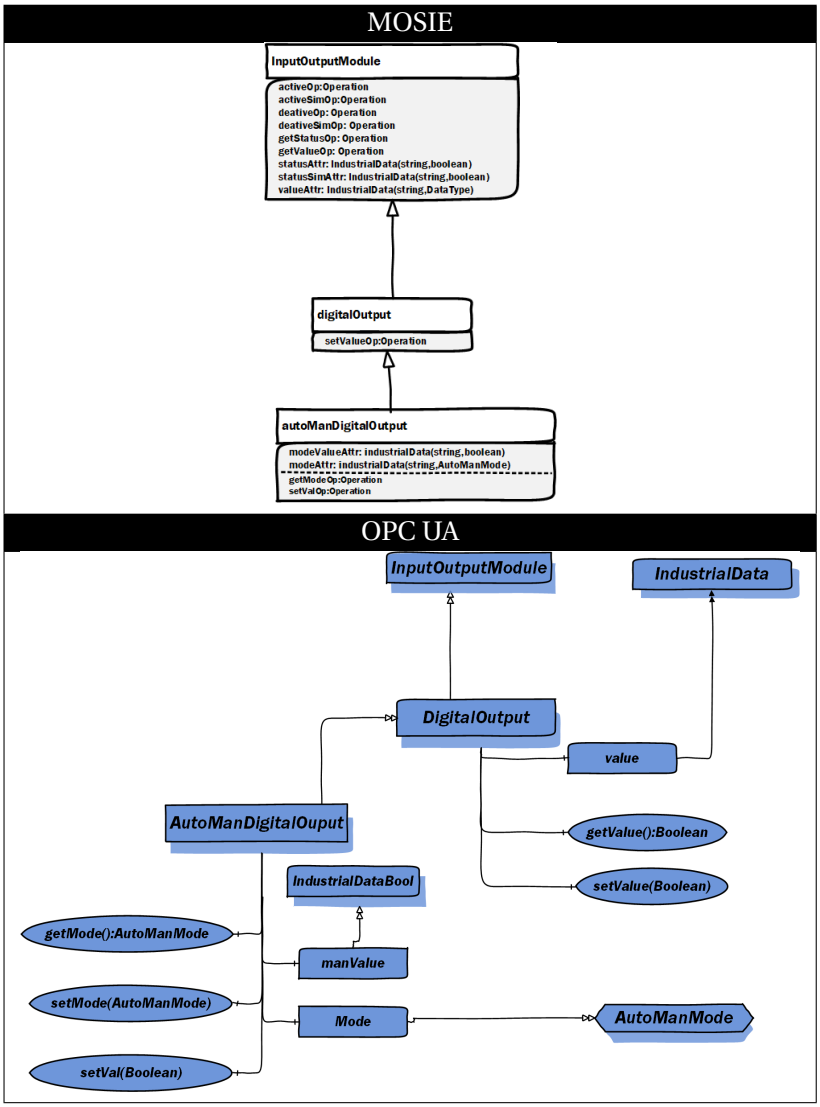


Tabla 6.12: Transformación realizada para DigitalOutput.

Para la transformación de los módulos *digitalOuput* y *autoManDigitalOuput* se ha seguido el mismo enfoque jerárquico llevando a cabo la transformación de

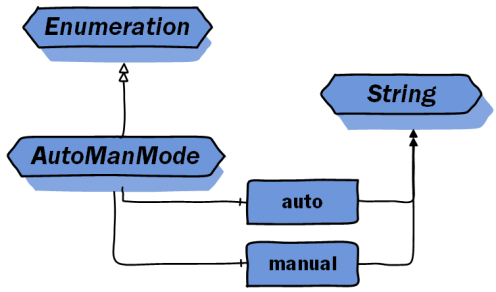


Figura 6.7: Definición de tipos para *modeAttr*.

cada uno de los elementos sintácticos que lo conforman, atributos que se convierten en variables y operaciones que se convierten en métodos en OPC UA. Por otra parte, se realizan las definiciones de subtipo en la relación de herencia. El atributo *ValueAttr* se convierte en OPC UA en una variable de tipo *IndustrialData* con valor *booleano*.

Para trabajar con el modo de funcionamiento de este elemento se ha definido un nuevo tipo de dato *AutoManMode*. La transformación a OPC UA se hace de forma directa definiendo un nuevo tipo de dato enumerado con las valores posibles, tal y como se indica en la figura 6.7.

DigitalInput:

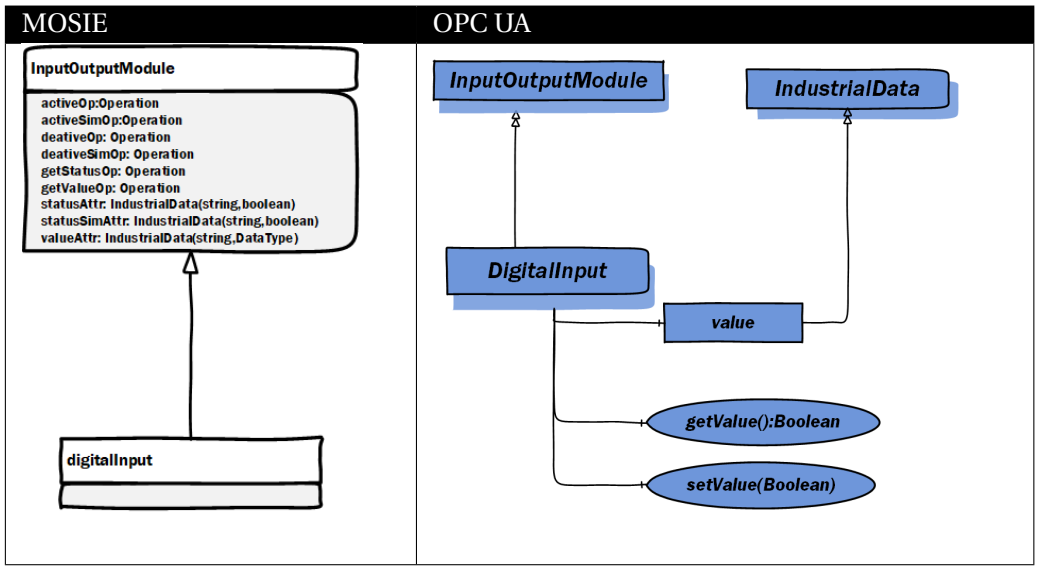


Tabla 6.13: Transformación realizada para DigitalInput.

En este caso al transformar *digitalInput* se obtiene como resultado un *ObjectType digitalInput* con un atributo *value* de tipo *IndustrialData* con valor booleano, y dos operaciones *geValue* con un parámetro de salida de tipo *boolean* y *setValue* con parámetro de entrada de tipo booleano.

AnalogOutput y autoManAnalogOuput:

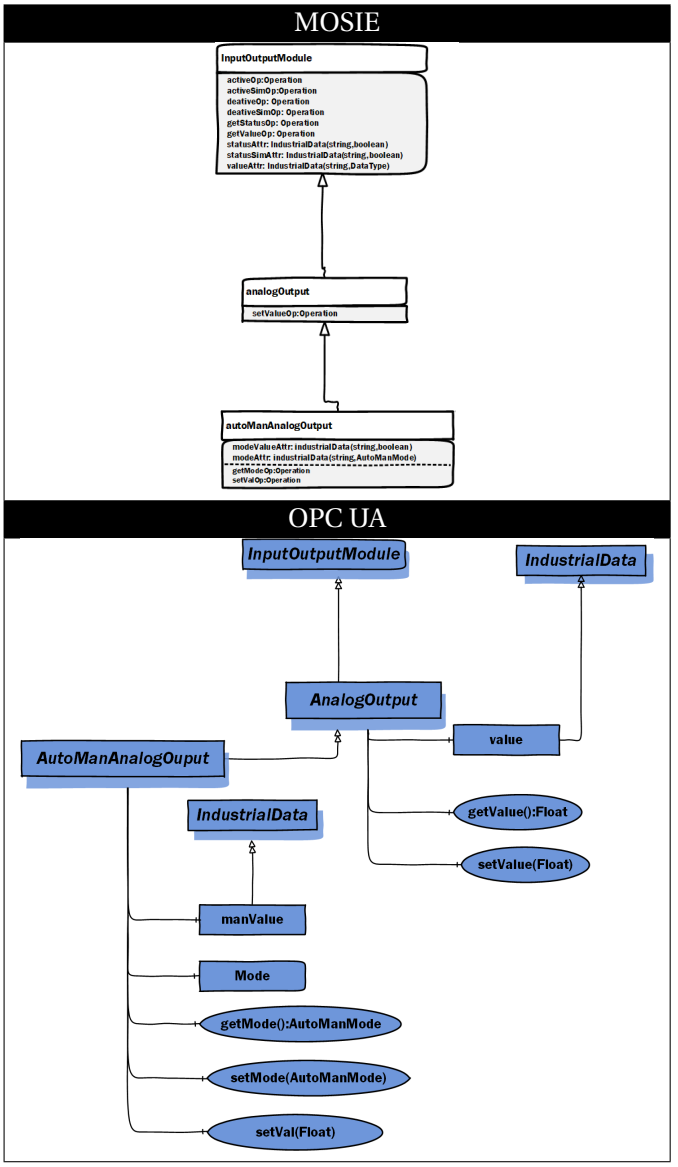


Tabla 6.14: Transformación realizada para AnalogOutput.

La transformación en este caso con *analogOuput* y *autoManAnalogOuput* de ha realizado de forma similar a *digitalOuput*, pero considerando valores analógicos de tipo *Float*. Se obtiene como resultado un *ObjectType* que se compone de la variable value de tipo *IndustrialData* con valor float, el método *getValue* con un parámetro de salida de tipo *Float* y el método *setValue* con un parámetro de entrada de tipo Float.

analogInput:

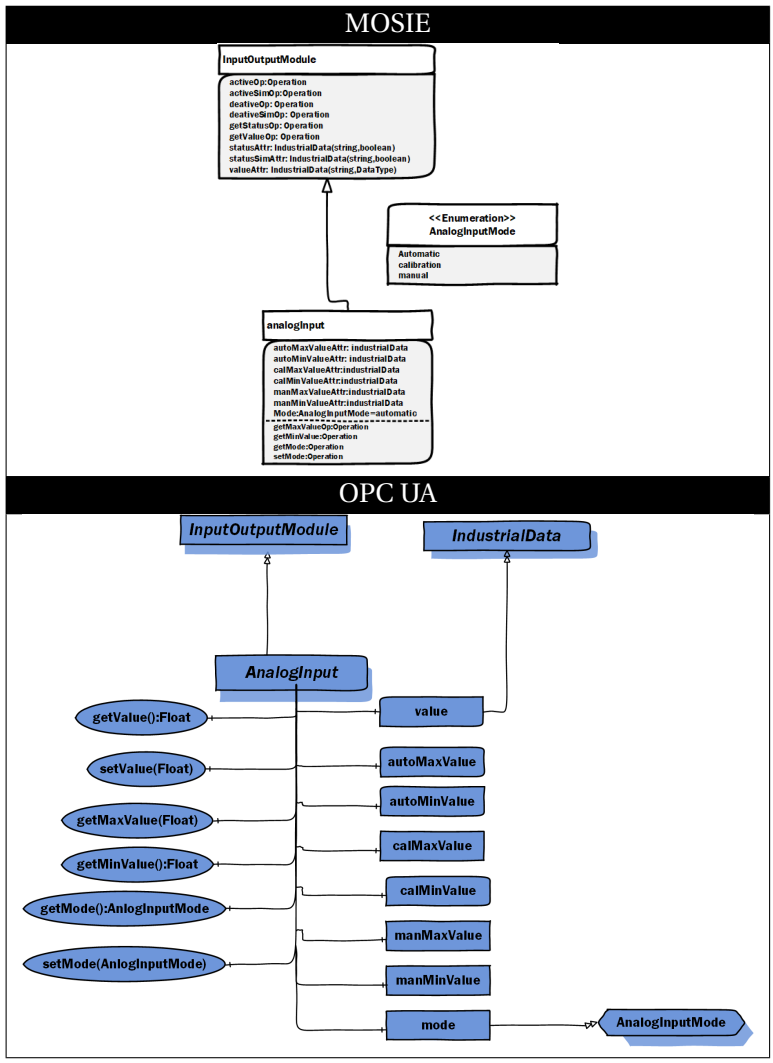


Tabla 6.15: Transformación realizada para analogInput.

En este caso, el tipo de módulo *AnalogInput* se transforma de modo analogo a casos anteriores como un *ObjectType* con un atributo *value* de tipo *IndustrialData* con valor float y con sus correspondientes métodos, *setVal* para escribir un valor y geVal con un parámetro de salida de tipo *Float*. De igual modo, se transforman los atributos en variables y las operaciones en métodos de OPC UA. Al igual que ocurría en el caso de los objetos, *AutoManDigitalOutput* y *AutoManAnalogOuput*, se ha creado un nuevo tipo de dato para especificar los modos de calibración. Su transformación se puede ver en la figura 6.8.

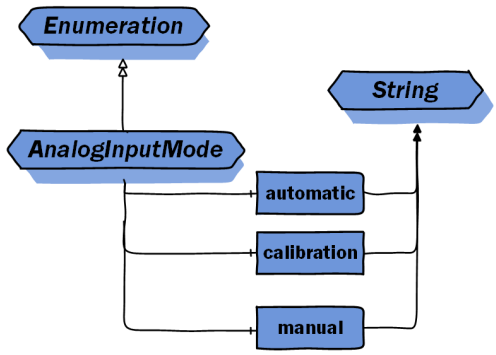


Figura 6.8: Definición de tipos para *modeAttr*.

EL PAQUETE ALARMS EN OPC UA

El paquete *alarms* ha sido creado para modelar tipos de módulos relacionados con el disparo de alarmas del sistema. A continuación, se muestran las transformaciones realizadas con los distintos elementos.

Alarm:

De forma equivalente la transformación a OPC UA se realiza a un *ObjectType* de tipo abstracto, dado que no es instanciable, con las variables y métodos correspondientes. El atributo *refMonitoredValue* ha sido eliminada de este elemento, ya que como ocurría en el paquete anterior con *InputOutOutModule*, dependiendo del tipo de alarma que se defina en *DataType*. Se definirá el tipo primitivo concreto (*booleano, entero o float*) en los tipos de módulos *DigitalAlarm* y *AnalogAlarm*.

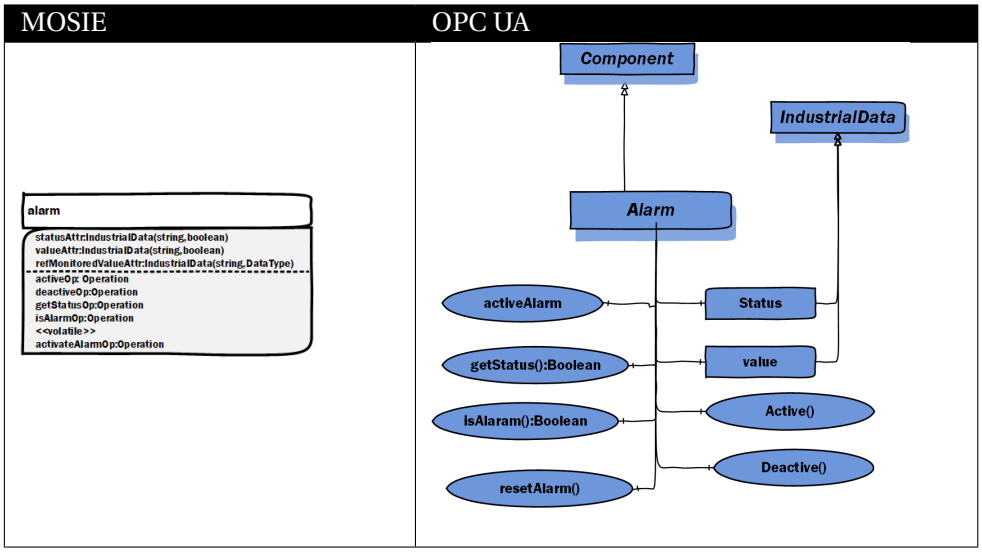


Tabla 6.16: Transformación realizada para Alarm.

DigitalAlarm:

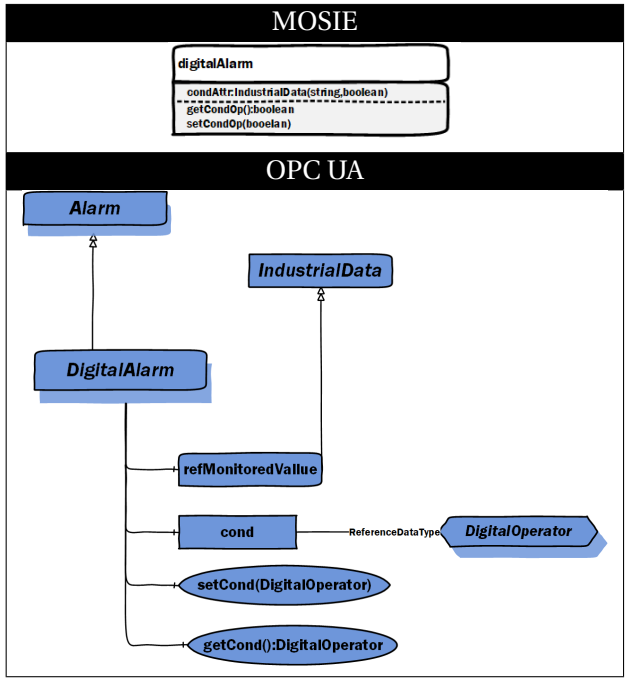


Tabla 6.17: Transformación realizada para DigitalAlarm.

Este tipo de módulo toma como base *Alarm* con lo que hereda todas sus variables y métodos. El atributo *refMonitoredValue*, que en MOSIE se encontraba en *Alarm*, se incluye como atributo de tipo *IndustrialData*con valor booleano, ya que estamos trabajando con alarmas digitales True/False.

AnalogAlarm:
Este elemento tiene como hereda de *Alarm*. En este caso, el atributo *refMonitoredValue* es de tipo *IndustrialData*, pero en este caso se albergará datos de tipo *float*, ya que este concepto trabaja con alarmas que son disparadas por valores analógicos.

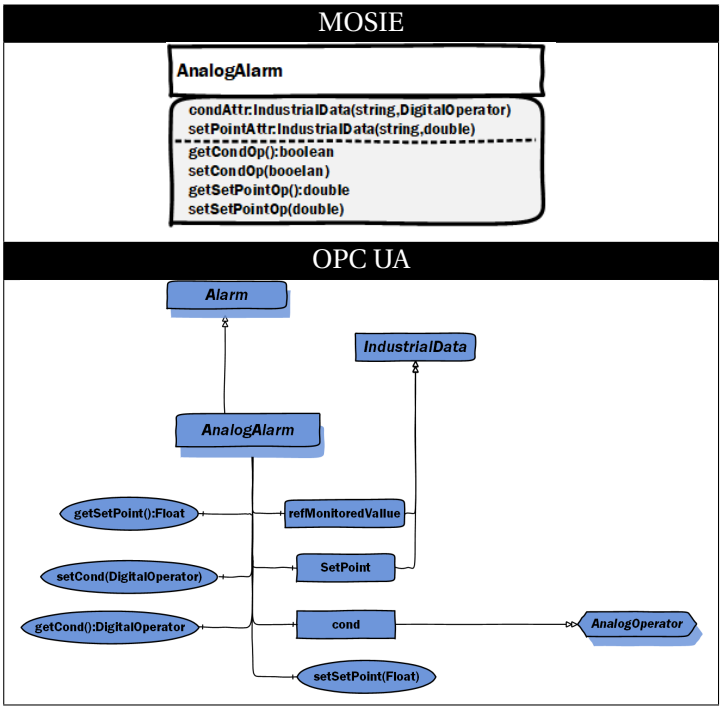


Tabla 6.18: Transformación realizada para AnalogAlarm.

Para este paquete se han definido también dos tipos de enumerados para expresar las comparaciones a realizar en cada tipo de alarma, *DigitalAlarm* y *AnalogAlarm*. Las transformaciones de estos enumerados en OPC UA se pueden ver en la figura 6.19

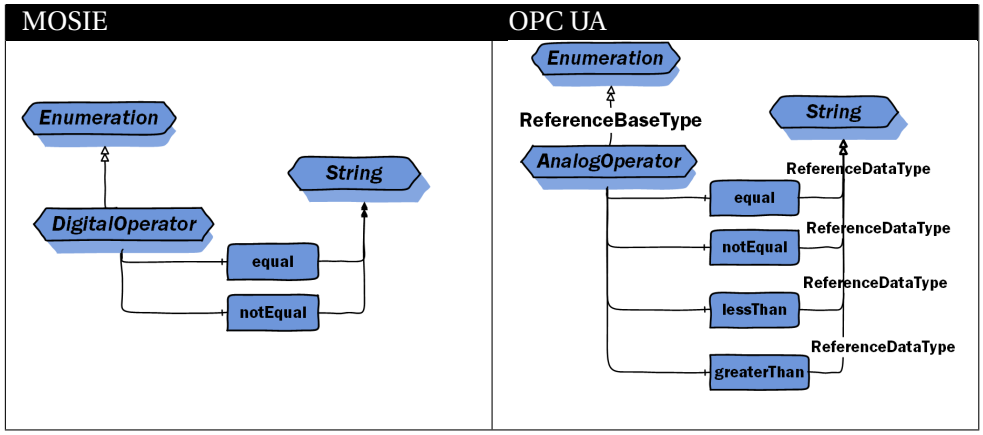


Tabla 6.19: Transformación realizada para AlarmEnumerate.

EL PAQUETE CONTROL EN OPC UA

De forma análoga a anteriores casos todos los tipos de módulos del paquete control de MOSIE son transformados a OPC UA siguiendo un enfoque jerárquico a *ObjectTypes* con las variables y métodos correspondientes en el modelo de información OPC UA.

ActivationControl:

La correlación realizada para este concepto se ha basado en la utilización de *Module* como *BaseType* y en la creación de los métodos y atributos necesarios recogidos para dicho concepto en su definición en MOSIE.

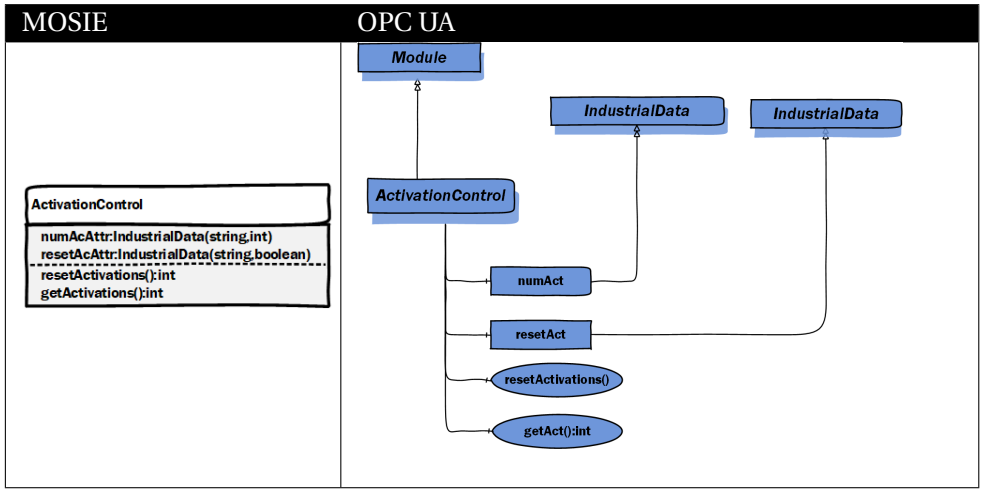


Tabla 6.20: Transformación realizada para ActivationControl.

Power:

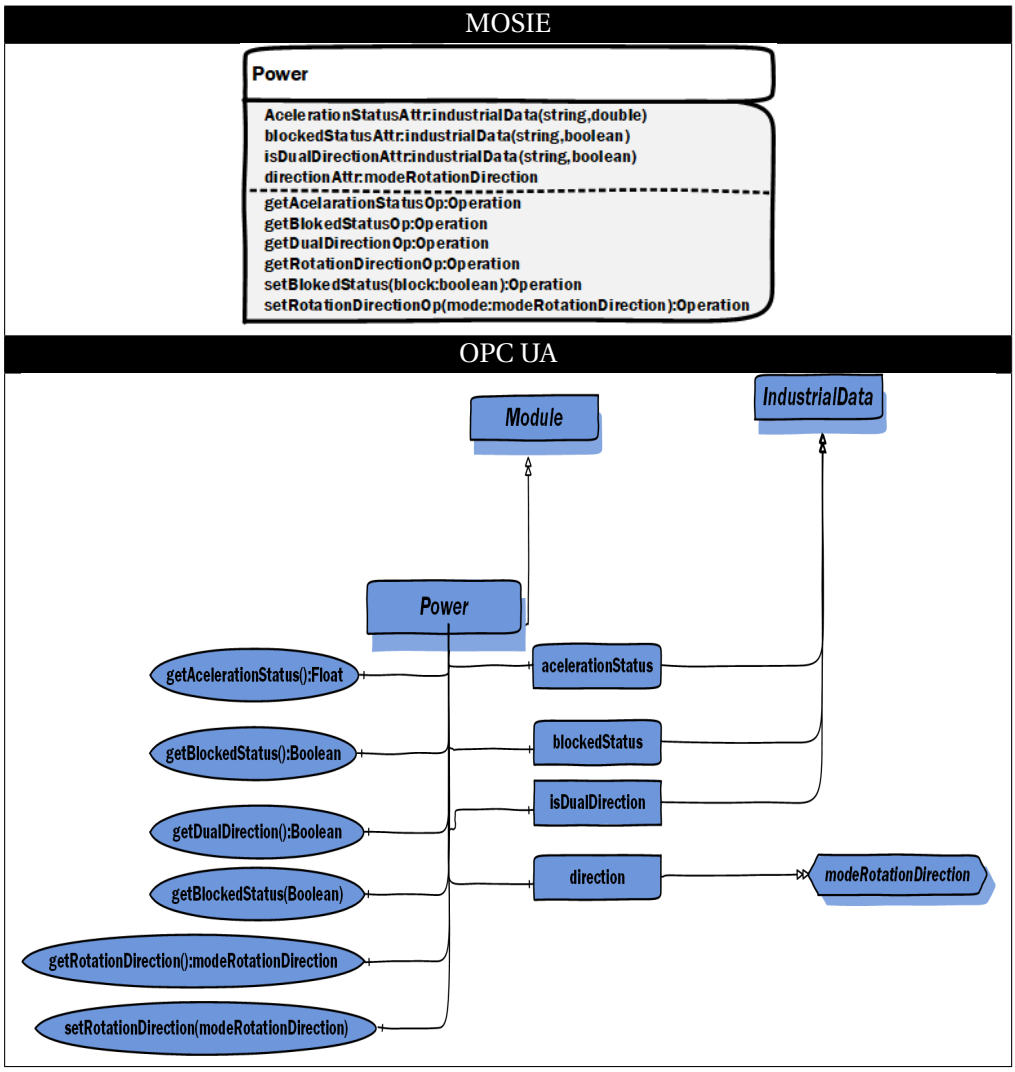


Tabla 6.21: Transformación realizada para Power.

Para el concepto Power se ha realizado un correlación directa en cuanto a métodos y atributos, utilizando como concepto base *Module* de iMMAS.

ControlParameters:

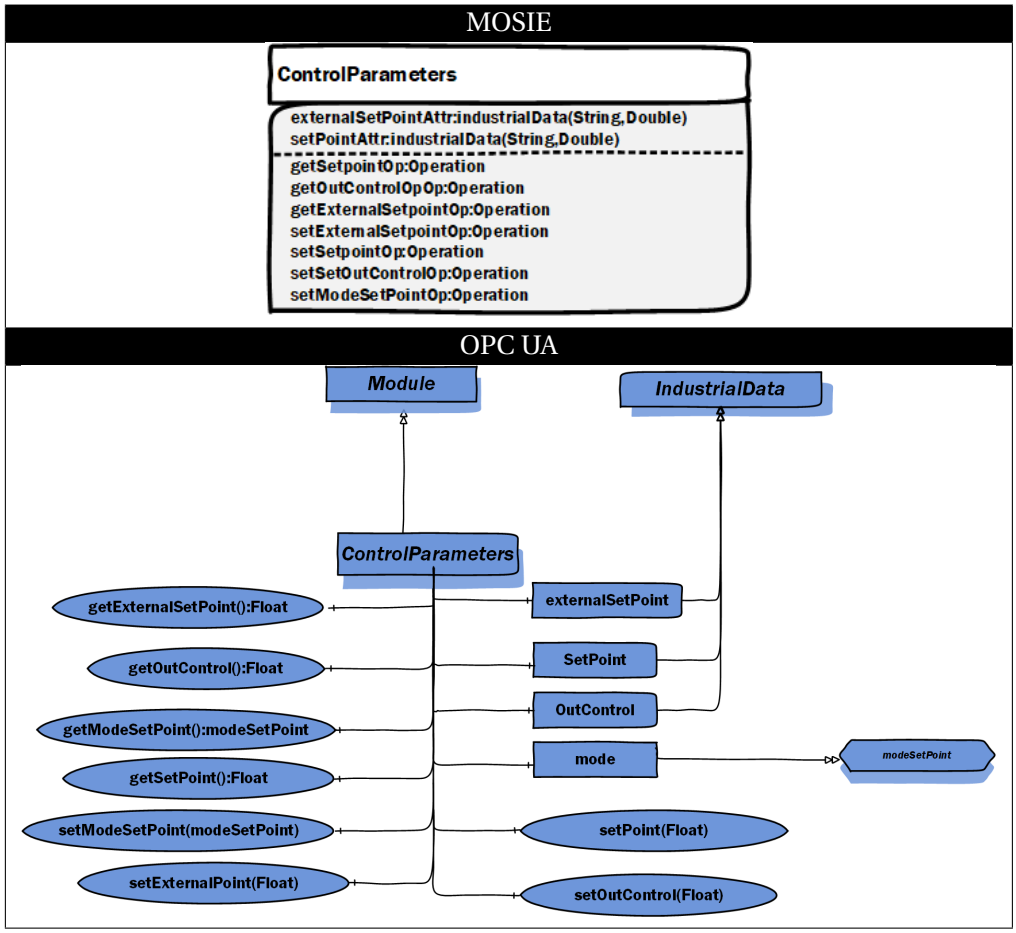


Tabla 6.22: Transformación realizada para ControlParameters.

El concepto *ControlParameters* recoge las atributos y operaciones para almacenar los parámetros de un controlador. Para realizar su correlación en OPC UA se ha utilizado como tipo base module y se han creado todos los métodos y variables acorde con su definición en MOSIE.

PIDControlParameters:

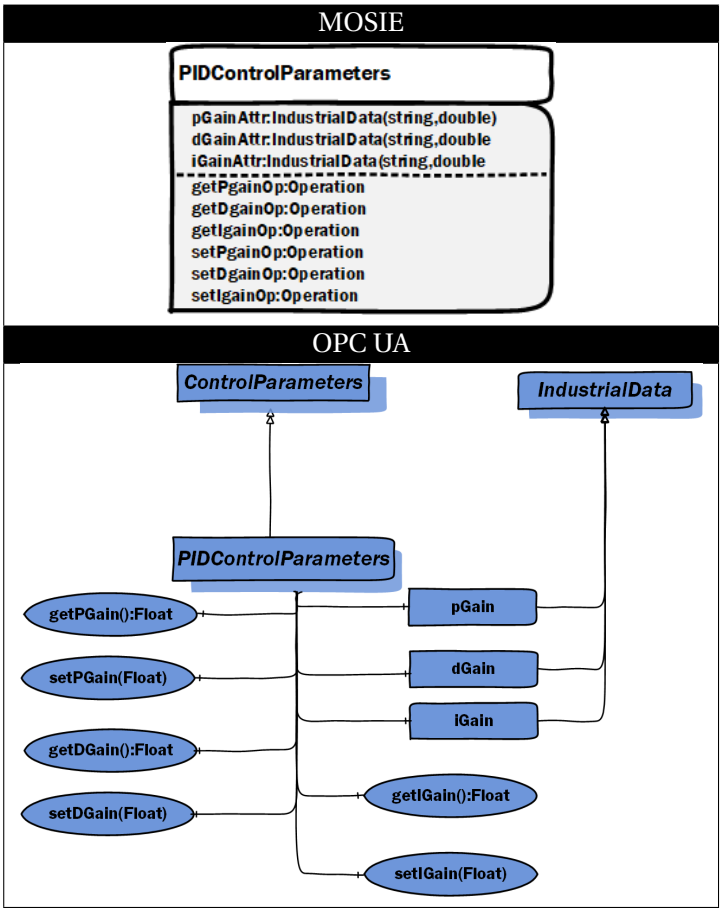


Tabla 6.23: Transformación realizada para PIDControlParameters.

La transformación realizada utiliza como *BaseType ControlParameters*, ya que según especifica MOSIE este concepto es una especialización de *ControlParameters*. Por último se le a dotado de los operaciones y variables necesarios de acuerdo a su definición en MOSIE.

En este paquete también se han definido dos tipos de datos enumerados *modeRotationDirection* y *modeSetPoint* que se transforman de forma directa a un *DataType* de tipo enumerado de forma análoga en OPC UA. En la tabla 6.24 se puede ver ambas definiciones.

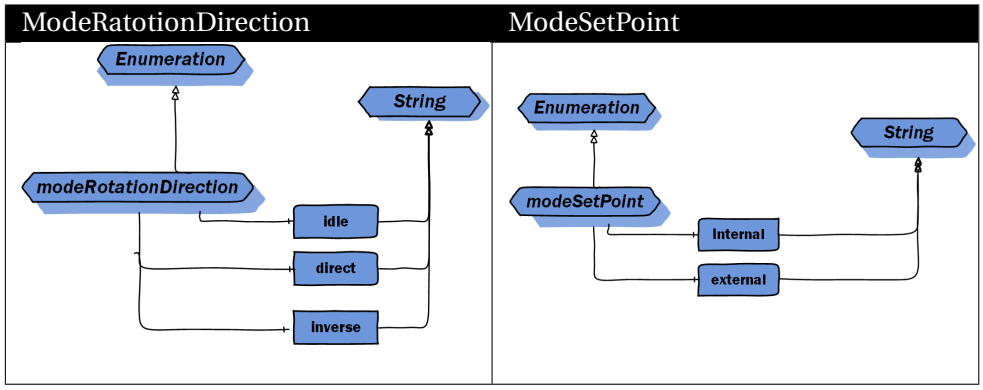


Tabla 6.24: Transformación realizada para modeRotatationDirection y modeSetPoint .

6.4.2. EL PAQUETE COMPONENTS EN OPC UA

El paquete componentes está formado por una serie de subpaquetes,*Devices, Controllers, Sensors* y *Actuators* en los que se agrupan el conjunto de tipos de componentes conformes a *Component* de iMMAS. En las transformaciones realizadas para este paquete también se ha seguido una aproximación jerárquica, utilizando el mecanismo de herencia de OPC UA. Los paquetes *Sensor* y *Actuators* no se presentan en esta tesis doctoral por requerir la definición de un conjunto amplio de atributos y operaciones, pero su transformación sería totalmente idéntica.

EL PAQUETE DEVICES EN OPC UA

Este paquete recoge dos conceptos fundamentales: *Sensor* y *Actuator*. El concepto sensor y actuador es una extensión del concepto de iMMAS *Component* como veremos a continuación.

Sensor:

Como se puede ver este concepto no posee ninguna propiedad ni método adicional. Pero si se le ha añadido una referencia de tipo *HasComponent* indicando que este elemento puede contener módulos de tipo *Alarm*, tal y como se indica en MOSIE con el atributo *alarm[0..1]:Module*.

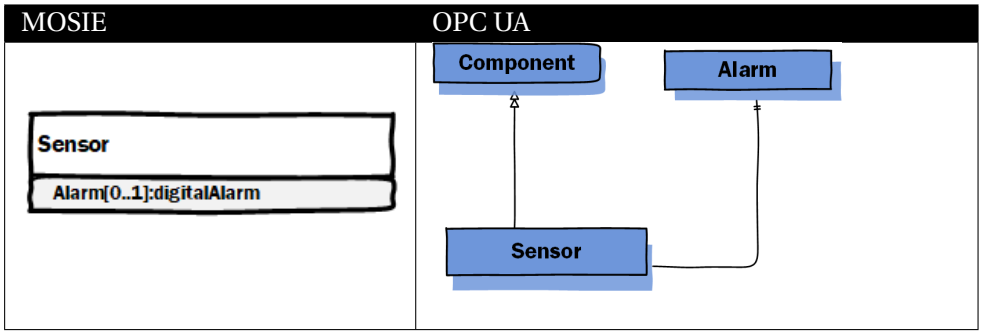


Tabla 6.25: Transformación realizada para Sensor.

DigitalSensor:

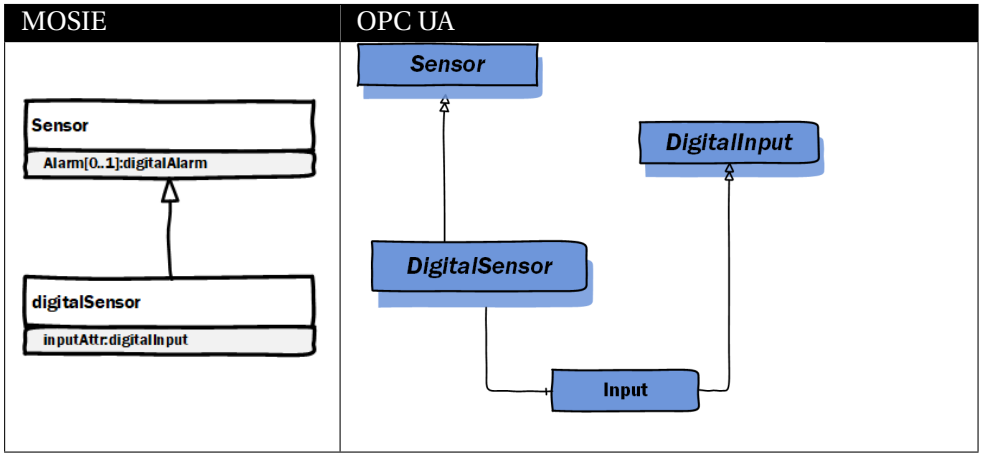


Tabla 6.26: Transformación realizada para DigitalSensor.

Para realizar la correspondencia del concepto *digitalSensor*, basta con definir en OPC UA un concepto que tenga tipo base *"Sensor"* y añadirle como módulo de tipo *digitialInput* como se puede ver en la tabla anterior.

AnalogSensor:

Para *AnalogSensor* se ha definido en OPC UA un nuevo concepto, cuyo tipo base es Sensor, al igual que en el caso de *DigitalSensor*, pero ahora se le han añadido un objeto módulo de tipo *analogInput*

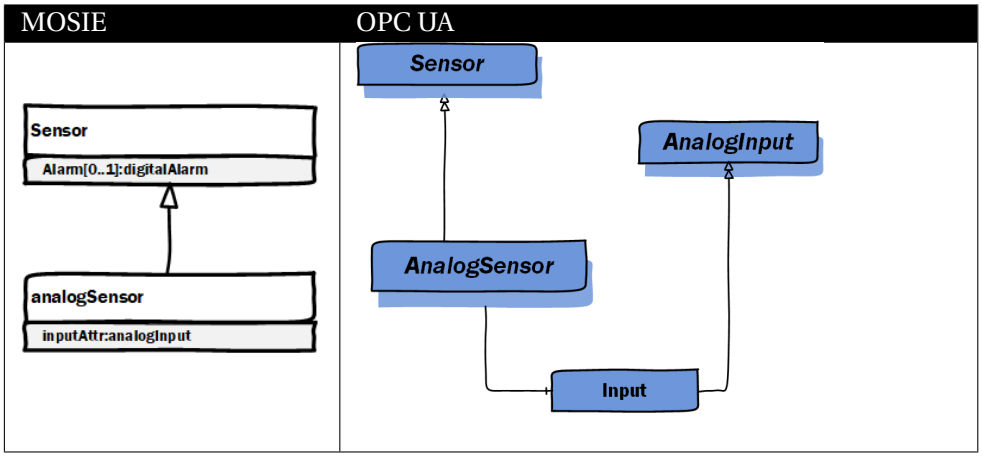


Tabla 6.27: Transformación realizada para AnalogSensor.

Actuator:

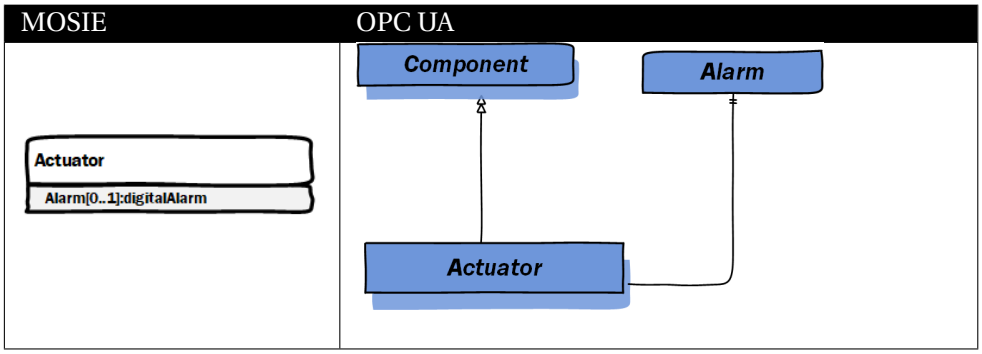


Tabla 6.28: Transformación realizada para Actuator.

Este concepto representa una clase abstracta que sirve de elemento base sobre el que se apoya el concepto *actuator*. Como en el caso de *Sensor* en la transformación de *Sensor* se obtiene un *ObjectType* con una referencia *HasComponent* para incluir un objeto módulo de tipo *Alarm*.

DigitalActuator:

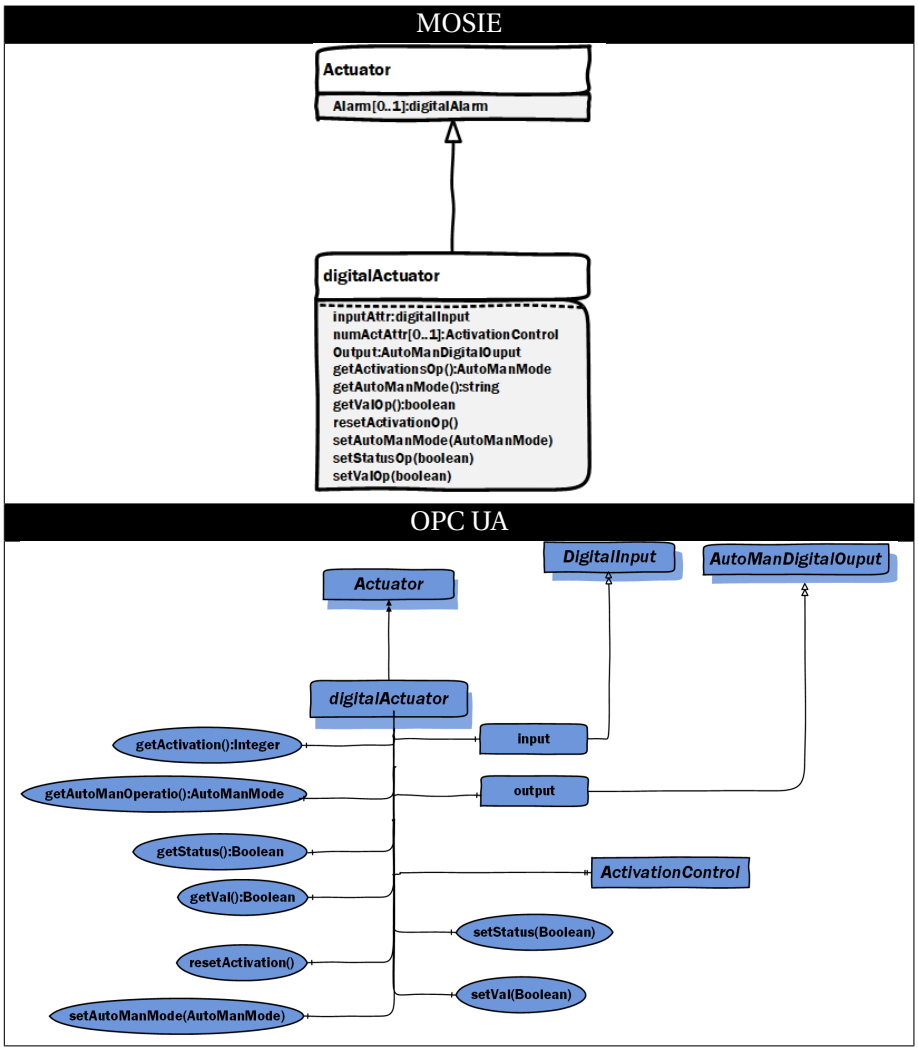


Tabla 6.29: Transformación realizada para DigitalActuator.

Para la transformación a OPC UA de *DigitalActuator* se ha utilizado como tipo base *Actuator* y se han añadido dos objetos *input* y *output* de tipo *digitalInput* y *AutoMandigitalOutput*.

AnalogActuator:

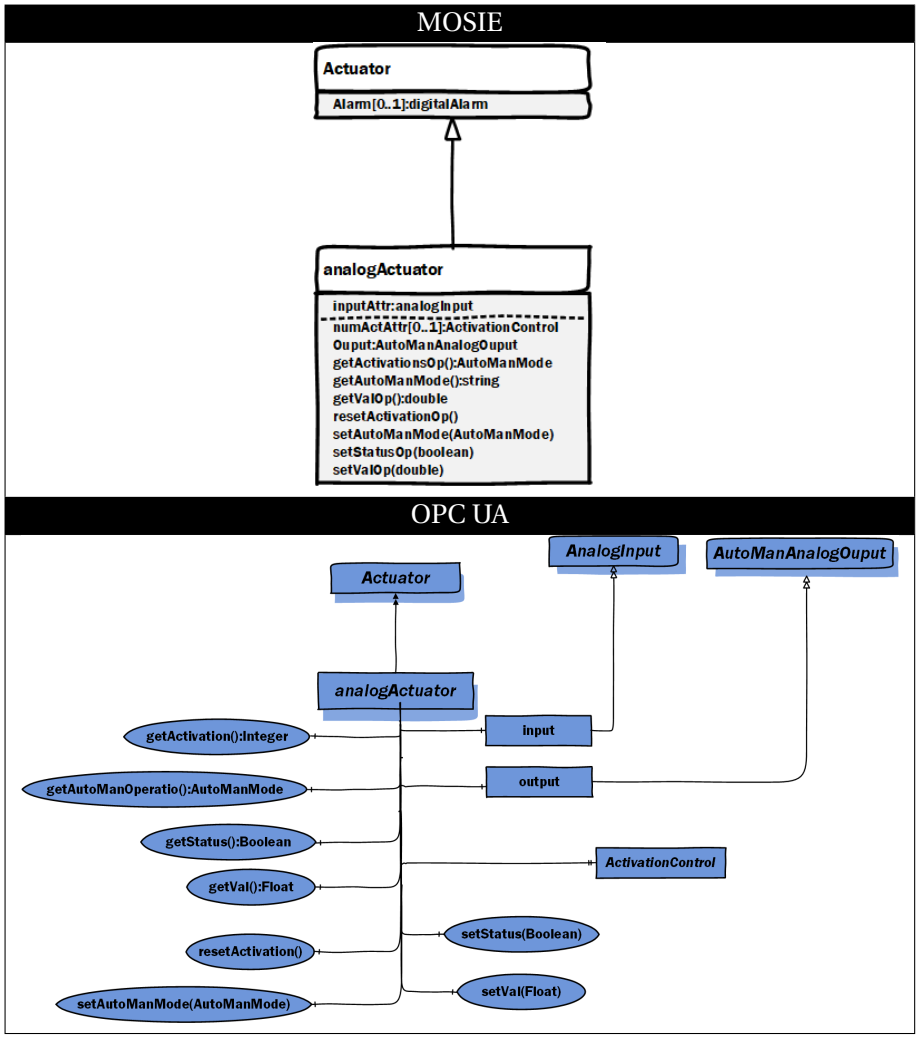


Tabla 6.30: Transformación realizada para AnalogActuator.

Para este concepto se ha realizado una correlación,muy parecida a la de *DigitalActuator*, con la salvedad de que los objetos *input* y *output* son de tipo *AnalogInput* y *AutoManAnalogOutput*.

VFDActuator:

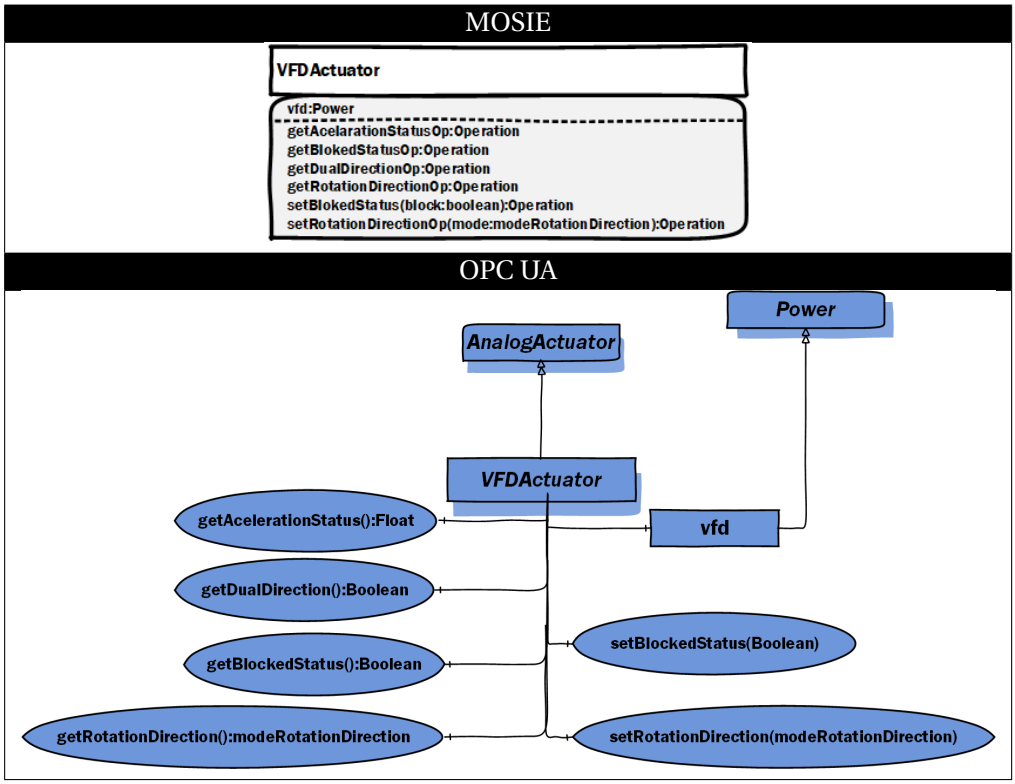


Tabla 6.31: Transformación realizada para VFDActuator.

Este concepto es una extensión de *AnalogActuator* por lo que se ha utilizado como BaseType dicho concepto. *VFDActuator* es un componente que tiene atributo de tipo *Power*.

EL PAQUETE CONTROLLERS EN OPC UA

Este paquete utiliza los módulos *ControlParameters* y *PIDControlParameters* para desarrollar dos componentes, uno para controladores sencillos: *Regulator* y otro para controladores PID: *PIDREgulator*. Las transformaciones realizadas son las siguientes:

Regulator:

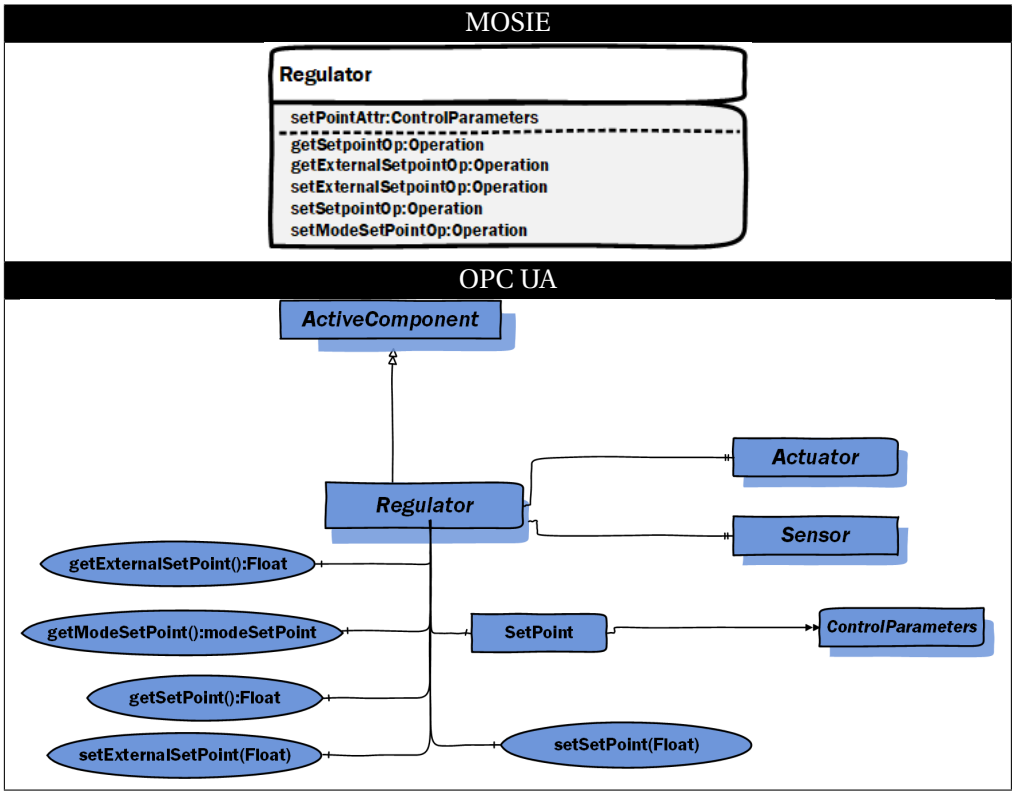


Tabla 6.32: Transformación realizada para Regulator.

Regulator tiene como tipo base *ActiveComponent* y se le ha definido un objeto de tipo *ControlParameters*, módulo que agrupaba todos los parámetros de un controlador. Por último, para indicar que puede estar relacionado con un sensor o un actuador, se le han creado dos referencias de tipo *HasComponent* a cada uno de estos conceptos.

PIDRegulator:

PIDRegulator tiene como elemento base *Regulator*. De este modo hereda todas sus propiedades y métodos. Se le han añadido una objeto módulo de tipo *PIDControlParameters* y los métodos propios para gestionar las constantes proporcional, integral y derivativa de este tipo de controladores.

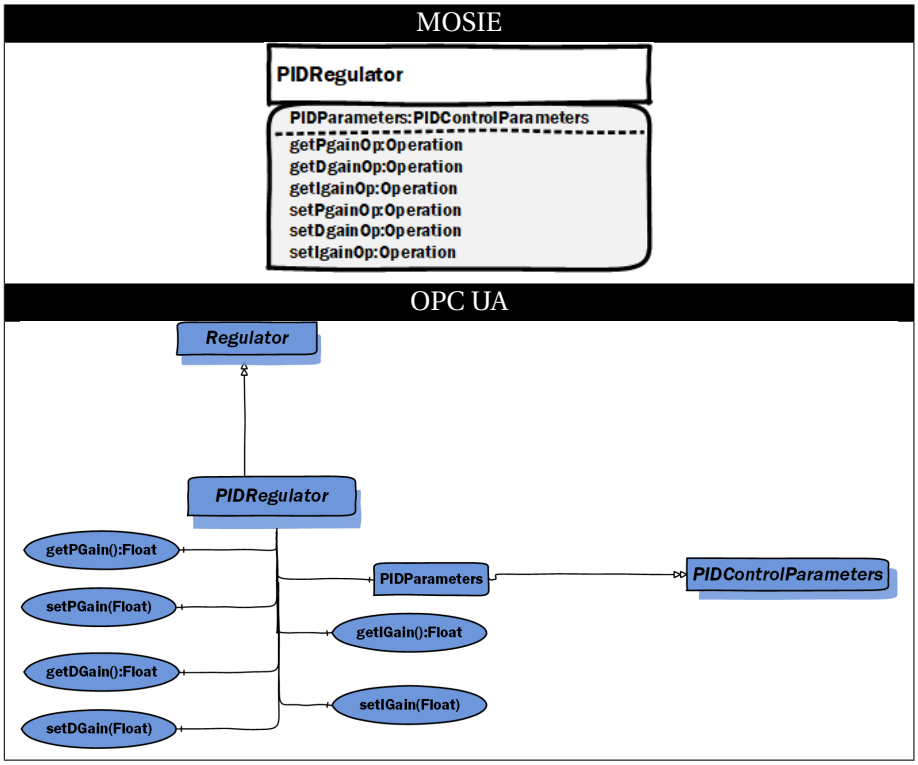


Tabla 6.33: Transformación realizada para PIDRegulator.

6.4.3. EJEMPLO DE MODELADO CON LOS CONCEPTOS DE MOSIE

Para mostrar como seria un modelo con los conceptos de MOSIE se ha modelado el mismo sistema industrial de la Cinta de Rodillos que se modeló previamente directamente con el lenguaje iMMAS. El modelo PIM creado utilizando los conceptos de MOSIE se encuentra en la figura 6.9.

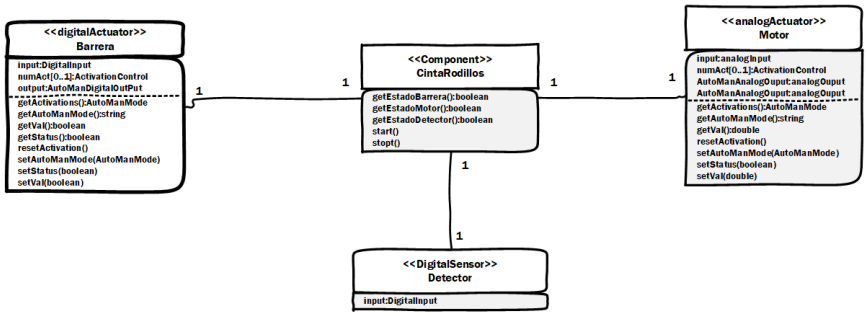


Figura 6.9: Modelo de MOSIE para el sistema industrial.

Para construir este modelo hemos utilizado los conceptos de *DigitalActuator* para modelar la barrera, *DigitalSensor* para modelar el detector y *AnalogActuator* para modelar el motor eléctrico de la cinta. Como se puede ver en la figura 6.9 el modelo es mas sencillo que el realizado utilizando solo conceptos de iMMAS, ya que MOSIE ofrece elementos que ya tiene sus propios módulos y que albergan atributos de tipo *IndustrialData*, por lo que no hay que crearlos de nuevo. La máquinas de estados creadas se puede ver en la figura 6.10.

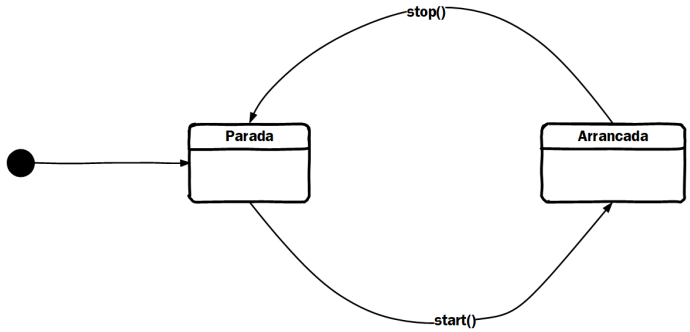


Figura 6.10: Modelo en iMMAS para el comportamiento de *CintaRodillos*.

La máquina de estados para este ejemplo es la misma que la que se realizó utilizando iMMAS y por tanto la transformación que se debe de realizar para que se puedan ser desplegadas en un servidor OPC UA es la misma que en el caso anterior, por simplicidad no se ha incluido en la figura 6.11.

El modelo PSM obtenido en OPC UA después de aplicar las tres fases de transformación sobre el modelo PIM se puede ver en la figura 6.11. En el dicho modelo se ha realizado una definición de tipos (TypeDefinition) de los elementos ya transformados a OPC UA, *DigitalSensor*, *DigitalActuator* y *AnalogActuator*, por lo que estos heredan todas sus variables, métodos y referencias.

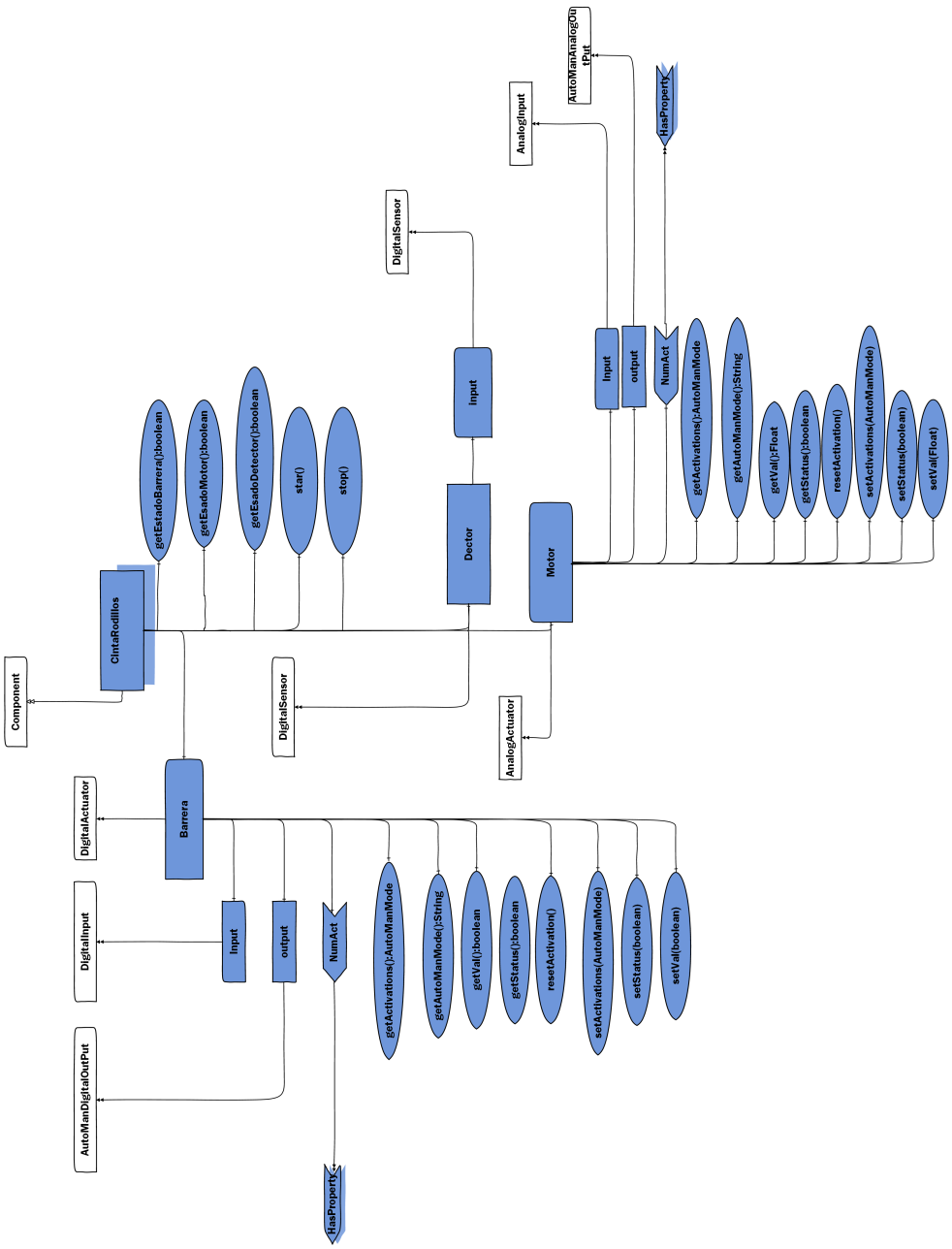


Figura 6.11: Modelo equivalente al realizado en MOSIE para OPC UA.

Como se puede apreciar en la figura 6.11, el modelo PSM que se ha realizado es mas simple con respecto al número de elementos, ya que con definir 4 componentes el modelo recoge todos los elementos del sistema industrial. Pero estos componentes tienen un mayor número de atributos y operaciones si lo comparamos con el modelo PSM obtenido utilizando directamente iMMAS, aunque con un nivel de abstracción mayor. Por lo tanto, una vez desplegado iMMAS y MOSIE en un OPC UA, parece mas interesante la utilización de MOSIE para modelar sistema industriales, ya que no solo facilita el modelado, sino porqué los modelos realizados utilizando MOSIE pueden desplegarse mas rápidamente y de manera mas sencilla en un servidor OPC UA.

6.5. DESPLIEGUE DEL MODELO PSM EN UN SERVIDOR OPC UA

Para el despliegue del modelo PSM en el servidor OPC UA tenemos que verificar que el modelo obtenido es conforme al modelo de información de OPC UA. Al aplicar las reglas de transformación el modelo PSM se garantiza que el modelo es correcto "por construcción.[en] OPC UA siempre que el modelo PIM lo fuera en iMMAS.

Para trabajar con el modelo de información en un servidor OPC UA necesitamos tener una descripción del modelo de información de OPC UA en XML con un esquema definido por la propia OPC Foundation [8]. En nuestro caso, junto con el modelo de información del modelo PSM tenemos que incluir el espacio de nombres de iMMAS, MOSIE y el espacio de nombre genérico de UA. Una vez que tenemos el modelo en XML podemos utilizar diferentes tipos de servidores OPC UA comerciales o implementar nuestro propio servidor para levantar el servidor junto con el modelo PSM que será accesible a los clientes OPC UA.

Podemos realizar el despliegue del servidor OPC UA en tres fases:

- Transformación de los modelos conceptuales a ficheros XML siguiendo las especificaciones del estandar UA.
- Compilación de estos modelos con la herramienta de la OPC Foundation: Opc.Ua.ModelCompiler.exe [8], generando los ficheros binarios y XML finales.
- Despliegue de los ficheros binarios o XML finales en un servidor OPC UA.

6.5.1. DEFINICIÓN DE IMMAS/MOSIE EN FICHEROS XML

Los modelos conceptuales definidos para iMMAS y MOSIE, han de ser representados mediante ficheros XML para poder ser introducidos en el espacio de direcciones de un servidor OPC UA, y estos ficheros deben de se conformes al modelo de información de UA.

Además deben de recoger todas las definiciones y relaciones de los elementos representados en los modelos conceptuales. Para la construcción de estos ficheros podemos apoyarnos en alguna herramienta de modelado comercial o podemos generar los ficheros XML directamente con un editor de texto. En nuestro caso, hemos utilizado una herramienta gráfica de modelado [134] con la que se han construido todas definiciones recogidas en los modelos conceptuales, para posteriormente generar los fichero XML a partir de dichas definiciones. A continuación podemos ver un ejemplo en XML del elemento *AnalogInput*, generado por esta herramienta:

```
<ObjectType SymbolicName="MOSIE:analogInput"
BaseType="MOSIE:InputOutputModule">
<Children>
<Object SymbolicName="MOSIE:value" TypeDefinition="iMMAS:IndustrialData" />
<Object SymbolicName="MOSIE:autoMaxValue" TypeDefinition="iMMAS:IndustrialData" />
<Object SymbolicName="MOSIE:autoMinValue" TypeDefinition="iMMAS:IndustrialData" />
<Object SymbolicName="MOSIE:calMaxValue" TypeDefinition="iMMAS:IndustrialData" />
<Object SymbolicName="MOSIE:calMinValue" TypeDefinition="iMMAS:IndustrialData" />
<Object SymbolicName="MOSIE:manMaxValue" TypeDefinition="iMMAS:IndustrialData" />
<Object SymbolicName="MOSIE:manMinValue" TypeDefinition="iMMAS:IndustrialData" />
<Property SymbolicName="MOSIE:modeAttr" DataType="MOSIE:AnalogInputMode" />
<Method SymbolicName="MOSIE:getValOp" TypeDefinition="iMMAS:iMMASMethod">
<OutputArguments>
<Argument Name="value" DataType="OpcUa:Float" />
</OutputArguments>
</Method>
<Method SymbolicName="MOSIE:setValOp" TypeDefinition="iMMAS:iMMASMethod">
<InputArguments>
<Argument Name="mode" DataType="iMMAS:Float" />
</InputArguments>
</Method>
<Method SymbolicName="MOSIE:getMaxValue" TypeDefinition="iMMAS:iMMASMethod">
<InputArguments>
<Argument Name="mode" DataType="MOSIE:AnalogInputMode" />
</InputArguments>
<OutputArguments>
<Argument Name="r" DataType="OpcUa:Float" />
</OutputArguments>
</Method>
<Method SymbolicName="MOSIE:getMinValue" TypeDefinition="iMMAS:iMMASMethod">
<InputArguments>
<Argument Name="mode" DataType="MOSIE:AnalogInputMode" />
</InputArguments>
<OutputArguments>
<Argument Name="r" DataType="OpcUa:Float" />
</OutputArguments>
```

```
</Method>
<Method SymbolicName="MOSIE:getmode" TypeDefinition="iMMAS:iMMASMethod">
<OutputArguments>
<Argument Name="r" DataType="MOSIE:AnalogInputMode" />
</OutputArguments>
</Method>
<Method SymbolicName="MOSIE:setMode" TypeDefinition="iMMAS:iMMASMethod">
<InputArguments>
<Argument Name="mode" DataType="MOSIE:AnalogInputMode" />
</InputArguments>
</Method>
</Children>
</ObjectType>
```

6.5.2. GENERACIÓN DE LOS FICHEROS BINARIOS Y XML FINALES

Una vez que tenemos recogidos todos los elementos de los modelos conceptuales en un único fichero XML, podemos proceder a compilar este fichero con la herramienta Opc.Ua.ModelCompiler.exe. Esta herramienta puede generará código fuente C# y ANSI C a partir de los archivos XML, que hemos creado anteriormente, además también nos genera numerosos archivos CSV que contienen NodeIds, códigos de error y atributos de todas las definiciones Esta herramienta de compilación, ajusta las definiciones del archivo XML al esquema definido en UA Model Design.xsd, generando lo siguiente:

- Un NodeSet que se ajuste al esquema definido en el anexo F de la especificación número 6.
- Un fichero XSD y otro BSD (definido en la especificación 3 del Anexo C) que describe cualquier tipo de dato.
- Definiciones de clases y constantes adecuadas para su uso en .NET.
- Otro tipo de archivos, por ejemplo binarios, para desplegar un modelo de información en un servidor OPC UA desarrollado en .NET, utilizando las clases creadas.
- Un archivo CSV que contiene identificadores numéricos.

Esta herramienta genera una serie de archivos que facilitan la utilización de modelos construidos a partir del modelo información de OPC UA, en nuestros caso iMMAS y MOSIE, en servidores OPC UA. Además crea código en .NET que puede ser utilizado para desarrollar un servidor OPC UA específico, que se adapte a las necesidades concretas de nuestro modelo.

6.5.3. DESPLIEGUE EN UN SERVIDOR OPC UA

Gracias a la herramienta de compilación que ofrece la OPC Foundation, podemos generar todos los ficheros necesarios para introducir iMMAS y MOSIE en una solución software que implemente un servidor OPC UA. Llegados a este punto tenemos dos opciones, utilizar una servidor OPC UA comercial desarrollado por algún fabricante de software o desarrollar nuestro propio servidor OPC UA. En ambos casos el objetivo es poder desplegar iMMAS y MOSIE e una plataforma software que contemple tanto el modelo de comunicaciones y el modelo de información del estándar UA, además otras características propias del estándar, como la seguridad.

Después de realizar una exploración de las soluciones software existentes, la cuales han sido recogidas en la tabla 6.34, llegamos a la conclusión de que la gran mayoría de los servidores OPC UA comerciales existentes soporta el protocolo de comunicaciones, pero muy pocos dan soporte para introducir modelos complejos. Las soluciones que si contemplan el modelo de información, que son 3 nadas mas, no ofrecen drivers de comunicación para dispositivos industriales como PLCs, por lo que aunque despleguemos iMMAS y MOSIE en ellos, los modelos creados a partir de ellos no se podrán comunicar con los dispositivos industriales, a menos de que se desarrollen drivers para estos servidores, utilizando las SDK que estos fabricantes ofrecen.

Como podemos observar el tabla 6.34, los principales fabricantes de servidores OPC se limitan a utilizar una envolvente sobre sus soluciones basadas en el OPC clásico. Por tanto, en general ofrecen la información del servidor a los clientes utilizando el modelo de comunicaciones de OPC UA, pero no tienen soporte para crear modelos, tan solo ofrecen la posibilidad de crear tipos de datos simples iguales a los del OPC clásico.

Fabricante	Producto	S.0	Versiones OPC	Modelado	Seguridad	Drivers
Kepware	Kepware Suit.	Microsfot Window.s	OPC DA V1.0a, OPC DA V 2.0, OPC DA V 2.05a, OPC DA V 3.0, OPC AE, OPC UA.	No	Authentication, encryption, electronic signature.	Multi-Drivers Supports connection to almost any industrial device.
Matrikon.	Matrikon OPC.	Microsfot Windows.	OPC DA V1.0a, OPC DA V 2.0, OPC DA V2.05a, OPC DA V 3.0, OPC AE, OPC HDA, OPC UA.	No	Authentication, encryption, electronic signature	Multi-Drivers Supports connection to almost any industrial device.
Softing Industrial Automation.	Softing.	Microsoft Windows, Linux.	(OPC UA), OPC DA V1.0a, OPC DA V 2.0, OPC DA V2.05a, OPC DA V 3.0, OPC XML-DA, OPC AE, OPC UA.	Yes	Encryption, electronic signature.	SIMATIC S7 / S5, Rockwell CLX, SLC-500, Mitsubishi (Melsec-Q), Schneider, Phoenix, Beckhoff, SNMP, ModBus
IBH Softec.	IBH OPC Server	Microsoft Windows.	OPC DA V1.0a, OPC DA V 2.0, OPC DA V2.05a, OPC AE, OPC HDA	No	No	SIMATIC S7 / S5
Deltalogic	OPC Server	Microsoft Windows	OPC DA V1.0a , OPC DA V 2.0, OPC DA V2.05a, OPC DA V 3.0, OPC XML-DA	No	No	SIMATIC S7 / S5
Prosys	OPC Server, OPC UA SDK	Microsoft Windows, Linux (OPC UA), OS X.	OPC DA V1.0a, OPC DA V2.05a, OPC DA V 3.0, OPC UA.	Yes	Authentication, encryption, electronic signature	None
Unified Automation	OPC UA Server	Microsoft Windows, Linux, QNX.	OPC-UA	Yes	Authentication, encryption, electronic signature.	None
OPCLabs	Quick OPC	Microsoft Windows	OPC DA V1.0a, OPC DA V2.05a, OPC DA V 3.0, OPC UA.	No	Authentication, encryption, electronic signature.	None

Tabla 6.34: Principales soluciones comerciales de servidores OPC UA. Julio de 2018.

Por ello, se ha optado por desarrollar un servidor OPC UA propio en .NET, en el que introducir los modelos de iMMAS y MOSIE generados a partir de la herramienta de compilación de la OPC Foundation. Ademas también se han desarrollado drivers propios de comunicaciones con dispositivos industriales, con los que poder comunicar los modelos alojados en el sevidor OPC UA con el PLC que controla el proceso industrial.

Para la implementación del servidor OPC UA, se ha utilizado el STACK en .NET que ofrece la OPC Foundation, que ofrece todas las funciones básicas con para desarrollar completamente el servidor OPC UA. A continuación podemos ver un trozo de código de dicho servidor, que se encarga de alojar iMMAS y MOSIE en el servidor UA:

```
Carga de nodos predefinods de iMMAS:
protected override NodeStateCollection LoadPredefinedNodes(ISystemContext
    context)
{

NodeStateCollection predefinedNodes = new NodeStateCollection();
predefinedNodes.LoadFromBinaryResource(context, "OPCUAV1.Common.iMMAS.iMMAS.
    PredefinedNodes.uanodes", null, true);

return predefinedNodes;
}

Carga de nodos predefinods de MOSIE:
protected override NodeStateCollection LoadPredefinedNodes(ISystemContext
    context)
{

NodeStateCollection predefinedNodes = new NodeStateCollection();
predefinedNodes.LoadFromBinaryResource(context, "OPCUAV1.Common.MOSIE.MOSIE.
    PredefinedNodes.uanodes", null, true);

return predefinedNodes;
}
```

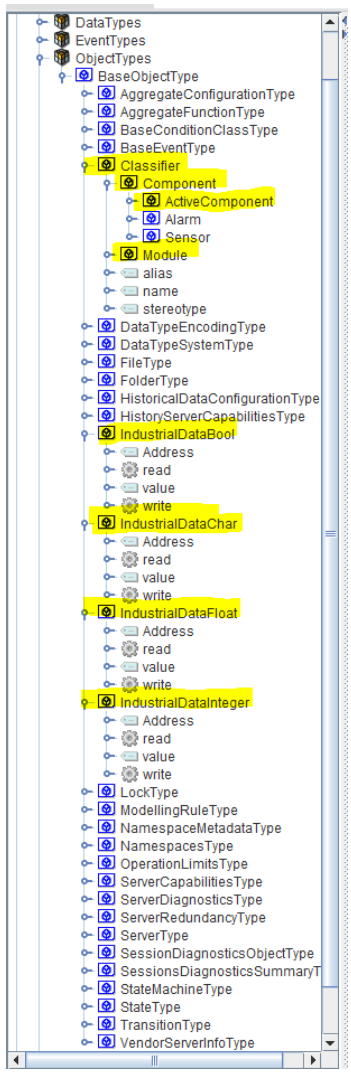


Figura 6.12: Visualización de algunos de los elementos transformados de iMMAS (Amarillo) desde un cliente UA.

Como se puede ver se han cargado los nodos predefinidos de iMMAS y MOSIE, a partir de los ficheros binarios generados por la herramienta de compilación:

```
predefinedNodes.LoadFromBinaryResource(context, "OPCUAV1.Common.iMMAS.iMMAS.
    PredefinedNodes.uanodes", null, true);

predefinedNodes.LoadFromBinaryResource(context, "OPCUAV1.Common.MOSIE.MOSIE.
    PredefinedNodes.uanodes", null, true);
```

La parte del desarrollo del driver de comunicaciones con el dispositivo industrial será explicada en el caso de estudio, ya que la implementación de este driver dependerá del dispositivo industrial elegido, para el sistema industrial en cuestión. Por último podemos ver en la figura 6.12 como vería un cliente UA por los conceptos de iMMAS y MOSIE de nuestro servidor.

6.6. CONCLUSIONES

El estándar OPC UA ofrece un modelo de información, bastante rico y con el que podemos llegar a desplegar modelos PIM diseñados con iMMAS aplicando una serie de reglas de transformación para que se obtenga como resultado un modelo PSM "por construcciónçonforme a iMMAS, ya que deriva del modelo PIM, y conforme a OPC UA, ya que el modelo PSM se puede desplegar en un servidor OPC UA.

Para construir iMMAS en OPC UA, se ha seguido una estrategia jerárquica, en la que se han utilizado los elementos que ofrece OPC UA, BASE, ObjecType y BaseObjectType, como los elementos que se encuentra en la parte más alta de la jerarquía, los cuales además hacen de pilares sobres los que se construye iMMAS en OPC UA. Estos conceptos de OPC UA poseen un nivel de abstracción bastante elevado y de ellos por herencia se van introduciendo los conceptos de los paquete de iMMAS, empezando por el paquete BASE, el cual recoge los elementos esenciales de iMMAS, una vez transformado este paquete, los demás conceptos de iMMAS se pueden construir a partir de estos conceptos, por lo que el proceso se simplifica bastante.

En cuanto a MOSIE, el proceso de transformación a OPC UA se simplifica bastante, ya que MOSIE se basa en la creación de perfiles en los que se recogen una serie de modelos, creados a partir de los conceptos de iMMAS, como estos ya han sido transformados a conceptos de OPC UA, se pueden transformar utilizando las transformaciones utilizadas para iMMAS. Todos los conceptos de MOSIE,

han sido transformados a OPC UA utilizando la misma estrategia jerarquíca utilizada en iMMAS, solo que en el caso de MOSIE los elementos base y pilares de las transformaciones han sido elementos de iMMAS. Al disponer de iMMAS y de MOSIE, como metamodelos alojados en el espacio de direcciones de un servidor OPC UA, podemos crear modelos complejos a partir de iMMAS o partir de MOSIE por construcción, lo que facilita la creación y el despliegue de aplicaciones en OPC UA, gracias a nuestra propuesta MDE basada en iMMAS y MOSIE. Por lo que iMMAS y MOSIE pueden ser vistos como un forma de modelar aplicaciones industriales, las cuales pueden ser desplegadas en un sistema software industrial, en este caso en un servidor OPC UA.

7

DESPLIEGUE DE iMMAS EN DISPOSITIVOS INDUSTRIALES

"Low-level programming is good for the programmer's soul."
John Carmack

7.1. INTRODUCCIÓN

En el anterior capítulo hemos visto como, a partir de un modelo PIM del sistema industrial conforme a iMMAS/MOSIE, se puede obtener un modelo PSM conforme a OPC UA aplicando un conjunto de reglas de transformación. El modelo PSM final se puede desplegar en un servidor OPC UA y servir de modelo de intercambio de datos entre dispositivos industriales y los sistemas de información del sistema industrial.

En este capítulo, de igual modo, vamos a estudiar con detalle como, a partir del mismo modelo PIM del sistema industrial modelado en iMMAS/MOSIE, es posible obtener un modelo PSM acorde con IEC61131-3 que puede ser desplegado directamente en los propios dispositivos PLCs. Para ello se describe en este capítulo como se han obtenido las reglas de transformación entre los dos lenguajes, iMMAS e IEC61131-3, y el proceso seguido para el despliegue del modelo PSM en los dispositivos industriales.

En este dominio hay que tener en cuenta que no todos los fabricantes siguen las normas del IEC61131-3, por lo que hemos encontrado problemas para la aplicación de las reglas de transformación, teniendo incluso que diseñar reglas de transformación específicas para algunos fabricantes como Siemens.

7.2. DESPLIEGUE DE iMMAS/MOSIE EN DISPOSITIVOS INDUSTRIALES CON IEC 61131-3

Para poder desplegar un modelo de iMMAS o MOSIE en un dispositivo industrial como un PLC, se debe de realizar una correlación entre los conceptos de iMMAS/MOSIE y el estándar IEC 61131-3, ya que como se ha explicado en el capítulo 4 este estándar fue creado para tratar de unificar los lenguajes de programación de los diferentes dispositivos o controladores industriales. En este estándar se recogen las características que debe de cumplir un lenguaje de programación de estos dispositivos, con el objetivo de estandarizar y homogeneizar la programación de PLCs.

De entre los aspectos que se recogen en este estándar, podemos destacar la necesidad de que los datos sean de un tipo concreto, para evitar por ejemplo que datos de tipo fecha sean divididos entre datos de tipo boolean. Así un lenguaje de programación para PLCs debe de ser tipado, como ocurre con los lenguajes tradicionales de programación. En esta línea la norma también contempla la posibilidad de que el usuario/programador pueda definir sus propios tipos de datos. Esto recibe el nombre de *derivación (derivated)* o *definición de tipos (type definition)*. La definición de tipos solo se puede hacer de forma textual. Según esta norma para definir tipos de datos hay que utilizar las palabras reservadas TYPE ... END_TYPE como se muestra en el siguiente ejemplo:

```
TYPE
(* definicion de tipo directa de un tipo de dato IEC *)
Flotante: LREAL;
(* definicion de tipo directa de un tipo de dato IEC *)
Marcha: BOOL;
(* definicion de tipo utilizando un tipo creado por el usuario *)
velocidad: Flotante;
(* definicion de tipo utilizando un tipo creado por el usuario *)
Arranca: marcha;
END_TYPE
```

Además de ofrecer la posibilidad de que un usuario pueda crear tipos de datos, el estándar también contempla tipos de datos complejos como los arrays y las estructuras. Esta últimas nos permiten crear conjuntos de variables de diferentes tipos, a los que se pueden acceder utilizando el nombre de la estructura seguido de un punto y el nombre de la variable.

Combinando la definición de tipos de usuario y las estructuras podríamos tener definiciones de tipos como las siguientes:

```
TYPE
Prueba:
STRUCT
Fase: UINT;
Temperatura:LREAL;
Start: BOOL;
END_STRUCT;
END_TYPE
```

Mediante esta construcción hemos creado un tipo de dato definido por el usuario en el que agrupamos 3 variables de tipos diferentes. A partir de esta construcción podríamos definir variables de este tipo:

```
T1: Prueba;
T1.Start:= True;
```

Apoyándonos en la definición de tipos de datos y en las estructuras de tipos de datos complejos, podemos realizar una correspondencia entre iMMAS/MOSIE y la norma IEC 61131-3, si bien es cierto que perderíamos ciertos aspectos que contemplan iMMAS o MOSIE, como son los métodos definidos para los conceptos de iMMAS/MOSIE, ya que el estándar IEC 61131-3 no contempla el concepto de método de un objeto o elemento, por lo menos en sus primeras versiones [1].

Pero esta restricción no implica que no se pueda desplegar iMMAS/MOSIE, o por lo menos realizar ciertas transformaciones de los conceptos que ofrecen, ya que iMMAS/MOSIE proporcionan conceptos con los que podemos estructurar y organizar los datos de un programa para un PLC, simplificando la programación del mismo. Además, podremos construir modelos de datos a partir de estos conceptos, por lo que estaremos llevando modelos al dominio de la automatización industrial. Por otro lado, para el caso del modelado del comportamiento que recoge iMMAS, podemos realizar una correspondencia utilizando los bloques de función o FBs que recoge esta especificación.

Estas correlaciones entre iMMAS y el estándar IEC 61131-3 nos permitiría desplegar los conceptos definidos en los modelos PIM de iMMAS en cualquier dispositivo industrial cuyo lenguaje de programación siga la norma IEC 61131-3. Por tanto, podremos obtener un modelo PSM acorde a IEC 61131-3 que, dependiendo del fabricante, podrá desplegarse en las plataformas de desarrollo de PLCs que siga dicha especificación. Esto permitiría posteriormente poder llevar a cabo el proceso de compilación con el dispositivo PLC concreto, y cargar el programa ob-

tenido en el dispositivo PLC específico.

7.3. TRANSFORMACIÓN DE iMMAS A IEC 61131-3

Para realizar la transformación de los modelos PIM modelados en iMMAS a los modelos PSM de OPC UA se ha tenido que estudiar las correspondencias que hay entre el lenguaje de modelado de iMMAS y OPC UA, para obtener finalmente una reglas de transformación. Del mismo modo, se ha buscado la correspondencia entre los elementos de iMMAS y el estándar IEC 61131-3, aunque como veremos, nos vamos a encontrar con ciertas limitaciones a la hora de realizar las trasformaciones entre iMMAS y la especificación IEC 61131-3.

Para la obtención de las reglas de transformación se ha seguido la misma estrategia empleada en el capítulo anterior:

- Identificación directa de los elementos sintácticos entre los dos lenguajes.
- Definición de los elementos sintácticos en IEC 61131-3 equivalentes a iMMAS.

7.3.1. REGLAS DE TRANSFORMACIÓN DIRECTA EN IEC 61131-3

Los elementos de este paquete iMMAS son: *NamedElement, Type, TypedElement, Asociation, DataType, Operation, Parameter, Property, Classifier, Module, Component, ActiveComponent, IndustrialData, Value y Address*. Para estos conceptos se ha tratado de realizar su correlación a la especificación IEC 611313-3. No obstante, para algunos ellos no se ha encontrado una correspondencia clara, debido a que la especificación IEC 611313-3 recoge sobre todo conceptos relacionados con la programación estructurada, dejando fuera o contemplando parcialmente paradigmas como la programación orientada a objetos. En consecuencia no contamos con conceptos como operación/método, asociación/relación y propiedad/atributo. Sin embargo, el estándar IEC 61131-3 si contempla la extensión entre UDTs y FBs, por lo que aprovechando esta propiedad podremos realizar ciertas transformaciones, ya que el mecanismos de extensión se puede aplicar tanto a UDTs como a FBs.

En la tabla 7.1 tenemos un resumen de las transformaciones que se han realizado:

Concepto de iMMAS	Concepto de IEC 61131-3	Correpondencia
Type	Type	Directa
DataType	Type	Directa
TypeElement	Struct Type	Directa
Operation	-	FB
Parameter	-	Variable de entrada/salida de un FB
Property	-	Variable de instancia de un FB
Asociation	-	Extension de UDTs o FBs
Classifier	-	UDT
Componet	-	UDT
ActiveComponet	-	UDT
Module	-	UDT
IndustrialData	-	Propia de cada Framework de programación

Tabla 7.1: Resumen de las correspondencias básicas entre iMMAS base y IEC 611313-3.

Como se puede ver en la tabla anterior, a excepción de los conceptos relacionados con la definición de tipos de datos, que si tiene una relación directa, para los demás conceptos no se puede realizar una correlación de forma tan directa, ya que la especificación IEC 61131-3 no los recoge directamente. Esto ocurre en el caso de Operation, Parameter, Property y Asociation. Sin embargo, adaptando ciertas características de esta especificación, si podemos llegar a establecer una relación que puede ayudarnos a la hora de transformar los conceptos de iMMAS.

7.3.2. DEFINICIÓN DE LOS ELEMENTOS SINTÁCTICOS EN IEC 61131-3

En el caso de los conceptos *Classifier, Component, ActiveComponent y Module* del paquete *Base* tendremos que utilizar la extensión de tipos o de bloques de función para poder realizar su correlación a IEC 61131-3. Aunque el mecanismo de extensión puede ser visto como una forma conceptual de herencia, debemos de realizar de forma separada la extensión de las UDTs y de la extensión de los FBs, ya que la especificación no contempla el concepto de objeto, en los que los mecanismos de herencia contemple a la vez métodos y propiedades de los objetos. Al no disponer del concepto de objeto y de herencia entre objetos en IEC 61131-3 [135], tenemos que ser muy selectivos a la hora de escoger el concepto

que vamos a trasformar como elemento base y del cual van a partir todas las definiciones posteriores. Como veremos en los siguientes apartados para cada caso se ha elegido un concepto derivado del concepto *classifier* como elemento base, del cual derivan los elementos *Module* y *Component*. Sobre el concepto *Module* se han construido los conceptos básicos de cada paquete. En el caso de los elementos más complejos, con mayor nivel de abstracción, se ha utilizado el concepto *Component* como concepto base y del que derivan el resto de los elementos.

Como se puede ver en la tabla 7.1 para este paquete se tienen que construir los conceptos de *Classifier*, *Componente*, *ActiveComponent* y *Module*. Como estos elementos no poseen ningún método, basta con crear una unidad de datos definida por el usuario o UDT para realizar la transformación de estos elementos a la especificación IEC 61131-3. Como veremos a continuación todos estos conceptos extiende de la UDT creada para el elemento *Classifier*.

Classifier:

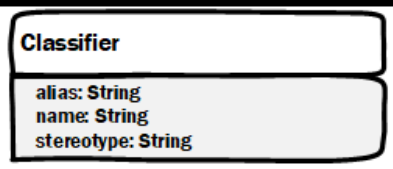
iMMAS	IEC 61131-3
	TYPE Classifier : STRUCT alias:STRING; name:STRING; isAbstract:BOOL; stereotype:STRING; END_STRUCT END_TYPE

Tabla 7.2: Transformación realizada para Classifier.

La UDT creada para *Classifier* agrupa cuatro variables que representan 4 atributos del elemento *Classifier*: 3 propios del elemento *Classifier* y *name* de *NamedElement*. Dado que no existe la herencia en IEC 61131-3, todos los atributos heredados tienen que incluirse directamente en el elemento hijo o especializados. En nuestro caso se ha introducido esta propiedad en la UDT que representa a *Classifier*.

Module:

iMMAS	IEC 61131-3

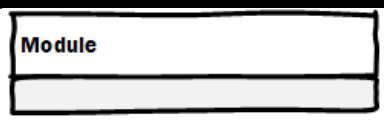
	`TYPE Module EXTENDS Classifier :`
	`STRUCT`
	`END_STRUCT`
	`END_TYPE`

Tabla 7.3: Transformación realizada para Module.

Para el caso de *Module*, se ha creado una UDT que extiende de la UDT *Classifier*, por lo que *Module* tiene todas las variables definidas en la estructura de datos de *Classifier*.

Component:

iMMAS	IEC 61131-3

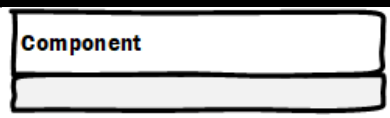
	`TYPE Component EXTENDS Classifier :`
	`STRUCT`
	`END_STRUCT`
	`END_TYPE`

Tabla 7.4: Transformación realizada para Component.

Component al igual que *Module* también extiende de *Classifier*.

ActiveComponent:

iMMAS	IEC 61131-3

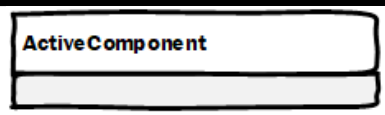
	`TYPE ActiveComponent EXTENDS Component:`
	`STRUCT`
	`END_STRUCT`
	`END_TYPE`

Tabla 7.5: Transformación realizada para ActiveComponent.

ActiveComponent extiende de la UDT de *Component*, tal y como especifica iMMAS.

IndustrialData:

IndustrialData está pensado para direccionar y acceder a la información que maneja un dispositivo industrial. En el caso que estamos tratando, donde pretendemos desplegar iMMAS en un dispositivo industrial, este concepto se encuentra recogido de forma subyacente en la plataforma de desarrollo. Esto es así ya que cuando se declaran variables o se trabaja con estructuras de datos, éstas reciben una dirección física de memoria por parte del compilador y es en esta dirección física donde se alojarán los datos para esta variable o estructura de datos. Por ello IndustriaData es un concepto inherente ya contemplado en los entornos de programación para dispositivos industriales y, por lo tanto, *industrialData* tiene una correlación directa recogida por el compilador de los diferentes entornos de programación para dispositivos industriales. La correlación realizada para este paquete se basa en la utilización de UDTs, ya que los elementos de iMMAS, a excepción de *industrialData*, solo poseen atributos, que pueden ser recogidos en estructuras pensadas para la definición de tipos de datos. Posteriormente, estas estructuras se utilizaran para realizar la correlación del resto de elementos de este paquete.

7.3.3. PACKAGE

Este paquete está formado por un único concepto *Package*, pensado para agrupar conceptos similares. En IEC 61131-3 este concepto no tiene cabida ya que el estándar no completa la utilización de paquetes de elementos. Por ello para este paquete no se ha contemplado ninguna trasformación. Por otro lado, en la gran mayoría de las plataformas de desarrollo para dispositivos industriales, existe la posibilidad de organizar tanto UDTs como FBs en carpetas, pudiendo agrupar así UDTs y FBs de acuerdo al criterio del programador, generando así estructuras organizativas jerárquicas en forma de árbol, como se puede observar en la figura 7.1.

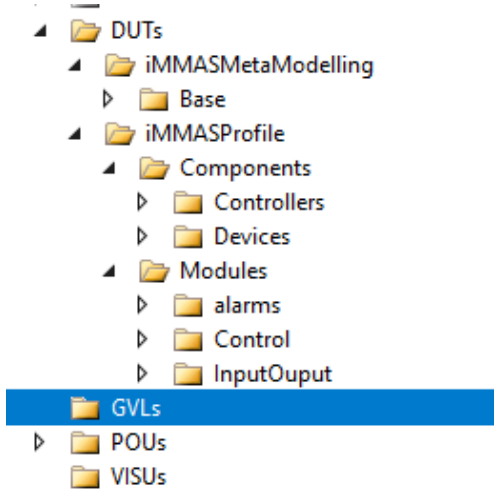


Figura 7.1: Ejemplo de estructura de carpetas para albergar los conceptos de iMMAS.

7.3.4. EL PAQUETE TYPE

El paquete *Type* recoge los tipos básicos de iMMAS. En principio iMMAS contempla cualquier tipo de dato primitivo, gracias al concepto *PrimitiveType*, por lo que podremos realizar una correlación de todos los tipos de datos primitivos de la especificación IEC 61131-3. Por simplicidad a la hora de realizar la correlación entre iMMAS e IEC 61131-3, hemos realizado una transformación de los 4 tipos de datos más importantes: *Double, Int, Boolean* y *String*. Estos tipos se puede ver en la tabla 7.6:

Concepto de iMMAS	Concepto de IEC 61131-3	Correpondencia
Int	Int	Directa
Double	Real	Directa
Boolean	Bool	Directa
String	String	Directa

Tabla 7.6: Resumen de las correspondencias entre iMMAS Type y IEC 611313-3.

7.3.5. EL PAQUETE BEHAVIOUR EN IEC 61131-3

El paquete Behavior recoge todos los conceptos necesarios para crear máquinas de estados finitas. Contemplando así conceptos como estado y transición. Estas máquinas de estado pueden utilizarse para modelar el comportamiento dinámico de un sistema industrial. Pero una vez más nos encontramos, que debi-

do a que las transformaciones de iMMAS se están realizando utilizando lenguaje estructurado, recogido en la especificación. IEC 61131-3 nos encontramos con ciertas limitaciones inherentes a este lenguaje, al contrario que ocurría con el estándar OPC UA, la especificación ICE 61131-3 contempla tan solo el concepto de POU o unidad organizativo de programa, existiendo 3 tipos principales de POU, funciones FCs, bloques de función FBs y PROG o programa principal, y por lo tanto no contempla conceptos como método, estado o transición.

Quizás se podría hacer una analogía entre un bloque de función y un estado, pero sería una correlación muy débil, ya que un estado necesita tener asociado métodos o eventos con los que disparar transiciones desde él hacia otros estados, o desde otros estados hacía él. Por otra parte, habría otros aspectos sin cubrir como, por ejemplo, la definición de transiciones o la asignación de la máquina de estado a conceptos como actuador. Por tanto, para los conceptos de este paquete tampoco se ha realizado ninguna trasformación. Es decir, las abstracciones sobre el comportamiento de un elemento que recoge iMMAS no se podrá utilizar de forma directa en ninguna plataforma de programación de dispositivos industriales basados en esta norma.

Sin embargo, como se verá para la transformaciones realizadas de los elementos de MOSIE, la utilización de FBs para realizar la correlación de elementos que contemplan operaciones nos proporciona la posibilidad de introducir el código necesario para programar la funcionalidad de los diferentes conceptos de MOSIE. Por lo que de forma indirecta podemos introducir el código necesario para representar el comportamiento de dichos conceptos, especialmente de todos los que extiendan de *ActiveComponent*. Recordemos que según iMMAS solos los conceptos que descienden de *ActiveComponent* pueden tener máquinas de estados. Aunque este código debe de ser generado por el programador o ingeniero que esté desarrollando el sistema, en las plantillas generadas para MOSIE se especifican los elementos susceptibles de tener este código, dejando al programador la libertad de personalizarlo.

7.4. TRANSFORMACIÓN DE MOSIE A IEC 61131-3

Los perfiles definidos por MOISE pretenden facilitar y ofrecer un conjunto de conceptos ya predefinidos, que faciliten la construcción de modelos para siste-

mas industriales software. Al igual que en el caso de OPC UA, las correlaciones de MOSIE se van a cimentar en la utilización de las correlaciones ya realizadas para iMMAS a IEC 61131-3.

Para el caso de MOSIE un concepto muy importante y que no ha sido utilizado en las transformaciones de iMMAS, son los FBs o bloques de función. Los FBs van a ser utilizados para realizar la correlación de todos los elementos que posean métodos. En dichos FBs se dispondrán de tantas variables de entrada como métodos ”set” (escritura) tenga el concepto y, de tantas variables de salida como métodos ”get” (lectura) tenga también el concepto, estableciendo una correspondencia entre métodos ”set” y variables de entrada del FB y métodos ”get” y variables de salida del FB.

Concepto de MOSIE	Concepto de IEC 61131-3	Correpondencia
Elemento con métodos y Propiedades	UDT + FB	Directa
Métodos de tipo Set	Variables de entra FB (VAR_INPUT)	Directa
Métodos de tipo Get	Variables de salida FB (VAR_OUPIT)	Directa
Elemento Asociado al FB	Variable local del tipo (VAR)	Directa

Tabla 7.7: Resumen de las correspondencias general entre los elementos MOSIE y IEC 611313-3.

7.4.1. EL PAQUETE MODULES EN IEC 61131-3

Al igual que hicimos en el caso de OPC UA, vamos a empezar transformando los elementos agrupados en los sub-paquetes de *Modules*, ya que estos nos hará falta para construir los elementos del paquete *Components* y al igual que en el caso de las transformaciones realizadas en OPC UA, nos vamos a basar en los elementos de iMMAS transformados a IEC 61131-3.

EL PAQUETE INPUT/OUPUT EN IEC 61131-3

Todos los conceptos de este paquete extienden del concepto *InputOutputModule*. Este concepto deriva a su vez de *Module*, concepto que se encuentra recogido en el paquete base de iMMAS. Por tanto tendremos que crear en primer lugar la regla de transformación para *InputOutputModule* y construir el resto de los elementos de este paquete a partir de dicha transformación.

InputOutputModel:

MOSIE	UDT	FB

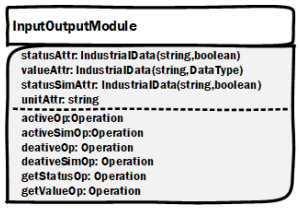
	TYPE InputOutPutModule
	EXTENDS Module :
	STRUCT
	status:BOOL;
	statusSim:BOOL;
	value:ANY;
	unit:String;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		InputOutputModuleFB
		VAR_INPUT
		i_status:BOOL;
		i_statusSim:BOOL;
		END_VAR
		VAR_OUTPUT
		o_status:BOOL;
		o_statusSim:BOOL;
		o_value:ANY;
		END_VAR
		VAR
		local:
		InputOutputModule;
		END_VAR
		//ENTRADAS
		local.statusSim:=
		i_status;
		local.statusSim:=
		i_statusSim;
		//TO DO (Customize
		Code)
		//SALIDAS
		o_status:=local.
		status;
		o_statusSim:=local.
		statusSim;
		o_value:=local.value;
		END_FUNCTION_BLOCK

Tabla 7.8: Transformación realizada para InputOutPutModule.

Para este elemento se han creado una UDT y un FB. En la UDT *InputOutPutModule* se ha agrupado todos los atributos de este elemento: *status, statusSim* y *Value*. El atributo *Value* se ha definido con tipo *ANY*, ya que este tipo de variables nos permite albergar cualquier tipo de dato, *entero*, *flotante*, *String*, *Bool*, etc., conforme a la especificación en MOSIE. Además de la UDT para *InputOutPutModule* se ha creado un FB *InputOutputModuleFB* con una variable local y persistente (por ser un FB) de tipo InputOutputModule. Esta variable tiene dos propósitos, por un lado relaciona el FB con la UDT correspondiente y, por otra parte, nos sirve para realizar una correlación entre las operaciones *InputOutModule* y la especificación IEC 61131-3, ya que este FB dispone de 2 variables de entrada en la UDT "local", *i_status* y *i_statusSim,* que representa las operaciones de modificación active, *activeSim*, *deactive* y *deactiveSim*. Por otra parte, tiene tres variables de salida, que hacen referencia a las operaciones de consulta *getValue*, *getStatusSim* y *getStatus*, ya que devuelven el contenido de las variables *o_statusSim*, *o_status* y *o_value* alojadas en la UDT "local". Las variables de entrada y salida actúan de interfaz con respecto a la variable "local" del FB cuyo valor persiste fuera del ámbito

del FB. Por último la plantilla dispone de una zona *//TO DO (Customize Code)* en la que el programador o ingeniero puede introducir todo el código que necesite.

Como se puede ver para el concepto *InputOutModule* se han creado una UDT y un FB a modo de plantilla para poder realizar una transformación lo mas completa posible a la especificacion IEC 61131-3, ya que este elemento es la base para el resto de los elementos de este paquete. El FB creado nos puede servir como una plantilla inicial con la que empezar a programar nuestro dispositivo industrial, y la UDT creada extiende de *Module* para que sea conforme a las especificaciones de MOSIE.

digitalOutput:

MOSIE	UDT	FB

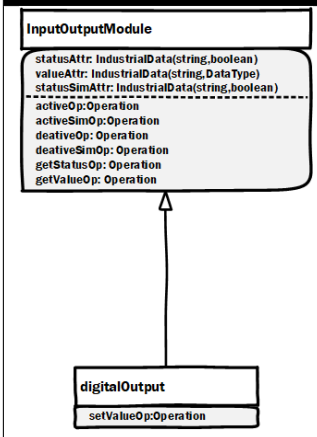
	`TYPE DigitalOutput`
	`EXTENDS`
	`  InputOutPutModule:`
	`STRUCT`
	`END_STRUCT`
	`END_TYPE`
		`FUNCTION_BLOCK`
		`  digitalOutputFB`
		`EXTENDS`
		`  InputOutputModuleFB`
		
		`VAR_INPUT`
		`  i_value:BOOL;`
		`END_VAR`
		`VAR_OUTPUT`
		`END_VAR`
		`VAR`
		`  diout_local:`
		`digitalOutput;`
		`END_VAR`
		`//ENTRADAS`
		`  diout_local.value:=`
		`  i_value;`
		`//TO DO (Customize`
		`  Code)`
		`END_FUNCTION_BLOCK`

Tabla 7.9: Transformación realizada para DigitalOutPut.

Según se especifica en MOSIE, *DigitalOutput* es una extensión de *InputOutputModule* para salidas de tipo digital que se manejan con datos de tipo *booleano*. Por ello el *DataType* del atributo *value* debe de ser de tipo *Bool*.

Por lo tanto, la correlación que se ha realizado de *DigitalOutput* consiste en la creación de una nueva UDT que extiende de la UDT *InputOutputModule* y un FB que extiende del FB asociado a *InputOutputModule*: *InputOutputModuleFB*. Este FB solo contiene una variable de entrada *i_value* de tipo BOOL, la cual se asigna a la variable contenida en la UDT local *diout_local.value*. Con este FB conseguimos transformar la única operación *setVal* que posee este módulo.

AutoManDigitalOuput:

MOSIE	UDT	FB

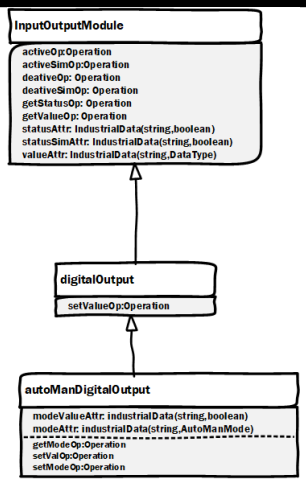
	`TYPE`
	`  AutoManDigitalOutput`
	
	`EXTENDS DigitalOutput`
	`  :`
	`STRUCT`
	`  manValue:ANY;`
	`  mode:AutoManMode;`
	`END_STRUCT`
	`END_TYPE`
		`FUNCTION_BLOCK`
		`  AutoManDigitalOutputFB`
		`EXTENDS DigitalOutputFB`
		`VAR_INPUT`
		`  i_mode:AutoManMode;`
		`  i_manValue:BOOL;`
		`END_VAR`
		`VAR_OUTPUT`
		`  o_mode:AutoManMode;`
		`  o_manValue:BOOL;`
		`END_VAR`
		`VAR`
		`  an_local:`
		`  AutoManDigitalOutput;`
		`END_VAR`
		`///ENTRADAS`
		`  an_local.mode:=i_mode;`
		`//TO DO (Customize Code)`
		`IF  (an_local.mode =`
		`  AutoManMode.auto) THEN`
		`  an_local.value:=i_value;`
		`ELSIF (an_local.mode =`
		`  AutoManMode.man) THEN`
		`  an_local.value:=i_manValue;`
		`END_IF;`
		`//SALIDAS`
		`  o_mode:=an_local.mode;`
		`END_FUNCTION_BLOCK`

Tabla 7.10: Transformación realizada para AutoManDigitalOutput.

AutoManDigitalOutput se ha transformando mediante la creación tanto de una nueva UDT y de un nuevo FB. Ambos extiende de la UDT y del FB creados para *DigitalOutput* tal y como se especifica en MOSIE, ya que AutoManDigitalOutput extiende de *DigitalOutput*. Las tres operaciones que posee este módulo se recogen el en FB mediante 2 variables de entrada *i_mode* y *i_manValue* para las operaciones *setMode* y *setVal* y una variable de salida *o_mode* para la operación *getMode*. Además posee una variable local del tipo de *AutoManDigitalOutput* que corresponde a la UDT creada. Esta contiene las variables *manValue* que es de tipo ANY tal y como se especifica en MOSIE (DataType) y mode que es de tipo *AutoManMode*. *AutoManMode* es un tipo enumerado, que se ha definido expresamente para recoger el modo de trabajo automático o manual. en automático es el PLC el que lo controla la salida y en manual hay un sistema externo que lo hace.

DigitalInput:

MOSIE	UDT	FB

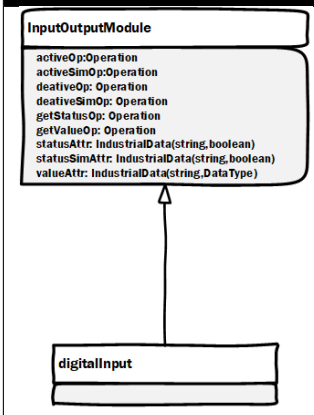
	TYPE digitalInput
	EXTENDS
	InputOutPutModule
	:
	STRUCT
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		digitalInputFB
		EXTENDS
		InputOutputModuleFB
		
		VAR_INPUT
		END_VAR
		VAR_OUTPUT
		o_value:BOOL;
		END_VAR
		VAR
		END_VAR
		//SALIDAS
		o_value:=local.value;
		//TO DO (Customize
		Code)
		
		END_FUNCTION_BLOCK

Tabla 7.11: Transformación realizada para DigitalInput.

La transformación realizada para *DigitalInput* ha consistido en la creación de una UDT *digitalInput* que extiende de la UDT *InputOutPutModule*. En este caso no ha sido necesario añadirle ningún atributo adicional, ya que como se especifica en MOSIE no es necesario. Además de la UDT se ha creado un FB *digitalInputFB* con una única variable de entrada *o_value*. Esta no representa a ninguna operación de digitalInput, ya que su propósito es indicar que la variable *value* de la UDT local de *InputOutPutModule* es de tipo BOOL.

AnalogOutput:

Para *analogOutput*, se ha procedido de forma análoga a *digitalInput* y *digitalOuput*. Se ha creado una UDT y un FB que extienden de la UDT y del FB de *InputOutPutModule*. La UDT creada no contiene ninguna variable adicional porque *analogOuput*, según su especificación en MOSIE, no contempla ningún atributo adicional. En cuando al FB creado este solo tiene una variable de entrada *i_value*, que pretende representar a la única operación que posee el elemento *setValue*. Esta variable de entrada es de tipo *REA*L para indicar que este elemento trabaja con valores analógicos y se le ha asignado a la variable *value* del tipo *InputOutPutModule* que esta definida en el FB *InputOutPutModuleFB*.

MOSIE	UDT	FB

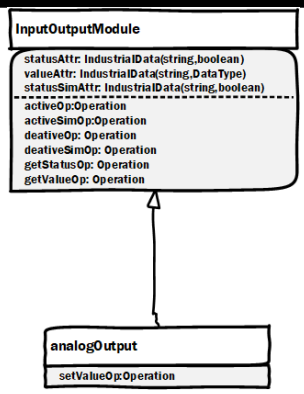
	TYPE AnalogOutput
	EXTENDS
	InputOutPutModule
	:
	STRUCT
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		analogOutputFB
		EXTENDS
		InputOutputModuleFB
		
		VAR_INPUT
		i_value:REAL;
		END_VAR
		VAR_OUTPUT
		END_VAR
		VAR
		END_VAR
		//ENTRADAS
		local.value:=i_value;
		//TO DO (Customize Code)
		END_FUNCTION_BLOCK

Tabla 7.12: Transformación realizada para AnalogOuput.

AutoManAnalogOuput:

MOSIE	UDT	FB

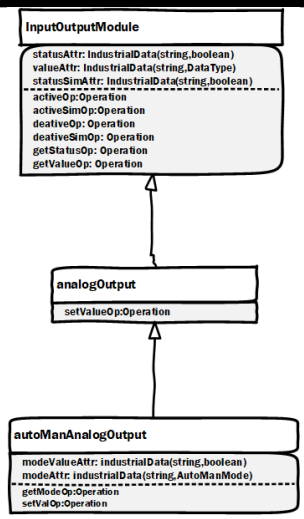
	TYPE
	AutoManAnalogOutput
	
	EXTENDS AnalogOutput :
	STRUCT
	manValue:ANY;
	mode:AutoManMode;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		AutoManAnalogOutputFB
		EXTENDS analogOutputFB
		VAR_INPUT
		i_mode:AutoManMode;
		i_manValue:REAL;
		END_VAR
		VAR_OUTPUT
		o_mode:AutoManMode;
		o_manValue:REAL;
		END_VAR
		VAR
		an_local:
		AutoManAnalogOutput;
		END_VAR
		//ENTRADAS
		an_local.mode:=i_mode;
		//TO DO (Customize Code)
		IF (an_local.mode =
		AutoManMode.auto) THEN
		an_local.value:=i_value;
		ELSIF (an_local.mode =
		AutoManMode.man) THEN
		an_local.value:=i_manValue;
		END_IF;
		//SALIDAS
		o_mode:=an_local.mode;
		END_FUNCTION_BLOCK

Tabla 7.13: Transformación realizada para AutoAnalogOuput.

Para *AutoManAnalogOuput* se han creado una nueva UDT y un nuevo FB. La transformación realizada es análoga a la realizada para *AutoManDigitalOutput*, solo que al trabajar con valores analógicos, este elemento extiendo de *AnalogOutPut* para su nueva UDT y AnalogOutPutFB para su FB. Por otro lado al tra-

tarse de un elemento que modela salidas analógicas la variable *i_manValue* y *O_manValue* son de tipo REAL. Por último la variable *mode* es del tipo de enumerado: *AutoManMode*.

AnalogInput:

AnalogInput es un elemento que presenta una gran número de atributos. Por ello la UDT y el FB incluyen un mayor número de variables. En la tabla 7.14 podemos ver la correlación entre las operaciones y las variables de este elemento y la UDT, así como el FB creados.

MOSIE	UDT	FB

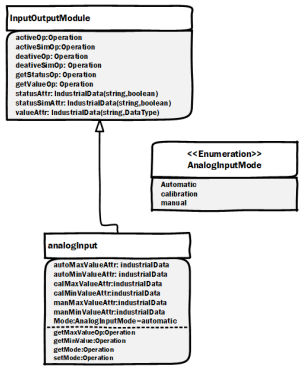
	`TYPE AnalogInput`
	`EXTENDS InputOutPutModule :`
	`STRUCT`
	`autoMaxValue:REAL;`
	`autoMinValue:REAL;`
	`calMaxValue:REAL;`
	`calMinValue:REAL;`
	`manMaxValue:REAL;`
	`manMinValue:REAL;`
	`mode:AnalogInputMode;`
	`END_STRUCT`
	`END_TYPE`
		`FUNCTION_BLOCK AnalogInputFB`
		`EXTENDS InputOutputModuleFB`
		`VAR_INPUT`
		`i_mode:AnalogInputMode;`
		`END_VAR`
		`VAR_OUTPUT`
		`o_MaxValue:REAL;`
		`o_MinValue:REAL;`
		`o_mode:AnalogInputMode;`
		`END_VAR`
		`VAR`
		`in_local:AnalogInput;`
		`END_VAR`
		`//ENTRADAS;`
		`in_local.mode:=i_mode;`
		`//TO DO (Customize Code)`
		`//SALIDAS`
		`IF(in_local.mode=AnalogInputMode.`
		`automatic) THEN`
		`o_MaxValue:=in_local.autoMaxValue`
		`;`
		`o_MinValue:=in_local.autoMinValue`
		`;`
		`ELSIF(in_local.mode=`
		`AnalogInputMode.calibration)`
		`THEN`
		`o_MaxValue:=in_local.calMaxValue;`
		`o_MinValue:=in_local.calMinValue;`
		`ELSIF(in_local.mode=`
		`AnalogInputMode.manual) THEN`
		`o_MaxValue:=in_local.manMaxValue;`
		`o_MinValue:=in_local.manMinValue;`
		`END_IF;`
		`o_mode:=in_local.mode;`
		`END_FUNCTION_BLOCK`

Tabla 7.14: Transformación realizada para AnalogInput.

Como se puede ver en el código del FB, dependiendo del valor de la variable *mode*, alojada en la variable *in_local*, el FB devolverán en las variables de salida *o_MaxValue* ó *o_MinValue*, los valores de calibración en función del modo de calibración manual, automático o calibración, en el que se encuentre el FB.

MOSIE	UDT	FB
autoMaxValue	autoMaxValue:REAL;	-
autoMinValue	autoMinValue:REAL;	-
calMaxValue	calMaxValue:REAL;	-
calMinValue	calMinValue:REAL;	-
manMaxValue	manMaxValue:REAL;	-
manMinValue	manMinValue:REAL;	-
mode	mode:AnalogInputMode;	-
getMaxValue	-	o_MaxValue:REAL;
getMinValue	-	o_MinValueREAL:
getMode	-	o_mode:AnalogInputMode
setMode	-	i_mode:AnalogInputMode

Por último vamos a ver la correlación realizada para los tipos enumerados de este paquete:

AutoManMode:

Para el enumerado AutoManMode se ha creado un nuevo tipo de enumerado que recoge los diferentes modos de trabajo

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE AutoManMode :
(
 auto := 0,
 man := 1
);
END_TYPE
```

AnalogInputMode:

Para el enumerado *AnalogInputMode* también se ha creado el siguiente nuevo tipo de enumerado.

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE AnalogInputMode :
(
 automatic := 0,
 calibration:=1,
 manual:=2
);
END_TYPE
```

EL PAQUETE ALARM EN IEC 61131-3

En este paquete se recoge el concepto de alarma, el cual puede asociarse a conceptos digitales y analógicos. A continuación vamos a ver las transformaciones realizadas para los conceptos de este paquete a la especificación IEC 61131-3.

Alarm:

MOSIE	UDT	FB

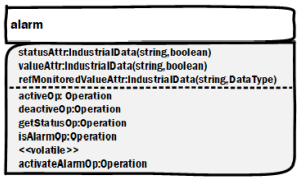
	TYPE Alarm EXTENDS
	Module :
	STRUCT
	status:BOOL;
	value:BOOL;
	refMonitoredValue:ANY
	;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK AlarmFB
		VAR_INPUT
		i_status:BOOL;
		i_value:BOOL;
		END_VAR
		VAR_OUTPUT
		o_status:BOOL;
		o_value:BOOL;
		END_VAR
		VAR
		al_local:Alarm;
		END_VAR
		//Entradas
		al_local.status:=
		i_status;
		al_local.value:=o_value
		;
		//TO DO (Customize Code)
		//Salidas
		o_status:=al_local.
		status;
		o_value:=al_local.value
		;
		END_FUNCTION_BLOCK

Tabla 7.15: Transformación realizada para Alarm.

Alarm es un elemento que tiene tres atributos. Estos se han recogido en la UDT *Alarm*, la cual extiende de Module tal y como se especifica en MOSIE. Para las operaciones de este concepto se ha creado un FB *AlarmFB* con dos variables de entrada: *i_status* e *i_value*. La variable *i_status* representa las operaciones *activate* y *deactivate* e *i_value* representa la operación *resetAlarm*. Además también se han definido 2 variables de entrada *o_status* y *o_value*, para representar las operaciones *getStatus* e *isAlarm* respectivamente, que nos indica si estamos en alarma (True) o no (False).

DigitalAlarm:

DigitalAlarm es un módulo pensado para la creación y abstracción de alarmas que se activan mediante un valor lógico la trasformación realizada se puede ver en la tabla 7.16.

MOSIE	UDT	FB

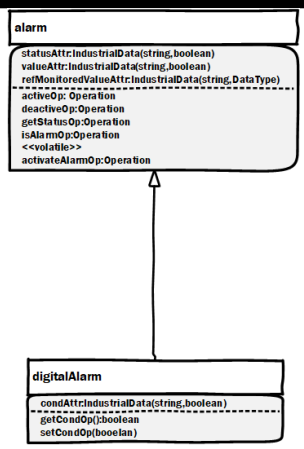
	TYPE DigitalAlarm
	EXTENDS Alarm :
	STRUCT
	cond:
	digitalOperator;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		DigitalAlarmFB
		EXTENDS AlarmFB
		VAR_INPUT
		i_cond:
		DigitalOperator;
		END_VAR
		VAR_OUTPUT
		o_cond:
		DigitalOperator;
		END_VAR
		VAR
		di_local:
		DigitalAlarm;
		END_VAR
		//Entradas
		di_local.cond:=i_cond;
		//TO DO (Customize Code)
		//salidas
		o_cond:=di_local.cond;
		END_FUNCTION_BLOCK

Tabla 7.16: Transformación realizada para DigitalAlarm.

Como se puede ver en la correlación realizada para este elemento, se ha creado una UDT que contiene el atributo *cond*. Este atributo es del tipo enumerado *digitalOperator*, el cual se ha representado en IEC 61131-3 definiendo un nuevo tipo de dato enumerado:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE DigitalOperator :
(
 equal := 0,
 noEqual:=1
);
END_TYPE
```

Para este elemento también se ha creado un FB, que recoge los métodos definidos para *DigitalAlarm*, con la variable de entrada *i_cond* representamos la operación *setCond* y con la variable de salida *o_cond* la operación *getCond*. Ambas variables son del tipo *DigitalOperator*.

Adicionalmente para este FB se ha incluido el siguiente código:

```
//TO DO (Customize Code)
IF(di_local.cond=DigitalOperator.equal) THEN
IF (di_local.value=TRUE) THEN
 di_local.value:=TRUE;
ELSE
 di_local.value:=FALSE;
END_IF
ELSIF (di_local.cond=DigitalOperator.noEqual) THEN
IF (di_local.value=FALSE) THEN
 di_local.value:=TRUE;
ELSE
 di_local.value:=FALSE;
END_IF
END_IF
//salidas
```

Con estas instrucciones activamos el valor de la alarma, almacenado en *di_local.value* en función del valor que tenga la variable de tipo enumerado *di_local.cond*.

AnalogAlarm:

MOSIE	UDT	FB

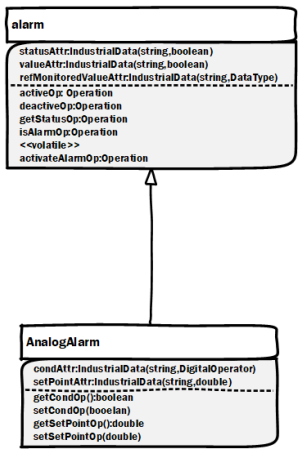
	`TYPE AnalogAlarm`
	`EXTENDS Alarm :`
	`STRUCT`
	`cond`
	`:AnalogOperator;`
	`setPoint:ANY;`
	`END_STRUCT`
	`END_TYPE`
		`FUNCTION_BLOCK AnalogAlarmFB`
		`EXTENDS AlarmFB`
		`VAR_INPUT`
		`i_cond`
		`:analogOperator;`
		`i_setPoint:REAL;`
		`i_refMonitoredValue:REAL;`
		`END_VAR`
		`VAR_OUTPUT`
		`o_cond:analogOperator;`
		`o_setPoint:REAL;`
		`END_VAR`
		`VAR`
		`an_local:AnalogAlarm;`
		`END_VAR`
		`//Entradas`
		`an_local.cond`
		`:=i_cond;`
		`an_local.setPoint`
		`:=i_setPoint;`
		`an_local.refMonitoredValue`
		`:=i_refMonitoredValue;`
		`//TO DO (Customize Code)`
		`//salidas`
		`o_cond:=an_local.cond;`
		`o_setPoint:=an_local.`
		`setPoint;`
		`END_FUNCTION_BLOCK`

Tabla 7.17: Transformación realizada para AnalogAlarm.

Este módulo está pensado para modelar alarmas que se disparan cuando se ha sobrepasado un valor, por encima o por debajo de un umbral. Como se puede ver en la transformación realizada, este módulo trabaja con valores analógicos haciendo que el atributo *value* sea de tipo REAL. Además posee un atributo con el que se puede comparar el valor de *value*, *setPoint*, dependiendo de la operación que se encuentre almacenada en *cond* que es de tipo *AnalogOperator*.De tal forma

que según el resultado el modulo estará en alarma o no.

Como se puede observar la correlación realizada para este elemento es muy parecida a la realizada para *DigitalAlarm*. Se ha creado una UDT que contiene el atributo *setPoint* de tipo *ANY*, que recogerá el umbral para disparar la alarma según un valor analógico y el atributo *cond* que es del tipo enumerado *AnalogOperator*. Este tipo de dato se ha creado también utilizando tipos enumerados, como se puede ver a continuación:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE AnalogOperator :
(
 equal := 0,
 noEqual:=1,
 lessThan:=2,
 greaterThan:=3
);
END_TYPE
```

Para este modulo también se ha creado un FB, que recoge los distintos métodos definidos para *AnalogAlarm*, las variables de entrada *i_cond* de tipo *AnalogOperator* que representa la operación *setCond* y *i_setPoint* para la operación *setSetPoint*. También cuenta con la variable de salida *o_cond* para el método *getCond* y *o_setPoint* para el método *getSetPoint*.

Para este FB se propone el siguiente código:

```
IF(an_local.cond=analogOperator.equal) THEN
 IF (an_local.setPoint=an_local.refMonitoredValue) THEN
  an_local.value:=TRUE;
 ELSE
  an_local.value:=FALSE;
 END_IF
 ELSIF (an_local.cond=analogOperator.greaterThan) THEN
 IF ( an_local.setPoint>an_local.refMonitoredValue) THEN
  an_local.value:=TRUE;
 ELSE
  an_local.value:=FALSE;
 END_IF
ELSIF (an_local.cond=analogOperator.lessThan) THEN
 IF ( an_local.setPoint<an_local.refMonitoredValue) THEN
  an_local.value:=TRUE;
 ELSE
  an_local.value:=FALSE;
 END_IF
 ELSIF (an_local.cond=analogOperator.noEqual) THEN
 IF ( an_local.setPoint<>an_local.refMonitoredValue) THEN
  an_local.value:=TRUE;
 ELSE
  an_local.value:=FALSE;
```

```
 END_IF
END_IF;
```

Mediante este código se cambia el valor de la variable *an_local.value* en función del tipo de condición recogido en *an_local.cond*. Este valor nos condiciona el tipo de comparación que tenemos que realizar entre *an_local.setPoint* y *an_local.refMonitoredValue*, previo casting a *REAL* de las dos variables.

EL PAQUETE CONTROL EN IEC 61131-3

La estrategia utilizada para transformar los elementos de este paquete a la especificación IEC 61131-3 es la misma que se ha utilizado en el resto de los paquetes de MOSIE que extienden del concepto de Module. Se ha creado un conjunto de UDTs y FBs para recoger los atributos y los métodos de los conceptos de este paquete además las UDTs extienden de la UDT creada para el concepto *Module*.

ActivationControl:

Este módulo nos sirve para recoger el número de activaciones de una válvula o la cantidad de horas de funcionamiento de un motor.

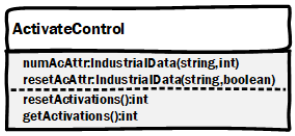
MOSIE	UDT	FB
	`TYPE ActivationControl EXTENDS Module : STRUCT i_numAct:INT; i_resetAct:BOOL:= FALSE; END_STRUCT END_TYPE`	`FUNCTION_BLOCK ActivationControlFB VAR_INPUT i_numAct:INT; i_resetAct:BOOL; END_VAR VAR_OUTPUT o_numAct:INT; END_VAR VAR ac_local:ActivationControl; END_VAR //Entradas ac_local.numAct:=i_numAct; ac_local.resetAct:= i_resetAct; IF(ac_local.resetAct=TRUE) THEN ac_local.numAct:=0; END_IF //TO DO (Customize Code) //Salidas o_numAct:=ac_local.numAct; END_FUNCTION_BLOCK`

Tabla 7.18: Transformación realizada para ActivationControl.

La transformación realizada para *ActivationControl* consta de una UDT que recoge los atributos *numAct* y *resteAct*. Esta última variable tiene predefinido el valor FALSE para no resetear el contador de activaciones a menos de que se indique implícitamente. Además se ha creado un FB para recoger las operaciones

setActivation y las variable de entrada *i_numAct* y *resetActivation* y *i_resteAct*. Como se puede ver en el código si la variable *i_resetAct* es igual a True el número de activaciones se pone a 0, reseteando por tanto el contador de activaciones u horas de funcionamiento. La única variable de salida *o_numAct* representa la operación *getActivations*.

Power:

Power es un módulo con el que se pretende recoger los atributos que caracterizan actuadores complejos, con dos sentidos de giro o rampas de aceleración. La transformación realizada se puede ver en la tabla 7.19.

MOSIE	UDT	FB

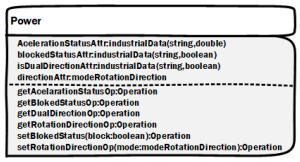
	TYPE Power EXTENDS Module :
	STRUCT
	acelerationStatus:REAL;
	blockedStatus:BOOL;
	direction:
	modeRotationDirection;
	isDualDirection:BOOL;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK PoweFB
		VAR_INPUT
		i_blockedStatus:BOOL;
		i_direction:
		modeRotationDirection;
		END_VAR
		VAR_OUTPUT
		o_acelerationStatus:REAL;
		o_blockedStatus:BOOL;
		o_direction:
		modeRotationDirection;
		o_isDualDirection:BOOL;
		END_VAR
		VAR
		pw_local:Power;
		END_VAR
		//ENTRADAS
		pw_local.blockedStatus:=
		i_blockedStatus;
		pw_local.direction:=
		i_direction;
		//TO DO (Customize Code)
		//SALIDAS
		o_acelerationStatus:=
		pw_local.acelerationStatus;
		o_blockedStatus:=
		pw_local.blockedStatus;
		o_direction:=
		pw_local.direction;
		o_isDualDirection:=
		pw_local.isDualDirection;
		END_FUNCTION_BLOCK

Tabla 7.19: Transformación realizada para Power.

La UDT creada para este elemento recoge los atributos: *acelerationStatus, blockedStatus, isDualDirection* y *direction*. Para este último se ha creado un nuevo tipo enumerado *modeRotationDirection*.

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE modeRotationDirection :
(
 idle:= 0,
 direct:=1,
 inverse:=2
);
END_TYPE
```

Este nuevo tipo de enumerado recoge los posibles estados en los que se puede

encontrar un actuador complejo. Para recoger los métodos de este concepto se ha creado también un FB con las variables de entrada *i_blockedStatus* y *i_direction*, representando las operaciones *setBlockedStatus* y *setRotationDirection* y con las variables de salida *o_acelerationStatus*, *o_blockedStatus*, *o_direction*, y *o_isDualDirection*, representando las operaciones *getAcelerationStatus*, *getBlockedStatus*, *getRotationDirection* y *getDualDirection* respectivamente.

ControlParameters:

Para ControlParameters también se ha creado una UDT y un FB, que se pueden ver en la tabla 7.20. Este módulo posee un gran número de atributos y de métodos. Los atributos como siempre han sido recogidos en la UDT *ControlParameters* y los métodos en el FB *ControlParametersFB* mediante variables de entrada y salida. En la tabla 7.21 podemos ver toda las correlaciones realizadas.

MOSIE	UDT	FB

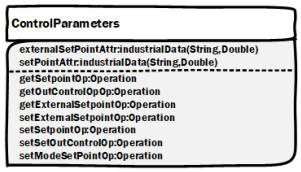
	TYPE ControlParameters
	EXTENDS Module :
	STRUCT
	externalSetPoint:REAL
	;
	mode:modeSetPoint;
	setPoint:REAL;
	OutControl:ANY;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		ControlParametersFB
		VAR_INPUT
		i_externalSetPoint:REAL;
		i_mode:modeSetPoint;
		i_setPoint:REAL;
		i_OutControl:REAL;
		END_VAR
		VAR_OUTPUT
		o_externalSetPoint:REAL;
		o_mode:modeSetPoint;
		o_setPoint:REAL;
		o_OutControl:REAL;
		END_VAR
		VAR
		co_local:ControlParameters;
		END_VAR
		//Entradas
		co_local.externalSetPoint
		:= i_externalSetPoint;
		co_local.mode:=i_mode;
		co_local.setPoint:=
		i_setPoint;
		co_local.OutControl:=
		i_OutControl;
		//TO DO (Customize Code)
		//salidas
		o_externalSetPoint:=
		co_local.externalSetPoint;
		o_mode:=co_local.mode;
		o_setPoint:=
		co_local.setPoint;
		o_OutControl:=
		co_local.OutControl;
		END_FUNCTION_BLOCK

Tabla 7.20: Transformación realizada para ControlParameters.

Para el atributo *mode* se ha creado un nuevo tipo de enumerado, con el que podemos saber si el controlador esta utilizando el setpoint interno o externo.

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE modeSetPoint :
(
 in_ternal := 0,
 external:=1
);
```

```
END_TYPE
```

MOSIE	UDT	FB
externalSetPoint	externalSetPoint:REAL;	-
outControl	outControl:ANY;	-
setPoint	setPoint:REAL;	-
mode	mode:modeStePoint;	-
getExternalSetpoint	-	o_externalSetPoint:REAL;
getMode	-	o_Mode:modeSetPoint;
getOutControl	-	o_outControl:REAL;
getSetPoint	-	o_setPoint:REAL;
setExternalSetpoint	-	i_externalSetPoint:REAL;
setMode	-	i_Mode:modeSetPoint;
setOutControl	-	i_outControl:REAL;
setSetPoint	-	i_setPoint:REAL;

Tabla 7.21: Correspondecia de los métodos y atributos para Controlparameters.

PIDControlParameters:

PIDControlParameters pretende modelar los parámetros de un controlador PID clásico. Este módulo extiende del concepto *ControlParameters* y la transformación realizada es la siguiente:

MOSIE	UDT	FB

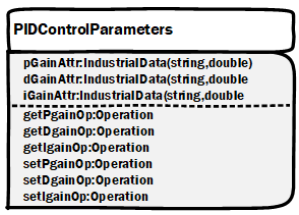
	TYPE
	PIDControlParameters
	EXTENDS
	ControlParameters
	:
	STRUCT
	DGain:REAL;
	PGain:REAL;
	IGain:REAL;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK
		PIDControlParametersFB
		EXTENDS
		ControlParametersFB
		VAR_INPUT
		i_DGain:REAL;
		i_PGain:REAL;
		i_IGain:REAL;
		END_VAR
		VAR_OUTPUT
		o_DGain:REAL;
		o_PGain:REAL;
		o_IGain:REAL;
		END_VAR
		VAR
		pid_local
		:PIDControlParameters;
		END_VAR
		//Entradas
		pid_local.DGain:=i_DGain;
		pid_local.PGain:=i_PGain;
		pid_local.IGain:=i_IGain;
		//TO DO (Customize Code)
		//SALIDAS
		pid_local.DGain:=i_DGain;
		pid_local.PGain:=i_PGain;
		pid_local.IGain:=i_IGain;
		END_FUNCTION_BLOCK

Tabla 7.22: Transformación realizada para PIDControlParameters.

Para *PIDControlParameter*s se ha creado una UDT que extiende de la UDT creada para *ControlParameters* que recoge los atributos de este módulo *DGain, PGain* y *IGain*. En el FB asociado a este módulo *PIDControlParametersFB*, se recogen los métodos del mismo, las variables de entrada *i_DGain, i_PGain* y *i_IGain* correponden a los métodos *getDGain, getPGain* y *getIGain* respectivamente y las variables de salida *o_DGain, o_PGain* y *o_IGain* correponden a los métodos *setDGain, setPGain* y *setIGain*.

EL PAQUETE DEVICES EN IEC 61131-3

Los dos principales conceptos de este paquete son *Sensor y Actuator*. Ambos conceptos pueden ser de tipo digital o analógico, pueden tener alarmas. Las transformaciones realizadas son las siguiente:

Actuator:

MOSIE	UDT	FB

	`TYPE Actuator`
	`EXTENDS Component :`
	`STRUCT`
	`alarm:`
	`ARRAY [0..1] OF Alarm;`
	`END_STRUCT`
	`END_TYPE`
		`Sin FB`

Tabla 7.23: Transformación realizada para Actuator.

Para *Actuator* se ha creado solo una UDT que extiende de *Component*, con un variable de tipo array de alarmas para recoger las alarmas asociadas a este elemento.

DigitalActuator:

La UDT creada para este componente posee tres variables que corresponden a un array de tipo *DigitalAlarm* para recoger las posibles alarmas del elemento, una variable de tipo *DigitalInput* para la entrada del actuador con la que recoger, por ejemplo, la confirmación de marcha o de paro, una variable output de tipo AutoManDigitalOutput con la que enviar la orden de marcha o parada al *actuador*. El FB creado para este concepto DigitalActuatorFB tiene cuatro variables de entrada *i_resetActivation, i_mode, i_status* y *i_value* que representan a las operaciones *resetActivations, setAutoManMode, setStatus* y *setValue* respectivamente.

Las variables de salida *o_numAct, o_mode, o_status y o_value* recogen las operaciones *getActivations, getAutoManMode, getStatus* y *getValue*. Para este concepto tanto la variaable *i_value* como *o_value* son de tipo *BOOL*, ya que estamos trabajando con actuadores digitales. Por último se han creado dos alarmas de tipo digital para inicializar el array de alarmas.

MOSIE	UDT	FB

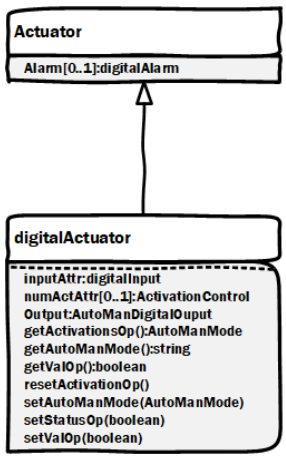
	`TYPE DigitalActuator EXTENDS Actuator :`
	`STRUCT`
	`input:DigitalInput;`
	`numAct:ARRAY [0..1] OF ActivationControl;`
	`output: AutoManDigitalOutput;`
	`END_STRUCT`
	`END_TYPE`
		`FUNCTION_BLOCK DigitalActuatorFB`
		`VAR_INPUT`
		`i_resetActivation:BOOL;`
		`i_mode:AutoManMode;`
		`i_status:BOOL;`
		`i_value:BOOL;`
		`END_VAR`
		`VAR_OUTPUT`
		`o_numAct:REAL;`
		`o_mode:AutoManMode;`
		`o_status:BOOL;`
		`o_value:BOOL;`
		`END_VAR`
		`VAR`
		`diac_local:DigitalActuator;`
		`alr0:DigitalAlarm;`
		`alr1:DigitalAlarm;`
		`END_VAR`
		`//entradas`
		`diac_local.output.value :=i_value;`
		`diac_local.numAct[0].resetAct :=i_resetActivation;`
		`diac_local.numAct[1].resetAct :=i_resetActivation;`
		`diac_local.output.mode :=i_mode;`
		`diac_local.output.status :=i_status;`
		`//incializacion de alarmas`
		`diac_local.alarm[0]:=alr0;`
		`diac_local.alarm[1]:=alr1;`
		`//TO DO (Customize Code)`
		`//salidas`
		`o_value:= diac_local.output.value;`
		`o_numAct:= diac_local.numAct;`
		`o_mode:= diac_local.output.mode;`
		`o_status:= diac_local.output.status;`
		`END_FUNCTION_BLOCK`

Tabla 7.24: Transformación realizada para DigitalActuator.

AnalogActuator:

Este componente conceptualmente es idéntico a *DigitalActuator*. La única diferencia reside en que *AnalogActuator* trabaja con señales analógicas, por lo que utiliza salidas y entradas de tipo *analogInput* y *AutoManAnalogOutput*, y alarmas de tipo analógico como se pude ver en la UDT y en el FB creado. Por último se han creado también dos alarmas de tipo analógico para incializar el array de las alarmas.

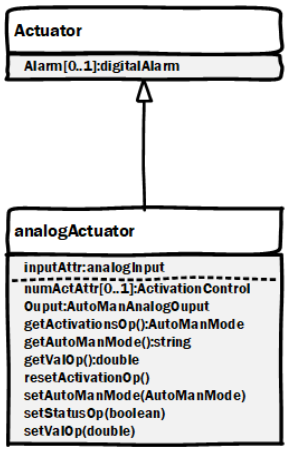
MOSIE	UDT	FB
	`TYPE AnalogActuator EXTENDS Actuator : STRUCT input:AnalogInput; numAct:ARRAY [0..1] OF ActivationControl; output:AutoManAnalogOutput; END_STRUCT END_TYPE`	`FUNCTION_BLOCK AnalogActuatorFB VAR_INPUT i_resetActivation:BOOL; i_mode:AutoManMode; i_status:BOOL; i_value:REAL; END_VAR VAR_OUTPUT o_numAct:REAL; o_mode:AutoManMode; o_status:BOOL; o_value:REAL; END_VAR VAR anac_local:AnalogActuator; alr0:AnalogAlarm; alr1:AnalogAlarm; END_VAR //entradas anac_local.output.value :=i_value; anac_local.numAct[0].resetAct :=i_resetActivation; anac_local.numAct[1].resetAct :=i_resetActivation; anac_local.output.mode :=i_mode; anac_local.output.status :=i_status; //Inicializamos el tipo de alarmas anac_local.alarm[0]:=alr0; anac_local.alarm[1]:=alr1; //TO DO (Customize Code) //salidas o_value :=anac_local.output.value; o_numAct :=anac_local.numAct; o_mode :=anac_local.output.mode; o_status :=anac_local.output.status; END_FUNCTION_BLOCK`

Tabla 7.25: Transformación realizada para AnalogActuator.

VFDActuator:

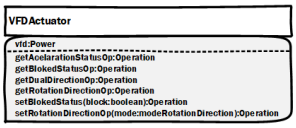
MOSIE	UDT	FB
	`TYPE VFDActuator EXTENDS AnalogActuator : STRUCT vfd:Power; END_STRUCT END_TYPE`	`FUNCTION_BLOCK VFDActuatorFB EXTENDS AnalogActuator VAR_INPUT i_blockedStatus:BOOL; i_rotationDirection :modeRotationDirection; END_VAR VAR_OUTPUT o_aceleration:REAL; o_blockedStatus:BOOL; o_rotationDireciont :modeRotationDirection; END_VAR VAR vf_local:VfdActuator; END_VAR //Entradas vf_local.vfd.blockedStatus :=i_blockedStatus; vf_local.vfd.direction :=i_rotationDirection; //TO DO (Customize Code) //Salidas o_aceleration :=vf_local.vfd.aceleration Status; o_blockedStatus :=vf_local.vfd.blockedStatus; o_rotationDireciont :=vf_local.vfd.direction; END_FUNCTION_BLOCK`

Tabla 7.26: Transformación realizada para VFDActuator.

Este elemento esta pensado para modelar actuadores analógicos mas complejos y es una extensión de *AnalogActuator*, como se pude ver tanto en la UDT como en el FB creado. La UDT creada contiene solo una variable de tipo *Power*, tal y como se especifica en MOSIE. Las variables de entrada *i_blockedStatus* e

i_rotationDirection representa los métodos *setAcelartion* y *setRotationDirection* y las variables de salida *o_blockedStatus*, *o_rotationDirection* y *o_celeration* a los métodos *getAcelartion*, *getRotationDirection* y *getDualDirection*.

Sensor.

MOSIE	UDT	FB
	TYPE Sensor EXTENDS Component : STRUCT alarm: ARRAY [0..1] OF Alarm; END_STRUCT END_TYPE	Sin FB

Tabla 7.27: Transformación realizada para Sensor.

La correlación realizada para el concepto *sensor* es practicamente la misma que para *Actuator*, una UDT con un array de alarmas.

DigitalSensor:

Para *DigitalSensor* se ha realizado una transformación basada en la creación de una UDT que extiende del concepto *Sensor*, en la que se ha definido una variable de tipo *digitalInputFB*. Esta variable se ha definido de tipo *digitalInputFB*, en primer lugar porque no existe ningún método asociado a este elemento, por lo que no es necesario crear ningún FB para este elemento de acuerdo a las correlaciones que estamos realizando entre iMMAS/MOSIE a IEC 61131-3 y en segundo lugar porque el FB *digitalInputFB* recoge todas las caracterizaras necesarias para trabajar con una variable de entrada digital.

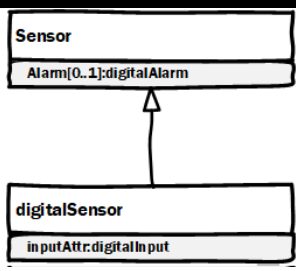
MOSIE	UDT	FB
	TYPE DigitalSensor EXTENDS Sensor : STRUCT input: digitalInputFB; END_STRUCT END_TYPE	Sin FB

Tabla 7.28: Transformación realizada para DigitalSensor.

AnalogSensor:

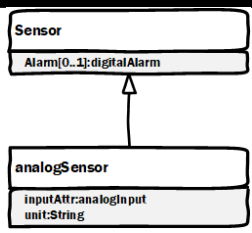
MOSIE	UDT	FB
	TYPE analogSensor EXTENDS Sensor : STRUCT input:analogInputFB; END_STRUCT END_TYPE	Sin FB

Tabla 7.29: Transformación realizada para AnalogSensor.

La transformación realizada para *AnalogSensor* es prácticamente idéntica a la realizada en el caso de *DigitalSensor*, solo que este elemento trabaja con valores analógicos y por lo tanto la variable *input* es de tipo *analogInputFB*, ya que dicho FB esta pensado para trabajar con entradas de tipo analógico.

EL PAQUETE CONTROLLERS EN IEC 61131-3

El paquete *Controllers* está compuesto por dos componentes *Regulator* y *PIDRegulator*.

Regulator:

MOSIE	UDT	FB

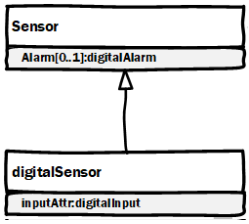
	TYPE Regulator
	EXTENDS
	ActiveComponent
	:
	STRUCT
	setPoint:
	ControlParameters;
	actuator:actuator;
	sensor:sensor;
	END_STRUCT
	END_TYPE
		FUNCTION_BLOCK RegulatorFB
		VAR_INPUT
		i_externalSetpoint:REAL;
		i_OutPutControl:REAL;
		i_setPoint:REAL;
		END_VAR
		VAR_OUTPUT
		o_externalSetpoint:REAL;
		o_OutPutControl:REAL;
		o_setPoint:REAL;
		o_modeSetPoint:modeSetPoint;
		END_VAR
		VAR
		re_local:Regulator;
		END_VAR
		//ENTRADAS
		re_local.setPoint.setPoint
		:=i_setPoint;
		re_local.setPoint.externalSetPoint
		:= i_externalSetpoint;
		re_local.setPoint.OutControl
		:=i_OutPutControl;
		//TO DO (Customize Code)
		//Salidas
		o_externalSetpoint
		:=re_local.setPoint.externalSetPoint;
		o_OutPutControl
		:=re_local.setPoint.OutControl;
		o_setPoint
		:=re_local.setPoint.setPoint;
		o_modeSetPoint
		:=re_local.setPoint.mode;
		END_FUNCTION_BLOCK

Tabla 7.30: Transformación realizada para Regulator.

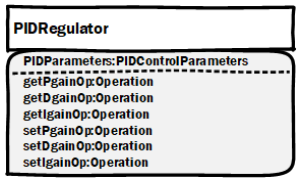
Para *Regulator* se han creado una UDT y un FB. La UDT creada extiende de

ActiveComponent y tiene 3 variables: *setPoint* que es de tipo *ControlParameters* y con la que recogemos todas las características de un controlador básico, Sensor para monitorizar la variable del proceso a controlar (sensor), y Actuator para actuar sobre el proceso y alcanzar la consigna del regulador (actuador).. En cuanto al FB creado este posee tres variables de entrada *i_externalSetpoint, i_OutPutControl* y *i_setPoint* para las operaciones *setExternalSetPoint, setOutPutControl* y *setSetPoint*, y cuatro variables de salida *o_externalSetpoint*, *o_OutPutControl*, *o_setPoint* y *o_modeSetPoint*, representan las operaciones *getExternalSetPoint, getOutPutControl, getSetPoint* y *getModeSetPoint* respectivamente.

PIDRegulator:

Este componente recoge todas las características de un controlador PID y ha sido transformado creando una UDT que extiende de *Regulator* y en la que se ha definido una variable de tipo *PIDControlParameters*. El FB creado también extiende del FB creado para Regulator y posee 3 variables de entrada *i_DGain, i_PGain* y *i_IGain* que corresponden a las operaciones *setDGain, setPGain* y *setIGain*, 3 variables de salida *o_DGain, o_PGain* y *o_IGain* que corresponden a las operaciones *getDGain, getPGain* y *getIGain*.

MOSIE	UDT	FB

	`TYPE PIDRegulator`
	`EXTENDS Regulator :`
	`STRUCT`
	`PIDParameters:`
	`PIDControlParameters;`
	`END_STRUCT`
	`END_TYPE`
		`FUNCTION_BLOCK PIDRegulatorFB`
		`EXTENDS Regulator`
		`VAR_INPUT`
		`i_DGain:REAL;`
		`i_PGain:REAL;`
		`i_IGain:REAL;`
		`END_VAR`
		`VAR_OUTPUT`
		`o_DGain:REAL;`
		`o_PGain:REAL;`
		`o_IGain:REAL;`
		`END_VAR`
		`VAR`
		`pidre_local:PIDRegulator;`
		`END_VAR`
		`//ENTRADAS`
		`pidre_local.PIDParameters.DGain`
		`:=i_DGain;`
		`pidre_local.PIDParameters.PGain`
		`:=i_PGain;`
		`pidre_local.PIDParameters.IGain`
		`:=i_IGain;`
		`//TO DO (Customize Code)`
		`//SALIDAS`
		`pidre_local.PIDParameters.DGain`
		`:=i_DGain;`
		`pidre_local.PIDParameters.PGain`
		`:=i_PGain;`
		`pidre_local.PIDParameters.IGain`
		`:=i_IGain;`
		`END_FUNCTION_BLOCK`

7.4.2. EJEMPLO DE MODELADO CON MOSIE PARA IEC 61131-3

Para probar las reglas de transformación hemos tomado el modelo MOSIE que se realizó para el sistema industrial utilizado en el capítulo 6 formado por un

motor eléctrico, un detector y una barrera de stop. En este caso, a partir del modelo PIM modelado con MOSIE se lleva a cabo la transformación a un modelo PSM para IEC 61131-3. Dicho modelo PSM esta constituido por una implementación de UDT y FB que se puede desplegar en un entorno de programación para PLCs basado en la especificación IEC 61131-3.

Para transformar tanto el modelo como las máquinas de estados creadas, hemos aplicado las reglas de transformación especificadas en la sección anterior entre MOSIE y IEC 61131-3, utilizando el enfoque jerárquico desde los elementos sintácticos como los componentes hasta los elementos sintácticos más sencillos como las variables.

Detector:

Este componente es quizás el elemento mas sencillo de todos los que forma nuestro sistema industrial, extiende de *DigitalSensor* y no posee ninguna operación, por lo que con crear una UDT que extienda de la UDT *DigitalSensor* es suficiente para realizar la transformación, de este el:

```
TYPE Detector EXTENDS DigitalSensor :
STRUCT
END_STRUCT
END_TYPE
```

Motor:

Para *Motor* se ha creado un FB *MotorFB* que extiende del FB creado para *AnalogActuator: AnalogActuatorFB*. No se ha creado ninguna UDT para Motor, ya que el *AnalogActuatorFB* ya posee una variable local de tipo UDT *AnalogActuator: anac_local*. Se podría haber hecho la transformación creando una UDT Motor, que extendise de la UDT *AnalogActuator*, para posteriormente crear una variable dentro del FB *MotorFB* de tipo UDT Motor. Quizas esta opción conceptualmente sería mas correcta, pero es un poco redundante ya que el FB *AnalogActuatorFB* ya posee una variable de tipo UDT *AnalogActuator* que podemos utilizar en el caso de la transformación realizada para la barrera si ha realizado la transformación creando una UDT adicional.

```
FUNCTION_BLOCK MotorFB EXTENDS AnalogActuatorFB
VAR_INPUT
```

```
 i_autoManMode:AutoManMode;
 i_status:BOOL;
 i_value:REAL;
 i_resetActivation:BOOL;
END_VAR
VAR_OUTPUT
 o_autoManMode:AutoManMode;
 o_status:BOOL;
 o_value:REAL;
 o_activation:REAL;
END_VAR
VAR
END_VAR
//ENTRADAS
 anac_local.numAct[0].resetAct:=i_resetActivation;
 anac_local.numAct[1].resetAct:=i_resetActivation;
//estatus para la las salidas y las entradas
 anac_local.output.status:=i_status;
 anac_local.input.status:=i_status;
//valor para el actuador
 anac_local.output.value:=i_value;

//Custom Code (To DO)

//SALIDAS
 o_autoManMode:=anac_local.output.mode;
 o_status:=anac_local.output.status;
//valor de la entrada
 o_value:=anac_local.input.value;
 o_activation:=anac_local.numAct;
END_FUNCTION_BLOCK
```

El FB creado tiene 4 variables de entrada *i_autoManMode, i_status, i_valu*e y *i_resetActivation* que representan las operaciones *setAutoManMode, setStatus, setValue* y *resetActivation*. Las variables de salida *o_autoManMode, o_status, o_value* y *o_Activation* son para las operaciones *getAutoManMode, getStatus, getValue* y *getActivation* respectivamente.

Barrera:

Para *Barrera* se ha creado un FB que extiende de *DigitalActuatorFB*.

```
FUNCTION_BLOCK BarreraFB EXTENDS DigitalActuatorFB
VAR_INPUT
 i_autoManMode:AutoManMode;
 i_status:BOOL;
 i_value:BOOL;
 i_resetActivation:BOOL;
END_VAR

VAR_OUTPUT
 o_autoManMode:AutoManMode;
 o_status:BOOL;
 o_value:BOOL;
 o_activation:REAL;
END_VAR
```

```
VAR
 local_barrera:Barrera;
END_VAR
//ENTRADAS
 diac_local.numAct[0].resetAct:=i_resetActivation;
 diac_local.numAct[1].resetAct:=i_resetActivation;
//estatus para la las salidas y las entradas
 diac_local.output.status:=i_status;
 diac_local.input.status:=i_status;
//valor para el actuador
 diac_local.output.value:=i_value;
//Custom Code (To DO)

//SALIDAS
 o_autoManMode:=diac_local.output.mode;
 o_status:=diac_local.output.status;
//valor de la entrada
 o_value:=diac_local.input.value;
 o_activation:=diac_local.numAct;
END_FUNCTION_BLOCK
```

El FB *BarreraFB* tiene el mismo número de variables de entrada y de salida que el FB *MotorFB*. La única diferencia existente en *BarreraFB* con respecto de *MotorFB* radica en que *Barrera* extiende de *DigitalActuatorFB*, por lo que las variables *i_value* y *o_value* son de tipo *BOOL*.

CintaRodillos:

Cinta rodillos es el elemento principal del modelo ya que desde este se controla todos los elementos que forman el sistema industrial. Con los métodos *start* y *stop* de este elemento se debería parar y arrancar el sistema industrial por completo. Para este elemento se han creado una UDT CintaRodillos y un FB CintaRodillosFB. La UDT CintaRodillos tiene 3 variables: *barrera* de tipo *BarreraFB* (FB), motor de tipo *MotorFB* (FB) y detector de tipo *Detector* (UDT). Esta UDT extiende de *Component* tal y como se especifica en el modelo.

```
TYPE CintaRodillos EXTENDS ActiveComponent :
STRUCT
 barrera:BarreraFB;
 Motor:MotorFB;
 detector:Detector;
END_STRUCT
END_TYPE
```

El FB *CintaRodillos* no extiende de ningún FB, ya que para *ActiveComponent* no se creó ningún FB en las transformaciones realizadas. El código es el siguiente:

```
FUNCTION_BLOCK CintaRodillosFB
```

```
VAR_INPUT
 i_estado:BOOL:=FALSE;
END_VAR
VAR_OUTPUT
 o_EstadoBarrera:BOOL;
 o_EstadoMotor:BOOL;
 o_EstadoDetector:BOOL;
END_VAR
VAR
 local_cintarodillos:CintaRodillos;
END_VAR

//inicialiamos todos los elementos en automatico
 local_cintarodillos.barrera.i_autoManMode:=autoManMode.auto;
 local_cintarodillos.motor.i_autoManMode:=autoManMode.auto;

//Custom code (TO DO)
 IF(NOT (i_estado)) THEN

  local_cintarodillos.motor.i_value:=0;
  local_cintarodillos.barrera.i_value:=FALSE;

 ELSIF
 //arrancamos motor y activamos barrera
  local_cintarodillos.motor.i_value:=80;
  local_cintarodillos.barrera.i_value:=TRUE;
 //si detectamos una pieza bajamos la barrera
  IF(detector.input.o_value) THEN
   local_cintarodillos.barrera.i_value:=FALSE;
  ELSIF (NOT (detector.input.o_value)) THEN //si no detectamos una pieza
      subimos la barrera
   local_cintarodillos.barrera.i_value:=TRUE;
  END_IF
 END_IF

//Salidas
 o_EstadoBarrera:=local_cintarodillos.barrera.o_status;
 o_EstadoMotor:=local_cintarodillos.motor.o_status;
 o_EstadoDetector:=local_cintarodillos.detector.input.o_status;
END_FUNCTION_BLOCK
```

Este FB tiene una sola variable de entrada *i_estado* que nos sirve para representar las operaciones *Start* y *Stop*. Las variables de salida *o_EstadoBarrera, o_EstadoMotor* y *o_EstadoDetector* representan las operaciones *getEstadoBarrera, getEstadoMotor* y *getEstadoDetector*. Además el FB tiene una variable interna *local_cintarodillos* que es de tipo UDT *Cintarodillos*. En esta UDT hemos agrupado los elementos que forman el sistema industrial. La funcionalidad del sistema industrial se ha implementado mediante el código siguiente:

```
IF(NOT (i_estado)) THEN

 local_cintarodillos.motor.i_value:=0;
 local_cintarodillos.barrera.i_value:=FALSE;
```

```
 ELSIF
  //arrancamos motor y activamos barrera
  local_cintarodillos.motor.i_value:=80;
  local_cintarodillos.barrera.i_value:=TRUE;
  //si detectamos una pieza bajamos la barrera
  IF(detector.input.o_value) THEN
   local_cintarodillos.barrera.i_value:=FALSE;
  ELSIF (NOT (detector.input.o_value)) THEN //si no detectamos una pieza
      subimos la barrera
   local_cintarodillos.barrera.i_value:=TRUE;
  END_IF
END_IF
```

En el código anterior se puede ver que si estamos con el sistema arrancado, siempre que el detector detecte una pieza u objeto, se baja la barrera para que este pase para posteriormente volver a subir la barrera cuando este haya pasado.

7.5. CONCLUSIONES

Actualmente la programación de dispositivos industriales está normaliza según la especificación tercera del estándar IEC 61131, y aunque existen trabajos académicos en los que se postulan la utilización de paradigmas como la programación orientada a objetos [136] a la hora de desarrollar software para estos dispositivos, lo cierto es que la utilización de dichos paradigmas en aplicaciones reales es todavía muy escasa [1], debido fundamentalmente a que las plataformas de desarrollo para PLCs soportan principalmente la programación estructurada o funcional.

La utilización de iMMAS y MOSIE creando modelos PIM de los sistemas industriales nos puede ayudar a conceptualizar estos sistema de modo independiente de la plataforma. La aplicación de modelos en sistemas software industriales presenta un gran desafío al poder transformar los modelos PIM creados a código que pueda ser alojado en un dispositivo industrial, principalmente un PLC. Por ello, las transformaciones presentadas en este capítulo son importantes, ya que nos permiten llevar un modelo MOSIE a código para un dispositivo industrial, siguiendo la especificación IEC 61131-3.

De este modo conseguimos que todos los dispositivos industriales que contemplen la especificación IEC 61131-3 y en los que se despliegue iMMAS/MOSIE, se organicen de la misma forma para el mismo modelo, lo cual nos permite desplegar el mismo modelo en varios tipos de dispositivos industriales, aunque sean de distintos fabricantes, consiguiendo un alto grado de interoperabili-

dad gracias a las transformaciones realizadas de iMMAS/MOSIE a IEC 61131-3, y obviando detalles específicos de alguna plataforma de programación de PLCs concreta. Además de la interoperabilidad, también mejoramos la escalabilidad de estos sistemas, ya que una ampliación pasaría por ampliar/rehacer el modelo y después volver a desplegarlo en el dispositivo industrial, utilizando las reglas de transformaciones de iMMAS y MOSIE. Adicionalmente, también, facilitamos aspectos como el mantenimiento y la sostenibilidad de los programas de estos dispositivos, ya que siempre dispondremos de un modelo que podrá ser consultado en caso de duda por cualquier programador o ingeniero, a modo de documentación del sistema.

8

METODOLOGÍA DESARROLLO DE SISTEMAS SOFTWARE INDUSTRIALES UTILIZANDO iMMAS

"The real problem is not whether machines think but whether men do."
B. F. Skinner. 1904-1990

8.1. INTRODUCCIÓN

El desarrollo de un sistema software es una tarea compleja que requiere la realización sistemática de un conjunto de actividades para obtener un sistema software ejecutable funcional. La secuenciación del conjunto de actividades y la aplicación sistemática de las mismas determinarán el proceso que se ha seguido para el desarrollo del software [137]. En este sentido las metodologías de desarrollo de software proporcionan una guía sistemática de cómo tienen que efectuarse dichas actividades para que el producto software final resultante tenga la calidad esperada y satisfaga todas las necesidades del cliente de forma fiable y puntual.

En el ámbito de los sistemas industriales el software desempeña un papel cada vez mas destacado [1], equiparando su importancia y criticidad al de los sistemas mecánicos, eléctricos, neumáticos o mecánicos. Por este motivo, no sólo es importante describir el software en base a modelos como se ha presentado en capítulos anteriores, sino que necesitamos disponer de una metodología que nos especifique el orden en el que deben desarrollarse los sistemas software.

En este capítulo se presenta una novedosa metodología para el desarrollo y despliegue de sistemas compuestos por dispositivos industriales y servidores OPC UA que hemos denominado MIAS (Methodology for the development of Industrial Automation Systems). Tradicionalmente en el mundo industrial el desarrollo de software se circunscribe al desarrollo de programas o, incluso, la configuración de alguna aplicación específica que se ejecuta en una ubicación concreta (p. ej., PLC) con objeto de que pueda integrarse en el contexto de la arquitectura jerarquizada presente en el entorno industrial. La metodología MIAS plantea el desarrollo sistemático de software para sistemas industriales siguiendo un enfoque dirigido por modelos mediante la conceptualización del sistema industrial, en lugar de enfocar el desarrollo en un elemento concreto del sistema industrial como puede ser un dispositivo PLC, una RTU, el sistema de recetas, o el sistema de intercambio de datos. La ventaja potencial de este enfoque es que nos permite trabajar con modelos independientes de la plataforma (en nuestro caso PIM utilizando el lenguaje de modelado iMMAS) que posteriormente pueden ser transformados para desarrollar aplicaciones dependientes de la plataformas en ubicaciones concretas (en nuestro caso en PLC y servidores OPC UA). La metodología simplifica el proceso de desarrollo de sistemas industriales y favorece la descripción de dichos sistemas industriales utilizando un mismo vocabulario, lo que además facilita el mantenimiento y la evolución del sistema [138].

8.2. METODOLOGÍA MIAS (METHODOLOGY FOR INDUSTRIAL AUTOMATION SYSTEM)

La metodología MIAS proporciona una guía sistemática dirigida por modelos del proceso que se debe seguir para el desarrollo de sistemas software en entornos industriales utilizando el lenguaje de modelado iMMAS. Dicha metodología ha sido desarrollada, tomando como base el estándar de validación para sistemas software industriales GAMP (Good Automated Manufacturing Practice) [139], de la asociación internacional de Ingenieros Farmacéticos ISP (International Society for Pharmaceutical Engineering)[140], siendo su mayor ámbito de aplicación el de la industria farmacéutica. La GAMP pueden considerarse como un estándar que ofrece enfoque estructurado para la validación de sistemas software industriales. Este estándar actualmente se encuentra en su versión 5, por lo que nor-

malmente se hace referencia como GAMP 5, donde se realiza mas énfasis en el control y análisis del riesgo y en la gestión de la calidad.

Para el desarrollo de un sistema industrial utilizando la metodología MIAS se tienen que realizar las siguientes etapas:

- Análisis del sistema industrial.
  - Requerimientos de usuario
  - Requerimientos funcionales
  - Requerimientos no funcionales y restricciones
  - Matriz de trazabilidad
- Diseño del sistema.
  - Identificar elementos y sus relaciones
  - Agrupación de los elementos en subsistemas
  - Modelo PIM
- Implementación y despliegue del sistema.
  - Modelo PSM
    - OPC UA
    - IEC 61131-3
  - Despliegue
- Pruebas.
  - Pruebas de testeo y validación.
    - Pruebas de señales
    - Pruebas de secuencias
    - Certificación del sistema.

A continuación se describe con detalle cada una de las etapas del proceso de desarrollo utilizando un ejemplo para la descripción de cada una de ellas.

8.3. ANÁLISIS DEL SISTEMA INDUSTRIAL

La etapa de análisis del sistema industrial se centra en determinar fundamentalmente cuál es el propósito del sistema qué se quiere hacer y qué se quiere conseguir. Para ello, se deben determinar los requerimientos que ha de satisfacer el sistema desde el punto de vista de los usuarios finales y los requerimientos funcionales del sistema que describen como va a operar el sistema para cubrir dichos requerimientos de usuario.

- Requerimientos de usuario.
- Requerimientos funcionales.
- Matriz de trazabilidad.

8.3.1. DEFINICIÓN DE LOS REQUERIMIENTOS DE USUARIO

El primer paso para empezar cualquier desarrollo o proyecto software es saber qué se quiere hacer y qué se desea conseguir con el proyecto o con el sistema software. Este "saber lo que se quiere" recibe el nombre de requisitos del sistema o requerimientos de usuario, y consiste en establecer lo que el sistema software debe de hacer o, lo que es lo mismo, lo que el usuario espera que este haga [141].

Para definir estos requerimientos se deben de realizar reuniones con los usuarios finales del sistema, con el fin de plasmar todas sus necesidades formalmente en un documento. Este documento debe de recoger todos los aspectos, requerimientos y objetivos de forma coherente, consistentes y con el mayor grado de exactitud y fidelidad posible. En el siguiente ejemplo se muestra el contenido de dicho documento basándonos en GAMP 5:

REQUERIMIENTOS DE USUARIO.

1. **Descripción del sistema.**

   1..1 Antecedentes históricos.

   Motivación y descripción de la necesidad del Sistema desde el punto de vista del cliente.

   1..2 Problema o deficiencia que se pretende solventar.

   1..3 Beneficios que se pretenden conseguir.

2. **Requerimientos.**

   2..1 Funciones que el Sistema debe de realizar.

ID Requerimiento	Descripción del requerimiento

Tabla 8.1: Tabla resumén de requerimientos de usuario.

   2..2 Datos.

   Descripción de los datos que se desean almacenar o gestionar.

   2..3 Interfaces.

   Descripción de los interfaces de usuario minimos que debe de tener el sistema.

   2..4 Entorno.

   Entorno en el que se prentende desplegar el Sistema, arquitectura necesaria (redes, servidores y hardware necesario)

3. **Restricciones.**

   Restricciones del sistema o necesidades del mismo para poder ser implementado o desplegado.

8.3.2. DEFINICIÓN DE LOS REQUERIMIENTOS FUNCIONALES DEL SISTEMA.

Una vez que se han recogido todos los requerimientos del usuario se debe de especificar cómo el sistema software va a operar o funcionar para cubrir esos requerimientos de usuario. Esta especificación recibe el nombre de requerimientos funcionales. Los requerimientos funcionales de un sistema describen lo que el sistema ha de hacer, enunciando que funcionalidad o conjunto de funcionalidades debe de proporcionar el sistema; cómo reacciona esté ante ciertas entradas y cómo debe de comportarse en ciertas situaciones [142]. Existen casos en los que los requerimientos funcionales recogen de forma explícita lo que el sistema no debe de hacer.

Los requerimientos funcionales, a diferencia de los requerimientos de usuario, deben de redactarse con un gran nivel de detalle, recogiendo por ejemplo las entradas y salidas del sistema, las excepciones, generación de alarmas y el funcionamiento interno del sistema software. Estos requerimientos se puede expresar de muy diversas formas, aunque quizás la mas extendida consiste en generar un documento tomando como base los requerimientos de usuario y en el que se definen las características específicas que el sistema ha de tener.

Las especificaciones de los requerimientos funcionales deben de ser completas y concisas, no deben de dar lugar a interpretaciones ambiguas. La ambigüedad es un de los mayores enemigos a la hora de afrontar un proyecto de desarrollo software, porque dejamos la puerta abierta a las interpretaciones de los ingenieros y los desarrolladores, lo cual nos puede llevar a soluciones cuyos resultados no sean los esperados por los clientes. El ser precisos nos asegura que los requerimientos no sean contradictorios y que todas las funcionalidades solicitadas por el cliente están recogidas.

El análisis de los requerimientos funcionales para sistemas complejos y de gran tamaño presenta cierto grado de dificultad. En la mayoría de los casos resulta prácticamente imposible recoger todos los requerimientos de forma consistente y completa por varias razones. En primer lugar porque suele ser difícil poner de acuerdo a todos los actores/usuarios del sistema, ya que cada uno tiene necesidades diferentes y con frecuencia contradictorias. En segundo lugar porque es fácil cometer errores y omisiones, no recogiendo ciertos aspectos o funcionalidades del sistema que en fases posteriores surgirán como necesarios y que formaban

parte de los requerimientos de usuario.

Para asegurarnos que se han elaborado unos requerimientos funcionales completos, claros y concisos, no deberíamos de dar como válida la primera versión del documento en el que se recogen los requerimientos de funcionales. Deberíamos realizar análisis más profundos y sobre todo realizar un proceso de interacción continua entre todo el equipo de desarrollo, clientes y expertos, para poder alcanzar un documento definitivo que recoja todos los requerimientos funcionales del sistema.

Para recoger los requerimientos funcionales de un sistema software industrial podemos utilizar el siguiente documento:

REQUERIMIENTOS FUNCIONALES.

1 **Descripción Funcional.**

1.1 Propósito.

1.2 Acrónimos y Definiciones.

Terminología	Definición

1.3 Referencias.

Nº Documento	Título del Documento

1.4 Alcance.

2 **Descripción General.**

2.1 Características de la aplicación.

2.1.1 Interfaces de usuario.

Especificación y descripción de los interfaces de usuario que el sistema requiere.

2.1.2 Interfaces Hardware.

Elementos e interfaces Hardware necesarios para que el sistema funcione, tarjetas de comunicaciones y E/S.

2.1.3 Interfaces Software.

Sistemas adyacentes con los que se debe de comunicar el sistema software, como por ejemplos sistemas de base de datos.

2.2 Restricciones.

Restricciones que se deban de tener en cuenta sobre el sistema software, dependencias del sistema operativo, necesidades de hardware, etc.

3 **Requerimientos.**

3.1 Requerimientos Funcionales.

ID RQU	ID RQF	Descripción del requerimiento	Criticidad	Version

Tabla 8.2: Tabla resumen de requerimientos funcionales.

3.2 Requerimientos de rendimiento.

Especificación de los requerimientos de rendimiento y productividad del sistema, restricciones de tiempo si las hubiese, cantidad de datos a procesar, etc.

3.3 Requerimientos de base de datos.

Necesidades relacionadas con las base de datos necesarias para que el sistema funcione.

3.4 Características Software del Sistema.

3.4.1 Fiabilidad.

3.4.2 Disponibilidad.

3.4.3 Seguridad.

3.4.4 Portabilidad

8.3.3. MATRIZ DE TRAZABILIDAD.

Hemos recogido en los apartados anteriores los requerimiento de usuario y los requerimientos funcionales. Para poder tener una trazabilidad de ambos tipos de requerimientos y poder facilitar su revisión en cualquier momento es necesario establecer una relación entre ambos tipos de requerimientos. Esta relación se recoge en lo que se denomina Matriz de Trazabilidad, que no es más que una tabla en la que se emparejan, requerimientos de usuario con los requerimientos funcionales que satisfacen dichos requerimientos de usuario. Adicionalmente también se recogen los test de validación que se le van a aplicar al sistema software en la etapa de *Pruebas* para verificar que este cumple con los requerimientos funcionales. En la siguiente tabla tenemos un ejemplo de matriz de trazabilidad:

ID RQU	**ID RQF**	**ID Protocolo Testeo**

Tabla 8.3: Tabla ejemplo de matriz de trazabilidad.

8.4. DISEÑO DEL SISTEMA.

Una vez que se han recogidos los requerimientos funcionales del sistema y su correspondencia con los requerimientos de usuario podemos proceder a realizar el diseño del sistema software. La parte central del diseño se dedicará a obtener el modelo PIM del sistema industrial para lo cual se deben de recoger todos los aspectos del sistema industrial, elementos que lo forman, su funcionalidad y las características del mismo.

Como ya se mencionó en el capítulo 5 el propio modelo puede hacer las veces de documentación del sistema, además de recoger todos los elementos de este y sus relaciones. En nuestro caso, el modelo PIM modelado en iMMAS, no sólo la organización de los elementos que conforman el sistema o modelo estático, sino también nos permite modelar el comportamiento de los elementos recogidos por el modelo, y por tanto también recoger los aspectos funcionales del sistema. Por ello, cabe esperar que el modelo recoja las especificaciones de diseño del sistema o al menos los aspectos más importantes y relevantes de este.

Por otro lado existen también aspectos del diseño como la arquitectura del sistema, las necesidades no funcionales o las decisiones de diseño derivadas de

la tecnología de implementación o despliegue, que no estarán recogidos por el modelo en sí mismo, pero que podrían formar parte de las especificaciones de diseño.

En nuestro caso nos vamos a centrar en cómo crear el modelo PIM del sistema/proceso industrial a partir de los conceptos que ofrece MOSIE o en su defecto directamente sobre iMMAS.Señalar que el modelo obtenido es independiente de la plataforma y describe de una forma compacta tanto el modelo estático y dinámico del sistema.

8.4.1. IDENTIFICAR ELEMENTOS DEL SISTEMA Y SUS RELACIONES.

Dado que un sistema industrial dependiendo de su complejidad puede estar formado por un gran número de elementos de muy diversa índole, la mejor estrategia bajo nuestro punto de vista para identificar los diferentes elementos del sistema y sus relaciones es basarse en el conocimiento del experto o expertos en el sistema industrial que se pretende modelar, e identificar así a partir de ese conocimiento los elementos del sistema y las relaciones entre ellos.

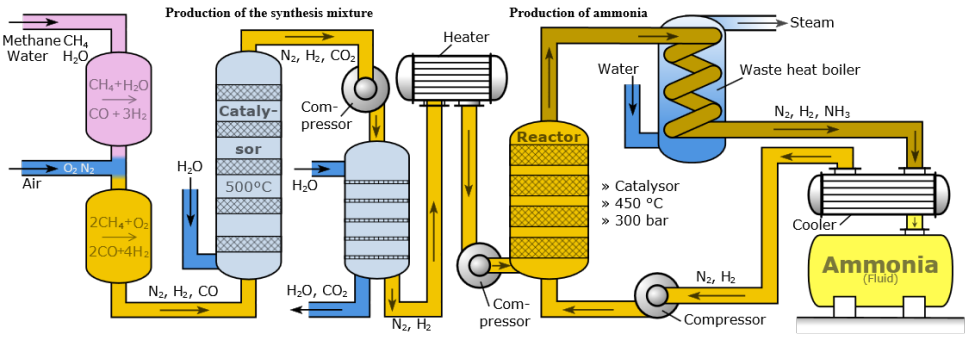


Figura 8.1: Diagrama de flujo para el proceso de producción de amoniaco.

A la hora de desarrollar un sistema industrial normalmente se trabaja en grupos formados por ingenieros industriales, eléctricos y expertos en el proceso. Los diseños y especificaciones de este grupo de expertos se suelen plasmar en diagramas de flujo, figura 8.1, así como en planos eléctricos y mecánicos, figuras 8.3 y 8.2 . Por lo tanto podremos partir de estos documentos como base para identificar los componentes del sistema industrial y sus relaciones.

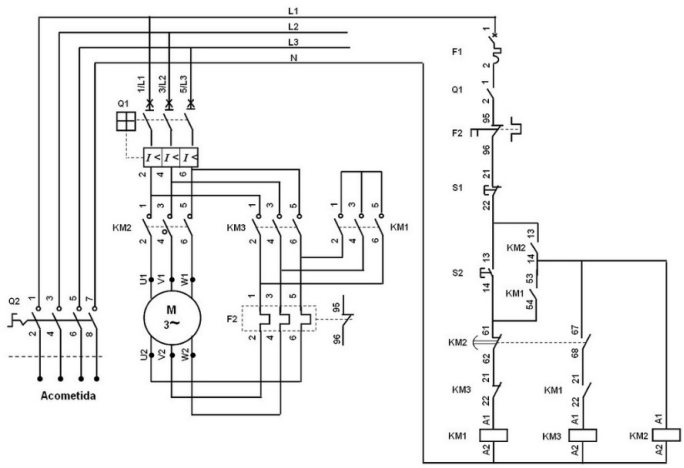


Figura 8.2: Ejemplo de esquema eléctrico.

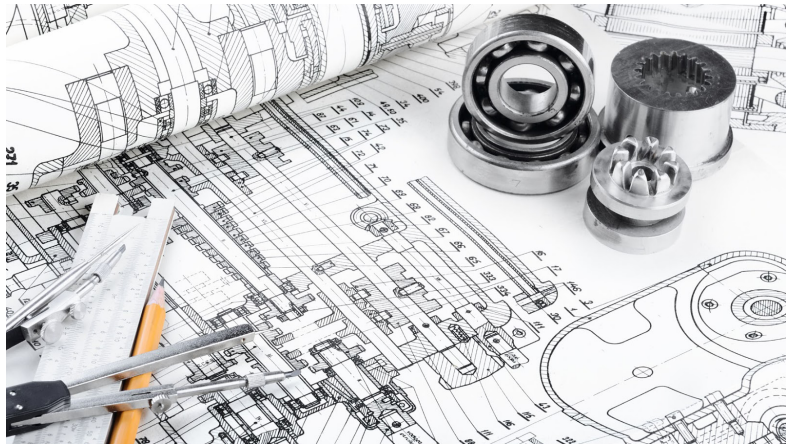


Figura 8.3: Ejemplo de plano mecánico.

Para recoger todos los elementos y sus relaciones proponemos crear tablas en la que se recojan todas las características de dichos elementos. En la tabla 8.4 podemos ver un posible formato para recoger los elementos de forma el sistema industrial y en la tabla 8.5 tenemos otro formato para recoger las relaciones entre los elementos:

Nombre Elemento	Tipo	Propiedades

Tabla 8.4: Tabla para identificar los elementos de un sistema industrial.

Nombre Relación	Elemento Origen	Elemento Destino	Cadinalidad

Tabla 8.5: Tabla para identificar los elementos de un sistema industrial.

Una vez terminado el proceso de identificación de elementos y sus relaciones, puede que hayamos identificado elementos que no se relacionan con ningún otro elemento. Esto puede deberse a que estos elementos sean parte del proceso industrial, pero no parte del sistema software industrial. En este caso dichos elementos deberían quedar excluidos en los siguientes pasos de la metodología MIAS propuesta, o puede que si pertenezcan al sistema software pero que ciertamente no se relacionen con ningún otro elemento por lo que quedaran recogidos en el modelo como un elemento aislado sin relaciones con los demás.

8.4.2. AGRUPACIÓN DE LOS ELEMENTOS EN SUBSISTEMAS.

En el paso anterior se han identificado todos los elementos y un conjunto de relaciones posibles entre ellos, pero ni la identificación de elementos, ni las relaciones entre estos se pueden dar por cerradas, ya que podemos haber pasado por alto algún elemento o algún tipo de relación. Las relaciones entre los elementos de un sistema pueden no ser obvias y difíciles de identificar, por lo que un segundo análisis tanto de la organización de los elementos como de las relaciones entre ellos puede ayudarnos a rehacer las relaciones existentes o a identificar nuevos elementos y sus relaciones.

En este sentido, y sobre todo a la hora de trabajar con sistemas industriales formados por una gran cantidad de elementos, puede ser interesante realizar agrupaciones de elementos y reevaluar las relaciones establecidas.

Las agrupaciones de elementos se puede hacer de muchas formas y siguiendo multitud de criterios, como por ejemplo el tipo de elemento actuador/sensor,

digital/analógico. Para poder conseguir ciertos niveles de abstracción quizás la mejor forma de agrupar los elementos es conforme a la funcionalidad de los mismos, pudiendo crear así subsistemas dentro del sistema industrial. Esta forma de proceder se basa en la descomposición de un problema complejo en problemas más pequeños y por lo tanto más simples, los cuales pueden ser atacados más fácilmente de acuerdo a la clásica estrategia de divide y vencerás. Hay que destacar que para realizar esta agrupación de elementos y poder así descomponer el sistema industrial en subsistemas es imprescindible contar con la ayuda del experto o grupo de expertos del sistema industrial en cuestión.

Una vez establecidos estas agrupaciones, se hace necesario reevaluar las tablas donde hemos identificado los elementos y las relaciones entre ellos. Siguiendo nuestro propuesta, se deben de definir una tabla de elementos y otra de relaciones por cada agrupación o subsistema detectado, identificando las relaciones de estos nuevos elementos en función del papel que desempeñan ahora como parte de subsistemas menos complejos. Además se debe de crear una o varias tablas que relacionen e identifiquen estos subsistemas como el sistema general.

Esta forma de agrupar elementos en subsistemas nos llevará como veremos mas adelante a generar modelos jerárquicos. Por lo que agrupar los elementos de un sistema industrial en varios subsistemas, los cuales a su vez forman un sistema industrial completo, podemos considerarlo como el primer paso hacia la creación de modelos jerárquicos para sistemas software industriales. Esta estrategia, como ya hemos comentado en capítulos anteriores, es quizás la que mejor se adapta a la hora de modelar un sistema industrial.

8.4.3. OBTENCIÓN DEL MODELO PIM

Siguiendo los pasos descritos en los apartados anteriores, dispondremos de un conjunto de tablas en las que tendremos recogidos todos los elementos de nuestro sistema industrial y su relaciones. Además también tendremos identificados cada uno de los subsistemas, en los que hemos dividido nuestro sistema industrial y los elementos que los componen. Esta clasificación previa nos va a facilitar todo el proceso de modelado, puesto que, a la hora de crear nuestro modelo no partimos de cero ya que previamente se ha realizado un análisis bastante exhaustivo. Este enfoque de organizar los elementos de un sistema industrial, en

grupos según su nivel de abstracción esta también recogido en la norma ISA 88 [15], en la que se proponen un modelo jerárquico conceptual, figura 8.4. Este tipo de modelos recogen perfectamente los conceptos de sistemas, sub-sistemas y elementos.

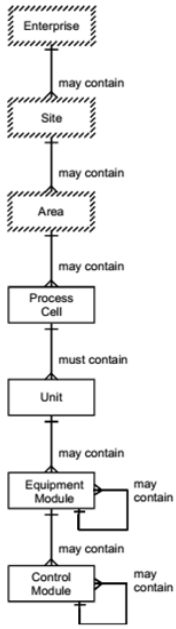


Figura 8.4: Conceptos recogidos por la Norma ISA88.

Llegados a este punto, solo nos resta construir el modelo PIM que recoja tanto los elementos identificados como sus relaciones. Dicho modelo debe de organizar los elementos entorno a los subsistemas o subconjuntos de elementos que se han identificado y organizarlos de forma jerárquica, utilizando directamente el lenguaje de modelado iMMAS o las abstracciones que proporciona el perfil MOSIE.

8.5. IMPLEMENTACIÓN Y DESPLIEGUE.

Después de modelar el sistema industrial con iMMAS, el siguiente paso consistiría en desplegar este modelo en el sistema software/hardware encargado de supervisar, monitorizar y controlar nuestro proceso industrial. Por tanto, una vez dispongamos del modelo PIM independiente de la plataforma, podremos llevar a

cabo la implementación y despliegue del sistema industrial mediante el proceso de transformación del modelo PIM a un modelo PSM, es decir, un modelo dependiente de la plataforma. Es decir, según la plataforma, el modelo puede ejecutarse y desplegarse directamente en la plataforma concreta, o bien puede suponer una nueva transformación en código que, después de un proceso de compilado, nos permite obtener un ejecutable que posteriormente se despliega en un punto concreto del sistema industrial.

En nuestro caso, aplicando el proceso de transformación, podemos obtener un modelo PSM para servidores OPC o un modelo PSM para dispositivos industriales PLCs.

La transformación de un modelo PIM a PSM se aplica en tres fases:

- Identificación de los elementos sintácticos de iMMAS contenidos en un modelo PIM.

- Transformación directa de cada uno de los elementos sintácticos de iMMAS al lenguaje de modelado destino.

- Modelado final de cada uno de los elementos sintácticos del modelo PIM identificado en el modelo PSM.

Para la obtención de la transformación se aplicarían las reglas de transformación especificadas en el capítulo 6 y 7. En el capítulo 9 se muestra cómo se aplica en un ejemplo.

8.6. PRUEBAS DE TESTEO Y VALIDACIÓN

Una vez que se ha desplegado el modelo PSM, se debe comprobar que cumple con los requerimientos de usuario iniciales y los requerimientos funcionales del sistema de acuerdo a la matriz de trazabilidad. La mejor forma de comprobar estos requerimientos es mediante la creación de uno o varios protocolos de testeo y validación, que nos aseguren que el sistema dispone de las funcionalidades especificadas. Estos protocolos de testeo pueden ser clasificados en pruebas de señales y pruebas de secuencias.

8.6.1. PRUEBAS DE SEÑALES

La primera comprobación que se debe de realizar consiste en asegurar que las señales eléctricas tanto de entrada como de salida están cableadas correctamente al dispositivo industrial (PLC) y que la activación/lectura de estas señales activan/corresponde con el elemento físico adecuado. Así, por ejemplo, cuando el PLC activa una salida digital para activar un motor se debe comprobar que se pone en marcha el motor esperado.

Las pruebas de señales son la primera prueba de validación que se debe de hacer en un sistema industrial, ya que si la lectura de señales o el envió de ordenes (escritura de valores) a los elementos del sistema industrial no es correcta, el sistema se puede comportar de cualquier forma. Imaginemos que queremos abrir una válvula y en lugar de esto arrancamos un motor, podríamos incluso generar daños en los elementos.

Este primer conjunto de pruebas también se utiliza a veces para validar los diseños eléctricos y las instalaciones eléctricas de los sistemas industriales. Para poder registrar estas pruebas podemos utilizar la siguiente tabla 8.6:

Nombre Elemento	Descripción	Señal Eléctrica	Direccion E/S	Tipo	Verificado por	Fecha

Tabla 8.6: Formato de tabla para recoger las pruebas de entradas y salidas.

8.6.2. PRUEBAS DE SECUENCIAS

Una vez se ha comprobado el correcto funcionamiento de todos los elementos individualmente podemos pasar a comprobar el funcionamiento del sistema en conjunto. El comportamiento de un sistema industrial suele estar especificado por un conjunto de secuencias, donde cada secuencia no es mas que un conjunto de acciones que se ejecutan de forma consecutiva. Las secuencias a su vez se pueden seguir agrupando en fases y operaciones según la norma ISA88 [15] dependiendo de la complejidad del propio sistema.

Para comprobar el correcto funcionamiento de las distintas fases del sistema, se comprueba el correcto funcionamiento del sistema en su conjunto. Además, si el sistema no está organizado en fases y secuencias, se tendrá también que comprobar la funcionalidad del sistema en su conjunto. En cualquier caso, una de las

forma de comprobar el correcto funcionamiento de un sistema industrial se basa en definir un conjunto de pruebas con el objetivo de asegurar que el sistema hace lo que tiene que hacer, detectando así los posibles fallos con el fin de que sean corregidos.

Este conjunto de pruebas recibe el nombre de Protocolo de Testeo o Protocolos de Pruebas, y en él se definen las acciones necesarias que nos aseguren el correcto funcionamiento de todo el sistema, comprobando, por ejemplo, que todas las secuencias del sistema son correctas y por consiguiente el sistema es correcto. Para recoger este conjunto de pruebas podemos utilizar el siguiente formato:

Nombre del Protocolo: _____

Tipo de Protocolo: _____

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:

Pre-requisitos:

Objectivos del Test:

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.		
2.		
3.		

Nº	Descripción	Resultados Esperados	Resultados Reales	O=Ok F=Fallo	Firma Fecha
1.					
1.1					
1.2					
2.					
2.1					
2.2					
3.					
3.1					
3.2					

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

8.6.3. CERTIFICACIÓN DEL SISTEMA

Una vez que hemos verificado las entradas y salidas y se han ejecutado los protocolos de testeo con resultados satisfactorios podemos proceder a realizar la certificación del sistema. En esta certificación se indica que el sistema software cumple con los requerimientos de usuario de forma satisfactoria, y ha pasado un conjunto de pruebas que lo certifican mediante evidencias o mediante la firma de aprobación del técnico que ha ejecutado los protocolos de pruebas. El documento que recoge la certificación del sistema es el último formalismo necesario para poner el sistema en producción de forma oficial. En este documento se recoge la validez del sistema software y el éxito del despliegue del mismo en un entorno industrial. Pueden existir certificaciones parciales pendientes de pruebas no críticas de algunos aspectos del sistema como el cambio del color de algunas luces (azul por amarillo), que no se han podido realizar o de algún tipo de documentación necesaria como los planos de ubicación o planos eléctricos. Un posible documento para certificar un sistema puede ser el siguiente:

Certificación del Sistema.

1 **Descripción del Sistema y Objectivos de la certificación.**

1.1 Descripción del sistema.

1.2 Impacto del sistema.

1.3 Alcance de la certificación.

2 **Estrategia de pruebas utilizadas.**

Ejemplo:

Las pruebas se han ejecutado en entorno de producción debido a que, en el momento de realizar las pruebas, la planta estaba en paro, por lo que su ejecución en entorno de producción no suponía una interferencia en los procesos habituales, ni existía riesgo de afectar a la calidad del producto. Los documentos que se listan a continuación son los que finalmente se han creado para la ejecución de las pruebas:

Titulo de los documentos, nº de versión

3 **Criterios de Aceptación.**

Ejemplo:

Los criterios de aceptación de cada una de las pruebas realizadas sobre el sistema se encuentran detallados en cada uno de los documentos de test correspondientes, especificados en el punto 2 del presente informe.

4 **Resolución de incidencias del proyecto.** A continuación se listan las desviaciones halladas durante la ejecución de las pruebas:

Protocolo	**Nº Desviación**	**Test**	**Descripción**	**Acción Correctora**	**Tipo de desviación**	**Estado**

Quedan los siguientes ítems pendientes en lo relativo a pruebas:

Item pendiente	Responsable	Estado

5 **Conclusiones de la certificación.**

Ejemplo: En el momento de la emisión del presente informe, no existen desviaciones pendientes de los test ejecutados, ya que las desviaciones existentes se dan por cerradas con el plan de acción definido, aunque si existe un listado de ítems pendientes. Estos ítems pendientes, detallados en el punto 4.0, son de los siguientes tipos:

- Pruebas de elementos no críticos que deben efectuarse en producción y por tanto requiere de la liberación previa del sistema
- Items de carácter documental, que si bien son necesarios no suponen un alto riesgo para el arranque de la producción.
- Registro de formación: Se ha planificado una formación en cada una de las áreas. Se ha generado un formato por escrito en el que cada responsable de proyecto ha incluido en que consiste el cambio y en qué afecta a la producción entre otros detalles. El personal de producción leerá y firmará este listado una vez leído y entendido. La formación está después planificada. Se verificará una vez ejecutada en el arranque.

8.7. CONCLUSIONES

La utilización de una metodología en el proceso de desarrollo del software nos permite obtener un producto software final que cumple con las necesidades de los usuarios y los requerimientos del sistema. La metodología MIAS se ha desarrollado alrededor del lenguaje de modelado iMMAS con objeto de proporcionar

de forma clara y concisa el conjunto de actividades que tiene que realizar el ingeniero para el diseño y desarrollo del software para sistemas industriales basados en iMMAS.

La metodología MIAS se basa en tres pilares importantes en el desarrollo del software de un sistema industrial. El primer pilar implica una comprensión detallada del sistema software a construir. El segundo pilar se basa en la creación de un modelo PIM que posteriormente se va a desplegar por transformación en modelos PSM dependientes de la plataformas. El tercer y último pilar es la ejecución de una serie de protocolos de testeo y pruebas que nos garanticen el correcto funcionamiento del sistema software, y que se materializa en una certificación del sistema.

En el próximo capítulo se mostrará cómo se aplica la metodología MIAS en un caso concreto.

9

CASO DE ESTUDIO: SISTEMA INDUSTRIAL DE EMBALAJE

"The best way to predict the future is to implement it."
David Heinemeier Hansson

9.1. INTRODUCCIÓN

En este capítulo se va a aplicar la metodología MIAS presentada en el capítulo anterior para el desarrollo del software de un sistema industrial concreto. Hemos escogido como caso de estudio una cadena de montaje y embalaje de piezas que pertenece al ámbito de los sistemas industriales discretos. En nuestro sistema se ensamblan piezas mediante un robot y se colocan mediante otro robot en una caja para, a continuación, ser almacenadas.

Para poder simular el sistema se ha utilizado el software Factory IO [87]. Este software nos permite crear y simular entornos industriales en 3D y conectarlo con dispositivos industriales reales como un PLC, de forma que se puede simular un proceso productivo complejo utilizando escenarios 3D animados. Además es posible actuar sobre la simulación en función de las ordenes que estos reciben de los dispositivos industriales a los que se conectan mediante la utilización de una red Ethernet. Podemos encontrar más información de este software en su página web [87].

En la figura 9.1 y en la figura 9.2 podemos ver el sistema 3D completo que se ha creado formado por dos sistemas, los cuales denominaremos sistemas de ensamblado y sistema de almacenaje. Ambos sistemas modelados en 3D con Factory IO tienen que estar completamente coordinados de forma que las piezas que genera

el primer sistema sean embaladas por el segundo. Para dar un mayor realismo a la simulación cada sistema se ejecuta en ordenadores diferentes, y se encuentran conectados a través de una red Ethernet.

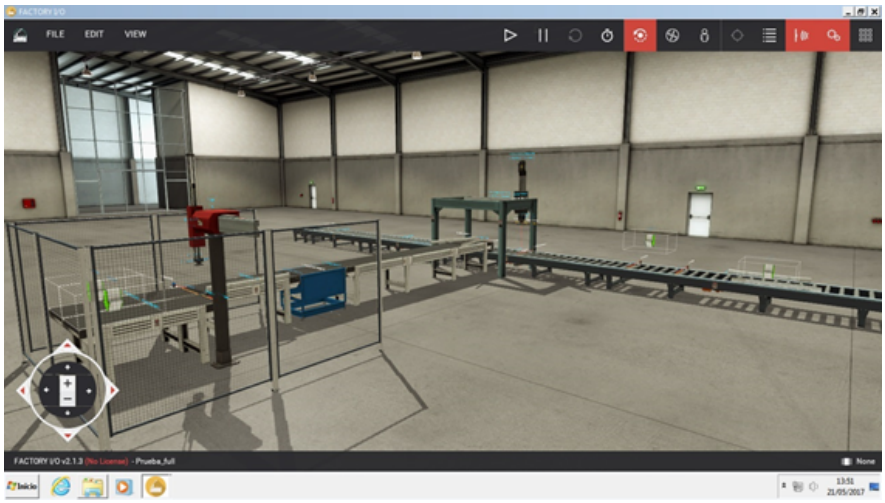


Figura 9.1: Sistema de embalaje para Piezas. Subsistema de montaje de piezas

Figura 9.2: Sistema de embalaje de Piezas.Subsistema de encajonado de piezas

Aplicando la metodología MIAS detallada en el capítulo anterior vamos a ver cómo se puede desarrollar el software de dicho sistema industrial para desplegarlo en dispositivos PLC y el despliegue del modelo en un servidor OPC UA. Para el despliegue en dispositivos industriales PLCs vamos a utilizar el entorno de pro-

gramación de Siemens TIA Portal [143]. En cambio, para el despliegue en OPC UA se ha programado un servidor OPC UA en .NET.

Una vez implementados y desplegados tanto el sistema software en el dispositivo PLC de Siemens como el modelo de intercambio de datos en el servidor OPC UA, vamos a conectar ambos sistemas desarrollando un driver en nuestro servidor UA especifico para PLCs de Siemens, de forma que cualquier cliente UA puede interactuar con dicho dispositivo utilizando el modelo alojado en el servidor OPC UA.

Es importante destacar que tendremos que tener el mismo modelo PIM diseñado en iMMAS, alojado en el dispositivo industrial y el servidor OPC UA, por lo que ambos sistemas manejaran y utilizarán los mismos conceptos, así como el mismo lenguaje, lo que facilita el trabajo en ambos sistemas software al manejar el mismo modelo.

9.2. REQUERIMIENTOS DE USUARIO

9.2.1. DESCRIPCIÓN DEL SISTEMA

Se pretende automatizar una cadena de ensamblaje de piezas. El sistema de ensamblaje consta de dos procesos, el primero de ellos consiste en la colocación de una pieza troquelada encima de otra pieza base, también troquelada. Esta operativa la realizará un robot, ya que por razones de seguridad y rendimiento no puede ser realizada por un operario humano. Una vez que la pieza está montada, dicha pieza será transportada hacia el segundo proceso por una cinta y luego pesada en una báscula con el fin de conocer su peso.

En esta segunda parte del sistema otro robot cogerá la pieza ensamblada y la colocará en una caja. La caja se ha de transportar previamente hasta la zona del robot de encajonado colocada encima de un palet, ya que una vez que el robot la llene con las piezas ensambladas, la caja debe de ser almacenada. La capacidad máxima de cada caja es de 5 piezas, ya que con esta cantidad no se sobrepasa el peso máximo permitido, ni la altura máxima para poder almacenar la caja. Por lo tanto, el robot de encajonado debe de colocar un máximo de 5 piezas en cada caja. Una vez que se ha terminado esta operación la caja debe de ser transportada automáticamente hacia la zona de almacén.

Tanto la zona de montaje como la zona de encajonado debe de estar comple-

tamente libre de la intervención de operarios humanos. Todo el proceso de montaje y encajonado de piezas debe de realizarse mediante robots. El funcionamiento del sistema estará controlado por un autómata, de forma que los operarios solo deben parar y arrancar el sistema mediante un cuadro de mandos basado en botoneras.

Deben existir dos paradas de emergencia, una situada en el cuadro de mandos general y otra en la zona de inspección de cajas. Estas paradas de emergencia detendrán completamente toda la cadena de embalaje.

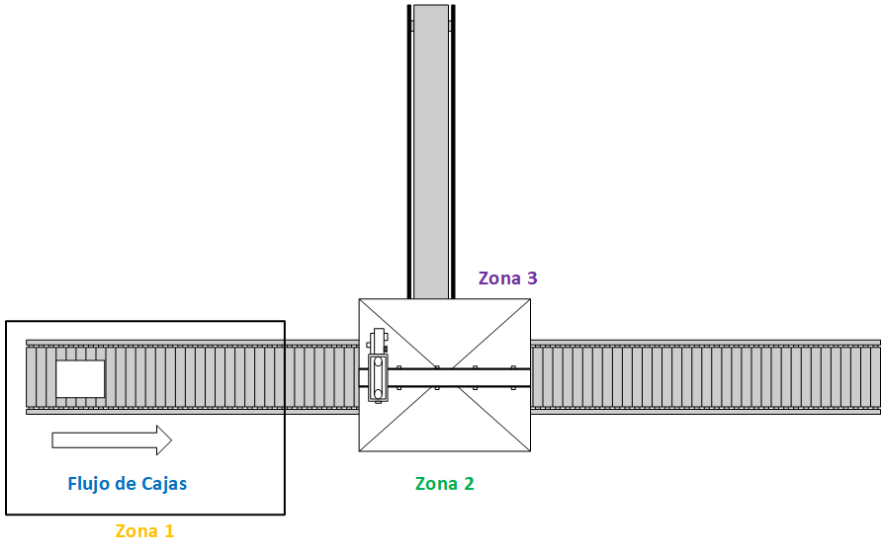


Figura 9.3: Diagrama de flujo de los elementos que forma el sistema.

En la figura anterior se puede ver un plano preliminar de la instalación con las dos zonas de trabajo de los robots de ensamblando y encajonado

9.2.2. ANTECEDENTES HISTÓRICOS

En el sistema actual el montaje y encajonado de piezas se realiza de forma manual. Estas operaciones conllevan un alto riesgo para los operarios que las realizan, ya que las piezas tienen aristas vivas bastante cortantes y un peso considerable. Además, debido a la creciente demanda de estas piezas es necesario aumentar la capacidad productiva de esta línea.

9.2.3. PROBLEMA O DEFICIENCIA QUE SE PRETENDE SOLVENTAR

Se requiere aumentar la seguridad de la operativa de forma que los trabajos que suponen un riesgo para los operarios se realicen utilizando robots. Además, se pretende aumentar la productividad de la línea en un 30 %, en una primera fase y en un 50 % en una segunda fase.

9.2.4. BENEFICIOS QUE SE PRETENDEN CONSEGUIR

Se pretende automatizar el proceso haciéndolo más seguro para los operarios que trabajan en la línea y a la vez se quiere aumentar la capacidad de la línea.

9.2.5. REQUERIMIENTOS.

FUNCIONES QUE EL SISTEMA DEBE DE REALIZAR

ID Requerimiento	Descripción del requerimiento
RQU-1	La operación de ensamblaje de piezas debe ser realizada por un robot.
RQU-2	La operación de encajonado de piezas debe ser realizada por un robot.
RQU-3	La intervención del operario debe de ser solo para arrancar el sistema y pararlo mediante pulsadores o botones. Tanto el transporte como el encajonado y ensamblado de piezas debe de realizarse de forma automática.
RQU-4	El sistema debe de poseer dos paradas de emergencia que deben de detenerlo por completo. RQU-5 La productividad de la línea debe de ser un 30 % más de la actual. De 100 piezas/minuto a 130 piezas/minuto.
RQU-6	Todas las piezas que genere el sistema debe de ser pesadas. Este peso ha de registrarse y almacenarse
RQU-7	Solo se pueden encajonar un máximo de 5 piezas por caja y palet.

Tabla 9.1: Tabla resumen de requerimientos de usuario.

DATOS

Durante todo el proceso de ensamblaje de piezas debemos registrar el peso de cada una de las piezas que se ensambla. Adicionalmente debemos conocer el total de piezas montadas en la zona de ensamblaje y la cantidad de piezas encajonadas en la zona de encajonado. En una primera fase estos registros se pueden de

realizar a mano, pero deben dejarse el sistema de control preparado para poder enviar estos datos de forma automática a otros sistemas software.

INTERFACES

Los únicos interfaces para que el operario pueda interactuar con toda la línea de empaquetado se reducen a un armario con botoneras y dos paradas de emergencia. Pero se debe dejar el sistema de control preparado para poder en una segunda fase ser controlado por un sistema SCADA.

ENTORNO

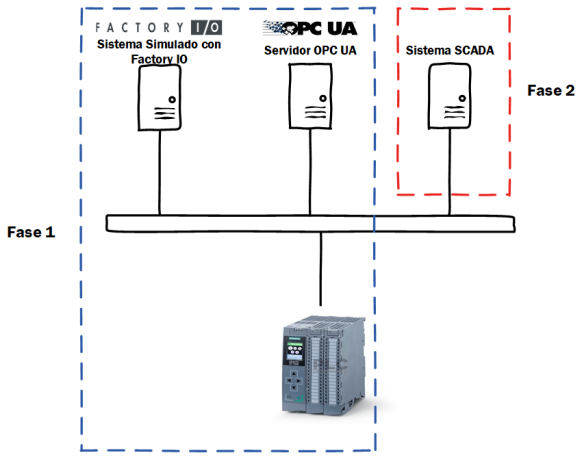


Figura 9.4: Arquitectura del sistema de control.

El entorno del sistema se reduce a las partes mecánicas, rodillos, cintas transportadoras, fotocélulas, motores, barreras y robots; así como todo el cableado eléctrico de todos estos componentes y los elementos necesarios para su conexionado y control (relés de seguridad, disyuntores, variadores de frecuencia, etc..). Todos los elementos deben estar controlados por un PLC de Siemens, preferiblemente de la familia S7-1500. Este debe de poder conectarse mediante ethernet a un sistema externo y comunicarse con algún sistema SCADA o HMI en una segunda fase del proyecto.

RESTRICCIONES

Las mejoras de la línea actual de ensamblaje deben realizarse en las paradas productivas programadas de la fábrica, sin afectar a la producción. El sistema de-

be cumplir con todas las normativas en materia de seguridad, tanto durante su despliegue e implantación, como en su posterior puesta en marcha y utilización.

9.3. DEFINICIÓN DE LOS REQUERIMIENTOS FUNCIONALES DEL SISTEMA

9.3.1. DESCRIPCIÓN FUNCIONAL

PROPÓSITO

Es necesario mejorar la línea de ensamblaje de piezas existente actualmente con dos objetivos principales: aumentar la seguridad para el operario y aumentar la productividad de la línea. Ambos objetivos se pretenden lograr mediante la introducción de robots y sistemas de transporte basados en rodillos y cintas transportadoras. Todos estos elementos se van a controlar de forma autónoma mediante un PLC, dejando la intervención del operario circunscrita a tareas de supervisión y monitorización. El sistema está compuesto por dos áreas, en la primera de ellas se ensamblarán las piezas y en la segunda se colocan en cajas para su posterior almacenaje.

Zona de ensamblaje.

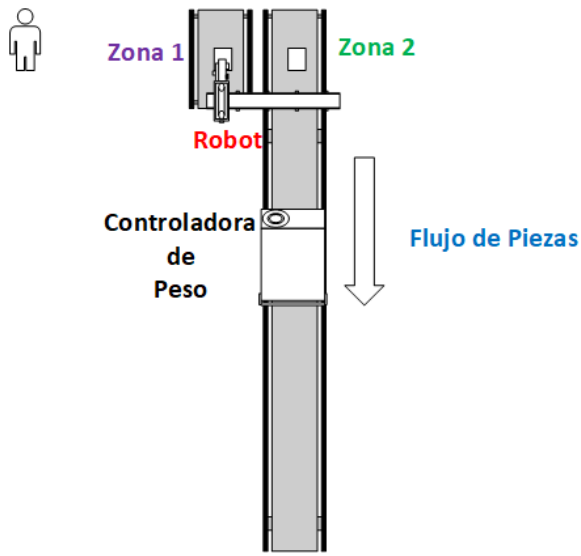


Figura 9.5: Diagrama de Flujo para la zona de ensamblaje.

En esta zona se realiza en el montaje de las piezas, se colocará una tapa que se encuentra en la zona 1, sobre una base que se encuentra en la zona 2, como se puede ver en la siguiente figura:

El montaje de las piezas se realizará mediante un robot, que solo necesita trabajar en coordenadas X y Z (desplazamiento horizontal y en profundidad). La zona 1 no generará otra pieza hasta que la existente no haya sido cogida por el robot. Esto se controlará mediante un foto-célula. Una vez que el robot coloque la tapa sobre la pieza base se bajará la barrera de bloqueo para que la pieza se pueda dirigir a la zona de encajonado, volviendo el robot a la zona 1 para coger la nueva tapa. Una vez que la pieza ensamblada abandone la zona dos se generará otra pieza base, esto se controlará mediante otra foto-célula. En el tránsito entre la zona de ensamblado y la de encajonado se pesará la pieza mediante una controladora de peso. Todos los elementos de esta zona estarán manejados por un PLC siemens S7-1500, robot, cinta, foto-células, barreras de bloqueo, controladora de peso y generación de piezas. En la figura 9.6 podemos ver todos estos elementos creados en el simulador de Factory IO:

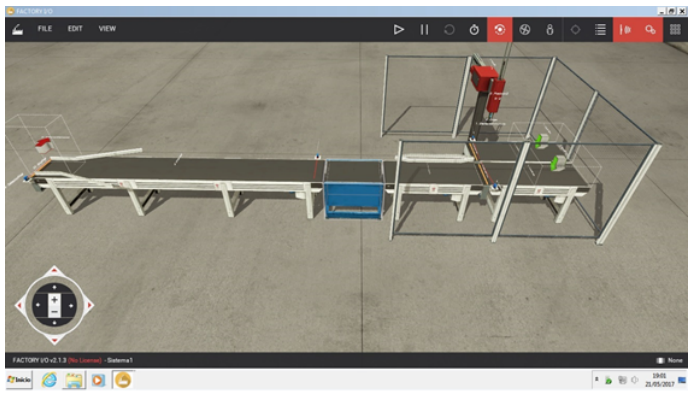


Figura 9.6: Modelo en 3D realizado para la zona de ensamblaje

Este escenario consta de:

- Sensores:
  - 4 detectores digitales, posicionados en las cintas, que indican la presencia de pieza o la ausencia de la misma.

- Actuadores:
  - 4 motores para el movimiento de las cintas transportadoras, estos motores son de velocidad variable utilizando una señal analógica.
  - Un motor con solo orden de marcha y paro, para la cinta de la controladora de peso.
  - 3 barreras digitales para parar las piezas.
  - 2 órdenes de generación de pieza, estas ordenes se le envían al sistema desde el PLC para que este genere piezas.
  - Un robot de ensamblado que esta compuesto por :
    - Orden en la coordenada X. Valor analógico.
    - Lectura de la coordenada X. Valor analógico.
    - Orden en la coordenada Z. Valor analógico.
    - Lectura de la coordenada Z. Valor analógico.
    - Detección de pieza. Valor digital.
    - Orden de soltar/coger pieza. Valor digital

Requerimientos del Sistema de Control:

- Entradas analógicas (0-10V) : 3
- Salidas analógicas (0-10V): 6
- Entradas digitales: 5
- Salidas digitales: 7

Zona de Encajonado.

El objetivo de esta parte del sistema es colocar las piezas que le llegan de la zona 1 en la caja que se encuentra en la zona 2. Para ello, el sistema generara cajas con palet en la estación 1. El conjunto caja/palet permanecerá bloqueado en esta estación hasta que se haya liberado la caja/palet de la zona 2. En ese momento la barrera de stop de la estación 1 se liberara para dejar pasar el conjunta caja/palet. Una vez que este haya pasado se volverá a subir la barrera de stop para bloquear la siguiente conjunto de caja/palet. Toda esta operativa será controlada mediante

foto-células que detectaran si las cajas están posicionadas o no. En la zona 2 un robot que trabaja en coordenadas X, Y y Z, cogerá la pieza bloqueada en la zona 3 mediante una barrera de stop, y la depositará en la caja bloqueada en zona 2. Esta operación se repetirá hasta cinco veces antes de dejar salir el conjunto caja/palet hacia la zona de almacenaje.

En la siguiente figura podemos ver todas las zonas y elementos del sistema:

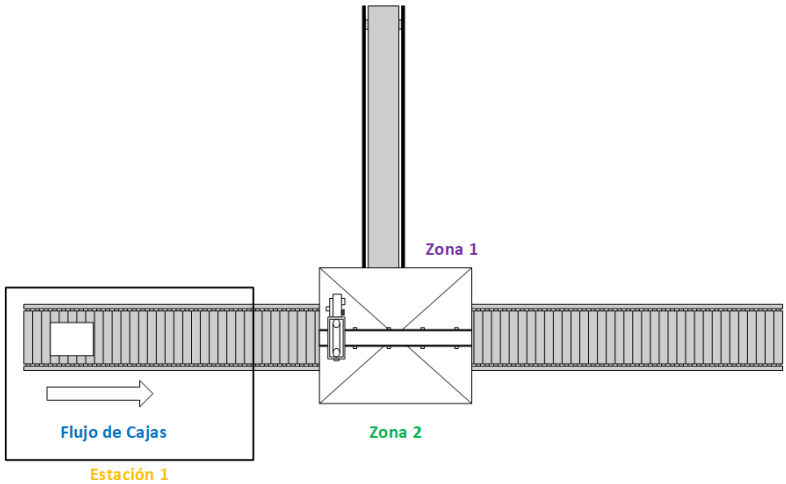


Figura 9.7: Diagrama de Flujo para la zona de encajonado.

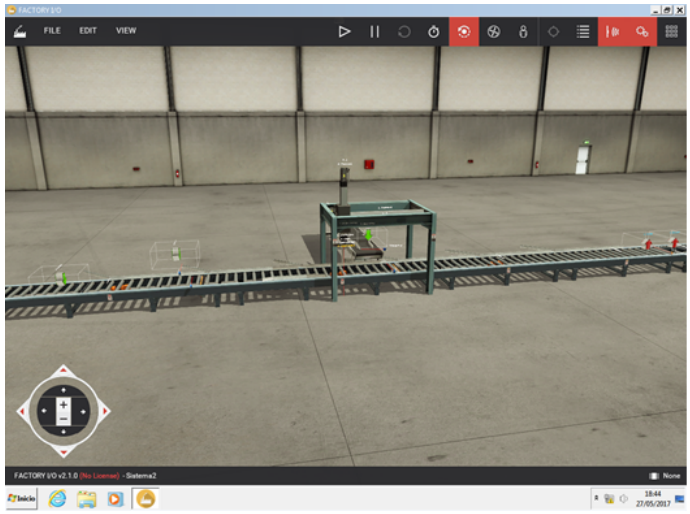


Figura 9.8: Modelo en 3D realizado para la zona de encajonado.

Todos los elementos de esta zona estarán manejados por el mismo PLC siemens S7-1500 de la zona de ensamblaje, robot, cinta, foto-células, barreras de stop y generación de cajas/palet. En la figura 9.8 podemos ver todos estos elementos creados en el simulador de Factory IO:

Este escenario consta de:

- Sensores
  - 4 detectores digitales, posicionados en las cintas, que indican la presencia de pieza/palets o la ausencia de los mismos.
- Actuadores:
  - 4 motores para el movimiento de las cintas transportadoras con motores de velocidad variable utilizando una señal analógica.
  - 2 topes digitales para para el conjunto palet/caja.
  - 1 orden de generación de palet y otra orden de generación de caja que se envían al sistema en 3D desde el PLC para que dicho sistema genere tanto palets como cajas.
  - Un robot de ensamblado que está compuesto por :
    - Orden en la coordenada X. Valor analógico.
    - Lectura de la coordenada X. Valor analógico.
    - Orden en la coordenada Y. Valor analógico.
    - Lectura de la coordenada Y. Valor analógico.
    - Orden en la coordenada Z. Valor analógico.
    - Lectura de la coordenada Z. Valor analógico.
    - Detección de pieza. Valor digital.
    - Orden de soltar/coger pieza. Valor digital

Requerimientos del Sistema de Control:

- Entradas analógicas (0-10V) : 3.
- Salidas analógicas (0-10V): 7
- Entradas digitales: 5
- Salidas digitales: 5

ALCANCE

El alcance del proyecto incluye la automatización de todos los elementos del subsistema de ensamblaje y encajonado, utilizando un único PLC S7-1500. Todos estos elementos deben de funcionar de forma autónoma, siendo el PLC el que los coordine y los controle según las especificaciones definidas en el apartado anterior. El sistema se implementará mediante un simulador en 3D de Factory IO, el cual deberá de estar controlado por el programa de PLC desarrollado.

En esta primera fase no se implementará ningún sistema SCADA o HMI que monitorice/supervise el sistema. El interfaz del operario se limitará a un conjunto de botones de Start y Stop y un tercer botón para una parada general de seguridad. Pero debe establecerse las bases para poder desarrollar un sistema SCADA en una segunda fase del proyecto, por lo que las comunicaciones, la programación y la infraestructura debe dejarse preparada para poder afrontar esta segunda fase. Para cubrir estas necesidades de comunicación se va a utilizar un servidor OPC UA, que se comunicará con el PLC de Siemens, el cual podrá ser utilizado por cualquier sistema SCADA o HMI que soporte este estándar.

ACRÓNIMOS Y DEFINICIONES

Term,Acronym	Definición
PLC	Programmable Logic Controller
HMI	Human Machine Interface
SCADA	Supervisión, Control y Adquisición de Datos
OPC UA	Ole for Process Control (Unified Architecture)

REFERENCIAS

Nº Documento	Titulo del Documento
RG-001-UR	User Requirements
RG-001-TP	Test Protocol
RG-001-MT	Matriz de trazabilidad
RG-001-CS	Certificación del Sistema.

9.4. DESCRIPCIÓN GENERAL

9.4.1. CARACTERÍSTICAS DE LA APLICACIÓN

INTERFACES DE USUARIO

El interfaz para el operario consistirá en un cuadro con botoneras y lámparas luminosas. El cuadro constará de tres botones, uno de arranque, otro de paro y otro de reset general del sistema. Una parada de emergencia que detendrá toda la línea de embalaje y dos indicadores luminosos, uno rojo para indicar que el sistema está parado y otro verde para indicar que el sistema está en marcha.

Figura 9.9: Modelo en 3D de la cuadro de control basado en botoneras.

INTERFACES HARDWARE

El PLC debe estar dotado de una tarjeta de comunicaciones ethernet y de tarjetas de entradas y salidas tanto analógicas como digitales para conectar todos los elementos del sistema industrial. En concreto se necesitan unas 6 entradas analógicas, 13 salidas analógicas, 14 entradas digitales y 15 salidas digitales. Se valora la instalación de tarjetas adicionales de E/S para cubrir futuras necesidades de ampliación.

INTERFACES SOFTWARE

En esta primera fase no se contemplan la conexión con ningún sistema externo. Sin embargo, se va a desplegar un servidor OPC UA, el cual se conectará al PLC mediante ethernet y en el que se podrán consultar toda la información que el PLC esta manejando. Este servidor OPC UA tiene el objetivo de cubrir futuras necesidades de integración y comunicación con otros sistemas o interfaces software.

9.4.2. RESTRICCIONES

Todo el sistema estará controlado por un PLC de Siemens S-7 1500, por lo que los sistemas software que se deseen utilizar ahora, o en fases posteriores de ampliación de la línea deben poder comunicarse con este dispositivo. El protocolo de comunicaciones debe de utilizar Ethernet como medio de interconexión entre el PLC y el sistema software.

9.5. REQUERIMIENTOS

9.5.1. REQUERIMIENTOS FUNCIONALES

ID RQU	ID RQF	Descripción del requerimiento	Criticidad	Version
RQU-1	RQF-1	Se instalará un robot de dos grados de libertad, en la zona de ensamblado, que se desplazará en las coordenadas X y Z.	Alta	1.0
RQU-2	RQF-2	Se instalará un robot de tres grados de libertad, en la zona de ensamblado, que se desplazará en las coordenadas X, Y y Z.	Alta	1.0
RQU-3	RQF-3	Se instalará un cuadro de control con tres botones e indicadores luminosos, con los que se podrá parar, arrancar y reiniciar el sistema.	Alta	1.0
RQU-3	RQF-4	Se instalaran cintas transportadoras de velocidad y cintas de rodillos que transportaran, las piezas y las cajas entre las diferentes zonas del sistema	Alta	1.0
RQU-3	RQF-5	Se instalarán barreras de stop y fotocélulas que regularán todo el flujo de elementos transportados por las diferentes cintas.	Alta	1.0
RQU-3	RQF-6	Todos los elementos del sistema, robots, cintas, barreras, botoneras, controladores de peso, fotocélulas y cuadros de control deben estar cableados al controlador PLC S7-1500 de siemens, ya que este elemento se encargará de controlar todo el sistema.	Alta	1.0
RQU-4	RQF-7	Se instalará una parada de emergencia en el cuadro de mandos y otra en la zona de encajonado cerca del robot, estas paradas de emergencia se cablearan al circuito de seguridad del sistema, relés de seguridad, que detendrá, cortando la corriente eléctrica, todo el sistema.	Alta	1.0
RQU-5	RQF-8	El robot de ensamblado tiene unas velocidades máximas, según el fabricante, de 190 piezas por minuto para el robot de ensamblado y 180 para el robot de encajonado. Durante la puesta en marcha del sistema se ajustaran todos los elementos para alcanzar las 130 piezas por minuto.	Media	1.0.
RQU-6	RQF-9	Se instalará una controladora de pesos en línea, que enviará el peso de las piezas al PLC.	Media	1.0
RQU-7	RQF-10	El robot de encajonado contará el número de piezas que coloca en la caja, siempre como máximo 5 y reiniciará el ciclo, dejando pasar la caja y esperando a la siguiente caja.	Alta	1.0

Tabla 9.2: Tabla resumen de requerimientos funcionales.

9.5.2. REQUERIMIENTOS DE RENDIMIENTO

La productividad de la línea debe aumentarse en un 30 % gracias a la automatización de todos los elementos de la línea y a la mayor cadencia de trabajo de los robots.

9.5.3. REQUERIMIENTOS DE BASE DE DATOS

No existe requerimientos de base datos para esta primera fase del proyecto. El registro los pesos de las piezas y el número de piezas ensambladas y encajonadas, se realizará manualmente.

9.5.4. CARACTERÍSTICAS SOFTWARE DE SISTEMA

FIABILIDAD.

Todo el sistema software debe instalarse/desplegarse con componentes que aseguren el funcionamiento 24 horas por 7 días semanales por 365 días al año de la línea de montaje con el menor número de paros y fallos posibles.

DISPONIBILIDAD

La disponibilidad del Sistema debe de ser de 24 horas x 7días semanales x 365 días al año, ya que la línea de ensamblaje y encajonado no debe de parar para entregar la producción comprometida.

SEGURIDAD

La comunicación e intercambio de información entre el servidor OPC UA y el PLC se realizará utilizando una red interna sin conexión a sistemas externos como Internet o redes de sistemas de negocio. Las conexiones al servidor OPC UA que se va a desplegar se hará utilizando los mecanismos de certificada digital soportadas por el propio estándar UA. Este estándar también establece que las conexiones entre los clientes y el servidor utilicen SSL.

PORTABILIDAD

En esta primera fase del proyecto no existe requisitos de portabilidad.

9.6. MATRIZ DE TRAZABILIDAD

ID RQU	ID RQF	ID Protcolo Testeo
RQU-1	RQF-1	PRT-1
RQU-2	RQF-2	PRT-4
RQU-3	RQF-3	PRT-6
RQU-3	RQF-4	PRT-2, PRT-5
RQU-3	RQF-5	PRT-2, PRT-5
RQU-3	RQF-6	PRT-1, PRT-2, PRT-3, PRT-4, PRT-5, PRT-6
RQU-4	RQF-7	PRT-6
RQU-5	RQF-8	PRT-1, PRT-2, PRT-3, PRT-4, PRT-5, PRT-6
RQU-6	RQF-9	PRT-2
RQU-7	RQF-10	PRT-2, PRT-3, PRT-4, PRT-5

Tabla 9.3: Matriz de trazabilidad.

9.7. DISEÑO DEL SISTEMA

9.7.1. IDENTIFICACIÓN DE ELEMENTOS Y AGRUPACIÓN EN SUBSISTEMAS

Se han identificado los siguientes subsistemas:

- Subsistema de Embalaje: Es el subsistema principal que se encarga de coordinar el funcionamiento del subsistema de generación de piezas y el de empaquetado.
- Subsistema de Generación de Piezas: Los elementos que forma este subsistema se encarga de ensamblar las piezas, pesarlas y transportarlas hacia la zona de empaquetado.
- Subsistema de Empaquetado: Los elementos de este subsistema se encarga de colocar las piezas en cajas y de transportar palet/pieza hacia la zona de almacén.

Tablas de identificación de subsistemas y relaciones entre ellos:

Nombre Elemento	Tipo	Propiedades
Sistema de Embalaje	Subsistema	
Sistema Genera Piezas	Subsistema	
Sistema de Encajonado	Subsistema	

Tabla 9.4: Tabla de los subsistemas detectados y sus relaciones.

SUBSISTEMA DE EMBALAJE

Para este subsistema se han detectado a su vez otros subsistemas anidados como el subsistema de Encajonado y el subsistema Genera Piezas.

Tabla de Elementos:

Nombre Elemento	Tipo	Propiedades
Subsistema de Embalaje	Subsistema	
Subsistema de Encajonado	Subsistema anidado	
Subsistema Genera Piezas	Subsistema anidado	
Botón de Inicio	Sensor Digital	
Botón de Paro	Sensor Digital	
Luz verde sistema en marcha	Salida Digital	
Luz Roja sistema en paro	Salida Digital	
Luz Amarilla sistema en avería	Salida Digital	

Tabla 9.5: Tabla de elementos para el Subsistema de Embalaje .

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
Pertenece	Subsistema de Embalaje	Subsistema Genera Piezas	1 a 1
Pertenece	Subsistema de Embalaje	Subsistema Encajonado	1 a 1
Stop	Subsistema de Embalaje	Boton	1 a 1
Start	Subsistema de Embalaje	Boton	1 a 1
Verde	Subsistema de Embalaje	Luz	1 a 1
Roja	Subsistema de Embalaje	Luz	1 a 1
Amarilla	Subsistema de Embalaje	Luz	1 a 1

Tabla 9.6: Tabla para identificar las relaciones entre los subsistemas identificados.

SUBSISTEMA GENERA PIEZAS

Para este subsistema se han detectado a su vez los siguientes subsistemas anidados, el Subsistema de Montaje de Piezas y el Subsistema de Transporte de Piezas.

Nombre Elemento	Tipo	Propiedades
Subsistema Genera Piezas	Subsistema	
Subsistema Montaje de Piezas	Subsistema anidado	
Subsistema de Transporte de Piezas	Subsistema anidado	

Tabla 9.7: Tabla de los subsistemas detectados y sus relaciones para el sistema Montaje de Pieza.

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
Pertenece	Subsistema Genera Piezas	Subsistema Montaje de Piezas	1 a 1
Pertenece	Subsistema Genera Piezas	Subsistema de Transporte de Piezas	1 a 1

Tabla 9.8: Tabla para identificar las relaciones entre los subsistemas identificados.

SUBSISTEMA MONTAJE DE PIEZAS

Tabla de Elementos:

Nombre Elemento	Tipo	Propiedades
Montaje de Piezas	Subsistema	
Barrera Pieza Base	Actuador Discreto	
Barrera Pieza Tapa	Actuador Discreto	
Generar Pieza Base	Actuador Discreto	
Generar Pieza Tapa	Actuador Discreto	
Motor Cinta Pieza Base	Actuador Analógico	
Motor Cinta Pieza Tapa	Actuador Analógico	
Robot	Actuador	
Detector de Pieza Base	Sensor Digital	
Detector de Pieza Tapa	Sensor Digital	

Tabla 9.9: Tabla de elementos para el subsistema Montaje de Piezas.

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
StopBase	MontajePieza	ActuadorDiscreto	1 a 1
StopTapa	MontajePieza	ActuadorDiscreto	1 a 1
GeneraTapa	MontajePieza	ActuadorDiscreto	1 a 1
GeneraBase	MontajePieza	ActuadorDiscreto	1 a 1
CintaGeneraBase	MontajePieza	ActuadorAnalogico	1 a 1
CintaGeneraTapa	MontajePieza	ActuadorAnalogico	1 a 1
DetectorBase	MontajePieza	Detector	1 a 1
DetectorTapa	MontajePieza	Detector	1 a 1
Pertenece	MontajePieza	Robot	1 a 1

Tabla 9.10: Tabla para identificar las relaciones del subsistema Montaje de Piezas.

SUBSISTEMA TRANSPORTE DE PIEZAS

Tabla de Elementos:

Nombre Elemento	Tipo	Propiedades
Transporte de Piezas	Subsistema	
Barrera Final	Actuador Discreto	
Motor Cinta Intermedia	Actuador Analógico	
Motor Cinta Final	Actuador Analógico	
Detector de Pieza Completa	Sensor Digital	
Detector de Pieza Final	Sensor Digital	

Tabla 9.11: Tabla de elementos para el subsistema Transporte de Piezas.

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cadinalidad
StopFinal	TransportePiezas	ActuadorDiscreto	1 a 1
DetectorPiezaCom	TransportePiezas	Detector	1 a 1
DetectorFinal	TransportePiezas	Detector	1 a 1
CintaIntermedia	TransportePiezas	ActuadorAnalogico	1 a 1
CintaFin	TransportePiezas	ActuadorAnalogico	1 a 1

Tabla 9.12: Tabla para identificar las relaciones del subsistema Transporte de Piezas.

SUBSISTEMA ENCAJONADO

Para este subsistema se han detectado a su vez los siguientes subsistemas:

Nombre Elemento	Tipo	Propiedades
Subsistema de Encajonado	Subsistema	
Subsistema de Envió de Palet	Subsistema anidado	
Subsistema de Recogida de piezas	Subsistema anidado	
Subsistema de Transporte de Embalaje	Subsistema anidado	

Tabla 9.13: Tabla de los subsistemas anidados detectados y sus relaciones para el subsistema Encajonado.

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
Pertenece	Subsistema Encajonado	Subsistema de Envío de Palet	1 a 1
Pertenece	Subsistema Encajonado	Subsistema de Recogida de piezas	1 a 1
Pertenece	Subsistema Encajonado	Subsistema de Transporte de Embalaje	1 a 1

Tabla 9.14: Tabla para identificar las relaciones entre los subsistemas identificados.

SUBSISTEMA DE ENVÍO DE PALET

Tabla de Elementos:

Nombre Elemento	Tipo	Propiedades
Envio de Palet	Subsistema	
Barrera Robot	Actuador Discreto	
Barrera Fin	Actuador Discreto	
Motor Cinta Robot	Actuador Analógico	
Motor Cinta Fin	Actuador Analógico	
Detector de Palet Robot	Sensor Digital	
Detector de Palet Fin	Sensor Digital	

Tabla 9.15: Tabla de elementos para el subsistema Montaje de Piezas.

Tabla de relaciones::

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
StopRobot	EnviaPalet	ActuadorDiscreto	1 a 1
StopFin	EnviaPalet	ActuadorDiscreto	1 a 1
CintaRobot	EnviaPalet	ActuadorAnalogico	1 a 1
CintaFin	EnviaPalet	ActuadorAnalogico	1 a 1
DetectorRobot	EnviaPalet	Detector	1 a 1
DetectorFin	EnviaPalet	Detector	1 a 1

Tabla 9.16: Tabla para identificar las relaciones del subsistema Montaje de Piezas.

SUBSISTEMA RECOGE PIEZA

Tabla de Elementos:

Nombre Elemento	Tipo	Propiedades
Recoge Pieza	Subsistema	
Robot	Actuador	
Detector Palet Posicionado	Sensor Digital	

Tabla 9.17: Tabla de elementos para el subsistema Recoge Pieza.

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
Pertenece	EnviaPalet	Robot	1 a 1
DetectorPalet	EnviaPalet	Detector	1 a 1

Tabla 9.18: Tabla para identificar las relaciones del subsistema Recoge Pieza.

SUBSISTEMA DE TRANSPORTE DE EMBALAJE

Tabla de Elementos:

Nombre Elemento	Tipo	Propiedades
Transporte Embalaje	Subsistema	
Barrera Palet	Actuador Discreto	
Genera Palet	Actuador Discreto	
Genera Caja	Actuador Discreto	
Motor Cinta Inicio	Actuador Analógico	
Detector de Palet	Sensor Digital	
Detector de Palet Caja	Sensor Digital	

Tabla 9.19: Tabla de elementos para el subsistema Transporte de Embalaje.

Tabla de relaciones:

Nombre Relación	Elemento Origen	Elemento Destino	Cardinalidad
StopPalet	TransporteEmbalaje	ActuadorDiscreto	1 a 1
GeneraPalet	TransporteEmbalaje	ActuadorDiscreto	1 a 1
GeneraCaja	TransporteEmbalaje	ActuadorDiscreto	1 a 1
CintaInicio	TransporteEmbalaje	ActuadorAnalogico	1 a 1
DetectorCaja	TransporteEmbalaje	Detector	1 a 1
DetectorPalet	TransporteEmbalaje	Detector	1 a 1

Tabla 9.20: Tabla para identificar las relaciones del subsistema Transporte de Embalaje.

9.7.2. MODELO PIM.

Una vez que hemos identificado todos los elementos, subsistemas y las relaciones entre ellos podemos empezar a construir el modelo PIM utilizando el lenguaje de modelado iMMAS con el perfil de MOSIE. Para ello y siguiendo una estrategia ascendente vamos a empezar a construir el modelo desde los subsistemas mas simples hasta obtener el modelo del sistema completo, siempre siguiendo una estrategia de modelado jerárquica.

MODELO PARA EL SUBSISTEMA MONTAJEPIEZA

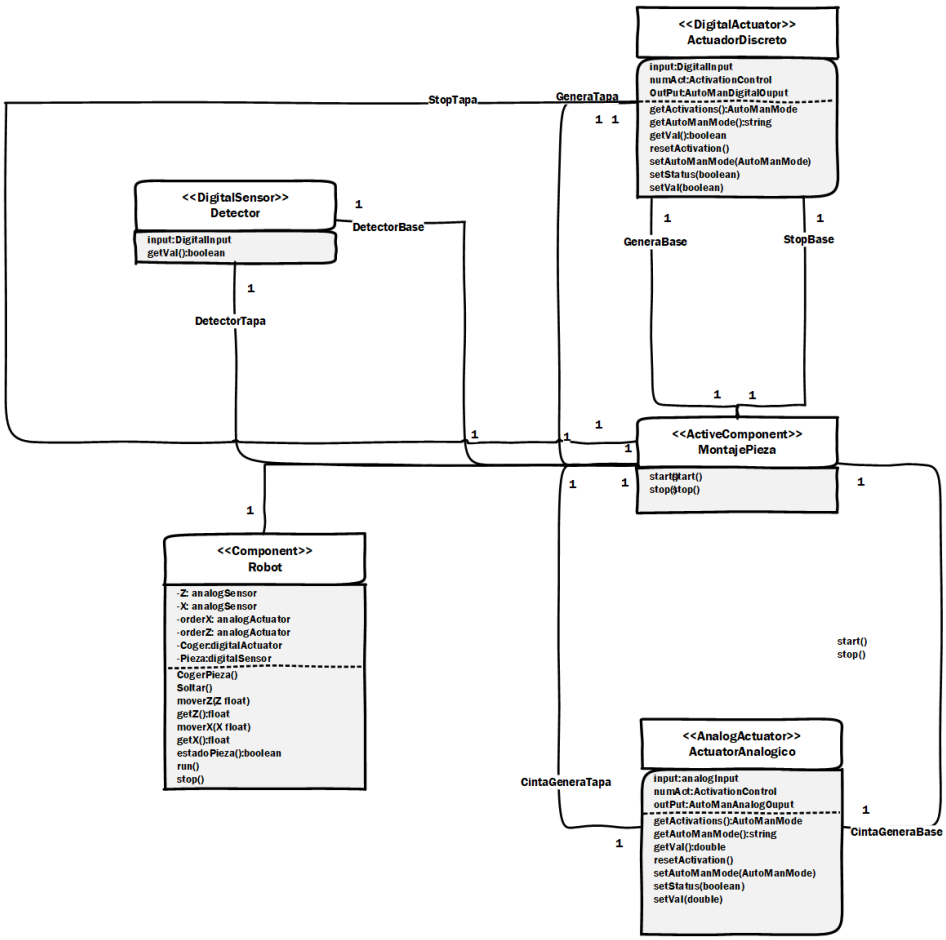


Figura 9.10: Modelo PIM con el perfil MOSIE realizado para el subsistema de Montaje de Piezas.

Para construir este modelo hemos utilizado los componentes de *DigitalActuator* y *DigitalSensor* del paquete Devices de MOSIE, y hemos definido un componente que representa al Robot de montaje de piezas y otro componente activo *MontajePieza* para modelar el subsistema *MontajePieza*. Además, para el componente activo *MontajePieza* se ha creado la siguiente máquina de estado:

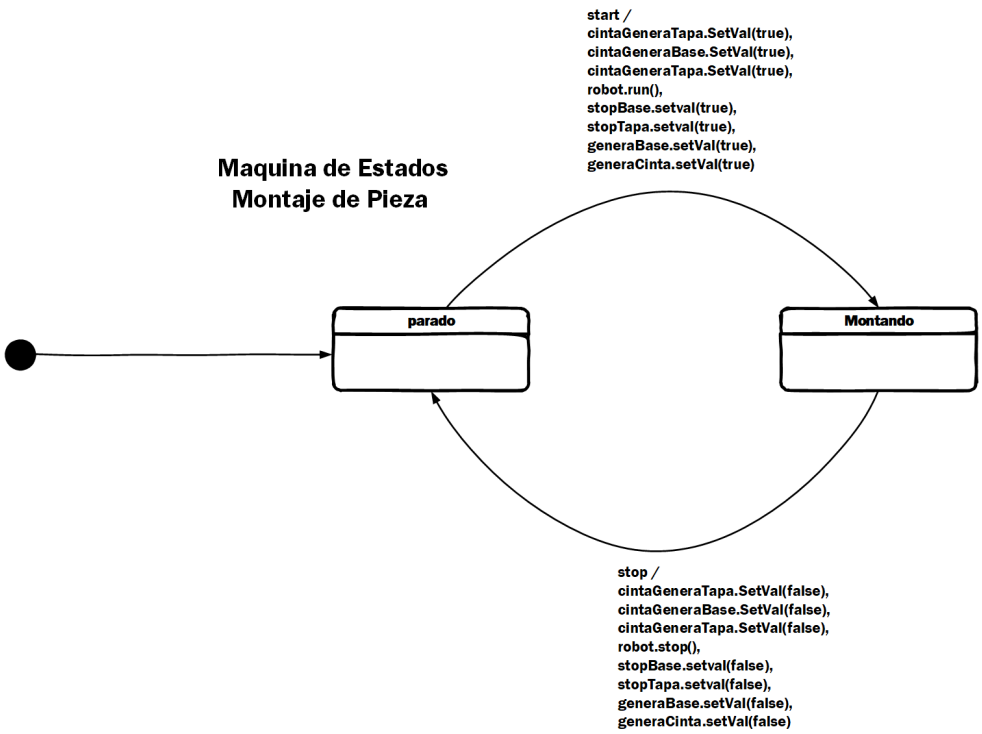


Figura 9.11: Máquina de estados para el subsistema de Montaje de Piezas .

Como se puede ver en la figura anterior la máquina de estados asociada al subsistema *MontajePieza*, consta tan solo de dos estados ”parado” y ”montando”. El primero es para representar cuando el sistema no esta realizando ninguna acción y el segundo cuando el sistema esta montando piezas. Las transiciones entre estos estados están asociados al arranque de todos los elementos del sistema cuando se invoca la operación *start()* del elemento *MontajePiezas* y a la parada de todos los elementos cuando se invoca la operación *stop()* del mismo elemento.

MODELO PARA EL SUBSISTEMA TRANSPORTEPIEZAS

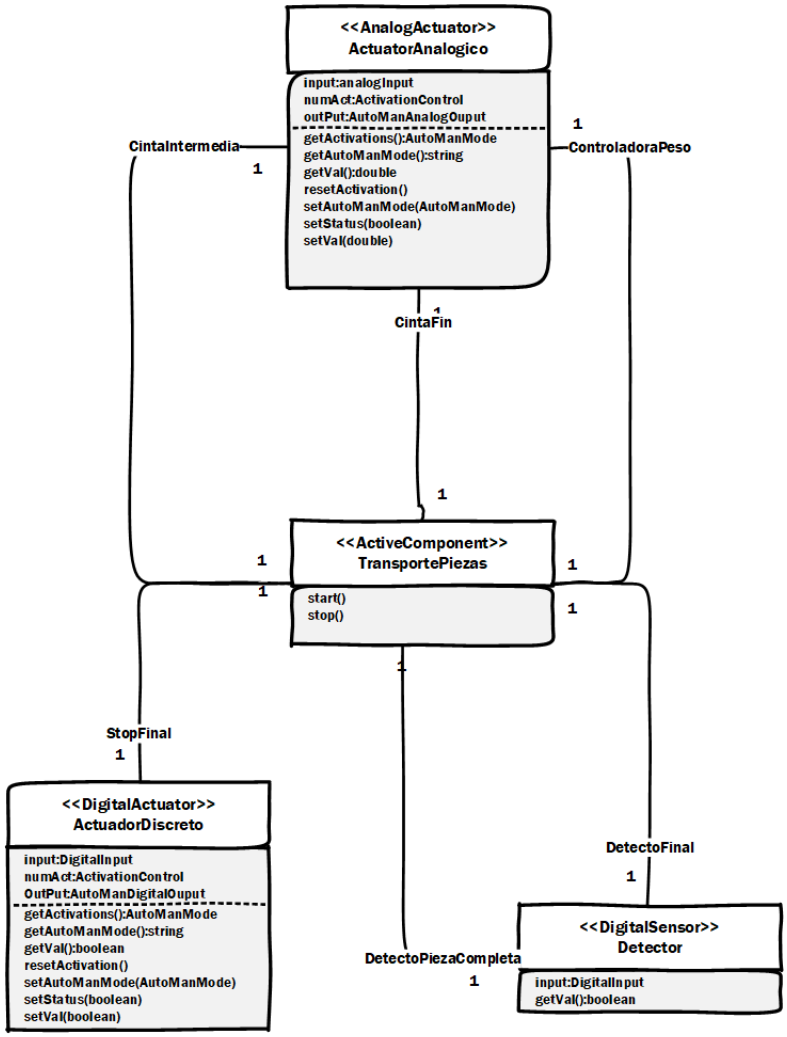


Figura 9.12: Modelo PIM con el perfil MOSIE realizado para el subsistema de Montaje de Piezas.

Para construir este modelo se ha utilizado *DigitalActuator* y *DigitalSensor* del perfil MOSIE, y se ha creado un nuevo componente activo *TransportePiezas* con la siguiente máquina de estados:

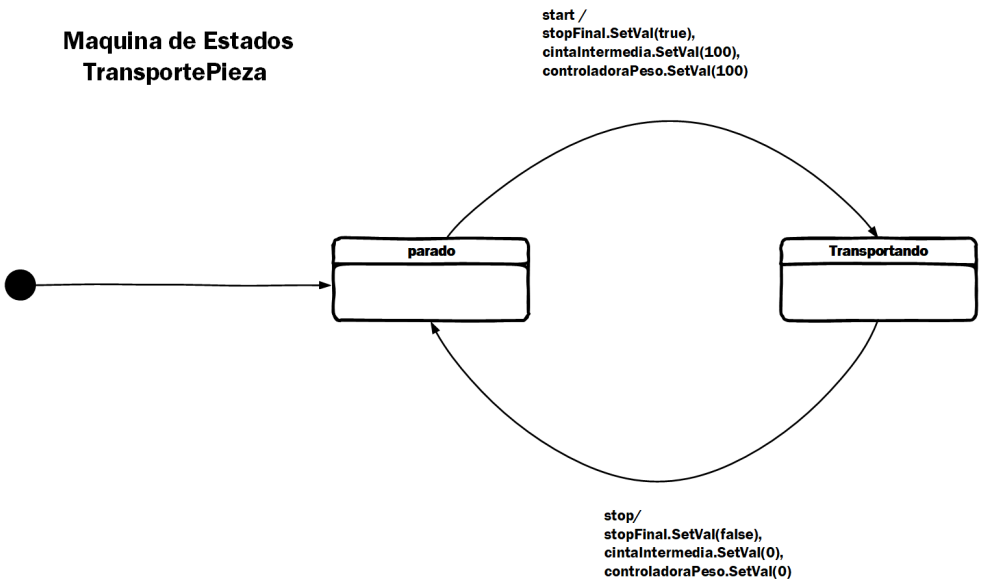


Figura 9.13: Máquina de estados para el sistema de Montaje de Piezas.

Como se puede ver en la figura anterior la máquina de estados asociada al subsistema *TransportePiezas* tiene dos estados ”parado” y "transportando". El primero es para representar cuando el sistema no esta realizando ninguna acción y el segundo cuando el sistema esta transportando piezas. Las transiciones entre estos estados están asociados al arranque de todos los elementos del sistema cuando se invoca la operación *start()* del elemento *TransportePiezas* y a la parada de todos los elementos cuando se invoca la operación *stop()*.

MODELO PARA EL SUBSISTEMA GENERAPIEZAS

Este subsistema esta formado por los dos subsistemas anteriores *MontajePieza* y *TransportePieza*, de forma que los operaciones del elemento *SistemaGeneraPiezas* controlan ambos subsistemas. El modelo para el subsistema *GeneraPiezas* es el siguiente:

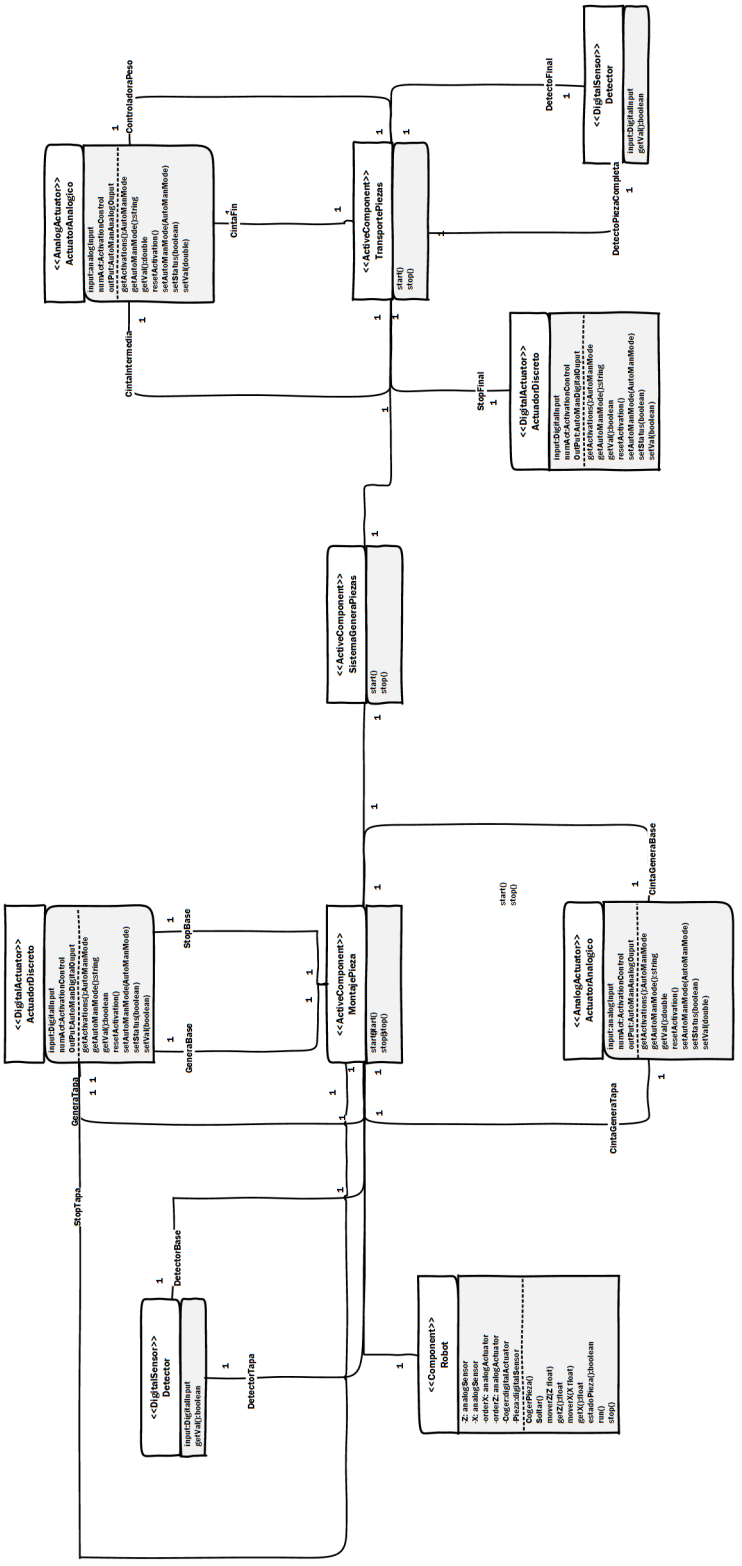


Figura 9.14: Modelo con MOSIE realizado para el sub-sistema de GeneraPiezas.

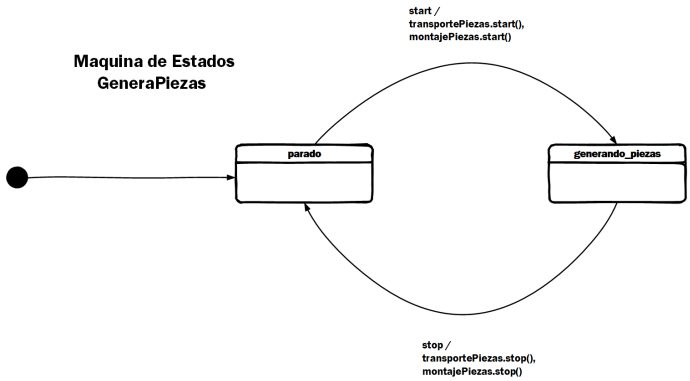


Figura 9.15: Máquina de estados para el subsistema sistemaGeneraPiezas .

La máquina de estados creada para este subsistema esta formado también por dos estados, uno que representa que el sistema esta parado (parado) y otro que esta en marcha (*generando_piezas*). Las transiciones entre estos estados están asociadas a los métodos *start* y *stop*, los cuales desencadenan el arranque y paro de los subsistemas *transportePiezas* y *montajePiezas*.

MODELO PARA EL SUBSISTEMA ENVIOPALET

La construcción de este modelo se ha basado en la utilización de los conceptos *AnalogActuator*, *DigitalActuator* y *DigitalSensor* del perfil de MOSIE.

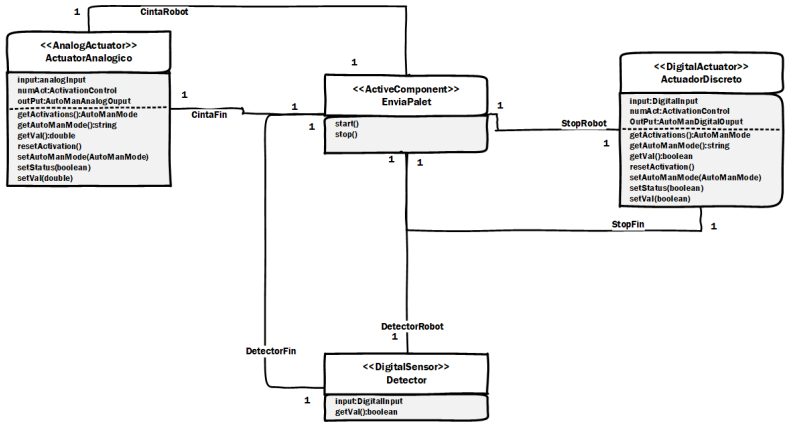


Figura 9.16: Modelo con el perfil MOSIE realizado para el subsistema de EnvioPalet.

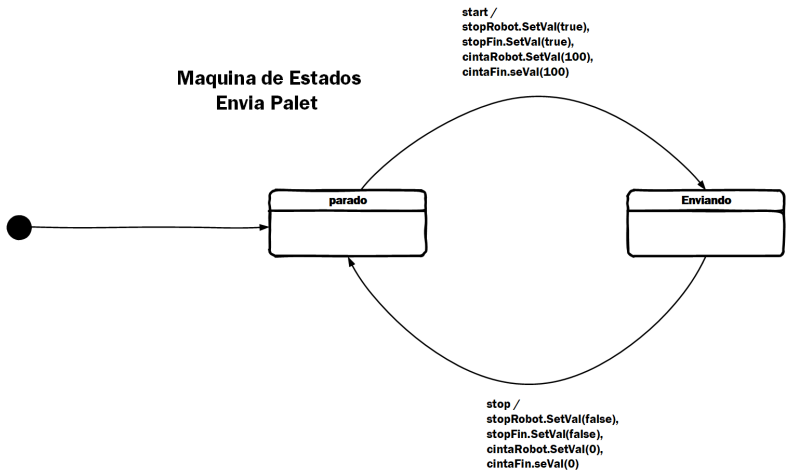


Figura 9.17: Máquina de estados para el subsistema EnvioPalet.

La máquina de estados creada para este sub-sistema esta formado también por dos estados, uno que representa que el sistema esta parado (parado) y otro que esta en marcha (enviando). Las transiciones entre estos estados están asociadas a los métodos start y stop, los cuales desencadenan el arranque y paro de los elementos activos del modelo: *cintaRobot,cintaFin,stopRobot* y *stopFin*.

MODELO PARA EL SUBSISTEMA TRANSPORTEEMBALAJE

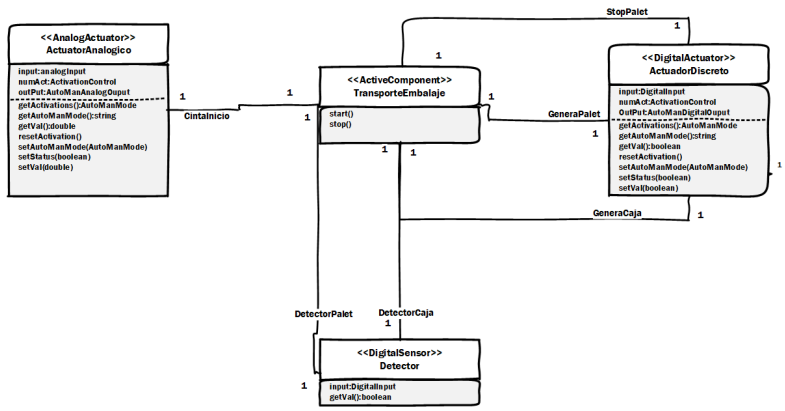


Figura 9.18: Modelo con MOSIE realizado para el subsistema TransporteEmbalaje.

El modelo creado para este subsistema utiliza los componentes *AnalogActuator*, *DigitalActuator* y *DigitalSensor* del perfil de MOSIE *device* y un ActiveComponent TransportEmbalaje que tiene asociada la siguiente máquina de estados:

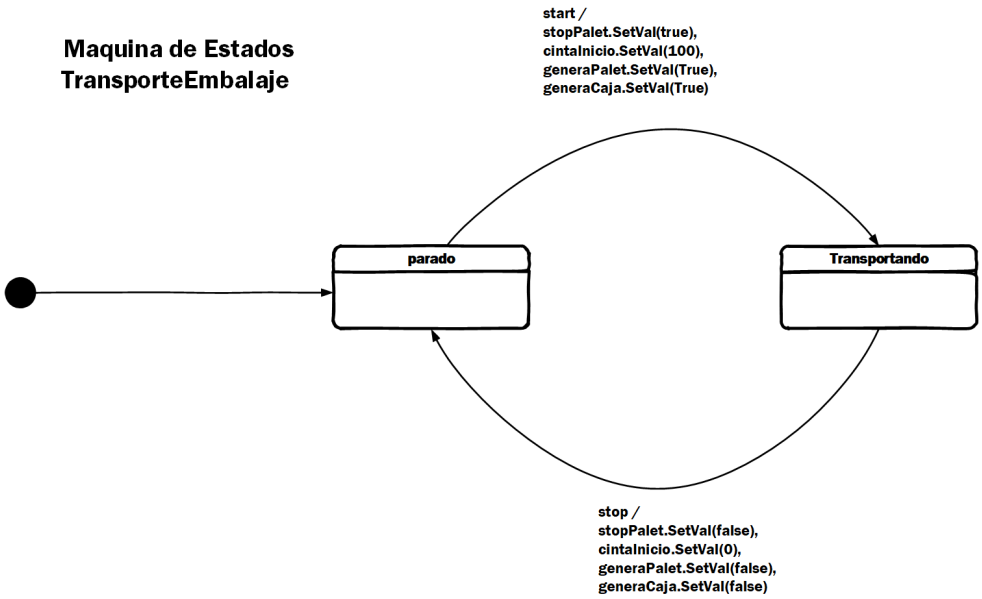


Figura 9.19: Máquina de estados para el subsistema TransporteEmbalaje.

La máquina de estados creada para este subsistema esta formado también por dos estados, uno que representa que el sistema esta parado (*Parado*) y otro que esta en marcha (*Transportando*). Las transiciones entre estos estados están asociadas a los métodos start y stop del elemento TransporteEmbalaje, los cuales desencadenan el arranque y paro de los elementos activos del modelo *cintaInicio*, *stopPalet*, *generaPalet* y *generaCaja*.

MODELO PARA EL SUBSISTEMA RECOGEPIEZA

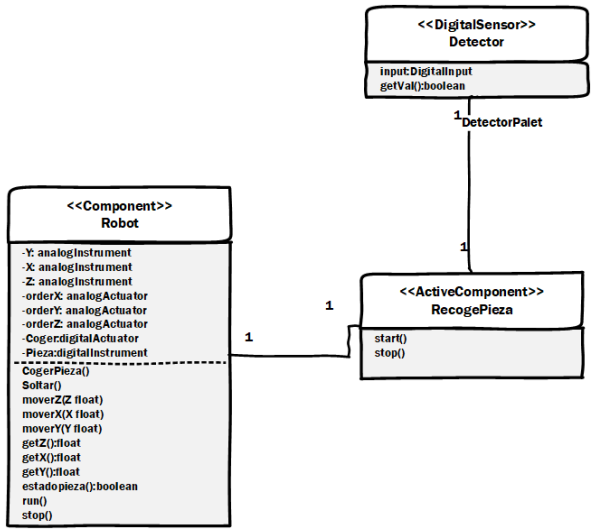


Figura 9.20: Modelo con MOSIE realizado para el subsistema RecogePieza.

El modelo creado posee solo tres componentes, un detector que extiende del componente *DigitalSensor*, un componente para encapsular el robot y un componente activo para gestionar el subsistema *RecogePieza*.

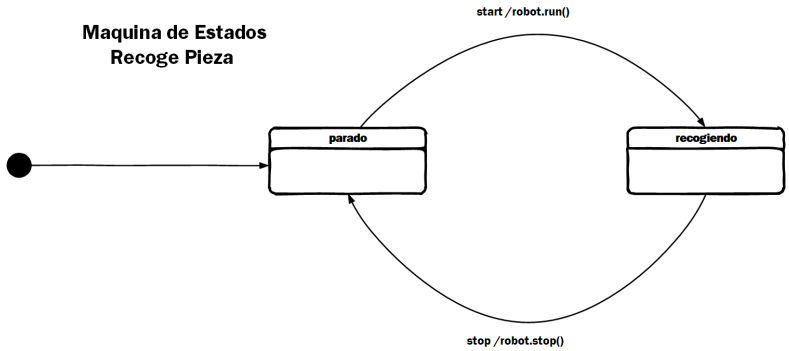


Figura 9.21: Máquina de estados para el subsistema TransporteEmbalaje.

La máquina de estados creada para este subsistema esta formado también por dos estados, uno que representa que el sistema esta parado (*parado*) y otro que esta en marcha (*Recogiendo*). Las transiciones entre estos estados están asociadas a los métodos start y stop del elemento *RecogePieza*. Como este subsistema solo

posee un concepto activo, el robot, los métodos del elemento *RecogePieza* tan solo arrancan o paran el robot que recoge las piezas.

MODELO PARA EL SUB-SISTEMA SISTEMAENCAJONADO

Este subsistema esta formado por los dos subsistemas anteriores *RecogePieza*, *EnviaPalet* y *TransporteEmbalaje*, de forma que los métodos del componente *SistemaEncajonados* controlan ambos subsistemas, en la figura 9.23 se puede ver dicho modelo.

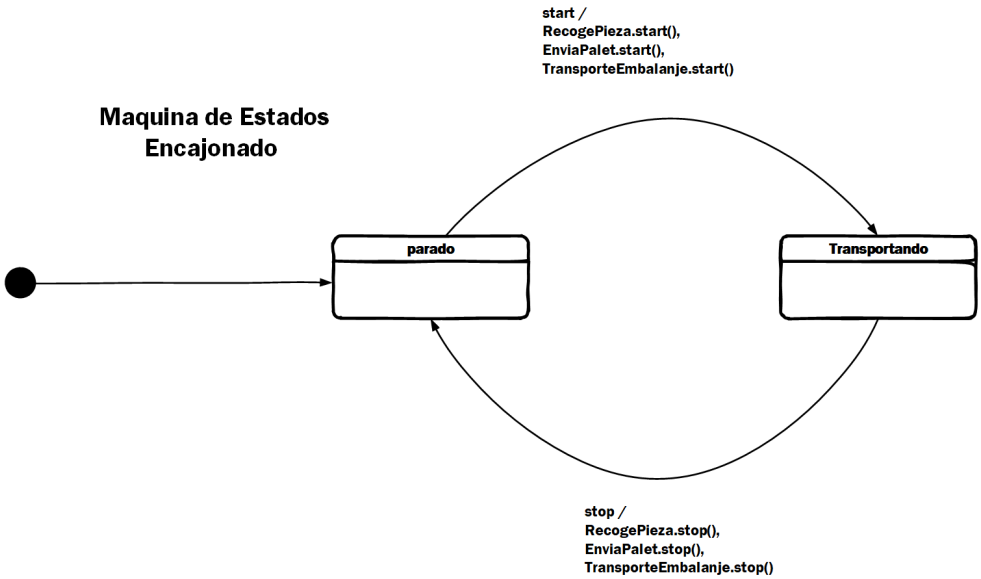


Figura 9.22: Máquina de estados para el subsistema SistemaEncajonado.

La máquina de estados creada para este subsistema esta formado también por dos estados, uno que representa que el sistema esta parado (*Parado*) y otro que esta en marcha (*Transportando*). Las transiciones entre estos estados estan asociadas a los métodos *start* y *stop*, los cuales desencadenan el arranque y paro de los subsistemas *SistemaEncajonado*.

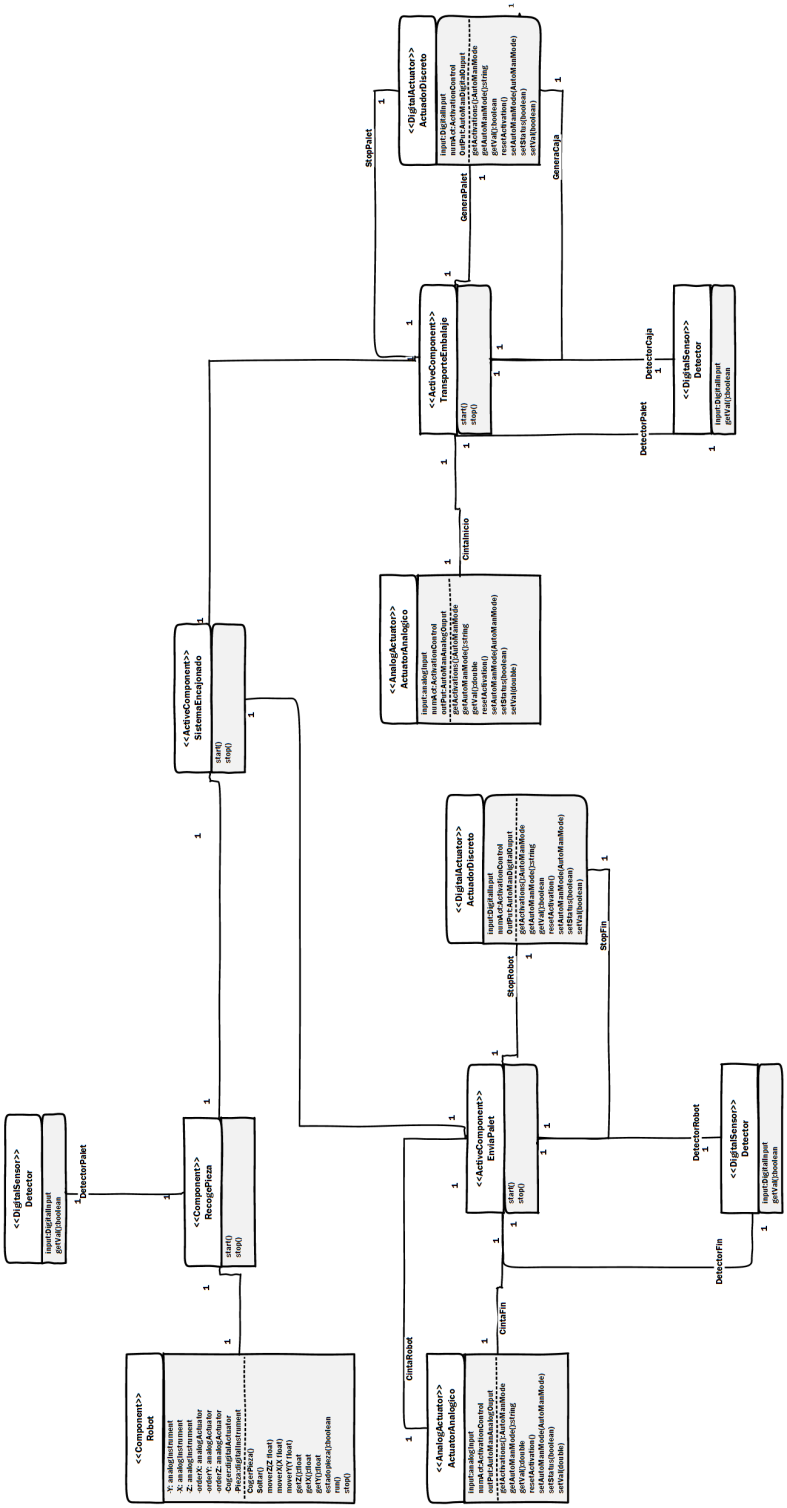


Figura 9.23: Modelo con MOSIE realizado para el subsistema de SistemaEncajonado.

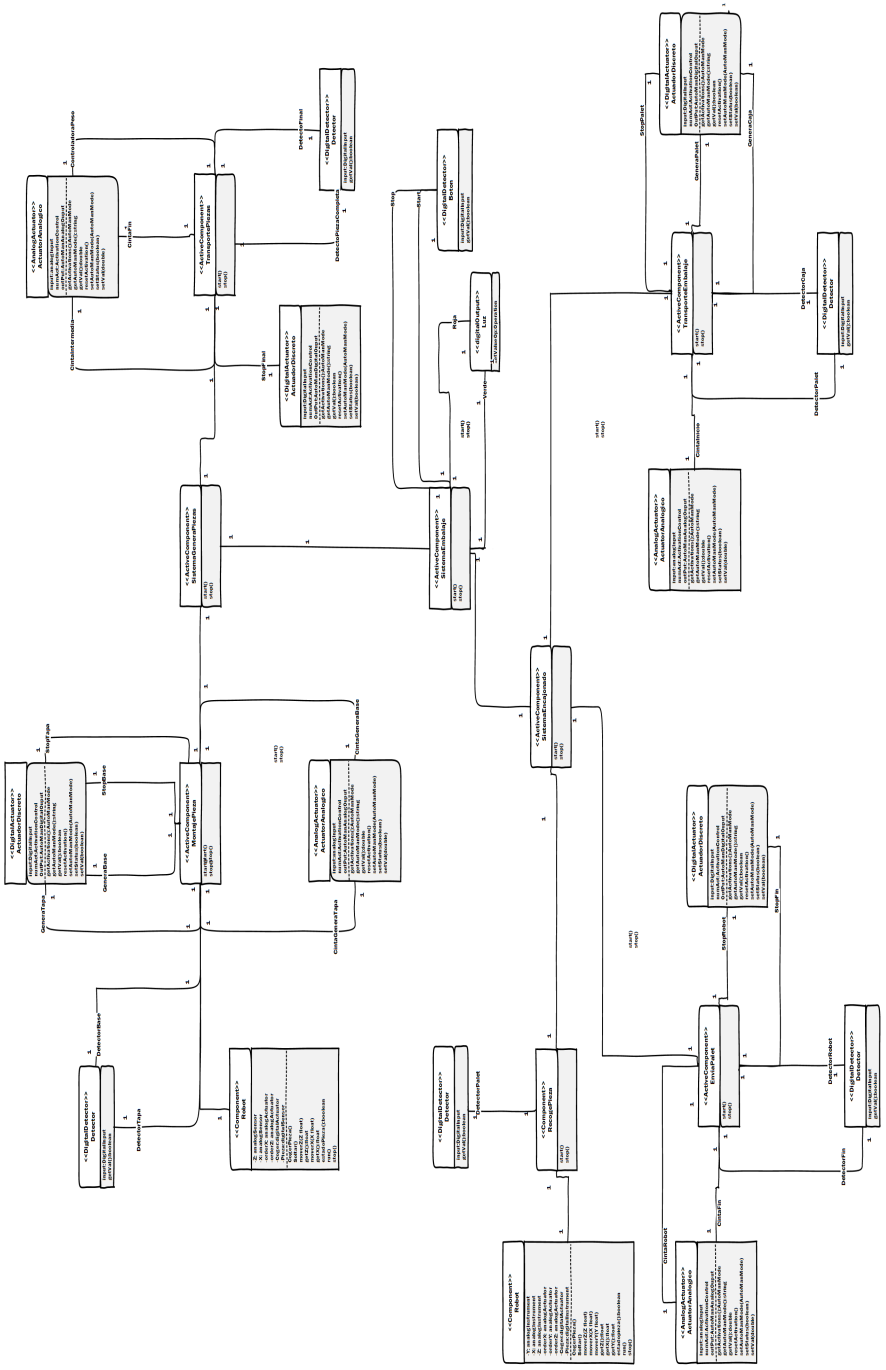


Figura 9.24: Modelo para el Sistema de Embalaje completo.

MODELO PARA TODO EL SISTEMA DE EMBALAJE

Hasta ahora hemos modelado todos los subsistemas, utilizando los elementos y relaciones recogidas en las tablas que se han construido antes de empezar el proceso de modelado. El último paso que nos queda es relacionar todos estos submodelos, creando así el modelo completo de todo el sistema industrial. El modelo completo obtenido se puede ver en la figura 9.24.

La parte nueva del modelo esta formada por el componente activo principal *SistemaEmbalaje* que se relaciona con los subsistemas *SistemaGeneraPiezas* y *SistemaEncajonado*. Además se ha creado dos elementos mas Luz y Botón utilizando los conceptos *DigitalOutput* y *DigitalDetector* de MOSIE. Con estos elementos recogemos las botoneras y luces que tiene el sistema industrial para que el operario pare o arranque toda la cadena de montaje. En la siguiente figura se puede ver con mas detalle estos elementos:

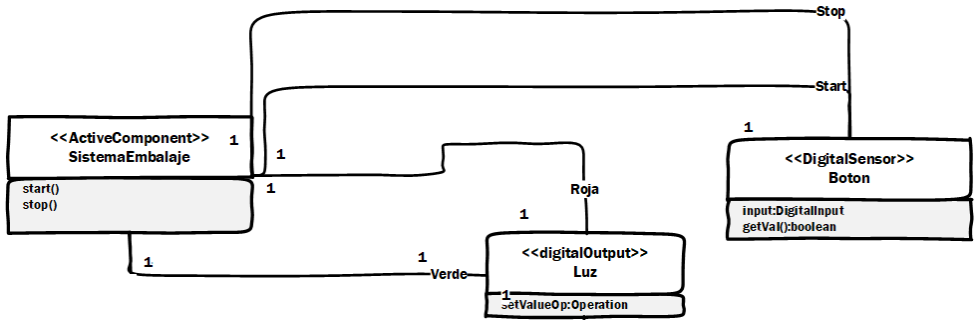


Figura 9.25: Detalle del Modelo para el Sistema de Embalaje completo.

Para el el elemento *SistemaEmbalaje* hemos creado la siguiente maquina de estados que recoge todo el comportamiento del sistema industrial:

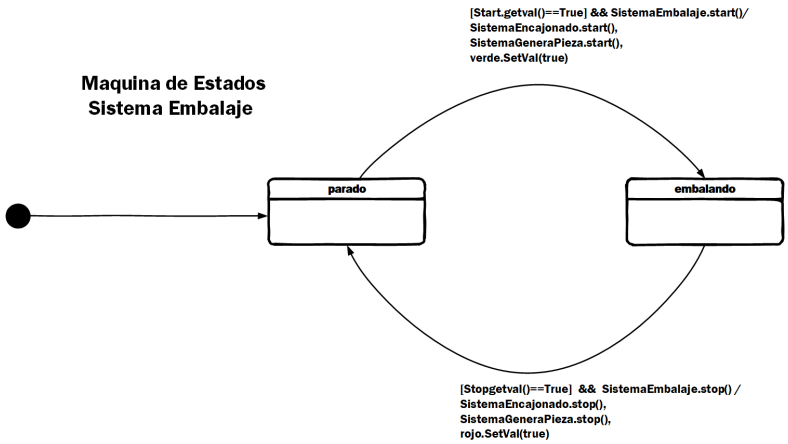


Figura 9.26: Máquina de estados para el sistema de Embalaje.

La máquina de estados que se ha creado, figura 9.26 tiene dos estados, uno que representa que el sistema esta parado (Parado) y otro que está en marcha (*Embalando*). Las transiciones entre estos estados están asociadas a la pulsación del botón *Start* y a que se active el método Start del componente *SistemaEmbalaje* para la transición de parado a embalando. Además en esta transición se activan los métodos *Start* de los subsistemas Encajonado y *GeneraPieza*, así como el luminoso que indica que el sistema esta funcionando. La transición de embalando a parado se activa con el botón *Stop* y el método *Stop*, también del componente *SistemaEmbalaje*. En esta transición se activan los métodos *Stop* de los subsistemas *Encajonado* y *GeneraPieza*, así como el luminoso rojo que indica que el sistema esta parado.

No se ha realizado ningún modelado para el caso en el que el sistema se encuentre en parada de emergencia, ya que las seguridades del sistema se realizaran utilizando relés de seguridad eléctricos que no estarán controlados por el PLC, ni supervisados por el servidor OPC UA.

9.8. DESPLIEGUE DEL MODELO DEL SISTEMA EN UN PLC SIEMENS

Una vez que hemos obtenido todo el modelo del sistema de embalaje, el primer paso para poder aplicarlo a un sistema industrial real es desplegar este modelo en un dispositivo industrial. En nuestro caso en un PLC de Siemens de la familia S7 1500. En concreto para este proyecto hemos escogido el modelo 1515-2

PN. Para poder realizar este despliegue utilizaremos el entorno de desarrollo TIA Portal version 13 [143].

9.8.1. TRANSFORMACIÓN DEL MODELO PIM.

Para transformar los conceptos del modelo PIM creado, primero tenemos que desplegar en TIA Portal los componentes de *ActuadorDiscreto*, *ActuadorAnalógico*, *Detector*, *Luz* y *Boton*. Como todos ellos extiende de un elemento sintáctico ya existente en MOSIE se han creado los siguientes FBs:

9.8.2. ELEMENTOS COMUNES

ActuadorDiscreto:

```
FUNCTION_BLOCK "ActuadorDiscreto"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_resetActivation : Bool;
 i_mode : "AutoManMode";
 i_status : Bool;
 i_manValue : Bool;
 i_value : Bool;
 i_numAct : Int;
 i_alarmCond : "DigitalOperator";
 i_statusSim : Bool;
END_VAR
VAR_OUTPUT
 o_numAct : Real;
 o_mode : "AutoManMode";
 o_status : Bool;
 o_statusSim : Bool;
 o_isAlarm : Bool;
 o_value : Bool;
END_VAR
VAR
 Actuador : "DigitalActuatorFB";
END_VAR
BEGIN
NETWORK
TITLE =
 CALL #Actuador
 (  i_resetActivation           := #i_resetActivation ,
 i_mode                      := #i_mode ,
 i_status                    := #i_status ,
 i_manValue                  := #i_manValue ,
 i_value                     := #i_value ,
 i_numAct                    := #i_numAct ,
 i_alarmCond                 := #i_alarmCond ,
 i_statusSim                 := #i_statusSim ,
 o_numAct                    := #o_numAct ,
 o_mode                      := #o_mode ,
 o_status                    := #o_status ,
 o_statusSim                 := #o_statusSim ,
 o_isAlarm                   := #o_isAlarm ,
 o_value                     := #o_value
```

```
 );

END_FUNCTION_BLOCK
```

ActuadorAnalogico:

```
FUNCTION_BLOCK "ActuadorAnalogicoFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_resetActivation : Bool;
 i_mode : "AutoManMode";
 i_status : Bool;
 i_manValue : Real;
 i_value : Real;
 i_numAct : Int;
 i_alarmCond : "AnalogOperator";
 i_statusSim : Bool;
END_VAR
VAR_OUTPUT
 o_numAct : Int;
 o_mode : "AutoManMode";
 o_status : Bool;
 o_statusSim : Bool;
 o_isAlarm : Bool;
 o_value : Real;
END_VAR
VAR
 Actuador : "AnalogActuatorFB";
END_VAR
BEGIN
NETWORK
TITLE =
 CALL #Actuador
 (  i_resetActivation           := #i_resetActivation ,
 i_mode                      := #i_mode ,
 i_status                    := #i_status ,
 i_manValue                  := #i_manValue ,
 i_value                     := #i_value ,
 i_numAct                    := #i_numAct ,
 i_alarmCond                 := #i_alarmCond ,
 i_statusSim                 := #i_statusSim ,
 o_numAct                    := #o_numAct ,
 o_mode                      := #o_mode ,
 o_status                    := #o_status ,
 o_statusSim                 := #o_statusSim ,
 o_isAlarm                   := #o_isAlarm ,
 o_value                     := #o_value
 );

END_FUNCTION_BLOCK
```

Detector:

```
FUNCTION_BLOCK "Detector"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_statusSim : Bool;
 i_status : Bool;
 i_cond : "DigitalOperator";
END_VAR
```

```
VAR_OUTPUT
 o_value : Bool;
 o_isAlarm : Bool;
 o_status : Bool;
 o_statusSim : Bool;
END_VAR
VAR
 Detector : "DigitalSensorFB";
END_VAR
BEGIN
NETWORK
TITLE =
 CALL #Detector
 (  i_statusSim                := #i_statusSim ,
 i_status                   := #i_status ,
 i_cond                     := #i_cond ,
 o_value                    := #o_value ,
 o_isAlarm                  := #o_isAlarm ,
 o_status                   := #o_status ,
 o_statusSim                := #o_statusSim
 );

END_FUNCTION_BLOCK
```

Luz:

```
FUNCTION_BLOCK "Detector"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_statusSim : Bool;
 i_status : Bool;
 i_cond : "DigitalOperator";
END_VAR
VAR_OUTPUT
 o_value : Bool;
 o_isAlarm : Bool;
 o_status : Bool;
 o_statusSim : Bool;
END_VAR
VAR
 Detector : "DigitalSensorFB";
END_VAR
BEGIN
NETWORK
TITLE =
 CALL #Detector
 (  i_statusSim                := #i_statusSim ,
 i_status                   := #i_status ,
 i_cond                     := #i_cond ,
 o_value                    := #o_value ,
 o_isAlarm                  := #o_isAlarm ,
 o_status                   := #o_status ,
 o_statusSim                := #o_statusSim
 );
END_FUNCTION_BLOCK
```

Boton:

```
FUNCTION_BLOCK "Boton"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
```

```
VAR_INPUT
 i_status : Bool;
 i_statusSim : Bool;
END_VAR
VAR_OUTPUT
 o_status : Bool;
 o_statusSim : Bool;
 o_value : Bool;
END_VAR
VAR
 boton : "digitalInputFB";
END_VAR


BEGIN
NETWORK
 TITLE =
 CALL #boton
 (  i_status                    := #i_status ,
 i_statusSim                 := #i_statusSim ,
 o_status                    := #o_status ,
 o_statusSim                 := #o_statusSim ,
 o_value                     := #o_value
 );
END_FUNCTION_BLOCK
```

Con estos FBs que hemos creado ya podemos crear el resto de los elementos contemplados por el modelo.

9.8.3. SUBSISTEMA MONTAJEPIEZAS.

Para este subsistema se han creado dos FBs, uno que recoge todo los elementos que forma el subsistema y otro para correlacionar el robot encargado de montar las piezas. Obviamente dentro del FB que recoge todos los conceptos del sistema existe una variable de tipo *RobotMontajeFB*.

RobotMontaje:

```
FUNCTION_BLOCK "RobotMontajeFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR

VAR_OUTPUT
 Arrancado : Bool;
 Parado : Bool;
END_VAR

VAR
 Z : "AnalogSensorFB";
 X : "AnalogSensorFB";
 OrdenX : "AnalogActuatorFB";
 OrdenY : "AnalogActuatorFB";
 Coger : "DigitalActuatorFB";
```

```
 Soltar : "DigitalActuatorFB";
END_VAR
BEGIN
NETWORK
TITLE =

END_FUNCTION_BLOCK
```

MontajePiezas:

```
FUNCTION_BLOCK "MontajePiezaFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR
VAR
 GeneraBase : "ActuadorDiscreto";
 GeneraTapa : "ActuadorDiscreto";
 StopBase : "ActuadorDiscreto";
 StopTapa : "ActuadorDiscreto";
 DetectorBase : "Detector";
 DetectorTapa : "Detector";
 CintaGeneraTapa : "ActuadorAnalogicoFB";
 CintaGeneraBase : "ActuadorAnalogicoFB";
 Robot : "RobotMontajeFB";
END_VAR
BEGIN
NETWORK
TITLE =

END_FUNCTION_BLOCK

FUNCTION_BLOCK "MontajePiezasFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR
VAR
 CintaFin  : "ActuadorAnalogicoFB";
 CintaIntermedia : "ActuadorAnalogicoFB";
 ControladoraPeso : "ActuadorAnalogicoFB";
 StopFina : "ActuadorDiscreto";
 DetectorPiezaCompleta : "Detector";
 DetectorFin : "Detector";
END_VAR
BEGIN
NETWORK
TITLE =

END_FUNCTION_BLOCK
```

9.8.4. SUBSISTEMA TRANSPORTEPIEZAS

Este sistema ha sido transformado mediante la creación de un único FB que recoge todos los conceptos del sistema:

```
FUNCTION_BLOCK "TransportePiezasFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR
VAR
 CintaFin : "ActuadorAnalogicoFB";
 CintaIntermedia : "ActuadorAnalogicoFB";
 ControladoraPeso : "ActuadorAnalogicoFB";
 StopFina : "ActuadorDiscreto";
 DetectorPiezaCompleta : "Detector";
 DetectorFin : "Detector";
END_VAR
BEGIN
NETWORK
TITLE =

END_FUNCTION_BLOCK
```

9.8.5. SUBSISTEMA GENERAPIEZAS

Este sistema recoge los dos subsistema que transportan y montan la pieza. Se ha creado un FB SistemaGeneraPiezaFB, con dos variables de tipo FB, una para MontajePiezaFB y otra para TransportePiezaFB:

```
FUNCTION_BLOCK "SistemaGeneraPiezaFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR

VAR
 MontajePieza : "MontajePiezaFB";
 TransportePieza : "TransportePiezasFB";
END_VAR
BEGIN
NETWORK
TITLE =
END_FUNCTION_BLOCK
```

Este FB seria el encargado de controlar todo el proceso de montaje y transporte de piezas.

9.8.6. SUBSISTEMA ENVIAPALET

Todos los conceptos de este subsistema están gestionados por el siguiente FB:

```
FUNCTION_BLOCK "SistemaEnvioPaletFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
```

```
VAR_INPUT
i_start : Bool;
i_stop : Bool;
END_VAR
VAR
 CintaRobot : "ActuadorAnalogicoFB";
 CintaFin : "ActuadorAnalogicoFB";
 DetectorFin : "Detector";
 DetectorRobot : "Detector";
 StopRobot : "ActuadorDiscreto";
 StopFin : "ActuadorDiscreto";
END_VAR
BEGIN
NETWORK
TITLE =
END_FUNCTION_BLOCK
```

Como se puede ver todas las variables de este FB representan los conceptos del sistema, siendo estas variables de los tipos de FBs definidos previamente.

9.8.7. SUBSISTEMA RECOGEPIEZA

Para este sub-sistema se han creado dos FBs, uno que controla el robot que recoge las piezas RobotRecogePiezaFB y otro para el sistema en si SistemaRecogePiezaFB:

RobotRecogePieza:

```
FUNCTION_BLOCK "RobotRecogPiezaFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stio : Bool;
END_VAR

VAR
 y : "AnalogSensorFB";
 X : "AnalogSensorFB";
 z : "AnalogSensorFB";
 OrderX : "AnalogActuatorFB";
 OrderY : "AnalogActuatorFB";
 OrderZ : "AnalogActuatorFB";
 Coger : "DigitalActuatorFB";
 Pieza : "DigitalSensorFB";
END_VAR
BEGIN
NETWORK
TITLE =
END_FUNCTION_BLOCK
```

SistemaRecogePieza:

```
FUNCTION_BLOCK "SistemaRecogePiezaFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
```

```
 i_start : Bool;
 i_stop : Bool;
END_VAR
VAR
 Robot : "RobotRecogPiezaFB";
 DetectorPalet : "Detector";
END_VAR
BEGIN
NETWORK
TITLE =
END_FUNCTION_BLOCK
```

9.8.8. SUBSISTEMA TRANSPORTEEMBALAJE

Para este subsistema se ha creado un solo FB, en el que se recogen todos los conceptos que forman el subsistema utilizando variables que son de los tipos de FBs definidos para los elementos comunes:

```
FUNCTION_BLOCK "SistemaTransporteEmbalajeFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR
VAR
 StopPalet : "DigitalActuatorFB";
 GeneraPalet : "DigitalActuatorFB";
 GeneraCaja : "DigitalActuatorFB";
 DetectorPalet : "Detector";
 DetectorCaja : "Detector";
 CintaInicio : "ActuadorAnalogicoFB";
END_VAR

BEGIN
NETWORK
TITLE =
END_FUNCTION_BLOCK
```

9.8.9. SUBSISTEMA ENCAJONADO

Este subsistema se encarga de controlar los subsistemas envioPalet, recogePieza y TransporteEmbalaje. Se ha creado el siguiente FB, con variables de tipo FB *SistemaEnvioPaletFB*, *SistemaRecogePiezaFB* y *SistemaTransporteEmbalajeFB*, para representar ha este subsistema.

```
FUNCTION_BLOCK "SistemaEncajonadoFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
 i_start : Bool;
 i_stop : Bool;
END_VAR
```

```
VAR
 SistemaRecogePieza : "SistemaRecogePiezaFB";
 SistemaEnvioPalet : "SistemaEnvioPaletFB";
 SistemaTransporteEmbalaje : "SistemaTransporteEmbalajeFB";
END_VAR

BEGIN
NETWORK
TITLE =
END_FUNCTION_BLOCK
```

9.8.10. SISTEMA EMBALAJE

Por último nos queda desplegar el elemento que representa el sistema de Embalaje. La primera opción elegida para correlacionar este elemento fue la de crear un FB con variables cuyo tipo fuese de los diferentes subsistemas que forma el sistema de Embalaje, así como las luces y los botones que recoge el modelo de MOSIE. Pero al compilar este FB nos hemos encontrado con otra limitación de TIA Portal, el número de llamadas anidadas a FBs que se pueden realizar está limitado, produciendo un error de compilación y por lo tanto no pudiendo desplegar el modelo completamente.

Como solución a este problema hemos creado un OB cíclico, denominado SistemaEmbalaje en el que gestionaremos todos los elementos del sistema. Este OB tiene la siguiente estructura:

```
ORGANIZATION_BLOCK "SistemaEmbalaje"
TITLE = "Main Program Sweep (Cycle)"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
BEGIN
NETWORK
TITLE =
 CALL "Luz", "Roja";
 CALL "Luz", "Verde";
 CALL "Boton", "Start";
 CALL "Boton", "Stop";

 CALL "SistemaEncajonadoFB", "SistemaEncajonadoFB_DB";
 CALL "SistemaGeneraPiezaFB", "SistemaGeneraPiezaFB_DB";

END_ORGANIZATION_BLOCK
```

Debido al utilización de un OB hemos tenido que crear los siguientes DBs asociados al las llamadas a los diferentes FBs:

```
DATA_BLOCK "Start"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
NON_RETAIN
```

```
"BotonFB"
BEGIN

END_DATA_BLOCK

DATA_BLOCK "Stop"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
NON_RETAIN
"BotonFB"
BEGIN

END_DATA_BLOCK

DATA_BLOCK "Verde"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
NON_RETAIN
"LuzFB"
BEGIN

END_DATA_BLOCK

DATA_BLOCK "Roja"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
NON_RETAIN
"LuzFB"
BEGIN

END_DATA_BLOCK

DATA_BLOCK "SistemaEncajonadoFB_DB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
NON_RETAIN
"SistemaEncajonadoFB"
BEGIN

END_DATA_BLOCK

DATA_BLOCK "SistemaGeneraPiezaFB_DB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
NON_RETAIN
"SistemaGeneraPiezaFB"
BEGIN

END_DATA_BLOCK
```

Una vez realizada la transformación del modelo, pasaríamos a crear el código conforme a las especificaciones funcionales del sistema y el comportamiento recogido en las diferentes máquinas de estados del modelo de MOSIE, para ello tendríamos que implementar las transiciones, estados y acciones de cada máquina de estados en el FB correspondiente.

9.9. DESPLIEGUE DEL MODELO DEL SISTEMA EN UN OPC UA

El modelo del sistema que hemos desarrollado puede ser también desplegado en un servidor OPC UA mediante la aplicación de las reglas de transformación sobre el modelo PIM. En nuestro caso vamos a utilizar nuestro propio servidor OPC UA desarrollado en .NET.

Para poder conectar nuestro servidor OPC UA con el PLC que controla todo el sistema industrial se ha desarrollado un driver de comunicaciones utilizando la librería **s7netplus** [144]. Dicho driver utiliza un fichero de configuración en XML, en el que se emparejan los identificadores de los objetos dentro del espacio de direcciones del servidor con las direcciones de memoria de los mismos conceptos en el PLC de siemens. Gracias a este fichero de configuración y al driver desarrollado podemos comunicar nuestro modelo MOSIE alojado en un servidor de OPC UA con nuestro modelo MOSIE alojado en nuestro PLC Siemens.

9.9.1. TRANSFORMACIÓN DEL MODELO PIM A PSM.

Para la transformación del modelo PIM al modelo PSM en OPC UA se aplicaron las reglas de transformación especificada en el capítulo 6. Posteriormente se ha modelado el PSM en OPC UA utilizando el UA Model Designer, un ejemplo de este modelo realizado con esta herramienta se puede ver en la figura 9.27, dicho modelo se ha compiladado generando el fichero XML que contiene la definición de los diferentes sistemas, sub-sistemas y elementos.

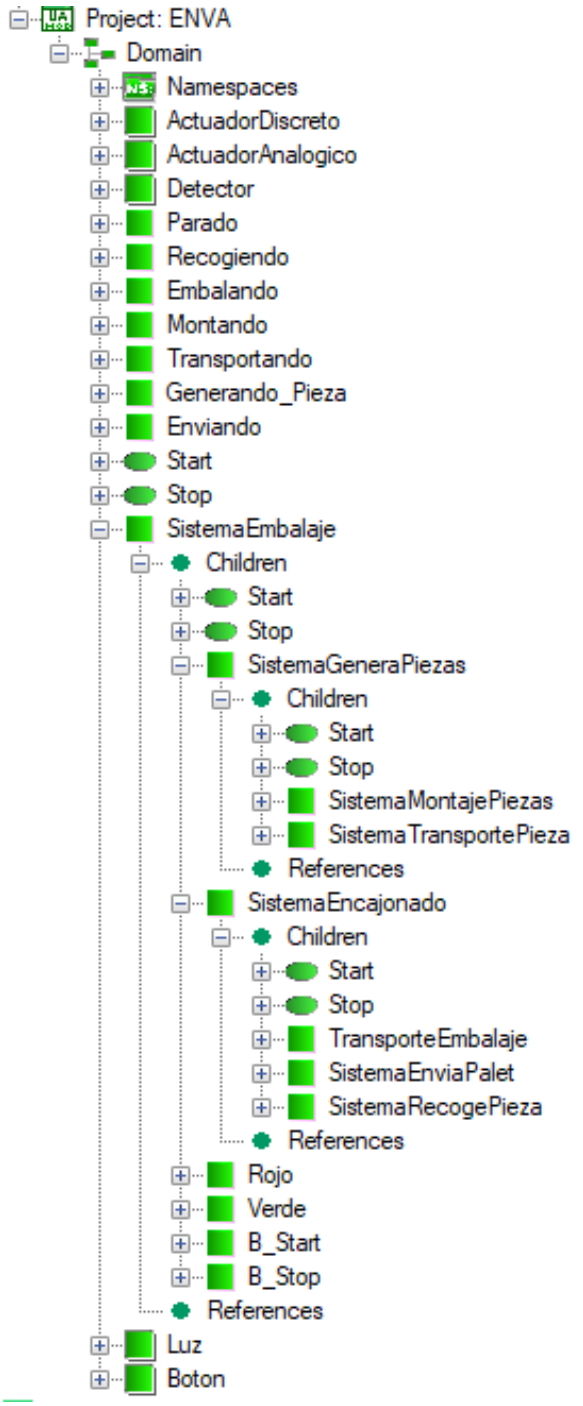


Figura 9.27: Modelo realizado para el sistema de Embalaje con UA Model Designer.

9.9.2. ELEMENTOS COMUNES

ActuadorDiscreto:

```
<ObjectType SymbolicName="ENVA:ActuadorDiscreto" BaseType="MOSIE:DigitalActuator"/>
```

ActuadorAnalogico:

```
][basicstyle=\fontsize{7}{7}\ttfamily]

<ObjectType SymbolicName="ENVA:ActuadorAnalogico" BaseType="MOSIE:AnalogActuator"/>
```

Detector:

```
][basicstyle=\fontsize{7}{7}\ttfamily]

<ObjectType SymbolicName="ENVA:Detector" BaseType="MOSIE:DigitalSensor" />
```

Luz:

```
][basicstyle=\fontsize{7}{7}\ttfamily]]

<ObjectType SymbolicName="ENVA:Luz" BaseType="MOSIE:DigitalOutput" />
```

Boton:

```
][basicstyle=\fontsize{7}{7}\ttfamily]]

<ObjectType SymbolicName="ENVA:Boton" BaseType="MOSIE:DigitalInput" />
```

9.9.3. SUB-SISTEMA MONTAJEPIEZAS.

```
<Object SymbolicName="ENVA:SistemaMontajePiezas" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:GeneraBase" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:GeneraTapa" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:StopBase" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:StopTapa" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:CintaGeneraBase" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:CintaGeneraTapa" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:DetectorBase" TypeDefinition="ENVA:Detector" />
 <Object SymbolicName="ENVA:DetectorTapa" TypeDefinition="ENVA:Detector" />
 <Object SymbolicName="ENVA:Robot" TypeDefinition="iMMAS:Component">
 <Children>
  <Object SymbolicName="ENVA:Z" TypeDefinition="MOSIE:AnalogSensor" />
  <Object SymbolicName="ENVA:X" TypeDefinition="MOSIE:AnalogSensor" />
  <Object SymbolicName="ENVA:orderZ" TypeDefinition="MOSIE:AnalogActuator" />
  <Object SymbolicName="ENVA:orderX" TypeDefinition="MOSIE:AnalogActuator" />
  <Object SymbolicName="ENVA:D_Coger" TypeDefinition="MOSIE:DigitalActuator" />
  <Object SymbolicName="ENVA:D_Pieza" TypeDefinition="MOSIE:DigitalSensor" />
  <Method SymbolicName="ENVA:moverZ" TypeDefinition="iMMAS:iMMASMethod">
   <InputArguments>
   <Argument Name="Z" DataType="OpcUa:Float" />
   </InputArguments>
  </Method>
  <Method SymbolicName="ENVA:getZ" TypeDefinition="iMMAS:iMMASMethod">
```

```
   <OutputArguments>
   <Argument Name="Z" DataType="OpcUa:Float" />
   </OutputArguments>
  </Method>
  <Method SymbolicName="ENVA:moverX" TypeDefinition="iMMAS:iMMASMethod">
   <InputArguments>
   <Argument Name="X" DataType="OpcUa:Float" />
   </InputArguments>
  </Method>
  <Method SymbolicName="ENVA:getX" TypeDefinition="iMMAS:iMMASMethod">
   <OutputArguments>
   <Argument Name="X" DataType="OpcUa:Float" />
   </OutputArguments>
  </Method>
  <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:Coger" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:Soltar" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:EstadoPieza" TypeDefinition="iMMAS:iMMASMethod">
   <OutputArguments>
   <Argument Name="pieza" DataType="OpcUa:Boolean" />
   </OutputArguments>
  </Method>
 </Children>
 </Object>
</Children>
</Object>
```

9.9.4. SUB-SISTEMA TRANSPORTEPIEZAS.

```
<Object SymbolicName="ENVA:SistemaTransportePieza" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:CintaIntermedia" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:CintaFin" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:ControladoraPeso" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:StopFinal" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:DetectorPiezaCompleta" TypeDefinition="ENVA:Detector" />
 <Object SymbolicName="ENVA:DetectorFinal" TypeDefinition="ENVA:Detector" />
</Children>
</Object>
```

9.9.5. SUB-SISTEMA GENERAPIEZAS.

```
<Object SymbolicName="ENVA:SistemaGeneraPiezas" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:MontajePiezas" TypeDefinition="iMMAS:ActiveComponent">
  ....
 </Object>
 <Object SymbolicName="ENVA:TransportePieza" TypeDefinition="iMMAS:ActiveComponent">
 ....
 </Object>
</Children>
</Object>
```

9.9.6. SUB-SISTEMA ENVIAPALET.

```
<Object SymbolicName="ENVA:SistemaEnviaPalet" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
```

```
 <Object SymbolicName="ENVA:CintaRobot" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:CintaFin" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:StopRobot" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:StopFin" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:DetectorRobot" TypeDefinition="ENVA:Detector" />
 <Object SymbolicName="ENVA:DetectorFin" TypeDefinition="ENVA:Detector" />
</Children>
</Object>
```

9.9.7. SUB-SISTEMA RECOGEPIEZA.

```
<Object SymbolicName="ENVA:SistemaRecogePieza" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:DetectorPalet" TypeDefinition="ENVA:Detector" />
 <Object SymbolicName="ENVA:Robot" TypeDefinition="iMMAS:Component">
 <Children>
  <Object SymbolicName="ENVA:Z" TypeDefinition="MOSIE:AnalogSensor" />
  <Object SymbolicName="ENVA:X" TypeDefinition="MOSIE:AnalogSensor" />
  <Object SymbolicName="ENVA:Y" TypeDefinition="MOSIE:AnalogSensor" />
  <Object SymbolicName="ENVA:orderZ" TypeDefinition="MOSIE:AnalogActuator" />
  <Object SymbolicName="ENVA:orderX" TypeDefinition="MOSIE:AnalogActuator" />
  <Object SymbolicName="ENVA:orderY" TypeDefinition="MOSIE:AnalogActuator" />
  <Object SymbolicName="ENVA:D_Coger" TypeDefinition="MOSIE:DigitalActuator" />
  <Object SymbolicName="ENVA:D_Pieza" TypeDefinition="MOSIE:DigitalSensor" />
  <Method SymbolicName="ENVA:moverZ" TypeDefinition="iMMAS:iMMASMethod">
   <InputArguments>
   <Argument Name="Z" DataType="OpcUa:Float" />
   </InputArguments>
  </Method>
  <Method SymbolicName="ENVA:getZ" TypeDefinition="iMMAS:iMMASMethod">
   <OutputArguments>
   <Argument Name="Z" DataType="OpcUa:Float" />
   </OutputArguments>
  </Method>
  <Method SymbolicName="ENVA:moverX" TypeDefinition="iMMAS:iMMASMethod">
   <InputArguments>
   <Argument Name="X" DataType="OpcUa:Float" />
   </InputArguments>
  </Method>
  <Method SymbolicName="ENVA:getX" TypeDefinition="iMMAS:iMMASMethod">
   <OutputArguments>
   <Argument Name="X" DataType="OpcUa:Float" />
   </OutputArguments>
  </Method>
  <Method SymbolicName="ENVA:moverY" TypeDefinition="iMMAS:iMMASMethod">
   <InputArguments>
   <Argument Name="Y" DataType="OpcUa:Float" />
   </InputArguments>
  </Method>
  <Method SymbolicName="ENVA:getY" TypeDefinition="iMMAS:iMMASMethod">
   <OutputArguments>
   <Argument Name="Y" DataType="OpcUa:Float" />
   </OutputArguments>
  </Method>
  <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:Coger" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:Soltar" TypeDefinition="iMMAS:iMMASMethod" />
  <Method SymbolicName="ENVA:EstadoPieza" TypeDefinition="iMMAS:iMMASMethod">
   <OutputArguments>
   <Argument Name="pieza" DataType="OpcUa:Boolean" />
   </OutputArguments>
  </Method>
 </Children>
 </Object>
</Children>
</Object>
```

9.9.8. SUB-SISTEMA TRANSPORTEEMBALAJE.

```
<Object SymbolicName="ENVA:SistemaTransporteEmbalaje" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:CintaInicio" TypeDefinition="ENVA:ActuadorAnalogico" />
 <Object SymbolicName="ENVA:GeneraCaja" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:GeneraPalet" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:StopPalet" TypeDefinition="ENVA:ActuadorDiscreto" />
 <Object SymbolicName="ENVA:DetectorPalet" TypeDefinition="ENVA:Detector" />
 <Object SymbolicName="ENVA:DetectorCaja" TypeDefinition="ENVA:Detector" />
</Children>
</Object>
```

9.9.9. SUB-SISTEMA ENCAJONADO.

```
<Object SymbolicName="ENVA:SistemaEncajonado" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:EnviaPalet" TypeDefinition="iMMAS:ActiveComponent">
 ....
 </Object>
 <Object SymbolicName="ENVA:RecogePieza" TypeDefinition="iMMAS:ActiveComponent">
 ....
 </Object>
 <Object SymbolicName="ENVA:TransporteEmbalaje" TypeDefinition="iMMAS:ActiveComponent">
 ....
 </Object>
</Children>
</Object>
```

9.9.10. SUB-SISTEMA EMBALAJE.

```
<Object SymbolicName="ENVA:SistemaEmbalaje" TypeDefinition="iMMAS:ActiveComponent">
<Children>
 <Method SymbolicName="ENVA:Start" TypeDefinition="iMMAS:iMMASMethod" />
 <Method SymbolicName="ENVA:Stop" TypeDefinition="iMMAS:iMMASMethod" />
 <Object SymbolicName="ENVA:SistemaGeneraPieza" TypeDefinition="iMMAS:ActiveComponent">
 ....
 </Object>
 <Object SymbolicName="ENVA:SistemaEncajonado" TypeDefinition="iMMAS:ActiveComponent">
 ....
 </Object>
</Children>
</Object>
```

9.9.11. MAQUINAS DE ESTADOS.

Las transformaciones realizadas para las maquinas de estados definidas en el modelo se pueden ver a continuación:

Máquina de estados EnviaPalet.

```
<Object SymbolicName="iMMAS:MaquinaEnviaPalet" TypeDefinition="OpcUa:FiniteStateMachineType">
 <Children>
  <Object SymbolicName="ENVA:Enviando" TypeDefinition="OpcUa:TransitionType">
  <References>
```

```
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Recogiendo</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:HasCause</ReferenceType>
   <TargetId>ENVA:Start</TargetId>
   </Reference>
   </References>
   </Object>
   <Object SymbolicName="ENVA:Parado" TypeDefinition="OpcUa:TransitionType">
   <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Recogiendo</TargetId>
   </Reference>
  </References>
  </Object>
 </Children>
</Object>
```

Máquina de estados GeneraPieza.

```
<Object SymbolicName="iMMAS:MaquinaGeneraPieza" TypeDefinition="OpcUa:FiniteStateMachineType">
 <Children>
  <Object SymbolicName="ENVA:Enviando" TypeDefinition="OpcUa:TransitionType">
  <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Generando_Pieza</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:HasCause</ReferenceType>
    <TargetId>ENVA:Start</TargetId>
   </Reference>
   </References>
  </Object>
  <Object SymbolicName="ENVA:Parado" TypeDefinition="OpcUa:TransitionType">
   <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Recogiendo</TargetId>
   </Reference>
  </References>
  </Object>
 </Children>
</Object>
```

Máquina de estados Encajonado.

```
<Object SymbolicName="iMMAS:MaquinaEncajonando" TypeDefinition="OpcUa:FiniteStateMachineType">
 <Children>
  <Object SymbolicName="ENVA:Enviando" TypeDefinition="OpcUa:TransitionType">
```

```
  <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Transportando</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:HasCause</ReferenceType>
    <TargetId>ENVA:Start</TargetId>
   </Reference>
   </References>
  </Object>
  <Object SymbolicName="ENVA:Parado" TypeDefinition="OpcUa:TransitionType">
   <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Recogiendo</TargetId>
   </Reference>
   </References>
  </Object>
 </Children>
</Object>
```

Máquina de estados MontajePieza.

```
<Object SymbolicName="iMMAS:MaquinaMontajePieza" TypeDefinition="OpcUa:FiniteStateMachineType">
 <Children>
  <Object SymbolicName="ENVA:Enviando" TypeDefinition="OpcUa:TransitionType">
   <References>
    <Reference>
    <ReferenceType>OpcUa:ToState</ReferenceType>
     <TargetId>ENVA:Montando</TargetId>
    </Reference>
    <Reference>
    <ReferenceType>OpcUa:FromState</ReferenceType>
     <TargetId>ENVA:Parado</TargetId>
    </Reference>
    <Reference>
    <ReferenceType>OpcUa:HasCause</ReferenceType>
     <TargetId>ENVA:Start</TargetId>
    </Reference>
   </References>
  </Object>
  <Object SymbolicName="ENVA:Parado" TypeDefinition="OpcUa:TransitionType">
   <References>
    <Reference>
    <ReferenceType>OpcUa:ToState</ReferenceType>
     <TargetId>ENVA:Parado</TargetId>
    </Reference>
    <Reference>
    <ReferenceType>OpcUa:FromState</ReferenceType>
     <TargetId>ENVA:Recogiendo</TargetId>
    </Reference>
   </References>
  </Object>
 </Children>
</Object>
```

Máquina de estados TransporteEmbalaje.

```
<Object SymbolicName="iMMAS:MaquinaTransporteEmbalaje" TypeDefinition="OpcUa:
     FiniteStateMachineType">
```

```
 <Children>
  <Object SymbolicName="ENVA:Enviando" TypeDefinition="OpcUa:TransitionType">
  <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Transportando</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:HasCause</ReferenceType>
    <TargetId>ENVA:Start</TargetId>
   </Reference>
  </References>
  </Object>
 <Object SymbolicName="ENVA:Parado" TypeDefinition="OpcUa:TransitionType">
  <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Recogiendo</TargetId>
   </Reference>
  </References>
 </Object>
 </Children>
</Object>
```

Máquina de estados TransportePieza.

```
<Object SymbolicName="iMMAS:MaquinaTransportePieza" TypeDefinition="OpcUa:
     FiniteStateMachineType">
 <Children>
  <Object SymbolicName="ENVA:Enviando" TypeDefinition="OpcUa:TransitionType">
  <References>
   <Reference>
   <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Transportando</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:FromState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
   </Reference>
   <Reference>
   <ReferenceType>OpcUa:HasCause</ReferenceType>
    <TargetId>ENVA:Start</TargetId>
   </Reference>
   </References>
  </Object>
  <Object SymbolicName="ENVA:Parado" TypeDefinition="OpcUa:TransitionType">
   <References>
    <Reference>
     <ReferenceType>OpcUa:ToState</ReferenceType>
    <TargetId>ENVA:Parado</TargetId>
    </Reference>
    <Reference>
    <ReferenceType>OpcUa:FromState</ReferenceType>
     <TargetId>ENVA:Recogiendo</TargetId>
    </Reference>
   </References>
  </Object>
 </Children>
</Object>
```

9.9.12. COMUNICACIÓN CON EL PLC SIEMENS E INTERACIÓN CON UN CLIENTE UA

Una vez que hemos creado todos los ficheros XML necesarios para desplegar el modelo que hemos creado, necesitamos comunicar los conceptos de dicho modelo con los conceptos desplegados en el PLC de Siemens.

Para ello hemos desarrollado un driver de comunicaciones en C#, gracias a la librería s7netPlus [144]. Dicho driver se ha incluido dentro del servidor OPC UA que hemos desarrollado en .NET, el cual tenia la capacidad de cargar modelos de información complejos, tal y como se vio en el capítulo 6.

Para comunicar los objetos alojados en el servidor OPC UA con los objetos alojados en la memoria del PLC, en forma de DBs, el servidor carga un fichero XML en el que se relacionan el identificador OPC UA o ID de los atributos de los conceptos alojados en el espacio de direcciones del servidor OPC UA con las direcciones físicas, número de DB y posición dentro del DB del mismo atributo para el mismo concepto alojado en el PLC. Esta correlación se utilizará, por ejemplo, para leer o escribir los valores de los atributos cuando un cliente UA invoque los metodos get o set del objeto UA. Un ejemplo de la estructura de este fichero XML, se puede ver a continuación, para el elemento DetectorPalet del sub-sistema TransporteEmbalaje:

```
<DetectorPalet>
 <object ID="452" name="input.value.value" >
  <dataType>BOOL</dataType>
  <deviceDirection>DB1.DBX 18.0</deviceDirection>
 </object>

 <object ID="439" name="input.value.status" >
  <dataType>BOOL</dataType>
  <deviceDirection>DB1.DBX 18.1</deviceDirection>
 </object>

 <object ID="439" name="input.value.statuSim" >
  <dataType>BOOL</dataType>
  <deviceDirection>DB1.DBX 18.2</deviceDirection>
 </object>
</DetectorPalet>
```

Estas direcciones son pre-cargadas por el servidor OPC UA en el atributo Address de los IndustrialData, de forma que ya conocemos la dirección de memoria del PLC del atributo y por consiguiente podemos leer su valor o escribir un valor en dicha dirección, utilizando el driver desarrollado.

9.10. DISEÑO DE LOS PROTOCOLOS DE TESTEO Y VALIDACIÓN

9.10.1. PRUEBAS DE SEÑALES

Nombre Elemento	Descripción	Señal Eléctrica	Direccion E/S	Tipo	Verificado por	Fecha
Cinta1	Orden Marcha Cinta de transporte de generacion de Tapa	AO	AW4	Int		
Cinta2	Orden Marcha Cinta de transporte de generación de Base	AO	AW6	Int		
Cinta3	Orden Marcha Cinta de transporte intermedia hacia bascula	AO	AW6	Int		
Cinta4	Orden Marcha Cinta de transporte final sistema montaje	AO	AW8	Int		
Cinta_Bascula_F	Orden Marcha Cinta Bascula Hacia Adelante	DO	A0.4	Bool		
Cinta_Bascula_B	Orden Marcha Cinta Bascula Hacia Atrás	DO	A0.5	Bool		
Cinta_Stop1	Orden Levantamiento Barrera Stop Cinta Generación de Tapa	DO	A0.6	Bool		
Cinta_Stop2	Orden Levantamiento Barrera Stop Cinta Generación de Base	DO	A0.7	Bool		
Cinta_Stop3	Orden Levantamiento Barrera Final Sistema Ensamblaje	DO	A1.0	Bool		
Eliminar_Final_Linea	Orden de elimnar elementos de final de linea (Simulador)	DO	A1.1	Bool		
GeneraBase	Orden de generar una pieza base (Simulador)	DO	A1.2	Bool		
GeneradorCajas	Orden de generar una Caja (Simulador)	DO	A1.3	Bool		
GeneradordePallet	Orden de generar una Palet (Simulador)	DO	A1.4	Bool		
GeneradorTapa	Orden de generar una pieza tapa (Simulador)	DO	A1.5	Bool		
Resetlight	Iluminación botón de reset	DO	A1.6	Bool		
Robot1_Coger	Orden para coger pieza robot de ensamblaje	DO	A1.7	Bool		
Robot2_Coger	Orden para coger pieza robot de empaquetado	DO	A2.0	Bool		
Rodillo1	Orden de marcha rodillos generacion de palet	AO	AW10	Int		
Rodillo2	Orden de marcha rodillos intermedios	AO	AW12	Int		
Rodillo3	Orden de marcha rodillos robot encajonado	A0	AW14	Int		
Rodillo4	Orden de marcha rodillos fin de encajonado	AO	AW16	Int		
Rodillo_Stop1	Orden levantamiento Barrera Stop rodillos palet	DO	A2.5	Bool		
Rodillo_Stop2	Orden levantamiento Barrera Stop rodillos intermedios	DO	A2.6	Bool		
Rodillo_Stop3	Orden levantamiento Barrera Stop rodillos robot encajonado	DO	A2.7	Bool		
Rodillo_Stop4	Orden levantamiento Barrera Stop rodillos final de encajonado	DO	A3.0	Bool		
Startlight	Iluminación botón de Start	DO	A3.1	Bool		
Stoplight	Iluminación botón de Stop	DO	A3.2	Bool		
Robot1_OrdenX	Orden Posicionamiento del robot ensamblado en el eje X	AO	AW18	Int		
Robot1_OrdenZ	Orden Posicionamiento del robot ensamblado en el eje Z	AO	AW20	Int		

Nombre Elemento	Descripción	Señal Eléctrica	Direccion E/S	Tipo	Verificado por	Fecha
Robot2_OrdenX	Orden Posicionamiento del robot encajonado en el eje X	AO	AW22	Int		
Robot2_OrdenY	Orden Posicionamiento del robot encajonado en el eje Y	AO	AW24	Int		
Robot2_OrdenZ	Orden Posicionamiento del robot encajonado en el eje Z	AO	AW26	Int		
Robot1_PosicionX	Posicion del Robot ensamblado en el eje X	AI	EW28	Int		
Robot1_PosicionY	Posicion del Robot ensamblado en el eje Y	AI	EW30	Int		
Robot2_PosicionX	Posicion del Robot encajonado en el eje X	AI	EW32	Int		
Robot2_PoiscionY	Posicion del Robot encajonado en el eje Y	AI	EW34	Int		
Robot2_PosicionZ	Posicion del Robot encajonado en el eje z	AI	EW36	Int		
Peso_Bascula	Lectura del peso de las piezas	AI	EW12	int		
Robot1_Detector	Detección de pieza Robot ensamblado	DI	E0.0	Bool		
Cinta_Detector1	Detector de llegada pieza Tapa	DI	E0.1	Bool		
Cinta_Detector2	Detector de llegada pieza Base	DI	E0.2	Bool		
ParadaEmergencia1	Parada de Emergecia Zona de ensamblado	DI	E0.3	Bool		
Reset	Rearme del sistema	DI	E0.4	Bool		
Start	Orden de marcha general	DI	E0.5	Bool		
Stop	Orden de para general	DI	E0.6	Bool		
Robot2_Detecto r	Detección de pieza Robot encajonado	DI	E0.7	Bool		
Rodillo_Detector1	Detector de llegada de palet	DI	E1.0	Bool		
Rodillo_Detector2	Detector de llegada de palet, orden de genaracion de caja	DI	E1.1	Bool		
Rodillo_Detector3	Detector de llegada de palet y caja	DI	E1.2	Bool		
Rodillo_Detector4	Detector de llegada de palet, caja y piezas al final de la linea	DI	E1.3	Bool		
ParadaEmergencia2	Parada de Emergecia Zona de empaquetado	DI	E1.4	Bool		

Tabla 9.21: Tabla de Entradas y Salidas para el sistema de ensamblaje.

9.10.2. PRUEBAS DE SECUENCIAS

Protocolos de testeo:

Nombre del Protocolo: Comprobación del funcionamiento del robot de ensamblado.

Tipo de Protocolo: Pruebas Funcionales

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:Comprobar que el robot de ensamblado es capaz de coger una pieza de tipo tapa y la coloca encima de una pieza de tipo base.
Pre-requisitos: El robot debe estar programado, instalado y conectado al PLC.
Objectivos del Test: Comprar que se ensamblan piezas en la primera zona de la línea.

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.	Generar una pieza de tipo tapa y que el robot la recoja	
2.	Generar una pieza de tipo base y que el robot el coloque encima la pieza tapa	
3.	Una vez ensambladas las piezas estas se liberan y se transportan hacia la bascula	

Nº	**Descripción**	**Resultados Esperados**	**Resultados Reales**	**O=Ok F=Fallo**	**Firma Fecha**
1.	Generar una pieza de tipo tapa y que el robot la recoja				
1.1	El sistema genera una pieza de tipo tapa.	Aparece una pieza .			
1.2	El robot localiza la pieza y es capaz de cogerla.	El robot coge la pieza.			
2.	Generar una pieza de tipo base y que el robot el coloque encima la pieza tapa.				
2.1	El sistema genera una pieza de tipo base	Aparece una base			
2.2	El robot ensambla la pieza tapa, que tiene recogida previamente y la coloca encima de la pieza base	El robot ensambla las piezas correctamente			
2.3	Cambiar el sistema a modo manual.El sistema debe de parar la auto-regulacion	El sistema se pone en modo manual. El sistema deja de regularse de forma automática.			
3.	Una vez ensambladas las piezas estas se liberan y se transportan hacia la bascula.				
3.1	El sistema detecta que están las piezas las ensambladas.	Las piezas se ensamblan correctamente			
3.2	El sistema dejar pasar las piezas ensambladas, bajando la barre de bloque Cinta_stop1	La barrera se baja correctamente y el las cintas las transporta hasta la báscula			

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

Nombre del Protocolo: Comprobación de la pesada de las piezas y transporte hacia la zona de encajonado.

Tipo de Protocolo: Pruebas Funcionales

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:Comprobación de que las piezas son pesadas, y dicho peso es recogido por le PLC. Además, las piezas se transportan hacia la zona de encajonado.

Pre-requisitos: La báscula y las cintas transportadoras deben de estar creadas y conectados al PLC.

Objectivos del Test: Comprar el pesaje de las piezas y el transporte a la zona de encajonado.

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.	Recogida del peso de la pieza	
2.	La pieza es transportada hacia la zona de encajonado.	

Nº Prueba	**Descripción**	**Resultados Esperados**	**Resultados Reales**	**O=Ok F=Fallo**	**Firma Fecha**
1.	Recogida del peso de la pieza.				
1.1	La bascula pesa la pieza y este peso es recogido por el PLC.	PLC recoge el peso de la pieza .			
2.	La pieza es transportada hacia la zona de encajonado.				
2.1	La cinta de después de ser pesada, se dirige hacia la zona de empaquetado, gracias a las cintas de transporte 3 y 4.	La pieza se mueve hacia la zona de encajonado.			
2.2	La barrera de stop de la cinta 4, bloquea la pieza hasta que esta es recogida por el robot de encajonado.	La cinta se queda a la espera de ser retirada por el robot de encajonado.			

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

Nombre del Protocolo: Generación de Palets y cajas.

Tipo de Protocolo: Pruebas Funcionales

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:Comprobación de la generación de un palet con una caja encima, para poder ser almacenadas.

Pre-requisitos: Los rodillos de transporte de la zona de encajonado deben de estar creados y conectados al PLC..

Objectivos del Test: Comprar al robot de encajonado le llega un palet con una caja.

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.	Generar un pallet y una caja encima del palet	
2.	Generar una pieza de tipo base y que el robot el coloque encima la pieza tapa	
3.	El conjunto palet/caja es transportado hacia la zona de encajonado	

Nº Prueba	**Descripción**	**Resultados Esperados**	**Resultados Reales**	**O=Ok F=Fallo**	**Firma Fecha**
1.	Generar un pallet y una caja encima del palet				
1.1	El sistema genera un palet.	Aparece un palet .			
1.2	El palet es transportado hacia la zona de generación de caja y es bloqueado por una barrera de stop.	El palet se queda a la espera de la colocación de una caja.			
1.3	El sistema genera una caja y esta se deposita encima del palet.	La caja se coloca encima del palet.			
2.	El conjunto palet/caja es transportado hacia la zona de encajonado.				
2.1	El sistema detecta que se ha generado un palet y este esta se encuentra encima del palet y desbloquea la barreda de stop.	El sistema detecta la caja y desbloque la barrera de stop.			
2.2	Los rodillos transporta el conjunto palet/caja hasta la zona de encajonado y el palet con la caja son bloqueados para que el robot introduzca piezas en la caja	El conjunto caja/palet llega a la zona de encajonado y se quedan a la espera de la introducción de piezas.			

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

Nombre del Protocolo: Encajonado de un máximo de 5 piezas.

Tipo de Protocolo: Pruebas Funcionales

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:Comprobar que el robot de encajonado coloca 5 piezas en el conjunto palet/caja.

Pre-requisitos: El robot debe estar programado, instalado y conectado al PLC.

Objectivos del Test: Comprar que colocan las piezas correctas dentro del conjunto palet/caja.

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.	El robot recoge las piezas bloqueadas por la barrera 4	
2.	El robot deposita en la caja un máximo de 5 piezas.	
3.	Una vez que se han colocado las 5 piezas se desbloquen la cuarta cinta de rodillos.	

Nº Prueba	**Descripción**	**Resultados Esperados**	**Resultados Reales**	**O=Ok F=Fallo**	**Firma Fecha**
1.	El robot recoge las piezas bloqueadas por la barrera 4				
1.1	El robot recoge la pieza bloqueada en la barrera 4.	La pieza es recogida por el robot .			
1.2	Al liberar la pieza de la barrera, se genera otra pieza y se bloque también en la barrera 4.	El sistema genera otra pieza y esta llega hasta la barrera 4.			
2.	El robot deposita en la caja un máximo de 5 piezas.				
2.1	El robot deposita la pieza recogida en la barrera 4 en la caja bloqueada en la cinta de rodillos número 4.	La pieza se deposita en la caja.			
2.2	El robot repite la operación de colocación de piezas un máximo de 5 veces.	El robot coloca en la caja un máximo de 5 piezas.			
3.	Una vez que se han colocado las 5 piezas se desbloquen la cuarta cinta de rodillos.				
3.1	El sistema detecta que hay 5 piezas en la caja y desbloquea la barrera.	El conjunt palet/caja queda libre y empieza a moverse			
3.2	El robot de encajonado se para, quedándose a la espera de la llegada de un nuevo conjunto palet/caja.	El robot se detiene y no empieza a encajonar piezas hasta que llega un palet con una caja.			

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

Nombre del Protocolo: Transporte hacia la zona de almacenaje.

Tipo de Protocolo: Pruebas Funcionales

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:Comprobar que la caja con 5 piezas se dirige hacia la zona de almacenaje.

Pre-requisitos: Se han introducido 5 pizas en la caja, toda la zona de rodillos de almacenaje está conectada al PLC.

Objectivos del Test: Comprar que las piezas encajonadas se dirigen a la zona de almacenaje.

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.	La caja con las cinco piezas se dirige a la zona de almacenaje	

Nº Prueba	**Descripción**	**Resultados Esperados**	**Resultados Reales**	**O=Ok F=Fallo**	**Firma Fecha**
1.	La caja con las cinco piezas se dirige a la zona de almacenaje				
1.1	Las cinta de rodillos desplazan al conjunto palet /caja /piezas hacia la zona de almacenaje.	El conjunto palet /caja /piezas se desplaza hacia la zona de almacenaje.			
1.2	El conjunto palet /caja /piezas queda bloqueada en la barrera de final de línea a la espera de ser llevado almacén automático.	Conjunto palet /caja /piezas se queda bloqueado.			

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

Nombre del Protocolo: Testeo operativa y seguridades del sistema.

Tipo de Protocolo: Pruebas Funcionales

Entorno de Test: _____
Nombre del Testeador: _____
Fecha de Inicio del Test: _____
Fecha de Finalización del Test: _____
Nombre del Equipo Utilizado: _____

Descripción del Test:Comprobar que el sistema puede ser operado desde el cuadro de mandos.

Pre-requisitos: Todo el sistema debe estar conectado al PLC.

Objectivos del Test: Comprar que los operarios pueden manejar el sistema desde el cuadro de mandos.

Número y descripción de las pruebas que forma el test:

	Descripción	**Comentarios**
1.	El operario puede manejar el sistema desde el cuadro de mandos	
2.	Comprobación de las paradas de emergencia.	

Nº Prueba	**Descripción**	**Resultados Esperados**	**Resultados Reales**	**O=Ok F=Fallo**	**Firma Fecha**
1.	El operario puede manejar el sistema desde el cuadro de mandos.				
1.1	El operario arranca el sistema al pulsar el botón de Start.	El sistema arranca y se enciende la luz luminosa del botón Start.			
1.2	El operario para el sistema al pulsar el botón de Stop.	El sistema se para y se enciende la luz luminosa del botón Stop.			
2.	Comprobación de las paradas de emergencia.				
2.1	El operario para el sistema al pulsar la parada de emergencia 1.	El sistema se detiene completamente. Se enciendo la luz luminosa del botón de reset.			
2.2	El operario rearma el sistema pulsando el botón de reset y a continuación el botón de start.	El sistema arranca correctamente, después de re-armarlo.			
2.3	EEl operario para el sistema al pulsar la parada de emergencia 2.	El sistema se detiene completamente. Se enciendo la luz luminosa del botón de reset..			
2.4	El operario rearma el sistema pulsando el botón de reset y a continuación el botón de start.	El sistema arranca correctamente, después de re-armarlo.			

Resultados del Test:

Comentarios del Técnico Responsable/Firma/Fecha:

9.11. CERTIFICACIÓN DEL SISTEMA.

9.11.1. DESCRIPCIÓN DEL SISTEMA Y OBJECTIVOS DE LA CERTIFICACIÓN.

DESCRIPCIÓN DEL SISTEMA

El sistema de ensamblaje consta de dos operativas, la primera de ella consiste en la colocación de una pieza troquelada encima de otra pieza base, también troquelada. Esta operativa la realizará un robot, ya que por razones de seguridad y rendimiento no puede ser realizada por un operario humano. Una vez que la pieza está montada, esta será transportada por una cinta y pesada en una báscula con el fin de registrar su peso, a continuación, otro robot cogerá la pieza ensamblada y la colocará en una caja.

IMPACTO DEL SISTEMA

El sistema ha reducido todas las operaciones manuales que entrañaban algún peligro para el operario, ya que estas se realizan ahora mediante robots. Así mismo se ha aumentado la productividad de la línea e un 30 % ya que los robots realizan en ensamblado y encajonado de una forma más eficiente y a mayor velocidad.

ALCANCE DE LA CERTIFICACIÓN.

La certificación se circunscribe a la primera fase de automatización de la línea, en la que el alcance comprende la introducción de robots y cintas de transporte, controlados por un PLC. Certificando el correcto funcionamiento de todos estos elementos.

9.11.2. ESTRATEGIA DE PRUEBAS UTILIZADA.

Las pruebas se han ejecutado en entorno simulado debido a que, no se dispone de una instalación física en la que poder probar todo el sistema. Las lecciones aprendidas, así como la programación ya testeada en este entorno puede ser aplicadas a un entorno real, reduciendo los tiempos de testeo y pruebas y por lo tanto el tiempo de puesta en marcha del sistema final. Los documentos que se listan a continuación son los que finalmente se han creado para la ejecución de las pruebas:

Titulo de los documentos, nº de versión
Requerimientos de Usuario ->RQUiMMAS/MOSIE1.0
Descripción Funcional ->RQFiMMAS/MOSIE1.0
Diseño detallado del sistema ->DDSiMMAS/MOSIE1.0
Protocolos de testeo y Validación ->PTViMMAS/MOSIE1.0
Certificación del sistema –>CSiMMAS/MOSIE1.0

9.11.3. CRITERIOS DE ACEPTACIÓN.

Los criterios de aceptación de cada una de las pruebas realizadas sobre el sistema se encuentran detallados en cada uno de los documentos de test correspondientes, especificados en el punto 2 del presente informe.

9.11.4. RESOLUCIÓN DE INCIDENCIAS DEL PROYECTO.

No se ha encontrado incidencias durante los protocolos de testeo.

9.11.5. CONCLUSIONES DE LA CERTIFICACIÓN

En el momento de la emisión del presente informe, no existen desviaciones pendientes de los test ejecutados. Al tratarse de un entorno de simulado no se pueden aplicar desviaciones de tipo documental ni procedimental, ya que estas aplican solo a entornos y equipos reales. Por lo tanto, el entorno simulado queda certificado y validad mediante los protocolos de testeo realizados.

9.12. CONCLUSIONES

En este capitulo se aplicado nuestra propuesta metodológica para el desarrollo de un sistema software industrial. Este sistema software industrial se ha simulado mediante el software Factory IO [87], que nos permite conectar escenarios en 3D virtuales a dispositivos físicos.

Para el desarrollo del sistema software, hemos seguido todos los pasos descritos en el capítulo 8:

- Análisis del Sistema.
- Diseño del sistema,
- Despliegue del sistema en servidor OPC UA,

- Despliegue del sistema en un dispositivo industrial.
- Pruebas de testeo y validación.

En la fase de diseño del sistema se ha creado un modelo conforme a los conceptos de iMMAS y MOSIE. En este modelo se recogen todos los elementos del sistema y las relaciones entre ellos. Posteriormente a su creación el modelo se ha desplegado en un servidor OPC UA, utilizando las transformaciones descritas en el capítulo 6.

También se ha desplegado en un PLC siemens, pero en este caso no se han podido utilizar las transformaciones descritas en el capítulo 7, ya que como se ha explicado, el framework de siemens TIA Portal no contempla plenamente todas las características de la especificación IEC 61131-3, por lo que hemos tenido que utilizar las transformaciones descritas en el anexo 1 del capítulo 7. Gracias ha esta correlación hemos conseguido desplegar el modelo creado en la fase de diseño, tanto el servidor OPC UA con el dispositivo Siemens.

Tener un modelo común que recoge todo los elementos del sistema y describe su funcionamiento, desde un punto de vista de diseño funcional del sistema nos ofrece una serie de ventajas, como se puede ver en este ejemplo. La primera ventaja que nos ofrece tener un modelo, es que es fácil comprender como esta organizado todo el sistema software, teniendo claro el número de elementos que lo forman, como se relaciona y que papel desempeñan dentro del sistema. Esto nos facilita el mantenimiento del propio sistema, así como la introducción de nuevos ingenieros ajenos al mismo, ya que todo esta descrito por el modelo iMMAS/MOSIE, que ha sido desplegado en dos elementos claves de un sistema industrial, el PLC y el servidor OPC. La segunda ventaja que ofrece la creación de un modelo es el luenguaje que nos proporciona, el cual ha sido utilizado tanto para crear un sistema software a nivel de dispositivo industrial (PLC), como a nivel de un elemento software (Servidor OPC UA), haciendo que en ambos mundos se utilicen los mismos conceptos y elementos, estandarizando y homogenizando ambos procesos de desarrollo de software, reduciendo tiempos y simplificando todo el proceso de desarrollo.

10

CONCLUSIONES Y TRABAJOS FUTUROS.

"Physics is the universe's operating system."
Steven R Garman. 1944-2016.

10.1. CONCLUSIONES

Este trabajo de tesis realiza varias contribuciones en el desarrollo del software en entornos industriales mediante la aplicación de la ingeniería dirigida por modelos. La utilización de técnicas de ingeniería de software a la hora de desarrollar un sistema industrial se ha quedado encuadrado tradicionalmente en el ámbito académico, ya que el conocimiento y la aplicabilidad de la ingeniería del software en estos entornos tradicionalmente ha sido muy escasa. Mediante este trabajo se pretende abrir una puerta a la utilización de paradigmas como MDE a la hora de desarrollar un sistema software industrial, siendo las principales aportaciones las siguientes:

1. La propuesta metodológica realizada en este trabajo doctoral se basa en la utilización de MDD como paradigma para el desarrollo de sistemas software en entornos industriales. Sin embargo, para utilizar MDD como paradigma en dichos sistemas es necesario por un lado poder crear modelos acordes a las características y necesidades de estos entornos y por otro poder desplegar estos modelos en los componentes software que forma un sistema industrial. Para ello necesitamos en primer lugar un lenguaje de modelado que nos permita crear modelos con características específicas y propias del dominio industriales.

2. Para utilizar paradigmas de desarrollo dirigida por modelos en entornos industriales se ha diseñado un nuevo lenguaje de modelado denominado iMMAS con el que se pueden construir modelos PIM independientes de la plataforma, pensados y diseñados para conceptualizar sistemas software industriales. La creación de este lenguaje se ha basado en la utilización de los conceptos y características de MOF (Meta Object Facility) y de la experiencia adquirida gracias al trabajo realizado durante años automatización y desplegando soluciones software en la industria.

3. Los modelos PIM diseñados en iMMAS permiten modelar la estructura o estática del sistema industrial especificando los elementos que conforman el sistema industrial, sus propiedades y las relaciones que tienen. Por otra parte, los modelos PIM permiten modelar la dinámica o el comportamiento de dichos elementos en base a máquina de estados finitas.

4. Sobre iMMAS se ha diseñado un nuevo perfil denominado MOSIE con el que hemos podido crear un conjunto de elementos predefinidos con características específicas y propias de los dominios industriales. La utilización de iMMAS con el perfil MOSIE simplifica la construcción de modelos PIM.

5. Se han implementado reglas de transformación que han permitido transformar los modelos PIM diseñados en iMMAS a modelos específicos de la plataforma, como por ejemplo los servidores OPC UA. Para ello, se ha tenido que estudiar a fondo el modelo de información de OPC UA para buscar la mejor equivalencia con el lenguaje iMMAS. A partir de modelos PIM correctos en iMMAS podemos obtener modelos PSM correctos en OPC UA.

6. Para poder desplegar iMMAS/MOSIE en un servidor OPC UA, aparte de crear las transformaciones necesarias, hemos tenido que realizar una exploración de las soluciones comerciales actuales que soportan este estándar, analizando la capacidad de estas para alojar modelos complejos. Dado el poco soporte que existente actualmente para ello, hemos tenido que desarrollar un servidor OPC UA propio en el que poder desplegar iMMAS/-MOSIE y que podamos comunicar con dispositivos industriales como PLCs.

7. Se han implementado reglas de transformación de iMMAS a la especifica-

ción IEC 61131-3 que permiten desplegar modelos creados en iMMAS/MOSIE en dispositivos industriales. Para ello se ha tenido que evaluar todas las características y propiedades de esta especificación, escogiendo aquellas que nos permitan establecer una correspondencia lógica y con sentido entre ambos mundos. Mediante esta transformación conseguimos que todos los dispositivos industriales que sigan la especificación IEC 61131-3 y en los que se despliegue iMMAS y MOSIE , se organicen de la misma forma para el mismo modelo. Así, podremos desplegar el mismo modelo en varios tipos de dispositivos industriales aunque sean de distintos fabricantes, con lo que conseguimos un alto grado de interoperabilidad obviando detalles específicos con alguna plataforma de programación de PLCs concreta como Siemens.

8. El proceso de transformación realizado para la especificación IEC 61131-3 ha presentado desafíos, ya que no todas las plataformas de desarrollo para PLCs siguen completamente la especificación IEC 61131-3. En el caso de la plataforma de programación TIA portal para dispositivos Siemens, hemos tenido que realizar una transformación especifica de iMMAS/MOSIE para esta plataforma, debido a la adopción parcial parcialmente la especificación IEC 61131-3 que presenta dicha plataforma.

9. Se ha creado una metodología específica para el desarrollo de sistemas software industriales denominada MIAS que puede ser aplicada a cada uno de los elementos software que forman parte de un entorno industrial. En esta metodología se proponen una serie de pasos concretos, que tiene como objetivo la implantación satisfactoria del sistema software que se está desarrollando. Dicha metodología se basa en tres pilares importantes en el desarrollo del software de un sistema industrial. El primer pilar implica una comprensión detallada del sistema software a construir. El segundo pilar se basa en la creación de un modelo PIM que posteriormente se va a desplegar por transformación en modelos PSM dependientes de la plataformas. El tercer y último pilar es la ejecución de una serie de protocolos de testeo y pruebas que nos garanticen el correcto funcionamiento del sistema software que se materializa en una certificación del sistema.

10. La aplicación de iMMAS/MOSIE de forma completa (metodología y modelado) durante todo el proceso de desarrollo de un sistema software industrial, servidor OPC UA y PLC Siemens, nos garantiza cierto grado de éxito en la entrega del sistema software industrial. En primer lugar por la documentación que se genera, en la que el modelo creado con iMMAS/MOSIE es parte de la misma. Esta documentación puede ser utilizada para realizar futuras ampliaciones del sistema, o consultada para entender el funcionamiento y la lógica sistema. En segundo lugar por el conjunto de pruebas y test al que se tiene que someter el sistema antes de ser utilizado, las cuales nos garantizan que la funcionalidad que se pretende conseguir con el sistema es correcta y por consiguiente el sistema hace lo que tiene que hacer.

En esta tesis se ha expuesto como se pueden utilizar técnicas de ingeniería del software como MDE en entornos industriales. Para ello primero se ha creado un lenguaje de metamodelado recogido en iMMAS, un conjunto de perfiles recogidos en MOSIE y se ha definido una metodología para el desarrollo de sistemas software industriales. Además se han creado todas las herramientas necesarias para que la utilización de iMMAS/MOSIE puedan ser utilizados en sistemas industriales reales y no se queden en tan solo aproximaciones teóricas.

10.2. TRABAJOS FUTUROS

El lenguaje iMMAS es un lenguaje de modelado que permite la extensión del lenguaje mediante la definición de perfiles. Es posible crear por tanto nuevos perfiles como MOSIE que cubran las características de sistemas industriales concretos o mas específicos que no se puedan modelar con los perfiles ya existentes. En este sentido podemos decir que iMMAS y MOSIE esta en continua evolución y crecimiento.

Una de las evoluciones que planteamos para MOSIE es la de cubrir controladores mas complejos basados en lógica difusa, creando nuevos elementos que nos permitan modelar este tipo de controladores. Además de estos controladores, también se podrían crear elementos específicos para modelar robots industriales, antropomórficos de tres grados de libertad, generando modelos que nos permitan programar estos elementos en lenguajes como Rapid [145].

Un nuevo perfil de MOSIE bastante interesante, sobre todo a la hora de traba-

jar con datos de sensores y que nos permitiría realizar ciertas inferencias sobre los datos que manejan estos sensores sería aquel que nos proporcionase conceptos para realizar fusión de datos sobre estos elementos, aplicando alguna técnica de fusión de datos como métodos de inferencia, métodos de estimación, etc..., proporcionando así modelos capaces de fusionar datos de varios sensores distintos y poder así tomar la mejor decisión.

Además de estos perfiles, también seria interesante crear nuevos perfiles de MOSIE para estándares como PackML [146], en el que se recogen como se deben de programar las maquinas de un sistema discreto de envasado. Con lo que podríamos crear modelos específicos para cadenas de envasado según las especificaciones de este estándar.

Aparte de crear mas perfiles en MOSIE, una linea de trabajo futuro bastante interesante desde nuestro punto de vista, consistiría en dotar a los perfiles de MOSIE ya existentes de contenido semántico, utilizando ontologías, especificas para entornos industriales, definiendo conceptos como motor, sensor, etc... que nos ayudarían a reconocer y contextualizar estos elementos, de forma mas precisa, facilitando la organización de los elementos del sistema, permitiendo analizar, evaluar y relacionar los datos y las decisiones tomadas por el propio sistema de forma más fácil gracias a la capacidad de inferencia proporcionada por la ontología. En esta linea podremos analizar cual es el margen de mejora del propio sistema industrial, buscando siempre los 3 objetivos de un sistema industrial, aumento de la calidad, aumento de la productividad y reducción de costes.

Otro aspecto de iMMAS/MOSIE en el que se podría profundizar, sobre todo en las transformaciones a plataformas de programación de PLCs, es el del modelado del comportamiento. Actualmente iMMAS/MOSIE posee la capacidad de crear máquinas de estados con las que describir el comportamiento, los elementos, subsistemas y sistemas industriales. La transformación realizada en este trabajo se ha centrado en la utilización de lenguajes textuales, en concreto el estructurado, pero se podría escoger otro tipo de lenguaje basado e gráficos, como los diagramas de funciones secuenciales y potenciar las transformaciones entre las maquinas de estados y los conceptos que ofrecen este tipo de lenguajes de programación.

Por último, aparte de profundizar en el modelado del comportamiento, otra

funcionalidad que podría ser introducida en iMMAS/MOSIE, utilizando las herramientas adecuadas para ello, como QVT [147], es el de la generación automática del modelo de la plataforma de despliegue, por ejemplo al modelo de información de OPC UA, a partir del modelo creado en iMMAS/MOSIE. Esta transformación automática debe de realizarse estableciendo un conjunto de reglas que nos relacionen los conceptos de iMMAS/MOSIE con los conceptos del modelo de la plataforma de despliegue.

10.3. LISTA DE PUBLICACIONES.

10.3.1. PUBLICACIONES EN REVISTAS Y CAPÍTULOS DE LIBROS

AUTORES: *José Miguel Gutiérrez Guerrero*, Juan A. Holgado-Terriza.
TÍTULO: IMMAS an industrial meta-model for automation system using OPC UA.
REVISTA/LIBRO: Elektronika ir Elektrotechnika 23(3) pp 3-11 (2017).
ISSN: 1392-1215.
INDICIOS DE CALIDAD: Indexado en SCI, SCOPUS, SJR y Google Académico entre otros. El artículo se encuentra en una revista que está en JCR (2017) con índice de impacto 1.088. Se encuentra en el primer cuartil Q3 en el área de ENGENIEERING, ELECTRICAL & ELECTRONICS (194/260). La revista tiene un índice SJR (SCImago Journal and Country Rank) de 0.258. Se encuentra en el primer cuartil Q3 en el área de ELECTRICAL AND ELECTRONICS ENGENIEERING (652/2352).
CLAVE: A.

AUTORES: Raúl Barragan-Gíl, **José Miguel Gutierrez-Guerrero**, Juan A. Holgado-Terriza.
TÍTULO: DaaS (Device as a Service): Un Paradigma de abstracción de dispositivos basados en servicios aplicado a sistemas domóticos.
REVISTA/LIBRO: Actas de XXIV Jornadas de Concurrencia y Sistemas Distribuidos.
VOL./PÁG./EDIT./AÑO: pp. 21-36 (2016).
ISBN: 978-84-16478-90-3.
INDICIOS DE CALIDAD: Cada artículo ha sido revisado por dos miembros del Comité de Programa, que han realizado comentarios sobre los mismos y sugerencias a los autores.

CLAVE: CL.

AUTORES: *José Miguel Gutierrez-Guerrero*, Juan A. Holgado-Terriza, Jesus Muros-Cobos, Gonzalo Pomboza-Junes.
TÍTULO: Redes de Sensores I2C Inteligentes.
REVISTA/LIBRO: Actas de XXIV Jornadas de Concurrencia y Sistemas Distribuidos.
VOL./PÁG./EDIT./AÑO: pp. 77-92 (2016).
ISBN: 978-84-16478-90-3.
INDICIOS DE CALIDAD: Cada artículo ha sido revisado por dos miembros del Comité de Programa, que han realizado comentarios sobre los mismos y sugerencias a los autores.
CLAVE: CL.

AUTORES: *José Miguel Gutiérrez-Guerrero*, Juan A. Holgado-Terriza.
TÍTULO: Mobile Human Machine Interface based in OPC UA for the control of industrial processes.
REVISTA/LIBRO: Actas de las XXXVI Jornadas de Automática.
VOL./PÁG./EDIT./AÑO: pp. 1073-1080. Comité Español de Automática (2015).
ISBN: 978-84-15914-12-9.
INDICIOS DE CALIDAD: Cada artículo ha sido revisado por tres miembros del Comité de Programa, que han realizado comentarios sobre los mismos y sugerencias a los autores.
CITAS WOS: 3 citas corruptas (10/09/2018).
CLAVE: CL.

AUTORES: Sandra Rodríguez-Valenzuela, Juan A Holgado-Terriza, *José M Gutiérrez-Guerre*ro, Jesús L Muros-Cobos.
TÍTULO: Distributed Service-Based Approach for Sensor Data Fusion in IoT Environments.
REVISTA/LIBRO: Sensors.
VOL./PÁG./EDIT./AÑO: Vol 14 (10) pp. 19200-19228. Multidisciplinary Digital Publishing Institute (2014).

ISSN: 1424-8220.
DOI:10.3390/s141019200.
INDICIOS DE CALIDAD: Indexado en SCI, SCOPUS, SJR y Google Académico entre otros. El artículo se encuentra en una revista que está en JCR (2014) con índice de impacto 2.045. Se encuentra en el primer cuartil Q1 en el área de INSTRUMENTS & INSTRUMENTATION (10/56). La revista tiene un índice SJR (SCImago Journal and Country Rank) de 0.707. Se encuentra en el primer cuartil Q1 en el área de INSTRUMENTATION (25/172) y a su vez encuentra en el primer cuartil Q1 en el área de ELECTRICAL AND ELECTRONIC ENGINEERING (155/2061).
CITAS WOS: 7 (10/09/2018), **Google Scholar:** 16 (10/09/2018), **Scopus:** 8 (10/09/2018).
CLAVE: A.

AUTORES: *J.M Gutierrez-Guerrero*, Jesus L. Muros, Sandra Rodriguez, Juan A. Holgado, Miguel Damas.
TÍTULO: Desarrollo de sistemas industriales a través de sistemas empotrados basados en Java.
TIPO DE PARTICIPACION: Comunicación Oral.
CONGRESO: IV Jornadas de Computación Empotrada dentro del marco del CEDI.
PUBLICACION: Publicación de Actas.
LUGAR DE CELEBRACION: Madrid, 17-20 de Septiembre de 2013.
AÑO: 2013.

AUTORES: Sandra Rodríguez-Valenzuela, J Holgado-Terriza, Jesús L Muros-Cobos, *José M Gutiérrez-Guerrero.*
TÍTULO: Data fusion mechanism based on a service composition model for the internet of things.
REVISTA/LIBRO: Actas de las III Jornadas de Computación Empotrada (JCE).
VOL./PÁG./EDIT./AÑO: Vol. 1, pp. 19-21. Universidad Miguel Hernández (2012). ISBN: 978-84-695-4424-2.
INDICIOS DE CALIDAD: Incluye los artículos de las 21 ponencias presentadas en las Jornadas. Número de citas a 20/12/2015 según SCOPUS (1), y según Google Académico (1).

CLAVE: CL.

10.3.2. CONGRESOS

AUTORES: Juan A. Holgado-Terriza, *José Miguel Gutiérrez Guerrero*.
TÍTULO: IMMAS an industrial meta-model for automation system using OPC UA.
TIPO DE PARTICIPACION: Ponencia Oral.
CONGRESO: 21st Internantional Conference Electronics.
PUBLICACION: Publicación de Actas.
LUGAR DE CELEBRACION: Palanga (Lithuania), 19-21 de Junio de 2017.
AÑO: 2017.

AUTORES: Raúl Barragan-Gíl, *José Miguel Gutierrez-Guerrero*, Juan A. Holgado-Terriza.
TÍTULO: DaaS (Device as a Service): Un Paradigma de abstracción de dispositivos basados en servicios aplicado a sistemas domóticos.
TIPO DE PARTICIPACION: Ponencia Oral.
CONGRESO: XXIV Jornadas de Concurrencia y Sistemas Distribuidos.
PUBLICACION: Publicación de Actas.
LUGAR DE CELEBRACION: Granada, 15-17 de Junio de 2016.
AÑO: 2016.

AUTORES: *José Miguel Gutierrez-Guerrero*, Juan A. Holgado-Terriza, Jesus Muros-Cobos, Gonzalo Pomboza-Junes.
TÍTULO: Redes de Sensores I2C Inteligentes.
TIPO DE PARTICIPACION: Ponencia Oral.
CONGRESO: XXIV Jornadas de Concurrencia y Sistemas Distribuidos.
PUBLICACION: Publicación de Actas.
LUGAR DE CELEBRACION: Granada, 15-17 de Junio de 2016.
AÑO: 2016.

AUTORES: *José Miguel Gutiérrez-Guerrero*, Juan A. Holgado-Terriza.
TÍTULO: Mobile Human Machine Interface based in OPC UA for the control of

industrial processes.
TIPO DE PARTICIPACION: Poster.
CONGRESO: XXXVI Jornadas de Automática.
PUBLICACION: Publicación de Actas.
LUGAR DE CELEBRACION: Bilbao, 2-4 de Septiembre de 2015.
AÑO: 2015.

AUTORES: *J.M Gutierrez-Guerrero*, Jesus L. Muros, Sandra Rodriguez, Juan A. Holgado, Miguel Damas.
TÍTULO: Desarrollo de sistemas industriales a través de sistemas empotrados basados en Java.
TIPO DE PARTICIPACION: Comunicación Oral.
CONGRESO: IV Jornadas de Computación Empotrada dentro del marco del CEDI.
PUBLICACION: Publicación de Actas.
LUGAR DE CELEBRACION: Madrid, 17-20 de Septiembre de 2013.
AÑO: 2013.

AUTORES: Sandra Rodríguez-Valenzuela, J Holgado-Terriza, Jesús L Muros-Cobos, *José M Gutiérrez-Guerrero.*
TÍTULO: Data fusion mechanism based on a service composition model for the internet of things.
TIPO DE PARTICIPACION: Comunicación Oral.
CONGRESO: III Jornadas de Computación Empotrada (JCE).
PUBLICACION: Actas de las III Jornadas de Computación Empotrada (JCE).
LUGAR DE CELEBRACION: Elche, 19-21 de Septiembre de 2012. **AÑO:** 2012.

10.3.3. OTRAS PUBLICACIONES

AUTORES: *José M Gutiérrez-Guerrero.*
TÍTULO: IoT in Healthcare: Using the PI System for Continuous Patient Monitoring.
TIPO DE PARTICIPACION: Comunicación Oral.

LUGAR DE CELEBRACION: Londres 16th October 2017.

AUTORES: *José M Gutiérrez-Guerrero.*
TÍTULO: Que puede aportar PI a una fábrica en el marco de la Industria 4.0. Life Science Forum.
TIPO DE PARTICIPACION: Comunicación Oral.
LUGAR DE CELEBRACION: Barcelona 7th November 2017.
AUTORES: *José M Gutiérrez-Guerrero.*
TÍTULO: Digital Trasnformation Forum.
TIPO DE PARTICIPACION: Comunicación Oral.
LUGAR DE CELEBRACION: Madrid 27th April 2017.

AUTORES: *José M Gutiérrez-Guerrero.*
TÍTULO: Participacion en Mesa redonda como experto en sistemas industriales.
TIPO DE PARTICIPACION: Comunicación Oral.
LUGAR DE CELEBRACION: Berlin 26th September 2016.

A

TRANSFORMACION DE iMMAS/MOSIE A TIA PORTAL DE SIEMENS

Los conceptos que ofrece iMMAS y el conjunto de perfiles que ofrece MOSIE han sido transformados conforme a la especificación IEC 61131-3, utilizando los conceptos de UDTs y FBs. El entorno de programación de Siemenes TIA Portal [143] contempla estos conceptos, pero a la hora de desplegar las transformaciones realizadas para IEC 61131-3 nos hemos encontrado con algunas dificultades:

- El entorno TIA Portal no soporta la declaración de variables de tipo ANY como parte de una estructura de definición de tipos de datos o UDT. Al no poder utilizar variables de tipo ANY se han tenido que adaptar las UDTs creadas en el proceso de transformación entre iMMAS/MOSIE y la especificación IEC 61131-3, dando lugar a un conjunto de UDTs y FBs específicos para este entorno.

- TIA Portal tampoco soporta las extensiones ni de UDTs ni de FBs, es decir, no admite definiciones como *TYPE InputOutPutModule EXTENDS Module :* o *FUNCTION_BLOCK AnalogInputFB EXTENDS InputOutputModuleFB*, de manera que perdemos el mecanismo de extensión que nos ha sido muy útil durante todo el proceso de transformación entre iMMAS/MOSIE y IEC 61131-3. Por lo que esta limitación también nos ha obligado a redefinir los FBs creados en la transformación realizada previamente entre iMMAS/MOSIE y IEC 61131-3. Además los UDTs también han tenido que volver a ser

revisados para solventar esta restricción.

- Otro limitación que nos hemos encontrado en este entorno de programación es el nivel de anidación entre FBs. Esta profundidad de anidamiento viene determinada por el tipo de CPU elegida para controlar nuestro sistema industrial. Se puede encontrar mas información en [148]. Esta restricción nos condiciona la utilización de FBs de forma anidada, por lo que nos podemos encontrar en la situación de no poder utilizar mas el concepto de FB, sobre todo en sistemas complejos, y en este caso nos veremos obligados a utilizar el concepto de OB (Organizational Blocks).

Todas estas dificultados son causadas por la adopción parcial de la especificación IEC 61131-3, que hace el entorno de programación de Siemens TIA Portal. En esta instrucción técnica [149] podemos ver las compatibilidades ente las especificación IEC 61131-3 y TIA Portal. Debido a esta adopción parcial debemos de realizar ciertas adaptaciones en las transformaciones realizadas para poder desplegar iMMAS/MOSIE en este entorno de desarrollo. Básicamente hemos suplido estas carencias, no transformando algunos conceptos abstractos como *Module* o *Component* y reutilizando variables de tipo UDT o FB a la hora de realizar la definición de UDTs o FBs nuevos, simulando así el mecanismo de herencia. Otra decisión que se ha tomado es la de no crear UDTs para elementos que no posean atributos nuevos con respecto al elemento del que extienden, creando solo en estos casos un FB que gestiona todo las estructuras de datos asociadas del elemento en cuestión.

A.1. DESPLIGUE DE iMMAS Y MOSIE EN TIA PORTAL

A continuación vamos a describir todas las transformaciones y decisiones tomadas para poder desplegar iMMAS/MOSIE en un PLC de siemens utilizando TIA Portal.

A.1.1. iMMAS EN TIA PORTAL

Despliegue del paquete Base:

Los elementos de este paquete son: *NamedElement, Type, TypedElement, Asociation, DataType, Operation, Parameter, Property, Classifier, Module, Component,*

iMMAS	IEC-61131-3	Correpondencia
NamedElement	-	No existe
Type	Type	Directa
DataType	Type	Directa
TypeElement	Struct Type	Directa
Operation	-	No Contemplada
Parameter	-	No Contemplada
Property	-	No Contemplada
Asociation	-	No Contemplada
Classifier	-	No Contemplada
Componet	-	No Contemplada
ActiveComponet	-	No Contemplada
Module	-	No Contemplada
IndustrialData	-	Propia de cada Framework de programación

Tabla A.1: Resumen de las correspondencias básicas entre iMMAS base y Siemens TIA Portal.

ActiveComponent, IndustrialData, Value y Address. La correlación realizada para estos elementos se puede ver en la tabla A.1.

Como se puede ver en la tabla anterior, a excepción de los conceptos relacionados con la definición de tipos de datos, que si tiene una relación directa, para los demás conceptos no se puede realizar una correlación, debido a las limitaciones, ya comentadas, que presenta la plataforma de desarrollo de siemens TIA Portal. Por ello para los conceptos *Classifier*, *Component*, *ActiveComponet* y *Module*, nos encontramos con que no tenemos una transformación directa como ocurría en las transformaciones realizadas para la especificación IEC-61131-3.

Al no disponer de mecanismos como la herencia, tenemos que ser muy selectivos a la hora de escoger el concepto que se va a transformar como elemento base y del cual van a partir todas las definiciones posteriores, de forma que no se compliquen excesivamente las transformaciones a realizar, transformando solo aquellos elementos que tengan un significado y contenido útil, a la hora de crear un modelo.

EL PAQUETE PACKAGE EN TIA PORTAL

Este paquete está formado por un único concepto Package, pensado para agrupar conceptos similares. En TIA Portal, al igual que ocurría en IEC 61113-3, este concepto no tiene cabida, tan solo disponemos de la posibilidad de agrupar UDTs y FBs en carpetas.

EL PAQUETE BEHAVIOUR EN TIA PORTAL

Para realizar las transformaciones de los elementos de este paquete vamos a utilizar las mismas correlaciones que se realizaron en el caso del estándar IEC 61131-3, apoyándonos en el concepto de FB, pero con la salvedad de que no podemos realizar extensiones de estos. Aunque esta dificultad la podremos solventar utilizando, como veremos en los siguientes apartados, mediante variables locales en el FB, siendo estas del tipo de FB del cual queremos extender.

EL PAQUETE TYPE EN TIA PORTAL

Las correlaciones para los tipos de iMMAS a los tipos soportados por TIA Portal se pueden ver en la siguiente tabla:

Concepto de iMMAS	TIA Portal	Correpondencia
Int	Int	Directa
Double	Real	Directa
Boolean	Bool	Directa
String	String	Directa

Tabla A.2: Resumen de las correspondencias entre iMMAS Type y TIA Portal

A.1.2. MOSIE EN TIA PORTAL

Una vez que hemos realizado la correlación de iMMAS, vamos a proceder con los paquetes y elementos de MOSIE:

MODULES EN TIA PORTAL

El paquete Input/Ouput en TIA Portal.

Todos los conceptos de este paquete extienden del concepto *InputOutputModel*, este concepto deriva a su vez de *Module*, concepto que se encuentra recogido en

el paquete base de iMMAS. Como en TIA portal no podemos utilizar la extensión y tenemos ciertas restricciones a la hora de anidar FBs, la correlación que se ha realizado parte del concepto *InputOutputModel* y no del concepto *Module* como se especifica MOSIE. Se han creado 4 tipos de *InputOutputModel*, uno por cada tipo de dato que contempla iMMAS:

InputOutputModelInteger	
UDT	**FB**
TYPE "InputOutputModuleInteger" VERSION : 0.1 STRUCT status : Bool; value : Int; statusSim : Bool; unit:String; END_STRUCT; END_TYPE	Sin FB.

InputOutputModelBool	
UDT	**FB**
TYPE "InputOutModuleBool" VERSION : 0.1 STRUCT status : Bool; value : Bool; statusSim : Bool; unit:String; END_STRUCT; END_TYPE	Sin FB.

InputOutputModelFloat	
UDT	**FB**
TYPE "InputOutModuleFloat" VERSION : 0.1 STRUCT status : Bool; value : Real; statusSim : Bool; unit:String; END_STRUCT; END_TYPE	Sin FB.

InputOutputModelString	
UDT	**FB**
TYPE "InputOutModuleString" VERSION : 0.1 STRUCT status : Bool; value : String; statusSim : Bool; unit:String; END_STRUCT; END_TYPE	Sin FB.

Una vez transformados los conceptos de *InputOutModule*, podemos pasar a realizar las transformaciones del resto de los conceptos de este paquete.

digitalOutput

UDT

```
TYPE "DigitialOutput"
VERSION : 0.1
STRUCT
out : "InputOutputModelBool
    ";
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "DigitalOutputFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_status : Bool;
i_value : Bool;
i_statusSim : Bool;
END_VAR
VAR_OUTPUT
o_status : Bool;
o_statusSim : Bool;
o_value : Bool;
END_VAR
VAR
diout_local : "DigitialOutput";
END_VAR
BEGIN
//ENTRADAS
#diout_local.out.value := #i_value;
#diout_local.out.status := #i_status;
#diout_local.out.statusSim:= #
    i_statusSim;


//SALIDAS
#o_value:=#diout_local.out.value;
#o_status:=#diout_local.out.status ;
#o_statusSim:=#diout_local.out.
    statusSim;
END_FUNCTION_BLOCK
```

digitalOutput

UDT

```
TYPE "DigitalInput"
VERSION : 0.1
STRUCT
input : "
    InputOutputModelBool";
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "digitalInputFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_status : Bool;
i_statusSim : Bool;
END_VAR
VAR_OUTPUT
o_status : Bool;
o_statusSim : Bool;
o_value : Bool;
END_VAR
VAR
din_local : "DigitalInput";
END_VAR
BEGIN
//ENTRADAS
#din_local.input.status := #i_status;
#din_local.input.statusSim := #
    i_statusSim;

//SALIDAS
#o_value := #din_local.input.value;
#o_status := #din_local.input.status;
#o_statusSim := #din_local.input.
    statusSim;
END_FUNCTION_BLOCK
```

AutoManDigitalOutput

UDT

```
TYPE "AutoManDigitalOutput"
VERSION : 0.1
STRUCT
mode : "AutoManMode";
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "AutoManDigitalOutputFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_mode : "AutoManMode";
i_manValue : Bool;
ii_value : Bool;
ii_status : Bool;
ii_statusSim : Bool;
END_VAR
VAR_OUTPUT
oo_statusSim : Bool;
oo_status : Bool;
oo_value : Bool;
o_mode : "AutoManMode";
END_VAR
VAR
au_localFB : "DigitalOutputFB";
au_local : "AutoManDigitalOutput";
END_VAR
BEGIN
//Entradas Propias
#au_local.mode.mode := #i_mode.mode;

IF (#au_local.mode.mode = "Auto") THEN
#au_localFB(i_status := #ii_status,
i_statusSim := #ii_statusSim,
i_value := #ii_value,
o_status => #oo_status,
o_statusSim => #oo_statusSim,
o_value => #oo_value);
ELSIF (#au_local.mode.mode = "Man")
    THEN
#au_localFB(i_status := #ii_status,
i_statusSim := #ii_statusSim,
i_value := #i_manValue,
o_status => #oo_status,
o_statusSim => #oo_statusSim,
o_value => #oo_value);
END_IF;
//Salidas Propias
#o_mode.mode:=#au_local.mode.mode;
END_FUNCTION_BLOCK
```

AutoManMode

UDT

```
TYPE "AutoManMode"
VERSION : 0.1
STRUCT
mode : Int;
END_STRUCT;
END_TYPE
```

Constantes

```
//Modos de trabajo para
    AutoManDigitalOuput
Auto Int 0
//Modos de trabajo para
    AutoManDigitalOuput
Man  Int 1
```

AnalogOutPut

UDT

```
TYPE "AnalogOutput"
VERSION : 0.1
STRUCT
out : "InputOutModuleFloat"
    ;
END_STRUCT
END_TYPE
```

FB

```
FUNCTION_BLOCK "AnalogOutputFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_status : Bool;
i_value : Real;
i_statusSim : Bool;
END_VAR
VAR_OUTPUT
o_status : Bool;
o_statusSim : Bool;
o_value : Real;
END_VAR
VAR
anout_local : "AnalogOutput";
END_VAR
BEGIN
//ENTRADAS
#anout_local.out.value := #i_value;
#anout_local.out.status := #i_status;
#anout_local.out.statusSim:= #
    i_statusSim;
//SALIDAS
#o_value:=#anout_local.out.value;
#o_status:=#anout_local.out.status ;
#o_statusSim:=#anout_local.out.
    statusSim;
END_FUNCTION_BLOCK
```

AutoManAnalogOutput

UDT

```
TYPE "AutoManAnalogOutput"
VERSION : 0.1
STRUCT
mode : "AutoManMode";
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "AutoManAnalogOutputFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_mode : "AutoManMode";
i_manValue : Real;
ii_value : Real;
ii_status : Bool;
ii_statusSim : Bool;
END_VAR
VAR_OUTPUT
oo_statusSim : Bool;
oo_status : Bool;
oo_value : Real;
o_mode : "AutoManMode";
END_VAR
VAR
au_localFB : "AnalogOutputFB";
au_local : "AutoManAnalogOutput";
END_VAR
BEGIN
//Entradas Propias
#au_local.mode.mode := #i_mode.mode;
IF (#au_local.mode.mode = "Auto") THEN
#au_localFB(i_status := #ii_status,
i_statusSim := #ii_statusSim,
i_value := #ii_value,
o_status => #oo_status,
o_statusSim => #oo_statusSim,
o_value => #oo_value);
ELSIF (#au_local.mode.mode = "Man")
    THEN
#au_localFB(i_status := #ii_status,
i_statusSim := #ii_statusSim,
i_value := #i_manValue,
o_status => #oo_status,
o_statusSim => #oo_statusSim,
o_value => #oo_value);
END_IF;
//Salidas Propias
#o_mode.mode:=#au_local.mode.mode;
END_FUNCTION_BLOCK
```

AnalogInput

UDT

```
TYPE "AnalogInput"
VERSION : 0.1
STRUCT
input : "
    InputOutModuleFloat";
autoMaxValue : Real;
autoMinValue : Real;
calMaxValue : Real;
calMinValue : Real;
manMaxValue : Real;
manMinValue : Real;
mode : "AnalogInputMode";
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "analogInputFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_status : Bool;
i_statusSim : Bool;
i_mode : "AnalogInputMode";
END_VAR
VAR_OUTPUT
o_status : Bool;
o_statusSim : Bool;
o_value : Real;
o_maxValue : Real;
o_minValue : Real;
o_mode : "AnalogInputMode";
END_VAR
VAR
in_local : "AnalogInput";
END_VAR
BEGIN
//ENTRADAS
#in_local.input.status := #
    i_status;
#in_local.input.statusSim := #
    i_statusSim;
#in_local.mode.mode := #i_mode.
    mode;

//SALIDAS
IF (#in_local.mode.mode = "
    automatic") THEN
#o_maxValue := #in_local.
    autoMaxValue;
#o_minValue := #in_local.
    autoMinValue;
ELSIF (#in_local.mode.mode = "
    calibration") THEN
#o_maxValue := #in_local.
    calMaxValue;
#o_minValue := #in_local.
    calMinValue;
ELSIF (#in_local.mode.mode = "
    manual") THEN
#o_maxValue := #in_local.
    manMaxValue;
#o_minValue := #in_local.
    manMinValue;
END_IF;

#o_value := #in_local.input.value
    ;
#o_status := #in_local.input.
    status;
#o_statusSim := #in_local.input.
    statusSim;
#o_value := #in_local.input.value
    ;
END_FUNCTION_BLOCK
```

AnalogInputMode	
UDT	**Constantes**
TYPE "AnalogInputMode" VERSION : 0.1 STRUCT mode : Int; END_STRUCT; END_TYPE	//Modos de trabajo para una salida analogica automatic Int 2 //Modos de trabajo para una salida analogica calibration Int 3 //Modos de trabajo para una salida analogica manual Int 4

Las transformaciones realizadas para este paquete se han basado en la creación de nuevas UDTs y FBs, para todos los elementos, en algunos casos, como por ejemplo en *AutoManDigitalOutput* y *AutoManAnalogOutput*, los FBs creados tiene dos variables propias una de tipo UDT *AutoManDigitalOutput* y otra de tipo FB *DigitalOutputFB*, que gestiona la entrada digital o analógica, de la que desciende el elemento. Como se dijo al principio de este anexo, las estructuras de datos de las que extiendan los diferentes elementos, se gestionaran utilizando los FBs de estos elementos y no la estructura directamente.

El paquete Alarms en TIA Portal.

Para este paquete solo se han realizado las transformaciones de *AnalogAlarm* y *DigitalAlarm*, el concepto *alarm* no ha sido transformado, debido a que introduce una complejidad adicional al no disponer del mecanismo de herencia en TIA Portal, ni de variables de tipo *ANY*, además así también reducimos la complejidad en el anidamiento de FBs.

DigitalOperator	
UDT	**Constantes**

TYPE "DigitalOperator"
VERSION : 0.1
STRUCT
operaror : SInt;
END_STRUCT;
END_TYPE
	//Comparacion para igual DigitalAlarm-AnalogAlarm
	equal Int 5
	//Comparacion diferente DigitalAlarm-AnalogAlarm
	noEqual Int 6

DigitalAlarm

UDT

```
TYPE "DigitalAlarm"
VERSION : 0.1
STRUCT
Cond : "DigitalOperator";
status : Bool;
value : Bool;
refMonitoredValue : Bool;
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "DigitalAlarmFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_cond : "DigitalOperator";
i_status : Bool;
i_refMonitoredValue : Bool;
END_VAR
VAR_OUTPUT
o_status : Bool;
o_value : Bool;
o_cond : "DigitalOperator";
END_VAR
VAR
di_local : "DigitalAlarm";
END_VAR
BEGIN
//Entradas
#di_local.status := #i_status;
#di_local.Cond := #i_cond;
#di_local.refMonitoredValue := #
    i_refMonitoredValue;
//TO DO (Customize Code)

IF (#di_local.Cond.operaror = "equal")
    THEN
IF (#di_local.refMonitoredValue) THEN
#di_local.value := FALSE;  //alarma no
    disparada  con un valor true
ELSE
#di_local.value := True;
END_IF;
ELSIF (#di_local.Cond.operaror= "
    noEqual") THEN
IF ( NOT #di_local.refMonitoredValue)
    THEN
#di_local.value := TRUE; //alarma
    disparada  con un valor true
ELSE
#di_local.value := FALSE;
END_IF;
END_IF;
//salidas
#o_status := #di_local.status;
#o_value := #di_local.value;
#o_cond := #di_local.Cond;
END_FUNCTION_BLOCK
```

AnaloglAlarm

UDT

```
TYPE "AnalogAlarm"
VERSION : 0.1
STRUCT
Cond : "AnalogOperator";
status : Bool;
value : Bool;
refMonitoredValue : Real;
setPoint : Real;
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "AnalogAlarmFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_cond : "AnalogOperator";
i_status : Bool;
i_refMonitoredValue : Real;
i_setPoint : Real;
END_VAR
VAR_OUTPUT
o_status : Bool;
o_value : Bool;
o_setPoint : Real;
o_cond : "AnalogOperator";
END_VAR
VAR
an_local : "AnalogAlarm";
END_VAR
BEGIN
//Entradas
#an_local.status := #i_status;
#an_local.refMonitoredValue := #
    i_refMonitoredValue;
#an_local.Cond := #i_cond;
#an_local.setPoint := #i_setPoint;
//TO DO (Customize Code)
IF (#an_local.Cond.operaror ="equal")
     THEN
IF (#an_local.setPoint = #an_local.
    refMonitoredValue) THEN
#an_local.value := TRUE;
ELSE
#an_local.value := FALSE;
END_IF;
ELSIF (#an_local.Cond.operaror ="
    greaterThan") THEN
IF (#an_local.setPoint > #an_local.
    refMonitoredValue) THEN
#an_local.value := TRUE;
ELSE
#an_local.value := FALSE;
END_IF;
ELSIF (#an_local.Cond.operaror = "
    lessThan") THEN
IF (#an_local.setPoint < #an_local.
    refMonitoredValue) THEN
#an_local.value := TRUE;
ELSE
#an_local.value := FALSE;
END_IF;
ELSIF (#an_local.Cond.operaror = "
    noEqual") THEN
IF (#an_local.setPoint <> #an_local.
    refMonitoredValue) THEN
#an_local.value := TRUE;
ELSE
#an_local.value := FALSE;
END_IF;

END_IF;
//salidas
#o_setPoint := #an_local.setPoint;
#o_status := #an_local.status;
#o_value := #an_local.value;
#o_cond := #an_local.Cond;
END_FUNCTION_BLOCK
```

AnalogOperator

UDT

```
TYPE "AnalogOperator"
VERSION : 0.1
STRUCT
operaror : SInt;
END_STRUCT;
END_TYPE
```

Constantes

```
equal  Int 5 //Comparacion para igual
    DigitalAlarm-AnalogAlarm
noEqual  Int 6 //Comparacion diferente
    DigitalAlarm-AnalogAlarm
lessThan Int 7 //Comparacion Menor que
    AnalogAlarm
greaterThan Int 8 //Comparacion Mayor
    que AnalogAlarm
```

El paquete Control en TIA Portal.

ActivationControl

UDT

```
TYPE "ActivationControl"
VERSION : 0.1
STRUCT
numAct : Int;
resetAct : Bool;
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "ActivationControlFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_numAct : Int;
i_resetAct : Bool;
END_VAR
VAR_OUTPUT
o_numAct : Int;
END_VAR
VAR
ac_local : "ActivationControl";
END_VAR
BEGIN
//Entradas
#ac_local.numAct := #i_numAct;
#ac_local.resetAct := #i_resetAct;
//TO DO (Customize Code)
IF (#ac_local.resetAct = TRUE) THEN
#ac_local.numAct := 0;
END_IF;
//Salidas
#o_numAct := #ac_local.numAct;
END_FUNCTION_BLOCK
```

ControlParameters

UDT

```
TYPE "ControlParameters"
VERSION : 0.1
STRUCT
 externalSetPoint : Real;
mode : "modeSetPoint";
 setPoint : Real;
 outControl : Real;
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "ControlParametersFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_externalSetPoint : Real;
i_mode : "modeSetPoint";
i_setPoint : Real;
i_outControl : Real;
END_VAR
VAR_OUTPUT
 o_externalSetPoint : Real;
o_mode : "modeSetPoint";
 o_setPoint : Real;
 o_outControl : Real;
END_VAR
VAR
 co_local : "ControlParameters";
END_VAR
BEGIN
//Entradas
#co_local.externalSetPoint := #
    i_externalSetPoint;
#co_local.mode := #i_mode;
#co_local.setPoint := #i_setPoint;
#co_local.outControl := #i_outControl;

//TO DO (Customize Code)

//salidas
#o_externalSetPoint := #co_local.
    externalSetPoint;
#o_mode := #co_local.mode;
#o_setPoint := #co_local.setPoint;
#o_outControl := #co_local.outControl;
END_FUNCTION_BLOCK
```

ModeSetPoint

UDT

```
TYPE "modeSetPoint"
VERSION : 0.1
STRUCT
mode : SInt;
END_STRUCT;
END_TYPE
```

Constantes

```
//Setpoint interno
internal Int 9
//SetPoint indicado por ub sistema
    externo
external Int 10
```

PIDControlParameters

UDT

```
TYPE "PIDControlParameters"
VERSION : 0.1
STRUCT
DGain : Real;
PGain : Real;
IGain : Real;
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "PIDControlParametersFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_externalSetPoint : Real;
i_mode : "modeSetPoint";
i_outControl : Real;
i_setPoint : Real;
i_DGain : Real;
i_PGain : Real;
i_IGain : Real;
END_VAR
VAR_OUTPUT
o_externalSetPoint : Real;
o_mode : "modeSetPoint";
o_setPoint : Real;
o_outControl : Real;
o_DGain : Real;
o_PGain : Real;
o_IGain : Real;
END_VAR
VAR
pid_local : "PIDControlParameters";
pid_localFB : "ControlParametersFB";
END_VAR
BEGIN
//Entradas
#pid_local.DGain := #i_DGain;
#pid_local.PGain := #i_PGain;
#pid_local.IGain := #i_IGain;
//TO DO (Customize Code)
#pid_localFB(i_externalSetPoint := #
    i_externalSetPoint,
i_mode := #i_mode,
i_outControl := #i_outControl,
i_setPoint := #i_setPoint,
o_externalSetPoint=>#o_externalSetPoint,
o_mode => #o_mode,
o_outControl => #o_outControl,
o_setPoint => #o_setPoint);
//salidas
#pid_local.DGain:=#i_DGain;
#pid_local.PGain := #i_PGain;
#pid_local.IGain := #i_IGain;
END_FUNCTION_BLOCK
```

Power

UDT

```
TYPE "Power"
VERSION : 0.1
STRUCT
 acelerationStatus : Real;
 blockedStatus : Bool;
 direction : "
     modeRotationDirection";
 isDualDirection : Bool;
END_STRUCT;
END_TYPE
```

FB

```
FUNCTION_BLOCK "PowerFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_blockedStatus : Bool;
i_direction : "modeRotationDirection";
END_VAR
VAR_OUTPUT
o_acelerationStatus : Real;
o_blockedStatus : Bool;
o_direction : "modeRotationDirection";
o_isDualDirection : Bool;
END_VAR
VAR
pw_local : "Power";
END_VAR
BEGIN
//ENTRADAS
#pw_local.blockedStatus := #
     i_blockedStatus;
#pw_local.direction := #i_direction;

//TO DO (Customize Code)

//SALIDAS
#o_acelerationStatus := #pw_local.
     acelerationStatus;
#o_blockedStatus := #pw_local.
     blockedStatus;
#o_direction := #pw_local.direction;
#o_isDualDirection := #pw_local.
     isDualDirection;
END_FUNCTION_BLOCK
```

ModeRotationDirection

UDT

```
TYPE "modeRotationDirection
     "
VERSION : 0.1
STRUCT
mode : SInt;
END_STRUCT;
END_TYPE
```

Constantes

```
//Paro
idle Int 11
//Movimiento directo
direct Int 12
//Movimiento inverso
inverse Int 13
```

El paquete Components en TIA Portal.

Los conceptos de este perfil extienden de *Component* y de *ActiveComponent*, además están formados por elementos del paquete Modules y no poseen ningún atributo adicional nuevo que no sea de algún tipo de modulo ya existente en MOSIE. Por ello la transformación realizada de estos elementos a TIA Portal se han basado en la creación de FBs que gestionan las diferentes variables que representan a cada concepto.

El paquete Devices en IEC-61131-3.

Los dos principales conceptos de este paquete son *Sensor* y *Actuator*, ambos elementos pueden ser de tipo digital o de tipo analógico, además también pueden tener alarmas. Por simplicidad y debido a las limitaciones (anidamiento y carencia de extensión), no se ha realizado ninguna transformación para los conceptos Sensor y Actuator, si no que directamente se han transformado los conceptos *DigitalSensor*, *AnalogSensor*, *DigitalActuator*, *AnalogActuator* y *VFDActuator* mediante FBs.

DigitalSensor

UDT	FB

Sin UDT	FUNCTION_BLOCK "DigitalSensor"
	{ S7_Optimized_Access := 'TRUE' }
	VERSION : 0.1
	VAR_INPUT
	i_statusSim : Bool;
	i_status : Bool;
	i_cond : "DigitalOperator";
	END_VAR
	VAR_OUTPUT
	o_value : Bool;
	o_isAlarm : Bool;
	o_status : Bool;
	o_statusSim : Bool;
	END_VAR
	VAR
	al_status : Bool;
	al_cond : "DigitalOperator";
	input: "digitalInputFB";
	alrm : "DigitalAlarmFB";
	in_status : Bool;
	in_statusSim : Bool;
	END_VAR
	BEGIN
	//gestion de elementos y Entradas/salidas
	#input(i_status:=#i_status,
	i_statusSim:=#i_statusSim,
	o_value=>#o_value,
	o_status=>#in_status,
	o_statusSim=>#in_statusSim);
	#alrm(i_cond := #i_cond,
	i_refMonitoredValue := #input.o_value,
	i_status := #i_status,
	o_value => #o_isAlarm,
	o_cond=>#al_cond,
	o_status=>#al_status);
	// el estado del objecto es el estado de todos us elementos
	#o_status := (#in_status AND #al_status);
	// el estado de simulacion del objecto es el estado de todos us elementos
	#o_statusSim := #in_statusSim;
	END_FUNCTION_BLOCK

AnalogSensor

UDT	FB

Sin UDT	FUNCTION_BLOCK "AnalogSensorFB"
	{ S7_Optimized_Access := 'TRUE' }
	VERSION : 0.1
	VAR_INPUT
	i_statusSim : Bool;
	i_status : Bool;
	i_cond : "AnalogOperator";
	END_VAR
	VAR_OUTPUT
	o_value : Real;
	o_isAlarm : Bool;
	o_status : Bool;
	o_statusSim : Bool;
	END_VAR
	VAR
	input : "analogInputFB";
	alrm : "AnalogAlarmFB";
	in_status : Bool;
	al_status : Bool;
	al_cond : "AnalogOperator";
	in_statusSim : Bool;
	END_VAR
	BEGIN
	//gestion de elementos y Entradas/salidas
	#input(i_status:=#i_status,
	i_statusSim:=#i_statusSim,
	o_value=>#o_value,
	o_status=>#in_status,
	o_statusSim=>#in_statusSim);
	#alrm(i_cond := #i_cond,
	i_refMonitoredValue := #input.o_value,
	i_status := #i_status,
	o_value => #o_isAlarm, o_cond=>#al_cond,o_status=>#
	al_status);
	// el estado del objecto es el estado de todos us
	elementos
	// Si uno esta desactivado el elemento esta desactivado
	#o_status := (#in_status AND #al_status);
	// el estado de simulacion del objecto es el estado de
	todos us elementos
	// Si uno esta desactivado el elemento esta desactivado
	#o_statusSim := #in_statusSim;
	END_FUNCTION_BLOCK

DigitalActuator

UDT	FB

Sin UDT	FUNCTION_BLOCK "DigitalActuatorFB"
	{ S7_Optimized_Access := 'TRUE' }
	VERSION : 0.1
	VAR_INPUT
	i_resetActivation : Bool;
	i_mode : "AutoManMode";
	i_status : Bool;
	i_manValue : Bool;
	i_value : Bool;
	i_numAct : Int;
	i_alarmCond : "DigitalOperator";
	i_statusSim : Bool;
	END_VAR
	VAR_OUTPUT
	o_numAct : Real;
	o_mode : "AutoManMode";
	o_status : Bool;
	o_statusSim : Bool;
	o_isAlarm : Bool;
	o_value : Bool;
	END_VAR
	VAR
	input : "digitalInputFB";
	output : "AutoManDigitalOutputFB";
	alrm : "DigitalAlarmFB";
	numAct : "ActivationControlFB";
	in_status : Bool;
	in_statusSim : Bool;
	ou_status : Bool;
	ou_statusSim : Bool;
	ou_value : Bool;
	al_cond : "DigitalOperator";
	al_status : Bool;
	END_VAR
	BEGIN
	//gestion de elementos y Entradas/salidas
	#input(i_status := #i_status,
	i_statusSim := #i_statusSim,
	o_value=>#o_value,
	o_status=>#in_status,
	o_statusSim=>#in_statusSim);
	//el Ordenes del actuador digital
	#output(i_manValue := #i_manValue,
	i_mode := #i_mode,
	ii_value:=#i_value,
	ii_status:=#i_status,
	ii_statusSim:=#i_statusSim,
	o_mode => #o_mode,
	oo_status=>#ou_status,
	oo_statusSim=>#ou_statusSim,
	oo_value=>#ou_value);
	#numAct(i_numAct := #i_numAct,
	i_resetAct := #i_resetActivation,
	o_numAct => #o_numAct);
	#alrm(i_cond := #i_alarmCond,
	i_status := #i_status,
	i_refMonitoredValue := #input.o_value,
	o_value => #o_isAlarm,
	o_cond=>#al_cond,
	o_status=>#al_status);
	// el estado del objecto es el estado de todos sus elementos
	#o_status := (#ou_status AND #in_status AND #al_status);
	// el estado de simulacion del objecto es el estado de todos sus elementos
	#o_statusSim := (#ou_statusSim AND #in_statusSim);
	END_FUNCTION_BLOCK

AnalogActuator

UDT	FB
Sin UDT	

```
FUNCTION_BLOCK "AnalogActuatorFB"
{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_resetActivation : Bool;
i_mode : "AutoManMode";
i_status : Bool;
i_manValue : Real;
i_value : Real;
i_numAct : Int;
i_alarmCond : "AnalogOperator";
i_statusSim : Bool;
END_VAR
VAR_OUTPUT
o_numAct : Int;
o_mode : "AutoManMode";
o_status : Bool;
o_statusSim : Bool;
o_isAlarm : Bool;
o_value : Real;
END_VAR
VAR
input : "analogInputFB";
output : "AutoManAnalogOutputFB";
alrm : "AnalogAlarmFB";
numAct : "ActivationControlFB";
in_status : Bool;
in_statusSim : Bool;
ou_status : Bool;
ou_statusSim : Bool;
ou_value : Real;
cond : "DigitalOperator";
al_cond : "DigitalOperator";
al_status : Bool;
END_VAR
BEGIN
//gestion de elementos y Entradas/salidas
#input(i_status := #i_status,
i_statusSim := #i_statusSim,
o_value=>#o_value,
o_status=>#in_status,
o_statusSim=>#in_statusSim);
//el Ordenes del actuador digital
#output(i_manValue := #i_manValue,
i_mode := #i_mode,
ii_value:=#i_value,
ii_status:=#i_status,
ii_statusSim:=#i_statusSim,
o_mode => #o_mode,
oo_status=>#ou_status,
oo_statusSim=>#ou_statusSim,
oo_value=>#ou_value);
#numAct(i_numAct := #i_numAct,
i_resetAct := #i_resetActivation,
o_numAct => #o_numAct);
//alarmas como accedemos al elemento input que activa la
    alarma
#alrm(i_cond := #i_alarmCond,
i_status := #i_status,
i_refMonitoredValue := #input.o_value,
o_value => #o_isAlarm,
o_cond=>#al_cond,
o_status=>#al_status
);
// el estado del objecto es el estado de todos sus elementos
#o_status := (#ou_status AND #in_status AND #al_status);
// el estado de simulacion del objecto es el estado de todos
    sus elementos
#o_statusSim := (#ou_statusSim AND #in_statusSim);
END_FUNCTION_BLOCK
```

VFDActuator

UDT	FB

Sin UDT	FUNCTION_BLOCK "VFDActuator" { S7_Optimized_Access := 'TRUE' }
	VERSION : 0.1
	VAR_INPUT
	i_blokedStatus : Bool;
	i_rotationDirection : "modeRotationDirection";
	i_resetActivation : Bool;
	i_mode : "AutoManMode";
	i_status : Bool;
	i_manValue : Real;
	i_value : Real;
	i_numAct : Int;
	i_alarmCond : "AnalogOperator";
	i_statusSim : Bool;
	END_VAR
	
	VAR_OUTPUT
	o_aceleration : Real;
	o_blockedStatus : Bool;
	o_isDualDirection : Bool;
	o_rotationDirection : "modeRotationDirection";
	o_numAct : Int;
	o_mode : "AutoManMode";
	o_status : Bool;
	o_statusSim : Bool;
	o_isAlarm : Bool;
	o_value : Real;
	END_VAR
	VAR
	an_local : "AnalogActuatorFB";
	vfd : "PowerFB";
	END_VAR
	BEGIN
	//actuador analogico asociado a VFD
	#an_local(i_resetActivation:=#i_resetActivation,
	i_mode:=#i_mode,
	i_status:=#i_status,
	i_manValue:=#i_manValue,
	i_value:=#i_value,
	i_numAct:=#i_numAct,
	i_alarmCond:=#i_alarmCond,
	i_statusSim:=#i_statusSim,
	o_numAct=>#o_numAct,
	o_mode=>#o_mode,
	o_status=>#o_status,
	o_statusSim=>#o_statusSim,
	o_isAlarm=>#o_isAlarm,
	o_value=>#o_value);
	//caracteristicas especiales de VFD
	#vfd(i_blockedStatus:=#i_blokedStatus,
	i_direction:=#i_rotationDirection,
	o_acelerationStatus=>#o_aceleration,
	o_blockedStatus=>#o_blockedStatus,
	o_direction=>#o_rotationDirection,
	o_isDualDirection=> #o_isDualDirection);
	END_FUNCTION_BLOCK

El paquete Controllers en TIA Portal.

El paquete *Controllers* esta compuesto por dos elementos *Regulator* y *PIDRegulator* las transformaciones realizadas pare estos elementos se han basado también en la utilización de FBs , no se ha creado ninguna UDT adicoinal ya que estos no poseen ningún atributo adicional.

Regulator

UDT	FB

Sin UDT	FUNCTION_BLOCK "RegulatorFB" { S7_Optimized_Access := 'TRUE' }
	VERSION : 0.1
	VAR_INPUT
	i_externalSetPoint : Real;
	i_mode_setPoint : "modeSetPoint";
	i_setPoint : Real;
	i_resetActivation : Bool;
	i_mode_actuator : "AutoManMode";
	i_status : Bool;
	i_manValue : Real;
	i_numAct : Int;
	i_alarmCondActuador : "AnalogOperator";
	i_satusSim : Bool;
	i_alarmCondSensor : "AnalogOperator";
	END_VAR
	VAR_OUTPUT
	o_externalSetPoint : Real;
	o_mode_SetPoint : "modeSetPoint";
	o_setPoint : Real;
	o_outControl : Real;
	o_numAct : Int;
	o_mode_actuator : "AutoManMode";
	o_status : Bool;
	o_statusSim : Bool;
	o_isAlarm_actuador : Bool;
	o_value_actuador : Real;
	o_value_sensor : Real;
	o_isAlarm_sensor : Bool;
	END_VAR
	VAR
	setPoint : "ControlParametersFB";
	actuator : "AnalogActuatorFB";
	sensor : "AnalogSensorFB";
	act_status : Bool;
	act_statusSim : Bool;
	sen_status : Bool;
	sen_statusSim : Bool;
	END_VAR
	BEGIN
	#sensor(i_statusSim:=#i_satusSim,
	i_status:=#i_status,
	i_cond:=#i_alarmCondSensor,
	o_value=>#o_value_sensor,
	o_isAlarm=>#o_isAlarm_sensor,
	o_status=>#sen_status,
	o_statusSim=>#sen_statusSim);
	//la lectura del sensor es la entrda del controlador
	#setPoint(i_externalSetPoint := #i_externalSetPoint,
	i_mode := #i_mode_setPoint,
	i_setPoint := #i_setPoint,
	i_outControl := #o_value_sensor,
	o_externalSetPoint => #o_externalSetPoint,
	o_mode => #o_mode_SetPoint,
	o_setPoint => #o_setPoint,
	o_outControl => #o_outControl);
	//la salida del controlador es la entrada del actuador
	#actuator(i_resetActivation := #i_resetActivation,
	i_mode := #i_mode_actuator,
	i_status := #i_status,
	i_manValue := #i_manValue,
	i_value := #o_outControl,
	i_numAct := #i_numAct,
	i_alarmCond := #i_alarmCondActuador,
	i_statusSim := #i_satusSim,
	o_numAct => #o_numAct,
	o_mode => #o_mode_actuator,
	o_status => #o_status,
	o_statusSim => #act_status,
	o_isAlarm => #act_statusSim,
	o_value => #o_value_actuador);
	#o_status := (#act_status AND #sen_status);
	#o_statusSim := (#act_statusSim AND #sen_statusSim);
	END_FUNCTION_BLOCK

PIDRegulator

UDT	FB
Sin UDT	

```
FUNCTION_BLOCK "PIDRegulator"{ S7_Optimized_Access := 'TRUE' }
VERSION : 0.1
VAR_INPUT
i_externalSetPoint : Real;
i_mode_setPoint : "modeSetPoint";
i_setPoint : Real;
i_resetActivation : Bool;
i_mode_actuator : "AutoManMode";
i_status : Bool;
i_manValue : Real;
i_numAct : Int;
i_alarmCondActuador : "AnalogOperator";
i_satusSim : Bool;
i_alarmCondSensor : "AnalogOperator";
i_DGain : Real;
i_PGain : Real;
i_IGain : Real;
END_VAR
VAR_OUTPUT
o_externalSetPoint : Real;
o_mode_SetPoint : "modeSetPoint";
o_setPoint : Real;
o_outControl : Real;
o_numAct : Int;
o_mode_actuator : "AutoManMode";
o_status : Bool;
o_statusSim : Bool;
o_isAlarm_actuador : Bool;
o_value_actuador : Real;
o_value_sensor : Real;
o_isAlarm_sensor : Bool;
o_DGain : Real;
o_PGain : Real;
o_IGain : Real;
END_VAR
VAR
actuator : "AnalogActuatorFB";
sensor : "AnalogSensorFB";
act_status : Bool;
act_statusSim : Bool;
sen_status : Bool;
sen_statusSim : Bool;
PIDParameters : "PIDControlParametersFB";
END_VAR
BEGIN
#sensor(i_statusSim:=#i_satusSim,
i_status:=#i_status,
i_cond:=#i_alarmCondSensor,
o_value=>#o_value_sensor,
o_isAlarm=>#o_isAlarm_sensor,
o_status=>#sen_status,
o_statusSim=>#sen_statusSim);
#PIDParameters(i_externalSetPoint:=#i_externalSetPoint,
i_mode:=#o_mode_SetPoint,
i_outControl:=#o_value_sensor,
i_setPoint:=#i_setPoint,
i_DGain:=#i_DGain,
i_PGain:=#i_PGain,
i_IGain:=#i_IGain,
o_externalSetPoint=>#o_externalSetPoint,
o_mode=>#o_mode_SetPoint,
o_setPoint=>#o_setPoint,
o_outControl=>#o_outControl,
o_DGain=>#o_DGain,
o_PGain=>#o_PGain,
o_IGain=>#o_IGain);
#actuator(i_resetActivation := #i_resetActivation,
i_mode := #i_mode_actuator,
i_status := #i_status,
i_manValue := #i_manValue,
i_value := #o_outControl,
i_numAct := #i_numAct,
i_alarmCond := #i_alarmCondActuador,
i_statusSim := #i_satusSim,
o_numAct => #o_numAct,
o_mode => #o_mode_actuator,
o_status => #o_status,
o_statusSim => #act_status,
o_isAlarm => #act_statusSim,
o_value => #o_value_actuador);
#o_status := (#act_status AND #sen_status);
#o_statusSim := (#act_statusSim AND #sen_statusSim);
END_FUNCTION_BLOCK
```

ÍNDICE DE FIGURAS

LISTADO DE TABLAS

BIBLIOGRAFÍA

[1] Valeriy Vyatkin. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, aug 2013. doi: 10.1109/tii.2013.2258165.

[2] Martin Hollender. *Collaborative Process Automation Systems*. ISA. International Society of Automation, 2010.

[3] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer-Verlag GmbH, 2009. ISBN 3540688986. URL `https://www.ebook.de/de/product/7392160/wolfgang_mahnke_stefan_helmut_leitner_matthias_damm_opc_unified_architecture.html`.

[4] Sebastian Rohjans, Klaus Piech, and Sebastian Lehnhoff. UML-based modeling of OPC UA address spaces for power systems. In *2013 IEEE International Workshop on Inteligent Energy Systems (IWIES)*. IEEE, nov 2013. doi: 10.1109/iwies.2013.6698587.

[5] OMG. Unified modeling language., 2018. URL `http://www.uml.org`.

[6] Jack Greenfield and Keith Short. Software factories. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03*. ACM Press, 2003. doi: 10.1145/949344.949348.

[7] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20 (5):19–25, sep 2003. doi: 10.1109/ms.2003.1231146.

[8] OPC-Foundation. OPC foundation homepage, 2018. URL `https://opcfoundation.org/`.

[9] Real Academia Real de la lengua española. Diccionario de la lengua española., 2018. URL `http://www.rae.es/`.

[10] Matthew White. The industrial revolution. Technical report, The British Library, 2009. URL `https://www.bl.uk/georgian-britain/articles/the-industrial-revolution`.

[11] Phyllis Deane. *The First Industrial Revolution*. Cambridge University, 2000.

[12] WJ Doll and MA Vonderembse. The evolution of manufacturing systems: Towards the post-industrial enterprise. *Omega*, 19(5):401–411, jan 1991. doi: 10.1016/0305-0483(91)90057-z.

[13] Aquilino Rodríguez Penin. *Sistemas SCADA*. Marcombo, 2012. ISBN 8426717810. URL https://www.ebook.de/de/product/18275921/aquilino_rodriguez_penin_sistemas_scada.html.

[14] Peter Neumann. Communication in industrial automation—what is going on? *Control Engineering Practice*, 15(11):1332–1347, nov 2007. doi: 10.1016/j.conengprac.2006.10.004.

[15] ISA. Isa 88, batch control, 1995. URL https://www.isa.org/isa88/.

[16] Jürgen Kletti. *Manufacturing Execution System - MES*. Springer Berlin Heidelberg, 2007. URL https://www.ebook.de/de/product/11429379/manufacturing_execution_system_mes.html.

[17] Jay Lee, Hung-An Kao, and Shanhu Yang. Service innovation and smart analytics for industry 4.0 and big data environment. *Procedia CIRP*, 16:3–8, 2014. doi: 10.1016/j.procir.2014.02.001.

[18] Marc Conrad Edewede Oriwoh. Things in the internet of things: Towards a definition. *International Journal of Internet of Things.*, 4(1):1–5, 2015.

[19] Marius Rosenberg Malte Brettel Niklas Friederichsen, Michael Keller. How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective. *International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering.*, 8(1):37–44, 2014.

[20] Omid Givehchi, Henning Trsek, and Juergen Jasperneite. Cloud computing for industrial automation systems - a comprehensive overview. In *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, sep 2013. doi: 10.1109/etfa.2013.6648080.

[21] AAmir Shahzad, Shahrulniza Musa, Abdulaziz Aborujilah, and Muhammad Irfan. A performance approach: SCADA system implementation within cloud computing environment. In *2013 International Conference on Advanced Computer Science Applications and Technologies*. IEEE, dec 2013. doi: 10.1109/acsat.2013.61.

[22] Stamatis Karnouskos and Armando Walter Colombo. Architecting the next generation of service-based SCADA/DCS system of systems. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, nov 2011. doi: 10.1109/iecon.2011.6119279.

[23] Pablo Gomez Perez, Wolfgang Beer, and Bernhard Dorninger. Remote rendering of industrial HMI applications. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, jul 2013. doi: 10.1109/indin.2013.6622895.

[24] Real Academia de Ciencias. Real academia de las ciencias exactas, fisicas y naturales., 2018. URL `http://www.rac.es/`.

[25] Edouard Tisserant, Laurent Bessard, and Mario de Sousa. An open source IEC 61131-3 integrated development environment. In *2007 5th IEEE International Conference on Industrial Informatics*. IEEE, jul 2007. doi: 10.1109/indin.2007.4384753.

[26] Mohamed Endi, Y Z Elhalwagy, and Attalla Hashad. Three-layer PLC/SCADA system architecture in process automation and data monitoring. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*. IEEE, feb 2010. doi: 10.1109/iccae.2010.5451799.

[27] V.C. Gungor and G.P. Hancke. Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *IEEE Transactions on Industrial Electronics*, 56(10):4258–4265, oct 2009. doi: 10.1109/tie.2009.2015754.

[28] The Modbus Organization. Modbus field bus protocols, 2018. URL `http://www.modbus.org/`.

[29] As-Interface. Asi field bus protocols, 2018. URL http://www.as-interface.com.

[30] PI Organization. Profinet field bus protocol, 2018. URL http://www.profibus.com/.

[31] IEEE 802.3 Working Group. Wireless ethernet specifications, 2018. URL http://www.ieee802.org/3/.

[32] Vicente Guerrero Jiménez, Lluís Martínez Novoa, and Ramón Luis Yuste Yuste. *Comunicaciones industriales*. Marcombo, 2010. ISBN 8426715745. URL https://www.ebook.de/de/product/13467665/vicente_guerrero_jimenez_lluis_martinez_novoa_ramon_luis_yuste_yuste_comunicaciones_industriales.html.

[33] International Organization for Standardization. Iso 9506-1:2003., 2018. URL https://www.iso.org/standard/37079.html.

[34] Larry P English. *Improving Data Warehouse and Business Information Quality: Methods for Reducing Costs and Increasing Profits*. Willey, 1999.

[35] Lian Duan and Li Da Xu. Business intelligence for enterprise systems: A survey. *IEEE Transactions on Industrial Informatics*, 8(3):679–687, aug 2012. doi: 10.1109/tii.2012.2188804.

[36] Jay Lee, Behrad Bagheri, and Hung-An Kao. Recent advances and trends of cyber-physical systems and big data analytics in industrial informatics. In *Proceeding of Int. Conference on Industrial Informatics (INDIN)*, 2014. doi: https://doi.org/10.13140/2.1.1464.1920.

[37] Dan Braha. *Data Mining for Design and Manufacturing: Methods and Applications*. Springer Science, 2001.

[38] Xiao-Lan Xie. Hierarchical production control of a flexible manufacturing system. *Applied Stochastic Models and Data Analysis*, 7(4):343–360, dec 1991. doi: 10.1002/asm.3150070405.

[39] J. Kaltwasser, A. Hercht, and R. Lang. Hierarchical control of flexible manufacturing systems. *IFAC Proceedings Volumes*, 19(2):37–44, apr 1986. doi: 10.1016/s1474-6670(17)64094-1.

[40] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '13*. ACM Press, 2013. doi: 10.1145/2430502.2430513.

[41] M.R. Anwar, O. Anwar, S.F. Shamim, and A.A. Zahid. Human machine interface using OPC (OLE for process control). In *Student Conference On Engineering, Sciences and Technology*. IEEE, 2004. doi: 10.1109/scones.2004.1564766.

[42] Al Chisholm. Opc/ole for process control overview. In *Presented at the World Batch Forum*, 1998.

[43] Li Zheng and H. Nakagawa. OPC (OLE for process control) specification and its developments. In *Proceedings of the 41st SICE Annual Conference. SICE 2002*. Soc. Instrument & Control Eng. (SICE), 2002. doi: 10.1109/sice.2002.1195286.

[44] Martin Wollschlaeger, Thilo Sauter, and Juergen Jasperneite. The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1):17–27, mar 2017. doi: 10.1109/mie.2017.2649104.

[45] A.N. Hagras. About hierarchical control systems organization. *IFAC Proceedings Volumes*, 15(9):221–222, dec 1982. doi: 10.1016/s1474-6670(17)62763-0.

[46] Mai Son and Myeong-Jae Yi. A study on OPC specifications: Perspective and challenges. In *International Forum on Strategic Technology 2010*. IEEE, oct 2010. doi: 10.1109/ifost.2010.5668110.

[47] Thuan L. Thai. *Learning DCOM*. O'Reilly, 1999.

[48] Zhao Xin Guo, Xing Quan Xie, and Zhi Gao Ni. The application of OPC DA in factory data acquisition. In *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*. IEEE, may 2012. doi: 10.1109/csae.2012.6272760.

[49] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, Luciana Moreira Sá de Souza, and Vlad Trifa. SOA-based integration of the internet of things in enterprise services. In *2009 IEEE International Conference on Web Services*. IEEE, jul 2009. doi: 10.1109/icws.2009.98.

[50] Juan Antonio Holgado Terriza Jose Miguel Gutierrez Guerrero. Mobile human machine interface based in opc ua for the control of industrial processes. In *Actas de las XXXVI Jornadas de Automatica, 2 - 4.*, 2015.

[51] J.-P. Redlich, M. Suzuki, and S. Weinstein. Distributed object technology for networking. *IEEE Communications Magazine*, 36(10):100–111, 1998. doi: 10.1109/35.722144.

[52] Vu Van Tan, Dae-Seung Yoo, and Myeong-Jae Yi. Security in automation and control systems based on OPC techniques. In *2007 International Forum on Strategic Technology*. IEEE, oct 2007. doi: 10.1109/ifost.2007.4798541.

[53] Ioan Ungurean, Nicoleta-Cristina Gaitan, and Vasile Gheorghita Gaitan. An IoT architecture for things from industrial environment. In *2014 10th International Conference on Communications (COMM)*. IEEE, may 2014. doi: 10.1109/iccomm.2014.6866713.

[54] Jose Miguel Gutierrez Guerrero and Juan Antonio Holgado Terriza. iMMAS an industrial meta-model for automation system using OPC UA. *Elektronika ir Elektrotechnika*, 23(3), jun 2017. doi: 10.5755/j01.eie.23.3.18324.

[55] Z.Q. Ding, A. Aendenroomer, H. He, and K.M. Goh. Opc based device management and communication in a distributed control application platform. In *IEEE International Conference on Industrial Informatics*. IEEE, 2003. doi: 10.1109/indin.2003.1300255.

[56] Tom Hannelius, Mikko Salmenpera, and Seppo Kuikka. Roadmap to adopting OPC UA. In *2008 6th IEEE International Conference on Industrial Informatics*. IEEE, jul 2008. doi: 10.1109/indin.2008.4618203.

[57] R. Cupek and A. Maka. OPC UA for vertical communication in logistic informatics systems. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. IEEE, sep 2010. doi: 10.1109/etfa.2010.5640978.

[58] Shuang Hua Yang, Chengwei Dai, and Roger P. Knott. Remote maintenance of control system performance over the internet. *Control Engineering Practice*, 15(5):533–544, may 2007. doi: 10.1016/j.conengprac.2006.10.006.

[59] Markus Graube, Stephan Hensel, Chris Iatrou, and Leon Urbas. Information models in OPC UA and their advantages and disadvantages. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, sep 2017. doi: 10.1109/etfa.2017.8247691.

[60] Internantional Electrotechnical Commission. Iec international standard iec 61131., 2003.

[61] ISA. The international society of automation, 1945. URL `https://www.isa.org/`.

[62] Field Device Integration. A unified vision for a smarter industry, 2015. URL `https://www.fieldcommgroup.org/`.

[63] Frank Naumann Matthias Riedl. *EDDL Electronic Device Description Language*. Vulkan Verlag GmbH, 2011. ISBN 3835631063. URL `https://www.ebook.de/de/product/5681963/matthias_riedl_frank_naumann_eddl_electronic_device_description_language.html`.

[64] FDT Group. Field device tool, 2005. URL `https://fdtgroup.org/`.

[65] OPC Foundation. Opc unified architecture specification part 1: Overview and concepts, 2012.

[66] OPC Foundation. Opc unified architecture specification part 2: Security model, 2009.

[67] OPC Foundation. Opc unified architecture specification part 3: Address space model, 2009.

[68] OPC Foundation. Opc unified architecture specification part 4: Services, 2009.

[69] OPC Foundation. Opc unified architecture specification part 6: Mappings, 2009.

[70] OPC Foundation. Opc unified architecture specification part 5: Information model, 2009.

[71] OPC Foundation. Opc unified architecture specification part 7: Profiles, 2009.

[72] OPC Foundation. Opc unified architecture specification part 8: Data access, 2008.

[73] OPC Foundation. Opc unified architecture specification part 9: Alarms & conditions, 2010.

[74] OPC Foundation. Opc unified architecture specification part 10: Programs, 2007.

[75] OPC Foundation. Opc unified architecture specification part 11: Historical acces, 2009.

[76] OPC Foundation. Opc unified architecture specification part 13: Agregations, 2012.

[77] OPC Foundation. Opc unified architecture specification part 12: Discovery, 2014.

[78] Goncalo Candido, Francois Jammes, Jose Barata de Oliveira, and Armando W. Colombo. SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications. In *2010 8th IEEE International Conference on Industrial Informatics*. IEEE, jul 2010. doi: 10.1109/indin.2010.5549676.

[79] W3C. Web services addressing (ws-addressing), 2004. URL http://www.w3.org/Submission/ws-addressing/.

[80] Florian Pauker, Thomas Frühwirth, Burkhard Kittl, and Wolfgang Kastner. A systematic approach to OPC UA information model design. *Procedia CIRP*, 57:321–326, 2016. doi: 10.1016/j.procir.2016.11.056.

[81] Miriam Schleipen, Syed-Shiraz Gilani, Tino Bischoff, and Julius Pfrommer. OPC UA & industrie 4.0 - enabling technology with high diversity and variability. *Procedia CIRP*, 57:315–320, 2016. doi: 10.1016/j.procir.2016.11.055.

[82] In-Jae Shin, Byung-Kwen Song, and Doo-Seop Eom. Auto-mapping and configuration method of IEC 61850 information model based on OPC UA. *Energies*, 9(11):901, nov 2016. doi: 10.3390/en9110901.

[83] OPC Foundation. Opc unified architecture for devices (di), 2013. URL https://opcfoundation.org/.

[84] Daniel Grossmann, Klaus Bender, and Benjamin Danzer. OPC UA based field device integration. In *2008 SICE Annual Conference*. IEEE, aug 2008. doi: 10.1109/sice.2008.4654789.

[85] Robert Henßen and Miriam Schleipen. Interoperability between OPC UA and AutomationML. *Procedia CIRP*, 25:297–304, 2014. doi: 10.1016/j.procir.2014.10.042.

[86] J Virta, I Seilonen, A Tuomi, and K Koskinen. SOA-based integration for batch process management with OPC UA and ISA-88/95. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. IEEE, sep 2010. doi: 10.1109/etfa.2010.5641286.

[87] FactoryIO. Simulación de fábrica 3d., 2005. URL https://factoryio.com.

[88] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Teoría de autómatas, lenguaje y computación*. Pearson Addison-Wesley, 2008. ISBN 8478290885. URL https://www.ebook.de/de/product/12754456/

john_hopcroft_rajeev_motwani_jeffrey_ullman_teoria_de_automatas_lenguaje_y_computacion.html.

[89] Hayrettin Toylan and Hilmi Kuscu. A research on scada application by the help of opc server for the water tank filling system. *Scientific Research and Essays Vol. 5*, 5(24):3932–3938, 2010.

[90] Ke Niu, Zhongbin Wang, Jun Liu, and Wenchuan Zhu. Application of OPC interface technology in shearer remote monitoring system. In *The 2nd International Conference on Information Science and Engineering*. IEEE, dec 2010. doi: 10.1109/icise.2010.5689892.

[91] Xu Hong and Wang Jianhua. Using standard components in automation industry: A study on OPC specification. *Computer Standards & Interfaces*, 28(4):386–395, apr 2006. doi: 10.1016/j.csi.2005.05.001.

[92] Thaddeus Czauski, Hamilton Turner, Jules White, and Sean Eade. NERD – no effort rapid development: A framework for provisioning mobile cloud industrial control applications. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, apr 2014. doi: 10.1109/mobilecloud.2014.10.

[93] Annerose Braune, Stefan Hennig, and Sebastian Hegler. Evaluation of OPC UA secure communication in web browser applications. In *2008 6th IEEE International Conference on Industrial Informatics*. IEEE, jul 2008. doi: 10.1109/indin.2008.4618370.

[94] Salvatore Cavalieri, Giovanni Cutuli, and Salvatore Monteleone. Evaluating impact of security on OPC UA performance. In *3rd International Conference on Human System Interaction*. IEEE, may 2010. doi: 10.1109/hsi.2010.5514495.

[95] M Bani Younis and G Frey. Formalization of existing plc programs : A survey. *Proccedings of CESA.*, 2003.

[96] Dong Yulin and Zheng Chunjiao. Design and research of embedded PLC development system. In *2011 3rd International Conference on Computer Research and Development*. IEEE, mar 2011. doi: 10.1109/iccrd.2011.5764286.

[97] Karl Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-12015-2.

[98] Kenwood H. Hall, Raymond J. Staron, and Alois Zoitl. Challenges to industry adoption of the IEC 61499 standard on event-based function blocks. In *5th IEEE International Conference on Industrial Informatics*. IEEE, 2007. doi: 10.1109/indin.2007.4384880.

[99] IEC. Iec international standard iec 61131-3: Programmable controllers - part 3: Programming languages. internantional electrotechnical commission., 2003.

[100] International Electrotechnical Commission. Sc 65b measurement and control devices. http://www.iec.ch, 1906. URL `http://www.iec.ch`.

[101] Michael Tiegelkamp Karl Heinz John. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010.

[102] DIN. Din standard., 2015. URL `http://www.din.de/en`.

[103] E. A. Bryan L. A. Bryan. *Programmable Controllers: Theory and Implementation*. Industrial Text Company., 1997.

[104] Telemecanique. https://www.schneider-electric.es/es/, 1836.

[105] Akshat Jain and Megha Gupta. Evolution and adoption of programming languages. *International Journal of Modern Computer Science (IJMCS)*, 5 (1), 2017.

[106] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In Springer-Verlag, editor, *European Conference on Object-Oriented Programming*, pages 220–242, 1997. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.180.2925`.

[107] Hayes-Roth, Frederick, Waterman, Donald A., Lenat, and Douglas B. *Building Expert System*. Addison-Wesley, 1986.

[108] Jacques Ferber. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman, 1999.

[109] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software for Development*. Wiley John + Sons, 2006. ISBN 0470025700. URL `https://www.ebook.de/de/product/5262389/markus_voelter_thomas_stahl_jorn_bettin_arno_haase_simon_helsen_model_driven_software_for_development.html`.

[110] Ivar Jacobson. Object-oriented development in an industrial environment. In *Conference proceedings on Object-oriented programming systems, languages and applications - OOPSLA '87*. ACM Press, 1987. doi: 10.1145/38765.38824.

[111] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002. doi: 10.1109/32.979986.

[112] Martin Obermeier, Steven Braun, and Birgit Vogel-Heuser. A model-driven approach on object-oriented PLC programming for manufacturing systems with regard to usability. *IEEE Transactions on Industrial Informatics*, 11(3): 790–800, jun 2015. doi: 10.1109/tii.2014.2346133.

[113] Per Lindgren, Marcus Lindner, Andreas Lindner, Valeriy Vyatkin, David Pereira, and Luis Miguel Pinho. A real-time semantics for the IEC 61499 standard. In *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, sep 2015. doi: 10.1109/etfa.2015.7301558.

[114] M. Bonfe and C. Fantuzzi. Design and verification of mechatronic object-oriented models for industrial control systems. In *EFTA IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.03TH8696)*. IEEE, 2003. doi: 10.1109/etfa.2003.1248708.

[115] ARC Advisory Group. Programmable logic controllers worldwide outlook. five year market analysis and technology forecast through 2013. Technical report, ARC Advisory Group., 2009.

[116] Jack Greenfield. Software factories: Assembling applications with patterns, models, frameworks, and tools. In *Software Product Lines*, pages 304–304. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-28630-1_19.

[117] Tariq Samad, Paul McLaughlin, and Joseph Lu. System architecture for process automation: Review and trends. *Journal of Process Control*, 17(3):191–201, mar 2007. doi: 10.1016/j.jprocont.2006.10.010.

[118] David Hästbacka, Timo Vepsäläinen, and Seppo Kuikka. Model-driven development of industrial process control applications. *Journal of Systems and Software*, 84(7):1100–1113, jul 2011. doi: 10.1016/j.jss.2011.01.063.

[119] K. Thramboulidis, D. Perdikis, and S. Kantas. Model driven development of distributed control applications. *The International Journal of Advanced Manufacturing Technology*, 33(3-4):233–242, apr 2006. doi: 10.1007/s00170-006-0455-0.

[120] Holger Giese and Stefan Henkler. A survey of approaches for the visual model-driven development of next generation software-intensive systems. *Journal of Visual Languages & Computing*, 17(6):528–550, dec 2006. doi: 10.1016/j.jvlc.2006.10.002.

[121] MOF. Meta object facility. omg., 2006. URL `http://www.omg.org/spec/MOF/2.0/`.

[122] Juha-Pekka Tolvanen Steven Kelly. *Domain-Specific Modeling: Enabling Full Code Generation*. JOHN WILEY & SONS INC, 2008. ISBN 0470036664. URL `https://www.ebook.de/de/product/7135970/steven_kelly_juha_pekka_tolvanen_domain_specific_modeling_enabling_full_code_generation.html`.

[123] OMG. Object management group., 2018. URL `http://www.omg.org/`.

[124] OMG. Architecture-driven modernization, 2018. URL `https://www.omg.org/adm/`.

[125] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development (Third Edition)*. 2015.

[126] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley, 2009. ISBN 0321331885. URL https://www.ebook.de/de/product/6903954/dave_steinberg_frank_budinsky_marcelo_paternostro_ed_merks_emf_eclipse_modeling_framework.html.

[127] OMG. Object constraint language (ocl), 2012. URL http://www.omg.org/spec/OCL/.

[128] Systems Modeling Language. http://www.omgsysml.org/, 1989.

[129] Jean-Philippe Babau, Mireille Blay-Fornarino, Jöel Champeau, Sylvain Robert, and Antonino Sabetta. *Model Driven Engineering for Distributed Real-Time Embedded Systems 2009: Advances, Standards, Applications and Perspectives*. ISTE LTD, 2010. ISBN 1848211155. URL https://www.ebook.de/de/product/10698251/model_driven_engineering_for_distributed_real_time_embedded_systems_2009_advances_standards_applications_and_perspectives.html.

[130] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press, 2011. doi: 10.1145/1985793.1985882.

[131] Edurne Irisarri, Marcelo V. Garcia, Federico Perez, Elisabet Estevez, and Marga Marcos. A model-based approach for process monitoring in oil production industry. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, sep 2016. doi: 10.1109/etfa.2016.7733633.

[132] Aldona Maka, Rafal Cupek, and Jakub Rosner. OPC UA object oriented model for public transportation system. In *2011 UKSim 5th European Symposium on Computer Modeling and Simulation*. IEEE, nov 2011. doi: 10.1109/ems.2011.84.

[133] Byunghun Lee, Dae-Kyoo Kim, Hyosik Yang, and Sungsoo Oh. Model trans-

formation between OPC UA and UML. *Computer Standards & Interfaces*, 50:236–250, feb 2017. doi: 10.1016/j.csi.2016.09.004.

[134] CAS. Cas.ua.modeldesigner version 3.20.1, 2016. URL `http://www.commsvr.com`.

[135] Uwe Katzke, Birgit Vogel-Heuser, and Member IEEE. COMBINING UML WITH IEC 61131-3 LANGUAGES TO PRESERVE THE USABILITY OF GRAPHICAL NOTATIONS IN THE SOFTWARE DEVELOPMENT OF COMPLEX AUTOMATION SYSTEMS. *IFAC Proceedings Volumes*, 40(16):90–94, 2007. doi: 10.3182/20070904-3-kr-2922.00016.

[136] Bernhard Werner. Object-oriented extensions for iec 61131-3. *IEEE Industrial Electronics Magazine*, 3(4):36–39, dec 2009. doi: 10.1109/mie.2009.934795.

[137] Kay Chen Tan Lingfeng Wang. *Modern Industrial Automation Software Design: Principles and Real-World Applications*. JOHN WILEY & SONS INC, 2006. ISBN 0471683736. URL `https://www.ebook.de/de/product/3269876/lingfeng_wang_kay_chen_tan_modern_industrial_automation_software_design_principles_and_real_world_applications.html`.

[138] Shaoying Liu, A.J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: a formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1):24–45, 1998. doi: 10.1109/32.663996.

[139] ISPE. Good automated manufacturing practice., 1980. URL `https://ispe.org/glossary?term=Good+Automated+Manufacturing+Practice+%28GAMP%29`.

[140] ISP. Connecting pharmaceutical knowledge, 1980. URL `https://ispe.org/`.

[141] C. Courage and K. Baxter. *Understanding your users: A practical guide to user requirements methods, tools, and techniques*. 2005.

[142] C. Wohlin. *Engineering and Managing Software Requirements*. Springer Berlin Heidelberg, 2006. URL https://www.ebook.de/de/product/8898977/engineering_and_managing_software_requirements.html.

[143] Siemens. Tia portal programming environment, 2014. URL https://w5.siemens.com/spain/web/es/industry/automatizacion/simatic/tia-portal/tia_portal/pages/tia-portal.aspx.

[144] Juerguen. A .net library for siemens s7 connectivity, 2009. URL https://github.com/S7NetPlus/s7netplus.

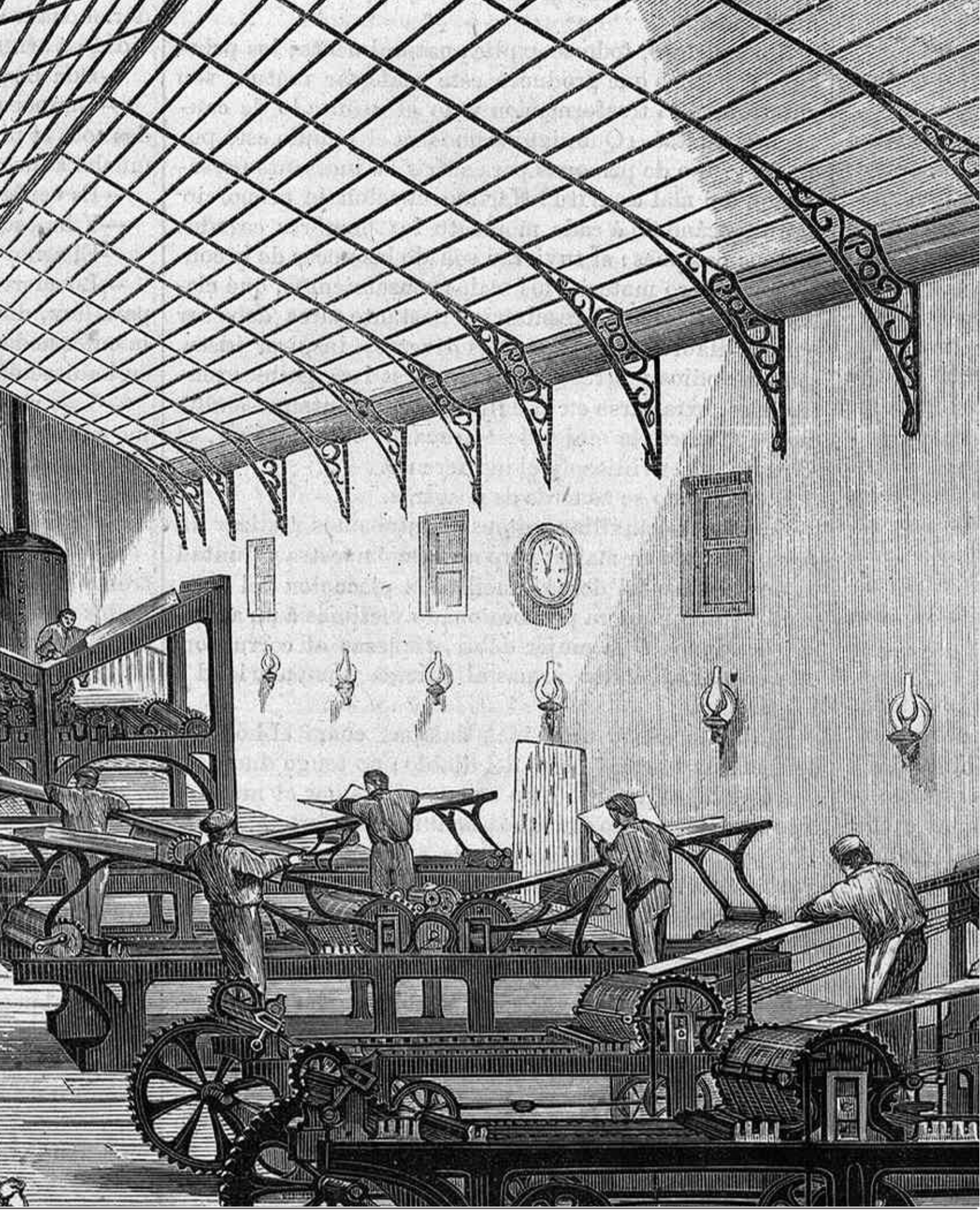
[145] ABB. Application manual rapid development guidelines for handling applications., 2018. URL http://search-ext.abb.com/library/Download.aspx?DocumentID=3HAC046417-001&LanguageCode=en&DocumentPartId=&Action=Launch.

[146] OMAC. Packml unit/machine implementation guide, 2018. URL http://omac.org/wp-content/uploads/2016/11/PackML_Unit_Machine_Implementation_Guide-V1-00.pdf.

[147] OMG. Query/view/transformation, 2016. URL http://www.omg.org/spec/QVT/.

[148] Siemens. Industry online support, 2018. URL https://support.industry.siemens.com/.

[149] Siemens. Standards compliance according to iec 61131-3. Technical report, Siemens, 2013. URL https://cache.industry.siemens.com/dl/files/938/50204938/att_78450/v1/iec_61131_compliance_e.pdf.



UNIVERSIDAD
DE GRANADA