UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Departamento de Tecnologías de la Información y las Comunicaciones

DESARROLLO DE SOFTWARE PARA ROBOTS DE SERVICIO: UN ENFOQUE DIRIGIDO POR MODELOS Y ORIENTADO A COMPONENTES

TESIS DOCTORAL

Diego Alonso Cáceres
Ingeniero Industrial

DIRECTORES
Dra. Dª Bárbara Álvarez Torres
Dr. D. Juan Ángel Pastor Franco

2008

DESARROLLO DE SOFTWARE PARA ROBOTS DE SERVICIO: UN ENFOQUE DIRIGIDO POR MODELOS Y ORIENTADO A COMPONENTES

Esta memoria se presenta en cumplimiento de los requisitos exigidos para la obtención del grado de Doctor y la acreditación de Doctorado Europeo, en el *Departamento de Tecnologías de la Información y las Comunicaciones* de la Universidad Politécnica de Cartagena. El trabajo que aquí se recoge se ha desarrollado durante el periodo comprendido entre marzo de 2.003 y octubre de 2.007 bajo la supervisión de los Doctores Dña. Bárbara Álvarez Torres y D. Juan Ángel Pastor Franco.

Agradecimientos

*Siempre es difícil recordar a todas las personas que, en un momento u otro, se han cruzado en el largo camino que hay que recorrer para realizar una Tesis Doctoral. En primer lugar, quiero agradecer a mis directores de Tesis, los doctores Dña. Bárbara Álvarez Torres y D. Juan Ángel Pastor Franco por su apoyo, supervisión y guía durante estos años. Sin su ayuda no hubiera sido capaz de llevar a cabo esta Tesis Doctoral. También quiero agradecer al Dr. D. Andrés Iborra García el haberme dado la oportunidad de integrarme y trabajar en el grupo de investigación División de Sistemas e Ingeniería Electrónica (**DSIE**) y a todos mis compañeros del Área de Lenguajes y Sistemas, especialmente a los doctores D. Pedro Sánchez Palma, D. Pedro Alcover Garau, D. Carlos Fernández Andrés y D. Juan Suardíaz Muro, por la orientación, ayuda, compañía y amistad a lo largo de estos años, así como a D. Pedro Navarro Lorente.*

Quiero dar especialmente las gracias a dos personas con las que he colaborado estrechamente para desarrollar esta Tesis Doctoral. La primera de ellas es el Dr. D. Francisco Ortiz Zaragoza, cuya Tesis sobre la arquitectura de referencia ACRoSeT ha servido de base al trabajo que se presenta. Con él y con mis directores de Tesis he compartido largas horas discutiendo sobre robótica y puliendo varios aspectos de ACRoSeT y su traducción. Y por otro lado, a la Dra. D^a Cristina Vicente Chicote, con la que comencé a recorrer, con paso lento pero seguro, los caminos del «meta-mundo», y con la que he discutido largas horas sobre V^{3}Studio. A todos ellos les doy las gracias por su amistad, guía y paciencia.

Por otro lado, también quiero darle las gracias al doctor D. Jean-Marc Jézéquel, director del grupo de investigación Triskell en el Institut de Recherche en Informatique et Systèmes Aléatoires en Rennes (Francia), y a todo su grupo, especialmente al Dr. D. Noël Plouzeau, Dr. D. Olivier Barais, Dr. D. Benoit Baudry, D. Didier Vojtisek, D. Franck Chauvel y D. Sébastien Saudrais, por haberme acogido cordialmente en Triskell durante el verano de 2.006, en el que tanto avancé en el desarrollo de esta Tesis Doctoral.

Por último, y ya fuera del ámbito académico, quiero agradecer a mi familia y mis amigos el apoyo, comprensión y cariño demostrados durante estos años. Sin su apoyo y aliento no hubiera sido capaz de acabar esta empresa. Y a Dios, porque con Él todo es posible y sin Él nada lo es.

A mis padres, por su apoyo e inspiración.

Resumen

Esta Tesis Doctoral continúa la línea de investigación en el desarrollo de sistemas robóticos teleoperados iniciada por el grupo de investigación **DSIE** de la Universidad Politécnica de Cartagena hace diez años, y parte de los resultados de otra Tesis anterior, en la que se desarrolló una arquitectura de referencia para el control de robots de servicio teleoperados denominada ACRoSeT. ACRoSeT propone una serie de componentes independientes de la plataforma de ejecución para diseñar la aplicación de control de un robot. Esta independencia permite reutilizar componentes en distintas aplicaciones y traducir un mismo diseño a distintos lenguajes de programación o utilizar diferentes frameworks robóticos. ACRoSeT, sin embargo, no proporciona ninguna guía para realizar la traducción de los componentes abstractos que forman la aplicación de control a código ejecutable.

Esta Tesis Doctoral completa el enfoque propuesto por ACRoSeT utilizando el novedoso *desarrollo dirigido por modelos* (Model-Driven Engineering, **MDE**) para (1) proponer una solución a la ambigüedad semántica y de implementación de algunos de los conceptos propuestos por ACRoSeT; (2) proporcionar una serie de herramientas para aumentar el nivel de abstracción del desarrollador; (3) cambiar el proceso de traducción manual de los componentes abstractos a código ejecutable por un proceso (semi-) automático que elimine errores en la implementación final y (4) permitir que el proceso de traducción pueda ser extendido a diversos lenguajes de programación e incluso frameworks robóticos.

Para lograr estos objetivos se ha desarrollado (1) un meta-modelo de componentes, denominado V^{3}Studio, que contiene los conceptos del dominio de la robótica definidos por ACRoSeT; (2) una transformación entre V^{3}Studio y UML que permite reducir la distancia semántica entre el diseño realizado y la implementación final, facilitando de este modo el desarrollo posterior de distintas implementaciones en otros lenguajes de programación y (3) una traducción entre el modelo UML y código, en este caso, Ada 2005. Como demostración de la viabilidad del enfoque, esta Tesis Doctoral incluye el desarrollo del sistema de control de una mesa cartesiana, realizado en el proyecto del VI Programa Marco Europeo **EFTCoR**.

Abstract

This Thesis continues the research line in the development of teleoperated robotic systems initiated by the **DSIE** research group of the Technical University of Cartagena ten years ago and starts from the results of a previous Thesis, in which a reference architecture for teleoperated robotic systems, called ACRoSeT, was designed. ACRoSeT proposes a series of platform independent components for designing the software control architecture of a robot. This independence allows not only the reuse of these components among different applications but also the translation to different programming languages or using different robotic frameworks. But ACRoSeT does not offer any guide regarding the software implementation of the concepts it defines.

The present Thesis completes the design of ACRoSeT using the newly *Model-Driven Engineering* (**MDE**) paradigm to (1) propose a solution to the semantic and implementation ambiguity of some of the concepts defined by ACRoSeT; (2) raise the level of abstraction used by the developer to design the software architecture of a robot using the concepts defined by ACRoSeT; (3) change the error-prone process of manual translation from the abstract ACRoSeT components to executable code by a (semi-) automatic process that erases this source of errors from the final implementation; and (4) allow different implementations in different programming languages and even robotic frameworks.

To achieve these objectives, this Thesis proposes (1) a component meta-model, called V^{3}Studio, that contains all the concepts of the robotics domain defined by ACRoSeT; (2) a model to model transformation between V^{3}Studio and UML that reduces the semantic distance between the component model and the final code implementation, and that also eases the development of different transformations to different implementation languages; and (3) a model to text transformation to translate the UML model to the Ada 2005 programming language. To demonstrate the viability of the proposed approach, this Thesis includes the development of the control system of a XYZ cartesian robot, designed in the context of the VI European Growth Programm **EFTCoR**.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

CAPÍTULO 1

PLANTEAMIENTO Y OBJETIVOS

Lo hicimos porque nadie nos dijo que era imposible.

ANÓNIMO

1.1 INTRODUCCIÓN

EUROPA, siglo XVIII. El sector de la manufactura industrial se encuentra en un callejón sin salida. Cada unidad de cada producto es fabricada a mano por uno o varios artesanos siguiendo unos planos de fabricación. Esta situación se volvió insostenible para la naciente industria, sobre todo en una Europa marcada por las continuas guerras. En aquel momento la industria de la fabricación de armas de fuego tomó la iniciativa y cambió el modelo de fabricación artesanal existente hasta entonces. En 1.790 Honoré Blanc comenzó a desarrollar mosquetes que incluían algunas partes intercambiables. Dieciocho años después, el americano Eli Whitney, inventor de la máquina para desgranar algodón, introdujo el concepto de «bloque constructivo predefinido» en el proceso de manufacturación de armas de fuego para el gobierno americano.

De esta forma comenzó el proceso de estandarización de componentes que revolucionó el sector de la producción industrial. Gracias a que los productos se diseñan partiendo de un conjunto de piezas (estandarizadas), cada unidad puede construirse rápidamente, sin más que ensamblar las piezas de que está compuesta. Esta forma de diseñar permite, además, analizar el comportamiento esperado del producto incluso antes de que comience

su fabricación y facilita enormemente la reparación, ya que tan solo hay que sustituir el componente dañado por uno nuevo. Henry Ford llevó este concepto un paso más allá cuando, a principios del siglo XX, comenzó a fabricar en serie su Ford *modelo T*, combinando el concepto de fabricación modular con el de línea de producción. Había comenzado la era de la producción industrial en serie.

Al igual que sucedió con el sector de la producción industrial a principios del siglo XVIII, la producción de software atravesó su propia crisis a mediados del siglo XX. En aquel entonces, ni las metodologías de desarrollo (básicamente inexistentes) ni los lenguajes de programación (principalmente ensamblador) podían abordar la creciente complejidad de los problemas que se le planteaban. Además, muchos de estos problemas requerían la utilización de sistemas heterogéneos en hardware (e incluso software), lo cual dificultaba aún más, si cabe, la obtención de soluciones. Edsger Dijkstra publicó en 1.972 un artículo en que enunciaba este problema [73]:

> *[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

Unos años antes de que Dijkstra escribiera este artículo, Douglas McIlroy hizo las siguientes declaraciones en una conferencia en la *Organización del Tratado del Atlántico Norte* (OTAN), en la que insta a los ingenieros del software a desarrollar programas basados en partes predefinidas [142]:

> *Software engineers should take a look at the hardware industry and establish a software component subindustry. The produced software components should not be tailored to specific needs but be reusable in many software systems.*

Como evolución lógica de sus palabras, McIlroy predijo el nacimiento de una industria de compra/venta de productos software, imitando a la industria existente en el mundo mecánico y electrónico. En 1.986, Brad Cox dió forma al concepto moderno de componente software, asemejándolo al de componente electrónico (de hecho, los denominó *Software ICs*, circuito integrado software). Cox comenzó el desarrollo de una infraestructura y un mercado para estos componentes con el malogrado lenguaje de programación *Objective-C*.

La «crisis del software» se manifestó en forma de proyectos gestionados por encima del presupuesto o del tiempo establecido, software de baja calidad (poco fiable, difícil de modificar o mantener), que no satisfacía los requisitos del cliente, etc. Las soluciones a

estos problemas pasaban por el diseño de nuevas metodologías para afrontar el proceso de desarrollo software; el desarrollo de nuevos lenguajes, con mayor expresividad; y de mecanismos que permitieran reutilizar el trabajo ya realizado. Es decir, se hizo patente la necesidad de disponer de un conocimiento que permitiera abordar los retos planteados en el desarrollo de software. Con estos objetivos nació la *Ingeniería del Software* (término acuñado en 1.968 por F.L. Bauer, presidente de la conferencia de la OTAN sobre software). Según el Standard 610.12 del Institute of Electrical and Electronics Engineers (**IEEE**), la ingeniería del software puede definirse como:

> *(1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software.*
>
> *(2) the study of approaches as in (1).*

A pesar del desarrollo de diversas metodologías de diseño software (entre los que destacan el *modelo incremental* y el *desarrollo en espiral* [187], y el reciente *Proceso Unificado de Desarrollo Software* [122]), lenguajes de tercera generación (orientados a objetos) o del Lenguaje Unificado de Modelado [170], el proceso de desarrollo software y su calidad final no ha mejorado mucho. O por lo menos no ha evolucionado a la misma velocidad que la complejidad de los problemas que pretende solucionar.

De hecho, no hay gran diferencia entre los bucles de repetición del primitivo Fortran y los del moderno Java, aunque sí que se ha avanzado mucho en el desarrollo de compiladores. También se ha demostrado que los objetos, contrariamente a lo que se esperaba, no son la panacea del diseño y la reutilización software [186]. La encapsulación, el polimorfismo y la composición se han mostrado como armas muy poderosas y realmente han mejorado los diseños, pero la mejora no ha sido tan grande como se esperaba ni la reutilización de código ha alcanzado la cuota prevista. El problema fundamental reside en que se sigue trabajando a pequeña escala, en lo que los anglosajones denominan *programming in the small*, cuando los problemas aparecen, principalmente, al tener que reutilizar artefactos software desarrollados con anterioridad de forma diferente a como fueron diseñados [90]. Durante estos últimos años hemos sido testigos de, más que avances, evoluciones en la concepción y diseño del software, pero la revolución está aún por llegar.

La revolución llegará cuando se realicen diseños a un mayor nivel de abstracción que el que proporcionan los lenguajes de programación, demasiado cercanos a la máquina. Si se quiere abordar con garantías de éxito los nuevos problemas que se plantean hoy en día se necesita un giro en la forma de entender el desarrollo software. Se necesita un nuevo salto en la abstracción usada para desarrollar los programas. Con estos objetivos apareció el *Desarrollo Basado en Modelos* (Model Driven Engineering, MDE) [49, 128, 191].

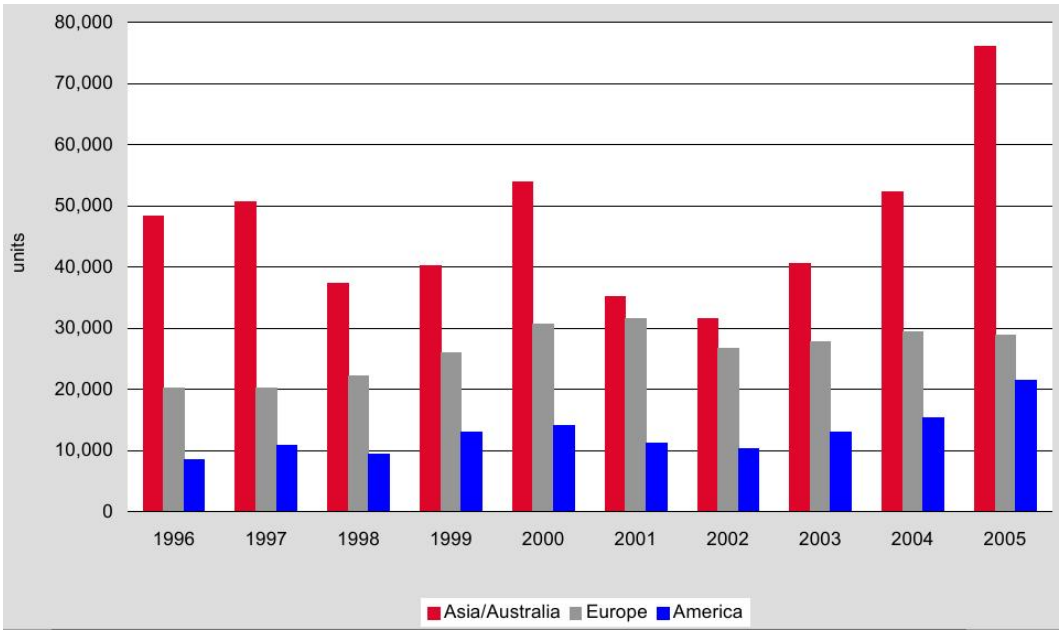
1.2 MOTIVACIÓN Y JUSTIFICACIÓN

DESDE que el hombre evolucionó a *homo sapiens* ha tratado de diseñar artefactos mecánicos que hicieran su trabajo o que, por lo menos, le ayudaran a realizarlo más fácil y rápidamente. Tras la revolución tecnológica producida a lo largo del siglo XX, el desarrollo de este tipo de sistemas, así como su complejidad y funcionalidad, se ha incrementado progresivamente. Actualmente se desarrollan sistemas robóticos y de inspección visual para realizar tareas repetitivas y monótonas (p. ej. inspección de la calidad superficial de la cerámica), en entornos peligrosos (e.g. centrales nucleares) o inalcanzables (e.g. exploración espacial). En todos estos casos, la contratación y formación de personal representa un importante desembolso económico (tanto monetario como en tiempo) para la empresa, lo que influye negativamente en su capacidad productiva. Es en este punto donde los sistemas automáticos muestran su verdadero potencial para reducir costes y mejorar el tipo de trabajo que realizan los trabajadores de la empresa.

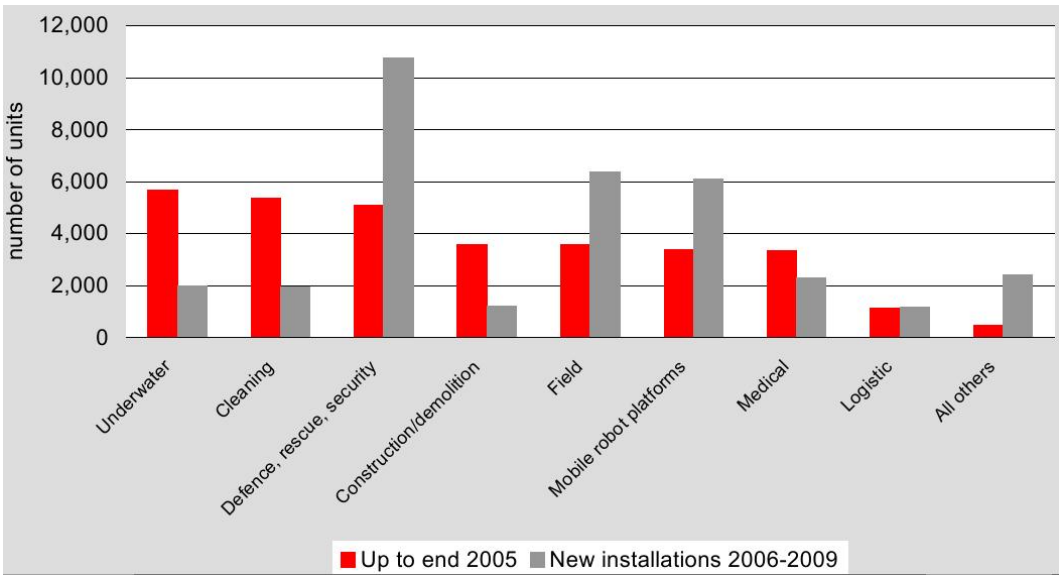
En un mundo que evoluciona cada vez más rápido y que es más competitivo, la necesidad de desarrollar soluciones integradas de automatización más fiables, con más funcionalidad y más rápidamente, para controlar los procesos industriales es cada vez mayor. En estas condiciones, es posible afirmar que la robótica es una de las áreas con mayor proyección de futuro y que mayores beneficios puede reportar. El informe[1] realizado por el IFR (*International Federation of Robotics*) para la UNECE (*United Nations Economic Commission for Europe*) en 2.006 así lo pone de manifiesto, y estima que el número total de robots en el mundo en 2.005 es de unas 923.000 unidades, un 9 % más que en 2.004, tal y como muestran las gráficas de la figura 1.1. Esta área de investigación y desarrollo es tan importante, que el *holding* de empresas inglés BT le ha dedicado una sección propia en su prospección de tecnologías futuras *BT 2.005 Technology Timeline*[2], mostrada en el cuadro 1.1. En un mundo que evoluciona cada vez más rápido y que es más competitivo, la necesidad de desarrollar soluciones integradas de automatización más fiables, con más funcionalidad y más rápidamente, para controlar los procesos industriales es cada vez mayor. En estas condiciones, es posible afirmar que la robótica es una de las áreas con mayor proyección de futuro y que mayores beneficios puede reportar. El informe[3] realizado por el IFR (*International Federation of Robotics*) para la UNECE (*United Nations Economic Commission for Europe*) en 2.006 así lo pone de manifiesto.

A nivel nacional, el «Libro Blanco de la Robótica» es la fuente más fiable sobre el estado de esta disciplina en España. Desarrollado por el Comité Español de Automática

[1] http://www.ifr.org/publications/World_Robotics.htm
[2] http://btplc.com/Innovation/News/timeline.htm
[3] http://www.ifr.org/publications/World_Robotics.htm



(a) Robots instalados en el mundo por año



(b) Robots de servicio para uso profesional clasificados por sectores

Figura 1.1: Estadísticas de la población de robots en el mundo (*World Robotics 2.006*)

ROBOTICS	
Totally automated factories	2006-2010
Global domestic robot numbers passes 4M	2006-2010
Global industrial robot numbers passes 1M	2006-2010
Fractal shape-changing robots	2008-2012
Insect-like robots used in warfare	2008-2012
Robotic dolls and pets account for 10 % of domestic telecomm traffic	2008-2012
Self monitoring infrastructures using smart materials and sensors	2008-2012
Micromechanical gnomes	2008-2012
Robots for cleaning, washing fetch and carry in hospital	2008-2012
Robot dance tutors	2011-2015
Nanowalkers, nanoworms, nanofish	2011-2015
Mechanical intelligence using MEMS and NEMS	2011-2015
Android robots used for factory jobs	2011-2015
Fleet of garden robots for plant and lawn care and tidying	2011-2015
Robots for cleaning, washing fetch and carry, in office	2011-2015
Robot pest killers	2011-2015
Housework robots - fetch, carry, clean & tidy, organise etc.	2013-2017
Robots for guiding blind people	2013-2017
Cybernetic use in sports	2013-2017
Robots for cleaning, washing, fetch and carry, in home	2013-2017
Self diagnostic self repairing robots	2016-2020
Actuators resembling human muscles	2016-2020
Insect sized robots banned in gardens due to effects on wildlife	2020s
Robotic delivery for internal mail	2020s
Robotic exercise companion	2020s
More robots than people in developed countries	2020s
Android gladiators	2020s
GM and robotics converge, GM used to make organic robots	2020s
Micro-Mechano fractal construction kit	2030s
i-Robot style robots with polymer muscles and strong AI	2040s

Cuadro 1.1: Prospección sobre el futuro de la robótica

(**CEA**) y avalado por la Red Nacional de Robótica del Ministerio de Educación y Ciencia, este Libro Blanco analiza el desarrollo del sector en España y las oportunidades de futuro que presenta. Según este libro, España ocupa el séptimo lugar en el mundo y el cuarto en Europa por número de robots instalados, con cerca de 22.000 unidades. Además, como muestra la figura 1.2, España tiene un enorme potencial investigador en robótica, al contar con más de 60 grupos de investigación que desarrollan su actividad en diversas áreas de la robótica y que tienen presencia en los congresos internacionales más importantes. Aunque, desgraciadamente, este potencial no es aprovechado por las empresas nacionales, que todavía contemplan con recelo este sector de negocio y no se deciden a invertir en él con decisión.

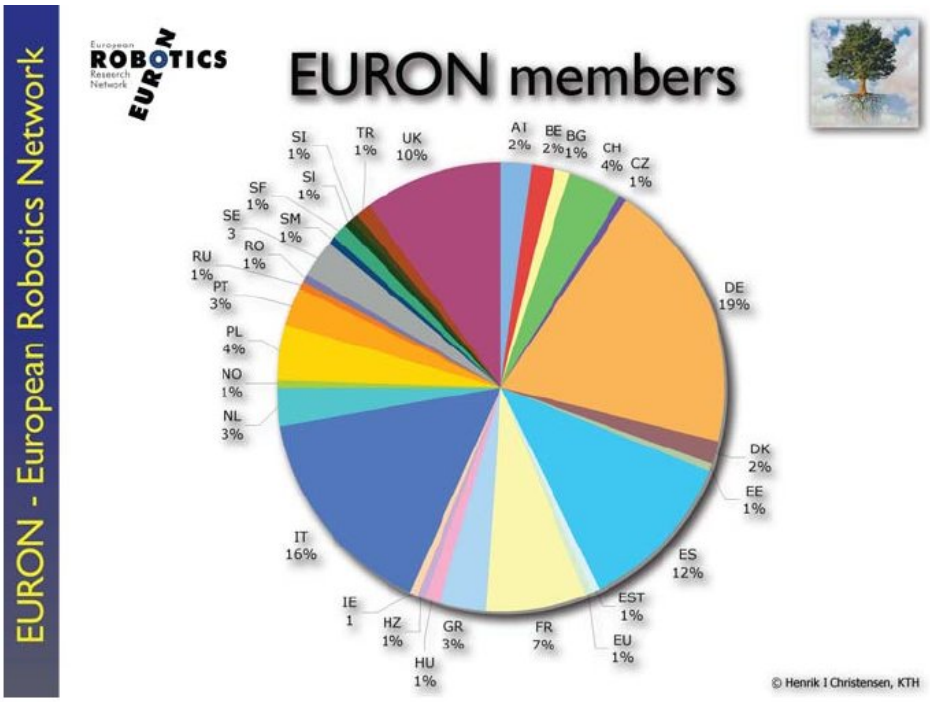


Figura 1.2: Distribución de grupos de investigación en Robótica en Europa

A pesar de que la robótica ha sido sujeto tradicional de investigación y se ha avanzado mucho en el conocimiento y desarrollo de estos sistemas, su diseño sigue planteando muchos retos, tanto en el diseño hardware de los sistemas como en el software y en la integración de ambos. Además, el hecho de ser un área pluridisciplinar le añade gran riqueza y una dificultad extra, ya que todo avance en cada una de las áreas que la componen repercute positivamente en el desarrollo de estos sistemas. Por tanto, un avance tan importante y prometedor como MDE augura una revolución en la concepción y diseño del software de control para robots de servicio.

Desde que Karel Capek acuñara en 1.921 el término checo «robota» (que significa esclavitud, servidumbre), del que posteriormente derivó la palabra «robot», el hombre ha soñado con alcanzar el nivel tecnológico necesario para fabricar estos artefactos y liberarse de la carga del trabajo diario. Sin embargo, este conocimiento le está resultando esquivo, a pesar de llevar varias décadas investigando en este campo. El término «robot» tiene muchos significados hoy en día, y todos ellos están en mayor o menor medida influidos por la literatura y el cine. Si bien es cierto que estos medios han contribuido sobremanera a su estudio y desarrollo, es igualmente cierto que la capacidad que le atribuyen está muy alejada de la capacidad real que tienen estos dispositivos actualmente.

La Tesis Doctoral que aquí se expone se enmarca dentro de los trabajos realizados por el grupo de investigación *División de Sistemas e Ingeniería Electrónica* (**DSIE**[4]) de la Universidad Politécnica de Cartagena (**UPCT**). El DSIE nació en 1.999 como grupo de investigación multidisciplinar e integra profesores e investigadores de los departamentos de Tecnología Electrónica (**DTE**), Ingeniería de Sistemas y Automática (**DISA**) y Tecnologías de la Información y las Comunicaciones (**TIC**) de la UPCT. El DSIE desarrolla su labor de investigación en las cinco áreas tecnológicas que se enumeran a continuación:

- Sistemas de control y robótica para aplicaciones industriales.
- Robots de servicio.
- Sistemas de inspección visual automatizados.
- Tecnología electrónica para robótica y visión artificial.
- Redes de sensores (*Wireless Sensor Network*).

Esta Tesis Doctoral profundiza en una de las líneas de investigación principales del grupo DSIE: la robótica de servicio. Esta línea fue abierta por la Dra. D^{a}. Bárbara Álvarez Torres, en cuya Tesis Doctoral desarrolló un arquitectura de referencia para el control de robots teleoperados [226], y continuada posteriormente por el Dr. D. Juan Ángel Pastor Franco, quien evaluó dicha arquitectura utilizando métodos semi-formales [185].

Más recientemente, la Tesis Doctoral del Dr. D. Francisco Ortiz Zaragoza [181] aporta los últimos resultados de investigación en este campo. En ella define ACRoSeT(*Arquitectura de Control de Robots de Servicio Teleoperados*), una arquitectura de referencia para diseñar el sistema de control de robots teleoperados, como continuación del trabajo realizado previamente por el grupo. ACRoSeT define el sistema de control de un robot mediante componentes software/hardware con interfaces claras y bien definidas, buscando ante todo la reutilización de componentes entre diversos sistemas robóticos, de forma que sea fácil amoldarse a distintas combinaciones hardware/software.

En el resto de este apartado se realiza un breve resumen de los principales proyectos de investigación abordados por el grupo DSIE relacionados con la robótica, marco de realización de las Tesis Doctoral anteriormente comentadas. Por último, se presentan los dos proyectos de investigación más actuales en los que está involucrado el grupo y que han servido de marco de realización de esta Tesis Doctoral.

[4] `http://www.dsie.upct.es`

1.2.1 Sistemas robóticos en el DSIE

El DSIE ha acumulado una considerable experiencia en el desarrollo de sistemas robóticos desde que comenzara su labor a mediados de los años noventa. La robótica de servicio, como se ha mostrado al principio de la presente sección, es un área de investigación pluridisciplinar con gran proyección de futuro. Entre el amplio abanico de posibilidades destacan aplicaciones como la realización de tareas de inspección, vigilancia, seguridad, limpieza, agricultura, etc. De acuerdo a la *International Federation of Robotics* (**IFR**), un robot de servicio es

> *[a service robot is] a robot which operates semi or fully autonomously to perform services useful to the well being of humans and equipment, excluding manufacturing operations.*

El desarrollo de robots de servicio es un área de investigación que ha tenido gran interés para el DSIE, que ha participado en el desarrollo de numerosos proyectos. Estos proyectos han tenido dos grandes marcos de trabajo, claramente diferenciados entre sí: en una primera época se desarrollaron robots teleoperados para realizar operaciones de mantenimiento en el interior de centrales nucleares, mientras que posteriormente se diseñó una familia de robots para realizar operaciones de mantenimiento en astilleros. En ambos casos se desarrolló una arquitectura para guiar el diseño y la construcción de la plataforma de teleoperación o de la unidad de control remota: la Tesis Doctoral de la Dra. Álvarez [226] en el caso de los robots para centrales nucleares y ACRoSeT [181] en el de los robots para astilleros.

A continuación se enumeran los proyectos relacionados con el desarrollo de robots para realizar diversas tareas de mantenimiento dentro de una central nuclear, cuya unidad de teleoperación fue desarrollada siguiendo la arquitectura diseñada por la Dra. Álvarez. Consulte [117] para ampliar la información sobre estos robots.

ROSA (Remotely Operated Service Arm): desarrollo de un sistema para realizar labores de mantenimiento de la placa de tubos de los generadores de vapor de una central nuclear (PIE-041049, PAUTA 753/95 y PAUTA 53/96, Ministerio de Industria). La arquitectura de la Dra. Álvarez fue reutilizada para el control de cada una de las herramientas de mantenimiento con que iba equipado el robot. La figura 1.3 muestra el diseño CAD y una fotografía del robot ROSA.

IRV (Inspection Retrieving Vehicle): desarrollo de un sistema de recogida de objetos caídos desde la caja de aguas a la tobera del primario. Como muestra la figura 1.4 (a), el IRV es un robot teleoperado equipado con luces, cámara de visión, diversos sensores y con cabezal para distintas herramientas, y fue diseñado para trabajar en sumergido

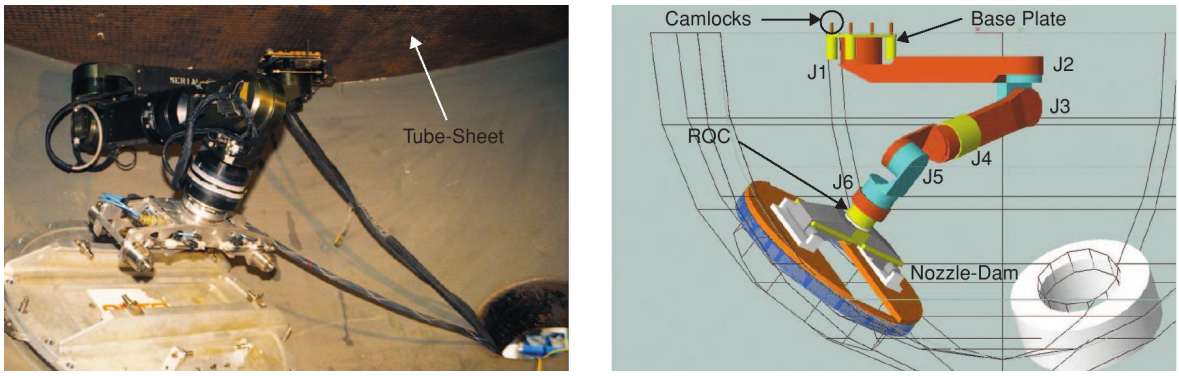


Figura 1.3: Robot ROSA para el mantenimiento de generadores de vapor

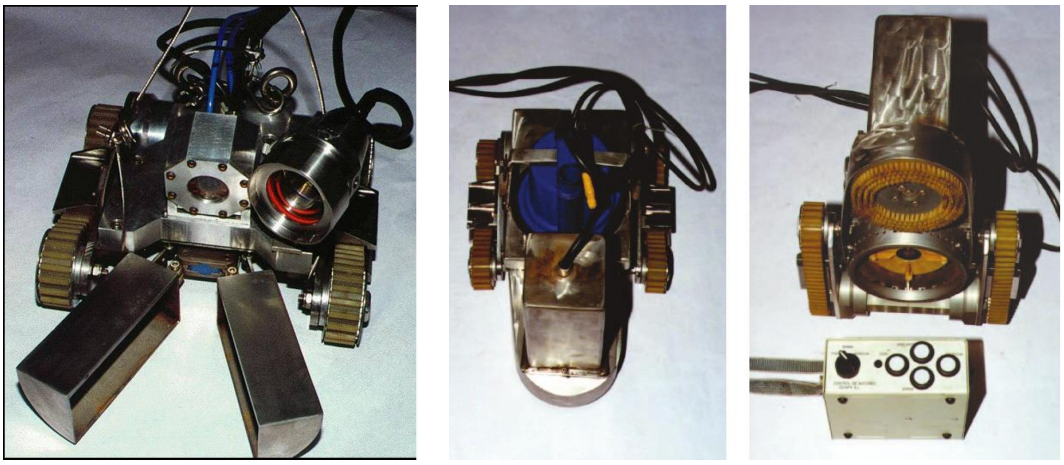
hasta a 10 m de profundidad. Este proyecto permitió validar la arquitectura para diseñar el sistema de control del vehículo teleoperado [184].

CRV (Cleaning and Retrieving Vehicle): desarrollo de un robot para aspirar pequeños charcos de fuel, depositados como consecuencia de las maniobras para repostar. El diseño del vehículo y del software de control está basado en gran medida en el IRV, pero va equipado con una bomba de aspiración y un cepillo rotativo. La figura 1.4 (b) muestra el robot móvil CRV.

TRON (Teleoperated and Robotized System for Maintenance Operation in Nuclear Power Plants Vessels): diseño de un robot destinado a la recuperación de objetos en el fondo de la vasija del reactor (EUREKA EU1565, 1996-1998). Permitió, tal y como se describe en [118], el traslado de la arquitectura a otras plataformas y demostró que era posible reutilizar los componentes genéricos desarrollados, en Ada'83, para el sistema ROSA. En la figura 1.5 se puede ver el robot TRON instalado en una maqueta de la vasija del reactor.

Durante la realización de cada uno de estos proyectos se pudo comprobar que el desarrollo de software de calidad, que facilite la modificación y el mantenimiento de los diferentes sistemas, tiene una importancia capital, no sólo en el desarrollo de los mismos sino en su posterior labor de mantenimiento. Sin embargo, en todos los casos la unidad de control fue diseñada *ad-hoc* para la aplicación, lo que aumentó considerablemente el coste de desarrollo. Los resultados obtenidos durante la realización de los proyectos anteriores pueden consultarse en la bibliografía [8, 118, 221, 222, 223].

En el momento en que se abordó el problema de diseñar una familia de robots para limpieza de barcos se detectaron dos grandes problemas que, sumados al hecho de que la unidad de control era diseñada *ad-hoc*, hicieron inviable la utilización de la arquitectura de la Dra. Álvarez en este nuevo entorno. Estos problemas surgieron porque



(a) Robot IRV (b) Robot CRV

Figura 1.4: Robots de recogida de objetos en una central nuclear

Figura 1.5: Maqueta del robot TRON

dicha arquitectura fue desarrollada para robots que trabajan en entornos estructurados, lo que dio lugar a una arquitectura muy rígida, con patrones de comunicación preestablecidos y poco flexibles. Estas limitaciones impulsaron la investigación en el desarrollo de una nueva arquitectura: ACRoSeT(*Arquitectura de Control de Robots de Servicio Teleoperados*) [181]. ACRoSeT fue desarrollada y validada en los siguientes proyectos:

GOYA (Robot escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente): desarrollo de un robot para la limpieza de barcos que fuera capaz de obtener un grado de rugosidad SA2($\frac{1}{2}$) en la superficie del acero y que reduzca el consumo de granalla y las emisiones de polvo y residuos (FEDER TAP IFD97-0823, 1999-2001). En este proyecto se iniciaron los estudios para proponer una arquitectura de referencia para el desarrollo de la unidad de control local que, en este caso, fue implementada en un PC industrial [179, 224], tal y como se muestra en la figura 1.6.

EFTCoR (Environmental friendly and cost-effective technology for coating removal): desarrollo de un sistema de limpieza automatizada de la superficie de un buque y del sistema de recogida y reciclaje de residuos[5] (Proyecto GROWTH del V Programa Marco de la Unión Europea G3RD-CT-2002-00794 y CICYT DPI2002-11583-E, 2002-2005). Abordado tras el desarrollo del GOYA, este proyecto diseña una familia de robots para llevar a cabo dos tipos de limpieza: *spotting* (limpieza de pequeñas áreas aisladas del barco) y *full blasting* (limpieza de grandes áreas). Los robots para realizar la primera operación tienen que ser capaces de moverse a lo largo de la superficie del barco (ver figura 1.7), mientras que la capacidad principal de los segundos tiene que ser el área abarcada con la herramienta de limpieza (ver figura 1.8). Para más información, puede consultar [9, 89, 178].

[5] http://www.eftcor.com

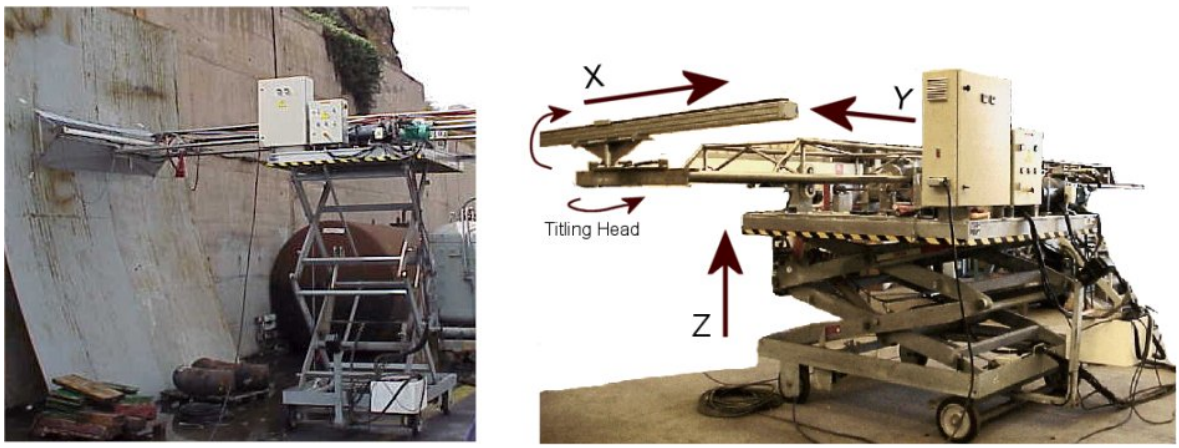
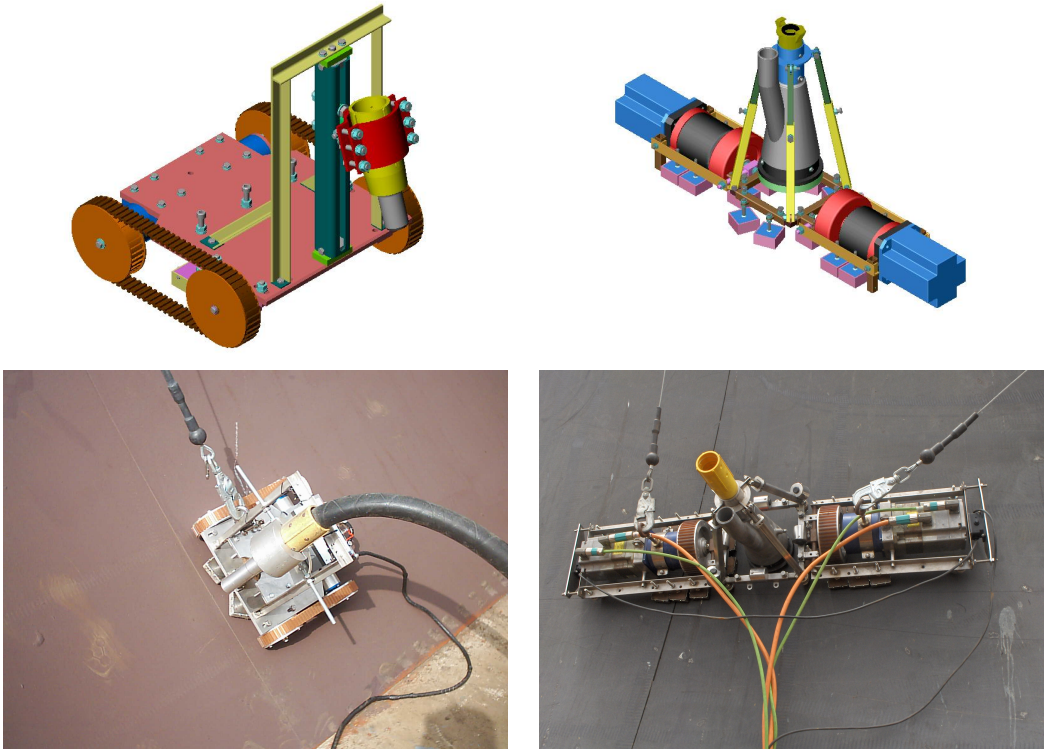


Figura 1.6: Prototipo de robot de limpieza para superficies de barco GOYA



(a) Versión 1 (b) Versión 2

Figura 1.7: EFTCoR: robots de limpieza trepadores para *spotting*

(a) Mesa XYZ con cabezal (b) Grúa de limpieza

Figura 1.8: EFCToR: robot de limpieza *full blasting*

1.2.2 MARCO DE DESARROLLO DE LA TESIS DOCTORAL

El desarrollo de la presente Tesis Doctoral comenzó durante el proyecto ANCLA (*Arquitecturas dinámicas para sistemas de teleoperación*), CICYT TIC2003-07804-C05-02. ANCLA es uno de los subproyectos coordinados por el proyecto DYNAMICA (*DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures*), desarrollado entre los años 2003-2006. El propósito inicial del proyecto ANCLA era resolver la problemática ligada al desarrollo de software para los sistemas robóticos de teleoperación, aprovechando en todo lo posible los últimos avances de la ingeniería del software, en especial el desarrollo basado en componentes y las arquitecturas software. ANCLA culminó con el desarrollo de la arquitectura ACRoSeT para control de robots de servicio teleoperados.

Actualmente el grupo DSIE se encuentra embarcado en el proyecto MEDWSA (*Marco conceptual y tecnológico para el desarrollo de software de sistemas reactivos*), CICYT TIN2006-15175-CO5-02. MEDWSA es uno de los subproyectos del proyecto coordinado META (*Models, Environments, Transformations and Applications*), que se desarrollará entre los años 2007-2009. El objetivo del proyecto MEDWSA es la definición de un marco conceptual y tecnológico para el desarrollo de sistemas reactivos, basado en líneas de producto, que aproveche las ventajas de las tendencias actuales del desarrollo dirigido por modelos. A la vista del objeto de la presente Tesis Doctoral, MEDWSA supone un marco de trabajo ideal para finalizar su desarrollo y validarla.

1.3 OBJETIVOS DE LA TESIS

EL OBJETIVO perseguido con la elaboración de la presente Tesis Doctoral **es mejorar el proceso de diseño y la calidad final del software de control de robots de servicio**, una de las principales líneas de investigación del grupo de investigación *División de Sistemas e Ingeniería Electrónica* (DSIE). La Tesis Doctoral parte de un resultado anterior del grupo: el marco arquitectónico basado en componentes ACRoSeT, obtenido recientemente como resultado de la Tesis Doctoral del Dr. Ortiz [181].

La robótica es un área multidisciplinar, cuya característica principal es que desarrolla sistemas que tienen que desenvolverse en el mundo real, y como tal tienen restricciones extras, como son características de tiempo-real y tolerancia a fallos. La robótica se encuentra, por tanto, entre dos mundos, el mundo del diseño software y el del diseño mecánico-electrónico (también denominado por algunos autores «mecatrónica»), y hereda los problemas asociados a ambos. Dentro del mundo del desarrollo software, el desarrollo basado en componentes [201] (Component-Based Development (**CBD**)) surge con el

propósito de acelerar el proceso de desarrollo software, al promulgar que el software debería realizarse mediante la unión de piezas prediseñadas, al igual que sucede en el mundo mecánico y electrónico. Hasta el momento, sin embargo, este enfoque no ha sido del todo fructífero, ya que no se ha conseguido desarrollar un verdadero lenguaje que permita crear y ensamblar componentes, sino que se han diseñado distintos *modelos de componentes*, que definen un conjunto de estándares y convenciones para el desarrollo, utilización y evolución de los mismos [66]. A pesar de ello, **el desarrollo basado en componentes desempeña un papel fundamental** en ACRoSeT y, por ende **en el desarrollo de este trabajo**.

Para lograr el objetivo marcado por la Tesis Doctoral **se va a aplicar el novedoso enfoque de desarrollo o ingeniería software basada en modelos** (Model-Driven Engineering (**MDE**)) [194]. MDE promete revolucionar la forma en que se realiza el desarrollo software. La utilización de modelos es una técnica que otras áreas de conocimiento llevan aplicando con éxito durante mucho tiempo, ya que aumenta el nivel de abstracción y la expresividad con que se plantean tanto los problemas como las soluciones. MDE tiene una máxima simple: «*todo es un modelo*» [49]. El otro pilar fundamental del desarrollo basado en modelos son las *transformaciones de modelos* [196] que, como su nombre indica, permiten convertir un modelo en otro diferente o a una representación textual, por ejemplo código. De esta manera los modelos representan distintas vistas estáticas de un sistema mientras que las transformaciones de modelos permiten pasar de una vista a otra.

Se espera que, gracias a la aplicación del enfoque MDE, los mundos de la robótica y la orientación a componentes puedan beneficiarse del aumento de nivel de abstracción que proporciona dicho enfoque, haciendo por fin viable su integración y la generación de diversos tipos de aplicaciones robóticas. Esta unión de generación automática y guiada de artefactos software, junto con el aumento del nivel de abstracción con que se realiza el diseño, debe elevar enormemente la productividad y la calidad final del software obtenido.

Resumiendo, esta Tesis Doctoral completa el enfoque propuesto por ACRoSeT utilizando el novedoso *desarrollo dirigido por modelos* (Model-Driven Engineering, **MDE**) para (1) proponer una solución a la ambigüedad semántica y de implementación de algunos de los conceptos propuestos por ACRoSeT; (2) proporcionar una serie de herramientas para aumentar el nivel de abstracción del desarrollador; (3) cambiar el proceso de traducción manual de los componentes abstractos a código ejecutable por un proceso semi-automático que elimine errores en la implementación final y (4) permitir que el proceso de traducción pueda ser extendido a diversos lenguajes de programación e incluso frameworks robóticos. Para llevar a cabo el objetivo principal se proponen los siguientes objetivos parciales:

- Estudiar las diversas alternativas que ofrece el enfoque de desarrollo MDE/MDA y seleccionar la más adecuada. Se realizará un estudio comparativo entre las soluciones propuestas por el OMG (MDA, UML 2 o *profiles* de UML 2) y el desarrollo de un DSL

específico. Este estudio permitirá guiar otros desarrollos que el DSIE pueda llevar a cabo en el futuro.

- En vista del enfoque finalmente adoptado se seleccionarán las herramientas de desarrollo que permitan llevar a cabo este trabajo. Entre las posibles herramientas candidatas se encuentran Eclipse, Rational Rose, Microsoft DSL Tools, Rhapsody y un largo etcétera.

- Una vez seleccionado el enfoque y la herramienta se pasará a desarrollar la primera etapa del desarrollo software dirigido por modelos: la creación de el o los meta-modelo/s del dominio. Para ello se partirá de los resultados de ACRoSeT, que tendrán que ser ajustados a la solución elegida (UML 2, *profile* de UML 2 o desarrollo de un meta-modelo desde cero).

- Se desarrollará una traducción automática del modelo de componentes generado en el punto anterior a un lenguaje de implementación orientado a objetos. Puesto que los conceptos que maneja ACRoSeT, principalmente los relacionados con la orientación a componentes, no tienen traducción directa a los lenguajes orientados a objetos (los más utilizados en la actualidad), será necesario desarrollar una traducción entre ambos modelos, máxime si se tiene en cuenta que otro de los objetivos de esta Tesis Doctoral y de ACRoSeT es obtener la máxima independencia de la plataforma final de ejecución.

- Por último, y para validar la aplicación del enfoque propuesto en esta Tesis Doctoral, se desarrollará una transformación a código, que genere parte del esqueleto de la aplicación orientada a componentes en un lenguaje orientado a objetos, concretamente Ada en su revisión 2005. Tras la generación del esqueleto el usuario tendrá que rellenar las partes de menor nivel en las que se definen los algoritmos de control, ya que la definición de estos no se contempla en el modelo ACRoSeT.

1.4 ESTRUCTURA DEL DOCUMENTO

LA PRESENTE Tesis Doctoral ha sido dividida en dos bloques de contenidos, contabilizando un total de nueve capítulos. Además, el documento contiene dos anexos con información adicional, un glosario para consultar el significado de los acrónimos más utilizados a lo largo del texto y una última sección en la que se recogen las citas bibliográficas. La estructura detallada de la Tesis Doctoral es la siguiente:

► **Capítulo 1: Planteamiento y objetivos.** En este capítulo se realiza una breve introducción en la que se han expuesto los problemas que se han abordado en el desarrollo

de esta Tesis Doctoral. Además, se ha presentado el marco de trabajo en el que se encuadra esta investigación y se han establecido los objetivos de la misma.

► **BLOQUE 1: ESTADO DE LA TÉCNICA.** Este bloque recoge todos los capítulos relacionados con el resumen del estado de las técnicas y tecnologías que se han utilizado para desarrollar esta Tesis Doctoral. Cada uno de los capítulos englobados en este bloque contiene, al final, una sección en la que se resumen las principales aportaciones de cada disciplina al desarrollo de esta Tesis Doctoral.

▷ **Capítulo 2: Desarrollo basado en componentes.** Este capítulo describe las características de los componentes como artefactos que facilitan la reutilización en pequeña escala (*reusing in the small*) y se centra, fundamentalmente, en resumir las características de los «modelos de componentes» y el uso de componentes de terceros (COTS). Este capítulo es necesario puesto que el objetivo de la presente Tesis Doctoral es desarrollar aplicaciones robóticas basadas en componentes.

▷ **Capítulo 3: Arquitectura software.** En este capítulo se describen las características de esta disciplina y se enumeran las principales tecnologías y métodos desarrollados para describir la arquitectura de una aplicación. También se describen las características de uno de los pilares básicos sobre los que se asienta esta Tesis Doctoral: los frameworks de componentes, como tecnología que facilita la reutilización en gran escala (*reusing in the large*).

▷ **Capítulo 4: Desarrollo basado en modelos.** En este capítulo se describe el novedoso enfoque de desarrollo software basado en modelos (MDE), que soporta gran parte del desarrollo de esta Tesis Doctoral, y sus conceptos básicos: modelo, metamodelo y transformaciones de modelos. También se describe la iniciativa *Model-Driven Architecture* (MDA) y todos los estándares definidos por el OMG que han impulsado el desarrollo de MDE. Como ya se ha comentado, el enfoque MDE se utiliza para crear aplicaciones para el control de robots basadas en componentes e implementadas en lenguajes orientados a objetos o utilizando otras tecnologías, por ejemplo frameworks de robótica (ver apartado 5.3).

▷ **Capítulo 5: Arquitecturas y frameworks de control de robots.** En este capítulo se realiza un recorrido histórico por las principales arquitecturas que se han desarrollado para llevar a cabo el control de robots móviles, así como se describen también la nuevas tendencias de diseño del software de control: los frameworks robóticos. Por último, se describe someramente la arquitectura de control ACRoSeT, desarrollada en el seno del grupo de investigación DSIE.

► **BLOQUE 2: APORTACIONES DE LA TESIS.** Este segundo gran bloque agrupa todos los capítulos en los que se exponen tanto el desarrollo como los resultados de la presente Tesis Doctoral. La estructura de este bloque es la siguiente:

- ▷ **Capítulo 6: Diseño de V^{3}Studio.** En este capítulo se realiza una descripción completa del desarrollo y utilización del meta-modelo de componentes V^{3}Studio. V^{3}Studio está formado por tres vistas separadas pero complementarias que permiten describir completamente un sistema basado en componentes. Estas tres vistas permiten describir, respectivamente, la arquitectura de una aplicación basada en componentes, el comportamiento de un componente y la secuencia de algoritmos que ejecuta. Este capítulo se complementa con las restricciones OCL descritas en el apéndice A.
- ▷ **Capítulo 7: Transformación V^{3}Studio a UML: de componentes a objetos.** Este capítulo describe dos de las aportaciones fundamentales de esta Tesis Doctoral: (1) la propuesta de una implementación de los conceptos básicos de CBD utilizando una tecnología orientada a objetos y (2) el desarrollo de una transformación de modelos que lleve a cabo esta traducción, convirtiendo un modelo V^{3}Studio en un modelo UML, listo para ser traducido a cualquier lenguaje orientado a objetos.
- ▷ **Capítulo 8: Aplicación de V^{3}Studio al control de una mesa XYZ.** En este capítulo se describe una transformación de modelo a texto para convertir el modelo UML obtenido en el capítulo anterior a código Ada así como la aplicación de V^{3}Studio al robot cartesiano desarrollado en el contexto del proyecto europeo EFTCoR.

► **Capítulo 9: Conclusiones y trabajos futuros.** En este último capítulo se exponen las aportaciones realizadas en esta Tesis Doctoral y los resultados concretos obtenidos. Además se proponen algunas líneas de investigación que continúan y amplían el trabajo realizado.

► **Apéndice A: Restricciones OCL adicionales.** Este apéndice recoge las principales restricciones OCL que se han definido para asegurar la corrección de los modelos V^{3}Studio. Las restricciones contenidas en este apéndice complementan la explicación del meta-modelo de V^{3}Studio, puesto que la formalidad del lenguaje OCL permite aclarar determinados aspectos que no hayan quedado claros en la explicación del capítulo 6.

► **Apéndice B: Una visión detallada de los diagramas de actividad.** Este apéndice resume las principales clases UML que aparecen en el diagrama de actividades y que se han utilizado a la hora de describir las actividades que se generan en el transcurso de la transformación de modelos que se detalla en el capítulo 7.

PARTE I
ESTADO DE LA TÉCNICA

En resolución, él se enfrascó tanto en su lectura que se le pasaban las noches leyendo de claro en claro, y los días de turbio en turbio; y así, del poco dormir y del mucho leer, se le secó el cerebro, de manera que vino a perder el juicio.

MIGUEL DE CERVANTES SAAVEDRA

CAPÍTULO 2

DESARROLLO BASADO EN COMPONENTES

HOY EN día la complejidad y el tamaño de los productos software crece rápidamente y los clientes demandan productos cada vez de más calidad, más complejos y con menores plazos de entrega. Para cubrir esta creciente demanda y ajustarse a las necesidades del productor es imprescindible desarrollar nuevas tecnologías que faciliten la reutilización del software previamente diseñado y su integración en los nuevos productos. Este ha sido el objetivo que tradicionalmente ha perseguido el estudio del desarrollo software basado en componentes (CBD). Este capítulo realiza un breve resumen del estado de la técnica en desarrollo de componentes, ensamblaje y demostración de propiedades.

El estudio de la tecnología de componentes comienza con un repaso de las principales definiciones de «componente» existentes en la bibliografía de esta disciplina. Posteriormente se aborda la metodología de desarrollo seguida cuando se utilizan componentes para diseñar una aplicación, así como las características fundamentales que tiene que cumplir un modelo de componentes, es decir, la aplicación que soporta la integración e interacción entre los distintos componentes que la forman. Por último, se abordan los problemas derivados del uso e integración de componentes desarrollados por terceras personas, componentes COTS.

2.1 INTRODUCCIÓN

COMO ya se dijo en la sección 1.1, los ingenieros del software llevan casi cuarenta años esperando a que la idea, enunciada por primera vez por McIlroy, de disponer de «componentes software» para la construcción de aplicaciones sea una completa realidad. En definitiva, la *Ingeniería del Software* quiere imitar la forma en que otras ingenierías resuelve sus problemas desde hacía casi un siglo, importando y adoptando uno de los conceptos claves y común a todas ellas: el concepto de *componente* como bloque constructivo. Tales componentes software deben estar diseñados de tal forma que puedan ser utilizados en distintas aplicaciones sin que el ingeniero tenga que modificarlos. La existencia de artefactos software con estas características provocará la aparición y expansión de una industria de compra/venta de componentes software, que a su vez permitirá la creación de nuevos componentes y aplicaciones a partir de otros preexistentes.

Sin embargo, el diseño software sigue siendo una actividad muy complicada. En el clásico *The Mythical Man-Month* [38], Brooks señala que la complejidad es un aspecto inherente al diseño software (*essential complexity*). Según Parsons [183], no existe un mundo ideal en el que las aplicaciones se construyan mediante el ensamblaje de componentes; el diseño de una aplicación nunca será, de forma general, tan sencillo como es para un niño crear un castillo a base de ensamblar piezas predefinidas. Parte de esta falsa concepción existente alrededor del concepto de «componente» reside, según Szyperski [201], en que la naturaleza del software no ha sido bien entendida.

Según este autor, la naturaleza del software es distinta a la de otras ingenierías, y los productos generados por el ingeniero de software son distintos a los producidos por los ingenieros de cualquier otra rama. El producto de un ingeniero del software, un programa, se asemeja más a lo que el resto de ingenierías considera como un plano o una guía que describe el producto que se está diseñando, y no a un producto final concreto. El producto final concreto es generado por la máquina que ejecuta este programa–guía. Siguiendo este razonamiento, esta máquina puede verse como una gran factoría, que construye un componente cada vez que ejecuta un programa. A la luz de esta aclaración, lo que McIlroy pretendía conseguir con sus declaraciones era, haciendo un símil con la terminología arquitectónica, que fuera posible componer los planos de distintos edificios para construir un tercero. Algo que, a priori, no parece tan sencillo y evidente como desarrollar bloques básicos con los que crear los edificios: el ladrillo.

El motivo de que el concepto de componente software esté tan arraigado y se haya estudiado tanto se debe a que surgió en el lugar adecuado en el momento justo. Surgió en un momento en que el mundo empezaba a depender cada vez más de las máquinas y de los

programas que las controlaban, pero en el que no se disponía de un método para diseñarlos ni mantenerlos (el momento conocido como «crisis del software»). El lugar adecuado fue la famosa conferencia de la OTAN en que se puso de manifiesto la necesidad de disponer de una rama del conocimiento específica para el desarrollo de programas: la *Ingeniería Software*. Cuando se le solicitó ayuda para resolver el problema del diseño software, la incipiente ingeniería recurrió a la solución proporcionada por otras ingenierías, bastante más maduras, equiparando conceptos de forma errónea. El hecho de que el –engañoso– símil entre componente software y bloque arquitectónico, componente mecánico y circuito integrado funcionara ha permitido que aquél se mantenga en el tiempo, pese a ser erróneo. Prueba de ésto es que, como se muestra en la siguiente sección, no existe consenso en la definición de componente, aunque la mayoría comparten ciertas características.

Pero tampoco hay que desechar todo el esfuerzo realizado por la comunidad de desarrollo software durante estos años. La maduración de la *Ingeniería del Software* como disciplina, el desarrollo de nuevas técnicas, metodologías y paradigmas de programación y la obtención de un conocimiento más profundo del producto, el software, está permitiendo redirigir el desarrollo de componentes. Prueba de ello son los entornos de desarrollo y *frameworks* orientados a componentes que están apareciendo en el mercado, así como el esfuerzo de algunas empresas por adoptar este enfoque para reducir el tiempo que tardan en diseñar el software de un nuevo producto, tal y como se muestra en la sección 2.4.2. La realización de aplicaciones nunca será tan sencillo como unir componentes, pero las nuevas tecnologías están investigando formas de conseguirlo que no añadan excesiva complejidad al proceso.

El actual paradigma de programación, la programación orientada a objetos, no ha cumplido del todo con las esperanzas depositadas en él, principalmente las relacionadas con la reutilización de código, quizá porque la unidad de reutilización es demasiado pequeña [104, 186]. Sin embargo, según afirma Uddel [207] «*Object orientation has failed but component software is succeeding*», aunque en su caso se refería a los componentes realizados con *Visual Basic* en 1.994. Pero ya en aquel momento se comenzaba a atisbar el potencial que podía llegar a ofrecer la programación orientada a componentes. Chambers adelanta en [56] los objetivos que tiene que alcanzar el desarrollo basado en componentes antes de que se convierta en una tecnología ampliamente utilizada: (1) desarrollo de lenguajes de programación verdaderamente orientados a componentes que los definan de forma flexible y descriptiva; (2) desarrollo de compiladores eficientes, que eliminen la sobrecarga asociada a que los componentes son generados independientemente; y (3) desarrollo de herramientas para organizar y crear aplicaciones a partir de componentes.

La generación de aplicaciones a partir de componentes se puede observar tanto desde el punto de vista del desarrollador de componentes (y del modelo de componentes y

framework, como se muestra en la sección 2.4) como desde el del que crea aplicaciones ensamblando componentes ya existentes. Esta sutil diferencia da lugar a la aparición de dos ramas en la Ingeniería del Software [75]: Component-Based Software Engineering (**CBSE**) y Component-Oriented Software Engineering (**COSE**) respectivamente. De forma genérica, los siguientes autores realizan sendas definiciones del desarrollo software utilizando componentes, independientemente de la rama (CBSE o COSE):

- **Catalysis (1998) [79]:** enfoque de desarrollo en el que todos los elementos software – desde el código ejecutable a las especificaciones de las interfaces, arquitecturas y modelos de negocio; y a todas las escalas, desde aplicaciones y sistemas completos hasta la creación de las partes más pequeñas– pueden ser construidas ensamblando, adaptando y conectando componentes existentes en una variedad de configuraciones.
- **Bachmann (2000) [17]:** CBSE consiste en el ensamblado rápido de sistemas a partir de componentes donde (a) los componentes y *frameworks* tienen propiedades certificadas y (b) estas propiedades establecen unas bases para predecir las propiedades de los sistemas construidos con estos componentes.

2.2 DEFINICIÓN DE COMPONENTE SOFTWARE

ACTUALMENTE sigue sin haber consenso en la definición fundamental, qué se entiende por «componente» y cuáles son sus características principales. A pesar de que existen diversas definiciones, más o menos aceptadas y con características similares, todavía no se ha alcanzado un amplio consenso en este tema. Aunque las definiciones de componente están repartidas por toda la bibliografía existente en el tema, Szyperski [201] y García [98] realizan sendas recopilaciones de la mayoría de ellas. Czarnecki [68] afirma incluso que no es posible definir el concepto «componente software», ya que es un concepto natural y no artificial, como el concepto «objeto». A continuación se enumeran algunas de las principales definiciones de componente software que se pueden encontrar en la bibliografía:

Booch (1987) [34]: un componente software es un módulo altamente cohesivo y con bajo acoplamiento que se refiere a una única abstracción y que puede ser reutilizado en diversos contextos.

ECOOP'96 : un componente software es una unidad de composición que posee interfaces especificadas mediante contratos y con dependencias explícitas del contexto únicamente. Un componente software puede ser desplegado independientemente y está sujeto a composición por parte de terceros.

Szyperski (1997) [201]: un componente es un conjunto de componentes atómicos que son normalmente desplegados de forma conjunta. Un componente atómico es aquel formado por un «módulo» y un conjunto de «recursos». Un módulo es un conjunto de clases y otras construcciones, como funciones o procedimientos. Un recurso es una colección estática de elementos que tienen un tipo y que parametrizan al componente.

Catalysis (1998) [79]: un componente software es una implementación software que (a) puede ser desarrollada y repartida de forma independiente; (b) tiene interfaces explícitas y bien definidas para los servicios que ofrece y espera de otros; (c) puede ser compuesto con otros componentes sin que tenga que modificarse.

Bosch (2000) [35]: un componente software es una unidad de composición que ofrece interfaces explícitamente especificadas, requiere de unas interfaces e incluye unas interfaces de configuración, incluyendo además atributos de calidad.

Bachmann (2000) [17]: un componente es una implementación opaca de una determinada funcionalidad (caja negra) que puede ensamblarse con otros componentes siguiendo las reglas establecidas por el modelo de componentes adoptado.

CBSE Handbook (2001) [110]: un componente software es un elemento software conforme a un modelo de componentes que puede ser desplegado independientemente y compuesto sin modificaciones, de acuerdo a unas reglas de composición.

UML v2.0 (2005) [170]: un componente es una unidad auto-contenida que encapsula el estado y el comportamiento de un conjunto de *Classifiers*. Un componente especifica un contrato formal de los servicios que proporciona a sus clientes y de los que requiere de ellos, en forma de interfaces provistas y requeridas. Un componente es una unidad sustituible, que puede ser cambiada tanto en tiempo de compilación como en tiempo de ejecución por otro componente que ofrece una funcionalidad equivalente, basada en la compatibilidad de sus interfaces.

> *While the concept of software components has been well known virtually since the beginning of software, the practical aspects and challenges have been more fully evolved over time.*
>
> — Jon Hopkins [116]

Vista la falta de consenso en la definición de componentes, Brown [40] propone una visión alternativa del concepto de componente, contemplándolo desde tres puntos de vista distintos. Según su trabajo, estas tres perspectivas son necesarias para poder comparar distintos modelos de componentes, ya que no existe una definición única de componente que aplicar. Estas tres perspectivas son:

Perspectiva de empaquetamiento. El componente se considera como una unidad de empaquetamiento y distribución. Es un concepto de organización, centrado en la identificación de un conjunto de elementos que pueden ser reutilizados como una unidad. El énfasis está puesto en la reutilización. Se pueden distinguir varios tipos de elementos software que pueden ser considerados como un componente. Un tipo especial de componente, bajo esta perspectiva, es aquél que está pensado para distribuirse en un formato binario (aunque no exclusivamente en un único fichero binario).

Perspectiva de servicio. Considera al componente como una entidad que ofrece servicios a sus clientes. El diseño e implementación de aplicaciones involucra comprender cómo se van a intercambiar estas peticiones entre los componentes que colaboran en la consecución del servicio pedido. Bajo esta perspectiva se remarca la importancia de los *contratos* entre componentes, como medio indispensable para coordinar la comunicación entre componentes y para permitir sustituir un componente por otro, ya que separan especificación de implementación. Los servicios se agrupan en unidades contractuales coherentes, conocidas como *interfaces*, equiparables a un contrato ya que especifican todos los servicios que ofrece el componente; además, representan la única manera que tienen los clientes de acceder a dichos servicios. La perspectiva de servicio presenta, por tanto, una vista lógica del componente. Entre esta perspectiva y la de empaquetamiento existe una relación muchos-a-muchos: un conjunto lógico de servicios puede estar empaquetado en distintos componentes, y un mismo componente puede empaquetar funcionalidad lógicamente relacionada.

Perspectiva de integridad. Esta perspectiva se centra en identificar los límites de un componente, de forma que sea posible su sustitución por otro, algo que la perspectiva de servicio no establece de forma clara. La perspectiva de integridad define a un componente como una cápsula de implementación que encierra todo el software que, de manera colectiva, mantiene la integridad de los datos que manipula y que, por tanto, es independiente de la implementación de otros componentes. El concepto de independencia se extiende también a los sub-componentes que lo forman, de forma que todo puede ser reemplazado como una única unidad.

Por último resta por definir el concepto «tamaño» de un componente, o más bien su granularidad. En principio, este concepto puede no parecer importante, ya que una de las principales características de los componentes es que son *componibles* y, por tanto, debería ser posible componer un componente muy grande con uno muy pequeño. Sin embargo, puesto que los componentes también son *unidades de sustitución*, es importante que su granularidad se adapte, en la medida de los posible, a su propio propósito y al rol que va a desempeñar en la aplicación. Szyperski [201] enumera distintos aspectos del funcionamiento de un

componente que pueden ayudar a elegir su tamaño. Un componente puede verse como una unidad de abstracción, de análisis, compilación, despliegue, distribución, extensión, contención de fallos, instanciación, instalación, unidad de carga, de localización (en aplicaciones distribuidas), mantenimiento o configuración del sistema. Toda esta heurística para determinar la granularidad de un componente se puede resumir en la siguiente frase: «*Component granularity should maximize reuse and minimize use*» [201].

Por otro lado, Gomaa [102] cita otros criterios que pueden ayudar a definir un componente, agrupando funcionalidad relacionada: componente cliente o componente servidor (al igual que el patrón con este nombre), componente de interfaz gráfica, componente de entrada/salida, componente de control, componente de coordinación, componente de análisis de datos y componente de almacenamiento. Beneken propone tres niveles de granularidad para los componentes [27]: básico (depende de otros componentes básicos, no es ejecutable), ensamblaje (está formado por componentes básicos u otros ensamblajes, no es ejecutable) y sistema (formado por componentes ensamblados, tiene dependencias mínimas de otros componentes y es ejecutable).

2.3 Fases de Desarrollo Software en CBD

TRADICIONALMENTE el ingeniero software ha seguido un enfoque descendente (*top-down*) para el desarrollo de sus sistemas [13, 182]. Sin embargo, el desarrollo basado en componentes apuesta por un enfoque ascendente (*bottom-up*), en el que los productos se construyen mediante el ensamblaje e integración de componentes software preexistentes. Como ya se comentó, existen básicamente dos enfoques para realizar un diseño basado en componentes: o bien se utilizan componentes desarrollados por terceros, en cuyo caso únicamente se tienen que seleccionar y ensamblar correctamente (enfoque COSE), o bien se decide desarrollar parte de los componentes que forman la aplicación (enfoque CBSE). Estos componentes comprados o adquiridos de terceros se denominan genéricamente COTS (Commercial Off-The-Shelf). Dependiendo del enfoque elegido, el ciclo de desarrollo cambia ligeramente. Brown [40] y Carney [52] (ver figura 2.1) proponen el siguiente ciclo para desarrollar aplicaciones siguiendo un enfoque puramente COSE:

1. Selección y evaluación de los componentes software, que deberán satisfacer las necesidades del usuario y ajustarse al esquema de diseño de la aplicación.
2. Adaptación de los componentes cuando sea necesario
3. Ensamblaje de los componentes como parte de la solución final.
4. Evolución del sistema si el usuario lo requiere.

Para el caso en que se sigue el enfoque CBSE, el proceso de desarrollo se modifica ligeramente, según menciona Cheesman [57] (ver figura 2.2). Cheesman aborda todas las fases del proceso de desarrollo de componentes por parte de una empresa, utilizando el Unified Modelling Language (**UML**) para describir la especificación. En este caso, existe un paso previo en que se crean los componentes. Para ello se parte de la especificación de la funcionalidad del sistema, que es agrupada en distintos componentes según dicte el modelo de negocio o la arquitectura final del sistema. El resto de pasos son equivalentes a los que expuso Brown [40]. Lo que resta de apartado se va a dedicar a exponer los pasos comunes a ambos enfoques.

2.3.1 SELECCIÓN Y EVALUACIÓN DE COMPONENTES

Esta etapa permite determinar qué componentes, de entre los disponibles, se ajustan mejor a las necesidades de una determinada aplicación. Encontrar el componente adecuado (o un conjunto de posibles candidatos) requiere establecer con precisión los criterios de búsqueda. Entre éstos, suelen incluirse los relativos a la funcionalidad (servicios que debe proporcionar) aunque también es común incluir otros relacionados con el coste, el fabricante, compatibilidad con una determinada plataforma, etc. Esta fase puede resultar tediosa, debido a que la abundante información disponible no siempre está correctamente clasificada, además de que los resultados de la búsqueda pueden resultar difíciles de cuantificar y ordenar [120]. Por tanto, es necesario que se profundice en el desarrollo de una taxonomía para clasificar componentes y de un repositorio donde almacenarlos.

Una vez encontrado el o los componentes que parecen adecuados se pasa a la fase de evaluación de la idoneidad del componente. Esta fase tampoco resulta sencilla, si bien

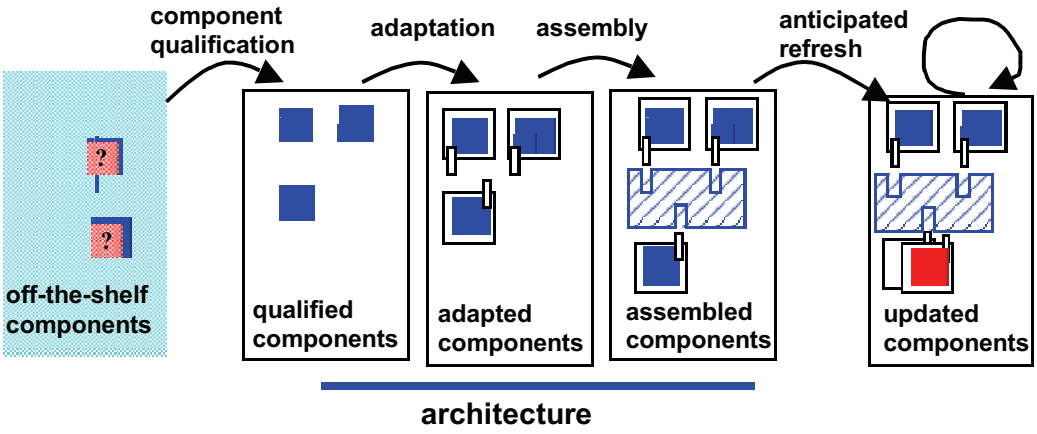


Figura 2.1: Desarrollo de una aplicación basada en componentes COTS (extraído de [52])

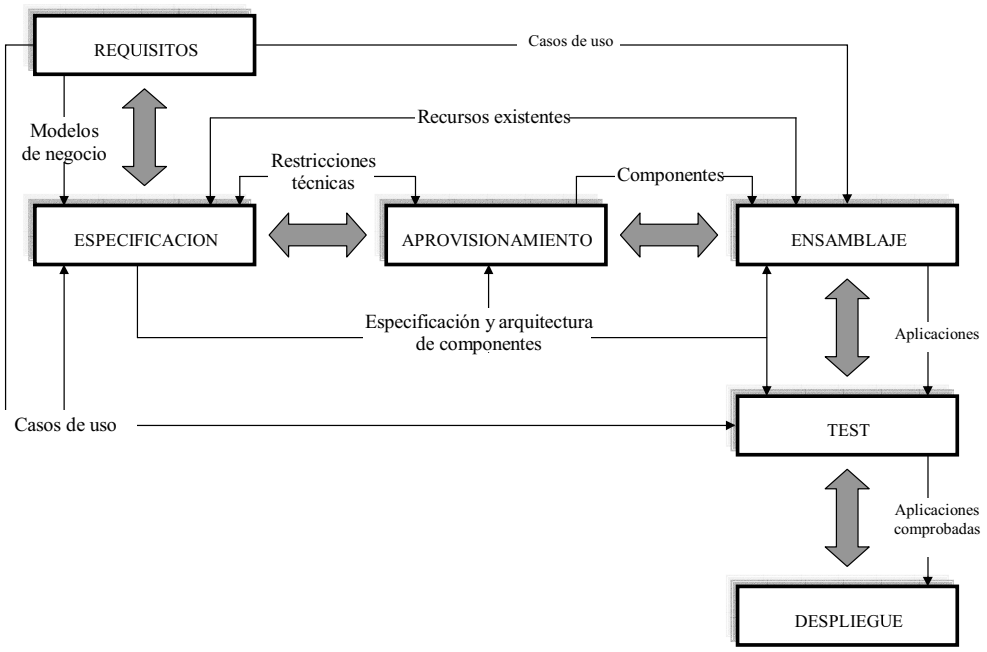


Figura 2.2: Proceso de desarrollo de aplicaciones basadas en componentes (extraído de [57])

la normativa ISO/IEC-9126_91 [119] recopila algunas técnicas relativamente maduras que pueden aplicarse a tal efecto, como pueden ser la realización de encuestas a los clientes y la construcción de prototipos con los que validar, de manera efectiva, el comportamiento de los componentes seleccionados. Gao [97] defiende que los componentes deben diseñarse de forma que la fase de comprobación sea lo más rápida y eficaz posible, y propone un «nuevo concepto, denominado *testable bean*, que ayude a entender las características y propiedades que debe tener un componente para que pueda ser comprobado». Finalmente, Vincenzi realiza en [214] un estudio exhaustivo del estado de la técnica en tests para componentes.

Otro de los aspectos más importantes dentro de la selección de componentes es la verificación de las características que exponen, i.e. verificar que el componente se comporta en la realidad tal y como afirma el vendedor. Puesto que una verificación completa, por parte de cada uno de los compradores de un componente dado, resultaría un proceso largo y costoso para ambas partes, la certificación de las características del componente surge como una solución a este problema. Aunque la certificación del software no es una idea nueva, la progresiva utilización de componentes para desarrollar software ha provocado la realización de estudios más detallados. Entre éstos destaca el esfuerzo del SEI en sus trabajos sobre el desarrollo utilizando componentes [17, 216], enfocados sobre todo a su última iniciativa, el PACC (*Predictable Assembly for Certified Components*) (ver sección 2.5). Meyer realiza un estudio [152] sobre el desafío que supone conseguir llegar a certificar

componentes software y las enormes ventajas que aportaría esta certificación al desarrollo de programas. En este artículo, Meyer escribe la siguiente definición de componente:

> *A component is a software element (modular unit) satisfying the following 3 conditions:*
>
> 1. *It can be used by other software elements, its "clients".*
> 2. *It possesses an official usage description, which is sufficient for a client to use it.*
> 3. *It is not tied to any fixed set of clients.*
>
> — Bertrand Meyer [152]

2.3.2 ADAPTACIÓN DE COMPONENTES

En el momento de crear nuevos componentes, los desarrolladores hacen ciertos supuestos sobre los sistemas en los que éstos se integrarán. Sin embargo, cuando estas suposiciones no se ajustan a las necesidades del cliente, como suele ocurrir normalmente, éste deberá realizar ciertas adaptaciones de forma que los componentes puedan «encajar» en el sistema. El procedimiento de adaptación depende de la accesibilidad del componente. Cuando éste se adquiere como una «caja negra» habrá que adaptar el sistema para integrar el componente, mientras que si el componente seleccionado es accesible, ya sea porque es de fabricación propia o porque se ha adquirido con ciertos privilegios para poder modificarlo, puede resultar más conveniente adaptar el componente al sistema.

Según menciona Szyperski [201], un componente puede requerir adaptaciones para limitar su funcionalidad, para extenderla o para hacerlo compatible con otros componentes, con la plataforma, etc. Uno de los mecanismos de adaptación más utilizados consiste en construir un componente *ex profeso* de tipo *wrapper* que envuelva al que se quiere adaptar, de forma que lo limite, extienda, o modifique según sea necesario.

2.3.3 ENSAMBLAJE O INTEGRACIÓN DE COMPONENTES

Como ya se ha comentado con anterioridad, el proceso de ensamblaje requiere de la existencia de un modelo de componentes, entendido como un conjunto de reglas que establecen qué componentes se pueden conectar con qué otros y cómo se comunican éstos entre sí. Dado que los componentes se comunican a través de sus interfaces, será necesario utilizar un lenguaje de definición de interfaces que permita definirlas de una manera precisa. Además, cuando se precise conectar dos componentes situados en distintas localizaciones, será necesario proporcionar los mecanismos adecuados para su comunicación remota. Todo

ello requiere, además, contar con una infraestructura o *middleware* que permita realizar el proceso de ensamblaje de los componentes.

Además de los problemas relativos al ensamblaje de componentes, Parsons [183] y Mogul [155] advierten del problema de la computación *emergente* que puede surgir tras el proceso de ensamblaje de componentes. En este caso la funcionalidad del sistema es mayor que la funcionalidad de cada uno de los componentes por separado. De esta forma el sistema puede comportarse de forma no deseada, aunque el comportamiento de cada componente sea el correcto. Y este problema no puede solucionarse haciendo tests de los componentes, ya que son tests locales y no alcanzan a toda la arquitectura.

2.3.4 Evolución del sistema

A priori podría parecer que los sistemas construidos a partir de componentes deberían poder evolucionar de manera sencilla, tal y como sucede en los procesos industriales de montaje, en los que la sustitución de piezas o su actualización con otras más modernas resulta una tarea casi trivial. Sin embargo, cuando se trata de productos software, la sustitución de un componente por otro suele ser una tarea compleja y tediosa. El nuevo componente debe someterse a un proceso minucioso de validación, para comprobar que se ajusta a lo que se espera de él, e incluso puede ser necesario plantear una modificación drástica de la estructura del sistema para permitir su integración.

Adicionalmente, debe tenerse en cuenta que la evolución de un sistema basado en componentes depende en gran medida de cómo los fabricantes decidan evolucionar dichos componentes. Szyperski [201] advierte que deben proponerse soluciones para el problema que él denomina *«version mess»* (conflicto de versiones) y de los problemas que puede crear el propio mercado. Puede ocurrir que el fabricante no esté interesado en desarrollar una nueva versión del componente para cubrir la demanda de un cliente puntual o incluso que decida crear una nueva versión del producto dejando sin soporte la versión previa, lo que obligará al cliente a decidir si cambia de versión o de proveedor.

2.4 Modelos de Componentes

TODAS las definiciones de componente presentadas en el apartado 2.2, aunque ligeramente distintas, coinciden en que un «componente» es una unidad para construir aplicaciones (que bien pueden ser otros componentes) y que su diseño tiene que permitir que sean reutilizados en distintos contextos. Esta última condición es la

más restrictiva de todas ya que, llevada al extremo, implica que no se puede depender de ninguna funcionalidad concreta de la plataforma de ejecución. Aunque ésto fuera posible y eficiente (algo impensable), todavía restaría por definir la forma en que se comunicarían los distintos componentes que forman el sistema. Y aquí sí que se necesita que los distintos fabricantes de componentes se pusieran de acuerdo, o incluso surgieran estándares. La necesidad de estándares (uno o varios) no es un restricción para el éxito en la implantación del desarrollo basado en componentes, aunque es algo común en el resto de ingenierías con las que tan a menudo se compara el software. Por ejemplo, en el mundo de la electrónica, los niveles de tensión y corriente están reglados, así como las características físicas de las partes involucradas en la conexión de los componentes electrónicos.

En esta sección se presentan las principales características de los sistemas y subsistemas que sirven para definir un marco (de diseño, ejecución, prueba, sustitución, provisión de servicios, etc) en el que poder ejecutar una aplicación basada en componentes: el modelo de componentes. Para ello, el siguiente apartado presenta las principales características que tiene que tener un sistema de este tipo, mientras que los dos apartados siguientes resumen los principales modelos de componentes de propósito general (javabeans, .NET, Corba CCM y Mono) así como algunos modelos de componentes de propósito específico (KParts, NesC, Pin, etc).

2.4.1 CARACTERÍSTICAS DE UN MODELO DE COMPONENTES

Bachmann [17] defiende que un sistema basado en componentes nace como resultado de la adopción de una estrategia de diseño que persiga este fin y que, por tanto, no puede ser fruto de la casualidad ni se puede improvisar conforme se va realizando el diseño. Denomina «patrón de diseño por componentes» a esta estrategia de diseño y también declara que debe existir una tecnología complementaria que incluya los productos y conceptos que necesitan estos componentes [22]. Esta tecnología auxiliar complementaria conforma un «modelo de componentes»: un conjunto de estándares y convenciones para el desarrollo, utilización y evolución de componentes [66]. Es justamente esta conformidad con un modelo (de componentes en este caso) lo que diferencia un componente de cualquier otra forma de software empaquetado. Ambas tecnologías, componentes y modelo de componentes, existen tanto en tiempo de diseño como en tiempo de ejecución, ya que forman parte del sistema desplegado. La figura 2.3 muestra todos estos conceptos.

En dicha figura (1) es un componente, una implementación software que puede ser ejecutada en un dispositivo físico o lógico. Un componente implementa una o más interfaces (2), que reflejan un contrato con sus clientes (3). Estas obligaciones contractuales permiten comprobar que varios componentes desarrollados independientemente puedan

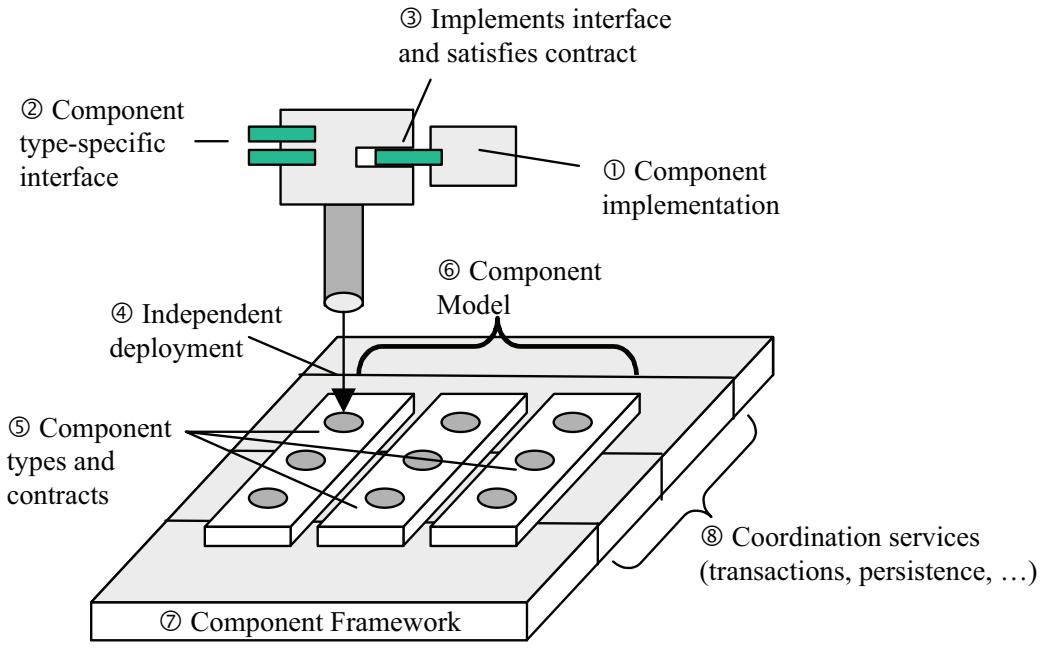


Figura 2.3: El patrón de diseño por componentes (extraído de [17])

interactuar entre sí de forma predecible (de acuerdo al contrato) y que pueden ser desplegados o instalados en un sistema tanto en tiempo de construcción como en tiempo de compilación (4). Un sistema basado en componentes está basado en un conjunto reducido de *tipos de componente*, que cumplen distintos cometidos en el sistema final (5), cometidos descritos por medio de interfaces (2). Un modelo de componentes (6) establece el conjunto de *tipos de componentes*, interfaces y patrones de interacción entre los *tipos de componentes* disponibles en el sistema. Un framework (consultar el apartado 3.4 para más información) de componentes (7) proporciona una serie de servicios, generalmente en tiempo de ejecución (8), que dan soporte al modelo de componentes. En cuanto a las características que debe tener un modelo de componentes, Bachmann [17] especifica algunas de las características principales que un modelo de componentes debe establecer para que pueda ser considerado como tal:

Los tipos de componente. Un tipo de componente puede definirse en términos de las interfaces que implementa. Un componente es del tipo de todas las interfaces que implementa; es decir, es polimórfico, y puede desempeñar el papel de estas interfaces en diferentes ocasiones. Un modelo de componentes puede requerir que los componentes implementen varias interfaces (de configuración), y en este sentido un modelo de componentes puede verse como la definición de uno o más tipos de componentes. Diferentes tipos de componentes pueden jugar distintos papeles en los sistemas software, y participar en diferentes tipos de esquemas de interacción.

Los esquemas de interacción. Los modelos de componentes especifican cómo se localiza un componente en concreto, qué protocolos de comunicación se usan y cómo se consigue la calidad de servicio (seguridad y transacciones, por ejemplo). También puede describir cómo interactúan entre sí, o cómo interactúan con el *framework* de componentes. Los esquemas de interacción pueden ser comunes a todos los tipos de componentes o únicos para un tipo de componente en particular. Aunque son dos características muy importantes para los sistemas basados en componentes, tanto los esquemas de interacción como los enlaces entre recursos van a describirse posteriormente en el apartado 3.2.1 sobre arquitectura software.

Los enlaces con los recursos. El proceso de composición es la forma de enlazar componentes con uno o más recursos. Un recurso es tanto un servicio ofrecido por un *framework* o por algún otro componente desplegado en el *framework*. El modelo de componentes describe qué recursos están disponibles para los componentes y cómo y cuándo los componentes se pueden enlazar a dichos recursos. Desde el punto de vista del *framework*, los componentes son recursos que tiene que manejar. Así, la distribución es el proceso por el que los *frameworks* son unidos a los componentes, y un modelo de componentes describe la forma en que los componentes son distribuidos.

Beneken realiza en [27] un resumen de los principales modelos y tecnologías que soportan el desarrollo basado en componentes, denominados genéricamente *componentware*. En este documento definen un marco para clasificar los modelos de componentes de acuerdo a cuatro parámetros: existencia de descripciones formales de los componentes, notación descriptiva (gráfica o textual), proceso de desarrollo de aplicaciones y soporte de herramientas.

El resto del apartado se dedica a presentar los principales modelos de componentes existentes en el mercado en el momento de escribir esta tesis y a realizar una comparativa entre ellos. Por último se presentan otras aplicaciones de la programación orientada a componentes, más o menos exitosas, aunque su dominio de aplicación es más restringido que en el caso anterior.

2.4.2 MODELOS DE COMPONENTES DE PROPÓSITO GENERAL

Como puede deducirse de la exposición realizada en este capítulo, la creación y mantenimiento de un modelo de componentes es una tarea costosa que requiere una cuidadosa planificación. Cuanto mayor es el espectro de aplicación del modelo, mayor es este esfuerzo. Por esta razón hay tres modelos de componentes que destacan sobre el resto, ya que tienen el apoyo de las mayores empresas del sector: *Enterprise Java Beans* (Sun

Microsystems); *Corba Component Model* (Object Management Group (**OMG**)); *.NET* y *COM+* (Microsoft) y el novedoso *Mono* (Novell), la respuesta de la comunidad de software libre a la iniciativa de Microsoft. No es objetivo de esta tesis comparar los principales modelos de componentes, más aún teniendo en cuenta la gran cantidad de comparativas existentes en la literatura [40, 41, 80, 201, 202, 211]. Sin embargo, ningún repaso del estado de la técnica en programación basada en componentes estaría completo sin un, al menos, breve resumen de los principales modelos de componentes existentes en la actualidad. El cuadro 2.1 muestra un resumen comparativo de la principales características de cada uno de estos modelos.

Sun Microsystems pone a disposición de los programadores dos tipos de componentes, siempre bajo el nombre genérico de *Bean* y programados en Java: componentes del lado del cliente (*JavaBean* a secas) [107] y componentes del lado del servidor (*Enterprise JavaBean*) [72]. Al estar programados en Java, estos componentes pueden ejecutarse en cualquier plataforma que disponga de una máquina virtual. La especificación de *JavaBeans* los define como «elementos reutilizables que pueden ser manipulados visualmente mediante una herramienta». Su uso más habitual es para encapsular componentes gráficos complejos, de forma que se facilite su reutilización. Los *Enterprise JavaBeans* (**EJB**) han tenido más éxito que su hermano menor, gracias a que forman parte de la edición empresarial de Java, que está pensada para entornos distribuidos (principalmente web). El objetivo de los EJBs es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, etc) para centrarse en el desarrollo de la lógica de negocio. Prueba de esto último son los tres tipos de EJB que existen: de sesión (gestionan el flujo de información entre cliente-servidor), de entidad (encapsula los datos utilizados en el servidor) y dirigidos por mensajes (para uso interno del servidor; son asíncronos).

Las tecnologías desarrolladas por Microsoft, siempre para su sistema operativo Windows, en el campo de la orientación a componentes han evolucionado durante estos años, desde la tecnología *Object Linking and Embedding* (**OLE**, 1.990), que más tarde evolucionaría a los controles *ActiveX* (1.996), pasando por *Component Object Model* (**COM**, 1.993) y **COM+** (versión mejorada de COM que permite la comunicación con objetos remotos) hasta su última y activa iniciativa *.NET Framework* (**.NET**, 2002). .NET es «la estrategia de servicios web para conectar información, usuarios, sistemas y dispositivos» desarrollada para competir con la plataforma Java (principalmente con los *EJBs*) y para sustituir, en un futuro, al API Win32 (que contiene las funciones del núcleo del sistema operativo). En la terminología de .NET un componente se denomina *assemblie* y es código ejecutable (EXE o DLL) que contiene toda la información necesaria para su ejecución en la plataforma: un manifiesto descriptivo, una descripción de los tipos de datos incluidos (metadatos), código

intermedio MSIL (*Microsoft Intermediate Lenguaje*, similar al *bytecode* de Java) y los recursos asociados (audio, vídeo, tipos de letra, etc). Un ensamblaje es, por tanto, una unidad lógica, que puede realizarse en varias unidades físicas.

.NET está estructurado en dos capas: *Common Language Runtime* (**CLR**) y *Base Class Library* (**BCL**). El CLR es el encargado de compilar el código escrito en cualquiera de los lenguajes soportados por la plataforma (C#, Visual Basic, Turbo Delphi for .NET, C^{++}, J#, Perl, Python, Fortran y Cobol.NET) en código MSIL, y luego ejecutarlo gracias a un compilador JIT (*Just-In-Time*), al igual que sucede con el código Java. El BCL encapsula la funcionalidad básica del sistema que necesitan los programas, como acceso a los dispositivos, administración de memoria, comunicaciones, gestión de aplicaciones gráficas, gestión de excepciones, etc. .NET parece el primer paso serio hacia la obtención de un framework de componentes que ofrezca la funcionalidad de un sistema operativo y que permita que los componentes estén perfectamente integrados con el entorno de ejecución.

El mundo del software libre tampoco se ha quedado atrás ante este paso de Microsoft y, con Novell a la cabeza, ha desarrollado el proyecto *Mono*[1] en respuesta. Mono es un proyecto de código abierto para crear un grupo de herramientas libres, basadas en GNU/Linux y compatibles con .NET, según la especificación del **ECMA**, una organización internacional que ha contribuido activamente en la estandarización, a nivel mundial, de la tecnología de la información y las telecomunicaciones durante cuarenta años. *Mono* está formado, al igual que .NET, por una capa CLR para compilar los lenguajes soportados (C#, Java, Python y gcc) a código intermedio y un compilador *JIT* para ejecutarlos, así como por la capa BCL, donde residen las funciones auxiliares del sistema (gestión de hilos, administración de memoria, bibliotecas, acceso a los dispositivos, etc). *Mono* es un proyecto independiente de la plataforma que actualmente funciona sobre Linux, FreeBSD, UNIX, Mac OS X, Solaris y plataformas Windows. La primera versión de *Mono* fue lanzada en 2.004.

El *Common Object Request Broker Architecture*[166] (**CORBA**) es un estándar definido por el OMG que establece una plataforma de desarrollo de sistemas distribuidos que facilita la invocación de métodos remotos y que sigue el paradigma de orientación a objetos. CORBA se conoce también como un *middleware* de comunicación, diseñado para funcionar sobre un conjunto heterogéneo de plataformas, con soporte para GNU/Linux, FreeBSD, Solaris, Windows, etc, y lenguajes de programación, como Ada, C, C^{++}, Smalltalk, Java y Python. El OMG comenzó el desarrollo de la especificación del modelo de componentes CORBA (*CORBA Component*

[1] `http://www.mono-project.com/`

Model, **CCM**) [172] para hacer frente al rápido crecimiento de los EJBs y proporcionar una solución compatible multi-lenguaje.

El modelo de componentes CORBA **CCM** combina conceptos tanto de COM como de EJB, intentando utilizar lo mejor de cada una de las dos aproximaciones. El OMG define un lenguaje de especificación de componentes (**CDL**, *Component Description Language*), con una sintaxis y un objetivo similar al de **IDL**. Con este lenguaje es posible especificar componentes de forma independiente a la plataforma de ejecución (lenguajes y sistemas operativos), componente que es instanciado en tiempo de ejecución de acuerdo a su descripción. Además, los componentes CCM pueden comunicarse entre ellos y con el entorno por medio de distintos tipos de puerto, dependiendo de si ofrece facilidades de configuración, son fuentes o sumideros de eventos, etc.

2.4.3 OTROS MODELOS DE COMPONENTES

No sólo las grandes empresas del software se han dado cuenta de las grandes ventajas que supone desarrollar un modelo de componentes que ayuden a los usuarios a desarrollar sus productos de forma más rápida, segura y con la posibilidad de reutilizar parte del trabajo ya realizado. Otras empresas, también grandes en su sector, y grupos de investigación han diseñado sus propios modelos de componentes, aunque más ajustados a sus necesidades.

El sector de la computación embebida, para dispositivos con recursos limitados, ha trabajado mucho en la obtención de sus propios modelos de componentes [209], que les permitan sacar al mercado productos nuevos y mejorados cada vez más rápido. Y en esta carrera el software juega un papel cada vez más influyente. Entre los primeros modelos de componentes aplicados al sector electrónico destaca *Koala*[2] [209, 210], desarrollado por la empresa Phillips y utilizado como base para desarrollar el software de cientos de sus productos de consumo.

El campo de las redes de sensores [4, 28] es otro ejemplo de la aplicación de componentes. En este caso, cada sensor tiene recursos muy limitados (8K bytes de memoria de programa y 512 bytes de memoria de datos), ya que están pensados para ser baratos y funcionar a pilas, por lo que es fundamental que los programas sean muy eficientes y pequeños. *NesC*[3] [100] es una extensión del lenguaje *C* para construir aplicaciones para redes de sensores utilizando componentes, aunque, debido a las restricciones de tamaño, su modelo de componentes no aporta ninguna funcionalidad extra.

Por último, y dentro de este campo de sistemas embebidos, quiero mencionar la

[2] http://www.extra.research.philips.com/SAE/koala/
[3] http://nescc.sourceforge.net/

	COM/DCOM, .NET	**EJB**	**CORBA**
Desarrolladores	Microsoft	Sun Microsystems IBM, BEA, etc.	OMG
Componentes	Módulos con múltiples clases u otras implementaciones	Módulos que pueden contener múltiples clases	Módulos que contienen cualquier implementación
Interfaz	OLE IDL (interfaces como una colección de funciones)	Lenguaje Java	IDL propio de el OMG
Conexión	Vía punteros a interfaces	Vía eventos y escuchadores	Vía IDL
Mecanismos de variabilidad	Genericidad, agregación, etc.	Herencia y agregación	Herencia y agregación
Plataforma	Windows (.NET es multi-plataforma)	Multi-plataforma, cualquier sistema con máquina virtual Java (JVM)	Multi-plataforma
Lenguaje de implementación	Cualquier lenguaje de una lista dada	Java	Cualquier lenguaje
Mecanismos de distribución	DCOM, Internet	EJB, Internet, RMI (Remote Method Invocation)	Un ORB de un proveedor

Cuadro 2.1: Comparativa entre distintos modelos de componentes de propósito general

tecnología *Pin* [112] del Software Enginnering Institute (**SEI**). Pin representa el esfuerzo del SEI por desarrollar una especificación común para el desarrollo de componentes para sistemas embebidos, aunque actualmente se encuentra en estado de desarrollo y define únicamente el nivel de ensamblaje.

Otros ejemplos, más cercanos, son los dos escritorios principales del mundo GNU/Linux: **KDE** (*K Desktop Environment*) y **GNOME** (*GNU Network Object Model Environment*). Ambos están desarrollados sobre un *middleware* de comunicaciones que hace las veces de modelo de componentes. KDE comenzó usando CORBA, pero poco después desarrolló su propio *middleware*, denominado *KParts*[4], ya que CORBA no satisfacía sus exigencias (latencia y velocidad). Además con KParts consiguieron que un mismo componente se pudiera ejecutar o bien de forma autónoma o bien formando parte de otra. Por su parte, GNOME[5] utiliza por debajo su propia implementación de CORBA, llamada ORBit, para permitir que las aplicaciones se comuniquen con el escritorio y entre ellas. Sin embargo GNOME no ha conseguido que las aplicaciones puedan ser embebidas unas dentro de otras, tal y como sucede en KDE.

[4] `http://phil.freehackers.org/kde/kpart-techno/kpart-techno.html`
[5] `http://developer.gnome.org/arch/gnome/`

La comunidad Java está desarrollando ya la siguiente versión de componentes que sustituirá en un futuro a los Java Beans y Enterprise Java Beans. Esta versión está siendo desarrollada en el *Java Specification Requests* (JSR) 291, denominado *Dynamic Component Support for JavaTM SE*[6]. El objetivo es desarrollar una especificación para un *framework* de componentes dinámico basado en el modelo de componentes dinámico OSGi, que pueda ser integrado en el contexto de la máquina virtual de Java. Esta especificación definirá no sólo el modelo de componentes dinámicos sino también el ciclo de vida de los mismos dentro de la plataforma Java. Este modelo dinámico de componentes soportará el ensamblaje de componentes para la creación de aplicaciones así como la encapsulación de su implementación y el control de su ciclo de vida.

2.5 Componentes de Terceros: COTS

El desarrollo de aplicaciones basadas en componentes fue ideado con un objetivo muy claro: facilitar la reutilización del software, independientemente del origen del artefacto software (diseño propio o de terceros). De esta forma se conseguiría desarrollar programas mucho más rápido y con mayores garantías de que su funcionamiento sea correcto; siempre claro que los componentes que se ensamblan están probados con antelación. Debido a que los componentes pueden ser adquiridos de distintos proveedores, la integración y ensamblaje de componentes COTS (Commercial Off-The-Shelf) presenta su propia problemática, estudiada por la rama conocida como COSE.

Intuyendo el gran potencial del desarrollo COSE (disminución de costes, mantenimiento y tiempo de desarrollo e incremento de la fiabilidad, estabilidad y disponibilidad), el gobierno americano financió al SEI durante la década de los noventa para que profundizara en el establecimiento de esta disciplina. En ese período el SEI realizó numerosas publicaciones y workshops monográficos sobre COTS, que han sentado las bases del avance de COSE. Prueba del conocimiento adquirido se encuentra en la sección monográfica sobre desarrollo basado en COTS disponible en su página web `http://www.sei.cmu.edu/cbs/`.

El impacto del desarrollo COSE sobre el ciclo de desarrollo y vida de los productos es bastante profundo. No sólo debe cambiarse las actividades clásicas, como la especificación de requisitos para que tenga en cuenta el contexto, la arquitectura y los productos disponibles en el mercado, sino que también debe adaptarse el proceso de adquisición y contratación de componentes [52]. Torchiano [206] presenta los resultados de un estudio realizado en siete empresas del sector de las PyMEs, centrado en descubrir cómo

[6] JSR 291: `http://jcp.org/en/jsr/detail?id=291`

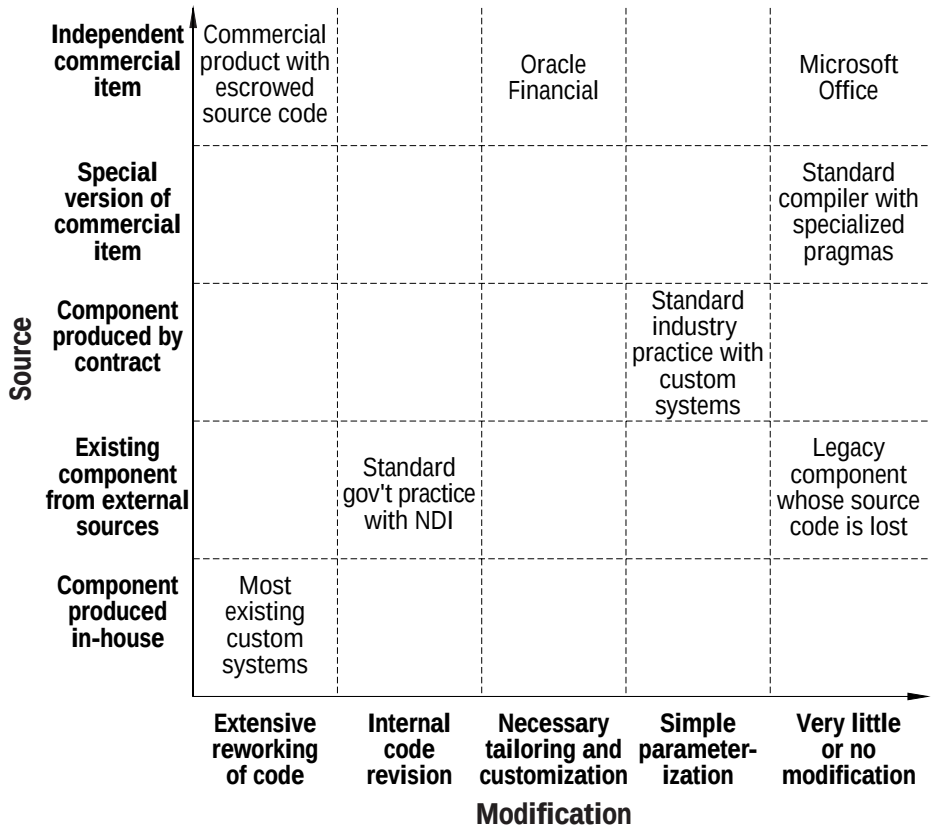
abordaban el diseño COSE, fundamentalmente la selección de componentes, experiencias con componentes COTS y elección de la arquitectura. Carney [53] propone una visión más profunda del componente COTS, estableciendo una taxonomía según su origen y posibilidad de modificación (ver cuadro 2.2).

En [153, 217] Wallnau y Meyers realizan, respectivamente, sendos estudios sobre el desarrollo de aplicaciones siguiendo un enfoque COSE y describen los principios fundamentales y las mejores prácticas para lograrlo, ilustrándolo con algunos ejemplos reales. Teschke propone en [202] un lenguaje unificado para la descripción de la estructura y el comportamiento de los componentes que proporciona una vista unificada de los principales modelos de componentes (EJB, COM y CCM). Siguiendo con el esfuerzo de estandarización, Franch [92] propone un marco de referencia que siente las bases para definir una ontología y terminología común y propia del desarrollo de sistemas COSE, que permita la realización de comparativas entre las distintas formas de creación de aplicaciones.

Por último, el SEI está desarrollando actualmente una iniciativa encaminada a predecir el comportamiento global de una aplicación desarrollada a partir de componentes certificados [152, 216] denominada **PACC**[7] (*Predictable Assembly From Certifiable Components*). El SEI quiere desarrollar una tecnología para predecir el comportamiento de una aplicación formada por la composición de componentes *COTS*, de forma que sea posible reducir el riesgo asociado al diseño. También están desarrollando nuevas tecnologías de componentes, extendiendo las teorías de propiedades [113] y diseñando las correspondientes herramientas.

Si bien los procesos de selección, integración y mantenimiento de los sistemas basados en componentes COTS resultan más complejos que cuando se utilizan componentes de desarrollo propio [121], por lo general, los primeros consiguen una mayor reducción de los tiempos de desarrollo y el coste final de los productos, por lo que la política de «comprar en lugar de construir» resulta cada vez más popular entre las empresas desarrolladoras de software. El cuadro 2.3 resume algunas de las ventajas e inconvenientes de los componentes COTS frente a los de desarrollo propio [5]. Carney define un método para evaluar el riesgo asociado a la utilización de productos *COTS* denominado **CURE** (*COTS Usage Risk Evaluation*) [54], cuyo objetivo es ayudar a elegir entre desarrollar el componente o buscarlo en el mercado.

[7] `http://www.sei.cmu.edu/pacc/index.html`



Cuadro 2.2: Clasificación de componentes COTS (extraído de [53])

COMPONENTES DE DESARROLLO PROPIO	COMPONENTES COTS	IMPLICACIONES DEL USO DE COMPONENTES COTS
• La persona encargada de su implementación conoce la arquitectura del sistema y el modelo de componentes subyacente, por lo que los construye para que sean compatibles con ellos. • Suministran exactamente la funcionalidad requerida ya que se construyen a medida. • Evolucionan cuando y como convenga ya que el código fuente está disponible y puede ser fácilmente modificado. • Están ampliamente documentados y pueden validarse fácilmente.	• Los fabricantes de componentes COTS desconocen la arquitectura de los sistemas concretos en los que posteriormente éstos se integrarán, aunque asumen algunas generalidades. • Por lo general proporcionan más funcionalidad de la estrictamente requerida por el cliente. • Es el vendedor el único que decide cómo y cuándo éstos evolucionan ya que por lo general el usuario no tiene acceso al código fuente o, si lo tiene, su modificación no está exenta de riesgos. • Pueden no estar completa y/o correctamente documentados.	• Puesto que los componentes COTS no suelen ser modificables, debe ser el sistema, y en particular su arquitectura, la que se acomode para permitir su integración. • La funcionalidad extra debe ser convenientemente ocultada. • A veces los clientes se ven obligados a actualizar los componentes sin desearlo. Otras, el vendedor no actualiza el componente como desearía el usuario. • La validación debe ser cuidadosa y exhaustiva.

Cuadro 2.3: Componentes de desarrollo propio frente a componentes COTS (extraído de [5])

2.6 CONCLUSIONES Y APORTACIONES A LA TESIS

UNO DE los objetivos fundamentales de esta Tesis Doctoral es desarrollar aplicaciones de control de robots basadas en componentes, por lo que el estudio de esta tecnología es vital para cumplir este objetivo. En este capítulo se ha mostrado que la falta de consenso en las cuestiones fundamentales de esta tecnología (qué es un componente, cómo se diseña y cómo se usa) ha evitado la aparición de una verdadera tecnología de componentes. En su lugar, los mayores fabricantes han diseñado y promocionado sus propias tecnologías que, como no podía ser de otra forma, son incompatibles entre sí, a pesar de que uno de los objetivos perseguidos por la tecnología de componentes es la interoperabilidad entre distintos componentes. Kang realiza un estudio [125] sobre la reutilización software cuyas conclusiones se van a utilizar para terminar el capítulo actual sobre componentes:

- La reusabilidad debe ser una característica inherente al software. La reutilización de software que no ha sido desarrollado con este objetivo en mente es técnicamente más difícil, más costosa y menos eficiente.
- El desarrollo de elementos software reutilizables requiere de la determinación de arquitecturas. Una arquitectura define cómo los elementos software se integran para crear un sistema, de forma que contar con arquitecturas estandarizadas permite definir los contextos en los que los elementos reutilizables pueden ser desarrollados.
- El desarrollo de arquitecturas estándares y de elementos software reutilizables requiere la comprensión de las características comunes y de las que pueden variar en los sistemas que conforman un dominio.

En cuanto a las aportaciones al desarrollo de esta Tesis Doctoral, se ha decidido adoptar la definición de componente de UML 2.0 [170]: «un componente es una unidad auto-contenida que encapsula el estado y el comportamiento de un conjunto de *Classifiers*. Un componente especifica un contrato formal de los servicios que proporciona a sus clientes y de los que requiere de ellos, en forma de interfaces provistas y requeridas. Un componente es una unidad sustituible, que puede ser cambiada tanto en tiempo de compilación como en tiempo de ejecución por otro componente que ofrece una funcionalidad equivalente, basada en la compatibilidad de sus interfaces». También se ha hecho patente la necesidad de desarrollar una infraestructura mínima común (un modelo de componentes) para que los componentes puedan comunicarse entre sí.

Vista la falta de consenso en las cuestiones fundamentales de la tecnología de componentes, en esta Tesis Doctoral se aboga por que el concepto «componente» sea tratado

desde un mayor nivel de abstracción y no como simple código fuente o binario. Como afirma Kang en su estudio, «la reusabilidad debe ser una característica inherente al software», y esto es algo que no puede conseguirse de forma casual ni, como se ha demostrado hasta ahora, desde el nivel del código fuente, por muy bien diseñado que esté. Es necesario que el concepto componente sea, como afirma ACRoSeT, independiente de la plataforma final de ejecución. MDE se perfila como la tecnología perfecta para tratar el componente de forma totalmente independiente de la tecnología de implementación final. En esta Tesis Doctoral se propugna que la verdadera reutilización software se va a conseguir cuando la unidad de reutilización sea el modelo, en vez de directamente el código fuente.

La aplicación de los conceptos enunciados en este capítulo se encuentra en los capítulos 6 y 7. El primero de ellos contiene el meta-modelo de componentes de V^{3}Studio. V^{3}Studio tiene una vista estructural que permite describir la arquitectura de una aplicación en base a los componentes que contiene. El modelado de las características de los componentes se ha realizado en base a lo descrito en este capítulo. Por otro lado, el capítulo 7 describe la transformación de modelos que se ha realizado entre V^{3}Studio y UML para traducir los conceptos básicos de CBD a un lenguaje de programación. Esta transformación no sólo embebe esta traducción, sino que también genera la infraestructura básica (modelo de componentes) necesaria para ejecutar estos componentes.

CAPÍTULO 3

ARQUITECTURA SOFTWARE

LA ARQUITECTURA software fue una de las primeras disciplinas que surgieron para hacer frente a la creciente complejidad del desarrollo software a principios de los ochenta. Esta disciplina reconoce que desarrollar software es complejo y que se necesita una estructura, una guía para abordarlo. Y reconoce también que dicha guía afecta sobremanera a muchas de las características del programa que se genera, tanto funcionales (e.g. ocupación memoria y tiempo de respuesta) como no funcionales (e.g. mantenimiento y evolución). Cualquier desarrollo software de entidad necesita el soporte proporcionado por un estudio arquitectónico. Así como en el capítulo anterior se hizo hincapié en la forma de desarrollar unidades reutilizables, en este capítulo se va describir la infraestructura necesaria para conseguir este objetivo.

> *Software architecture overlaps and interacts with the study of software families, domain-specific design, component-based reuse, software design, specific classes of components, and program analysis. Trying to separate these areas rigidly isn't productive.*
>
> — Mary Shaw & Paul Clements [197]

En este capítulo se describen, además las principales características de calidad que se persiguen cuando se diseña la arquitectura software de un sistema y presenta diversos estudios sobre su influencia en el mismo. Posteriormente se describen las principales técnicas y herramientas que ayudan al arquitecto: lenguajes de descripción de arquitecturas, patrones de diseño y vistas. Por último, se repasan las principales técnicas de evaluación de una arquitectura que existen, teniendo en cuenta que no existe la arquitectura perfecta que permita a un sistema cumplir todos los requisitos de calidad que se le piden.

3.1 INTRODUCCIÓN Y ALCANCE

DESDE que se comenzó a profundizar en el estudio de la estructura del software, a principios de los años ochenta, la *arquitectura software* no ha dejado de aportar resultados que permiten mejorar la calidad de los programas y que la han hecho merecedora de un puesto de honor dentro de la *Ingeniería del Software*. Gracias a las herramientas y técnicas desarrolladas por la arquitectura software se ha podido aumentar el nivel de abstracción con el que se diseña el software y se ha pasado de los informales diagramas de cajas y líneas a disponer de catálogos[1] [47] que muestran las arquitecturas más adecuadas según la aplicación que se quiere construir. Como ya observaron en su día Dijkstra y Parnas, «*no es suficiente que un programa produzca el resultado correcto. Existen otras características importantes, como fiabilidad y facilidad de mantenimiento, que también son importantes y que pueden ser logradas mediante una adecuada estructuración del software*». La siguiente frase plasma la importancia que tiene la arquitectura de un sistema:

> *Software architecture forms the backbone for any successful software-intensive system. An architecture is the primary carrier of a software system's quality attributes such as performance or reliability. The right architecture – correctly designed to meet its quality attribute requirements, clearly documented, and conscientiously evaluated – is the linchpin for software project success. The wrong one is a recipe for guaranteed disaster.*
>
> — The Software Engineering Institute

Han pasado más de veinte años desde que comenzó el estudio de la arquitectura del software, partiendo de los lenguajes de interconexión de módulos (*Module Interconnection Languages*, **MIL**) [188] hasta los modernos lenguajes de descripción de arquitectura (Architecture Description Language (**ADL**)) [144]. Gracias, sobre todo, al esfuerzo del Software Enginnering Institute (**SEI**) y otras organizaciones, la arquitectura del software se considera hoy en día una disciplina relativamente madura, que, como muestra la figura 3.1, ha pasado por una larga evolución en estas últimas décadas. Prueba del innegable éxito de esta nueva disciplina dentro de la Ingeniería Software son los dos números monográficos especiales que la revista *IEEE Software* le dedicó en los años 1.996 y 2.006, la gran cantidad de material bibliográfico, cursos y conferencias que existen (en [135] se puede encontrar una lista exhaustiva de ellas) y su inclusión en los planes de estudio de ciertas carreras en algunas universidades.

El SEI mantiene una web[2] dedicada exclusivamente al estudio de la arquitectura

[1] Manual *online* mantenido por Grady Booch en `http://www.booch.com/architecture/`

[2] `http://www.sei.cmu.edu/architecture/`

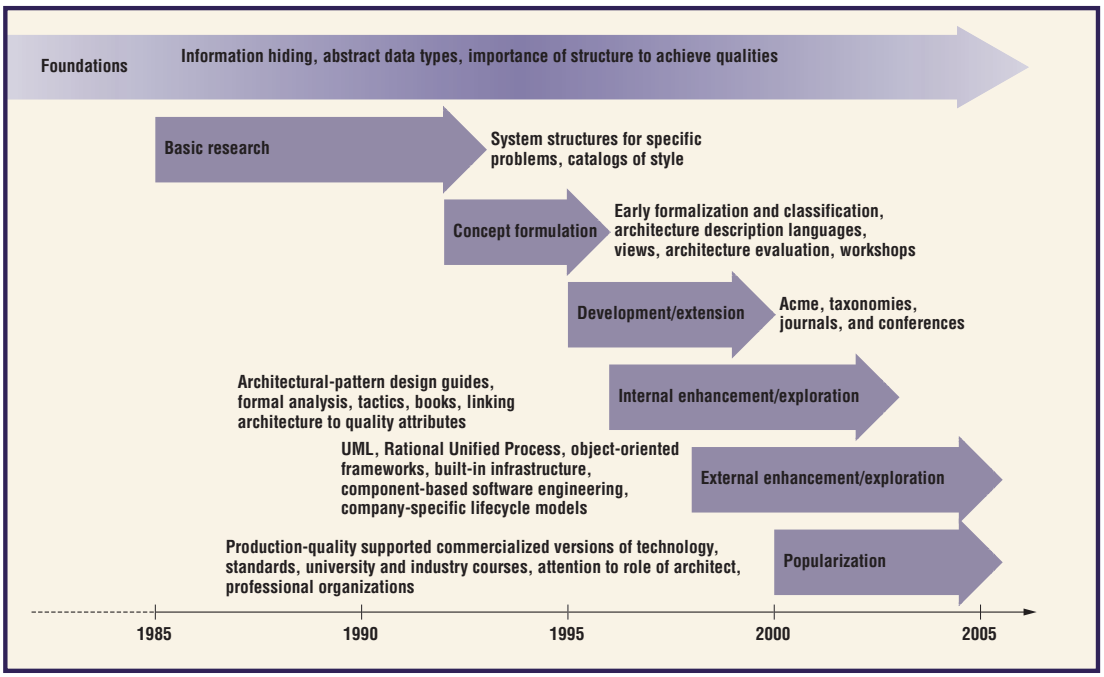


Figura 3.1: Evolución del estudio de la arquitectura software [197]

del software, en la que se puede encontrar todo el material producido por el instituto en los últimos años. Según Shaw [197], en un futuro cercano la arquitectura software habrá obtenido el estatus de toda tecnología verdaderamente exitosa: se dará por hecho que se sigue una.

El resto de la sección presenta algunas de las definiciones de arquitectura existentes en la literatura, haciendo especial hincapié en que en el diseño arquitectónico influyen muchos factores a parte del puramente funcional. El diseño de la arquitectura de un sistema es un compromiso entre los distintos requisitos del sistema y las entidades participantes en el mismo. La sección concluye haciendo un breve repaso del concepto «calidad» aplicado en el contexto de la arquitectura software.

3.1.1 Definición de arquitectura software

El repaso del estado de la técnica en arquitecturas software comienza con la definición de qué se entiende por «arquitectura». Como sucede con el caso de «componente», en la literatura existen numerosas definiciones. Sin embargo, en este caso el SEI mantiene en su página una sección[3] con todas las definiciones que han ido apareciendo hasta el momento. Por tanto tan sólo se expondrán algunas de las más relevantes:

[3] `http://www.sei.cmu.edu/architecture/published_definitions.html`

ANSI-IEEE 1471 (2.000) [203]. La arquitectura es la organización fundamental de un sistema, plasmada en sus componentes, las relaciones entre ellos y con su entorno, y los principios que guían su diseño y evolución.

SEI (2.003) [23]. La arquitectura software de un sistema es la estructura o estructuras del sistema que comprende el conjunto de elementos software, las propiedades externamente visibles de esos elementos y sus relaciones.

Institute for Software Research, University of California (UCI). La arquitectura no es sólo un paso o una actividad en el ciclo de desarrollo software, sino que impregna todas sus fases. La arquitectura es el conjunto de decisiones de diseño «principales» del sistema. La elección de los requisitos «principales» corresponde a las personas o entidades involucradas en él, tanto desarrolladores como clientes o promotores.

Rapid Object-oriented Process for Embedded Systems (ROPES) (1.999) [77] . Arquitectura es el conjunto de decisiones estratégicas de diseño que afectan la estructura, comportamiento o funcionalidad del sistema.

La definición dada por el estándar *IEEE-Std-1471* intenta abarcar, además, una variedad de usos del término «arquitectura», reconociendo sus elementos comunes. El concepto principal que subyace en esta definición es la necesidad de comprender y controlar los elementos del sistema que capturan la utilidad, coste y riesgo del mismo. En algunos casos, estos elementos son componentes físicos y sus relaciones. En otros, estos componentes no son físicos, sino lógicos. O incluso pueden referirse a los principios o patrones que crea la gerencia de la organización.

Otro aspecto de la arquitectura de un sistema es que existen otras personas o entidades, a parte del propio equipo de desarrollo, que pueden modificar el diseño de la arquitectura de la aplicación. Las necesidades particulares de cada una de las entidades involucradas pueden forzar un diseño que favorezca determinadas características, como muestra la figura 3.2. Al final, el arquitecto tiene que llegar a una solución de *compromiso* para diseñar la arquitectura [23].

Actualmente la investigación en arquitectura software tiene tres objetivos: (1) obtener un catálogo de patrones arquitectónicos (o estilos arquitectónicos para algunos autores) que expongan las mejores soluciones a problemas repetitivos y cómo identificarlos y aplicarlos a un sistema; (2) obtener, de forma más o menos automatizada, la arquitectura que mejor se adapta a los requisitos de calidad impuestos al sistema y (3) mantener de forma coherente la vista arquitectónica y de implementación, y de esta forma evitar que la vista arquitectónica se convierta, en el mejor de los casos, en simple documentación del proyecto.

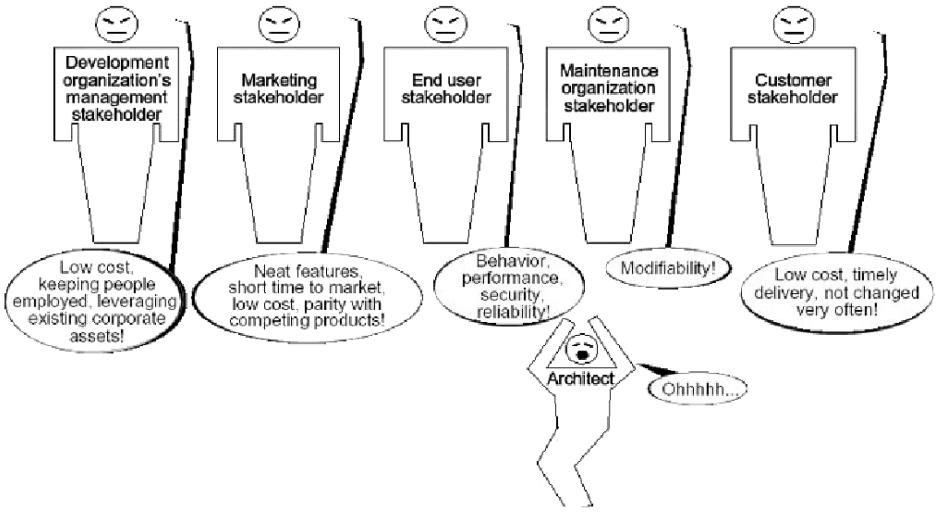


Figura 3.2: Influencia de entidades implicadas en el diseño de la aplicación [23]

3.1.2 Alcance y atributos de calidad

La arquitectura forma parte del núcleo de todo sistema software. Marca sus características y afecta a toda su estructura. La arquitectura no fuerza una determinada implementación, pero la condiciona, ya que esta deberá adaptarse a la descomposición del sistema y a las relaciones que se hayan definido entre los distintos subsistemas [198]. La arquitectura condiciona la organización y gestión del proyecto: define una descomposición estructural del sistema que hace posible la división del proyecto en paquetes de trabajo, que se asignarán a diferentes equipos de desarrollo, facilitando, pero también condicionando la planificación y la asignación de recursos. El reverso de la moneda es que una modificación de la arquitectura puede producir un enorme impacto sobre la gestión del proyecto, pues supone la reestructuración de los grupos de trabajo y la reasignación de recursos y responsabilidades.

La arquitectura influye también en los atributos de calidad del sistema, mejorando unos a costa de empeorar otros. Los atributos de calidad no existen de forma independiente, sino que interaccionan entre sí [32, 88]. La arquitectura adoptada finalmente puede verse como el consenso sobre los atributos de calidad del sistema al que llegan todas las partes implicadas, ya que generalmente es imposible contentar a todas las partes consiguiendo todos los objetivos. Las mayoría de las decisiones arquitectónicas se suelen tomar en las primeras etapas de la vida de un proyecto y condicionan todo el posterior proceso de desarrollo [139], por lo que tienen que ser cuidadosamente tomadas. Obviamente, la consecución de los atributos de calidad de un sistema no depende sólo de la arquitectura. Es también función de otras muchas variables. La gestión del proyecto, la elección de lenguajes

de programación y de algoritmos apropiados y la calidad de la implementación influyen también de forma decisiva en el resultado final. Pero aunque una buena arquitectura no sea condición suficiente para asegurar la calidad del sistema, es sin ninguna duda una condición necesaria. La arquitectura de un sistema forma el *núcleo del desarrollo* del mismo, alrededor del cual giran el resto de las fases [198]. La arquitectura sirve de eje central que enlaza todos los pasos: especificación de requisitos, diseño de la aplicación, documentación, evaluación y evolución.

Por último, resta por aclarar el concepto de «calidad» software, ya que se ha afirmado que la arquitectura del sistema se diseña de forma que cumpla con determinados atributos de calidad, pero no se ha dicho qué se entiende por calidad ni cuáles son sus atributos. El estándar *[ANSI-IEEE 1061-1998]* [204] define calidad software como «el grado en que el software posee una combinación deseada de atributos». En cuanto a sus atributos, el cuadro 3.1 muestra los atributos de calidad software definidos tanto por el estándar *IEEE 1061-1998* como por el *[ISO/IEC 9126-1:2001]* [205], mientras que en el cuadro 3.2 se pueden encontrar los definidos por el *MITRE* [154] para el *Department of Defense* (**DoD**).

3.2 DISEÑO ARQUITECTÓNICO DEL SOFTWARE

COMO YA se desribió, la arquitectura de un sistema es una descripción de alto nivel de los principales bloques funcionales que forman la aplicación y de sus patrones y protocolos de comunicación. La arquitectura software adoptó, desde un principio, el concepto de *componente* como bloque básico que encapsula la funcionalidad «local». El componente fue dotado de *puertos* para controlar sus interacción con el exterior (el resto de componentes que forman la aplicación) y definieron los *conectores* para abstraer los distintos canales y protocolos de comunicación entre componentes (siempre a través

Funcionalidad	**Matenibilidad**	**Fiabilidad**
▷ Integridad	▷ Corrección	▷ Libre de fallos
▷ Corrección	▷ Ampliación	▷ Tolerancia a errores
▷ Seguridad	▷ Comprobabilidad	▷ Disponibilidad
▷ Compatibilidad	**Portabilidad**	**Usabilidad**
▷ Interoperabilidad	▷ Independencia HW	▷ Facilidad entender
Eficiencia	▷ Independencia SW	▷ Facilidad aprendizaje
▷ Economía temporal	▷ Facilidad instalación	▷ Operabilidad
▷ Economía en recursos	▷ Reusabilidad	▷ Facilidad comunicación

Cuadro 3.1: Atributos de calidad del software (I) (*[IEEE 1061]* [204] e *[ISO/IEC 9126-1]* [205])

▷ Usabilidad	▷ Corrección	▷ Mantenimiento	▷ Verificabilidad	▷ Expansión
▷ Flexibilidad	▷ Interoperabilidad	▷ Portabilidad	▷ Reusabilidad	▷ Supervivencia
▷ Eficiencia	▷ Integridad	▷ Fiabilidad		

Cuadro 3.2: Atributos de calidad del software (II) (extraídos del *MITRE* [154])

de sus puertos). La línea que separa la investigación en la orientación de componentes y en arquitectura es tan difusa que se confunden a menudo, sobre todo cuando se trata de los términos básicos, ya que existe un fuerte acoplamiento entre ambas disciplinas. En este sentido, la arquitectura es acorde y sinérgica con el enfoque de orientación a componentes y contribuye de manera definitiva a su difusión y definición.

Cuando se describieron los modelos de componentes en la sección 2.4 y, concretamente, el patrón de diseño «basado en componentes», se mencionó que lo que diferenciaba a un componente de cualquier otra forma de software empaquetado era la existencia de un modelo de componentes que condicionaba el diseño y proporcionaba servicios extra. En dicha sección se hacía referencia explícita a la existencia de un plan de alto nivel que controlaba y establecía el diseño: la arquitectura del aplicación.

Esta sección está organizada en cinco apartados, en los que se realiza un breve resumen de las principales teorías y tecnologías que se han desarrollado para realizar el diseño de la arquitectura software de una aplicación. Los dos primeros se centran en describir la forma en que los componentes que forman la arquitectura del sistema se comunican entre sí (conectores) y en describir las principales formas de agrupar estos componentes y repartir sus responsabilidades (patrones arquitectónicos). Posteriormente se estudia cómo representar la arquitectura de un sistema, de forma que sea fácilmente inteligible por otro desarrollador y documente fielmente el desarrollo. La sección acaba presentando los lenguajes de descripción de arquitectura que surgieron a lo largo de la década de los noventa para estudiar con más detalle la arquitectura de un sistema y se termina presentando otros estudios y herramientas más recientes para desarrollar la arquitectura de un programa.

3.2.1 Comunicación entre componentes: conectores

Si los componentes, como artefactos que contienen parte de la funcionalidad de un sistema, son uno de los conceptos principales del desarrollo basado en componentes, no son menos importantes los artefactos que permiten que se comuniquen entre sí utilizando distintos patrones de interacción: los conectores. En este apartado se presentan, de forma independiente y complementaria al estudio pormenorizado que realizan los lenguajes de

descripción de arquitecturas (ADLs, ver apartado 3.2.4), algunos trabajos que estudian la comunicación entre componentes y sus patrones de interacción. Las razones que llevan a tratar de forma distinta conectores de componentes son las siguientes:

- La complexidad de las interacciones entre componentes hace que sea deseable su separación de la funcionalidad propia del componente.
- Los conectores tienen una naturaleza abstracta. Pueden ser parametrizados para dar lugar a conectores ligeramente diferentes e instanciados gran cantidad de veces en un mismo sistema.
- Los componentes deben ser independientes del resto de su entorno, mostrando para ello unas interfaces bien definidas que los aislen del sistema.
- La separación entre computación y coordinación permite que los sistemas evolucionen y se adapten a cambios en la política de comunicación, sin que se interfiera en la computación llevada a cabo por los componentes. También facilita que el sistema se pueda amoldar a nuevos componentes.
- Las relaciones entre componentes no son estáticas, de forma que el uso de conectores hace posible que un componente se utilice con distintos tipos de conectores.
- El tratamiento de conectores como entidades de primera clase permite la definición de nuevos tipos de conectores usando mecanismos como la herencia o la composición.

La posibilidad de estimar el rendimiento futuro de una aplicación software en tiempo de diseño puede reducir sobremanera el coste y el riesgo inherente al desarrollo. Gomaa investiga en [103] el diseño y modelado de los principales patrones de interacción entre componentes, siguiendo el paradigma cliente-servidor. En este caso, Gomaa utiliza modelos UML anotados de los patrones de interacción, de forma que sea posible verificar sus características mediante posterior análisis.

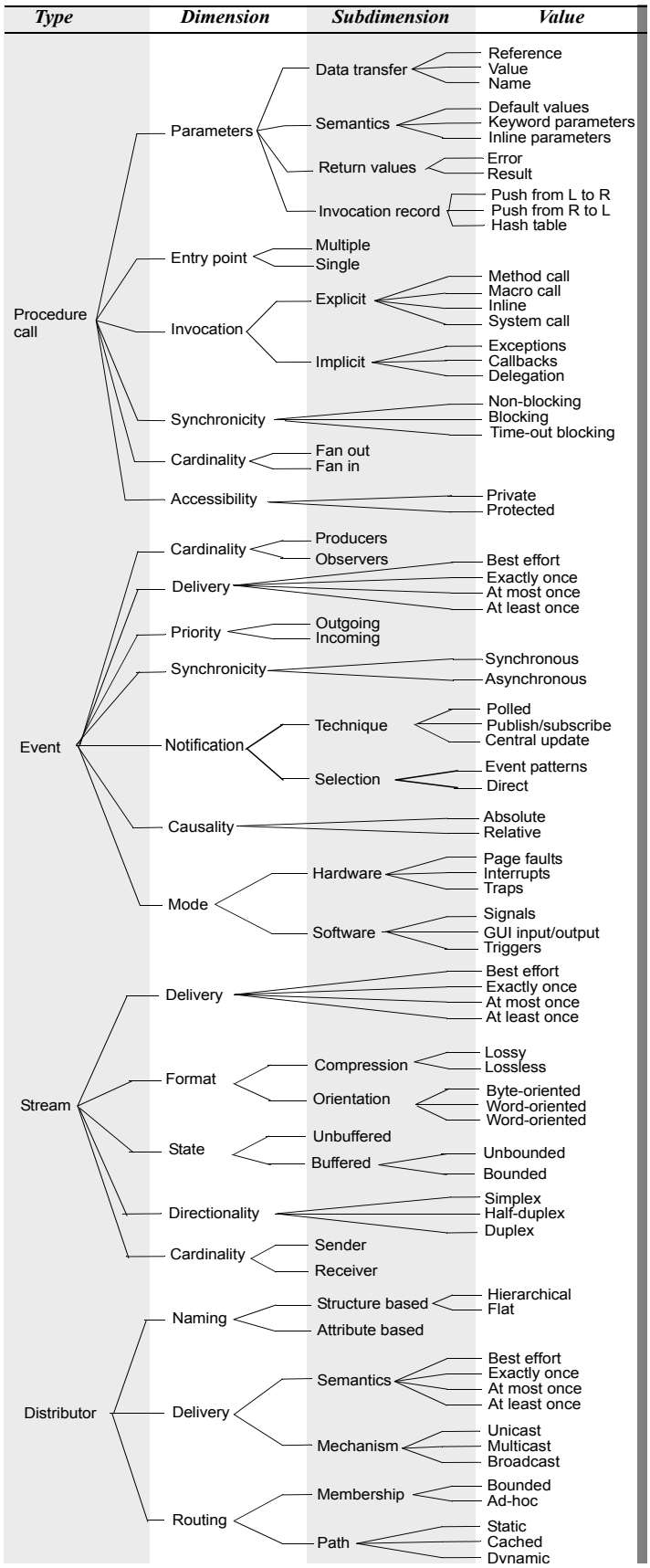
El «diseño por contrato», desarrollado por Meyer [151], se basa en el concepto paralelo de contrato entre humanos. Un contrato especifica claramente las obligaciones de ambas partes, cliente y servidor, en forma de (1) pre-condiciones, esto es, condiciones que tienen que cumplir cliente y servidor antes de que se realice la transacción, (2) invariantes que tienen que verificarse en el estado de la partes implicadas antes y después de la ejecución del contrato y (3) post-condiciones. Beugnard [29] aplicó el concepto de contrato al diseño de sistemas basados en componentes especificando las relaciones entre ellos por medio de contratos. Asimismo identificó cuatro niveles de detalle en que el contrato podía especificar la intercomunicación entre componentes: básico (semántica únicamente, coincide con la invocación de métodos); de comportamiento (especificación de invariantes, pre y

post-condiciones); de sincronización y, por último, de calidad de servicio (ver figura 3.3). Andrade [11, 12] también propone en sus estudios sobre arquitectura y comunicación de sistemas software el uso de contratos como mediadores de las relaciones entre componentes, que facilitan la corrección y evolución de los sistemas. Concretamente, en [11] estudia la utilización de las clases de asociación de UML como mecanismo para expresar los contratos entre clases.

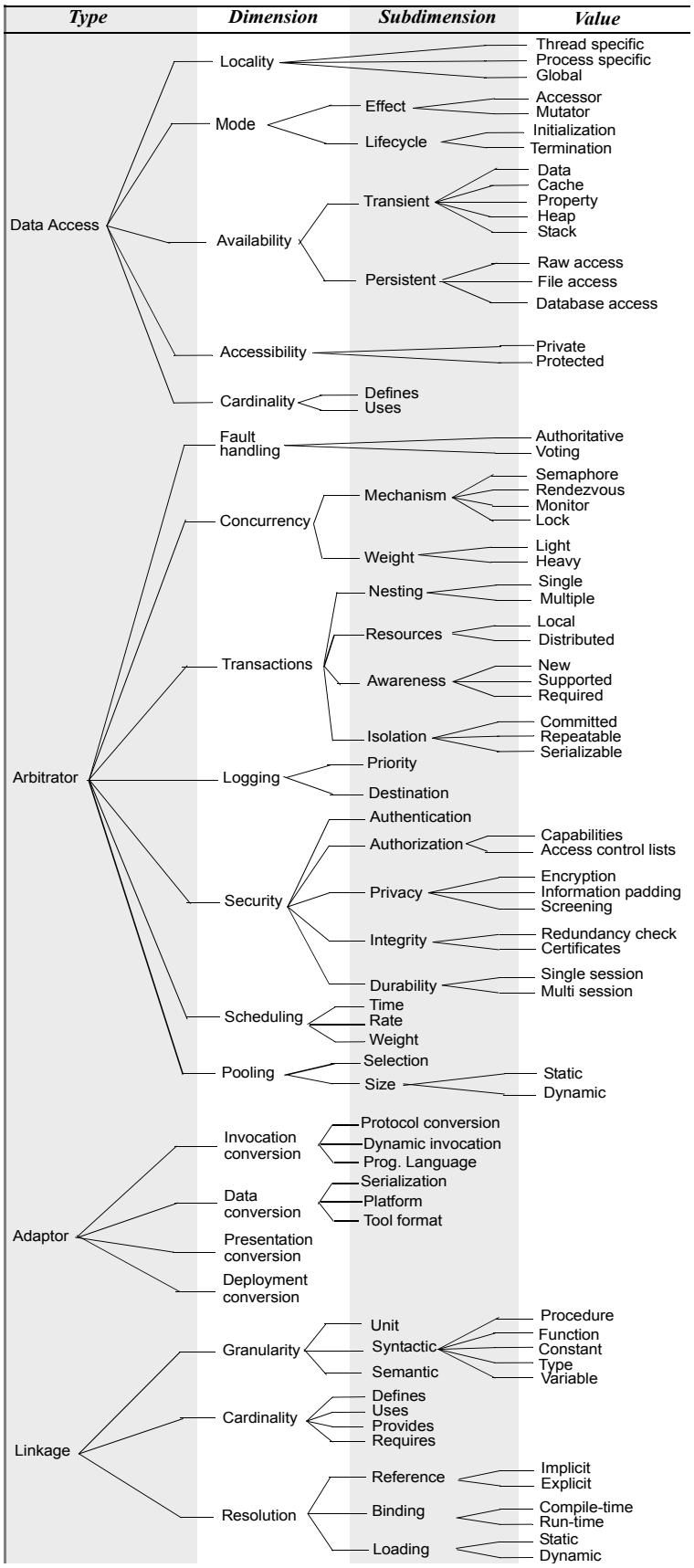
Conscientes de la importancia que tienen los patrones de comunicación entre componentes, Mehta realiza en [145, 146] sendos estudios pormenorizados en los que trata de establecer una taxonomía de conectores, partiendo de estudios anteriores. Los resultados de este estudio pueden consultarse en los cuadros 3.3 y 3.4. Como puede observarse, existen ocho categorías principales, cuyas características son descritas con mayor grado de detalle por medio de sus «dimensiones» y «sub-dimensiones». Una particularidad del estudio es que es posible combinar las «dimensiones» de un conector para crear conectores más complejos, aunque no todas las combinaciones son posibles, tal y como muestran los autores en la matriz de compatibilidad descrita en [145].

En otro trabajo posterior, Roshandel [189] identifica cuatro niveles en que se puede modelar la comunicación entre componentes, que ordenados en grado ascendente de detalle son: declaración de interfaces (usando IDLs), descripción estática del comportamiento (mediante contratos [151]), descripción dinámica del comportamiento (usando máquinas de estado [108] o **CSP** [114]) y descripción de los protocolos de comunicación. El autor declara que la sustitución de componentes basada únicamente en la igualdad de sus interfaces no garantiza la interoperabilidad entre componentes y defiende la necesidad de describir con mayor detalle las relaciones entre estos niveles.

Por último, cabe mencionar el enfoque **SOFA** (*Sofware Appliances*) [44, 48], en el que se realiza otro estudio pormenorizado de las características y la estructura interna de diversos tipos de conectores. SOFA identifica cuatro formas de comunicación entre componentes: llamada a procedimiento, paso de mensaje, *streaming* y pizarra. La estructura interna de los conectores que utiliza SOFA contiene distintos elementos, como un encriptador, monitor, sincronizador, etc., que aumentan sobremanera las posibilidades que exhiben estos conectores. El código de los conectores puede ser generado de forma automática de acuerdo a las características que se hayan elegido.



Cuadro 3.3: Taxonomía de conectores software (I) (extraída de [145])



Cuadro 3.4: Taxonomía de conectores software (II) (extraída de [145])

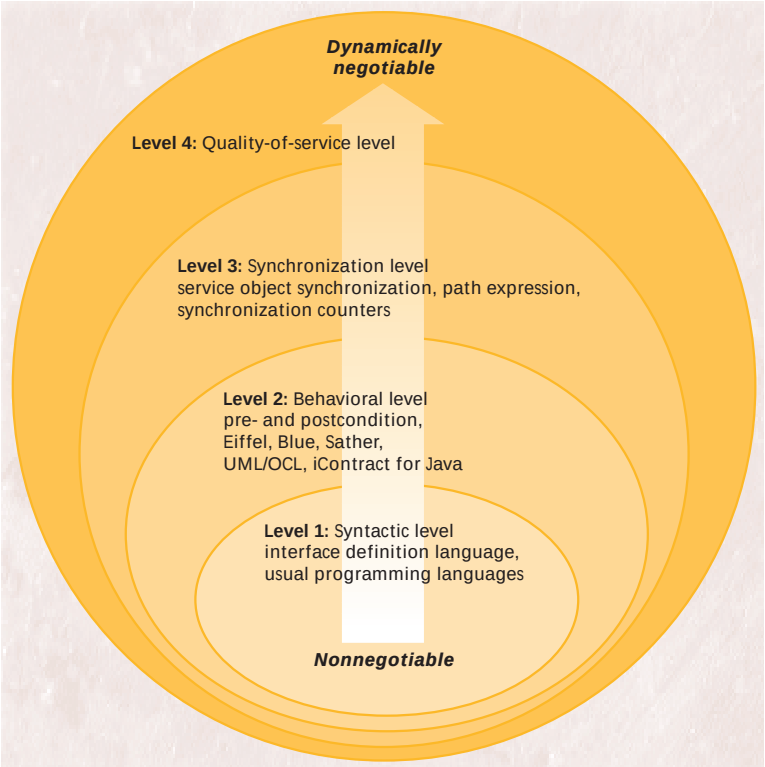


Figura 3.3: Las cuatro clases de contratos entre componentes (extraído de [29])

3.2.2 PATRONES ARQUITECTÓNICOS

Cuando se está diseñando la arquitectura de un sistema es frecuente encontrarse con problemas que ya se han resuelto antes. Aunque el contexto suele ser diferente (otro sistema operativo, otro lenguaje de programación), la sensación de *déjà vu* arquitectónico permanece. Sin embargo, generalmente el diseñador no es capaz de reutilizar el conocimiento adquirido y acaba realizando de nuevo el mismo diseño, pero partiendo de cero (lo que los anglosajones denominan *reinventing the wheel*). Imitando los catálogos de soluciones existentes en otras ingenierías, Gamma propuso en su libro *Design Patterns* [96] (1.995) un conjunto de *patrones de diseño* para solucionar algunos de los problemas más frecuentes:

> *A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
>
> — Christopher Alexander (*Timeless Way of Building*)

De forma general, Gamma describe en [96] cuatro aspectos fundamentales de cada patrón y sienta las bases para la elaboración de futuros catálogos. Describe su *nombre*, el

problema y el contexto en que aparece, la *solución* (los elementos involucrados, sus relaciones, responsabilidades y las colaboraciones) y, por último, las *consecuencias* de la aplicación del patrón (impacto sobre los atributos de calidad del sistema, interacciones con otros patrones y otras posibles implicaciones).

Aunque numerosos autores han defendido la necesidad de disponer de catálogos de patrones o estilos arquitectónicos [22, 198], todavía no se ha publicado el libro de referencia. El primer estudio es el presentado por Buschmann [47] en el año 1.996, en el que los autores describen patrones que comprenden distintos niveles de abstracción, desde el existente en los niveles de descripción arquitectónica hasta algunos *idioms*[4], pasando también por los patrones de diseño. Posteriormente se publicaron otros compendios de patrones arquitectónicos. Schmidt [190] presenta un catálogo especializado en sistemas concurrentes y distribuidos, mientras que Gamma [102] propone diversos patrones para definir la arquitectura de una línea de producto. La serie *Pattern-Oriented Software Architecture* continuó con la aparición de tres nuevos volúmenes que describen el uso de patrones para el manejo de recursos y repositorios de información [129], para diseñar sistemas de computación distribuida [45] y para el diseño de lenguajes de patrones [46]. Grady Booch mantiene incluso un catálogo *online* en `http://www.booch.com/architecture/`. El mundo del desarrollo de sistemas de tiempo-real tiene también su catálogo de patrones gracias a Douglass [78].

De la gran cantidad de información existente se puede deducir que existen una gran cantidad de patrones arquitectónicos, dependiendo de la aplicación que se esté desarrollando y de cuáles son las características (requisitos) más importantes. Así pues, a modo de ejemplo, se enumeraran los patrones arquitectónicos clásicos (y más utilizados) del primer volumen de la saga *Pattern-Oriented Software Architecture*: distribución arquitectónica por capas (*layers*), tubería y filtro (*pipes and filters*), arquitecturas de pizarra (*blackboard*), intermediario (*broker*), modelo-vista-controlador (*model-view-controller*, **MVC**), micronúcleo (*microkernel*) y reflexión (*reflection*) [47].

> *Standard architectures exist for countless domains and applications. For example, nobody will ever again have to design from scratch a banking system, an avionics system, a satellite ground system, or a Web-based e-commerce system.*
>
> *Where total architectural solutions don't yet exist, partial ones certainly do in the form of catalogs of architectural patterns and tactics that help solve a myriad of problems and achieve various quality attributes.*
>
> — Mary Shaw & Paul Clements [197]

[4] Patrones de bajo nivel de abstracción, generalmente específicos de lenguaje de programación en particular.

3.2.3 VISTAS DE UNA ARQUITECTURA

En cuanto el desarrollo de la arquitectura software permitió plasmar en papel los diseños, surgió el problema de que un único diagrama no era suficiente para mostrar toda la información que se necesitaba analizar. Lo único que se consigue con un sólo diagrama es sobrecargar de información el diagrama, lo cual no sólo dificulta enormemente su interpretación sino que además da pie a cometer muchos errores. Kruchten se dió cuenta de que en estos diagramas se solían mezclar distintos tipos de información y de que no se capturaban todos los aspectos esenciales para todas las personas/entidades involucradas en el proyecto, por lo que propuso *«the "4+1" View Model of Software Architecture»* [134]. El modelo **"4+1"** utiliza distintas vistas concurrentes para modelar cada uno de los aspectos de la arquitectura de una aplicación.

Aún cuando existe un consenso general acerca de la necesidad de representar la arquitectura utilizando diferentes vistas, tal consenso desaparece cuando hay que definir cuáles son esas vistas. El estándar *ANSI-IEEE 1471-2000* [203] define de alguna manera los puntos de vista que deben tenerse en consideración para describir la arquitectura, pero a un nivel de abstracción tan alto que las vistas pueden seleccionarse o definirse siguiendo criterios muy diferentes. Ejemplos de otros modelos que definen puntos de vista explícitos, que además son similares a los definidos por el **"4+1"** de Kruchten, son el propuesto por Hofmeister en [115] (propone cuatro vistas: vista conceptual, vista de módulos, vista de ejecución y vista de código) y ROPES [76, 77] (propone cinco vistas: vista de subsistemas y componentes, vista de concurrencia y recursos, vista de distribución, vista de fiabilidad y seguridad y vista de despligue).

Por su parte, UML 2.1.1 [177] no define puntos de vista explícitos, pero estos pueden deducirse de la semántica asociada a cada uno de sus diagramas. Esta vistas coinciden básicamente con las **"4+1"** vistas de Krutchen: en UML la vista física es referida como vista de despliegue, la de desarrollo como vista de componentes y la de escenarios es la de vista de casos de uso. Aunque UML no está pensado expresamente para modelar arquitecturas, existen diversos artículos en que se realizan estudios sobre la aplicabilidad de UML en esta área, utilizando básicamente el diagrama de componentes, y que pueden ser consultados en [30, 143]. Lange realiza en [137] un estudio sobre la aplicación de UML para describir la arquitectura en 14 casos de estudio de empresas.

Para finalizar esta sección resta por hablar del enfoque del SEI, denominado *"Views and Beyond"* (**V&B**) [60]. El enfoque del instituto es más cercano al que propone el estándar *ANSI-IEEE 1471-2000*, en el sentido de que no fija ni el número ni el contenido de las vistas. El enfoque **V&B** sostiene que para describir una arquitectura hay que elegir el conjunto de vistas relevantes, documentarlas por separado y luego recoger la información común a

dos o más vistas en un documento aparte. Este último aporta consistencia a la vistas y al documento, ya que enlaza todas las vistas entre sí y proporciona una visión integrada y coherente de la arquitectura, en contraposición a «las desacopladas instantáneas tomadas desde distintos ángulos» (haciendo referencia a los modelos de vistas fijas). **V&B** incluye un método para elegir las vistas más relevantes y plantillas para documentar las vistas y sus relaciones. Clements realiza en [59] un estudio comparativo entre **V&B** y el estándar *IEEE 1471-2000*.

3.2.4 LENGUAJES DE DESCRIPCIÓN DE ARQUITECTURA

Conforme el estudio de la arquitectura de los sistemas iba avanzando, se detectó la necesidad de definir un lenguaje para especificar los conceptos de alto nivel que maneja la disciplina, y de esta forma poder intercambiar diseños y razonar sobre ellos. Para permitir el desarrollo de la arquitectura software se necesitaban notaciones formales y herramientas de diseño y análisis que trabajaran a ese nivel de abstracción y que permitieran reutilizar los diseños. Así aparecieron los lenguajes de descripción de arquitectura o Architecture Description Language (**ADL**). Una de las primeras definiciones de ADL es la proporcionada por Vestal [212]: «un ADL se centra en la estructura de alto nivel global de la aplicación en vez de en los detalles de implementación de cualquier módulo de código».

Sin embargo, a pesar de todos los esfuerzos realizados, parecía haber poco consenso acerca de qué era un ADL, qué aspectos de la arquitectura debía modelar y cuál era el mejor ADL para modelar un problema en particular. Además, los ADLs eran comúnmente confundidos con especificaciones formales, lenguajes de interconexión de módulos (MIL), simuladores o lenguajes de programación. En este contexto, Medvidovic publicó en [144] un estudio comparativo de todos los ADLs existentes hasta el momento, en el que establecieron sus características y un marco en el que poder compararlos. Medvidovic concluye que, para que un lenguaje se pueda considerar un verdadero ADL, tiene que soportar las primitivas de modelado arquitectónico resumidas en el cuadro 3.5 así como proporcionar herramientas que soportaran: el uso del ADL desde distintas vistas; la evolución asistida y el análisis del diseño; la generación de código y soporte para modelar arquitecturas dinámicas. La conclusión del estudio fue una nueva definición de ADL: «un ADL se distingue de cualquier otra notación en que se centra explícitamente en describir conectores y configuraciones arquitectónicas del sistema».

La lista de ADLs creados hasta el día de hoy es muy extensa, por lo que sólo se mostrará un pequeño resumen de los más extendidos o los que presentan características especiales para modelar algunos sistemas, sobre todo sistemas de tiempo-real. Para ampliar la información se puede consultar la página web `http://www.sei.cmu.edu/`

Componente	**Conector**	**Configuración arquitectónica**
▷ Interfaces	▷ Interfaces	▷ Inteligibilidad
▷ Tipos	▷ Tipos	▷ Componibilidad
▷ Semántica	▷ Semántica	▷ Trazabilidad
▷ Restricciones	▷ Restricciones	▷ Heterogeneidad
▷ Evolución	▷ Evolución	▷ Escalabilidad
▷ Aspectos no funcionales	▷ Aspectos no funcionales	▷ Evolución
		▷ Dinamismo
		▷ Restricciones
		▷ Aspectos no funcionales

Cuadro 3.5: Características de modelado arquitectónico de un ADL (extraído de [144])

`architecture/adl.html`, mantenida por el SEI, y el artículo de Medvidovic [144], en el que se muestra un resumen comparativo de la mayoría. En los cuadros 3.6 y 3.7 se muestra una comparativa de muchos de los ADLs que se definen a continuación desde el punto de vista del soporte que ofrece el lenguaje para modelar componentes y conectores, los conceptos base de CBD.

ACME (1.995). Desarrollado por la CMU, es un lenguaje general de intercambio con otros ADLs o que puede usarse de base para crear otros nuevos.

Aesop (1.994). Desarrollado por la CMU, es un ADL de propósito general que hace especial énfasis en la especificación de la arquitectura siguiendo estilos arquitectónicos específicos, como el cliente-servidor o tubería y filtro.

C2 (1.996). Desarrollado por la UCI, es un ADL para sistemas heterogéneos altamente distribuidos y dinámicos, que utilizan comunicación por paso de mensaje.

Darwin (1.991). Desarrollado por el Imperial College, utiliza π-calculus para comprobar las características dinámicas del sistema.

MetaH (1.993). Desarrollado por Honywell especialmente para el desarrollo de sistemas de navegación y control, particularmente sistemas de aviónica y tiempo-real.

Rapide (1.990). Desarrollado en Stanford, se centra en el modelado y simulación del comportamiento dinámico expresado por la arquitectura. Permite especificar complejos protocolos de comunicación utilizando *secuencias de eventos parcialmente ordenados*.

SADL (1.995). Desarrollado en SRI International, este ADL formaliza la arquitectura de forma que luego se pueda probar los sucesivos refinamientos que se hagan sobre ella.

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Component*; implementation independent	interface points are *ports*	extensible type system; parameterization enabled with templates	no support; can use other ADLs' semantic models in property lists	via interfaces only	structural subtyping via the *extends* feature	allows any attribute in property lists, but does not operate on them
Aesop	*Component*; implementation independent	interface points are *input* and *output ports*	extensible type system	(optional) style-specific languages for specifying semantics	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with components
C2	*Component*; implementation independent	interface exported through top and bottom *ports*; interface elements are *provided* and *required*	extensible type system	component invariants and operation pre- and postconditions in 1st order logic	via interfaces and semantics; stylistic invariants	heterogeneous subtyping	none
Darwin	*Component*; implementation independent;	interface points are *services* (*provided* and *required*)	extensible type system; supports parameterization	π-calculus	via interfaces and semantics	none	none
MetaH	*Process*; implementation constraining	interface points are *ports*	Predefined, enumerated set of types	ControlH for modeling algorithms in the GN&C domain; implementation semantics via paths	via interfaces and semantics; modes; non-functional attributes	none	attributes needed for real-time schedulability, reliability, and security analysis
Rapide	*Interface*; implementation independent	interface points are *constituents* (*provides*, *requires*, *action*, and *service*)	extensible type system; contains a types sublanguage; supports parameterization	partially ordered event sets (posets)	via interfaces and semantics; algebraic constraints on component state; pattern constraints on event posets	inheritance (structural subtyping)	none
SADL	*Component*; implementation independent;	interface points are input and output *ports* (*iports* and *oports*)	extensible type system; allows parameterization of component signatures	none	via interfaces; stylistic invariants	subtyping by constraining supertypes; refinement via pattern maps	requires component modification (see Section 4.3.9)
UniCon	*Component*; implementation constraining	interface points are *players*	predefined, enumerated set of types	event traces in property lists	via interfaces and semantics; attributes; restrictions on players that can be provided by component types	none	attributes for schedulability analysis
Weaves	*Tool fragments*; implementation constraining	interface points are *read* and *write ports*; interface elements are *objects*	extensible type system; types are component *sockets*	partial ordering over input and output objects	via interface and semantics	none	allows association of arbitrary, uninterpreted annotations with components
Wright	*Component*; implementation independent;	interface points are *ports; port interaction semantics specified in CSP*	extensible type system; parameterizable number of ports and computation	not the focus; allowed in CSP	protocols of interaction for each port in CSP; stylistic invariants	via different parameter instantiations	none

Cuadro 3.6: Características del modelado de componentes de distintos ADLs (extraído de [144])

UniCon (1.995). Diseñado por la CMU específicamente para generar el código de interconexión de los componentes usando distintos protocolos de comunicación y definiendo una serie de restricciones entre ellos.

Weaves (1.991). Desarrollado por The Aerospace Corporation para el modelado de grandes flujos de datos en sistemas concurrentes y distribuidos, con restricciones de tiempo-real.

Wright (1.994). Diseñado por la CMU, está diseñado para el modelado y análisis del comportamiento de sistemas concurrentes, especialmente para la detección de *deadlocks*. Está basado en el lenguaje *Communicating Sequential Processes* (**CSP**) [114].

xADL (2.000). Desarrollado por el UCI para soportar expresamente las descripciones arquitectónicas en formato XML. Actualmente se está trabajando en la versión 2.0 [70].

AADL (2.004). Desarrollado conjuntamente por el SEI y SAE (*Society of Automotive Engineers*), es el primer ADL que se combina con un enfoque basado en modelos (ver

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Connector*; explicit	interface points are *roles*	extensible type system, based on protocols; parameterization via templates	no support; can use other ADLs' semantic models in property lists	via interfaces and structural for type instances	structural subtyping via the *extends* feature	allows any attribute in property lists, but does not operate on them
Aesop	*Connector*; explicit	interface points are *roles*	extensible type system, based on protocols	(optional) semantics specified using Wright	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with connectors
C2	*Connector*; explicit	interface with each component via a separate *port*; interface elements are *provided* and *required*	extensible type system, based on protocols	partial semantics specified via message filters	via semantics; stylistic invariants (each port participates in one link only)	context-reflective interfaces; evolvable filtering mechanisms	none
Darwin	*Binding*; in-line; no explicit modeling of component interactions	none; allows "connection components"	none	none	none	none	none
MetaH	*Connection*; in-line; allows connections to be optionally named	none	none; supports three general classes of connections: port, event, and equivalence	none	none	none	none
Rapide	*Connection*; in-line; complex reusable connectors only via "connection components"	none; allows "connection components"	none	posets; conditional connections	none	none	none
SADL	*Connector*; explicit	connector signature specifies the supported data types	extensible type system; parameterized signatures and constraints	axioms in the constraint language	via interfaces; stylistic invariants	subtyping; connector refinement via pattern maps	requires connector modification (see Section 4.3.9)
UniCon	*Connector*; explicit	interface points are *roles*	predefined, enumerated set of types	implicit in connector's type; semantic information can be given in property lists	via interfaces; restricts the type of players that can be used in a given role	none	attributes for schedulability analysis
Weaves	*Transport services*; explicit	interface points are the encapsulating socket *pads*	extensible type system; types are connector *sockets*	via naming conventions	via interface	none	allows association of arbitrary, uninterpreted annotations with transport services
Wright	*Connector*; explicit	interface points are *roles*; role interaction semantics specified in CSP	extensible type system, based on protocols; parameterizable number of roles and glue	connector *glue* semantics in CSP	via interfaces and semantics; protocols of interaction for each role in CSP; stylistic invariants	via different parameter instantiations	none

Cuadro 3.7: Características del modelado de conectores de distintos ADLs (extraído de [144])

sección 4.1). Ha sido específicamente diseñado para modelar, analizar y generar código para sistemas empotrados de tiempo-real [86].

Como se ha mostrado, existe una gran variedad de ADLs centrados en el modelado de uno o varios aspectos de un sistema. El problema aparece cuando se quiere utilizar el mismo lenguaje para modelar otros aspectos, ya que generalmente no han sido diseñados para permitirlo. Este problema acaba con el desarrollo de un nuevo ADL, su notación y las transformaciones necesarias para utilizar el modelo anterior. Para evitar este problema, el **ISR** (Universidad de California) está desarrollando unas herramientas que permiten generar automáticamente nuevos ADLs [71]. Este enfoque se basa en la encapsulación de los conceptos básicos de todo ADL en módulos *xADL v2.0*[5]. Haciendo uso de las herramientas reflexivas que proporciona el ISR, es posible transformar de forma automática descripciones arquitectónicas entre módulos *xADL*, y por ende entre distintos ADLs.

[5] `http://www.isr.uci.edu/projects/xarchuci/`

3.2.5 LENGUAJES «ORIENTADOS A COMPONENTES»

El desarrollo de los lenguajes de descripción de arquitectura (ADLs) durante la década de los noventa dió paso a la creación de los primeros lenguajes completamente orientados a componentes. El objetivo perseguido en la elaboración de estos lenguajes era que los conceptos básicos de CBD aparecieran, de forma explícita, en el lenguaje y se generara código que directamente representara los conceptos del lenguaje CBD. Es decir, estos lenguajes orientados a componentes pretendían ser el siguiente paso en la evolución de la Ingeniería del Software y aprovechar todo el conocimiento desarrollado anteriormente. Sin embargo, por una causa u otra, todavía no han tenido una amplia penetración en el mundo del desarrollo software. En este apartado se van a describir someramente dos lenguajes de este tipo: *Lagoona* y *Component Pascal*.

Una de las principales causas del poco éxito de los lenguajes orientados a componentes es que, generalmente, se utilizan únicamente para proporcionar una sintaxis abstracta, que posteriormente es traducida a un lenguaje de programación «tradicional» (orientado a objetos o no). Este hecho es analizado por Fröhlich [93] quien llega a la conclusión de que los lenguajes orientados a objetos no cumplen los requisitos necesarios para poder ser utilizados como plataforma de implementación de componentes, sino que es necesario desarrollar un nuevo lenguaje que tenga en cuenta las características particulares del desarrollo basado en componentes. En este sentido propone el lenguaje Lagoona.

El lenguaje de programación orientado a componentes Lagoona [95] fue desarrollado como un lenguaje académico experimental. Lagoona contempla todos los conceptos de CBD, haciendo especial hincapié en la comunicación entre componentes, ya que promueve la utilización de *mensajes* como elementos de primera clase [94] del lenguaje. Lagoona diferencia definición de implementación, de forma que los mensajes definen qué hace un componente mientras que los métodos describen cómo lo hace; por tanto es posible definir un componente cuyas instancias implementan las mismas interfaces, por lo que son componentes compatibles y sustituibles, utilizando distintos algoritmos. De igual forma define interfaces, como conjunto de mensajes, y tipos de implementación, como conjunto de métodos. Lagoona define también reglas de sustitución de componentes basadas en la compatibilidad entre interfaces. Sin embargo, parece que Lagoona no ha pasado de ser un lenguaje experimental para la comunidad académica.

Por otro lado, Component Pascal [201] es un lenguaje, también orientado a componentes, que ha tenido más aceptación. Component Pascal ha sido el último lenguaje desarrollado por Niklaus Wirth y sigue la línea de diseño de otros lenguajes anteriormente diseñados por Wirth, como Oberon and Oberon–2. Con una sintaxis reducida y similar a la sintaxis clásica de los lenguajes inspirados en Pascal, Component Pascal contempla

los conceptos básicos en que se basa CBD. Actualmente existen dos compiladores[6] para Component Pascal con versiones recientes, por lo que parece que este lenguaje no ha caído en el olvido. Sin embargo, hasta que los lenguajes realmente orientados a componentes sean una realidad y alcancen el reconocimiento y el grado de uso de otros paradigmas, la utilización del paradigma de orientación a objetos parece una forma natural de modelar e implementar componentes [116].

3.2.6 OTRAS HERRAMIENTAS PARA DISEÑAR ARQUITECTURAS

El SEI está trabajando actualmente en un método que ayude al desarrollador a diseñar la arquitectura de su aplicación de tal forma que se pueda asegurar que el producto final tendrá las cualidades deseadas. Este método se denomina *Attribute-Driven Design* (**ADD**) [219]. ADD es un método de descomposición recursivo, que parte de una clara especificación de los requisitos funcionales y de calidad del sistema final. Cada una de estas etapas de descomposición es validada contra dichos requisitos, por lo que su recogida es una actividad imprescindible para ADD. ADD está siendo aplicado a diversos dominios, desde sistemas de información a sistemas empotrados. Y según el SEI, los primeros productos cuyo software ha sido desarrollado con ADD están en fase de producción.

La herramienta *Architecture Expert* (**ArchE**) [18] permite el diseño de una arquitectura que satisfaga una serie de requisitos de calidad, utilizando un núcleo que se apoya en un sistema experto basado en reglas diseñado por el propio SEI [24, 138]. Este núcleo contiene teorías para los atributos de calidad, técnicas para establecer el modelo que mejor se ajusta a estos atributos de calidad a partir de la descripción arquitectónica, técnicas para integrar dichos modelos en una única solución y para determinar las respuestas previstas ante determinadas situaciones. El núcleo ofrece también la posibilidad de utilizar diseños heredados (*legacy*) como entradas al proceso [19].

El objetivo final del SEI es, partiendo de los resultados de estudios anteriores sobre atributos de calidad y arquitectura [88, 130, 139], desarrollar un conjunto de reglas específicas para ayudar a desarrollar la arquitectura que cumpla con un único atributo de calidad. Estas reglas serán posteriormente ejecutadas de forma paralela por ArchE para obtener, con ayuda del usuario, la arquitectura final del sistema que cumple con los atributos de calidad que se han usado como entradas al proceso. Finalmente, la intención del SEI es fusionar en un futuro ArchE con ADD.

[6] Gardens Point Componente Pascal (`http://plas2003.fit.qut.edu.au/gpcp/`), desarrollado en la Universidad de Queensland y BlackBox (`http://www.oberon.ch/blackbox.html`) [218], desarrollado por Oberon Systems.

ArchJava[7] [7] es una extensión de Java que unifica la arquitectura con la implementación, usando un sistema de tipos para asegurar que la implementación es conforme con determinadas restricciones arquitectónicas. De esta forma se pretende evitar la evolución por separado de código y arquitectura, con las consecuencias de inconsistencia, confusión, violación de propiedades y dificultad en la evolución que conlleva [123]. ArchJava extiende el lenguaje Java con las palabras clave típicas de un ADL: *component* (describe partes de la arquitectura), *connection* (establece comunicación entre componentes) y *port* (punto de comunicación del componente con su entorno). Actualmente se está aplicando ArchJava para capturar la arquitectura del *Mission Data System* que está desarrollando el *Jet Propulsion Lab* (**JPL**) para sus aplicaciones espaciales.

ArchEvol[8] [163] está siendo diseñado por el *Institute for Software Research* (Universidad de California). Al igual que otros, ArchEvol pretende no sólo mantener de forma uniforme y consistente la vista arquitectónica y la de código, sino establecer también un sistema de control de versiones de ambas vistas. ArchEvol se distribuye como un *plug-in* para el entorno de desarrollo *Eclipse* y requiere además el sistema de mantenimiento de versiones *Subversion*[9]. Este *plug-in* proporciona un entorno de diseño con los clásicos elementos arquitectónicos: componente, puerto y conector. Una vez que se ha realizado el diseño a nivel arquitectónico, ArchEvol transforma cada componente en un proyecto Java, de forma que el usuario puede utilizar *Eclipse* para implementar el código del mismo, y usa los conectores para establecer las relaciones de dependencia entre los distintos proyectos, i.e. componentes de la arquitectura. Tanto la descripción arquitectónica como cada uno de los proyectos Java (componentes) son posteriormente incluidos en el repositorio de control de versiones para continuar con el ciclo de desarrollo.

3.3 Evaluación y Evolución Arquitectónica

LAS DECISIONES arquitectónicas que se toman al principio del desarrollo de un sistema afectan en gran medida al éxito o fracaso del mismo, y por tanto la detección precoz de errores se está convirtiendo en una área fundamental. El coste de corregir un error aumenta conforme pasa el tiempo y el ciclo de vida de un producto o una familia de productos. Por ello es necesario disponer de métodos fiables para evaluar la arquitectura y aplicarlos tan pronto como sea posible. Es tan importante ser capaz de desarrollar una arquitectura que se ajusta a las necesidades del sistema como ser capaz de evaluar si realmente lo consigue.

[7] `http://archjava.fluid.cs.cmu.edu/`
[8] `http://webfiles.uci.edu/enistor/archevol/`
[9] `http://subversion.tigris.org/`

Entre las metodologías existentes para evaluar la arquitectura de un sistema destacan las desarrolladas por el SEI [61, 126]): *Software Architecture Analysis Method* (**SAAM**), *Architecture Tradeoff Analysis Method* (**ATAM**), el novedoso *Analytic Principles and Tools for The Improvement of Architectures* (**APTIA**) y un largo etcétera. También destaca el trabajo de Boehm [32], con el desarrollo de *WinWin* y *Quality Attribute Risk and Conflict Consultant* (**QARCC**). Así como existen numerosos métodos para evaluar si las características de una arquitectura se ciñen a los requisitos especificados para el sistema, existen numerosos estudios comparan estos métodos. Dobrica [74] realiza, quizá, uno de los análisis más completos, en el que comparan ocho de los métodos disponibles en el año 2.002. En un trabajo más reciente, Kazman propone nuevos criterios para analizar y comparar los métodos de evaluación de arquitecturas [127].

Sólo en la más rara de las ocasiones un sistema se mantiene durante todo su ciclo de vida sin sufrir modificaciones. Lo normal es que los sistemas tengan que evolucionar. Aparecen nuevas versiones que solucionan algún fallo que no se había detectado en la versión anterior o que aumentan la funcionalidad del sistema. Por tanto, se tienen que desarrollar nuevas técnicas que soporten un estudio de la evolución de un sistema, de forma que sea posible evaluar si una arquitectura es capaz de cumplir los nuevos requisitos o dónde tiene que ser modificada para lograrlo.

3.4 FRAMEWORKS ARQUITECTÓNICOS

LOS FRAMEWORKS o marcos de trabajo fueron uno de las primeras soluciones para reutilizar software que tuvieron realmente éxito. El interés por el desarrollo de frameworks y por el estudio de sus características y métodos de desarrollo comenzó en el mundo de la orientación a objetos con el que, posiblemente, ha sido el más famoso y fructífero de todos: el framework *Model-View-Controller* [133] de Smalltalk. Este tipo de frameworks se denominan genéricamente *frameworks de objetos*, en referencia a que son soluciones basadas en los principales mecanismos de dicho paradigma, i.e. herencia y composición.

El desarrollo software basado en componentes (Component-Based Development (**CBD**)) también ha adoptado el término framework pero con un significado ligeramente distinto. Como ya se mencionó en el apartado 2.4, los componentes son desplegados en un entorno que ofrece una serie de servicios para facilitar la integración y comunicación entre los componentes y con el entorno, servicios que son proporcionados por un framework, pero que en este caso, y para diferenciarlo de los frameworks de objetos, se denomina *framework de componentes* [17].

Los frameworks de objetos se pueden ver como aplicaciones incompletas o medio desarrolladas, que ofrecen una serie de puntos de extesión para que el usuario complete el framework con el objetivo de obtener la aplicación final. Sin embargo, los frameworks de componentes son más bien el núcleo que proporciona el soporte a un desarrollo basado en componentes, facilitando una serie de servicios auxiliares. En este sentido, todo framework puede verse como una aplicación incompleta, ya que en sí mismo sólo proporciona el soporte necesario para que el usuario acabe de añadir el código que completa la aplicación; pero como programa, el framework de componentes es una apliación completa que no tiene mecanismos de extensión sino más bien de utilización de sus servicios.

3.4.1 FRAMEWORKS DE OBJETOS

Los frameworks de objetos fueron los primeros que aparecieron en la escena del desarrollo software y siguen siendo uno de los métodos más efectivos y usados para reutilizar software. Comparten gran cantidad de características con las técnicas propias de la orientación a objetos, en las que están basadas. Sin embargo estos frameworks están desarrollados de forma que se potencia el ámbito de reutilización, pasando de la reutilización a pequeña escala (clases) a otra a gran escala, en la que se reutiliza la estructura (arquitectura) del sistema. A continuación se presentan algunas de las diversas definiciones de «framework» que existen en la bibliografía:

- **Gamma (1.995) [96]:** un framework es un conjunto de clases que cooperan y forman un diseño reutilizable para un tipo específico de software. Un framework ofrece una guía arquitectónica partiendo el diseño en clases abstractas y definiendo sus responsabilidades y sus colaboraciones. Un desarrollador personaliza el framework para una aplicación particular mediante herencia y composición de instancias de las clases del framework.

- **Fayad (1.999) [85]:** un framework es un diseño reutilizable de un sistema que describe cómo se descompone en un conjunto de objetos que interaccionan. Algunas veces el sistema es una aplicación completa, mientras que otras es simplemente un subsistema. El framework describe tanto los objetos que lo componen como sus interacciones. Describe cómo se reparten las responsabilidades del sistema entre los objetos, la interfaz de cada objeto y el flujo de control entre ellos.

- **Szyperski (2.002) [201]:** un framework es un conjunto de clases cooperantes, algunas de las cuales pueden ser abstractas, que forman un diseño reusable para un tipo específico de software.

Como se deduce de las anteriores definiciones, un framework de objetos proporciona un diseño abstracto para un dominio concreto. El diseño abstracto contiene las clases que integran el framework y las interacciones que ocurren entre los objetos de las clases que lo componen, de forma que queda plasmada una solución genérica al conjunto de problemas presentes en el dominio en que se aplica. Las aplicaciones se construyen extendiendo el framework con la funcionalidad propia de la aplicación que se quiere desarrollar. Un framework consta de una arquitectura (plasmada por el diseño abstracto) y, posiblemente, un conjunto de clases concretas que ofrecen parte de la infraestructura del framework. El funcionamiento de un framework responde al llamado principio de *Hollywood*: «*don't call us, we'll call you*», que refleja la inversión de control respecto al desarrollo de aplicaciones basadas en librerías [35].

Un framework asiste a los desarrolladores en la creación de soluciones a problemas dentro de un dominio y les facilita su mantenimiento. Ofrece una estructura que, si está bien diseñada, permite la sustitución de unos elementos por otros con un impacto mínimo sobre el resto del framework. El diseño de un framework difiere en gran medida del diseño de una aplicación convencional, ya que ofrecen soluciones genéricas e incompletas a los problemas de un dominio. Los frameworks ofrecen una serie de puntos de extensión en los que el usuario añade la funcionalidad propia de la aplicación, por lo que tampoco contemplan toda la funcionalidad del dominio. Más bien contienen la funcionalidad mínima común a todas las aplicaciones y proporcionan los mecanismos necesarios para que el usuario añada la que necesite. Existen tres tipos de frameworks, dependiendo de qué mecanismo tenga que utilizar el usuario del mismo [85]:

Caja blanca: son frameworks que utilizan principalmente el mecanismo de herencia para su extensión. El usuario añade la funcionalidad propia de su aplicación extendiendo determinadas clases del framework, denominadas puntos de extensión. Este tipo de frameworks es el más difícil de utilizar puesto que requiere un profundo conocimiento de la clase que se va a extender y de sus métodos, ya que por herencia se pueden sobrescribir por error métodos de la clase padre. El nombre refleja la característica de visibilidad completa de la implementación del framework que tiene el usuario del mismo (por lo menos en determinadas clases).

Caja negra: son frameworks en los que el usuario utiliza el mecanismo de composición y delegación para desarrollar la aplicación. Enfatizan las relaciones dinámicas entre los objetos en vez de las relaciones estáticas entre clases. El usuario no tiene por qué conocer los detalles del mismo, sino sólo cómo utilizar y combinar los objetos existentes o cómo crear unos nuevos. Los frameworks de caja negra son más fáciles y seguros de utilizar, pero más difíciles de diseñar, ya que requieren que el diseñador anticipe las interfaces y los puntos de extensión para soportar un conjunto amplio de

posibilidades de ensamblaje. Además, un uso excesivo de la composición de objetos puede producir diseños difíciles de comprender.

Caja gris: a menudo los frameworks no son exclusivamente de un tipo u otros, sino que contemplan las dos posibilidades de expansión en distinto grado, en un intento por aportar las ventajas de ambos enfoques y contrarrestar, en la medida de lo posible, sus inconvenientes.

En el mercado existen diversos tipos de frameworks, desde framewoks de bajo nivel que ofrecen servicios básicos (comunicaciones, sistemas de ficheros, soporte de impresión, etc) hasta frameworks de alto nivel (interfaces de usuario, servicios multimedia, etc). A pesar de que se aplican en dominios diferentes, todos ellos tienen un conjunto de conceptos en común. De acuerdo a estos conceptos comunes, Fayd [85] propone la siguiente clasificación:

Frameworks de infraestructura del sistema: diseñados para simplificar el desarrollo de una infraestructura de sistema portable y eficiente. Incluye el sistema operativo, el sistema de comunicaciones y el de interacción con el usuario. Generalmente son utilizados internamente por las empresas y no se venden directamente.

Frameworks de soporte: también llamados *middleware*. Su función es integrar aplicaciones y componentes distribuidos y diseñados en distintos lenguajes. De esta forma potencian la división del software en módulos que puedan ser distribuidos en distintos nodos, reutilizados y fácilmente extendidos.

Frameworks de dominio o aplicación: son utilizados y desarrollados para generar aplicaciones y productos finales.

Prueba que los frameworks han tenido éxito en cumplir con el objetivo de reutilización es la cantidad de frameworks existentes y la variedad de dominios en los que se han desarrollado. Salvo contadas excepciones, la mayoría de los frameworks existentes utilizan el patrón *Model-View-Controller* en entornos tales como la creación de sitios web (Apache *Cocoon* y *Struts*, Microsoft *Atlas* y un largo etcétera) o la generación de interfaces gráficas (el API *Swing* de Java, las librerías *Qt* y *GTK+* de Linux o los elementos gráficos multiplataforma *wxWidgets*). Entre las excepciones cabe destacar *Cactus* (un framework para la computación distribuida de altas prestaciones), *Microsoft Foundation Classes* (agrupa parte de la funcionalidad del API de Windows para desarrolladores) o GStreamer (un entorno multimedia para Linux).

3.4.2 FRAMEWORKS DE COMPONENTES

En un entorno de desarrollo basado en componentes, un framework de componentes es una implementación de los servicios que debe proporcionar el modelo de componentes, que fuerza al desarrollador a cumplir los estándares y las convenciones impuestas por este. Bachmann [17] establece que un framework de componentes es como un sistema operativo de propósito especial, aunque a un nivel de abstracción mucho mayor, ya que proporciona los servicios básicos a los componentes que integran la aplicación. Szyperski [201] destaca la intima relación que existe entre framework de componentes y la arquitectura de un sistema, y define un framework como una «arquitectura dedicada y un conjunto de principios de composición al nivel de componente».

Aunque el framework de componentes proporciona los servicios básicos de apoyo al modelo de componentes, no es necesario que el framework exista como una entidad independiente en tiempo de ejecución, sino que puede estar integrado en los componentes. Sin embargo es habitual que tenga una existencia independiente en tiempo de ejecución [85], principalmente porque uno de los servicios que suelen ofrecer es la carga dinámica de componentes en tiempo de ejecución. Bachmann [17] señala que un framework de componentes puede verse también como un componente, de grano más grueso al habitual, y es, por tanto, susceptible de ser compuesto con otros componentes o frameworks. Por tanto, describe tres posibles formas de composición:

Componente-Componente: composición que permite la interacción entre los componentes. Es, por tanto, la que realiza la funcionalidad de la aplicación. Los contratos que especifican la comunicación a este nivel se denominan *contratos de aplicación*.

Framework-Componente: composición que permite la interacción entre el componente y los servicios del framework. Los contratos de este nivel se denominan *contratos de sistema*.

Framework-Framework: permite la interacción entre frameworks. Los contratos de este nivel se denominan *contratos de interoperabilidad*, ya que permiten la comunicación entre componentes desplegados en distintos frameworks, que pueden estar también distribuidos en distintos nodos.

Aunque los frameworks fueron ideados antes de la aparición de las líneas de productos [62], ha sido en su contexto en el que están encontrando un mayor éxito. No hay que olvidar que los frameworks han sido el primer modelo de reutilización que adoptó un enfoque que contempla la variabilidad presente en las aplicaciones de un dominio, enfoque que ha sido plasmado con mayor detalle y profundidad en las líneas de productos.

3.5 Conclusiones y Aportaciones a la Tesis

Tras estudiar en el capítulo anterior las características fundamentales de las unidades básicas de reutilización software, este capítulo ha descrito algunas de las tecnologías desarrolladas para organizar y describir la estructura de alto nivel del software de la aplicación. Además de resumir brevemente algunas de las herramientas y tecnologías que ha desarrollado la disciplina de la arquitectura software, como ADLs, lenguajes de componentes, patrones arquitectónicos, vistas, etc., se han descrito también las principales características y tipos de frameworks. Los frameworks fueron el primer tipo de aplicaciones que se crearon expresamente para permitir la reutilización del software. El framework contiene un núcleo reutilizable y compartido por todas las aplicaciones que se generen a partir de él, y una serie de puntos de extensión en los que el usuario añade su funcionalidad, creando finalmente la aplicación deseada.

A parte de la lección y la aportación en sí misma que supone planificar el diseño software con el fin de alcanzar los requisitos impuestos a la aplicación, las aportaciones de esta disciplina al desarrollo de esta Tesis Doctoral son también muy importantes. En primer lugar, el estudio de las características de los ADLs y del soporte que proporcionan para modelar explícitamente los conceptos básicos de CBD ha sido vital para el diseño del meta-modelo V^3Studio (ver capítulo 6) y para seleccionar las decisiones de diseño que posteriormente guiaron la generación de un modelo UML (consultar capítulo 7). Esta transformación a UML ha utilizado, como se describe en dicho capítulo, diversos patrones arquitectónicos de los expuestos en este capítulo para superar las limitaciones de los lenguajes orientados a objetos que hacen imposible una traducción directa de los conceptos básicos CBD y proponer una implementación viable de dichos conceptos bajo una tecnología de orientación a objetos.

Por otro lado, tras el estudio de los distintos tipos de frameworks y sus características se ha hecho patente la necesidad de desarrollar una infraestructura mínima de soporte para los componentes que forman la aplicación. Pero, en el caso de esta Tesis Doctoral, el framework se va a generar a la vez que se crean los componentes, durante la transformación de V^3Studio a UML (consultar capítulo 7). De esta forma se consigue que el framework también sea independiente de la plataforma de ejecución final a la par que se consigue una mejor integración entre todos los elementos de la aplicación.

CAPÍTULO 4

DESARROLLO BASADO EN MODELOS

ESTE capítulo presenta el novedoso enfoque de desarrollo software basado en el uso de modelos a lo largo de las distintas etapas del proceso de diseño software. Model-Driven Engineering (**MDE**) promete revolucionar la forma en que se desarrolla el software, proporcionando una teoría de desarrollo y una serie de herramientas de soporte para aplicarla. MDE busca no solo aumentar el nivel de abstracción con que se realiza el diseño del software, sino también obtener de forma (semi-) automática distintas representaciones del mismo, hasta llegar finalmente a código ejecutable. En este sentido, MDE proporciona las herramientas necesarias para que el desarrollador pueda generar una herramienta adaptada a sus necesidades particulares.

El capítulo está organizado en cinco secciones. La primera realiza una introducción más detallada al enfoque, presentando sus características principales y plasmando la evolución que supone este enfoque frente a los métodos de desarrollo existentes hasta el momento. Las dos secciones siguientes presentan las bases del enfoque de desarrollo por modelos MDE, que son principalmente los conceptos de modelo, meta-modelo y transformación de modelos. La sección posterior describe los estándares que ha desarrollado el OMG para llevar a cabo su visión particular de MDE, denominada MDA, y realiza una breve comparativa entre ambas. Finalmente, la última sección describe algunas tecnologías que están estrechamente relacionadas con MDE, como son los lenguajes específicos de dominio (Domain-Specific Language (**DSL**)), que ya forman parte también del modelado de sistemas, y la visión particular de MDE que ha desarrollado Microsoft: las factorías software (Software Factories (**SF**)).

4.1 INTRODUCCIÓN A MDE

SEGÚN el diccionario de la *Real Academia de la Lengua Española*, un modelo es un «esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento». Hasta hace relativamente poco tiempo la ingeniería del software no se había fijado en esta forma de desarrollo, que durante tanto tiempo y con tan buenos resultados lleva aplicándose a otras ramas de la ciencia, como la física o la química. El uso de modelos para desarrollar software se denomina genéricamente Model-Driven Engineering (**MDE**) [51, 128, 194], y su estudio y aplicación en la Ingeniería del Software comenzó a principios del siglo XXI.

> *Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively.*
>
> — Douglas C. Schmidt [191]

El término «desarrollo o ingeniería basada en modelos» hace referencia a la técnica que hace uso, de forma sistemática y reiterada, de modelos como elementos primordiales a largo de todo el proceso de desarrollo [128], que en este caso particular es software. Es decir, MDE trabaja con modelos como entradas al proceso y produce modelos como salidas del mismo. La utilización de modelos, representaciones simplificadas de la realidad, permite aumentar el nivel de abstracción con que se realiza el diseño así como la reutilización de los mismos, gracias a que los modelos son considerados entidades de primera clase en el enfoque MDE. De esta forma, los artefactos que se generan son independientes del lenguaje y el paradigma de programación que se vaya a utilizar posteriormente para traducirlos en código. Además, se facilita mucho la comunicación de ideas, ya que están expresadas de forma explícita y no diluidas en el código del programa.

La evolución de la *Ingeniería del Software* es clara: se pasó de la programación estructurada (en la que lo importante eran las funciones) a la orientación a objetos (en la que lo importante son los objetos, como unión $datos + funciones$) y actualmente al desarrollo basado en modelos (en el que lo importante son los conceptos, presentes en los modelos). Se ha aumentado considerablemente el nivel de abstracción, centrándose en los conceptos y relegando los detalles de la implementación a un segundo plano. Aquí es donde reside el gran potencial de esta nueva forma de desarrollo: se diseñan y manipulan los conceptos importantes para la aplicación que se pretende desarrollar, en vez de líneas de código en un lenguaje de programación. MDE supone un paso revolucionario, esta vez sí, para

el desarrollo software, comparable solo al salto cualitativo que supuso el diseño de los primeros compiladores de FORTRAN en 1.957 [148]. Bézivin [49] realiza una analogía entre lo que supuso el cambio de la programación estructurada a la orientación a objetos con el cambio al nuevo enfoque MDE: «*de todo es un objeto a todo es un modelo*».

> *Model-driven development holds promise of being the first true generational leap in software development since the introduction of the compiler. The key lies in resolving pragmatic issues related to the artifacts and culture of previous generations of software technologies.*
>
> — Bran Selic [194]

Un aspecto muy importante de MDE y que lo distingue de otro estándar como MDA (ver sección 4.4) es que MDE es un enfoque abierto e integrador que acoge varios *espacios tecnológicos* (Technological Spaces (**TS**) [136]) de forma uniforme [83]. Un TS es un marco de trabajo con un conjunto de conceptos asociados, un cuerpo de conocimiento, herramientas y habilidades requeridas en este marco. Ejemplos de TS son XML (TS de intercambio de información) y MDA (TS de meta-modelado). Puesto que no existe «el mejor espacio tecnológico para resolver todos los problemas», parte del éxito del enfoque MDE depende tanto de que la comunidad sea capaz de integrar y adaptar el conocimiento que poseen las disciplinas que integran espacios tecnológicos más maduros [83], como de que no olvide el principio que rige su desarrollo («todo es un modelo») [49].

MDE utiliza modelos como elementos básicos del desarrollo software. Pero un programa es una entidad dinámica, que cambia, evoluciona en el transcurso del tiempo, y por tanto se necesita una forma de permitir que los modelos también se transformen y evolucionen. Un programa es una entidad compleja, que no puede ser abarcada por un único modelo sino que requiere el uso combinado de varios modelos que muestren distintos niveles de abstracción y puntos de vista del mismo. Incluso que permitan comprobar distintas propiedades y probar distintas suposiciones. Tampoco hay que olvidar que el objetivo último del enfoque es la generación de código, de un ejecutable del problema que se modela, y que los modelos permiten describir el problema desde un nivel de abstracción adecuado para el trabajo del diseñador.

MDE persigue que la conexión entre los modelos y el producto final sea automática, de forma que el modelo no se quede en mera documentación (como sucede actualmente con gran cantidad de diagramas UML). Estas consideraciones, junto con el principio «todo es un modelo», llevan a Bézivin [49] a afirmar que un programa es un modelo (haciendo explícita la relación entre gramática y meta-modelo), que la plataforma de ejecución es un modelo (esto permite generar distintas implementaciones del mismo modelo para distintas plataformas), que los sistemas heredados son modelos, que las transformaciones entre

modelos son también modelos (con su propio meta-modelo), que los meta-modelos son modelos en esencia, que los componentes son modelos, que las trazas son modelos (tanto la traza de ejecución de un programa como la de ejecución de una transformación), etc. Como solución a estos problemas de traducción automática surgen las *transformaciones de modelos*, como principal herramienta de soporte que completa el enfoque MDE.

Para que el enfoque de desarrollo software MDE sea aplicable en la realidad es imprescindible que existan una serie de herramientas que proporcionen el soporte necesario al diseñador, de forma que este se concentre únicamente en el sistema que se va a modelar y no en los pequeños detalles del enfoque utilizado. Por tanto, gran parte del éxito de desarrollo de MDE se debe al esfuerzo realizado por el Object Management Group (**OMG**) por desarrollar una serie de estándares para su enfoque Model-Driven Architecture (**MDA**) [165] (ver sección 4.4).

Existen otros aspectos de MDE que van a ser únicamente mencionados pero que tienen también una importancia capital de cara a que el enfoque alcance con éxito las metas que se ha marcado. La mayoría de ellos son actualmente objeto de una profunda actividad investigadora, ya que son problemas que afectan directamente a la utilización de MDE. Entre estos aspectos cabe destacar el desarrollo de modelos que puedan ser directamente ejecutables sin tener que pasar por una transformación intermedia a código (UML ejecutable) [147, 148]; el problema de la verificación y validación de las transformaciones de modelos, i.e. cómo comprobar que el modelo generado sigue siendo compatible, desde el punto de vista semántico, con el modelo origen [50]; la generación de escenarios de pruebas (*test suites*) para comprobar que los modelos se comportan de forma correcta y se ajustan a la especificación [25, 26]; la predicción de algunas características del comportamiento en ejecución de los modelos [20], etc.

4.2 REPRESENTACIÓN DE LA REALIDAD EN MDE

BÉZIVIN [49] afirma que, así como el principio de la orientación a objetos («todo es un objeto») se basa en las relaciones de *instanciación* y *herencia*, las relaciones de *representación* (un meta-modelo *representa* un sistema) y de *conformidad* (un modelo es *conforme* a su meta-modelo) son las relaciones básicas sobre las que descansa el principio de MDE «todo es un modelo». Favre [83] defiende que, a pesar de que MDE depende del desarrollo y manipulación de modelos de forma precisa, los conceptos básicos que definen MDE y el proceso no están definidos con el mismo rigor. Defiende, por tanto, la necesidad de definir un modelo del propio proceso MDE, en que se especifiquen todos sus conceptos clave y las relaciones entre ellos. Favre denomina «*megamodelo*» [84] a este modelo particular

de MDE, y aumenta las relaciones básicas del mismo a cinco: *descompuesto en (δ)*, *representado por (μ)*, *elemento de (ε)*, *conforme a (χ)* y *transformado en (τ)* (ver figura 4.1).

La figura 4.1 plasma los distintos roles que puede desempeñar un *sistema* en base a las relaciones básicas definidas anteriormente. En esta figura puede observarse, por ejemplo, que un sistema puede verse (1) como una representación de otro sistema, (2) como una de las partes que componen un sistema más grande, (3) como el resultado de una transformación de modelos, (4) como una transformación en sí misma, etc. Este *megamodelo* de MDE permite describir distintas vistas estáticas de un mismo sistema mediante el uso de modelos y permite también describir la forma en que estos modelos evolucionan dinámicamente, por medio de transformaciones (e incluso las mismas transformaciones). Pero independientemente de lo que se describa, todos los artefactos utilizados a lo largo de esto proceso son considerados siempre modelos.

El resto de esta sección presenta los conceptos básicos en que se basa el enfoque MDE y que permiten describir lo que se ha nombrado previamente como «vista estática»: modelo y meta-modelo. En cada uno de los siguientes apartados se enumeran algunas de las definiciones que aparecen en la bibliografía de la materia y se describen las principales características de cada uno de ellos. El último apartado realiza una pequeña crítica a la utilización exclusiva de una visión lingüística para crear los tres niveles de la pirámide MDE y desechar la visión ontológica de la realidad, lo cual reduce la capacidad de modelado de MDE.

4.2.1 El concepto «modelo» en MDE

Un modelo, como representación simplificada de una realidad que se quiere plasmar o manipular, es uno de los conceptos fundamentales en que se basa el enfoque MDE. Entre las definiciones de «modelo» presentes en la bibliografía se destacan las siguientes:

- **Seidewitz (2.003) [193].** Un modelo es un conjunto de declaraciones sobre un sistema bajo estudio.
- **Kleppe (2.004) [131].** Un modelo es una descripción de (una parte de) un sistema y está escrito en un lenguaje bien definido. Un lenguaje bien definido tiene una forma (sintaxis) y un significado (semántica) correctamente especificados, de forma que puede ser interpretado automáticamente por una máquina.
- **UML v2.1.1 (2.007) [176].** Un modelo es una simplificación de un sistema, que ha sido construida con un objetivo en mente. Un modelo debe comportarse igual que se comportaría el sistema que modela.

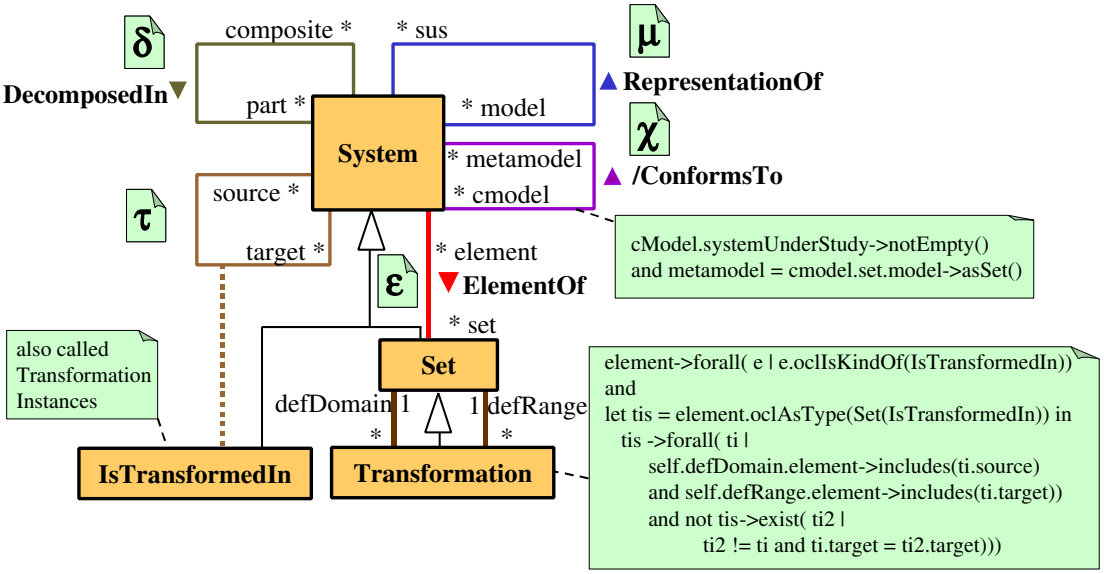


Figura 4.1: Megamodelo de **MDE** (extraído de [84])

El hombre ha desarrollado modelos desde la antigüedad para mejorar su entendimiento de un problema y sus posibles soluciones, antes de realizar el esfuerzo de implementar la solución completamente. Sin embargo, Selic [194] afirma que para que un modelo sea realmente útil y efectivo tiene que cumplir con las siguientes características:

1. **Abstracción.** Un modelo ofrece siempre una visión reducida y simplificada de la realidad que ayuda a comprender mejor el problema gracias a que elimina los detalles superfluos o poco importantes. Pero tampoco se puede reducir excesivamente, porque se obtiene un modelo que ya no representa la realidad. Para poder modelar completamente un sistema se necesitan distintos modelos, que muestren distintos puntos de vista del mismo. Según dijo Einstein «*make everything as simple as possible, but not simpler*».

2. **Precisión.** Un modelo debe describir con precisión y sin ambigüedad la realidad. Debería ser posible utilizar un modelo para predecir correctamente algunas propiedades no obvias ni modeladas directamente por él, mediante experimentación (ejecución del modelo) o análisis formales (en caso de que sea posible).

3. **Intelegibilidad.** No es suficiente con que un modelo sea capaz de expresar correctamente la realidad, tiene que ser inteligible por los usuarios. Puesto que además un modelo es una representación de la realidad, un modelo difícil de comprender puede denotar una falta de precisión o abstracción en el mismo.

4.2.2 El concepto «meta-modelo » en MDE

El concepto de «meta-modelo» es otro pilar sobre el que descansa el enfoque MDE. El prefijo «meta-» proviene del griego $\mu\epsilon\tau\alpha$, que significa «después de», «más allá». Este prefijo se utiliza para indicar que un concepto es una abstracción de otro y que se usa para completar o añadir información a este. Alguna de las principales definiciones de «metamodelo» son las siguientes:

- **Seidewitz (2.003) [193].** Un metamodelo es un modelo de especificación para una clase de sistemas que cumplen que cada uno de ellos es en sí mismo un modelo válido en un cierto lenguaje de modelado.
- **Mellor (2.004) [148].** Un metamodelo es un modelo que define la estructura, semántica y restricciones de una familia de modelos.
- **MOF v2.0 (2.004) [167].** Un metamodelo es un modelo que define el lenguaje para definir un modelo.

Como puede deducirse de estas definiciones, un meta-modelo es una representación que describe un dominio de sistemas. En un meta-modelo están presentes todos los conceptos del dominio que son importantes para el usuario y las relaciones entre ellos. Un meta-modelo, por tanto, restringe completamente los elementos que pueden formar parte de un modelo de un sistema en particular y sus relaciones. De esta forma, un modelo siempre es conforme a su meta-modelo (por definición). Sin embargo, el concepto de «meta-modelo» no es un concepto absoluto, sino que depende del nivel de abstracción utilizado. Por tanto, lo que con un determinado nivel es un meta-modelo se puede convertir en modelo en otro.

Además, un meta-modelo de un dominio no tiene por qué ser único, sino que pueden co-existir diversos meta-modelos que permiten describir el dominio desde distintos puntos de vista y con distintos niveles de detalle. La figura 4.2 muestra varios ejemplos de esta relación entre meta-modelo y modelo. Por ejemplo, la figura b) muestra que el diccionario es un meta-modelo del dominio formado por un idioma, ya que el diccionario contiene los conceptos del dominio, pero que a su vez el mismo diccionario es un modelo del idioma, ya que está escrito en dicho idioma, haciendo usos de sus conceptos.

Siguiendo el razonamiento anterior de que un modelo es siempre conforme a su meta-modelo, el meta-modelo también debe tener su propio meta-meta-modelo (que contiene los elementos básicos para describir meta-modelos), y al que por supuesto tiene que ser conforme; y así sucesivamente. La única forma para que el enfoque MDE pueda ser aplicado en la práctica pasa por romper esta relación recursiva en la definición de meta-modelos. Esta es, precisamente, la característica principal de un *modelo reflexivo* [193]: tanto el lenguaje

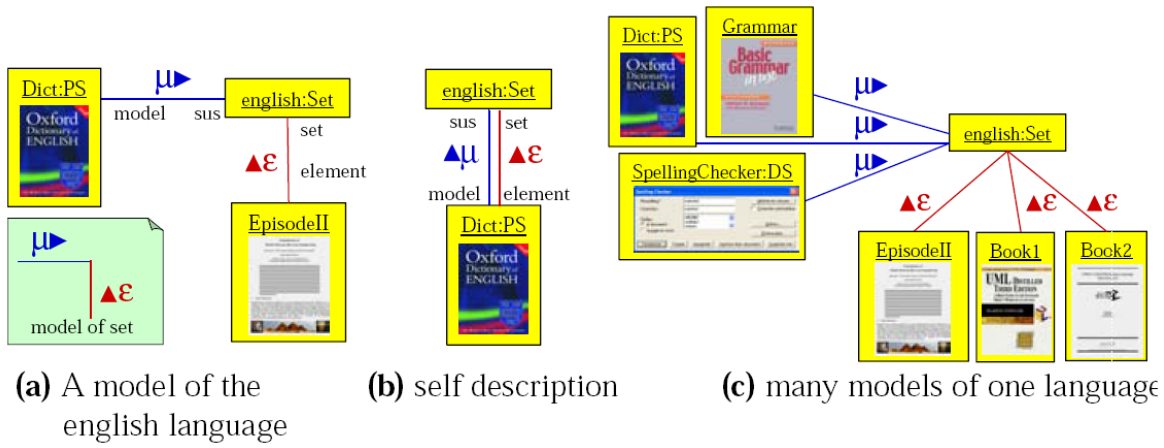


Figura 4.2: Relación entre meta-modelo y modelo (obtenido de [82])

de modelado como el meta-modelo de dicho lenguaje coinciden y son el mismo. Es decir, un meta-modelo reflexivo es su propio modelo, lo que acaba con la recursividad en la definición. Esta es la solución adoptada en MDE y MDA gracias a MOF (ver apartado 4.4).

4.2.3 LIMITACIONES DEL ENFOQUE MDE: VISIÓN ONTOLÓGICA

Atkinson propone en [16] las características que debe cumplir un verdadero entorno de desarrollo MDE para que el enfoque tenga éxito y cumpla todas las expectativas depositadas en él. Destaca, además, la íntima relación que existe entre un meta-modelo y una ontología, como dos formas distintas pero complementarias de plasmar la misma realidad. Afirma que MDE tiene que ser capaz de unificar estas dos vistas de un modelo, de forma que se evite la asimetría producida cuando el modelado se centra únicamente en una de ellas. Es lo que el autor denomina «las dos vistas de la instanciación: la instanciación lingüística y la ontológica».

El enfoque MDE más utilizado actualmente, MDA, está organizado en cuatro capas de abstracción obtenidas desde el punto de vista lingüístico únicamente (ver figura 4.3), lo que provoca que relaciones ontológicas que están en niveles de abstracción distintos queden mezcladas, confundidas en un mismo nivel (lingüístico en este caso), e.g. los conceptos *clase* y *objeto* se encuentran ambos en dicha figura en el nivel *M1*, cuando conceptualmente no se encuentran al mismo nivel de abstracción (una clase define un tipo y un objeto es una instancia de dicho tipo). Organizar las capas de desarrollo MDE desde un punto de vista ontológico, como muestra la figura 4.4, tampoco soluciona el problema, ya que en este caso los conceptos de meta-clase, clase y objeto se encuentran entremezclados en el mismo nivel. La solución reside en la unificación de ambas vistas [106].

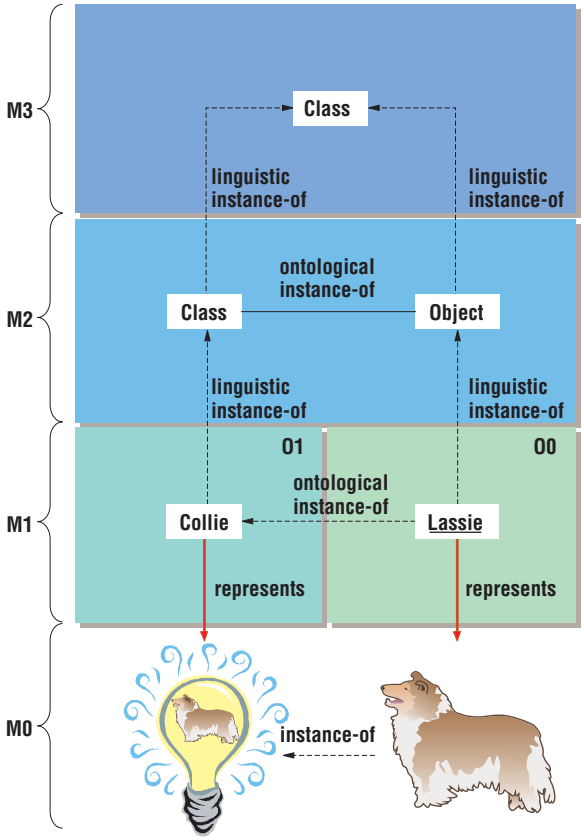


Figura 4.3: Distribución en capas de **MDE** desde un punto de vista lingüístico (extraído de [16])

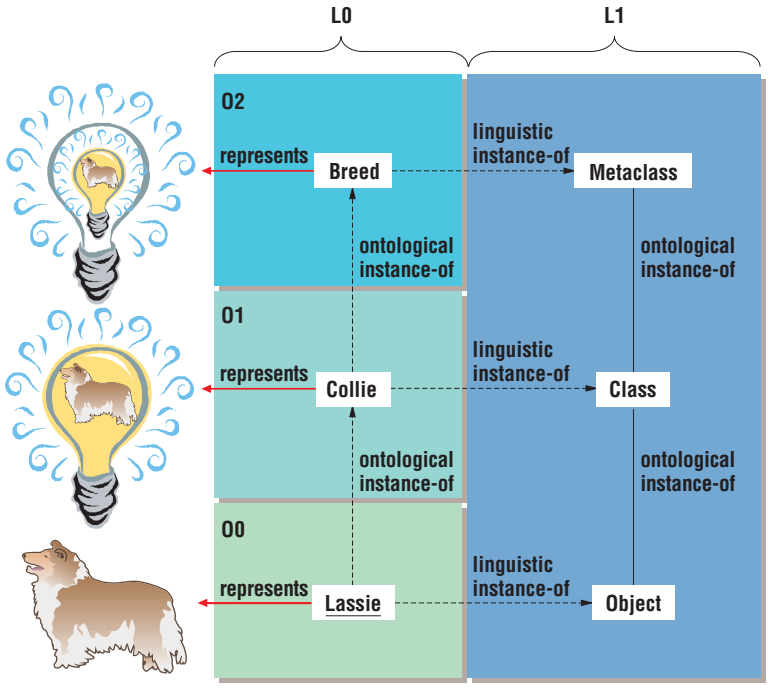


Figura 4.4: Distribución en capas de **MDE** desde un punto de vista ontológico (extraído de [16])

4.3 TRANSFORMACIONES DE MODELOS

LAS TRANSFORMACIONES de modelos son el mecanismo fundamental que provee MDE para manejar la evolución de los modelos en el proceso de desarrollo. Las transformaciones de modelos permiten convertir un modelo entre distintos meta-modelos y mantener los cambios de forma coherente y sincronizada, de forma que no exista ninguna diferencia funcional entre los artefactos que usan los desarrolladores (modelos) y los que utiliza la plataforma (binarios). Los meta-modelos que se utilizan como origen y destino pueden expresar distinto niveles de abstracción (transformación vertical) o distintos puntos de vista del mismo sistema (transformación horizontal) [196].

El cumplimiento del principio de MDE *«todo es un modelo»* implica que la transformación debe definirse también como un modelo, lo que conlleva la necesidad de definir un meta-modelo con que expresarla. La figura 4.5 muestra el esquema genérico de una transformación entre distintos meta-modelos generados a partir del mismo meta-metamodelo, como sucede en el enfoque MDA. En dicha figura puede observarse cómo se transforma un modelo M^1-a (conforme al meta-modelo M^2-a) en otro modelo M^1-b (conforme al meta-modelo M^2-b) por medio de una transformación M^1Tab (conforme al meta-modelo M^2T) que define la relación entre los elementos del meta-modelo M^2-a y M^2-b. De esta figura cabe destacar que la transformación M^1T-ab se define con los elementos de los meta-modelos cuyos modelos se quieren transformar, pero que se aplica finalmente a dichos modelos, es decir M^1-a $\overset{M^1Tab}{\longrightarrow}$ M^1-b. Las transformaciones de modelos no tienen por qué ser 1:1, i.e. un modelo de entrada genera uno de salida, sino que es posible generar varios modelos de salida a partir de uno de entrada (transformaciones PIM⇒PSMs en MDA) o que varios modelos de entrada generen un único modelo de salida (fusión de varias vistas). A continuación se enumeran algunas de las definiciones de «transformación» presentes en la bibliografía:

Kleppe (2.003) [131]. Una transformación es la generación automática de un modelo a partir de otro modelo fuente, de acuerdo a una serie de reglas. Una definición de transformación es un conjunto de reglas de transformación que juntas describen cómo se transforma un modelo descrito en el lenguaje origen a un modelo descrito en el lenguaje destino. Una regla de transformación es una descripción de cómo se transforma una o más construcciones del lenguaje origen en una o más construcciones en el lenguaje destino.

Mellor (2.004) [148]. Una transformación es la aplicación de una función de transformación para transformar un modelo en otro. Una función de transformación es un conjunto

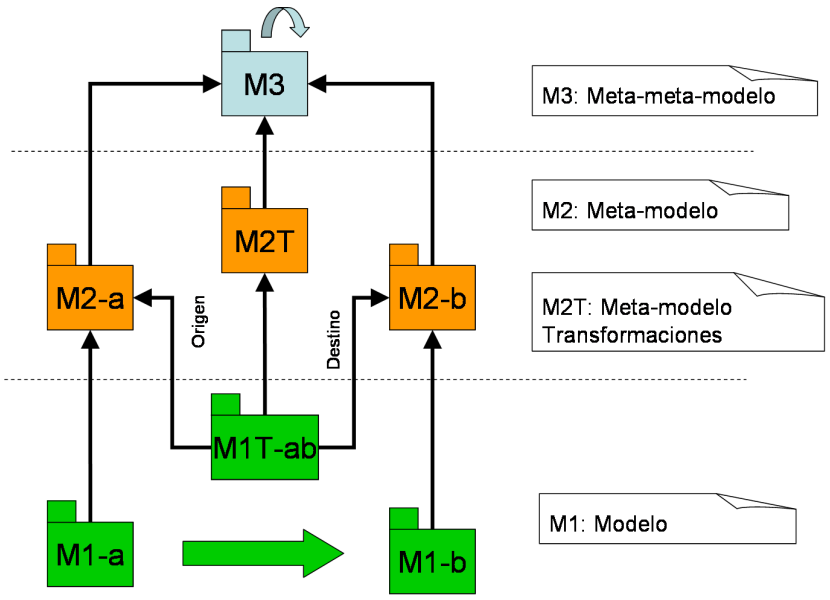


Figura 4.5: Esquema de transformación de modelos con meta-metamodelo común (extraído de [49])

de reglas que definen cada uno de los aspectos de funcionamiento de una función de transformación.

MOF-QVT v2.0 (2.005) [168]. Una transformación genera un modelo destino a partir de un modelo fuente. Una vista es un tipo de transformación restringida, que impone la restricción de que el modelo obtenido no puede ser modificado independientemente del modelo fuente.

Existen diversas tecnologías para acometer la tarea de transformar modelos: (1) manipulación directa del modelo; (2) utilización de una representación intermedia y (3) utilización de un lenguaje de transformación. Sendall [196] compara las ventajas e inconvenientes de cada una de estas opciones y concluye que la mejor solución es el desarrollo de lenguajes específicos para realizar estas transformaciones, lenguajes que pueden ser híbridos (e.g. imperativo y declarativo) para aprovechar mejor las características de cada uno de ellos. Sendall afirma que la clave del diseño de estos lenguajes es que ofrezcan abstracciones que sean intuitivas y que cubran el mayor número posible de situaciones. Mens [149] describe las principales características de una transformación de modelos:

Automatización. Es más que deseable que existan mecanismos que permitan realizar o programar en lotes un conjunto de transformaciones para que sean aplicadas a un conjunto de modelos origen. También es necesario que existan transformaciones que requieran cierto nivel de intervención manual por parte del usuario. Es necesario que se contemplen ambas posibilidades.

Complejidad de la transformación. La realización de transformaciones sencillas puede utilizar unas técnicas y métodos totalmente distintos de los que puede necesitar una transformación más compleja y profunda.

Preservación del significado. Cada transformación preserva ciertos aspectos del modelo origen en el modelo destino. Por ejemplo, algunas transformaciones deben mantener el comportamiento mientras que otras deben preservar la estructura. El espacio tecnológico [136] en que se desarrolle dicha transformación también influye sobre los aspectos que debe preservar la transformación.

Czarnecki [69] primero y posteriormente Mens [149] proponen sendas taxonomías para clasificar los distintos tipos de transformaciones de modelos. Czarnecki es el primero en distinguir dos tipos de transformaciones de modelos, dependiendo del tipo de artefacto generado y la fase del desarrollo en que se realizan:

Modelo-a-modelo (Model-To-Model (**M2M**)). Como su nombre indica, estas transformaciones producen como salida otro modelo, otra representación de la realidad. Las transformaciones M2M son las herramientas utilizadas a lo largo del proceso de desarrollo para hacer evolucionar los modelos, ya que lo normal es que progresivamente se vaya añadiendo detalle al modelo original. Czarnecki distingue distintas formas de abordar la transformación típica: (1) manipulación directa de la representación de bajo nivel del modelo (posible con cualquier lenguaje); (2) mediante la teoría de transformaciones de grafos; (3) utilización de lenguajes declarativos, relaciones matemáticas o reglas de conversión y (4) aproximaciones híbridas que mezclan varias de las técnicas mencionadas.

Modelo-a-texto Model-To-Text (**M2T**). Es una particularización de las transformaciones M2M para generar directamente una representación textual del modelo de origen sin la necesidad de que exista un meta-modelo destino. En este sentido, son transformaciones libres, no restringidas. Las transformaciones M2T se suelen utilizar en las últimas etapas de desarrollo para generar el código de la aplicación, y de esta forma reutilizar la tecnología de compiladores existente. Aunque se pueden utilizar para otras misiones, como generar documentación u otras representaciones, por ejemplo en formato HTML. Czarnecki identifica dos métodos para realizar las transformaciones M2T: (1) utilizando el patrón *Visitor* [96] para recorrer el modelo y generar texto o (2) mediante la utilización de soluciones basadas en plantillas, que mezclan el texto de la salida con comandos para recorrer el modelo.

La propuesta de Mens [149] clasifica las transformaciones de modelos en dos ejes ortogonales: según el lenguaje (meta-modelo) y dependiendo del nivel de abstracción

(transformaciones horizontales y verticales). Además establece una tercera clasificación, dependiendo de si la transformación cambia únicamente la sintaxis del modelo o de si cambia también su semántica. Dependiendo de si el lenguaje de los modelos origen y destino es el mismo (transformación *endógena*) o distinto (transformación *exógena*) realiza la siguiente clasificación:

Transformación endógena: *optimización* (mejora de alguna característica del modelo original manteniendo su semántica), *refactorización* (cambio de la estructura interna para mejorar alguna característica sin modificar el comportamiento [141]), *simplificación y normalización* (para disminuir la complejidad sintáctica) y *adaptación* (para modificar o adaptar modelos ya existentes, posiblemente tras la aparición de una nueva versión de su meta-modelo).

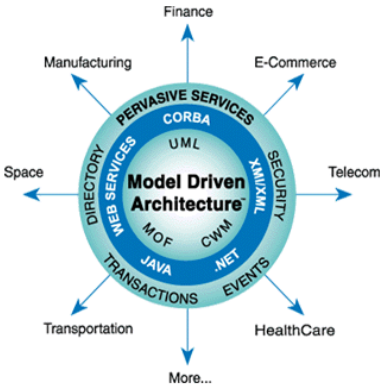
Transformación exógena: *síntesis* (obtención de un modelo en un nivel de abstracción más bajo), *ingeniería inversa* (obtención de un modelo a un nivel de abstracción mayor que el original) y *migración* (cambio de representación manteniendo el mismo nivel de abstracción).

Finalmente, este tipo de transformaciones pueden verse como otra aplicación de la disciplina conocida como «programación generativa» (Generative Programming (**GP**)) [67, 68]. Según Czarnecki, la programación generativa es «un paradigma que permite la creación bajo demanda y de forma personalizada de un producto intermedio o final altamente optimizado a partir de componentes elementales y reutilizables».

> *Every software modification can be seen as a transformation instance [...] In fact currently software evolution is driven by ad-hoc transformation instances while the goal of Model Driven Engineering is to drive the process through a set of reusable transformation functions.*
>
> — Jean-Marie Favre & Tam NGuyen [84]

4.4 Arquitectura Guiada por Modelos (MDA)

La arquitectura guiada por modelos, Model-Driven Architecture (**MDA**) [131, 148, 165], es la versión del Object Management Group (**OMG**) del enfoque de desarrollo por modelos MDE. En esta sección se van a describir únicamente las diferencias que existen entre el enfoque genérico MDE y el particular de el OMG, así como las principales herramientas y estándares que el OMG ha desarrollado en el contexto de su enfoque MDA y que han contribuido enormemente al despegue de MDE.



La iniciativa MDA[1] fue presentada como una especificación de el OMG en el año 2.001. MDA organiza el desarrollo software en tres capas organizadas desde el punto de vista lingüístico (no contemplan ninguna organización ontológica), denominadas *M3* a *M1* como muestra la figura 4.6. La capa superior (*M3*) contiene el meta-meta-modelo *reflexivo* común a todos los estándares de el OMG, denominado Meta-Object Facility (**MOF**) [167]. La capa intermedia (*M2*) contiene distintos meta-modelos (como el Unified Modelling Language (**UML**) [170] o el Common Warehouse Metamodel (**CWM**) [164]) que son utilizados por el usuario para crear los modelos que describen sistemas reales, modelos que residen en el nivel más bajo *M1*. Además, el OMG ha desarrollado el estándar XML Metadata Interchange (**XMI**) [171] para describir el formato de intercambio y almacenamiento persistente de modelos y meta-modelos; el estándar Object Constraint Language (**OCL**) [174] para definir restricciones sobre los modelos; el estándar Query, View and Transformation (**QVT**) [168] para especificar transformaciones entre modelos así como el modelo de desarrollo software Software Process Engineering Metamodel (**SPEM**) [169]. Estos estándares proporcionan un conjunto de herramientas y meta-modelos que completan su enfoque MDA.

Además de la organización del desarrollo en tres capas con un meta-meta-modelo *reflexivo* (MOF) en la capa superior, el *patrón MDA* [165] se basa en la utilización de UML como meta-modelo con el que especificar el diseño software. En una última etapa, estos modelos UML son utilizados para llevar a cabo la generación de código. MDA organiza el proceso de desarrollo software en torno a tres familias de modelos: CIM, PIM y PSM (ver más adelante). Aunque el nivel de abstracción de cada una de las familias es distinto (va de mayor nivel de abstracción a menor), todos los modelos se encuentran en el nivel *M1*; por tanto, el proceso (o patrón) MDA se aplica únicamente de manera horizontal. Esta son las dos diferencias fundamentales entre el enfoque genérico MDE y MDA: la utilización del meta-modelo de UML (incluyendo sus *profiles*, como SysML) y la organización del nivel de abstracción en tres capas (CIM, PIM y PSM). La figura 4.6 muestra las diferencias entre MDA y MDE de forma gráfica utilizando la pirámide MDA.

Como ya se ha mencionado, MDA organiza el proceso de desarrollo software en torno a tres familias de modelos. El primero de estos modelos es el más abstracto y representa el modelo de negocio de forma independiente a la computación o Computation Independent Model (**CIM**). El CIM sitúa y relaciona al sistema con el entorno en que opera, y permite no solo ayudar a entender la problemática sino que también proporciona el vocabulario

[1] La página web `http://www.omg.org/legal/tm_list.htm` contiene todos los acrónimos registrados por el OMG, entre los que destaca el conocido Model-Driven Development (**MDD**) [165]

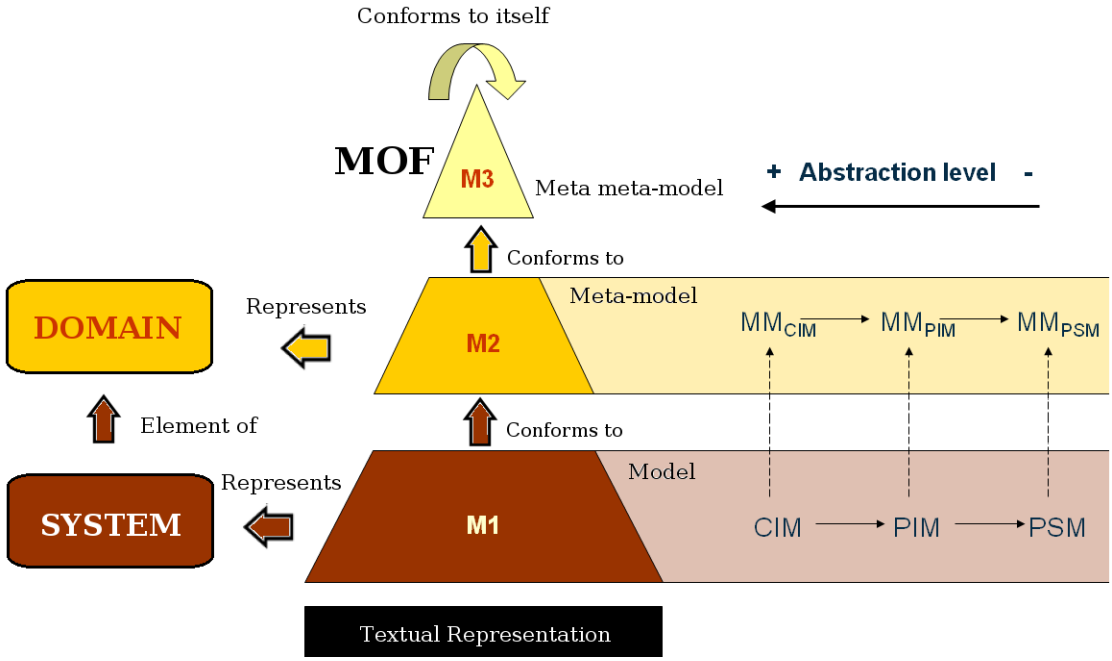


Figura 4.6: Estructura de **MDA** en comparación con **MDE**

que van a utilizar el resto de familias de modelos. En el siguiente nivel de abstracción aparecen los modelos independientes de la plataforma o Platform-Independant Model (**PIM**), que proporcionan mayor nivel de detalle que el CIM pero que no muestran aspectos de implementación, como sistema operativo, *middleware* de comunicación, *framework* de componentes, etc. Estos detalles son mostrados, con mayor o menor detalle, por los modelos específicos de la plataforma o Platform-Specific Model (**PSM**). La transformación PIM→PSM, que no tiene porque ser 1:1, se realiza mediante unos *modelos de marcas*, que contienen la información adicional necesaria para poder realizar la transformación. La información que contienen los *modelos de marcas* no es necesaria desde el punto de vista del diseño, por lo que es colocada en ellos para evitar que contaminen los modelos del diseño (PIMs y PSMs). El enfoque MDA utiliza UML como meta-modelo con el que crear cada uno de estos tres niveles intermedios.

El Meta-Object Facility (**MOF**) [167] es el meta-meta-modelo *reflexivo* que constituye el corazón de MDA y que se asienta en la capa *M3*. Según la página web de el OMG, MOF proporciona el entorno en que los modelos pueden ser exportados e importados entre herramientas, almacenados y extraídos de repositorios, transformados entre distintos meta-modelos y usados para generar código. Esta aplicación no está restringida a modelos basados en UML, sino que es extensible a modelos cuyo meta-modelo esté basado en MOF. MOF es, por tanto, una especificación orientada a desarrolladores de herramientas **CASE** (*Computer Aided Software Engineering*) que pretendan

seguir un enfoque de desarrollo MDE. MOF permite, como puede observarse en la figura 4.6-a, que cualquier usuario cree un nuevo meta-modelo para describir sus aplicaciones. De esta forma es posible realizar transformaciones tanto entre distintos meta-modelos (que pueden mostrar distintas vistas de un mismo sistema o distintos niveles de abstracción) como entre distintos espacios tecnológicos [136].

Los dos últimos estándares que se van a describir son el lenguaje para la especificación de restricciones OCL [174] y el lenguaje de definición de transformaciones de modelos QVT [168], ambos basados en el estándar MOF y desarrollados por el OMG para completar su enfoque MDA. Aunque el OMG ha publicado recientemente la especificación de un lenguaje de transformación de modelo a texto [173], todavía no existe ninguna implementación del mismo.

OCL. Es un lenguaje formal de especificación, por lo que la evaluación de sus expresiones no provoca efectos colaterales en los modelos (como la alteración de propiedades o relaciones entre *Metaclases*). OCL permite definir aquellas restricciones adicionales que no pueden expresarse utilizando MOF únicamente. OCL puede utilizarse para (1) examinar modelos; (2) especificar invariantes en atributos y pre y post-condiciones en operaciones, en forma de contratos [151]; (3) describir guardas y (4) definir restricciones adicionales en los modelos y comprobar que un modelo es correcto.

QVT. Es un lenguaje híbrido declarativo–imperativo que permite examinar un modelo en búsqueda de determinadas características o relaciones y definir transformaciones entre modelos conformes al mismo o a distintos meta-modelos. El diseño de la parte declarativa de QVT está dividida en dos capas que forman la base sobre la que se ejecuta la parte imperativa del lenguaje. QVT puede utilizarse para (1) definir transformaciones uni y bidireccionales; (2) realizar actualizaciones incrementales entre modelos; (3) crear y destruir cualquier objeto presente en el meta-modelo y (4) definir transformaciones para comprobar si determinados modelos están relacionados.

4.5 OTRAS HERRAMIENTAS RELACIONADAS

PARA CONCLUIR este capítulo sobre el enfoque de desarrollo software basado en modelos se describen someramente los lenguajes específicos de dominio y las factorías software de Microsoft. La primera tecnología lleva utilizándose satisfactoriamente muchos años en el mundo software y su uso se ha incrementado gracias a su perfecta integración con MDE. La segunda de ellas, factorías software, es la respuesta de Microsoft al desarrollo de MDE.

4.5.1 Lenguajes específicos de dominio

Los lenguajes específicos de dominio ((Domain-Specific Language (**DSL**))) son «lenguajes de programación o especificaciones ejecutables que ofrecen, gracias a una notación y a un nivel de abstracción adecuados, un gran poder expresivo centrado, y generalmente limitado, a un dominio concreto» [208]. Un DSL se distingue de un lenguaje de programación de propósito general precisamente en que su ámbito de aplicación es un dominio concreto, pero su poder expresivo en este ámbito es mucho mayor porque ha sido diseñado específicamente para el dominio. En su estudio del año 2.000, Deursen [208] identifica más de 75 DSLs en la bibliografía.

Los DSLs existen desde hace más de treinta años y se aplican a diversos dominios: especificación de sintaxis (*BNF*), construcción software (*Makefile*), bases de datos (*SQL*), diseño hardware (*VHDL*), edición de textos (*LaTEX*), web (*HTML*), compiladores (*yacc* y *lex*) y un largo etcétera. En general es fácil distinguir un DSL de un lenguaje de propósito general, pero a veces la línea que los separa es difusa, y la clasificación depende de quién la realiza [150] (como por ejemplo el *shell* de Linux). A pesar de las ventajas que supone disponer de un DSL para una aplicación, la decisión de emprender su diseño debe ser meditada, contrastando las ventajas e inconvenientes que muestra el cuadro 4.1. En su artículo, Mernik [150] propone un método para discernir cuándo es necesario desarrollar un DSL y cómo hacerlo una vez llegado el momento; y junto a Spinellis [200] proponen una serie de patrones para ayudar al diseñador de este tipo de lenguajes.

A pesar de que se llevan utilizando durante muchos años, existe un reciente y creciente interés por su diseño y utilización, coincidiendo con el auge del enfoque de desarrollo basado en modelos (MDE) y del desarrollo basado en líneas de producto [62]. La unión entre DSLs y MDE es sinérgica y esta es la razón de que haya aumentado tanto el interés por su desarrollo: el enfoque MDE puede ayudar a reducir el coste (económico, temporal, recursos, de mantenimiento, etc) de desarrollo de un DSL y los DSLs pueden utilizarse en un contexto MDE para guiar las distintas etapas de diseño. Existen diversas herramientas en el mercado especializadas en el desarrollo de DSLs, como *MetaEdit+*[2] y *Microsoft DSL Tools*[3]. El desarrollo de lenguajes específicos es una de actividades principales en el enfoque de Microsoft *Software Factories* [105].

A pesar de las ventajas de diseño y mantenimiento que aporta un DSL y de la existencia de herramientas que reducen los costes de desarrollo, existe todavía el problema de la incompatibilidad entre programas escritos en distintos DSLs [81]. Dada la utilidad de los DSLs, sobre todo para el desarrollo basado en líneas de productos, Estublier propone

[2] `http://www.metacase.com`
[3] `http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx`

VENTAJAS	INCONVENIENTES
Expresan soluciones en el lenguaje y nivel de abstracción del dominio del problema. Facilitan a los expertos del dominio el desarrollo, entendimiento y validación de los programas.	La dificultad para encontrar un equilibrio entre los conceptos propios del dominio y la necesidad de disponer de construcciones de propósito general.
Los programas son concisos, auto-explicativos y a menudo pueden ser reutilizados.	El coste de diseño, implementación y mantenimiento.
Mejoran la productividad, la fiabilidad, la portabilidad y el mantenimiento de los programas.	La baja disponibilidad de DSLs
Un DSL contiene el conocimiento del dominio y contribuye a su conservación y reutilización	La dificultad para encontrar el alcance adecuado que justifique su desarrollo.
Permiten la validación y optimización a nivel de dominio.	La potencial pérdida de eficiencia frente a software desarrollado «a mano».
Facilitan la realización de tests, que además son más significativos.	El coste de formación de los usuarios.

Cuadro 4.1: Ventajas e inconvenientes de desarrollar un DSL (extraído de [208])

en [81] un enfoque para facilitar la composición de programas escritos en distintos DSLs y de esta forma soportar la utilización de distintos DSLs en una línea de productos: la plataforma *Mélusine*.

4.5.2 FACTORÍAS SOFTWARE

Las factorías software (Software Factories (**SF**)) [105] representan la visión particular de Microsoft del desarrollo basado en modelos. Una factoría software permite definir una metodología adaptada a una familia de productos, que ayuda a los diseñadores a crear de forma rápida nuevas aplicaciones dentro de la familia. Una factoría software es, según Greenfield, «una línea de productos software que proporciona las herramientas necesarias para su diseño gracias a que es capaz de configurar otras herramientas extensibles por medio de plantillas software basadas en esquemas software» [105]. La figura 4.7 muestra un esquema de una factoría software.

Como ya se ha comentado, las factorías software se emplean para desarrollar productos en el contexto de un dominio concreto usando lo que el enfoque denomina «un grafo de puntos de vista del producto». Cada uno de estos puntos de vista define algún aspecto del ciclo de desarrollo del producto, como la captura de requisitos, el diseño de contratos e interfaces, la definición de tablas para una base de datos, etc. Los artefactos

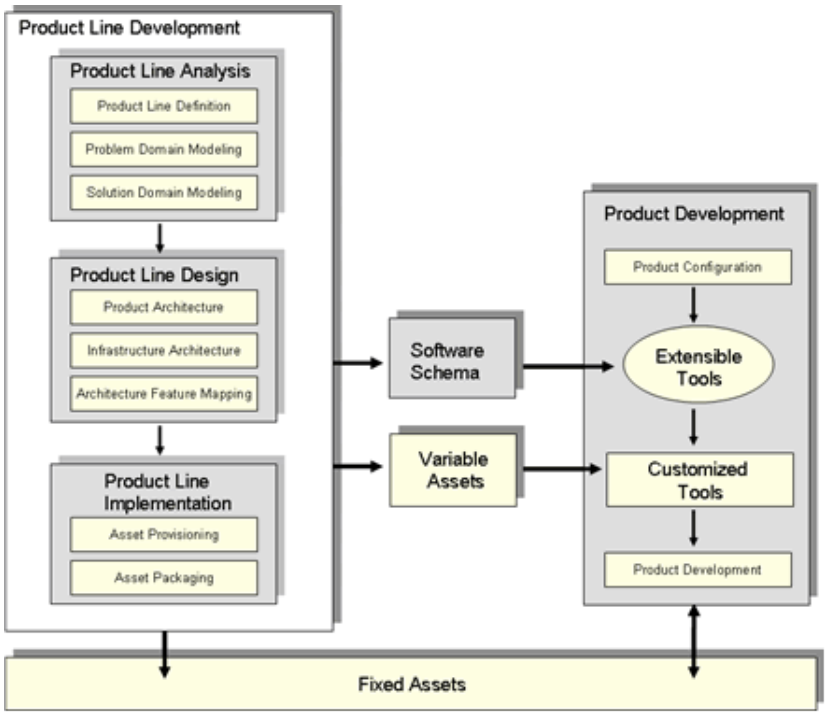


Figura 4.7: Esquema de una factoría software

producidos por las herramientas suministradas en cada uno de estos puntos de vista son luego reutilizables en otros productos y puntos de vista. De esta forma, una factoría software puede verse como un paquete que contiene la configuración de las distintas herramientas que se utilizan cuando se sigue un esquema de desarrollo basado en líneas de producto y de las relaciones entre los artefactos generados por cada una de ellas.

Para conseguir su objetivo la factoría está formada por dos componentes principales: la *plantilla* de la factoría y el *esquema* de la factoría. La misión de la *plantilla* es configurar un entorno de desarrollo extensible (como *Eclipse* o Microsoft *Visual Studio*) con las herramientas adecuadas a la elaboración de los artefactos definidos por cada punto de vista. El enfoque SF promueve el uso de los lenguajes específicos de dominio (DSLs) [208] para el desarrollo de estas herramientas. El *esquema* de la factoría contiene las relaciones entre los artefactos software generados por las herramientas en forma de grafo que une los puntos de vista y marca el flujo de diseño.

4.6 CONCLUSIONES Y APORTACIONES A LA TESIS

EL DESARROLLO software basado en modelos es una técnica relativamente nueva. Aunque los primeros artículos datan del año 2.000, hasta hace unos años no han estado disponibles las primeras herramientas que hicieran posible su utilización. En este punto ha sido fundamental la contribución del OMG al desarrollar numerosos estándares para completar su enfoque MDA. En este capítulo se han expuesto las características fundamentales del desarrollo MDE: del modelado de la realidad con distintos niveles de detalle a la definición de las transformaciones que guían la evolución de los modelos a lo largo del proceso de desarrollo software.

Las aportaciones de este capítulo al desarrollo de esta Tesis Doctoral son fundamentales. En primer lugar, el nivel de abstracción que se alcanza cuando se trabaja con modelos es mucho mayor que el que se puede obtener al utilizar cualquier otra técnica o paradigma de programación existente, por lo que MDE se perfila como la tecnología perfecta para desarrollar unos componentes realmente independientes de la plataforma. En segundo lugar, MDE trabaja con los conceptos del dominio de aplicación en vez de con los conceptos, en este caso, del lenguaje de programación, lo que redunda en la obtención de modelos más sencillos, expresivos y comprensibles, incluso por personas que no iniciadas en la disciplina de Ingeniería del Software. El capítulo 6 describe el meta-modelo de componentes V^3Studio, que ha sido especialmente diseñado para describir aplicaciones basadas en el ensamblaje de componentes.

En tercer lugar, las transformaciones de modelos hacen posible que los modelos evolucionen y no se queden en meras representaciones o documentación, desconectados de la implementación real en un lenguaje de programación. Gracias a las transformaciones de modelos se va a hacer posible otro de los objetivos de esta Tesis Doctoral: ser capaces de obtener distintas implementaciones de una misma aplicación basada en componentes en distintos lenguajes de programación. Este objetivo está plasmado en los capítulos 7 y 8. Concretamente, el capítulo 7 describe una transformación de modelos cuyo objetivo es proponer una posible traducción de los conceptos CBD en que se basa V^3Studio a una tecnología orientada a objetos [116]. Para mantener la generalidad y la independencia de la plataforma final de ejecución, se eligió UML como meta-modelo de destino para esta transformación de modelos. Por otro lado, el capítulo 8 muestra una posible transformación a código del modelo UML obtenido en el paso anterior. Esta transformación a código es sólo una muestra de que todo el proceso descrito es factible y de que sería posible desarrollar subsiguientes transformaciones a otros lenguajes de programación (no necesariamente orientados a objetos).

CAPÍTULO 5

ARQUITECTURAS Y FRAMEWORKS DE CONTROL DE ROBOTS

ESTE capítulo sobre el estado de la técnica en arquitecturas software de control de robots finaliza el primer bloque en que se ha estructurado esta Tesis Doctoral. La historia del desarrollo de arquitecturas de control de robots ha sido guiada por la búsqueda de una arquitectura única, que fuera capaz de adaptarse a cualquier situación y controlar cualquier robot, independientemente de su diseño. Sin embargo, Tom Smither refleja en su ejemplo de la araña («*extremely competent at survival in the countryside, but utterly incompetent in the bathtub!*») que no existe la arquitectura perfecta, ideal para todas las situaciones posibles.

Este capítulo comienza realizando un breve repaso de las arquitecturas de control clásicas y exponiendo las nuevas tendencias en diseño software para el control de robots, para terminar realizando una breve descripción de ACRoSeT. ACRoSeT es un marco arquitectónico basado en componentes abstractos, independientes de la tecnología de implementación final, que no impone ningún estilo arquitectónico en particular sino que permite posponer dicha elección hasta el momento de realizar la implementación final. Gracias a estas características es posible desarrollar distintas implementaciones de un mismo sistema de control, adaptadas a las necesidades específicas de cada robot. La unión entre el enfoque de desarrollo MDE y ACRoSeT permite no sólo simplificar el proceso de traducción de la arquitectura sino que también permitirá adaptar los componentes ACRoSeT de forma que será posible utilizar los componentes propuestos por cualquiera de los frameworks de control de robots que existen actualmente.

5.1 BREVE HISTORIA DE LA ROBÓTICA

EN EL cuadro 5.1 se presenta un breve resumen con algunos de los hitos más importantes en la corta historia de la robótica moderna, mientras que la figura 5.1 muestra algunos de los robots cuyo diseño marcó hitos históricos en la historia de la robótica reciente. En dicha figura pueden verse, de izquierda a derecha y de arriba a abajo, respectivamente, el primer robot diseñado por Unimate, el robot Shakey, el Stanford Cart, el robot Dante II, el rover de exploración de Marte Sojourner, el perro robot AIBO, la saga de robots humanoides de Honda y, por último, el robot HUBO.

1953	Se diseña una tortuga robótica a partir de las ideas de Norbert Wiener, fundador de la rama conocida como cibernética, una combinación de informática, teoría de control y biología.
1956	Joseph Engelberger y George Deroe fundaron la primera compañía de manufactura de robots, denominada Unimate.
1961	Unimate distribuye el primer robot comercial de la historia.
1968	El Instituto de Investigación de Standford (**SRI**) diseña y construye el robot Shakey. Shakey dispone de una cámara de vídeo y sensores de fin de carrera para detectar colisiones.
1970	La compañía sueca ASEA crea una división robótica, actualmente ABB Robotics, que distribuye su primer robot en 1974.
1973	La compañía KUKA desarrollo el primer robot industrial.
1975	El Laboratorio de Arquitectura y Análisis de Sistemas (**LAAS**) de Toulouse, Francia diseña el robot Hilare, equipado con una cámara de vídeo, un sensor láser y varios sensores de ultrasonidos.
1977	El **SRI** diseña el *Stanford Cart*, un vehículo que utiliza visión estereoscópica para navegar. El proceso de visión era tan lento que el vehículo sólo podía desplazarse cuatro metros por hora.
1978	Unimation desarrolla el robot *PUMA* (*Programmable Universal Machine for Assembly*)
1984	La universidad japonesa de Waseda diseñó un robot humanoide que tocaba el piano.
1990	La Carnegie Mellon University (**CMU**) diseñó el vehículo Navlab 5 y lo probó en autopista.
1994	La **CMU** desarrolla el vehículo Dante-II, un robot de seis patas con el que explora el monte Spurr en Alaska.
1995	En un proyecto denominado *«No Hands Across America»*, el robot Navlab 5 recorrió por carretera 4.500 km entre las ciudades de Pittsburgh y San Diego.
1996	La compañía sueca Husqvarna desarrolla un robot corta-césped.
1996	HONDA presenta el primero de su saga de robots humanoides, el P2.
1997	El robot rover Sojourner de la **NASA** se desplaza por la superficie de Marte.
1999	Sony crea el robot perro AIBO destinado al entretenimiento.
2000	Electrolux diseña y distribuye el primer robot aspiradora Trilobite.
2002	HONDA crea el robot humanoide ASIMO, capaz de reconocer y hablar con su dueño.
2005	El Instituto de Ciencia y Tecnología de Corea (**KIST**) diseña el robot humanoide HUBO.

Cuadro 5.1: Breve historia de la robótica

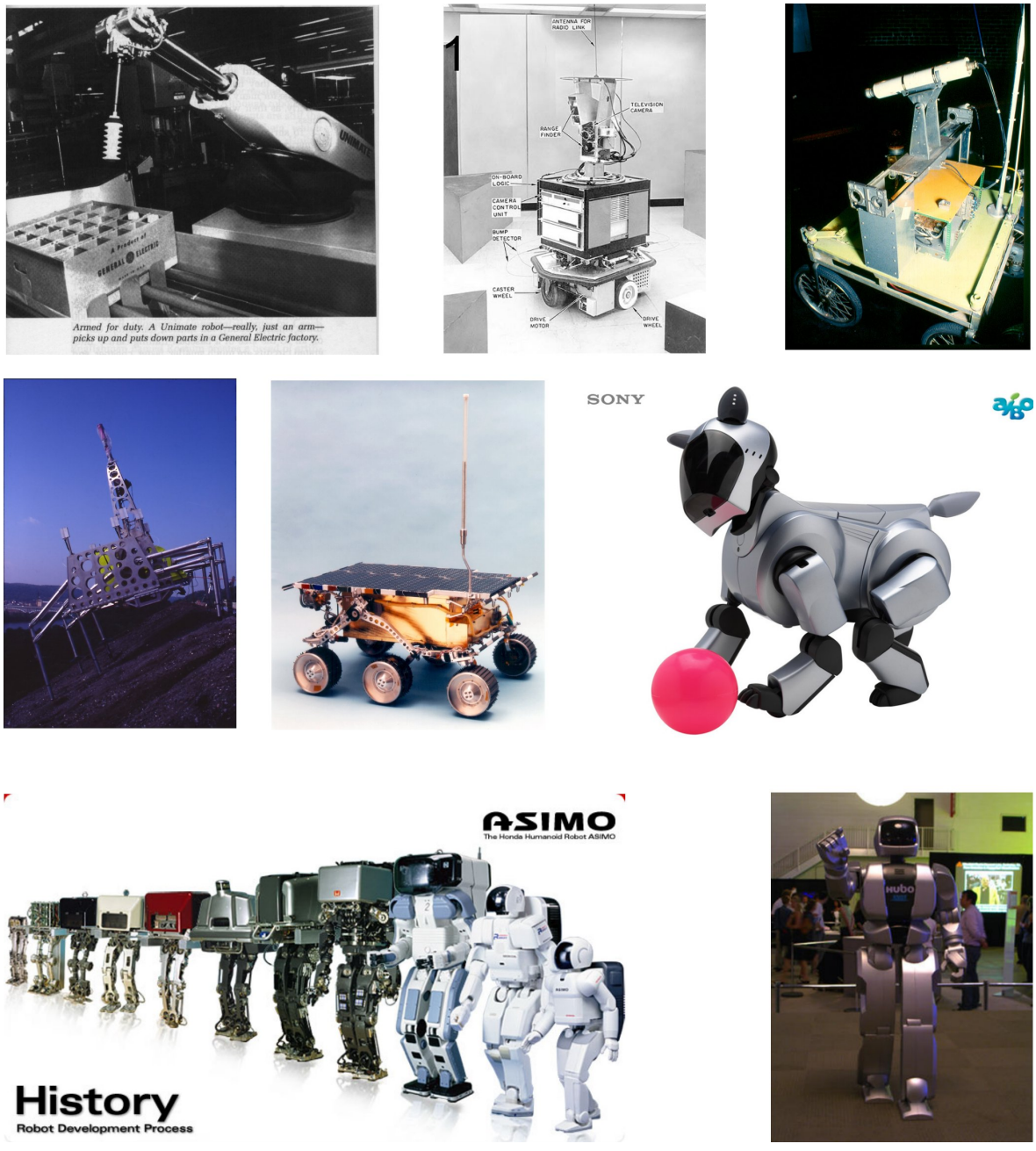


Figura 5.1: Algunos robots que marcaron un hito histórico

5.2 ARQUITECTURAS DE CONTROL DE ROBOTS

COMO ya se comentó en la sección 3.1, la arquitectura de un sistema es uno de los pilares en que se asienta su diseño y que condiciona tanto sus prestaciones como sus posibilidades futuras de ampliación y modificación. Por tanto, no es de extrañar que el estudio de la arquitectura de los sistemas robóticos sea un área de gran importancia dentro de esta disciplina.

> *An architecture should facilitate the development of robotic systems by providing beneficial constraints on design and implementation of the desired applications, without being overly restrictive.*
>
> — Coste-Maniere & Simmons [64]

Aunque este criterio es fácil de expresar, es ciertamente difícil de llevar a la práctica. En particular, distintos tipos de aplicaciones pueden tener necesidades diferentes, que requieren del desarrollo de arquitecturas diferentes, adaptadas y ajustadas a las necesidades particulares de cada sistema y cada robot. El diseño de la arquitectura de un robot ha ido evolucionando desde que aparecieron los primeros robots controlados por computador en los años ochenta.

A lo largo de todos estos años se han propuesto, con mayor o menor éxito, distintas arquitecturas software para realizar el software de control de estos robots. Arkin [15] y Coste-Manière [64] realizan, respectivamente, sendos estudios de las principales arquitecturas de control para robots y ambos destacan que no puede existir una arquitectura genérica única para diseñar cualquier sistema. Esta sección realiza un repaso cronológico de los principales modelos de arquitectura de control de robots existentes hasta el momento: la arquitectura deliberativa, la arquitectura reactiva y la arquitectura híbrida, mezcla de las dos anteriores.

5.2.1 ARQUITECTURA DELIBERATIVA O JERÁRQUICA

También denominadas «Sensor-Plan-Acción», estas arquitecturas fueron las primeras en ser desarrolladas y su diseño está influido en gran manera por el ciclo de control de los actuadores del robot, tal y como muestra la figura 5.2. Estas primeras arquitecturas estaban más centradas en conseguir que el robot fuera capaz de moverse que en considerar las limitaciones que la arquitectura imponía en el rendimiento del robot y en otras características no funcionales, como la modificabilidad, mantenimiento, etc.

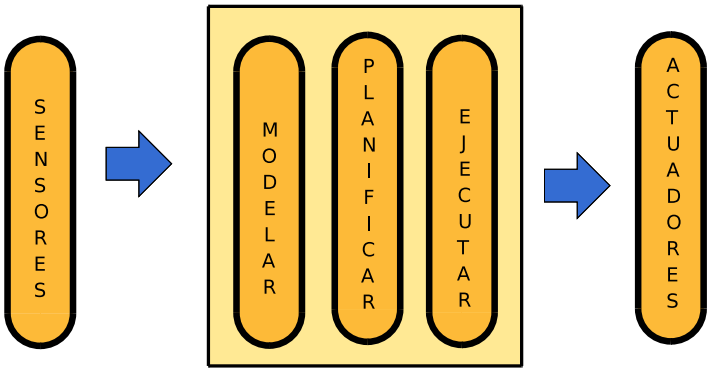


Figura 5.2: Arquitectura de control deliberativa

Una arquitectura deliberativa [162] divide el problema de controlar un robot en funciones o componentes aislados que tienen que ser ejecutados en orden, con la salida de un módulo actuando como entrada del siguiente. En cada paso, el robot planea explícitamente el siguiente movimiento, para lo cual necesita almacenar información detallada del entorno y procesarla utilizando algoritmos que pueden tener largos tiempos de cómputo. Por estas razones, esta rígida arquitectura produce robots con grandes necesidades de cómputo. Además, debido a que no se actúa hasta que no se ha acabado de ejecutar el lazo de control, los robots que son diseñados siguiendo esta arquitectura son generalmente lentos, incapaces de reaccionar rápidamente ante su entorno. Estas limitaciones se hacen más patentes conforme aumenta el número de sensores, la información que tiene que procesar el robot y la complejidad de los algoritmos utilizados. La modelización del entorno es un problema importante en este tipo de arquitecturas, que ha sido resuelto tradicionalmente mediante dos métodos:

Sistemas jerárquicos. Descomponen el proceso de control en funciones, de forma que los procesos de bajo nivel proporcionan funciones simples, que son agrupadas sucesivamente en procesos de alto nivel, hasta llevar a cabo el control del robot. Muchas de las arquitecturas desarrolladas según este esquema suelen agrupar la funcionalidad en dos capas: una de acceso y control del hardware y otra en la que se llevan a cabo las labores de planificación y monitorización. La arquitectura **NASREM** (*NASA/NBS Standard Reference Model for Telerobot Control System Architecture*) [6] es un ejemplo clásico de arquitectura de control jerárquica.

Sistemas de pizarra. La centralización de la información en una pizarra [109], que es compartida por todos los sistemas que integran la arquitectura, permite desacoplar los procesos entre sí y minimiza la comunicación entre ellos. Esto es debido a que la pizarra actúa como núcleo común que almacena la información. A pesar de las ventajas de diseño y bajo acoplamiento que aporta este repositorio central de

información, presenta como gran desventaja el que la pizarra es un cuello de botella en la comunicación y que además introduce un nivel de indirección adicional. Además, la naturaleza asíncrona de la comunicación con la pizarra puede introducir un grado extra de complejidad al diseño de determinados sistemas de la arquitectura y puede introducir errores de temporización. A pesar de estos inconvenientes, numerosas arquitecturas utilizan diferentes versiones de pizarra para compartir datos.

A finales de los años 80 se hizo patente que el modelo deliberativo tenía muchos inconvenientes. Entre los puntos más débiles de esta arquitectura destaca que estos sistemas eran incapaces de reaccionar con rapidez, ante una emergencia por ejemplo, lo cuál las hacía inaceptables para controlar determinados sistemas. Además de estos problemas de latencia los sistemas deliberativos suelen asumir que el entorno no cambia entre sucesivas activaciones del módulo de percepción. Si estos cambios llegaban a suceder podían causar grandes problemas en el control. Por todas estas razones se llegó al diseño de las arquitecturas reactivas.

5.2.2 ARQUITECTURA REACTIVA

Las arquitecturas reactivas surgieron como respuesta ante las limitaciones de las arquitecturas deliberativas, sobre todo en tiempo de respuesta y rigidez. En este caso, se fue directamente al extremo opuesto: los sensores están directamente acoplados con los actuadores (ver figura 5.3). El paradigma de control reactivo está basado en los modelos de inteligencia de algunos animales: el movimiento del robot se descompone en una serie de comportamientos que, una vez combinados, dan lugar al comportamiento global, en vez de ser este obtenido como resultado de un proceso deliberativo. Esta forma de obtener la acción de control es muy rápida, pero tiene el inconveniente de que los comportamientos básicos son muy sencillos, por lo que comportamientos más complejos quedan codificados en función de otros más simples, lo cual dificulta su depuración y entendimiento. Además, existe siempre la incertidumbre de si el comportamiento global emergente permitirá finalmente, y tras ser combinada, alcanzar el objetivo seleccionado.

Las arquitecturas reactivas pretenden eliminar también otro de los inconvenientes de las arquitecturas deliberativas: la necesidad de modelar el entorno y compartir este conocimiento entre todos los sistemas del robot: «*The world is its own best model*» [39]. Básicamente, existen dos métodos para calcular el comportamiento global:

Modelo cooperativo: permite utilizar concurrentemente las salidas de muchos comportamientos individuales. Puesto que la salida global del sistema es una media ponderada de los comportamientos activos en cada caso, la respuesta tiende a ser suave. Sin

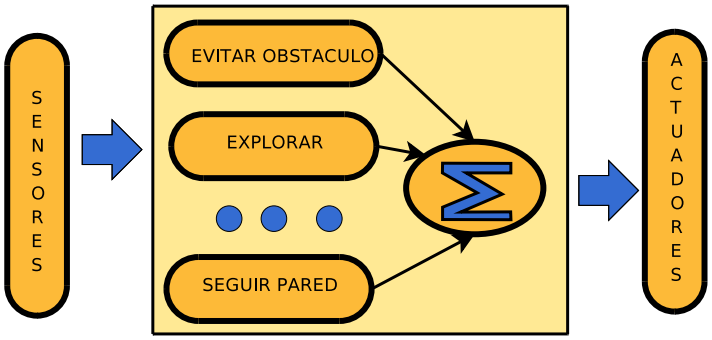


Figura 5.3: Arquitectura de control reactiva

embargo, es propenso a quedarse bloqueado en mínimos locales como consecuencia de los posibles conflictos que pueden crearse entre distintos comportamientos.

Modelo competitivo: la acción global de control depende únicamente del comportamiento elegido en cada caso. Este modelo evita el problema del mínimo local y los conflictos entre comportamientos a costa de sacrificar la suavidad y continuidad de la respuesta del sistema.

Por último, resta por solucionar la selección y coordinación de los comportamientos básicos que están activos en cada momento. Este problema ha dado lugar, entre otras, a una arquitectura reactiva parcialmente estratificada en capas denominada *Subsumption* [39] y a los esquemas de motor (*Motor Schema*) [14]. La primera de ellas propone la agrupación de los comportamientos en distintas capas jerárquicas, que aumentan en nivel de complejidad conforme se asciende en dicha jerarquía. Este método de coordinación está basado en el modelo competitivo de obtención del comportamiento global, ya que las capas inferiores tienen una prioridad mayor que las superiores.

En un trabajo más reciente, Brooks propone **COG** [37]. COG está basado en la idea de aprendizaje, de forma que los comportamientos se van adquiriendo con el tiempo, desde los más simples hasta los más complejos. Los diseñadores de COG sostienen que la interrelación entre los distintos módulos de control, planificación y representación puede ser a veces demasiado compleja para ser programada en un sistema, por lo que puede ser muy beneficioso aplicar técnicas de aprendizaje para modelar dichas interrelaciones.

Sin embargo, la arquitectura propuesta por Brooks [39] era incapaz de manejar objetivos complejos, por lo que rápidamente se propusieron distintas alternativas o mejoras. Entre ellas cabe destacar la propuesta de Arkin [14] *Motor schema*, caracterizada por su base neuro-científica y psicológica. Los comportamientos sencillos son denominados *Motor schema* porque cada uno produce como salida un vector de velocidad que representa la dirección y la velocidad a que debe desplazarse el robot. La acción global de movimiento

es obtenida como resultado de la suma vectorial de todos los vectores velocidad resultado de los comportamientos activos en cada momento. Es decir, en este caso se sigue un modelo cooperativo.

Rápidamente se llegó a la conclusión de que si era posible diseñar un conjunto de comportamientos para llevar a cabo una tarea de control, sería posible diseñar distintos módulos reactivos que llevaran a cabo porciones de la tarea global del robot. Estos módulos serían elegidos dinámicamente según fueran requeridos. Entre todos los mecanismos propuestos para diseñar este tipo de módulos cabe destacar *Reactive Action Packages* (**RAP**) [91]. RAP proporciona una arquitectura para integrar y moderar entre distintos módulos de control reactivo. RAP permite agrupar primitivas reactivas continuas y su ejecución en secuencia o de forma paralela para llevar a cabo una tarea compleja. Para llevar a cabo su misión, el controlador del sistema RAP tiene acceso al estado de las distintas primitivas reactivas, al estado del robot y al del entorno.

Otra arquitectura de control de robots móviles autónomos en la que pueden verse reflejados los aspectos claves mencionados en *Subsumption* [39] es **Saphira**[1] [132]. La arquitectura Saphira está organizada alrededor de un espacio de percepción local (*Local Perceptual Space*, **LPS**), que contiene diversas representaciones del entorno que rodea al robot, tal y como es percibido por los sensores del mismo. El LPS no sólo actúa como un repositorio centralizado de la información sensorial y de los mapas suministrados a priori, sino que también proporciona diversas interpretaciones de la misma, según las necesidades del módulo que la requiere. Un rasgo distintivo respecto de otras arquitecturas es que proporciona una fusión eficiente de distintos comandos de comportamiento gracias a un enfoque de lógica difusa [220]. Saphira es una arquitectura de robots móviles autónomos bastante madura, que ha sido adoptada por varios robots comerciales (Pioneer, Khepera); es extensible, modular y portable a distintas plataformas gracias a su estructura cliente/servidor.

5.2.3 ARQUITECTURA HÍBRIDA

Las arquitecturas de control reactivas se comportan globalmente de forma correcta cuando la tarea a realizar es sencilla, incluso en entornos cambiantes, pero no son capaces de realizar de forma efectiva tareas complejas. Las arquitecturas híbridas fueron el siguiente paso en la evolución natural. Como muestra la figura 5.4, una arquitectura híbrida está formada por controladores reactivos en lazo cerrado , un módulo deliberativo más lento para planificar y activar distintos comportamientos (módulo «modelar» en dicha figura) y un mecanismo secuenciador que conecta ambas capas (módulo «planificar»). De forma genérica, este tipo

[1] http://www.ai.sri.com/~konolige/saphira/

de arquitecturas son conocidas como *arquitecturas en tres capas* [33, 99], y suelen ejecutar cada uno de estos módulos en procesos separados, ya que cada uno tiene necesidades y restricciones temporales distintas.

Controlador. Consiste en uno o varios procesos de ejecución que implementan varios lazos de control que se ejecutan frecuentemente. El controlador aporta, por tanto, el comportamiento reactivo al sistema en forma de una librería de comportamientos, que acoplan sensores y actuadores en ciclos cortos. El conjunto de comportamientos que están activos en cada momento está determinado por otro de los sistemas de la arquitectura: el secuenciador. Ejemplos clásicos de comportamiento son evitación de obstáculos, seguimiento de paredes, atravesar puerta, etc. Lógicamente, los comportamientos embebidos en el controlador tienen que cumplir una serie de características, como (1) tiempo de ejecución constante y lo suficientemente pequeño para realizar un control en lazo cerrado estable; (2) no deben depender, en la medida de lo posible, de su estado anterior, salvo los algoritmos de filtrado; (3) la salida de cada comportamiento debe ser continua, desde el punto de vista matemático.

Secuenciador. Se encarga de seleccionar el conjunto de comportamientos que tienen que estar activos en el controlador en un instante de tiempo y de aportar los parámetros de configuración en caso necesario. La alternancia de conjuntos de comportamientos permite que el robot ejecute tareas complejas. Los secuenciadores más sencillos suelen consistir en máquinas de estado o en conjuntos de reglas. El secuenciador debe ser capaz de reaccionar de forma acorde a todas las situaciones en que se encuentre el robot, ya que puede darse el caso de que los comportamientos elegidos no lleven a cabo el objetivo perseguido en un primer momento. En este caso el secuenciador ha de ser capaz de corregir esta situación, lo que no resulta sencillo. Estos secuenciadores más complejos pueden ser desarrollados en lenguajes específicos de descripción de comportamientos, como RAP [91].

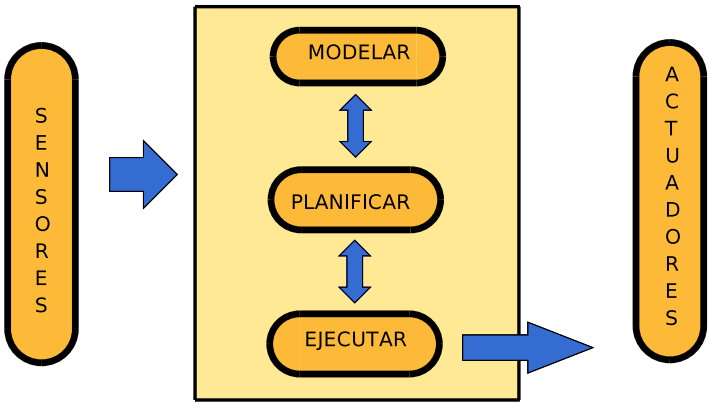


Figura 5.4: Arquitectura de control híbrida

Planificador. Contiene todos los algoritmos con grandes tiempos de ejecución, como algoritmos cúbicos $O(n^3)$ o exponenciales $O(2^n)$, que son normalmente ejecutados por algoritmos de visión, planificación de movimiento, localización, etc. Este módulo contiene y calcula el estado global del robot. El planificador calcula los objetivos a largo plazo, que son traducidos por el secuenciador en un conjunto de objetivos a medio-corto plazo, con los que configura los comportamientos que se activan en el controlador. La clave de este sistema es que se pueden haber activado distintos comportamientos durante el tiempo que transcurre entre dos invocaciones sucesivas del algoritmo deliberativo sin que afecte ni al control del robot ni a la obtención del objetivo global.

CIRCA (*Cooperative Intelligent Real-time Control Architecture*) [158] es otro ejemplo de arquitectura híbrida cooperativa en la que las necesidades de tiempo-real son controladas por la capa deliberativa. Una realimentación desde la capa reactiva permite que el subsistema de inteligencia artificial razone sobre la lógica y las restricciones temporales. Así se pueden tomar las decisiones adecuadas para evitar condiciones operativas incorrectas, adaptando el conjunto de comportamientos, cambiando los parámetros de tiempo-real, o dando una escala distinta a los objetivos a nivel de tarea, e.g. disminuyendo la velocidad de navegación del robot. Como las capas reactivas y deliberativas están desacopladas, la ejecución de tiempo-real está siempre garantizada en la capa reactiva, mientras que la flexibilidad y la robustez a nivel de tarea se demanda a la capa deliberativa. Esta idea persigue el diseño de un sistema que pueda razonar acerca de las necesidades de tiempo-real y sus propias limitaciones computacionales, con objeto de poder adaptar la computación a los eventos entrantes.

Por último, se va a mencionar la arquitectura de tareas de control (*Task Control Architecture*, **TCA**)[2] [199], que aunque en desuso, dio lugar a la aparición del mecanismo de comunicación entre procesos **IPC** (*Inter-Process Communication*). TCA se diferencia de otras arquitecturas en que, según su página web, se parece más a un sistema operativo para diseñar sistemas robóticos. Como tal, TCA proporciona mecanismos para la comunicación entre módulos, la composición de comportamientos, la planificación de tareas y asignación de los recursos del robot. TCA es un framework de control genérico, que integra y coordina los principales actividades que realiza un robot móvil: percepción, planificación y control en tiempo-real.

[2] `http://www.cs.cmu.edu/~TCA/`

5.3 Últimas Tendencias

A LOS robots actuales se les exige que se desenvuelvan cada vez en más situaciones y que realicen de forma eficiente tareas cada vez más complejas. Hay una creciente demanda de que tales sistemas realicen no sólo una tarea, sino una serie de operaciones distintas en entornos dinámicos, no estructurados y que además lo hagan durante largos periodos de tiempo sin interrupción. Otras características, como fácil reconfiguración y ampliación son también ampliamente demandadas. Por todo ello, existe un creciente interés en el desarrollo de arquitecturas modulares y frameworks de componentes, que fomentan la creación de sistemas abiertos, fácilmente modificables y ampliables. Esta sección presenta, sin ningún orden en particular, algunas de las arquitecturas, librerías y frameworks orientados a objetos y de componentes más utilizados a la hora de desarrollar robots. En [15, 64] se pueden encontrar más detalles sobre la mayoría de las arquitecturas que aquí se describen.

Coupled Layer Architecture for Robotic Autonomy (**CLARAty**) [159, 160, 161] es un proyecto que comenzó en 1.999 y en el que colaboran el *Jet Propulsion Laboratory* (**JPL**) de la NASA, Ames Research Center y la Universidad Carnegie Mellon. CLARAty nació como iniciativa de la NASA para obtener un *framework* para diseñar la arquitectura de control software de los distintos vehículos de exploración espacial y que fuera capaz de reutilizar, sin mucho esfuerzo, algoritmos previamente definidos. CLARAty define, como muestra la figura 5.5, una arquitectura dividida en dos capas: la capa funcional y la capa deliberativa. La capa funcional abstrae el hardware, proporciona la funcionalidad básica para realizar el control y está diseñada para ser reutilizable y extensible. La capa deliberativa se encarga del control de alto nivel del robot: estado del sistema, seguimiento del objetivo, gestión de los recursos, etc. Cada capa está, además, programada siguiendo el paradigma más adecuado para lograr sus objetivos. De esta forma la capa funcional utiliza un modelo imperativo, mientras que la capa deliberativa utiliza un modelo declarativo.

El sistema operativo para el control de robots autónomos (*Operating System for the Control of Autonomous Robots*, **OSCAR**) [31] ha sido diseñado por la universidad de München para desarrollar sus robots de interiores. A pesar de su nombre, OSCAR es un framework de componentes que utiliza una base de datos centralizada para representar toda la información relativa al entorno (tanto los datos «en crudo» como representaciones de alto nivel, i.e. mapas topológicos y geométricos). Los componentes OSCAR pueden ser distribuidos entre distintos nodos gracias a la utilización de CORBA, lo que permite además utilizar distintas plataformas y lenguajes de programación.

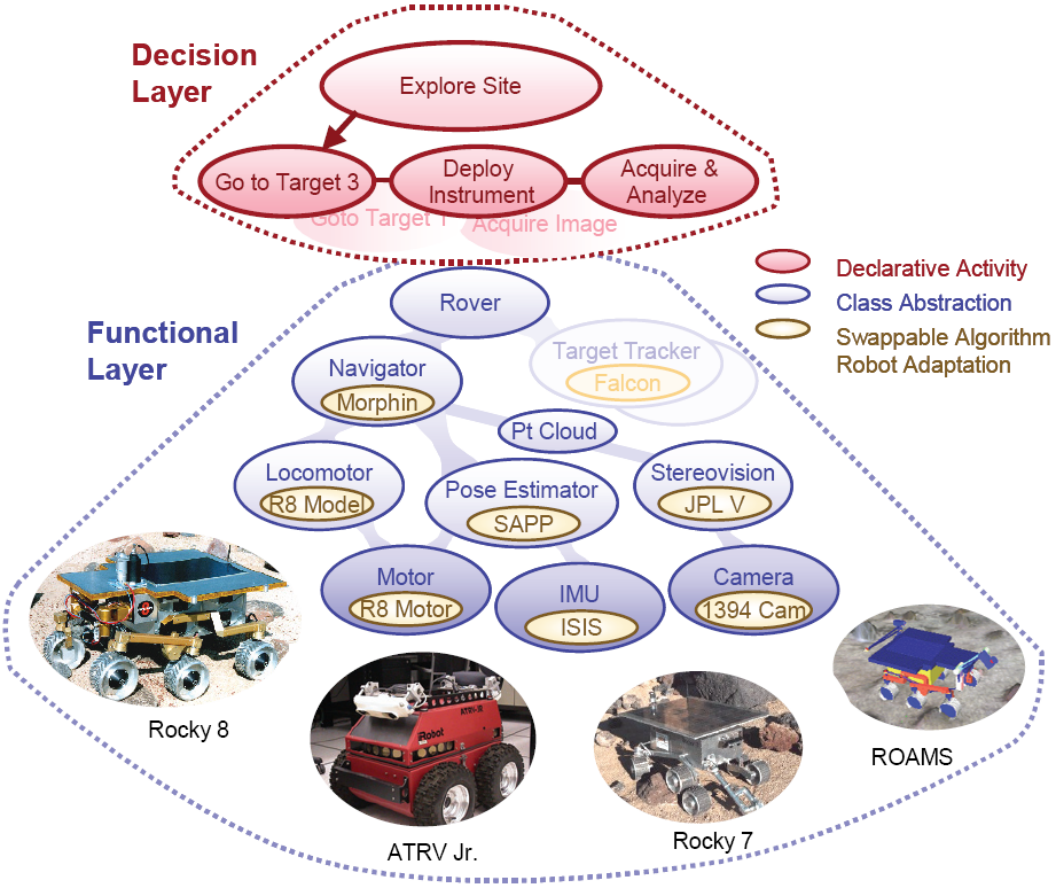


Figura 5.5: La arquitectura de control CLARAty (extraído de [159])

El proyecto **OROCOS**[3] (*Open RObot COntrol System*) [42] nació en el año 2.001 con el objetivo de proporcionar un marco libre para desarrollar software de control de robots. Diseñado, principalmente, como colaboración entre las universidades de Leuven (Bélgica) y LAAS (Francia), OROCOS proporciona un conjunto de librerías y utilidades escritas en C++ para realizar el control de robots y utiliza el sistema operativo de libre distribución GNU/Linux. OROCOS está integrado como una línea de interés en la red europea EURON (*EUropean RObotics Networ*k), que integra los más importantes centros de investigación y universidades europeas en este área. El objetivo principal de OROCOS no es definir o proporcionar una arquitectura de control, sino proporcionar un *framework* de componentes, con interfaces bien definidas, que puedan combinarse entre sí para conseguir la arquitectura más adecuada para cada aplicación. Este framework separa la estructura del control (por ejemplo, la subdivisión en distintos hilos de control, comunicación entre procesos, manejo de eventos, distribución entre nodos de procesamiento, funcionalidad requerida del SO de tiempo-

[3] `http://www.orocos.org/`

real, etc) de su funcionalidad (por ejemplo, interpolación del movimiento, algoritmos de control, procesamiento de sensores, etc). Proporciona componentes independientes de la arquitectura que se pueden agrupar para formar aplicaciones de control de movimiento.

OROCOS nace con la idea de construir un robot genérico, por lo menos a nivel de software, pero el hecho de ser tan genérico puede ser un arma de doble filo, puede solucionar muchos problemas, pero precisamente por ser muchos pueden quedar resueltos a medias. Otro inconveniente importante y que repercute en su aplicación en dominios amplios es la granularidad de los componentes: adoptan una micro-arquitectura para cada tipo de componente, con una granularidad definida; el hecho de que sea obligatorio combinar dichos componentes para generar una arquitectura más amplia (del sistema global) hace que el rendimiento obtenido pueda no ser precisamente el deseado, aunque bien es cierto, que se debe adoptar el compromiso de llegar a perder cierto rendimiento a costa de aumentar la posibilidad de reutilización y reconfiguración.

Player/Stage[4] [63, 101] es un framework de control de robots creado originariamente para el robot Pioneer 2, que ha evolucionado hasta convertirse en el estándar *de facto*, utilizado en muchas universidades y centros de investigación y con soporte para gran número de plataformas. Player se ejecuta en el robot como un servidor, que proporciona una serie de interfaces de control y monitorización a través de conexiones TCP. De esta forma, aunque Player esté escrito en C++, permite que el programa de control del robot pueda ejecutarse en cualquier plataforma y sobre cualquier sistema operativo. Player además no fuerza ninguna arquitectura, sino que ofrece únicamente las interfaces necesarias para llevar a cabo el control del robot: lazos de control y acceso a los sensores. La elección de la arquitectura se deja al usuario (cliente). Stage y Gazebo son dos simuladores, desarrollados dentro del mismo proyecto, que complementan a Player. Stage proporciona un simulador multi-robot en dos dimensiones de entornos interiores mientras que Gazebo es un simulador multi-robot en tres dimensiones para espacios abiertos. Ambos son capaces de simular el movimiento y las lecturas de sensores de los distintos robots que estén simulando.

El *Carnegie Mellon Robot Navigation Toolkit*[5] (**CARMEN**) [156] fue diseñado para proporcionar a los desarrolladores de robots un conjunto consistente de interfaces y de primitivas de control que sean fácilmente intercambiables entre distintos grupos y que permitan desarrollar prototipos rápidamente. CARMEN fuerza una arquitectura híbrida en tres capas [33]: la capa base abstrae el hardware del robot y proporciona las

[4] http://playerstage.sourceforge.net/
[5] http://carmen.sourceforge.net/

primitivas básicas de control en bucle cerrado y de obtención de las medidas sensoriales; la capa intermedia proporciona primitivas de navegación como localización, seguimiento de obstáculos, planificación de movimientos, etc; la capa superior está pensada para que los usuarios de CARMEN desarrollen sus programas, que utilizan las primitivas de la segunda capa. CARMEN utiliza el mecanismo de comunicación entre procesos IPC para facilitar la modularidad de los diseños, su distribución y ampliación, y por tanto, está limitado a sistemas operativos que soporten este tipo de mecanismo de comunicación (generalmente Unix).

Orca[6] [36, 140] es un framework abierto basado en componentes para diseñar aplicaciones de control de robots. Orca proporciona la infraestructura de soporte que permite diseñar y ensamblar componentes desarrollados por personas o grupos independientes, y permite diseñar desde robots con distintos grados de funcionalidad hasta enjambres de robots (*robot swarms*). Para ello define una serie interfaces de uso común que pueden ser reutilizadas entre distintos componentes, utiliza el middleware de comunicación *Internet Communication Engine*[7] (**Ice**) [111] para permitir una distribución transparente, multi-lenguaje y multi-plataforma y mantiene un repositorio de componentes ya creados para facilitar la reutilización de código. Como sucede con el resto de frameworks de componentes, Orca no impone una arquitectura específica, sino que permite que construir cualquier tipo de arquitectura (jerárquica, de pizarra, híbrida, etc) utilizando los componentes adecuados.

Modular Controller Architecture-2[8] (**MCA-2**) [192], desarrollada por la Universidad de Karlsruhe (Alemania), surge por la necesidad del instituto de robótica de dicha universidad de disponer de una plataforma software común para todos sus desarrollos, basados en el sistema operativo RT-Linux [21] y C++. MCA-2 es un framework con características de tiempo-real. En un robot diseñado siguiendo MCA-2 existen dos fuentes de datos que circulan entre los distintos módulos del sistema, dependiendo de si circulan de forma ascendente o descendente en la jerarquía MCA-2: los datos de los sensores y los de control. Según se organicen los módulos y los flujos de datos entre ellos, se puede obtener la arquitectura concreta para cada sistema.

Remote Objects Control Interface (**ROCI**) [55, 65] está siendo desarrollado por el laboratorio GRASP, en la Universidad de Pensilvania (Estados Unidos). El objetivo de ROCI es desarrollar un framework de programación de robots autónomos móviles que puedan ser desplegados en entornos no estructurados y desconocidos. Estos robots realizan tareas de

[6] http://orca-robotics.sourceforge.net/
[7] http://www.zeroc.com/
[8] http://www.mca2.org/

reconocimiento, vigilancia, adquisición de blancos, detección de explosivos, suministración de información, etc. ROCI proporciona una arquitectura que descansa en un sistema de tipos reflexivo que mantiene meta-datos del resto de objetos distribuidos. ROCI se ha diseñado con el principio de sencillez en mente, con módulos que exhiben interfaces muy claras y sin apenas solapamiento de funcionalidad.

La mayoría de las soluciones para desarrollar robots presentadas están basadas en frameworks de componentes, ya que son los que mayores tasas de reutilización y mayor facilidad de uso tienen. Así pues, ¿por qué desarrollar uno nuevo? ACRoSeT fue desarrollado para suplir la falta de flexibilidad que tienen estos frameworks en cuanto a la plataforma de ejecución. La mayoría de estos frameworks no contemplan la posibilidad de que los programas de control puedan ser utilizados en distintas plataformas de ejecución o con otro lenguaje de programación distintos de C++; este es el caso de MCA-2, OROCOS y Player. El resto de frameworks consiguen una mayor independencia gracias a la utilización de algún *middleware* de comunicaciones (CORBA, ACE o Ice), pero aún así siguen centrando la atención del desarrollador en los detalles de implementación en vez de en la funcionalidad y en la estructura del robot.

Es decir, sería deseable ser capaces de desarrollar componentes que pudieran ser abstractos, en el sentido de que no dependieran de ninguna plataforma ni lenguaje específico de programación y que plasmaran la funcionalidad del componente. De esta forma sería posible desarrollar la misma aplicación en distintas plataformas o lenguajes o reutilizar los mismos componentes en distintas aplicaciones. Estas fueron algunas de las limitaciones que encontró el grupo DSIE a lo hora de diseñar nuevas aplicaciones robóticas y que, como se comentó en el apartado 1.2.1, justificaron el desarrollo de ACRoSeT.

5.4 Breve Descripción de ACRoSeT

ACROSET (*Arquitectura de Referencia para Robots de Servicio Teleoperados*) es un marco arquitectónico para el diseño del software de control de robots de servicio teleoperados, obtenido como resultado de la tesis del Dr. Ortiz [181]. ACRoSeT está basado en componentes que son independientes de la infraestructura de ejecución utilizada. Es decir, ACRoSeT representa un marco de componentes abstractos, que sirve para guiar el diseño del software de un robot. Para tal fin, define los principales subsistemas que deben estar presentes en cualquier sistema robótico, sus responsabilidades y las relaciones entre ellos. ACRoSeT proporciona un marco de componentes *abstractos*, que puede ser implementado de diversas formas (integrando distintas soluciones software/hardware e incluso COTS), que puede ejecutarse sobre distintas plataformas, para desarrollar robots

con comportamientos muy diferentes. La figura 5.6 muestra un diagrama conceptual de los principales subsistemas que integran ACRoSeT, mientras que en la figura 5.7 se muestra como un diagrama de componentes UML. Para más información sobre ACRoSeT y los distintos robots en que ha sido utilizada consulte [89, 178, 179, 224, 225]

Según Shaw [198], la arquitectura de un sistema contempla la descripción de los elementos que lo forman, la interacción entre dichos elementos, la descripción de los patrones que guían su composición y las restricciones que imponen a la misma. ACRoSeT se diseñó teniendo en cuenta estas guías generales y las peculiaridades del dominio de la robótica de servicio. Entre los principios que gobernaron su diseño destacan los siguientes:

- El marco arquitectónico no debe imponer ninguna arquitectura concreta, sino que tiene que permitir definir arquitecturas que se ajusten a las restricciones particulares de la aplicación. Como consecuencia de este primer principio, se deduce que debe ser posible describir con ACRoSeT las principales arquitecturas robóticas (descritas al principio de la presente sección), como son la arquitectura deliberativa y la reactiva.
- Debe ser posible reutilizar componentes en sistemas con diferentes arquitecturas. Esto implica que se tiene que separar claramente la funcionalidad de un componente de sus patrones de interacción.
- Debe ser posible especificar y verificar el comportamiento temporal de los componentes, ya que el correcto funcionamiento de estos sistemas impone restricciones de tiempo-real al software de control.
- Los componentes pueden ser, finalmente, software o hardware, siendo más que aconsejable el uso de COTS.
- Debe ser posible añadir «inteligencia» a estos sistemas o la capacidad de interactuar con otros «sistemas inteligentes».

Estos requisitos impuestos al diseño de ACRoSeT representan las características de flexibilidad y extensibilidad que se necesitan para desarrollar un sistema robótico. Sin embargo, ni el paradigma de la orientación a objetos ni la modularidad conseguida con la programación estructurada se muestran suficientes para alcanzarlos [186], ya que ni los objetos ni las librerías son realmente componentes. En consecuencia, en lugar de proponer, como hacen otros *frameworks* robóticos, una librería de componentes que ofrecen la funcionalidad básica para estos sistemas, ligada al lenguaje de programación y a la plataforma, ACRoSeT propone componentes abstractos genéricos. Estos componentes son conformes a la descripción que realiza ROOM (*Real-time Object-Oriented Modelling*) [195] y que posteriormente actualiza Hofmeister en [115]: los componentes contienen la

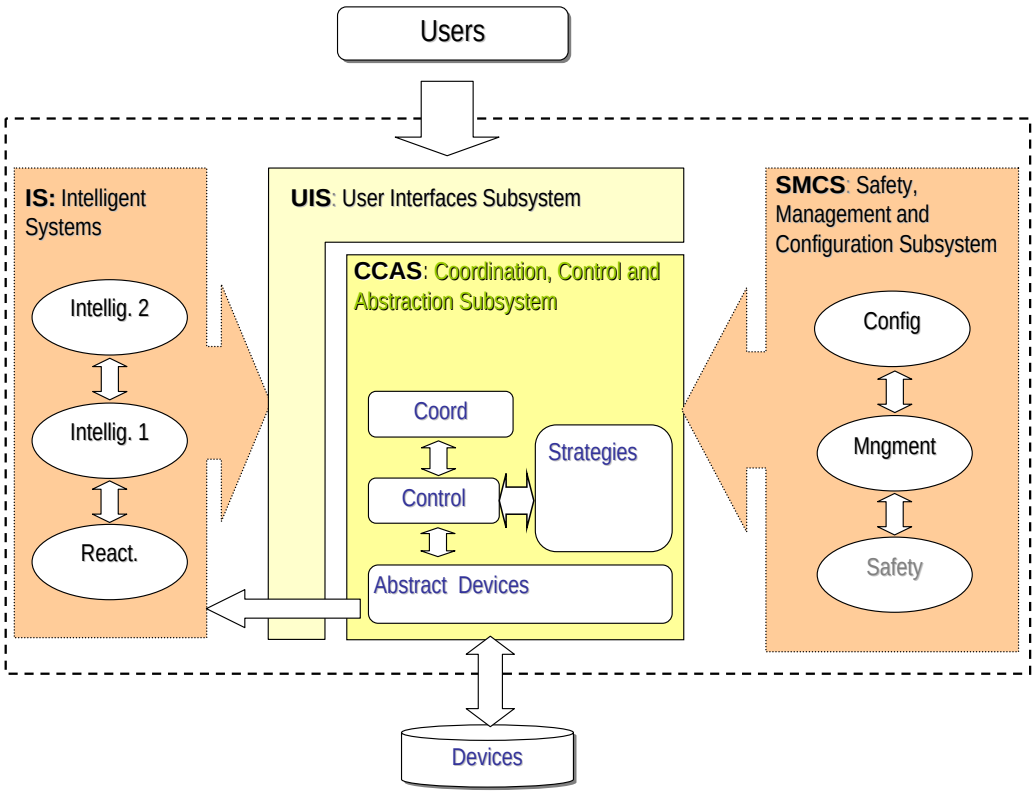


Figura 5.6: Diagrama conceptual del marco arquitectónico ACRoSeT

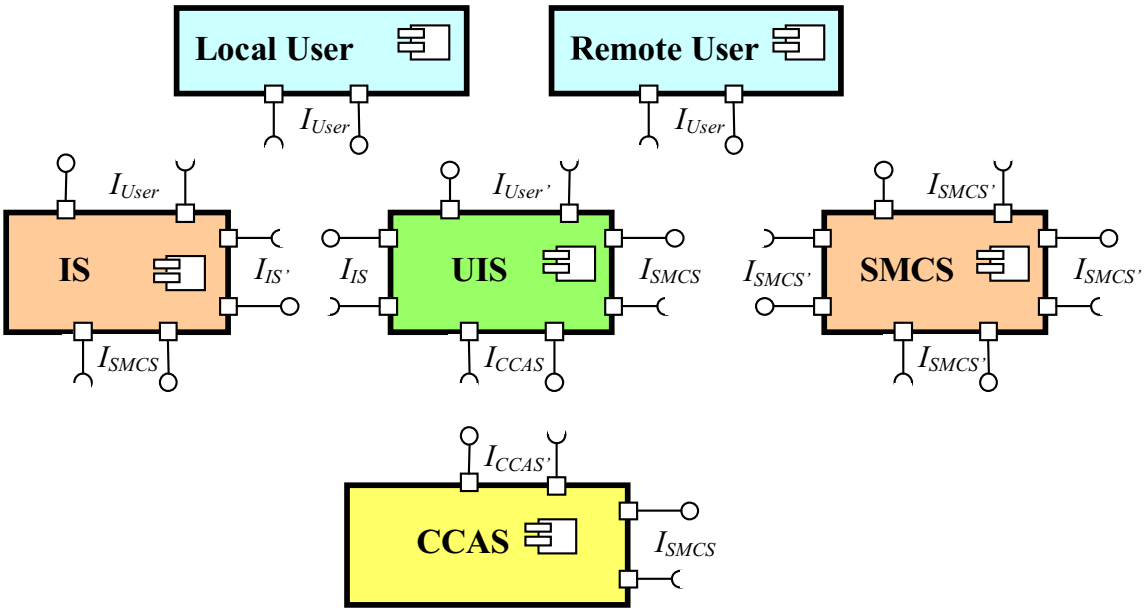


Figura 5.7: Diagrama de componentes del marco arquitectónico ACRoSeT

funcionalidad, que es expuesta al entorno en forma de interfaces (independientes del lenguaje de programación) agrupadas en puertos, y que se comunican entre sí mediante conectores. Separando estos tres conceptos (realización de la funcionalidad, ofrecimiento de la interfaz a dicha funcionalidad y distribución) se consigue la máxima flexibilidad a la hora de definir y reutilizar componentes entre distintos sistemas.

ACRoSeT depende de los conceptos de componente, puerto y conector según los define UML [177]: un componente es una unidad auto-contenida que encapsula el estado y comportamiento de un conjunto de *Classifier* contenidos en el componente. Un componente especifica un contrato formal, en forma de interfaces, entre los servicios que proporciona a sus clientes y los que requiere de ellos o del sistema para funcionar correctamente. Un componente es una unidad reemplazable que puede ser sustituida en tiempo de ejecución o de compilación por otro componente que ofrezca una funcionalidad equivalente, basada en la compatibilidad entre sus interfaces.

Como se ha mencionado al comienzo de este apartado, ACRoSeT define no sólo los componentes principales que aparecen en el diseño del software de control de un robot de servicio, sino que también propone distintos subsistemas que agrupan la funcionalidad que debe ofrecer el sistema, y las comunicaciones entre ellos. Estos subsistemas, tal y como muestra la figura 5.6, son los siguientes:

Coordinación, control y abstracción (CCAS). El CCAS abstrae y encapsula la funcionalidad de los dispositivos físicos del robot. A este nivel se definen una serie de componentes conceptuales, de forma que se logra independencia de la plataforma de ejecución. Estos componentes conceptuales pueden ser reutilizados entre aplicaciones y finalmente traducidos a software o hardware, dependiendo de la aplicación. Este primer subsistemas está jerárquicamente dividido en los siguientes componentes:

- **Componentes atómicos.** Son los más sencillos de todos y modelan las características de los elementos más básicos del hardware de un robot, como son sensores y actuadores.
- **Controlador simple (SC).** Modela el control sobre un actuador con uno o varios sensores, y representan el componente con comportamiento más sencillo de toda la arquitectura. El elemento más importante de estos componentes es el algoritmo de control utilizado, que puede ser intercambiado gracias a que utiliza el patrón *Strategy*[9] [96].
- **Controlador de mecanismo (MC).** Modela el control y, lo que es más importante, la coordinación entre los diferentes actuadores que forman un mecanismo

[9]El patrón *Strategy* permite agrupar un conjunto de algoritmos de forma transparente para el usuario, quien puede elegir aquel que le conviene e intercambiarlo según sus necesidades.

(conjunto de articulaciones que requieren una coordinación). Un MC está formado por una agrupación de SC y un coordinador, cuyo algoritmo de coordinación puede ser cambiado (de nuevo, gracias al patrón *Strategy*). Aunque un MC parece únicamente un agregado de SCs, puede llegar a convertirse en un componente cuando la arquitectura se instancia para una plataforma en concreto. Este caso puede darse cuando, por ejemplo, se utiliza una tarjeta controladora de varios ejes de movimiento.

- **Controlador de robot (RC).** Es el componente de mayor nivel y modela el control sobre un robot. Un RC está formado por una agregación tanto de MCs como de SCs y por un algoritmo de coordinación entre ellos. El RC recibe, como elemento que encapsula toda la funcionalidad del robot, todas las órdenes de movimiento del robot y las distribuye entre sus diferentes componentes.

Interacción con el usuario (UIS). Está encargado de interpretar, combinar y arbitrar entre los diferentes comandos (que serán ejecutados finalmente por el CCAS) que pueden llegar al robot procedentes de distintas fuentes o usuarios del sistema. Los usuarios del sistema no tienen que ser únicamente humanos (estación de teleoperación o mando de control), sino que pueden ser otros sistemas, como el que se describe a continuación.

Inteligencia (IS). Este subsistema, que actúa como otro usuario más de la funcionalidad del robot (a la cual accede a través del UIS), es el encargado de contener los distintos algoritmos que añaden un cierto comportamiento autónomo o deliberativo al robot. De esta forma es posible hacer crecer el comportamiento «inteligente» del robot sin necesidad de modificar ningún otro subsistema.

Seguridad, mantenimiento y configuración (SMCS). Este gran subsistema se encarga de agrupar el resto de la funcionalidad que debe exhibir cualquier robot. Entre los objetivos de este subsistema se encuentra la gestión de la configuración del sistema, tanto durante el arranque de la misma como en tiempo de ejecución, de forma que sea posible su adaptación a distintas circunstancias. El SMCS contempla también la gestión del sistema de monitorización del correcto funcionamiento tanto del software del resto de subsistemas como del hardware del robot. Es, por tanto, el subsistema de mayor nivel de todos. Aunque las reacciones más inmediatas ante fallos se realizan localmente el los componentes del CCAS, es tarea del SMCS la coordinación global y la ejecución de las modificaciones pertinentes que permitan que el robot se adapte a las nuevas circunstancias.

Aunque la capacidad y las posibilidades que ofrece ACRoSeT para describir la arquitectura de control de un robot ha sido muy valiosa para el grupo, la traducción de sus componentes abstractos a componentes concretos, dependientes de la plataforma, ha

sido una tarea manual, difícil y propensa a errores. Así pues, es necesario dotar a ACRoSeT de un mecanismo que haga posible traducir, de forma más o menos automatizada, sus componentes abstractos en otros concretos que puedan ser ejecutados. Estos componentes concretos pueden ser incluso componentes de alguno de los frameworks descritos en la sección anterior, en caso de que la aplicación pudiera ser realizada en la plataforma que fuerza el framework. En este sentido el desarrollo basado en modelos (MDE) (ver sección 4.2) se perfila como la herramienta ideal para llevar a cabo esta tarea. Además de esta dificultad de implementación, los conceptos básicos sobre los que se asienta ACRoSeT adolecen de una definición precisa y de reglas de implementación concretas que permitan traducir los componentes abstractos a componentes concretos.

5.5 CONCLUSIONES Y APORTACIONES A LA TESIS

EN ESTE capítulo se ha realizado un recorrido histórico por las principales arquitecturas de control de robots que se han diseñado en los cincuenta años de existencia de esta disciplina. Aunque esta Tesis Doctoral no propone ninguna nueva arquitectura ni la mejora de ninguna de las existentes, el diseño de componentes de ACRoSeT permite el desarrollo de cualquier tipo de arquitectura, por lo es necesario conocer las ventajas e inconvenientes de las arquitecturas clásicas a la hora de diseñar una aplicación de control basada en componentes para controlar un robot móvil.

Además de las arquitecturas de control clásicas, se han expuesto las últimas tendencias para desarrollar el software de control de robots: los frameworks de componentes. Estos frameworks embeben todo el conocimiento arquitectónico de sus creadores y proporcionan una serie de componentes con la funcionalidad típica (e incluso avanzada) para desarrollar una aplicación en el dominio de la robótica móvil: algoritmos de localización, generación de mapas, generadores de trajectorias, controladores de todo tipo, etc. Por último, también se ha descrito de forma escueta pero concisa las principales características que diferencian ACRoSeT de cualquier otra arquitectura de control de robots: su independencia de la plataforma final de ejecución.

Las aportaciones de este capítulo a esta Tesis Doctoral son más bien escasas. Esto es debido a que, como ya se ha mencionado, no se está desarrollando una nueva arquitectura de control, sino más bien proporcionando el soporte necesario para utilizar un trabajo anterior del grupo de investigación en el que se realizó este esfuerzo. Sin embargo, y además del aporte que supone siempre actualizar el estado de la técnica de cualquier disciplina, sí que es bastante importante el hecho de contrastar que la mayoría de soluciones que actualmente están proponiendo las entidades más importantes dentro del mundo

de la robótica móvil (el proyecto europeo EURON, el toolkit CARMEN de la Carnegie Mellon o la arquitectura CLARAty de la NASA) apuestan por la utilización de frameworks de componentes para desarrollar la aplicación de control de robots móviles. Este hecho respalda el enfoque adoptado por ACRoSeT, que es continuado en esta Tesis Doctoral.

Además, la calidad y fiabilidad del software generado puede verse ampliamente incrementada si, finalmente, se utilizara alguno de los frameworks de control de robots que se han comentado en este capítulo.

PARTE II
DESARROLLO DE V^{3}STUDIO

Sabes que has conseguido la perfección en el diseño no cuando ya no tienes nada más que añadir, sino cuando no tienes nada más que quitar.

ANTOINE DE SAINT-EXUPÉRY

CAPÍTULO 6

DISEÑO DE V^{3}STUDIO

DESARROLLAR una aplicación para generar aplicaciones dentro de un dominio dado es una labor que requiere un gran esfuerzo, un amplio conocimiento del dominio y la existencia de herramientas de soporte. El desarrollo dirigido por modelos (MDE) se perfila como la tecnología perfecta para completar el diseño de ACRoSeT y permitir que este alcance todo su potencial. En este capítulo se describe la primera parte de este esfuerzo: la definición de un meta-modelo para describir las características de los sistemas basados en componentes, de los que ACRoSeT hace uso. Este meta-modelo se denomina V^{3}Studio.

V^{3}Studio nace con dos premisas básicas: la sencillez de diseño y utilización. Se pretende desarrollar un meta-modelo que sea lo suficientemente sencillo para que sea fácil de utilizar, en el que todos los elementos estén claramente definidos y diferenciados, que no disponga de formas alternativas para modelar la misma realidad y que mantenga, en la medida de lo posible, la capacidad expresiva y de modelado suficiente para desarrollar el esqueleto completo de un sistema del dominio. V^{3}Studio, por tanto, no rivaliza con UML, ya que este es un lenguaje de propósito general que puede ser utilizado en cualquier dominio. Sin embargo, V^{3}Studio sí que adopta y adapta distintas partes del mismo para facilitar su uso a aquellos desarrolladores que ya estén familiarizados con UML.

6.1 EL ENTORNO DE DESARROLLO DE V3STUDIO

ECLIPSE[1] es el entorno de desarrollo elegido para realizar este trabajo. Eclipse es un entorno de libre distribución, ampliable y configurable gracias a un diseño modular, que permite la adición de extensiones mediante el uso de plug-ins. Aunque está vinculado a Java, existen numerosos plug-ins que permiten desarrollar software en otros lenguajes de programación, crear páginas web, aplicaciones J2EE, aplicaciones gráficas, control de versiones y un largo etcétera. Eclipse es actualmente el único entorno de desarrollo gratuito que proporciona el soporte necesario para realizar un desarrollo basado en modelos y el más extendido en esta comunidad. A continuación se describen brevemente los principales plug-ins de Eclipse que se han utilizado para desarrollar la presente tesis:

Eclipse Modelling Framework (EMF). **EMF**[2] [43] es el plug-in principal del soporte MDE de Eclipse, ya que contiene la implementación de parte del estándar MOF [167] (concretamente de la parte básica Essential-MOF (**EMOF**)) de el OMG. El proyecto EMF contiene otros plug-ins que permiten especificar y realizar consultas sobre los elementos de un modelo EMF y su contenido, así como para verificar la integridad de los modelos realizados, i.e. si el modelo es conforme a su meta-modelo. La figura 6.1 muestra el meta-modelo de EMF para facilitar su consulta durante la exposición del desarrollo de V3Studio. A continuación se van a describir brevemente los principales conceptos de EMF para facilitar la comprensión de V3Studio. Estos conceptos, que son los que se utilizan para realizar cualquier meta-modelo, son *EClass, EReference* y *EAttribute*:

EClass: cada EClass representa un concepto del dominio que se está modelando. Como se observa en la figura, las EClass contienen EReference y EAttribute para modelar sus características.

EAttribute: representa una propiedad de una EClass. Esta propiedad es de uno de los tipos primitivos definidos por EMF (EInteger, EString, etc) o de un tipo enumerado definido por el usuario.

EReference: modela las relaciones entre los conceptos (EClass) del dominio. La EReference permite especificar la multiplicidad, el rol y la navegabilidad de la relación. Las EReference controlan, mediante el atributo de composición, la forma en que se guardan en fichero los datos relativos a los modelos creados con el meta-modelo.

[1] http://www.eclipse.org
[2] http://www.eclipse.org/emf/

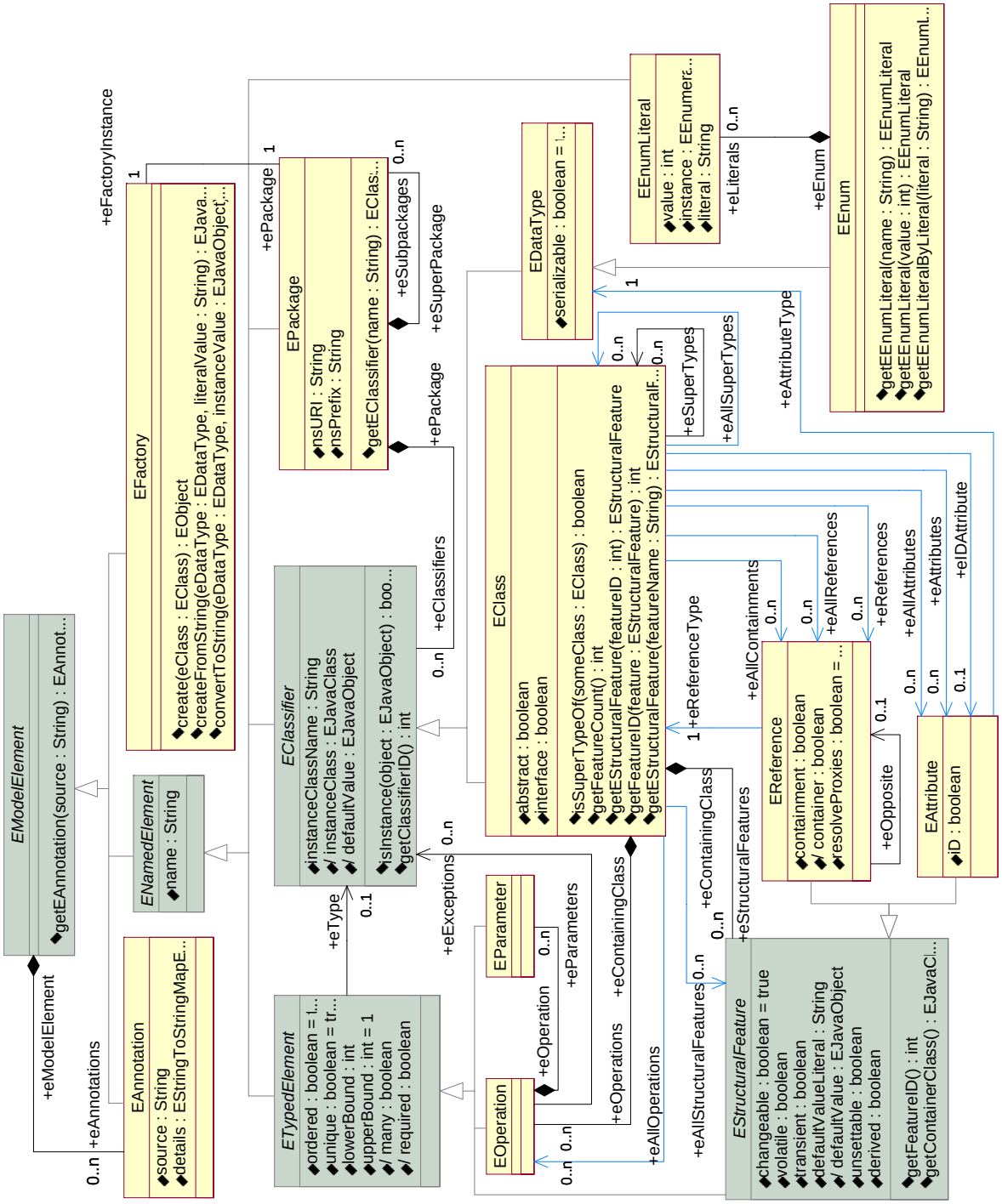


Figura 6.1: Meta-modelo de EMF (Essential MOF)

UML2, OCL y UML2Tools. El primer plug-in contiene una implementación completa de la especificación de la versión 2.x del estándar UML [170] para el entorno Eclipse. Mediante el uso de este plug-in se pueden crear, modificar y recorrer modelos UML. El segundo de ellos contiene un intérprete de restricciones OCL [174] que permite incorporarlas al resto de herramientas MDE, aunque su integración en el entorno todavía no es completa. Por último, el UML2Tools aporta una serie de herramientas que se han usado para mostrar de forma gráfica los diversos diagramas que se generan. Todos estos plug-ins forman parte del *Model Development Tools (MDT)*[3].

ATLAS Transformation Language (ATL) [124]. **ATL**[4] es un lenguaje de transformación de modelos mixto (declarativo/imperativo), que está basado en el lenguaje OCL. ATL soporta transformaciones con múltiples modelos de entrada y múltiples modelos de salida. Una transformación entre modelos descrita en ATL consta de un conjunto de «reglas» que especifican cómo se transforman los elementos del meta-modelo de origen al meta-modelo de destino y de un conjunto de funciones auxiliares. Además, ATL soporta todas las operaciones sobre colecciones definidas en OCL y la navegación por el modelo origen. ATL no es una implementación de QVT [168], sino que fue la respuesta del grupo de investigación ATLAS del INRIA al *«Request for Proposals»* realizado por el OMG para definir el estándar QVT.

MofScript. MofScript[5] es un lenguaje de transformación de modelo a texto mitad declarativo mitad imperativo, desarrollado en el contexto del proyecto Modelplex[6] y remitido a el OMG en respuesta al *«Request for Proposals»* para definir un estándar que cubra este tipo de transformaciones. MofScript admite diversos modelos como entrada al proceso y puede producir múltiples ficheros de texto como salida. Al igual que ATL, MofScript utiliza parte de la sintaxis de OCL, proporciona un amplio soporte a las operaciones sobre colecciones, permite navegar el modelo de entrada y organiza la transformación en una serie de reglas que describen cómo se traducen los elementos del modelo de entrada a texto.

El desarrollo del modelo de componentes V3Studio que se presenta en este capítulo se basa principalmente en el plug-in EMF, ya que se desarrolla utilizando los conceptos que en él se definen, aunque también se utilizan otros plug-ins adicionales, como el de OCL. El capítulo 7 describe la transformación M2M desarrollada para transformar el modelo V3Studio del software de control del robot a un modelo UML. Para llevar a cabo esta transformación se han utilizado los plug-ins ATL y UML2; los diagramas gráficos que se muestran en dicho capítulo hacen uso del plug-in UML2Tools. Por último, el capítulo 8

[3] http://www.eclipse.org/mdt/
[4] http://www.eclipse.org/m2m/atl/
[5] http://www.eclipse.org/gmt/mofscript/
[6] http://www.modelplex-ist.org/

describe la transformación M2T que se ha desarrollado, utilizando el plug-in MofScript, para transformar el modelo UML obtenido en la transformación anterior al lenguaje Ada 2005 [1].

6.2 El Modelo de Componentes V^{3}Studio

Puesto que el diseño del software de control de un robot con ACRoSeT se basa en el diseño y ensamblaje de componentes, de acuerdo a las reglas de composición y a la especificación del mismo, el primer paso consiste en definir perfectamente los elementos básicos que sustentan el enfoque CBD, esto es, *componente, puerto* y *conector*. Aunque en un principio se desarrolló este meta-modelo de componentes para ser utilizado únicamente dentro del marco definido por ACRoSeT, pronto se hizo patente que los componentes que se definen con V^{3}Studio podían ser utilizados en otros dominios y áreas de investigación en las que está involucrado el grupo DSIE, como son las redes de sensores. Así pues se decidió generalizar el diseño para ampliar su campo de aplicación.

Esta sección presenta las razones principales que justifican el desarrollo de V^{3}Studio a partir de un meta-modelo inspirado en UML, en lugar de haber realizado un *profile* de UML o haber desarrollado un DSL específico para sistemas robóticos. Posteriormente se realiza una somera descripción del meta-modelo de V^{3}Studio, presentando las partes principales en que se ha estructurado. Cada una de estas partes se describe posteriormente con más detalle en el resto de secciones del capítulo.

Este capítulo se completa con el apéndice A, en el que se recogen las restricciones OCL que han sido añadidas a V^{3}Studio para comprobar que los modelos generados son semánticamente correctos. Estas restricciones han sido reunidas en un apéndice para no complicar la lectura de un capítulo ya complejo y denso de por sí, pero son necesarias para comprender completamente y utilizar correctamente V^{3}Studio, ya que algunos detalles sólo están descritos en forma de restricción OCL en dicho apéndice.

6.2.1 Justificación del enfoque

Como menciona Abouzahra [3], actualmente existen dos tendencias para realizar un desarrollo basado en modelos. La primera de ellas promueve la utilización de lenguajes de modelización estándar, como UML [177] o el lenguaje de modelado de sistemas (System Modelling Language (**SysML**)[7]) [175], mientras que la segunda tendencia apuesta por la utilización de lenguajes específicos de dominio (DSL) [208].

[7] SysML es realmente un *profile* de UML para modelar y analizar sistemas ingenieriles.

UML es el primer lenguaje de modelado que ha conseguido alcanzar una gran difusión en el mundo del software. UML proporciona un amplio conjunto de notaciones que permiten modelar los sistemas desde distintos puntos de vista, manteniendo la coherencia entre todas estas vistas. En su última versión, UML 2, añade incluso un diagrama específico para modelar sistemas basados en componentes, por lo que resulta, a priori, el candidato más adecuado para desarrollar V^{3}Studio. Además, UML ofrece un mecanismo de extensión conocido como *profile* que permite modificarlo y adaptarlo a las necesidades particulares del dominio. Este mecanismo ha sido tradicionalmente usado para generar herramientas gráficas, ya que la mayoría de las herramientas CASE que soportan UML permiten crear editores gráficos basados precisamente en un *profile*.

Aunque los *profiles* son un mecanismo de *extensión* con el que adaptar UML a las necesidades particulares de un proyecto o dominio, su uso implica la necesidad de trabajar por *restricción*, i.e. añadiendo restricciones extras a UML para detectar y eliminar los atributos o relaciones no deseados. Un *profile* se desarrolla tomando como base alguno de los elementos del meta-modelo de UML y, por tanto, el nuevo concepto que aparece en el *profile* «hereda» una gran cantidad de relaciones y atributos del concepto base. Relaciones y atributos que pueden ser o no necesarios para el dominio en cuestión, y que generalmente complican el uso del *profile*. La infraestructura de UML 2 [176] afirma lo siguiente sobre los escenarios en que es conveniente desarrollar un *profile*:

> *(The Profiles chapter) deals with use cases comparable to the MOF at the meta-meta-level, which is one level higher than the UML metamodel specification. [...]*
>
> *The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile. [...]*
>
> *First-class extensibility is handled through MOF, where there are no restrictions on what you are allowed to do with a metamodel: you can add and remove metaclasses and relationships as you find necessary. [...]*
>
> — UML 2.0 Infrastructure [176]

Por ejemplo, los conceptos que aparecen en un *profile* para modelar los elementos de un dominio suelen heredar de la meta-clase `Class`; de hecho, hasta MOF utiliza `Class` en su definición, por lo que se heredan todas sus relaciones y atributos. Pero es posible que el diseñador no necesite todo lo que hereda; o que incluso le estorbe y dificulte la inteligibilidad del *profile*. Siguiendo con el ejemplo, todos los conceptos del *profile* que

hereden de `Class` obtienen también una referencia a un caso de uso (`UseCase`). Esta referencia al caso de uso aparece en el modelo generado a partir del *profile*, pero es posible que el diseñador no quiera que aparezca. El desarrollador del *profile* tiene que recurrir, en este caso, a la definición de una restricción para evitar que se puedan añadir casos de uso a los modelos que se realizan con su *profile*, lo que se puede convertir en un trabajo pesado en caso de que se quieran eliminar muchas de estas relaciones y atributos.

Por otro lado, los DSLs proporcionan un conjunto reducido y específico de conceptos y una notación adaptada especialmente para describir sistemas en un dominio concreto. Pero su gran ventaja es también su gran desventaja: al ser tan específicos su adaptación a otros dominios es compleja (y no siempre posible) y, por lo general, no es posible encontrar un único DSL que permita describir la solución con todo detalle, sino que a menudo se desarrolla un DSL para cada vista, con el consiguiente problema añadido de coordinación y fusión entre los distintos modelos [81].

Otra tercera solución al problema de describir los conceptos elementales en que se basa CBD sería la utilización de un lenguaje de descripción arquitectónica (ADL) [144] (ver apartado 3.2.4). Si bien su utilización reportaría grandes beneficios (por ejemplo, Wright está basado en CSP y por tanto puede usarse para demostrar propiedades del sistema), los ADLs están generalmente demasiado centrados en la descripción arquitectónica en base a componentes, y descuidan la forma en que dichos componentes son construidos. Los ADLs podrán llegar a ser una herramienta muy eficaz en el desarrollo de sistemas basados en componentes si algún día existe un mercado de componentes en el que comprarlos ya hechos.

Rápidamente se desechó la idea de desarrollar un nuevo ADL, aunque V^{3}Studio ha adoptado parte de los conceptos definidos en ellos. En cuanto a la pugna entre el desarrollo de un DSL o un *profile* UML/SysML, Bézivin [49] defiende que a menudo es mucho más difícil trabajar por restricción, i.e. desarrollando un *profile* de UML, que por extensión, i.e. definiendo un nuevo DSL, y por tanto se decidió que V^{3}Studio fuera desarrollado como un DSL, especialmente dirigido al dominio de la robótica. Sin embargo, no se quiso desarrollar completamente un nuevo lenguaje empezando desde cero. Puesto que UML es un lenguaje de propósito general muy desarrollado y ampliamente aceptado, y que existen numerosas herramientas compatibles con él, se decidió que V^{3}Studio se basaría en gran medida en UML. De esta forma se aprovecha el conocimiento embebido en su meta-modelo y se consigue que las transformaciones entre V^{3}Studio y UML sean relativamente fáciles de realizar. Se obtiene además un meta-modelo adaptado a las necesidades del grupo, con el necesario poder expresivo, fácil de comprender y utilizar y que contiene los elementos estrictamente necesarios.

6.2.2 DESCRIPCIÓN GENERAL DE V³STUDIO

Como ya se ha descrito, V³Studio ha sido diseñado para modelar el tipo de sistemas basados en componentes que desarrolla el grupo de investigación DSIE. Por tanto, tiene en cuenta los conceptos básicos en que se basa CBD: componente, puerto y conector. El componente encapsula completamente su funcionalidad, que expone en sus puertos en forma de servicios provistos y requeridos. Estos servicios serán, respectivamente, utilizados o atendidos por el resto de componentes del sistema. Los componentes se comunican entre sí mediante el uso de conectores, que conectan puertos *compatibles*, i.e. aquellos cuyos servicios provistos aparecen como requeridos en el conjugado y viceversa.

Puesto que se persigue un enfoque completamente generativo a partir de un modelo, es necesario añadir los conceptos imprescindibles que permitan describir completamente el comportamiento de los componentes, de forma que se pueda generar automáticamente la traducción a distintos lenguajes. Para esto se estudió la forma en que UML describe el comportamiento de sus elementos, y se decidió reutilizar parte de la estructura que define. Concretamente, se decidió que el comportamiento de un componente se iba a describir por medio de una máquina de estados, cuyo propósito es especificar las acciones que realiza un componente y sus reacciones frente al envío de mensajes por parte del resto de componentes del sistema. Por último, se ha adaptado el diagrama de actividades de UML para describir la secuencia de algoritmos que es ejecutada por el componente en cada estado, independientemente de si la ejecución se produce como consecuencia de una petición de servicio de otro componente del sistema o si es consecuencia de la funcionalidad intrínseca del componente (las dos posibles fuentes de ejecución de una actividad).

V³Studio contempla tres «vistas» o conjunto de clases relacionadas que permiten modelar cada uno de los tres aspectos de un sistema basado en componentes. Cada una de estas «vistas» describe alguno de los tres aspectos descritos anteriormente: organización arquitectónica, descripción del comportamiento o descripción algorítmica. La figura 6.2 muestra de forma esquemática las tres partes en que se ha organizado el meta-modelo de V³Studio y las relaciones entre ellas, mientras que en la figura 6.3 puede observarse el esquema de paquetes generado y una enumeración de las clases definidas en cada uno de ellos. Las vistas, que son descritas con mayor detalle en las siguientes secciones, son las siguientes:

Vista arquitectónica: permite definir los componentes (ya sean simples o compuestos) que forman la aplicación, sus puertos, los tipos de datos y las interfaces globales del sistema, los servicios ofrecidos por las interfaces, asociar al componente una de las máquinas de estados ya definidas y conectar componentes entre sí.

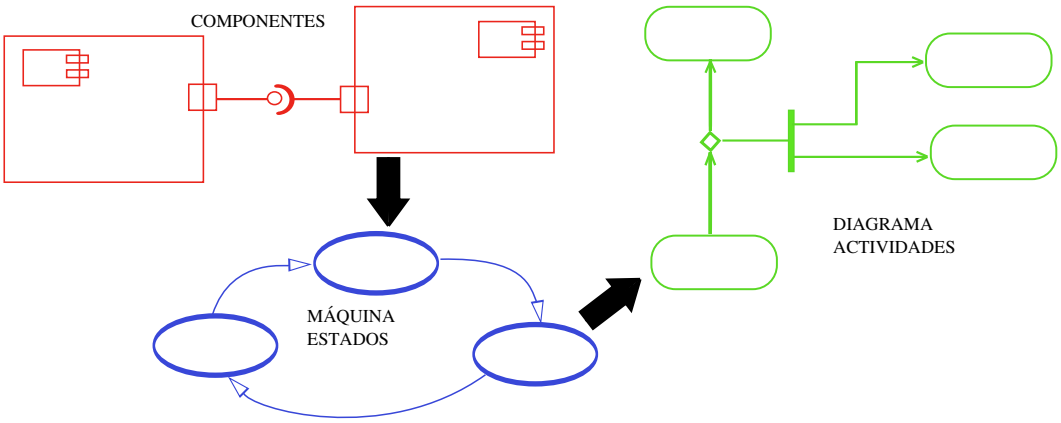


Figura 6.2: Esquema del meta-modelo de V^{3}Studio

Vista de comportamiento: esta vista permite definir una máquina de estados que describa tanto el comportamiento interno de un componente como su reacción ante la solicitud, por parte de otros componentes, de alguno de los servicios que proporciona.

Vista algorítmica: permite describir la secuencia de algoritmos que van a ser ejecutados cuando un componente se encuentre en un estado concreto o como respuesta a la solicitud de alguno de los servicios ofrecidos. Estos algoritmos son ejecutados únicamente cuando el componente se encuentre en el estado adecuado.

Al igual que sucede en UML, se han utilizado relaciones (EReference) de composición entre todos los conceptos que están íntimamente relacionados entre sí, i.e. los conceptos que al combinarse conforman cada una de las «vistas», y las relaciones de asociación entre los conceptos que sirven de unión entre cada vista. Gracias al débil acoplamiento que aportan las relaciones de asociación en EMF es posible diseñar cada uno de los modelos por separado (en su propio editor) y reutilizar el mismo diagrama en distintos contextos (cuando tenga sentido hacerlo). Por ejemplo, es posible declarar una misma máquina de estados que puede ser reutilizada en distintos componentes o un diagrama de actividad que se puede repetir en distintos estados de distintas máquinas de estados.

El meta-modelo de V^{3}Studio se ha diseñado de forma que permite desarrollar cada una de las partes que definen un aspecto de la arquitectura del sistema por separado, ya que la unicidad del meta-modelo asegura la coherencia de las mismas cuando se unen. Para que esto sea así, es preciso definir una serie de restricciones OCL que aseguren que el modelo está bien formado y es correcto. Estas restricciones pueden consultarse en el apéndice A, aunque aparecerán referenciadas a lo largo del presente capítulo, cuando se utilicen. Para reducir el número de restricciones necesarias, V^{3}Studio impone más limitaciones que UML en muchos casos para reducir las posibilidades de generación de modelos *mal formados*, i.e. modelos que son correctos sintácticamente pero no semánticamente.

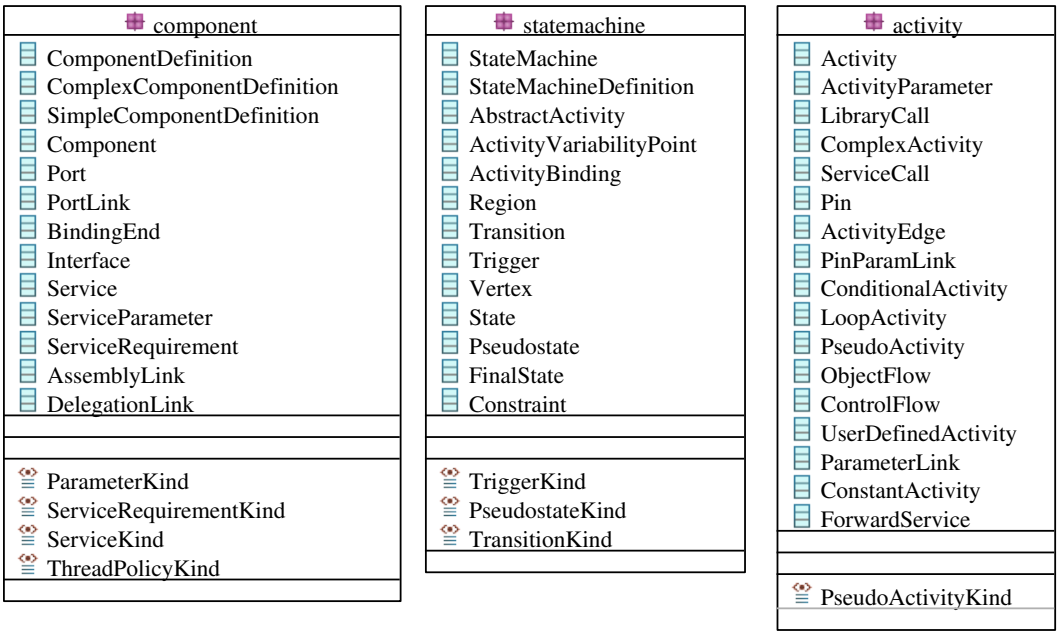


Figura 6.3: Esquema de paquetes de V³Studio

Llegados a este punto, cabe destacar las dos primeras diferencias entre V³Studio y UML: (1) V³Studio no contempla otros diagramas presentes en UML, e.g. casos de uso, despliegue, objetos, etc, y (2) V³Studio fuerza que la descripción del comportamiento se realice con una máquina de estados. En UML es posible describir el comportamiento de un *BehavioredClassifier* utilizando indistintamente un diagrama de actividades o una máquina de estados, de forma que debe ser posible obtener una representación partiendo de la otra (son representaciones equivalentes). Si bien es cierto que existen componentes cuyo comportamiento se ajusta más a una u otra forma de representación, la necesidad de proporcionar estas dos posibilidades y la transformación entre ambas introducen una complicación mayor que los posibles beneficios que podría aportar su adición al meta-modelo de V³Studio.

Por tanto, se ha tomado la decisión de forzar que el comportamiento de todo componente se exprese utilizando una máquina de estados. Esta máquina de estados describe no sólo el comportamiento interno del componente sino también la relación de un componente con el resto de componentes que forman la aplicación. Está decisión de diseño no limita el poder expresivo de V³Studio, ya que aquellos componentes cuya funcionalidad se expresa de forma más natural utilizando un diagrama de actividades (e.g. una secuencia de ejecución de algoritmos de localización o navegación) pueden ser modelados como componentes con una máquina de estados muy sencilla, cuyos estados pueden contener diagramas de actividad muy complejos que describen la secuencia de ejecución de los algoritmos.

6.3 VISIÓN GENERAL DEL USO DE V^{3}STUDIO

Esta sección pretende realizar un pequeño resumen de V^{3}Studio, presentándolo desde el punto de vista del usuario, desarrollador de aplicaciones basadas en componentes, en vez de desde el punto de vista del diseñador del meta-modelo. En esta sección se va a procurar presentar una visión práctica de V^{3}Studio, posponiendo los detalles de uso y las decisiones que han desembocado en el diseño final de V^{3}Studio a secciones posteriores (secciones 6.4, 6.5 y 6.6). El diseño de una aplicación basada en componentes con V^{3}Studio conlleva los siguientes pasos:

1. Definición de los tipos de datos (`DataType`) y las interfaces (`Interface`) globales del sistema, que serán compartidos por todos los componentes del mismo. En el caso de los tipos de datos sólo es necesario definir el nombre; V^{3}Studio no modela ninguna información extra en este nivel y deja los detalles de implementación al desarrollador. En cuanto a las interfaces, el desarrollador tiene que definir los servicios (`Service`) expuestos por la interfaz y las características de cada servicio: el tipo de servicio (`ServiceKind`) y los parámetros del servicio (`ServiceParameter`). En estos últimos tiene que especificar, además, el tipo de dato y la dirección (`ParameterKind`), i.e. si son de entrada o de salida.

2. Definición de los tipos (`ComponentDefinition`) de los componentes que integran el sistema. V^{3}Studio distingue entre «definición de un componente» e «instancia de un componente», de la misma forma y con el mismo significado que en la orientación a objetos tienen los conceptos «clase» y «objeto». Esta separación permite la reutilización de una misma definición de componente. Se comienza por la definición de los componentes simples (`SimpleComponentDefinition`), ya que estos forman el núcleo básico y de menor nivel de la aplicación. El desarrollador añade puertos (`Port`) a cada componente simple que esté modelando y selecciona las interfaces que cada puerto va a requerir y las que va a ofrecer al resto de componentes del sistema de las que ha definido previamente.

3. Definición de los componentes complejos (`ComplexComponentDefinition`). Un componente complejo sirve para agrupar un conjunto de componentes (simples o compuestos) y para controlar, de esta manera, el acceso a los servicios provistos por los componentes internos. Además, esta agrupación facilita la reutilización de grandes grupos de componentes. El desarrollador comienza por añadir al componente complejo los puertos por los que interacciona con el sistema. A continuación añade los componentes (`Component`) que forman el componente complejo y elige el tipo de cada uno de estos componentes de entre las definiciones

de componente creadas con anterioridad (ya sea un `SimpleComponentDefinition` o un `ComplexComponentDefinition`). Una vez añadidos todos los componentes tan sólo resta por enlazar sus puertos, uno a uno. Para ello se crea un `PortLink` por cada puerto que se enlaza y se selecciona, en cada uno de sus extremos (`BindingEnd`) el componente y el puerto que enlaza.

4. Configuración de cada componente (`Component`). El diseñador selecciona, para cada componente que haya añadido a un componente complejo, su política de ejecución (`ThreadPolicy`) y la forma en que quiere que se le sirvan los servicios requeridos (`ServiceRequirement`). Actualmente V³Studio contempla dos políticas de ejecución: o bien el componente se ejecuta en su propio hilo o bien se ejecuta en el hilo de su contenedor. En cuanto a la solicitud de servicios, dependiendo del tipo que se le haya asignado a un servicio (ver punto 1) se puede elegir o bien suscripción o sondeo o bien notificación de la ejecución o sin notificación.

5. Diseño de la definición de máquina de estados (`StateMachineDefinition`) que acompaña a una definición de componente. De nuevo, V³Studio distingue entre «definición de máquina de estados» e «instancia de máquina de estados». Una definición de componente referencia una definición de máquina de estados, mientras que un componente referencia una máquina de estados. De esta forma es posible reutilizar la misma definición de máquina de estados en varias definiciones de componente. Utilizando la vista de comportamiento, el desarrollador crea la definición de máquina de estados que modela tanto el comportamiento propio del componente (comportamiento «espontáneo») como el que muestra en sus relaciones con el resto de componentes del sistema (respuesta frente a las distintas peticiones de los servicios que ofrece).

   Para crear esta máquina de estados el diseñador dispone de los elementos clásicos: regiones (`Region`), estados (`State`), transiciones (`Transition`) y pseudo-estados (`PseudoState`). En las transiciones puede, además, elegir el tipo de evento que va a provocar su disparo (`Trigger`) y añadirle una guarda (`Constraint`) para controlar las condiciones bajo las que se produce el disparo de la transición. Mediante el uso de regiones puede crear macro-estados con regiones ortogonales, lo que amplía la potencia de modelado de esta vista.

6. Definición de las actividades que va a ejecutar un componente dependiendo del estado en que se encuentre. El desarrollador utiliza la vista algorítmica y las clases que ofrece para modelar distintos tipos de actividades (simples o compuestas) junto con sus respectivos parámetros (`ActivityParameter`). Entre las actividades simples que puede realizar un componente se encuentran la invocación de un servicio (`ServiceCall`), la ejecución de una función de librería (`LibraryCall`)

o la ejecución de un código arbitrario, suministrado por el propio diseñador (`UserDefinedActivity`). Las actividades pueden ser enlazadas entre sí (mediante dos tipos de `ActivityEdge` que modelan flujo de control y flujo de datos) para crear actividades más complejas (`ComplexActivity`). Actividades condicionales (`ConditionalActivity`) y bucles (`LoopActivity`) también se encuentran entre las actividades que pueden ser seleccionadas por el desarrollador.

La actividad de invocación de librería proporciona, además, un mecanismo para relacionar los parámetros de la actividad con los parámetros que requiere la función de la librería así como la convención de invocación de la función de la librería, ya que no todas las librerías se utilizan igual ni declaran ni los mismos tipos de datos ni en el mismo orden.

7. Completar las actividades asociadas a los estados y transiciones de la definición de máquina de estados de cada definición de componente. En este punto el desarrollador puede elegir entre parametrizar esta actividad (mediante un `ActivityVariabilityPoint`), y permitir que cada componente que se instancie elija la actividad que quiere ejecutar, o bien seleccionar una de las actividades previamente diseñadas, estableciendo de este modo una actividad común para todas las instancias de esta definición de máquina de estados.

8. Asociar cada definición de máquina de estados a su correspondiente definición de componente. En este punto se puede encontrar que cada definición de componente necesita su propia máquina de estados o, lo más probable, que varias definiciones de componente puedan utilizar la misma definición de máquina de estados (parametrizada o sin parametrizar).

9. Crear la máquina de estados (`StateMachine`) en cada uno de los componentes que se han definido en cada uno de los componentes complejos que forman la aplicación. Esta máquina de estados representa una instancia de la definición de la máquina de estados asociada a la definición del componente actual. En este momento el diseñador selecciona las actividades «reales» con que va a completar los posibles parámetros que pudiera tener la definición de la máquina de estados.

10. En este punto ya tienen que estar definidos completamente todos los componentes que forman la aplicación, su comportamiento y las actividades que ejecutan dependiendo del estado en que se encuentre el componente. Tan sólo resta por crear la definición de un componente complejo que representa la aplicación como un todo y añadirle los componentes que forman el sistema.

6.4 VISTA ARQUITECTÓNICA

LA VISTA arquitectónica de V3Studio está formada por un conjunto de clases que permiten describir la vista «externa» de un componente, como si fuera una «caja negra». V3Studio utiliza la definición de componente que realiza UML 2 [170]: «una unidad autocontenida que encapsula el estado y comportamiento de un conjunto de clasificadores. Un componente también especifica un contrato entre los servicios que proporciona a sus clientes y los que requiere del entorno para asegurar su correcto funcionamiento, en forma de interfaces requeridas y ofrecidas. Un componente es una unidad sustituible que puede ser reemplazada en tiempo de diseño o en tiempo de ejecución por otro componente que ofrezca un contrato, en forma de servicios ofrecidos y requeridos, equivalente». La figura 6.4 muestra en detalle las clases del meta-modelo de V3Studio que permiten realizar la descripción arquitectónica.

La figura 6.4 muestra, además, la relación que existe entre la vista arquitectónica y la vista de comportamiento. Las dos clases con fondo verde pertenecen a la vista de comportamiento y permiten definir y parametrizar la máquina de estados (`StateMachineDefinition`) asociada a una definición de componente y adaptar dicha máquina de estados (`StateMachine`) a las necesidades de un componente en particular. A pesar de que la descripción de estas dos clases se pospone a la sección 6.5, se ha decidido adelantar la relación que existe entre ambas vistas para facilitar la descripción de V3Studio. Las relaciones de asociación punteadas simbolizan que estas clases no pertenecen a la vista arquitectónica, aunque están directamente relacionadas con los conceptos básicos de esta vista (`ComponentDefinition` y `Component`, respectivamente) mediante sendas relaciones de asociación. Son justamente estas relaciones de asociación las que proporcionan la flexibilidad necesaria para poder diseñar por separado la máquina de estados.

El apéndice A.1 recoge todas las restricciones OCL relacionadas con esta vista. Las primeras restricciones que aparecen afectan a la totalidad del sistema y se corresponden con la detección de tipos repetidos, tarea básica que desempeña cualquier compilador. En el caso de V3Studio los tipos son siempre globales, por lo que esta detección es más sencilla. En concreto, existen restricciones para detectar la repetición de la definición de interfaces (restricción 7), servicios (restricción 2) y tipos de datos (restricción 12).

Esta sección se ha dividido en cuatro apartados en los que se exponen las características más sobresalientes de la vista arquitectónica de V3Studio y las decisiones de diseño que han posibilitado este diseño.

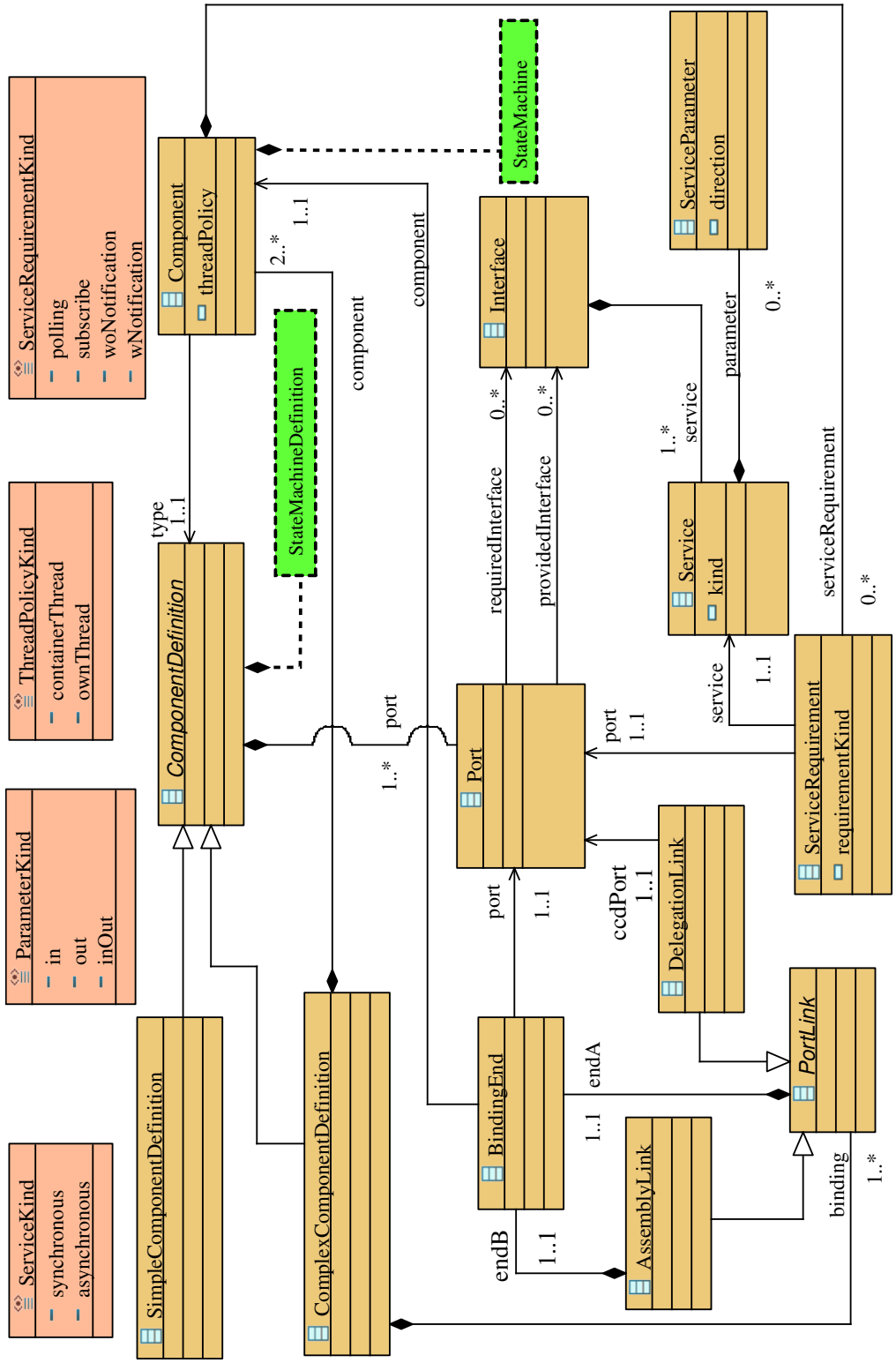


Figura 6.4: Extracto del meta-modelo de V^{3}Studio que muestra la descripción arquitectónica

6.4.1 CARACTERÍSTICAS DE LA ECLASS COMPONENT

La principal característica de la vista arquitectónica de V3Studio es la distinción que se realiza entre los conceptos «definición» (EClass abstracta `ComponentDefinition`) e «instancia» (EClass `Component`) de un componente, como menciona Atkinson en su artículo [16]. Esta separación de conceptos es similar a la que se realiza en los lenguajes orientados a objetos con los conceptos «clase» y «objeto». Esta distinción facilita (1) la reutilización de una misma definición de componente (`ComponentDefinition`) para instanciar cualquier número de componentes (`Component`), y (2) la modificabilidad de los componentes, ya que la descripción se encuentra localizada en un único sitio, lo que permite la fácil actualización de todas las instancias del sistema sin más que modificar la definición.

La EClass `Component` modela un componente instanciable, un artefacto software que existe en tiempo de ejecución, como sucede con el concepto «objeto». Como tal, un `Component` no puede existir aisladamente, sino que es necesario indicar cuál es su definición (ver apartado 6.4.2). Un `Component` obtiene, dependiendo del tipo de definición que referencia, un conjunto de puertos y servicios configurables y una máquina de estados parametrizable. Toda esta información variable ha de ser completada antes de que el `Component` pueda considerarse completamente inicializado y listo para ser ejecutado. Por tanto, el diseñador tiene que configurar los servicios que están definidos como configurables en la definición de cada `Component` (ver apartado 6.4.3) y completar las actividades marcadas como parametrizables en la máquina de estados asociada a su definición (ver apartado 6.5.1).

Toda esta parametrización sirve al objetivo de maximizar la reutilización de las definiciones de componente que se han realizado hasta el momento. Se intenta evitar que pequeñas modificaciones en la forma en que se utiliza un servicio (por ejemplo, sondeo de un valor o subscripción a cambios en su valor) o en el algoritmo en particular que se aplica en un estado determinado (por ejemplo, un PID o un control *fuzzy*) conlleven la creación de una nueva definición de componente, aún cuando la funcionalidad sea prácticamente la misma.

Por último, V3Studio no olvida que el software de forma general, y el software de control de robots en particular, hace uso de las facilidades de multi-tarea que ofrecen la mayoría de sistemas operativos. Un `Component`, como única unidad instanciable que representa a un componente en tiempo de ejecución, utiliza su EAttribute `threadPolicy` para especificar la forma en que se va a ejecutar. Para ello V3Studio ofrece dos posibilidades: o bien el componente se quiere ejecutar en su propio hilo (valor `ownThread`) o bien se va a ejecutar en el hilo de su contenedor (valor `containerThread`). Como muestra el apartado 6.4.2, V3Studio organiza todos los componentes que forman la

aplicación en una estructura jerárquica, en la que siempre existe un componente raíz que contiene al resto de componentes del sistema, por lo que siempre existe por lo menos un hilo de ejecución.

6.4.2 TIPOS DE DEFINICIÓN DE COMPONENTE

La figura 6.4 muestra que V^{3}Studio permite modelar dos tipos de definiciones de componente, componentes simples (`SimpleComponentDefinition`) y componentes compuestos (`ComplexComponentDefinition`), cuyas características comunes están agrupadas en la super-clase abstracta `ComponentDefinition`. Tal y como se dijo en el apartado 6.4.1, la definición de un componente juega el mismo papel que la definición de una clase en un lenguaje orientado a objetos. La definición de un componente es, por tanto, un concepto presente únicamente en tiempo de modelado que no existe en tiempo de ejecución, ya que sólo `Component` es «instanciable».

La EClass abstracta `ComponentDefinition` agrupa las características comunes a los dos tipos de definición de componente. Estas características son la existencia de puertos (EClass `Port`, mostrada en el diagrama), mediante los que el componente se comunica con el exterior, y la definición del comportamiento del componente (EClass `StateMachineDefinition`, que aunque no se muestra en el diagrama se describe con más detalle en la sección 6.5). Las principales características de estos tipos de definición de componente son las siguientes:

`SimpleComponentDefinition`: define la unidad más simple de composición, aquella que ya no se puede o no se quiere descomponer en componentes de grano más fino. Los componentes simples son los únicos que aportan realmente funcionalidad al sistema, ya que tienen un comportamiento propio. Como muestra la figura 6.4, un componente simple no añade nuevos conceptos a los ya definidos por la super-clase abstracta.

`ComplexComponentDefinition`: define un componente que está compuesto por la unión de otros componentes (EClass `Component`), ya sean simples o compuestos, haciendo uso del patrón de diseño [96] *Composite*[8]. Además de los propios componentes, un `CompositeComponentDefinition` contiene los enlaces entre los puertos compatibles de aquellos, enlaces que son modelados mediante la EClass abstracta `PortLink` y sus descendientes (ver sección 6.4.4 y restricción 14, A.1).

[8] El patrón *Composite* sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados ya que, al poseer todos ellos una interfaz común, se tratan todos de la misma manera.

Existen dos diferencias básicas entre estos dos tipos de definición. La primera y más importante es que sólo un `ComplexComponentDefinition` describe una aplicación, ya que es el único que contiene componentes que realmente puedan ser «ejecutados», i.e. instancias de componentes en tiempo de ejecución (modeladas con el concepto `Component`). Haciendo un símil con la orientación a objetos, un `ComplexComponentDefinition` define una clase que contiene objetos (otros componentes en este caso particular), mientras que el `SimpleComponentDefinition` sólo contiene la definición de una clase. Por tanto, un sistema robótico modelado con V^{3}Studio vendrá definido por un `ComplexComponentDefinition` que contiene al resto de componentes (`Component`) en que se ha organizado el sistema, ya sean simples (`SimpleComponentDefinition`) o compuestos (`ComplexComponentDefinition`).

La segunda diferencia atañe a la función desempeñada por cada uno de los tipos de `ComponentDefinition` de V^{3}Studio. Un `SimpleComponentDefinition` modela un componente con comportamiento propio, un componente que añade funcionalidad al sistema. Por tanto, su máquina de estados debe describir el comportamiento y la secuencia de algoritmos que ejecuta el componente, dependiendo del estado en que se encuentre. Por el contrario, un `ComplexComponentDefinition` ha sido diseñado para modelar una agregación de componentes y para controlar el acceso a los mismos, más que para crear un nuevo componente con un comportamiento propio. De esta afirmación se desprende que un componente complejo muestra un comportamiento *emergente*, fruto de la combinación del comportamiento de los componentes que contiene.

Por tanto, un `ComplexComponentDefinition` *debe* actuar únicamente como un *proxy*, que encamina mensajes entre los puertos de los componentes que contiene y el exterior, y viceversa. La redirección de las peticiones de servicio se modela mediante la actividad `ForwardService` (consultar apartado 6.6.2), mientras que la restricción 16 comprueba que todas las actividades de la máquina de estados (ver vista de comportamiento en la sección 6.5) que describe el comportamiento de un componente complejo son de este tipo. Como se desprende de esta última diferencia, la clase `Port` tiene asociada un comportamiento ligeramente distinto dependiendo del tipo de componente que lo contenga, lo que se traduce en que la transformación de los puertos se lleva a cabo con ligeras variaciones entre un caso y otro (consultar apartado 7.3.3).

La decisión de distinguir entre dos tipos de definición de componente simplifica el diseño y la comprensión de los modelos V^{3}Studio, ya que cada definición de componente tiene un rol y unas características diferenciadoras y únicas. Esta decisión diferencia V^{3}Studio de UML, ya que UML sólo contempla un tipo de definición de componente. Aunque en este caso es sencillo relacionar ambos meta-modelos: un `ComponentDefinition` en V^{3}Studio es un `Component` en UML y un `Component` en V^{3}Studio se convierte

en un `ComponentRealization` en UML. La utilización de dos tipos de definición de componente repercute también sobre el tipo y la funcionalidad de los puertos, tal y como se explica en el apartado 6.4.3. La decisión de definir dos tipos de componente sólo tiene una consecuencia negativa: la necesidad de crear un `SimpleComponentDefinition` «artificial» en caso de que el `ComplexComponentDefinition` tenga que realizar alguna función extra además de la redirección de las peticiones de servicio a su correspondiente componente interno. Pero es un pequeño precio a pagar por la simplicidad de diseño e implementación que se consigue con esta decisión de diseño.

6.4.3 PUERTOS E INTERFACES EN V^{3}STUDIO

Los puertos de un componente, modelados mediante la EClass `Port`, representan los puntos de interacción del componente con el resto de componentes del sistema. Al ser un concepto común a los dos tipos de definición de componente mostradas en el apartado 6.4.2, esta relación está recogida en la EClass abstracta `ComponentDefinition`. Como muestra la figura 6.4, un puerto tiene sendas referencias a los conjuntos de interfaces que modelan los servicios ofrecidos y requeridos por el mismo en cada uno de sus puerto. Para que un puerto esté correctamente definido es condición necesaria que ninguna de las interfaces ofrecidas o requeridas aparezcan repetidas en el mismo puerto (restricción 1, A.1).

Como se observa en dicha figura, entre los conceptos `Port` e `Interface` existe una relación de asociación y no de composición, lo que hace posible (1) que distintos `Port` hagan referencia a la misma `Interface`, de forma que es posible comprobar fácilmente si dos puertos son *compatibles* y (2) que las `Interface`s sean comunes a todos los componentes que forman un sistema. Cada `Interface` lleva asociada un conjunto de servicios (`Service`) que modelan las operaciones que forman parte de la interfaz. Una `Interface` puede verse como un conjunto de `Service`s agrupados y relacionados por el diseñador por alguna razón (por ejemplo, por estar semánticamente relacionados o porque es necesario que todos los servicios sean proporcionados o requeridos por el mismo componente). Para definir completamente un `Service` se han añadido los conceptos necesarios para modelar:

- Los parámetros (`ServiceParameter`) asociados a la operación, tanto el tipo de dato del parámetro (`DataType`, que, al igual que la interfaz, es un tipo definido globalmente) como su dirección (`ParameterKind`), que puede ser de entrada, salida o entrada/salida.

- El comportamiento del servicio, i.e. la forma en que un componente va a resolver la petición de un servicio (en caso de que forme parte de una interfaz ofrecida) o el

comportamiento que se puede esperar del componente que ofrece el servicio. Esta es una de las características que un `Component` puede configurar mediante el uso de la EClass `ServiceRequirement`, tal y como se describe más abajo. Actualmente se pueden especificar dos tipos de comportamientos: `synchronous` y `asynchronous`, representados en el tipo enumerado `ServiceKind`. El primero establece que el servicio está siempre disponible y que el componente que lo ofrece va a procesar la petición de servicio tan pronto como la reciba (la mayor parte de los servicios que devuelven algún dato encajan en este tipo), mientras que el segundo especifica que la petición de servicio se va a resolver de forma asíncrona por el componente que lo ofrece.

Una `Interface` no especifica, en ningún caso, el uso que de ella tiene que hacer el puerto (y en última instancia el componente que lo contiene), sino que sólo especifica el nombre y el contenido (en forma de servicios, `Service`) del contrato que define. Es el `ComponentDefinition`, mediante cada uno de sus `Port`, quien especifica cómo va a utilizar cada una de las `Interface` (contrato) referidas en él, dependiendo de si forman parte de la EReference *providedInterface* (interfaces provistas) o *requiredInterface* (interfaces requeridas por el componente).

Un `Component`, como ya se dijo en el apartado 6.4.1, contiene una serie de `ServiceRequirement` que le permiten especificar la forma en que quiere que se le sirvan las peticiones de servicio (una de las características configurables en un componente). Dependiendo de la forma en que se va a proporcionar el servicio (síncrono o asíncrono), existen dos alternativas de configuración. Para servicios síncronos se ofrece las alternativas de sondeo (`polling`) o subscripción a cambios en valor (`subscribe`), mientras que para servicios asíncronos se puede pedir notificación de la ejecución del servicio (`wNotification`) o no (`woNotification`).

El concepto `ServiceRequirement`, como se mencionó en el apartado 6.4.1, se añadió a V3Studio para aumentar los posibles escenarios en que un `ComponentDefinition` se puede reutilizar. El objetivo es evitar, o limitar en lo posible, que pequeños cambios en la forma en que un componente realiza la petición de un servicio requieran la creación de una nueva definición de componente, aún cuando la máquina de estados sea exactamente la misma. En este punto es necesario definir una serie de restricciones OCL para asegurar que se especifica un requisito de utilización para todos los servicios referidos por todos los puertos de un componente o que dicho requisito toma un valor adecuado al tipo de servicio. Las restricciones que comprueban la correcta especificación de un `ServiceRequirement` se pueden consultar en el apéndice A.1, restricciones 6, 13 y 8.

6.4.4 COMUNICACIÓN ENTRE COMPONENTES

CBD establece que la comunicación entre componentes se lleva a cabo por medio de conectores, que permiten controlar la forma en que esta comunicación se desarrolla. Sin embargo, en la figura 6.4 puede observarse que V^{3}Studio no modela directamente el concepto «conector» tal y como se define en CBD, sino que en su lugar modela uniones simples entre puertos mediante la EClass abstracta `PortLink`. V^{3}Studio establece actualmente que los conectores, en el sentido CBD, son componentes especiales que median en la comunicación entre componentes, pero que no tienen una definición propia y diferenciada de la del componente. Si se requieren protocolos de comunicación más complejos se recurrirá al desarrollo de componentes específicos que implementen estos protocolos y que actúen como mediadores entre los componentes que se tienen que comunicar utilizando estos protocolos.

Además, los conectores en CBD se utilizan también para realizar la distribución de los componentes que integran la aplicación en distintos nodos de computación. De nuevo, V^{3}Studio no contempla actualmente la posibilidad de distribuir los componentes que forman la arquitectura, aunque, como en el caso del diseño de conectores complejos, ofrece una alternativa: diseñar un conjunto de componentes simples que encapsulen la funcionalidad del conector y que permitan realizar la distribución de la aplicación. La figura 6.5 muestra un ejemplo en que se emula el comportamiento de un conector (CORBA en este caso) mediante la generación de dos componentes simples que realizan la labor de distribución. Cada uno de los sistemas resultantes (parte inferior de la figura) se ejecutarían en su propio nodo de computación.

Como muestra la figura 6.4, V^{3}Studio contempla dos tipos de conexiones entre puertos de distintos componentes. Estos dos tipos de conexiones se materializan en la existencia de dos clases que heredan de la clase `PortLink`: `AssemblyLink` y `DelegationLink`. Un `AssemblyLink` modela la unión entre dos puertos de dos componentes (`Component`) contenidos en un mismo componente complejo (`ComplexComponentDefinition`), mientras que un `DelegationLink` modela la unión entre un `Component` y la definición de componente complejo que lo contiene. Esta separación es necesaria ya que los puertos que une un `PortLink` cumplen un cometido diferente dependiendo del tipo de componente y, por tanto, existen unas restricciones adicionales asociadas a cada tipo de `PortLink`:

Un `AssemblyLink` permite conectar los puertos *compatibles* (i.e. aquellos en los que las interfaces ofrecidas en un extremo son requeridas por el otro y viceversa) de dos componentes contenidos en un mismo componente complejo (consultar restricción OCL 4). Además, se ha de asegurar también que se conectan los puertos de los componentes correctos (consultar restricción OCL 5).

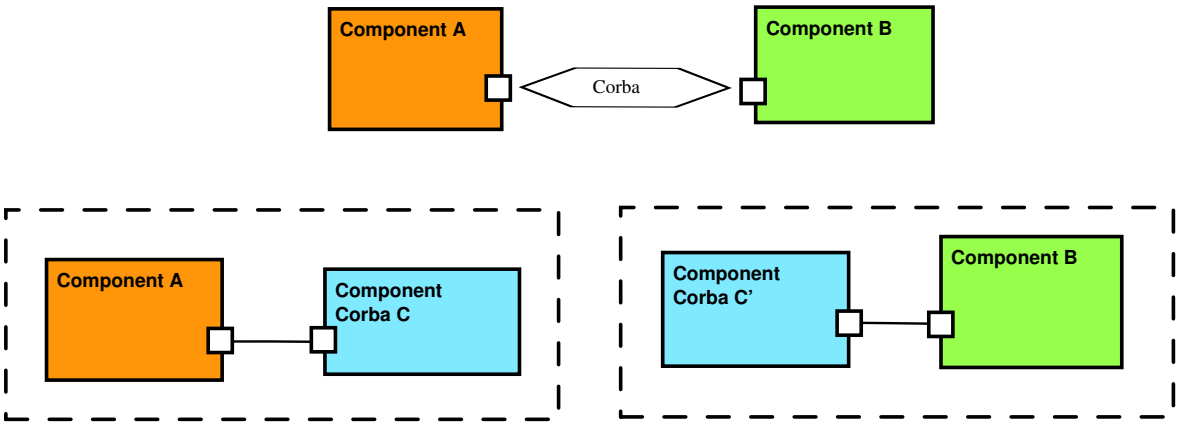


Figura 6.5: Ejemplo de modelado de un conector (CORBA en este caso) en V3Studio para desarrollar un sistema distribuido. El diagrama superior muestra la visión CBD, en la que el conector CORBA permite realizar la distribución. El diagrama inferior muestra la visión V3Studio, en la que el conector se transforma en dos componentes simples que ocultan la distribución y que se conectan, respectivamente, a los mismos componentes que la versión original.

Un `DelegationLink` sirve para redirigir las peticiones de servicio de un componente complejo hacia el componente (simple o complejo) contenido en él que va procesarla. En este último caso, la conexión sólo se puede realizar si ambos puertos tienen las mismas interfaces ofrecidas y requeridas, ya que el puerto de un componente complejo sólo realiza labores de redirección, tanto hacia el exterior como al interior del componente (consultar restricción OCL 3). Además, se ha de asegurar también que se conectan los puertos de los componentes correctos (consultar restricción OCL 9).

La conexión entre puertos en UML se lleva a cabo de forma ligeramente distinta. En UML la conexión entre puertos se realiza mediante la clase `Connector`, que tiene una propiedad para especificar el tipo de conexión (que coincide con los dos tipos de `PortLink` modelados en V3Studio).

La EClass `BindingEnd` representa cada uno de los extremos que contiene una unión entre puertos. De las definiciones anteriores de los tipos de uniones se puede extraer que estas uniones tienen siempre un extremo que enlaza con el puerto de un componente (`Component`), mientras que el otro extremo puede enlazar, o bien con otro puerto de otro componente (en este caso, se trataría de un `AssemblyLink`), o bien con uno de los puertos de la definición de componente complejo en que está contenido (y se trataría de un `DelegationLink`). Por tanto, en la figura 6.4 se puede observar que el extremo común (representado por la EClass `BindingEnd`) se ha definido en la clase padre `PortLink`, mientras que los hijos contienen, respectivamente, o bien otro `BindingEnd` (para enlazar con el correspondiente `Component`), o bien una referencia al puerto contenido en la definición de componente complejo que se quiere enlazar (referencia `ccdPort`).

Aunque parezca un poco artificial, la EClass `BindingEnd` es necesaria, ya que los `Component` que se enlazan son *instancias* de una definición de componente y, por tanto, no contienen directamente los puertos involucrados en la conexión (consulta figura 6.4). Puesto que es posible reutilizar la misma definición en varias instancias, se hace necesario disponer de una forma de seleccionar el componente (instancia concreta, representada por la clase `Component`) a cuyo puerto se está conectando los extremos correspondientes de un `PortLink` y sus descendientes. Esta flexibilidad requiere la adición de una restricción extra (restricción 10) para comprobar que, efectivamente, las referencias al puerto y al componente que contiene un `BindingEnd` son correctas, i.e. el puerto está contenido en la definición de componente que referencia el componente.

Actualmente, el meta-modelo requiere que los puertos de los componentes que se conecten sean perfectamente simétricos, es decir, que los servicios provistos por un puerto sean requeridos por el puerto conjugado y viceversa. Sin embargo, esta condición es demasiado rígida, ya que algunas de las interfaces ofrecidas por un componente podrían quedar perfectamente desconectadas. Esto responde al hecho de que se puede tener un componente con más funcionalidad de la que se necesita, lo cuál no debería evitar que dicho componente fuera reutilizado en una aplicación que necesita menos funcionalidad que la que aporta. Por otro lado, V^{3}Studio también limita la capacidad de conexión de los componentes a sólo una conexión por puerto (consultar restricción 11), limitación que no existe en UML.

Esta restricción es debida al hecho de que los servicios que ofrecen los puertos de un componente a menudo tienen que devolver un valor al puerto que originó la petición. En caso de que se pudiera realizar esta conexión muchos-a-uno (muchos puertos con servicios requeridos a uno con servicios ofrecidos) resultaría difícil retornar los valores al puerto que la originó. Estas conexiones muchos-a-uno sí tendrían sentido en puertos que ofrecieran un servicio de *multi-cast*, pero actualmente V^{3}Studio no contempla esta posibilidad. La posibilidad inversa, i.e. conexiones uno-a-muchos (un puerto con servicios requeridos conectado a muchos con servicios ofrecidos), están desde luego prohibidas en V^{3}Studio y se descarta su adopción en un futuro.

6.5 Vista de Comportamiento

Las clases que forman esta vista permiten describir, mediante el uso de una máquina de estados, el comportamiento interno de un componente y su respuesta ante la invocación de alguno de los servicios que ofrece. La figura 6.6 muestra un extracto de las clases de V^{3}Studio que cumplen este cometido. Como puede observarse en

dicha figura, esta parte está inspirada en su homóloga del meta-modelo de UML 2, aunque se ha modificado ligeramente para hacerla más simple. Entre estas diferencias destaca que (1) V3Studio contempla la definición de máquinas de estados parametrizadas [108, 215] para aumentar el número de escenarios en que se pueden reutilizar las máquinas de estados definidas y (2) V3Studio distingue, al igual que sucede en la vista arquitectónica (consultar sección 6.4), entre definición e instancia de máquina de estados. En el apéndice A.2 están recogidas todas las restricciones OCL relacionadas con esta vista.

Dicha figura 6.6 muestra también la relación que existe entre la vista de comportamiento y la vista algorítmica, representada mediante la clase azul `Activity`. Esta clase, descrita en la sección 6.6, es la clase base de la que heredan el resto de clases que permiten describir un algoritmo en V3Studio. Aunque forma parte de la vista algorítmica, su inclusión en la figura 6.6 ayuda a comprender mejor la relación entre ambas vistas.

En cuanto a la semántica de la ejecución y la transición de estados en una máquina de estados, V3Studio mantiene la definición que realiza UML 2, en la que tanto los estados como las transiciones tienen asociados unas actividades que se ejecutan o bien mientras la máquina de estados está en un estado (referencia `doActivity`) o bien en los momentos en que se cambia de estado (referencias `entry`, `exit` y `effect`). No todas las actividades que se pueden definir mediante la vista algorítmica (consultar sección 6.6) pueden ser ejecutadas en este caso, sino que se ha definido una restricción para evitar la creación de modelos incorrectos, e.g. aquellos en los que una pseudo-actividad (ver apartado 6.5.3) pueda ser ejecutada (restricción 21, A.2). Para que la máquina de estados sea correcta también es necesario que, de forma general, no quede ningún estado sin conectar (restricción 26).

En cuanto a la dinámica de ejecución de la máquina de estados, V3Studio adopta la misma definición que realiza UML 2. Es decir, tan pronto se produce una transición que se puede ejecutar (i.e. la guarda se evalúa afirmativamente) se produce el cambio de estado. Este proceso conlleva la finalización de la actividad `doActivity` que pudiera estar ejecutando el estado actual y la posterior ejecución de la actividad referenciada como `exit`. A continuación se ejecuta la actividad `effect` de la transición y, por último y en este orden, las actividades `entry` y `doActivity` del estado entrante, siempre que existieran.

Los siguientes apartados describen en detalle esta parte del meta-modelo de V3Studio. Concretamente, se presenta primero la infraestructura que permite parametrizar la máquina de estados asociada a la definición de un componente y la forma en que cada componente «rellena los huecos», adaptando la máquina parametrizada a sus necesidades. Los dos apartados siguientes describen las clases que permiten desarrollar los modelos básicos de máquinas de estados, para terminar con un apartado sobre macro-estados y regiones ortogonales, un mecanismo que reduce el número de estados necesarios para expresar determinados conceptos y que mejora, además, la inteligibilidad de los diagramas.

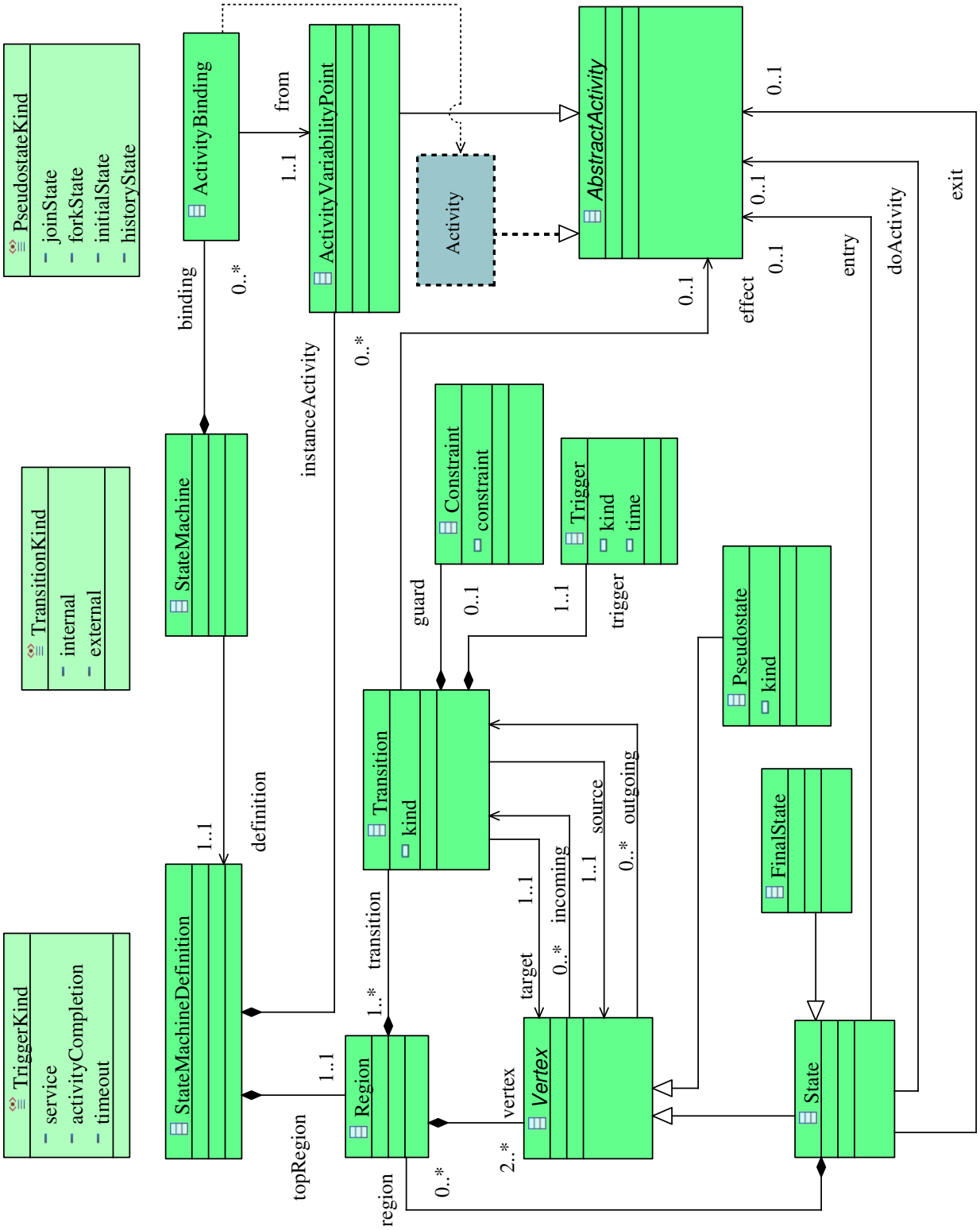


Figura 6.6: Extracto del meta-modelo de V^{3}Studio que muestra la descripción del comportamiento.

6.5.1 PARAMETRIZACIÓN DE LA MÁQUINA DE ESTADOS

Para aumentar el número de escenarios en que una máquina de estados puede ser reutilizada y evitar, de esta forma, que cambios en la secuencia de algoritmos que tiene que ejecutar el componente en un estado requieran la creación de una nueva máquina de estados, se decidió aumentar el meta-modelo de V^{3}Studio con máquinas de estados parametrizables. Al igual que sucede en los lenguajes de programación que soportan genéricos o plantillas, para llevar a cabo esta modificación fue necesario separar los conceptos «definición» (EClass `StateMachineDefinition`) e «instancia» (EClass `StateMachine`) de una máquina de estados, así como establecer una relación entre ambas (denominada `definition`).

La separación entre definición e instancia en este caso coincide con la realizada en la vista arquitectónica (consultar apartado 6.4.1) y está directamente relacionada con ella. Esta relación aparece reflejada, para facilitar la comprensión de V^{3}Studio, en la figura 6.4 mediante unas relaciones de composición punteadas. Como sucede en la vista arquitectónica, la «definición» (`StateMachineDefinition`) contiene todos los elementos comunes a todas las instancias (en este caso, la propia máquina de estados y los puntos de variación), mientras que la «instancia» (`StateMachine`) representa la máquina de estados concreta de un componente (`Component`) concreto, en el que se han completado todos los punto de variación que se hayan podido establecer en la «definición» con algunos de los algoritmos que defina el usuario mediante la vista algorítmica (consultar sección 6.6).

La relación entre vista arquitectónica y vista de comportamiento establece que un `ComponentDefinition` tiene asociado un `StateMachineDefinition`. Esta clase especifica tanto el comportamiento común que tendrán todos los componentes con esta definición como que ciertas actividades de la máquina de estados asociada a la definición del componente son parametrizables y, por tanto, pueden variar entre distintas instancias de la definición del mismo. Un `Component`, como instancia de un `ComponentDefinition`, contiene un `StateMachine` en la que completa las actividades variables (parámetros) de su definición (`StateMachineDefinition`) con algoritmos que son modelados utilizando la vista algorítmica (ver sección 6.6).

Como se desprende de estas líneas, existe un fuerte acoplamiento entre la definición y la instancia de un componente y de la máquina de estados (definición e instancia) asociada a cada uno de ellos. Este acoplamiento se ha relajado en V^{3}Studio mediante relaciones de asociación para poder diseñar cada vista (arquitectónica, diagrama de componentes, y de comportamiento, máquina de estados) por separado y para poder reutilizar una máquina de estados (parametrizada) en distintos componentes. Sin embargo, es preciso establecer una restricción OCL que asegure que la máquina de estados

(StateMachine) de un componente (Component) se corresponde con la definición de la máquina de estado (StateMachineDefinition) asociada a la definición del componente ComponentDefinition a la que hace referencia el Component en cuestión (restricción 27).

Un StateMachineDefinition contiene, además de la máquina de estados común para todos los componentes cuya definición coincida con la definición de componente que contenga este StateMachineDefinition, un conjunto de ActivityVariabilityPoint que permiten al diseñador parametrizar algunas de las actividades asociadas la definición de la máquina de estados. Esta parametrización es posible gracias a que ActivityVariabilityPoint hereda de la super-clase AbstractActivity, que, como se muestra en el apartado 6.6.1, es la clase base de la que heredan el resto de clases que hacen posible la descripción algorítmica en V^{3}Studio. De esta forma, el diseñador de la definición de la máquina de estados puede elegir qué actividades de las asociadas a un estado o a una transición son fijas y comunes para todas las instancias de la máquina de estados, y cuáles son parametrizables y pueden variar entre las distintas instancias.

Por otro lado, un Component tiene un StateMachine que contiene, a su vez, una serie de ActivityBinding que le permiten fijar los puntos de variación especificados en el StateMachineDefinition mediante las actividades variables ActivityVariabilityPoint. Los ActivityBinding permiten al componente especificar qué actividad (consultar el apartado 6.6.1 sobre la EClass Activity) se va a invocar finalmente para esta instancia en particular. Puesto que el meta-modelo de V^{3}Studio por sí solo no puede verificar estas condiciones, es necesario añadir una restricción OCL para verificar que todos los puntos de variación (ActivityVariabilityPoint) han sido completados por uno y sólo un ActivityBinding (restricción 25, A.2).

Para aclarar un poco estos conceptos y la relación entre ellos se presenta el siguiente ejemplo. Se quiere definir una máquina de estados que contiene un estado, denominado *controlling*, en el que se realiza el control de un actuador. El diseñador ha detectado que se va a querer reutilizar esta máquina de estados en otros componentes del sistema, aunque en cada caso se utilizará un algoritmo de control distinto. Para parametrizar la máquina de estados, el diseñador añade un ActivityVariabilityPoint al StateMachineDefinition y lo asocia a la actividad doActivity del estado *controlling*. A partir de este momento, cuando se cree un nuevo componente de este tipo, en la StateMachine asociada tendrá que crear un ActivityBinding que enlazará el ActivityVariabilityPoint asociado al estado *controlling* con la actividad que quiera que ejecute realmente el componente en particular.

6.5.2 MODELADO DEL COMPORTAMIENTO

En este apartado se describen las características básicas de modelado del comportamiento presentes en V³Studio. Como ya se mencionó al principio de la presente sección, la vista de descripción del comportamiento de un componente en V³Studio está basada en el diagrama de máquinas de estado de UML y mantiene también la semántica que le asigna UML. Como se ha mencionado en el apartado 6.5.1, es la EClass `StateMachineDefinition` la que contiene realmente la máquina de estados que describe el comportamiento de un `ComponentDefinition`. La existencia de una relación de asociación entre ambos conceptos permite que una misma `StateMachineDefinition` pueda ser reutilizada en distintos `ComponentDefinition`, siempre claro que tenga sentido hacerlo, i.e. siempre y cuando la máquina de estados sea compatible con la definición del componente. Las principales características de las máquinas de estados son las siguientes:

- La máquina de estados contiene una región (representada por la EClass `Region`), denominada `topRegion`, en la cual están contenidos todos los estados y las transiciones definidas por la máquina de estados. Al igual que en UML, los estados pueden contener regiones con sus propios estados y transiciones anidadas, de forma que es posible definir macro-estados y regiones ortogonales (AND-regions) [108]. La utilización de máquinas de estados jerárquicas con regiones ortogonales dota de gran expresividad a los modelos y evita, hasta cierto punto, la «explosión de estados» que puede ocurrir cuando se utilizan máquinas de estados no jerárquicas o planas. Los macro-estados y las regiones ortogonales son tratados con más detalle en el apartado 6.5.4.

- La EClass `State` modela cada uno de los estados de la máquina. Como se ha mencionado en el punto anterior, cada `State` puede contener un número indeterminado de regiones (`Region`) internas para modelar macro-estados. Los estados tienen unas referencias (denominadas `entry`, `exit` y `doActivity`) a una serie de algoritmos (actividades) que son ejecutados con la misma secuencia que se establece en UML, i.e. cuando se produce una transición que abandona un estado para entrar en otro primero se ejecuta la actividad `exit` del estado saliente y posteriormente la actividad `entry` del nuevo estado. Estas actividades se definen por separado mediante la vista algorítmica (ver sección 6.6), y como tal, pueden ser reutilizados en distintos puntos. En este caso se ha definido una restricción OCL para evitar que el algoritmo ejecutado sea del tipo pseudo-actividad (ver restricción 21, A.2).

- La EClass `FinalState` modela el estado de la máquina de estados que representa el cese de la ejecución de la misma. Todo componente que alcanza este estado deja

de responder a las peticiones de servicio que pueda recibir del resto del entorno. Este `FinalState` es realmente un estado, ya que es observable desde el exterior, por lo que hereda de la clase `State`. Sin embargo, es necesario añadir una restricción que compruebe que no existen transiciones que salgan de un estado final y que, por tanto, tampoco está definida la actividad `exit` (ver restricción 28).

- La EClass `Pseudostate` modela un subconjunto de los tipos de pseudo-estados que contempla UML, con su mismo significado: en pseudo-estado es un estado no observable desde el exterior pero necesario para especificar algunos aspectos de la máquina de estados. V^{3}Studio contempla la existencia de cuatro tipos: *inicial*, *histórico*, *join* y *fork*. Tanto `State` como `Pseudostate` heredan de la EClass abstracta `Vertex`, que es utilizada para establecer el origen y el destino de una transición (EClass `Transition`). Todos estos tipos de pseudo-estado son tratados en mayor detalle en el apartado 6.5.3.

- La EClass `Transition` modela el arco que define la transición entre estados (concretamente cualquier tipo de `Vertex`) de la máquina de estados. Este concepto tiene varias propiedades que permiten al diseñador controlar y especificar el comportamiento de la transición. Una `Transition` (1) contiene una guarda (EClass `Constraint`) para habilitar el disparo de la transición; (2) tiene una referencia a una actividad (denominada `effect`) que será ejecutada en caso de que finalmente se produzca la transición; (3) contiene un disparador (EClass `Trigger`) para especificar el evento que va a disparar la transición y (4) tiene una propiedad (`TransitionKind`) para que el diseñador seleccione si la transición va a provocar un cambio de estado (transición externa), con la subsecuente ejecución de las actividades `exit` y `entry`, o si por el contrario la transición representa una reacción del estado, en la que no se abandona el estado actual (ver restricción 22).

- El enumerado `TriggerKind` modela las posibles fuentes que pueden provocar la activación (`Trigger`) de una transición. Concretamente, V^{3}Studio contempla las tres siguientes fuentes de activación: (1) `service` modela la recepción de una petición de servicio (invocación de un `Service`); (2) `timeout` modela el paso del tiempo y (3) `activityCompletion` modela la compleción de la actividad principal (`doActivity`) del estado actual en que se encuentra el componente. Cada una de estas posibles fuentes de disparo tiene atributos distintos, por lo que es necesario especificar una restricción OCL para verificar que las propiedades del `Trigger` se ajustan a su tipo (ver restricción 19).

6.5.3 DESCRIPCIÓN DE LOS PSEUDO-ESTADOS MODELADOS

Según afirma UML, un pseudo-estado representa «una abstracción que agrupa diferentes tipos de vértices transitorios de una máquina de estados». Estos pseudo-estados representan estados no observables desde el exterior de la máquina de estados pero que son necesarios para completar en algunas situaciones la semántica de la máquina de estados. UML contempla la existencia de una gran cantidad de pseudo-estados, que son utilizados con diversos fines. Durante el desarrollo de V3Studio se decidió que muchos de estos pseudo-estados no eran necesarios, por lo que se eliminaron del enumerado `PseudostateKind`, que contiene una lista de todos los tipos de pseudo-estados que puede modelar V3Studio. Entre los primeros en desaparecer se encuentran todos los pseudo-estados relativos a la reutilización de sub-máquinas de estado (*terminate, exitPoint, entryPoint*, etc). Los pseudo-estados que finalmente aparecen en V3Studio se pueden clasificar en dos grupos: los que indican por dónde comienza la ejecución (`initialState` y `historyState`) y los que se utilizan cuando existen regiones ortogonales (`joinState` y `forkState`).

V3Studio establece que todas las regiones de una máquina de estados tienen que tener alguno de los dos tipos de pseudo-estados iniciales, lo que permite marcar el punto de comienzo de forma determinista (ver restricciones 23 y 18). La semántica de los tipos de pseudo-estado inicial (`initialState`) e histórico (`historyState`) en V3Studio es la misma que en UML: un pseudo-estado inicial marca el estado en que se inicia una región (siempre que una transición no apunte directamente a un estado interno), y este estado es siempre el mismo, mientras que un pseudo-estado histórico aporta además la capacidad de recordar el estado en que se encontraba la región antes de ser abandonada, y de esta forma es posible volver al último estado en que se encontraba la región cuando se vuelva a entrar en el macro-estado que la contiene. UML define un tercer tipo de pseudo-estado inicial, denominado histórico profundo (`deepHistory`), para recordar todos los estados en que se encontraban todas las regiones internas. Pero esta alternativa, al igual que sucede en muchas herramientas CASE no existe en V3Studio, debido fundamentalmente a la dificultad que entraña su implementación en código.

Por otro lado se encuentran los pseudo-estados que sirven para guiar la entrada y la salida a estados con regiones ortogonales (consultar apartado 6.5.4) en su interior, `joinState` y `forkState`. El primero de ellos, `joinState`, permite ejecutar una salida controlada y sincronizada desde distintos sub-estados. Por tanto, un `joinState` contiene muchas transiciones de entrada y sólo una de salida. Sólo cuando todas las regiones se encuentran en un estado conectado con un `joinState` y se dispara la transición de salida del mismo, se produce la transición. En este momento todos los estados de las regiones internas se abandonan simultáneamente tras tomarse la transición.

El `forkState` es similar, sólo que en lugar de controlar la salida de un macro-estado con regiones ortogonales permite controlar la entrada al mismo. En este caso existen muchas transiciones de salida del `forkState`, tantas como regiones ortogonales tenga el macro-estado de destino, y una transición de entrada, que es la que da origen a la multiplicación del flujo de control en la máquina de estados. Estos dos tipos de pseudo-estado se utilizan exclusivamente cuando se está diseñando una máquina de estados jerárquica, con macro-estados o regiones ortogonales, por lo que tanto `joinState` como `forkState` son tratados en mayor detalle en el siguiente apartado. Dada su semántica particular, se han tenido que definir una serie de restricciones OCL para comprobar que se utilizan correctamente (consultar las restricciones 17, 29, 20, 30, 31 y 24 en A.2).

6.5.4 MACRO-ESTADOS Y REGIONES ORTOGONALES

La vista de comportamiento de V^{3}Studio permite definir macro-estados, i.e., estados que contienen regiones (y por ende otros estados y transiciones). El concepto «macro-estado» fue introducido por Harel en [108] para reducir la «explosión de estados y transiciones» que puede darse en una máquina de estados cuando existen varios estados que comparten una serie de propiedades y transiciones. El uso de macro-estados también mejora la inteligibilidad del diagrama al presentar agrupados y perfectamente separados todos los estados relacionados. Las máquinas de estados que incluyen macro-estados suelen denominarse también jerárquicas, ya que los estados se encuentran anidados en su interior en distintos niveles. Además, el concepto de «macro-estado» no es algo ajeno a la forma de pensar humana, ya que es fácil pensar que un objeto puede estar en el estado «en funcionamiento» o «en error», y cada uno de estos puede a su vez dividirse en otros (como «ejecutando comando», «esperando orden», «configurando», etc).

Generalmente se suele denominar macro-estado a aquel estado que tiene una región embebida y estado con regiones ortogonales a el estado que tiene más de una región embebida. La adición de regiones extras a un macro-estado aumenta aún más, si cabe, el poder de modelado de las máquinas de estados, y permite disminuir el número de elementos necesarios para expresar, en este caso, que un objeto puede tener varias vistas o realidades independientes entre sí, pero necesarias para definir completamente el estado en que se encuentra. Además, al existir un único macro-estado, se obtiene una mejora en la cohesión e inteligibilidad del diagrama, que se hace más intuitivo. Se tiene que tener la precaución, sin embargo, de que todos los estados que aparecen en las regiones ortogonales modelen aspectos independientes de la realidad del objeto en general en UML, componente en el caso de V^{3}Studio. La figura 6.7 presenta estos conceptos de forma gráfica por medio de un modelo (muy simplificado) de una persona. En dicha figura se puede observar también

cómo el uso de regiones ortogonales (marcadas con distintos colores) simplifica el diseño y mejora su inteligibilidad, gracias a que evita la «explosión de estados» necesaria para modelar con una región lo que se hace, en este caso, con dos regiones.

El uso de regiones ortogonales no implica en ningún momento concurrencia en la ejecución. Sólo implica distintos aspectos independientes de una misma realidad. En UML la decisión de ejecutar de forma concurrente o no un estado con regiones ortogonales se deja en manos del diseñador, mientras que en V³Studio las regiones se ejecutan siempre de forma secuencial. En cuanto a la dinámica de ejecución de un macro-estado con regiones ortogonales, V³Studio mantiene la que define UML. En concreto, los macro-estados, como estados normales que son, pueden tener sus propias actividades `entry`, `exit` y `doActivity`, que son ejecutadas en la misma secuencia que define UML. Es decir, cuando se produce una transición se termina el `doActivity` y se ejecuta el `exit` del estado saliente; posteriormente se ejecuta el `effect` de la transición y, por último, el `entry` y el `doActivity` del estado entrante, siempre que existieran y en este orden. En caso de que el estado tenga sub-estados esta cadena se extiende:

- En caso de que se entre en un macro-estado, primero se ejecutan las actividades `entry` de todos los estados, comenzando por el del propio macro-estado y continuando por los de todos los estados anidados (si tuviera más de uno) hasta ejecutar, en última instancia, la actividad `entry` del estado alcanzado por la transición. En cuanto se ejecuta el último `entry` se procede a ejecutar, de nuevo siguiendo el mismo orden anterior, cada una de las actividades `doActivity`. Así como la especificación de la ejecución es muy clara para el caso del macro-estado, la ejecución de las actividades para el caso de los estados anidados contenidos en regiones ortogonales no se especifica, y por tanto se pueden ejecutar en cualquier orden. V³Studio, al igual que hace UML, establece que si el comportamiento depende del orden de ejecución de los sub-estados internos la máquina de estados es semánticamente incorrecta.

- En el caso en que se salga de un macro-estado, el orden de ejecución de las actividades es el contrario del que se ha especificado anteriormente. Es decir, primero se abortan todas las actividades `doActivity` de los sub-estados y posteriormente el del macro-estado. Acto seguido se ejecutan todas las actividades `exit`, de nuevo, de «dentro a fuera». En este momento ya se ha salido del macro-estado y la ejecución de las actividades continúa el orden normal previamente descrito, i.e. se ejecuta el `effect` de la transición y se entraría en el estado alcanzado por la transición.

V³Studio añade una restricción extra que no está presente en UML: todas las regiones tienen que tener, por fuerza, un pseudo-estado (EClass `PseudoState`) de tipo inicial o histórico, que marque el punto por el que se tiene que iniciar la ejecución en la región (ver

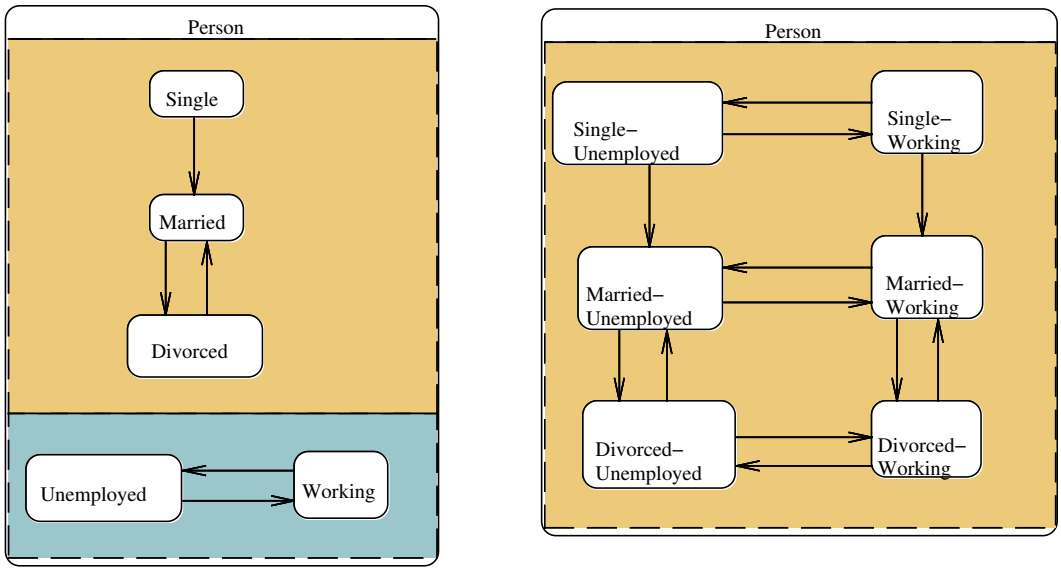


Figura 6.7: Máquina de estados simplificada de una persona. La figura de la izquierda utiliza regiones ortogonales. La figura de la derecha muestra un diagrama equivalente.

restricciones 23 y 18 en A.2). Esta restricción extra evita la creación de máquinas de estados mal formadas en las que, por descuido del diseñador, falte parte de la información necesaria para la correcta ejecución de la máquina de estados. Además, esta restricción aporta flexibilidad al diseñador, ya que permite reducir el número de transiciones salientes de un `forkState` a sólo las que apuntan a estados que no son, a su vez, apuntados por un pseudo-estado `initialState` o `historyState`. Las regiones que no son alcanzadas por una de las transiciones que salen de un `forkState` comienzan su ejecución de forma estándar. La figura 6.8 muestra de forma gráfica dos ejemplos del uso del `forkState`: el de la izquierda muestra cómo no es necesario que existan transiciones a todas las regiones ortogonales y el de la derecha muestra el caso degenerado, en que existe una sóla transición sin `forkState`; este caso se describe más abajo. Todas estas condiciones y restricciones sobre la utilización de los pseudo-estados de tipo joinState se verifican mediante las restricciones OCL 30, 31 y 24 (apartado A.2).

Para mantener la semántica de las máquinas de estado V^{3}Studio sencilla y coherente, se decidió que el `joinState`, pseudo-estado simétrico al `forkState`, también podría prescindir de algunas de las transiciones procedentes de estados contenidos en regiones ortogonales. De esta forma, en V^{3}Studio es posible que tanto un `joinState` como un `forkState` tengan menos transiciones entrantes y salientes, respectivamente, que regiones ortogonales contiene el macro-estado del que provienen o al que alcanzan. En el caso del `joinState`, la falta de transiciones entrantes indica que dicha región ortogonal puede encontrarse en cualquier estado, i.e. el estado en que se encuentra la región no interviene en la decisión de la ejecución de la transición. La figura 6.9 muestra de forma gráfica dos

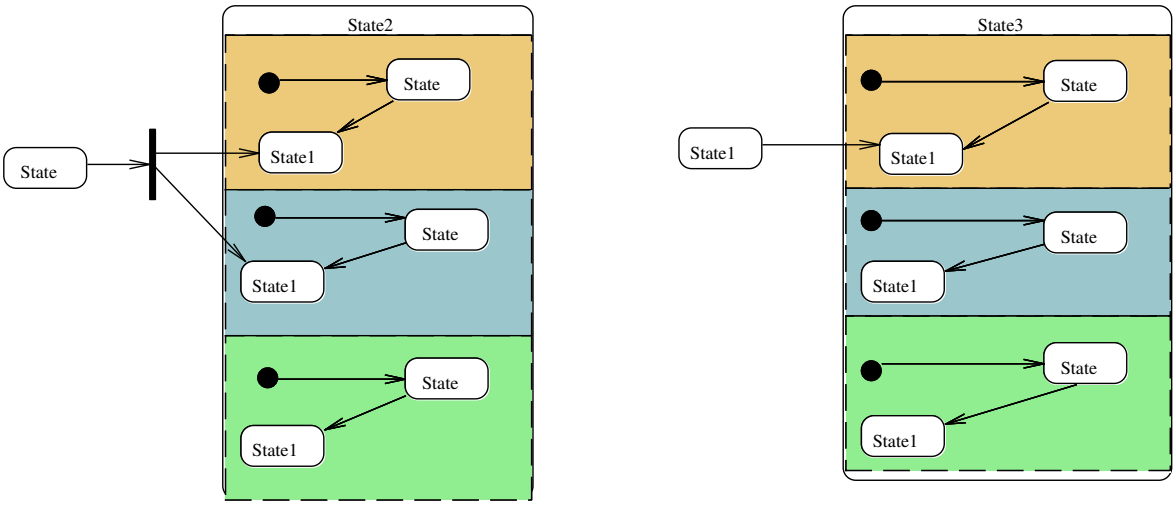


Figura 6.8: Ejemplos de utilización del `forkState`. La figura de la izquierda muestra un ejemplo genérico, mientras que la de la derecha muestra el caso degenerado.

ejemplos del uso del `joinState`, similares a los presentados en el caso del `forkState`: la figura de la izquierda muestra que no es necesario que existan transiciones que provengan de todas las regiones ortogonales y la figura de la derecha muestra el caso degenerado, en que existe una sólo transición sin `joinState`. Todas estas condiciones y restricciones sobre la utilización de los pseudo-estados de tipo joinState se verifican mediante las restricciones OCL 17, 29 y 20 (apartado A.2).

En la versión más degenerada de estas características, ambos tipos de pseudo-estado pueden ser sustituidos por una única transición de entrada o salida a uno de los estados pertenecientes a una de las regiones ortogonales contenidas en el macro-estado. En este caso, la ejecución de la máquina de estados sigue siendo la misma:

- Si la transición entra en el macro-estado entonces la región ortogonal que contiene el estado al que apunta la transición comienza su ejecución por dicho estado. El resto de regiones ortogonales utilizan o bien el pseudo-estado inicial o bien el histórico para comenzar su ejecución.
- Si la transición es saliente entonces todas las regiones ortogonales contenidas en el macro-estado serán abandonadas, independientemente del sub-estado en que se encuentren, en caso de que se finalmente se ejecute la transición.

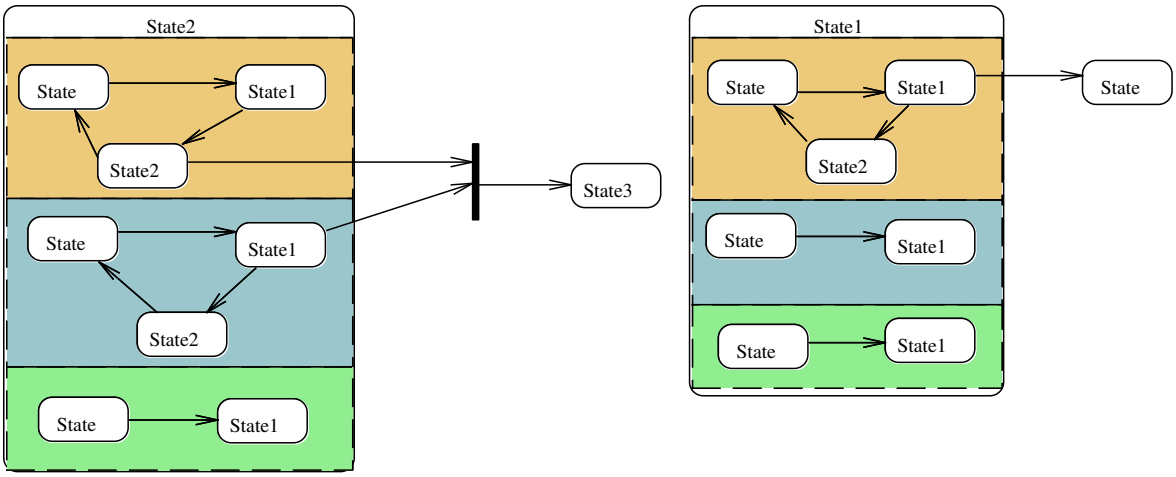


Figura 6.9: Ejemplos de utilización del `joinState`. La figura de la izquierda muestra un ejemplo genérico, mientras que la de la derecha muestra el caso degenerado.

6.6 Vista Algorítmica

La última vista de V^{3}Studio permite al diseñador especificar la secuencia de algoritmos que es ejecutada cuando se activa cada uno de los elementos de la máquina de estados que tienen asociados una actividad: estados (`State`) y transiciones (`Transition`). Esta vista otorga, por tanto, la capacidad de modelado de más bajo nivel, ya que es la más cercana al código final. La figura 6.10 muestra el extracto del meta-modelo de V^{3}Studio que contiene las clases que definen esta vista, mientras que en el apéndice A.3 están recogidas todas las restricciones OCL relacionadas con ella.

Como ya se comentó, esta vista está basada en los diagramas de actividades de UML 2. La razón que justifica su adopción es exactamente la misma por la que se adoptó la máquina de estados: esta parte de UML 2 se ajusta a las necesidades de modelado de V^{3}Studio, por lo que se decidió adaptar el diagrama de actividades en vez de partir de cero y realizar un diseño alternativo. V^{3}Studio también adopta la semántica de redes de Petri y en el paso de *tokens* asociada a los diagramas de actividades en UML 2.

La vista algorítmica de V^{3}Studio está organizada alrededor de la clase base abstracta `Activity`, de la que heredan el resto de clases que forman la vista. En el contexto de la definición de máquinas de estados parametrizadas, la EClass abstracta `Activity` (y todos sus descendientes) permite definir el algoritmo que se va a ejecutar en todas las instancias de la máquina de estados (`StateMachine`), en contraposición a la clase `ActivityVariabilityPoint`, que sirve para establecer puntos de variación en la definición (`StateMachineDefinition`), que deben ser completados en cada instancia de la definición de la máquina de estados (`StateMachine`).

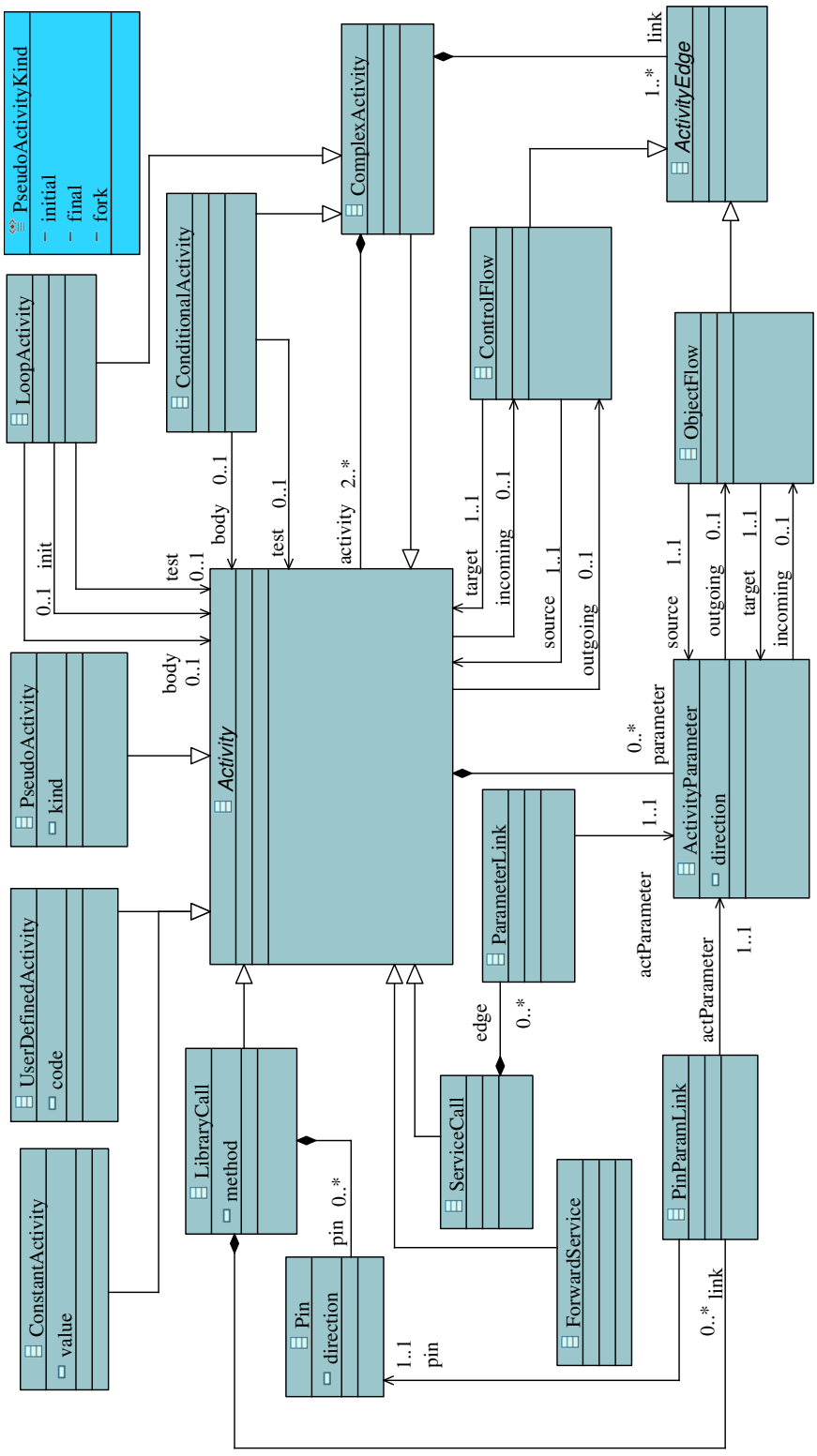


Figura 6.10: Extracto del meta-modelo de V³Studio que muestra la descripción algorítmica

Esta sección está organizada en cuatro apartados, en los que se describe con detalle cada una de las partes de esta vista algorítmica. En concreto, el primer apartado describe las características principales de la EClass abstracta que forma el núcleo de la vista: la EClass `Activity`. Posteriormente, los dos siguientes apartados presentan los dos tipos de actividades de que dispone el diseñador para describir los algoritmos en V^{3}Studio: actividades simples y complejas. Por último, se detalla un mecanismo para abordar la variabilidad en la implementación de las actividades, debido a la utilización de distintas librerías y productos COTS.

6.6.1 Descripción general de la EClass Activity

La super-clase abstracta `Activity` define la infraestructura común que es utilizada por el resto de actividades de V^{3}Studio. Esta infraestructura, descrita en el enumerado que se expone a continuación, proporciona los conceptos básicos necesarios para que el resto de actividades que forma V^{3}Studio puedan modelar el nivel de detalle propio de esta vista. V^{3}Studio no distingue, como hace UML, entre acción y actividad; en V^{3}Studio todo son actividades. Tradicionalmente, el concepto «acción» hacía referencia a una ejecución instantánea e ininterrumpible. Para todo lo demás se utilizaba «actividad». Aunque esta diferencia ya no existe en UML 2, en V^{3}Studio se ha querido eliminar esta posible fuente de confusión, por lo que no existen las acciones; en V^{3}Studio todos los elementos que se utilizan para describir un algoritmo son actividades y no existe ninguna connotación temporal en cuanto a la duración de las mismas. Se supone, eso sí, que las actividades `effect`, `onEntry` y `onExit` tienen una semántica de «ejecución hasta compleción», por lo que el diseñador tiene que tomar la precaución de diseñarlas lo más cortas posibles, puesto que el componente se encuentra en un estado inestable mientras las está ejecutando.

`ActivityParameter`. Modela la existencia de parámetros de entrada y salida, que proporcionan los datos necesarios a/desde el algoritmo. El diseñador puede especificar para cada parámetro (1) su dirección (mediante el enumerado `ParameterKind`, de forma similar a como se realiza en la vista arquitectónica, apartado 6.4.3) y (2) el tipo de dato del parámetro (de los definidos globalmente mediante la vista arquitectónica). Un `ActivityParameter` indica, dentro de la semántica de redes de Petri asociada a los diagramas de actividad, el tipo de *token* que lo atraviesa y la dirección de dicho *token*.

`ControlFlow`. Sirve para modelar, de forma explícita, el flujo de control que guía la ejecución secuencial de un conjunto de actividades. Los conceptos relacionados con el flujo de control en las actividades se exponen más adelante en el apartado 6.6.3,

cuando se aborde el tema de las actividades complejas y los dos tipos de flujos: control y datos.

V3Studio clasifica las actividades en simples y complejas. Las actividades simples son las actividades atómicas que contienen las unidades fundamentales de descripción de un algoritmo. Actualmente, V3Studio modela cuatro tipos de actividades simples: la invocación de un servicio ofrecido por un componente, la invocación de una función de una librería, la redirección de una petición de servicio (por parte de un componente complejo) y un bloque de código genérico (a rellenar por el usuario). Todas estas actividades simples son descritas con mayor detalle en el siguiente apartado. Las actividades complejas están formadas por la unión de actividades simples o de otras actividades complejas y especifican el flujo de control y datos entre ellas. V3Studio diferencia dos tipos de actividades complejas en particular: bucles y actividades de selección. Las actividades complejas son descritas con mayor detalle en el apartado 6.6.3.

6.6.2 ACTIVIDADES ATÓMICAS O SIMPLES

Las actividades atómicas o simples son, como su nombre indica, aquellas actividades que no pueden ser subdivididas en otras más pequeñas. Estas actividades pueden ser directamente utilizadas para describir la actividad que realiza un estado (`State`) o una transición (`Transition`) o pueden agruparse para formar una actividad compleja, tal y como se expone en el siguiente apartado. V3Studio proporciona varios tipos de actividades simples, con los que modela las mayor parte de situaciones que pueden aparecer a la hora de diseñar un algoritmo:

ServiceCall: modela la invocación de uno de los servicios requeridos por el componente por uno de sus puertos. Por tanto, una actividad de este tipo tiene sendas referencias al servicio que debe invocar y al puerto en que se encuentra. Además, y puesto que lo más común es que el servicio tenga una serie de parámetros, un `ServiceCall` contiene una serie de `ParameterLink` que sirven para enlazar los parámetros de la actividad con los parámetros del servicio que se está invocando. La restricciones 43 y 46 del apartado A.3 comprueban que los tipos de datos de los parámetros conectados concuerdan y que no se deja ningún parámetro sin conectar entre la actividad y el servicio.

LibraryCall: esta actividad modela la invocación de una función o procedimiento definido en una librería o *API* (Interfaz de Programación de Aplicaciones). Es común que en dominios tan asentados como la robótica existan librerías comerciales o de libre distribución que provean las funciones más utilizadas en el dominio. La actividad

`LibraryCall` facilita la reutilización de estas librerías y dispone de un mecanismo para adaptar las convenciones de llamada, como puede ser el orden y el tipo de los argumentos. Esta característica de la vista algorítmica se trata con detalle en la sección 6.6.5. La restricciones 44 y 45 comprueban que los tipos de datos de los parámetros y los pines conectados concuerdan y que no se deja ningún parámetro sin conectar a un pin.

ForwardService: esta actividad modela el reenvío, por parte de un componente complejo, de una petición de servicio al componente interno que va a procesarla finalmente. En el apartado 6.4.2 se dijo que un componente complejo no añade funcionalidad propia al sistema, sino que sólo encapsula y controla el acceso a sus componentes internos. La actividad `ForwardService` cumple con esta misión y, por tanto, sólo puede estar definida en el contexto de la máquina de estados asociada a un componente complejo; de hecho, es la única actividad permitida en este caso (consultar restricción 16).

UserDefinedActivity: esta actividad modela, de forma general, un bloque de código que va a ser rellenado por el usuario y que no se corresponde con ninguna de las actividades definidas en los puntos anteriores. Dentro de los objetivos de esta Tesis Doctoral de generar el esqueleto de las aplicaciones, las actividades anteriormente descritas tienen un papel claramente diferenciado y V^{3}Studio aprovecha esta diferencia para generar las partes adecuadas del esqueleto de la aplicación. Una `UserDefinedActivity` queda, pues, como último recurso del diseñador, cuando ninguna de las actividades anteriores sirve para describir el algoritmo. Es, por tanto, la actividad que más veces aparece en un algoritmo.

PseudoActivity: como en el caso de las máquinas de estado, los diagramas de actividad necesitan un tipo especial de actividad, que no representa la ejecución de ningún algoritmo, pero que es necesaria para definir correctamente el diagrama de actividad. Puesto que estas actividades no son realmente actividades, las restricciones 39, 33, 35, 36 y 32 del apartado A.3 se han definido para detectar un uso incorrecto de las mismas. Estas actividades se utilizan en el contexto de una actividad compleja, por lo que su explicación se pospone hasta el apartado 6.6.4, tras describir las actividades complejas en el contexto de V^{3}Studio.

ConstantActivity: una actividad constante modela una actividad que únicamente produce un valor de salida preestablecido por el desarrollador (ver restricción 47). Es labor del diseñador comprobar que el valor asignado a la actividad constante forma parte del rango de valores posibles que puede adoptar, según el tipo de dato que tenga asignado el parámetro (`ActivityParameter`) de la actividad.

UML define muchos más tipos de actividades (más de veinticinco), lo que lo dota de un mayor poder expresivo. Sin embargo, se reitera, el objetivo tras el diseño de V^{3}Studio no es rivalizar con UML, sino desarrollar un lenguaje sencillo, que sea fácil de comprender y utilizar, pero que a la vez tenga la suficiente potencia expresiva para ser útil. V^{3}Studio no persigue ser capaz de describir con todo detalle una aplicación. Para esto existe UML. Sin embargo, no se descarta ampliar esta parte de V^{3}Studio diseñando nuevos tipos de actividades simples para mejorar la capacidad de modelado.

6.6.3 ACTIVIDADES COMPLEJAS. FLUJO DE CONTROL

Las actividades complejas, representadas por la EClass `ComplexActivity`, permiten agrupar otras actividades, ya sean simples o complejas, y controlar el flujo de ejecución de las mismas. Como en el caso de la máquina de estados, existe el tipo especial `PseudoActivity` que, sin ser una actividad propiamente, permite especificar algunos aspectos de la ejecución de la actividad que no están directamente relacionados con la ejecución de código asociado. Como el resto de actividades, un `ComplexActivity` puede tener definidos cualquier número de parámetros (`ActivityParameter`) de entrada o salida. Sin embargo, en este caso estos parámetros tienen que estar directamente involucrados en la ejecución del algoritmo, por lo que es necesario comprobar que todos ellos, ya sean de entrada o de salida, están conectados (ver restricción 41) con parámetros compatibles, i.e. que comparten el mismo tipo de dato (ver restricción 48).

V^{3}Studio añade, además, dos clases que heredan de `ComplexActivity` y que permiten implementar las construcciones típicas de los lenguajes imperativos. Estas clases presuponen la existencia de algún tipo de dato booleano, que es retornado por las actividades que se ejecutan para comprobar el estado de la guarda.

ConditionalActivity: representa una sentencia de selección, en la que se ejecuta una actividad (representada por la referencia `body` en la figura 6.10) dependiendo de si se cumple una determinada condición (representado por la referencia a la actividad `test`). Es necesario comprobar, para que el diagrama esté bien formado, que ninguna de las referencias anteriores apunta a una pseudo-actividad (consultar restricción 38) y que dichas referencias pertenecen al `ConditionalActivity`, i.e., están contenidas en la actividad condicional (ver restricción 37).

LoopActivity: representa una sentencia de repetición genérica, que tiene una parte de inicialización (representada por la referencia a la actividad `init`) y una referencia a una actividad (`body`) que se repite mientras que la condición (referencia a la actividad `test`) sea cierta. La restricción 40 del apartado A.3 comprueba que ninguna de las

referencias anteriores apunta a una pseudo-actividad, mientras que la restricción 34 comprueba que las actividades están contenidas en el `LoopActivity`.

Como ya se ha mencionado, una `ComplexActivity` también puede controlar el flujo de actividades que ejecuta. Este control lo realiza mediante las clases que descienden de la EClass abstracta `ActivityEdge`. Esta clase permite especificar, de forma genérica, la secuencia de actividades (simples o complejas) que realiza la actividad compleja que las contiene. Las dos clases que heredan de `ActivityEdge` permiten particularizar y describir los dos flujos principales: flujo de control (modelado mediante la EClass `ControlFlow`) y flujo de datos (modelado mediante la EClass `ObjectFlow`).

Un `ObjectFlow` establece una relación de concordancia entre los parámetros (modelados mediante la EClass `ActivityParameter`) de entrada y salida de dos actividades. Lógicamente, es necesario definir una restricción OCL para asegurar que los tipos de datos y la dirección de los `ActivityParameter` que se conectan son compatibles (ver restricción 48). Sin embargo, un `ObjectFlow` no fuerza a que la ejecución de todas las actividades que conecta tenga que ser secuencial. Por ejemplo, es posible definir una actividad (ACT^A) que ejecute un procedimiento que produzca como resultados dos valores, que se utilizan como parámetros de entrada de otras dos actividades (ACT^B y ACT^C). En este caso, el flujo de datos no permite establecer la secuencia; tanto $ACT^A \longrightarrow ACT^B$ como $ACT^A \longrightarrow ACT^C$ son perfectamente plausibles.

La EClass `ControlFlow` permite deshacer esta falta de unicidad en el estableciemiento del flujo de control de la actividad que se puede producir si se especifica únicamente el flujo de datos. Como ya se ha expuesto, un `ControlFlow` enlaza directamente actividades (EClass abstracta `Activity`) para establecer el flujo de control del algoritmo. Por tanto, para que un diagrama de actividad sea correcto es necesario que el flujo de control esté correctamente definido. La restricción 42 se encarga de comprobar esta condición (salvo en el caso del `fork`, ya que sólo clona valores y, como se expone en el apartado 6.6.4, sólo utiliza flujo de datos).

Como se desprende de lo expuesto a lo largo de este apartado, en ningún momento se ha contemplado la posibilidad de definir actividades que pudieran ser ejecutadas de forma paralela, como es típico encontrar en los diagramas de actividades. Esto es así debido a que V^{3}Studio no contempla directamente la ejecución en paralelo de actividades porque las actividades representan la unidad de menor nivel en V^{3}Studio, ya que están contenidas en otras unidades más grandes, como es el propio componente, que son las que establecen la política de concurrencia de la aplicación.

6.6.4 DESCRIPCIÓN DE LOS TIPOS DE PSEUDO-ACTIVIDAD

Una pseudo-actividad, como ya se ha comentado, no define realmente un algoritmo, sino que sólo ayuda a completar su definición. Debido a su naturaleza, las pseudo-actividades no pueden definir parámetros, ya que esto supondría un trasiego de datos que realmente no existe. Esta condición es comprobada por la restricción OCL 39. Por esta misma razón tampoco se puede utilizar una pseudo-actividad como la actividad que se ejecuta en un estado o en una transición (consultar las restricciones 40, 38 y 49).

Para mantener la coherencia con la definición de pseudo-actividades (EClass `PseudoActivity`) que se ha realizado en la vista de comportamiento, se decidió que el tipo de pseudo-actividad estaría definido mediante el uso del tipo enumerado `PseudoActivityKind`. Esto contrasta con el enfoque que sigue UML, en el que las pseudo-actividades están modeladas mediante clases distintas, que heredan de la misma clase común `ControlNode`. El enumerado `PseudoActivityKind` permite definir los siguientes tipos de pseudo-actividades:

Initial: marca la actividad por la que comienza la ejecución del algoritmo. A diferencia de lo que sucede en la vista de comportamiento, no es necesario que exista una pseudo-actividad de este tipo en cada diagrama. Esta pseudo-actividad sólo es necesaria en caso de que se esté definiendo una actividad compleja (`ComplexActivity`), ya que es necesario indicar cuál es la primera actividad que se va a ejecutar en este caso. La restricción 36 comprueba que una pseudo-actividad de este tipo está correctamente definida, según se acaba de indicar.

Final: establece el final de la ejecución del algoritmo. Una vez que la actividad ha acabado, lo valores de los parámetros de vuelta se colocan en los respectivos `ActivityParameter` (si los hubiera). Como en el caso anterior, esta pseudo-actividad sólo es necesaria en caso de que se esté definiendo una actividad compleja. La restricción 32 comprueba que una pseudo-actividad de este tipo está correctamente definida, según se acaba de indicar.

Fork: esta pseudo-actividad permite clonar el valor de un parámetro de salida, de forma que pueda ser utilizado en distintos parámetros de entrada de diversas actividades. Mediante el uso del `Fork` es posible utilizar un mismo valor, obtenido como parámetro de salida de una actividad ejecutada anteriormente, como parámetro de entrada en distintas actividades. Esta pseudo-actividad controla únicamente el flujo de datos del algoritmo, por lo que se han definido sendas restricciones OCL para controlar esta condición (restricciones 33 y 35, apartado A.3).

Antes de acabar el presente apartado hay que resaltar que la vista algorítmica de V^{3}Studio se ha diseñado para que estos modelos sean finalmente ejecutados de forma secuencial. Corresponde a las otras vistas de «mayor nivel» decidir si finalmente el componente se va a ejecutar en uno o varios hilos y si va a haber partes que se ejecuten de forma concurrente o no. Partiendo de esta premisa, es lógico que no se haya añadido un pseudo-estado de tipo *join* para establecer la convergencia de flujos. Como no se contempla concurrencia a este nivel, cualquier flujo concurrente puede convertirse en secuencial. El caso del `fork` es una excepción, ya que lo que se multiplica es el dato que se pasa a las actividades posteriores; pero nunca implica ejecución concurrente. Por esto, además, se fuerza que los arcos que llegan y salen de un `fork` sean únicamente de tipo `ObjectFlow`, para especificar sólo la interconexión entre datos (restricción 35).

6.6.5 VARIABILIDAD EN LA IMPLEMENTACIÓN

Actualmente, el mercado ofrece una gran cantidad de productos COTS que proporcionan parte de la funcionalidad que es típicamente requerida cuando se desarrolla una nueva aplicación, sobre todo en dominios maduros. Sin embargo, generalmente estos COTS utilizan sus propios tipos de datos, sus propias convenciones de invocación de funciones y sus propios mecanismos para el manejo de errores. Estas diferencias provocan que, a menudo, sea difícil utilizar distintos COTS conjuntamente. V^{3}Studio permite a los diseñadores utilizar la funcionalidad provista por distintos COTS mediante un mecanismo para encapsular las funciones heterogéneas de estas librerías y construir funciones homogéneas, y por ende unidades de implementación conectables. Estas unidades están representadas por la actividad `LibraryCall`.

La actividad `LibraryCall` representa un bloque funcional que es importado de alguno de los productos COTS que quieren integrarse en el sistema. Esta actividad contiene un conjunto de `Pin` que representan los parámetros de entrada y salida de la función que encapsula el `LibraryCall`. El mecanismo de encapsulación y adaptación de la función lo realiza la EClass `PinParamLink`, que conecta cada uno de los `Pin` definidos para la función con uno de los `ActivityParameter` que define la actividad. El concepto `PinParamLink` juega, por tanto, el papel del patrón de diseño *Adapter* [96][9]. Para llevar a cabo su objetivo, el `PinParamLink` debe realizar las siguientes acciones:

1. Realizar la correspondiente conversión entre los tipos de datos de las dos actividades que se unen.

[9]Según Gamma, este patrón «convierte la interfaz de una clase en otra interfaz que el cliente espera. *Adapter* permite a las clases trabajar juntas, lo que de otra manera no podrían hacerlo debido a sus interfaces incompatibles.»

2. Invocar la función de la librería utilizando la convención adecuada y colocando los parámetros en la posición correcta.

3. Interpretar y manejar los posibles códigos de error y excepciones que pudieran surgir durante la ejecución de la función.

Obviamente, este mecanismo lleva asociado una serie de restricciones OCL para asegurar la coherencia de los tipos de datos entre los parámetros y que ningún parámetro ha quedado por conectar (consultar restricciones 45 y 44). Actualmente, V3Studio no proporciona una implementación completa de este mecanismo de variabilidad en la última etapa de desarrollo (la transformación a texto), debido a la complejidad de su desarrollo. Sin embargo, sí que era un objetivo importante que esta variabilidad estuviera recogida desde las primeras fases de desarrollo (los modelos), objetivo que es cubierto mediante el `PinParamLink`.

6.7 OTRAS HERRAMIENTAS ALTERNATIVAS

FINALMENTE, en esta sección se describen otras herramientas existentes actualmente que permiten, a semejanza de la herramienta V3Studio presentada en este capítulo, desarrollar también sistemas basados en componentes. Para poder realizar la comparación con V3Studio, se han seleccionado únicamente aquellas herramientas que están basadas en Eclipse y que utilizan los plug-ins básicos que soportan el desarrollo MDE en Eclipse.

El lenguaje de análisis y diseño arquitectónico (*Architecture Analysis & Design Language*, **AADL**)[10] [86, 87] ha sido estandarizado por la *International Society for Automotive Engineers* (**SAE**) en el año 2.004 en colaboración con el SEI. AADL es un lenguaje textual y gráfico que permite diseñar la arquitectura hardware y software de los sistemas de tiempo-real y analizar su rendimiento en condiciones críticas. AADL fue diseñado expresamente para el sector aero-espacial, aviónica y para la industria de automatización. El lenguaje describe la estructura de dichos sistemas por medio de ensamblajes de componentes software, que son posteriormente traducidos a la plataforma de ejecución.

AADL permite describir cómo interactúan entre sí los distintos componentes que integran el sistema, respondiendo a cuestiones como la forma en que se conectan las entradas y salidas de datos o cómo se destinan los componentes software de la aplicación a la plataforma de ejecución. El lenguaje puede describir también el comportamiento dinámico

[10] `http://www.aadl.info/`

de la arquitectura de ejecución gracias al soporte que proporciona para los conceptos de «modos operativos» y «transición entre modos». Además, AADL tiene un diseño extensible que permite el desarrollo de nuevas herramientas de análisis. AADL se basa en la amplia experiencia adquirida por el SEI durante los años en que diseñó distintos ADLs, como ACME o UniCon.

Cadena/CALM[11] [58] es un entorno de desarrollo para sistemas basados en componentes que hace énfasis en el desarrollo de líneas de producto software que hacen uso de un *middleware* de comunicación para lograr independencia de la plataforma. Cadena proporciona un entorno gráfico para la creación, manipulación, aplicación y transformación de modelos escritos en el lenguaje CALM. CALM es un lenguaje de descripción de arquitecturas que permite especificar y forzar una variedad de restricciones arquitectónicas que son relevantes para el desarrollo de grandes sistemas y líneas de producto software.

Cadena proporciona distintas herramientas y vistas a los arquitectos del sistema, los desarrolladores de infraestructuras y desarrolladores del sistema. Estas herramientas les permiten (1) definir entornos de modelado para los modelos de componentes más utilizados, e.g. Enterprise Java Beans, .NET, CORBA Component Model, etc.; (2) definir sus propios modelos de componentes, ajustados a las necesidades concretas del domino o del *middleware* utilizado; (3) combinar y extender múltiples modelos de componentes en un único sistema; (4) ampliar la funcionalidad del entorno mediante la adición de nuevos plug-ins al entorno de desarrollo Eclipse.

Cadena/CALM ha sido descartada porque es una herramienta que está pensada, principalmente, para el desarrollo de líneas de producto. Cadena sigue un enfoque más bien integrador, que toma los componentes como algo que ya existe, con vistas a integrarlo en el producto final y no permite la descripción del «contenido» del componente (modo caja blanca).

openArchitectureWare (**oAW**)[12] es, según su página web, una «herramienta para generar herramientas MDE/MDA» de forma rápida. Por tanto, oAW tiene que proporcionar soporte para llevar a cabo todas y cada una de las etapas que forman el proceso de desarrollo MDE; oAW soporta el análisis sintáctico de modelos, dispone de un lenguaje para transformar y comprobar que los modelos son correctos así como para generar código a partir de ellos. oAW proporciona soporte para trabajar con modelos UML 2, XMI e incluso JavaBeans. El corazón de la herramienta está formado por una secuenciador de procesos que

[11] http://www.cadena.projects.cis.ksu.edu/
[12] http://www.openarchitectureware.org

permite la definición de procesos de generación y transformación y su aplicación a distintos modelos. Entre los procesos que se suministran con la herramienta destacan los procesos de lectura e instanciación de modelos, comprobación de restricciones, transformaciones de modelos y transformaciones a código.

El lenguaje Kernel Metamodelling (**Kermeta**)[13] [157] ha sido desarrollado por el grupo Triskell como un lenguaje experimental para especificar no sólo la estructura sino también el comportamiento de un meta-modelo. Kermeta es un lenguaje imperativo, orientado a objetos (con herencia múltiple), orientado a modelos (conceptos como asociación y composición son entidades de primer nivel), funcional (incluye un subconjunto de expresiones del *lambda-calculus*), fuertemente tipado y reflexivo. Estas características hacen de Kermeta una solución integrada, que proporciona un entorno uniforme, de forma que no se tengan que manejar diversos programas y lenguajes diferentes para realizar cada una de las tareas involucradas en el desarrollo MDE.

Kermeta es un lenguaje académico y experimental, desarrollado principalmente para especificar el comportamiento de los elementos de un meta-modelo y para abordar todas las operaciones que se pueden realizar con los modelos y meta-modelos (creación, transformaciones, entretejido, etc). Y todo esto con una sintaxis única. El problema es que determinadas operaciones (como la definición de una transformación) se expresan mejor utilizando un lenguaje mixto declarativo/imperativo. Además, Kermeta no se ajusta expresamente al estándar de transformación de modelos QVT.

A la vista de las soluciones alternativas que se han propuesto, se concluye que, en principio se podría haber utilizado cualquiera de las herramientas actualmente disponibles (especialmente oAW o AADL) para llevar a cabo el desarrollo de V3Studio, aunque todas ellas están evolucionando muy rápido y constantemente. Sin embargo, se decidió no utilizar ninguna de estas soluciones integradas, sino desarrollar el meta-modelo y las transformaciones «desde cero», utilizando las herramientas básicas, para de esta forma mejorar el conocimiento del grupo DSIE de las bases que sustentan MDE. Es posible, sin embargo, que trabajos futuros hagan uso de alguna de estas herramientas para repetir el desarrollo de V3Studio. Particularmente, AADL parece la más madura, flexible y completa de todas ellas, ya que, además de ser un estándar, proporciona una serie de herramientas que permiten analizar el comportamiento de los modelos que se diseñan con ella.

[13] `http://www.kermeta.org`

CAPÍTULO 7

TRANSFORMACIÓN V^{3}STUDIO A UML: DE COMPONENTES A OBJETOS

EN ESTE capítulo se describe la transformación de modelos desarrollada para convertir un modelo diseñado con V^{3}Studio en un modelo UML. Esta transformación intermedia entre ambos modelos es necesaria, ya que el modelo de componentes desarrollado con V^{3}Studio tiene un nivel de abstracción muy alejado de la implementación final en código, dado que los conceptos que en él se modelan no están presentes directamente en ningún lenguaje de programación orientado a objetos. Además, el modelo UML facilita la consecución de otro de los objetivos de la Tesis Doctoral: ser capaces de generar distintas implementaciones del modelo V^{3}Studio en distintos lenguajes de programación e incluso utilizando distintos frameworks robóticos (como los descritos en el apartado 5.3).

En este capítulo se describe no sólo la transformación de cada una de las vistas de V^{3}Studio a distintos diagramas de UML, sino también la implementación de diversas operaciones derivadas de la semántica asociada a los conceptos de V^{3}Studio (como por ejemplo, la operación para enlazar puertos). La transformación que se describe en este capítulo utiliza los diagramas y conceptos de UML más cercanos a los conceptos de los lenguajes orientados a objetos, i.e. clases, propiedades, acciones, etc, prescindiendo del resto de diagramas (casos de uso, colaboraciones, despliegue, etc). Es decir, UML se utiliza como lenguaje de descripción de la implementación y no como lenguaje de modelado (ése es V^{3}Studio). Para guiar y facilitar la lectura y el seguimiento de la transformación se propone un sencillo ejemplo basado en un componente complejo formado por la unión de dos componentes simples.

7.1 INTRODUCCIÓN Y JUSTIFICACIÓN

FRÖHLICH realizó en 1.999 un estudio [93] sobre la viabilidad de la utilización de los lenguajes orientados a objetos para implementar la funcionalidad básica que se puede esperar de un lenguaje orientado a componentes. Las conclusiones del estudio son que los lenguajes orientados a objetos no se pueden utilizar directamente, ya que no disponen de los mecanismos básicos que requeriría un lenguaje orientado a componentes, como es por ejemplo el tratamiento de mensajes como entidades de primer nivel [94]. Sin embargo, y aunque se estuviera de acuerdo con las conclusiones de este estudio, la utilización de un hipotético lenguaje realmente orientado a componentes traería otros problemas insalvables en algunos dominios maduros, como es el caso de la robótica. En estos dominios existen un gran número de librerías que proporcionan la funcionalidad típica y básica de las aplicaciones del dominio, y que posiblemente no serían utilizables. Por tanto, el objetivo de la transformación que se describe en este capítulo es:

> *Utilizar diversos patrones de diseño para completar aquellas características que son necesarias para que un diseño orientado a objetos cumpla con los requisitos de un sistema realmente orientado a componentes, y utilizar el desarrollo basado en modelos para supervisar y controlar la generación de código que cumpla con estas características.*

La falta de un lenguaje completamente orientado a componentes y de amplia penetración en el mercado ha evitado el éxito al que este paradigma parecía estar predestinado. Aunque existen algunos lenguajes de este tipo (como Component Pascal o Lagoona, consultar apartado 3.2.5), la mayor parte de ellos se han quedado en meros experimentos académicos o aplicados a dominios muy concretos (por ejemplo NesC). Hasta que este lenguaje sea desarrollado, la utilización del paradigma de orientación a objetos parece una forma natural de modelar e implementar componentes [116].

Como ya se ha mencionado, la transformación de un modelo V3Studio a un modelo UML es necesaria para reducir el nivel de abstracción del modelo de componentes V3Studio, ya que los conceptos que éste modela no están presentes directamente en ningún lenguaje de programación orientado a objetos. Por tanto, se hace necesario desarrollar una transformación de modelos que permita reducir la distancia semántica existente entre los conceptos presentes en V3Studio (principalmente los conceptos de CBD componente, conector y puerto) y los conceptos presentes en los lenguajes de programación orientados a objetos (clases, herencia, composición, polimorfismo, etc). Para ello, en esta sección se presenta y describe una posible implementación de los conceptos en que se basa CBD utilizando la tecnología de orientación a objetos.

Esta transformación de modelos entre V^{3}Studio y UML cumple, además, con otro de los objetivos de esta Tesis Doctoral: ser capaces de generar distintas implementaciones en distintos lenguajes. Puesto que UML es independiente del lenguaje de programación seleccionado para llevar a cabo la implementación final en código, el modelo UML que se obtiene como resultado de esta transformación puede ser reutilizado para implementar el sistema en otro lenguaje de programación. Además, UML es un lenguaje muy difundido y utilizado por la comunidad de desarrollo software, lo cuál ha propiciado el desarrollo de numerosas herramientas para analizar los diagramas (por ejemplo, cálculo de métricas a partir del diagrama de clases) y simular el comportamiento (por ejemplo, análisis del comportamiento temporal). Gracias a la transformación a un modelo UML se podrá hacer uso de estas herramientas tanto para mejorar el diseño de la transformación como para estudiar el comportamiento y las características de un modelo V^{3}Studio en particular.

Es incluso posible, como se comentará en el capítulo de conclusiones, desarrollar otras transformaciones de modelos que permitan la utilización de algunos de los frameworks de robótica que existen en la actualidad. Esta transformación redundaría en una mejora tanto en el tiempo de desarrollo de la aplicación final como en su fiabilidad, ya que la mayoría de estos frameworks tienen varios años de existencia, han sido muy probados y están muy depurados.

La transformación que se describe a lo largo del capítulo ha sido diseñada con cuatro ideas principales en mente: (1) mantenerla simple siempre que sea posible (siguiendo el principio KISS, *Keep It Simple, Stupid!*); V^{3}Studio está pensado para el dominio concreto de la robótica, con sus características propias; si se requiere un meta-modelo de propósito general se puede recurrir a UML; (2) mantener la encapsulación de cada una de las partes; (3) seguir el principio de diseño «programación contra una interfaz, no contra una implementación» [96]; y (4) mantenerse fiel a la definición adoptada de los conceptos CBD, especialmente componente y puerto. Estos principios son los que han guiado todas las decisiones de diseño que han desembocado en la transformación de los conceptos CBD de V^{3}Studio a UML que se expone a lo largo del presente capítulo.

Además de la transformación entre modelos V^{3}Studio y UML, se han desarrollado otras transformaciones auxiliares de UML a UML que, aunque no se describen en este documento, son necesarias para poder mostrar los diagramas de clases, máquinas de estados y actividades que aparecen a lo largo de este capítulo. Estas transformaciones eliminan toda la información que no se puede mostrar en un diagrama determinado, e.g. las actividades en un diagrama de clases, y adaptan el modelo de entrada a las necesidades de la herramienta de visualización, el plug-in de Eclipse *UML2Tools*.

Este capítulo está organizado en cinco secciones más. La sección 7.2 describe de forma general la transformación de modelos, enumerando las principales decisiones de diseño

que se han tomado para convertir componentes en objetos y justificando la elección de los diagramas UML utilizados. Las tras secciones siguientes presentan en detalle la descripción de la transformación de cada una de las vistas V³Studio, junto con numerosos diagramas de clases, máquinas de estados y actividades que se generan a lo largo de la transformación. Por último, la sección 7.6 presenta una estadística de la transformación del modelo V³Studio del ejemplo que se describe a continuación, con el objetivo de plasmar con números el tamaño del modelo UML obtenido finalmente.

7.1.1 EJEMPLO DE APLICACIÓN DE LA TRANSFORMACIÓN

Puesto que en este capítulo se describe de forma genérica el proceso de transformación de modelos V³Studio → UML, en la figura 7.1 se presenta un sencillo ejemplo que va a ser utilizado a largo del capítulo para ilustrar la transformación y facilitar su seguimiento. Este ejemplo es un extracto simplificado del caso de estudio que se presenta en la sección 8.3. Como puede observarse en la figura, el ejemplo está formado por un componente complejo (`ComplexComponentDefinition`), denominado *XYZ_Robot*, que contiene dos componentes simples (`SimpleComponentDefinition`), denominados *HAL* y *SC* respectivamente.

El ejemplo modela, de forma simplificada, la estructura de componentes que podría formar la aplicación de control de uno de los ejes de un robot cartesiano (representado por el componente complejo *XYZ_Robot*). La arquitectura de la aplicación se ha obtenido a partir de la especificación de los componentes del CCAS que establece ACRoSeT (ver sección 5.4), y consta de un componente que abstrae la comunicación con el hardware (*HAL*) y de otro componente que realiza el control (*SC*). El cuadro 7.1 muestra los servicios que forman parte de las interfaces que aparecen en el ejemplo. Estos servicios aparecerán a lo largo del capítulo en los ejemplos de generación de distintos diagramas de actividad.

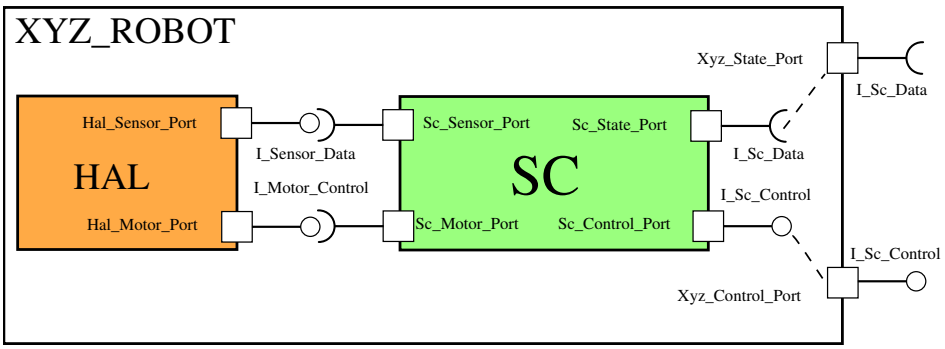


Figura 7.1: Diagrama de componentes ejemplo que se va a utilizar para ilustrar la transformación V³Studio → UML.

DESCRIPCIÓN DE LAS INTERFACES DEL EJEMPLO			
I_Motor_Control	stop_M (); kind: *asynchronous*	I_SC_Control	enable (); kind: *asynchronous*
	jogN_M (); kind: *asynchronous*		stop (); kind: *asynchronous*
	jogP_M (); kind: *asynchronous*		jogP (); kind: *asynchronous*
	moveTo_M (position : **in** DT); kind: *asynchronous*		jogN (); kind: *asynchronous*
I_Sensor_Data	getSensor (reading : **out** DT); kind: *synchronous*		disable (); kind: *asynchronous*
			reset (); kind: *asynchronous*
	getStatus (status : **out** DT); kind: *synchronous*		resume (); kind: *asynchronous*
			abort (); kind: *asynchronous*
I_SC_Data	getState (state : **out** DT); kind: *synchronous*		moveTo (position : **in** DT); kind: *asynchronous*
			changeMoveParms (parm : **in** DT; value : **in** DT); kind: *synchronous*

Cuadro 7.1: Descripción de las interfaces del ejemplo de la figura 7.1. Los servicios se expresan utilizando una notación similar a Ada, ya que se indica de forma explícita el sentido de los parámetros. El tipo «DT» hace referencia a un tipo de dato cualquiera.

Por otro lado, la figura 7.2 muestra una captura de pantalla del entorno Eclipse en la que se pueden observar la estructura de clases V^3Studio que conforman el modelo de ejemplo. En esta captura puede observarse también cómo el uso de las relaciones de asociación y de la separación entre instancia y definición permiten declarar por separado cada uno de los elementos del sistema. Concretamente, se puede observar que la definición de las interfaces y los servicios, la definición de los componentes simples y del componente complejo se han realizado en ficheros independientes, que posteriormente son enlazados para realizar la descripción arquitectónica de la aplicación.

7.2 Visión General de la Transformación

Esta sección presenta las principales decisiones que se han tomado a la hora de diseñar la transformación de modelos que se expone en este capítulo. En esta sección se exponen (1) las decisiones relativas a la utilización de un determinado diagrama UML en detrimento de otro, (2) la estructura general de la transformación y (3) las decisiones de diseño arquitectónicas de alto nivel, aquellas que afectan a la estructura de la aplicación y que condicionan la mayor parte de las decisiones que aparecen en secciones posteriores, particularmente en la sección 7.3.

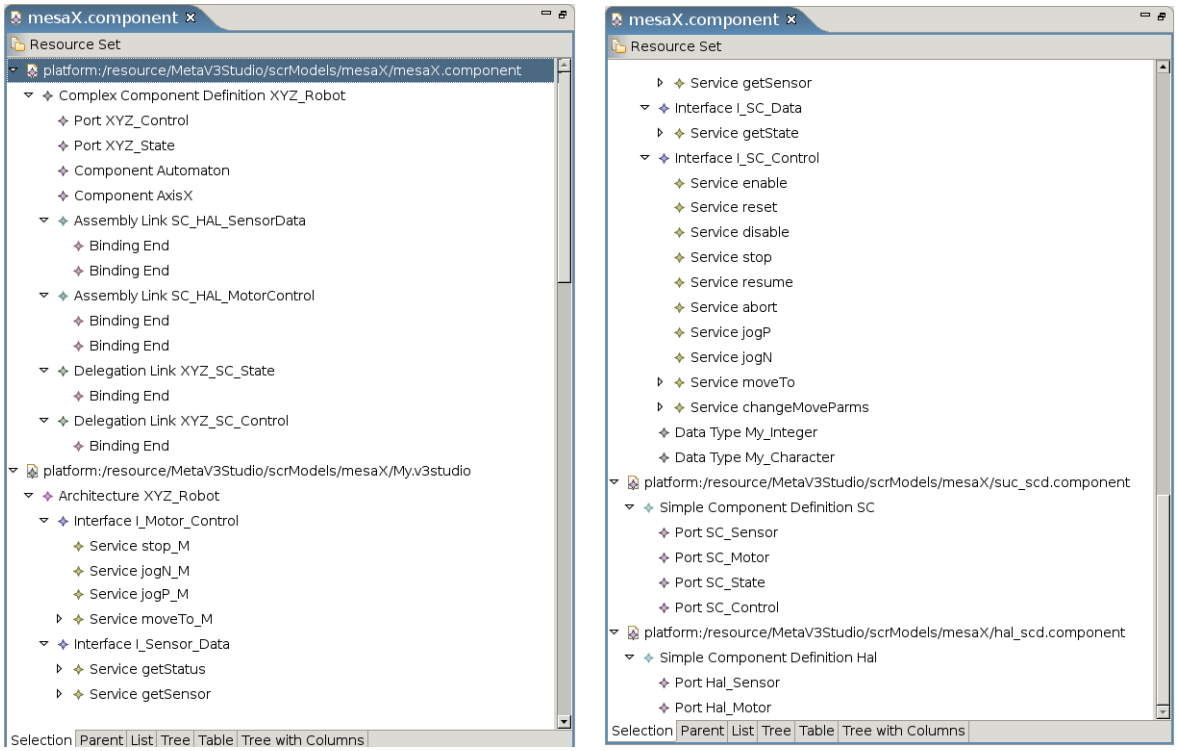


Figura 7.2: Captura de pantalla del entorno Eclipse en que se muestra la estructura de clases V³Studio que describen el ejemplo de la figura 7.1. Obsérvese como cada uno de los componentes y las interfaces que forman el sistema se han definido por separado.

Además, esta sección expone las decisiones de diseño que afectan a la totalidad de la transformación y que, por tanto, no aparecen en ninguna de las secciones dedicadas a una vista en particular. Para ello, esta sección está organizada en cuatro apartados, de los cuales el más importante, sin duda, es el siguiente. En él se describe la macro-estructura de la aplicación y la arquitectura interna que se ha elegido para describir los componentes. Los siguientes apartados describen someramente la estructura de diagramas y de modelado de UML y justifican la elección de los diagramas que se generan en el modelo UML destino. La sección se cierra presentando la estructura de paquetes y componentes que se genera durante la transformación, ya que es utilizada a lo largo de todo el capítulo.

7.2.1 DE COMPONENTES A OBJETOS: EL PATRÓN *Active Object*

En este apartado se propone la utilización de una serie de patrones de diseño arquitectónicos para traducir los conceptos principales en que se basaría un hipotético lenguaje basado en componentes a los conceptos que se utilizan en un lenguaje orientado a objetos. La propuesta que se describe a continuación contempla los conceptos «componente» y «mensaje», ya que:

- El concepto «puerto» se considera parte del concepto componente, ya que no tiene sentido que exista un puerto si no aparece definido en el contexto de aquel.
- El concepto «conector», como se mencionó en el apartado 6.4.4, no está explícitamente modelado en V^{3}Studio, sino que se considera un tipo especial de componente simple (ver figura 6.5 en la página 138). Es posible que en un futuro este concepto aparezca explícitamente en V^{3}Studio para generar de forma automática los componentes simples que implementen la funcionalidad de alguno de los catálogos de conectores, al estilo del definido por Medvidovic en su estudio [146].

A la vista de la definición de componente adoptada (la que realiza UML 2) y de sus implicaciones sobre comportamiento, independencia del entorno y forma de comunicación, se hizo patente que ninguno de los patrones descritos por Gamma [96] sería suficiente para completar y tratar correctamente el concepto «componente». En este punto se decidió recurrir a los textos de la serie *Pattern Oriented Software Architecture*, en la que se presentan soluciones de alto nivel para estructurar la arquitectura de un sistema, dependiendo de la finalidad del mismo. De toda la serie se utilizó finalmente el volumen II [190], en el que se describen distintos patrones para desarrollar sistemas multi-tarea y distribuidos, justamente el tipo de sistemas que se necesitan diseñar en un dominio como el robótico.

De todos los tipos de patrones que se describen en el volumen II, finalmente se decidió que los más adecuados eran los patrones englobados dentro del apartado «patrones para el diseño concurrente». Y dentro de todos los allí presentados (*Active Object*, *Monitor Object*, *Half-Sync/Half-Async*, *Leader/Followers* y *Thread-Specific Storage*) se decidió utilizar el patrón *Active Object* para implementar la arquitectura interna de un componente. Los patrones restantes fueron desechados debido a que, o bien no aportaban tanta flexibilidad como el *Active Object* (éste es el caso del *Monitor Object*), o bien su propósito es definir la arquitectura de concurrencia global del sistema (hecho incompatible con el uso de componentes, puesto que cada componente tiene que ser capaz de establecer, hasta ciertos límites, su propia política de concurrencia).

El patrón *Active Object* permite «desacoplar la ejecución de un método de la invocación del mismo» [190], ya que ambos son ejecutados en distintos hilos de ejecución. Es decir, el código asociado al método invocado se ejecuta en un hilo distinto de el hilo en que se ejecuta el cliente que lo invoca. El *Active Object* permite, por tanto, mejorar la concurrencia del sistema y simplificar el acceso al mismo. La utilización de un hilo de ejecución diferente añade flexibilidad extra al patrón *Active Object*, ya que permite definir distintas políticas de ejecución. Sin embargo, esta flexibilidad no es gratuita, ya que se incurre en una penalización en tiempo de ejecución. A pesar de este inconveniente, se juzgó necesario disponer de esta flexibilidad a la hora de hacer un diseño que pueda evolucionar en el futuro, y por esto se decidió que los componentes se implementarían utilizando el patrón *Active Object*, en

detrimento del la segunda opción considerada, el *Monitor Object*. La figura 7.3 muestra el diagrama de clases modelo del *Active Object*, que está formado por las siguientes clases:

Servant: servidor, clase que implementa realmente los métodos invocados por el cliente.

Proxy: clase que actúa como delegado del servidor.

Client: cliente, clase que realiza la invocación de un método del servidor a través del proxy.

MethodRequest: petición de servicio, clase abstracta que modela la solicitud de ejecución de un servicio.

ConcreteMethodRequest: clases que contienen los parámetros necesarios para llevar a cabo la ejecución de un método en particular.

ActivationQueue: lista de activación, clase que almacena las peticiones de servicio.

Scheduler: planificador, clase encargada de seleccionar de la lista la petición de servicio que se tiene que ejecutar a continuación.

Future: depositario, clase que contiene el resultado del método invocado por el servidor en respuesta a una petición de servicio.

En tiempo de ejecución, el patrón funciona de la siguiente manera: el cliente (que se ejecuta en un hilo distinto al del servidor) utiliza el proxy para invocar uno de los métodos del servidor. El proxy crea (1) un objeto de la clase concreta (modela la petición de servicio realizada por el cliente) y solicita al planificador que la encole en la lista de activación, y (2) un objeto de la clase depositario, en el que se escribirá el resultado de la ejecución de la petición de servicio, que devuelve al cliente. La lista de activación actúa como buffer de desacoplo entre los hilos del cliente y del servidor, es decir, a partir de este punto todo el código se ejecuta en el hilo del servidor, quedando libre el cliente para continuar. Posteriormente, el planificador selecciona, dependiendo de sus propios heurísticos, la siguiente petición de servicio a ejecutar. La petición de servicio elegida contiene una operación para comprobar si se puede ejecutar o no, y otra para efectivamente invocar el método adecuado en el servidor. Una vez que el método ha terminado, la petición de servicio escribe el resultado en el objeto depositario, con el que puede sincronizarse el cliente para obtener finalmente el resultado. Todos los roles que desempeñan las clases que integran este patrón han sido adaptados a los conceptos presentes en V^{3}Studio de la siguiente manera:

Servant: componente que ofrece el servicio requerido por el cliente.

Proxy: el puerto del componente que proporciona el servicio actúa como proxy para el componente que realiza la petición.

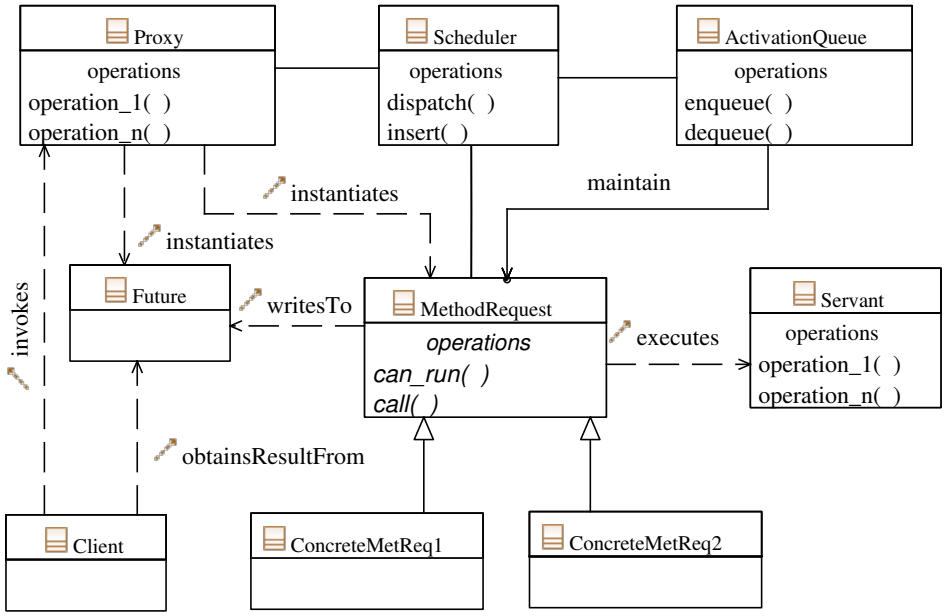


Figura 7.3: Diagrama de clases del patrón *Active Object* (extraído de [190])

Client: componente que solicita la ejecución de un servicio en el servidor mediante el uso del puerto (proxy) correspondiente.

MethodRequest: para modelar la petición de servicio se utiliza el patrón *Command*[1] [96]. En cuanto el servidor recibe una petición de servicio, el puerto crea un objeto del tipo adecuado y lo encola en la lista de activación.

ConcreteMethodRequest: modela los atributos y características particulares de un método.

ActivationQueue y Scheduler: el rol desempeñado por estas clases es asumido por la máquina de estados asociada al componente. La máquina de estados dispone de una lista en la que se van encolando las peticiones de servicio, que pueden ser atendidas o descartadas dependiendo del estado actual de la máquina de estados.

Future: en V^{3}Studio se corresponde a los servicios asíncronos con notificación o sin notificación de la ejecución del mismo.

En el apartado 6.4.3 se describió que los servicios asociados a una interfaz pueden ser servidos de forma síncrona o asíncrona por el componente que los ofrece o requiere. El tratamiento del primer tipo de servicio, servicio síncrono, no reviste ninguna complejidad, ya que este tipo de servicios siempre están disponibles y se sirven tan pronto son solicitados por el cliente (y en el hilo de ejecución del cliente). Los servicios asíncronos, por otro lado, son tratados y ejecutados en el contexto del objeto activo, que encola las peticiones para

[1] El patrón de diseño *Command* (Comando) permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además se facilita la parametrización de los métodos.

procesarlas posteriormente. Por tanto, el patrón *Active Object* permite no sólo modelar la arquitectura interna del componente, sino también resolver las peticiones de servicio de tipo asíncrono.

De lo expuesto en este apartado se puede deducir que tanto la elección del patrón *Active Object*, para implementar la arquitectura interna de un componente, como el uso del patrón *Command*, para llevar a cabo la comunicación entre los mismos, guían el desarrollo de la transformación y limitan sobremanera ciertas decisiones para implementar los conceptos presentes en el modelo V^{3}Studio. Como se enumera en el capítulo de conclusiones, otro de los posibles trabajos futuros consiste en estudiar otros patrones y otras formas posibles de implementar los conceptos CBD.

7.2.2 DIAGRAMAS UML USADOS EN LA TRANSFORMACIÓN

UML es un lenguaje de descripción software que abarca todas las etapas del desarrollo de una aplicación, y por tanto, dispone de muchos tipos de diagrama para mostrar cada una de las distintas etapas de su desarrollo. Aunque UML se divide en diagramas y normalmente se utiliza esta división para trabajar con él, no hay que olvidar que cada diagrama muestra únicamente un conjunto de clases relacionadas semánticamente entre sí, pero que por debajo de todo diagrama se encuentra el meta-modelo completo de UML. Es decir, aunque en el diagrama (o vista) de casos de uso sólo aparecen las clases relacionadas (actor, caso de uso, conector, etc), el modelo UML puede contener otras clases, por ejemplo estado, componente, actividad, etc.

La figura 7.4 muestra la clasificación de los diagramas UML, tal y como aparece en el anexo F de la super-estructura de UML 2. En esta figura puede observarse que los diagramas se clasifican en dos ramas, dependiendo de si el diagrama modela el comportamiento o la estructura del software (vista dinámica o vista estática). En dicha figura se han resaltado los diagramas que se han utilizado para llevar a cabo la transformación de un modelo V^{3}Studio a un modelo UML:

Vista estructural: para describir la estructura del software se utilizan principalmente las clases que conforman los diagramas de clases y paquetes de UML, ya que cada uno cumple con un objetivo diferente. El diagrama de clases, completado con clases de otros diagramas[2], permite describir la estructura de clases que genera la

[2]El diagrama de clases de UML modela únicamente entidades genéricas y, en determinadas circunstancias, puede ser necesario hacer referencia a instancias concretas, representadas mediantes las clases que aparecen en otras vistas, como la de objetos o estructuras compuestas. El apartado *6.4.2 Semantic Levels and Naming* de la superestructura de UML 2 [177] aclara este punto y describe los distintos niveles de clasificación semántica que contempla UML en las clases que define.

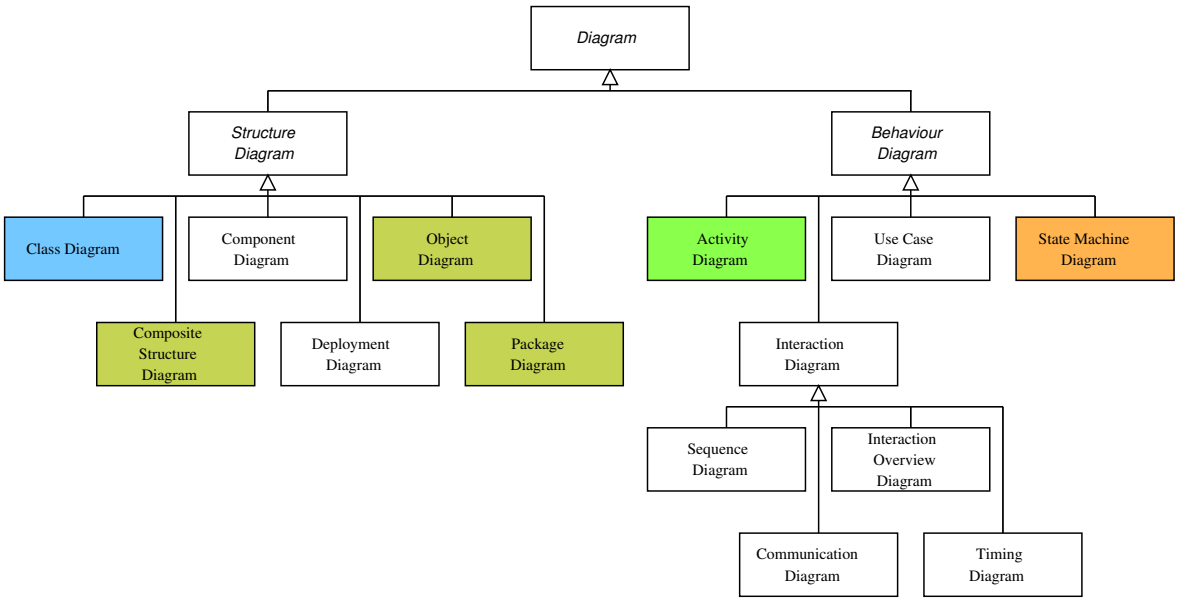


Figura 7.4: Jerarquía de diagramas de UML (extraído de [177])

transformación M2M, mientras que el diagrama de paquetes permite organizar las clases en distintos paquetes, tal y como se muestra en la sección 7.2. La generación de paquetes ha servido tanto para organizar la propia transformación a UML como para facilitar la posterior transformación a código Ada (ver capítulo 8), y ha resultado ser de gran utilidad (mayor que la que se esperaba).

La transformación de la vista arquitectónica de V^{3}Studio es la única que utiliza clases definidas tanto en el diagrama de clases como en el de paquetes de UML. Las características particulares de esta parte de la transformación se describen en el sección 7.3. Además de generar los paquetes y las clases que modelan un componente V^{3}Studio, la transformación crea una serie de operaciones derivadas de la semántica asociada a los conceptos de esta vista, como son: constructores, operaciones para enlazar puertos, operaciones para resolver servicios, etc.

Vista dinámica: en este caso se han utilizado las clases presentes en los diagramas de máquinas de estados y en los diagramas de actividades. A pesar de que se ha comentado que se utilizan los diagramas UML que proporcionan los elementos más cercanos a los lenguajes de implementación, la transformación V^{3}Studio $\rightarrow$ UML mantiene las máquinas de estados en el modelo de destino por las siguientes razones: (1) la máquina de estados es un elemento muy localizado que afecta únicamente al componente y no a la arquitectura del sistema (como sucede en la vista arquitectónica); (2) la máquina de estados mantiene toda la carga semántica asociada a la vista de comportamiento, i.e. la máquina de estados no se deshace en un conjunto de clases que se confunden con el resto de clases de la transformación y

(3) su mantenimiento facilita el desarrollo de distintas transformaciones a código de la lógica de la máquina de estados. La sección 7.4 describe en detalle la transformación de la vista de comportamiento y su integración con el diagrama de clases.

Por otro lado, los diagramas de actividad aparecen tanto como resultado de la transformación de la vista algorítmica (la que le corresponde por su propia definición) como, sorprendentemente, en la vista arquitectónica. En la transformación de la vista arquitectónica se generan muchas operaciones como consecuencia de la semántica asociada a los conceptos V^{3}Studio, tal y como se ha expuesto al principio de esta sección. Por su naturaleza, un diagrama de actividades describe perfectamente la secuencia de ejecución de una operación, por lo que son los candidatos perfectos para modelar, a partir de la información que aporta el modelo V^{3}Studio y de la semántica asociada a los conceptos que en él aparecen, el contenido de este tipo de operaciones. En el apartado 7.2.3 se justifica con mayor detalle la decisión de utilizar diagramas de actividades para generar la descripción de dichas operaciones en detrimento del resto de diagramas que proporciona UML, mientras que en el apéndice B se puede consultar un resumen de las clases UML que se han utilizado en la generación de esta transformación. La descripción específica de la transformación de la vista algorítmica se encuentra en la sección 7.5, mientras que las operaciones que se generan en las otras vistas se describen en el mismo apartado en que se explica la transformación de la vista. Los diagramas de actividad que se muestran a lo largo de todo este capítulo plasman de forma explícita tanto el flujo de datos como el flujo de control. Esto es necesario ya que, aunque muchas veces el flujo de datos implica flujo de control, existen situaciones en las que el flujo no es claro, e.g. cuando se utiliza un `ForkNode` para clonar el resultado de una acción.

En UML, los diagramas que describen el comportamiento (actividades y máquinas de estados) están supeditados a otras clases y diagramas que describen la estructura del software. Por tanto, todas las clases que forman parte de los diagramas que describen el comportamiento están contenidas por algunas de las clases o paquetes que describen la estructura del modelo. Y de hecho es así como aparecen en el modelo UML que se obtiene como resultado de la transformación. La figura 7.5 muestra un extracto del meta-modelo de UML 2 en el que se puede observar la relación existente entre los elementos que realizan la descripción estructural (principalmente `Class`, `Feature` y `Operation`) y los que describen su comportamiento (`Behavior` y sus descendientes). Concretamente, se quiere destacar la relación entre las clases que modelan ambas realidades: en esta figura puede observarse que un `Class` contiene un conjunto de clases `Behavior`, que se usan tanto para describir su comportamiento (referencia `classifierBehavior`) como para describir el comportamiento de las `Operation` que contiene. Esta estructura es fundamental tanto para la realización de la transformación como para su comprensión. El resto de secciones del

capítulo harán referencia a esta figura para explicar determinados aspectos de la generación de las operaciones y de la descripción del comportamiento.

El resto de diagramas UML, junto con los elementos asociados, no son utilizados de momento en esta transformación. Aquellos casos que planteen dudas sobre la elección de un tipo de diagrama u otro serán comentados y justificados en el correspondiente apartado. El objetivo tras el desarrollo de V^3Studio es crear un meta-modelo y un conjunto de transformaciones y herramientas asociadas que sean lo suficientemente simples para facilitar su uso y lo suficientemente complejas para modelar con detalle en el dominio asociado, en este caso la robótica. No se descarta, sin embargo, que en un futuro se añadan nuevos diagramas a la transformación, aunque posiblemente para captar otros aspectos del diseño software, como por ejemplo casos de uso (relacionados con requisitos software) o diagramas de despliegue.

7.2.3 JUSTIFICACIÓN DEL USO DEL DIAGRAMA DE ACTIVIDAD

Según se mostró en el apartado 7.2.2, UML permite describir el comportamiento y la implementación de una operación mediante diagramas de actividades, diagramas de interacción (secuencia, colaboración y comunicación) o máquinas de estados. Se desechó la idea de describir las operaciones mediante una máquina de estados ya que las operaciones de conexión son fundamentalmente secuenciales y, por tanto, no dependen de la ocurrencia de eventos (hecho en que se basa la descripción mediante máquinas de estados). Además, las máquinas de estado describen el comportamiento aislándose del resto de elementos del sistema, característica principal por la que se utilizan para describir el comportamiento de los componentes, pero que no es deseada en este caso.

Los diagramas de interacción hacen hincapié en el intercambio de mensajes entre objetos, por lo que no son adecuados para describir operaciones que no involucran intercambio de mensajes entre objetos, sino que describen la ejecución interna de una operación. Finalmente se eligió utilizar el diagrama de actividades en detrimento de uno de interacción (particularmente, diagrama de secuencia) por las razones siguientes:

- El diagrama de actividades proporciona un modelo muy cercano al código final de implementación.
- UML establece que las acciones son las unidades más pequeñas que permiten describir el comportamiento, y por tanto, cualquier descripción de un comportamiento tiene que utilizarlas tarde o temprano. Aunque se utilizara un diagrama de interacción, finalmente tendría que ser completado con la inclusión de acciones para modelar la ejecución del comportamiento asociado a la recepción de los distintos mensajes.

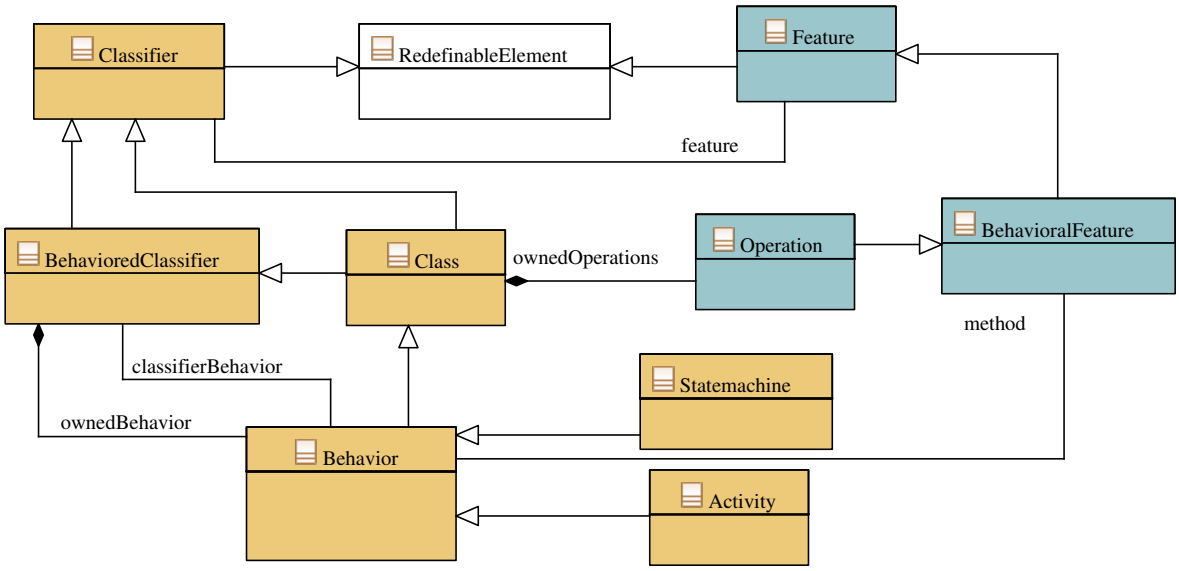


Figura 7.5: Diagrama que muestra la relación entre las clases que describen la estructura (en naranja) y el comportamiento (en azul) en UML.

- La utilización de un diagrama de interacción complica aún más el modelo UML puesto que añade muchos conceptos nuevos sin mejorar la potencia expresiva del modelo, ya que el diagrama de interacción y el de actividades modelan la misma realidad, aunque desde distintos puntos de vista.

- Los diagramas de interacción, y particularmente el más utilizado de ellos, el de secuencia, hacen hincapié en la ordenación temporal de cada uno de los mensajes que se intercambian los objetos entre sí. UML establece que esta ordenación se realiza mediante la especificación de cuatro tipos de eventos (envío y recepción de un mensaje, comienzo y finalización de la ejecución de una acción) que se ordenan parcialmente y uno a uno mediante el uso del concepto `GeneralOrdering`. La información queda, en este caso, dispersa y desordenada, por no hablar de la cantidad de elementos que tienen que añadirse al modelo (cuatro por cada mensaje que se intercambia).

El modelado del comportamiento mediante actividades ha sufrido una profunda modificación entre las versiones de UML 1.5 y 2.0. Estos diagramas han pasado de ser una mera representación alternativa de las máquinas de estados a estar dotados de su propia semántica, ya que actualmente están basados en redes de Petri y en el paso de *tokens*. Las actividades, generadas de forma automática, que describen la implementación de las operaciones de conexión se han diseñado teniendo en cuenta la nueva naturaleza de las actividades en UML, de forma que puedan ser utilizadas, además de para generar código en un paso posterior, en un simulador de redes de Petri.

El apéndice B contiene un breve resumen de las acciones y actividades UML 2 que se han utilizado para desarrollar los diagramas de actividades que se exponen a lo largo de todo este capítulo. Se recomienda su consulta para facilitar el seguimiento de los mismos, ya que en dichos diagramas se utilizan acrónimos para referirse a las acciones UML que se describen en este apéndice.

7.2.4 V^{3}Studio → UML: estructura de la transformación

Aunque UML dispone de varios diagramas para describir las distintas etapas del diseño software y los distintos aspectos del mismo, no todos los diagramas están situados en el «mismo nivel» jerárquico y de abstracción. Concretamente, los diagramas de comportamiento que atañen a la transformación están subordinados a la existencia de un diagrama estructural que describa, valga la redundancia, la estructura de los elementos cuyo comportamiento modelan. Además, estos diagramas de comportamiento están lógicamente contenidos por el elemento estructural cuya dinámica describen (ver figura 7.5). Por tanto, el primer objetivo de la transformación V^{3}Studio → UML es la generación de los elementos UML que definen la estructura de la vista arquitectónica de V^{3}Studio: clases (`Class`), componentes (`Component`), paquetes (`Package`), propiedades (`Property`) y características (`Feature`).

La transformación del modelo V^{3}Studio comienza por el componente complejo de mayor nivel, el que describe la aplicación. Este componente se transforma en un paquete que contendrá el resto de paquetes y elementos que se van a generar durante la transformación. Estructurar la transformación en paquetes ayuda no sólo a mantenerla ordenada y bajo control, sino que también permite establecer y comprobar propiedades, como la encapsulación de sus elementos, lo que no hubiera sido posible en caso contrario. La figura 7.6 muestra el diagrama de paquetes y componentes del contenido del paquete principal (el componente complejo que define la aplicación), generado a partir del modelo V^{3}Studio de ejemplo de la figura 7.1. En esta figura se puede observar que la transformación genera los siguientes elementos:

Clases básicas del framework (en verde): representadas por las clases abstractas `Component` y `Port`. Estas dos clases contienen la funcionalidad básica que se quiere añadir a todas las clases que van a representar estos conceptos CBD. Todos los puertos y componentes definidos por el modelo V^{3}Studio heredan de una u otra cuando son transformadas al diagrama UML. Estas clases se crean siempre que se realiza la transformación, su definición es fija y no depende del modelo de entrada. En el apartado 7.3.1 se realiza una descripción pormenorizada de estas dos clases.

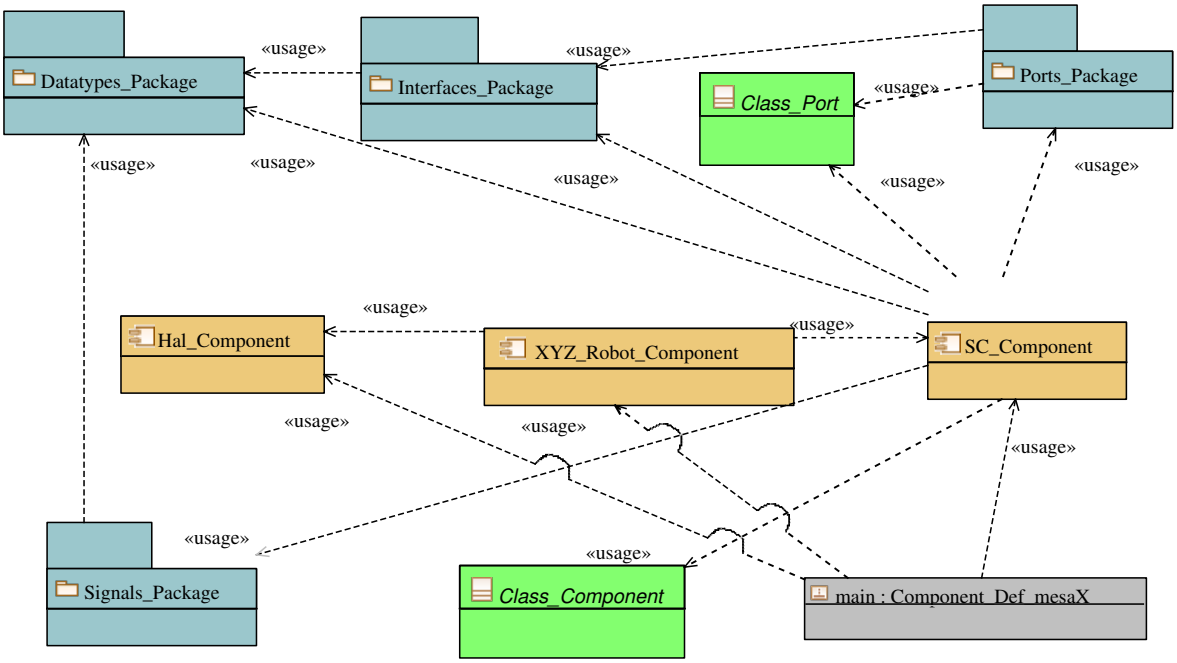


Figura 7.6: Diagrama de paquetes generado para el ejemplo de la figura 7.1. Todos los paquetes están al mismo nivel, i.e. no existen relaciones de contención entre ellos.

Paquetes que contienen la definición de los puertos (en azul): son, respectivamente, los paquetes `Interfaces_Package` y `Ports_Package`. Aunque siempre se generan ambos paquetes durante la transformación, su contenido depende del modelo de entrada V³Studio. El primero de ellos contiene la definición de las interfaces, junto con sus servicios, que aparecen en el modelo de entrada V³Studio. El segundo contiene un conjunto de clases abstractas que heredan de `Port` y que implementan todos los conjuntos de interfaces ofrecidas presentes en el modelo de entrada. Las razones que justifican la necesidad de este paquete se exponen posteriormente en el apartado 7.3.3. Todas las clases e interfaces definidas en estos paquetes son utilizados por el resto de elementos generados por la transformación como tipos que son comunes a todos ellos.

Paquetes que contienen la definición de los tipos de datos y las señales (en azul): son los paquetes `Datatypes_Package` y `Signals_Package`, respectivamente. Estos paquetes se muestran en azul para reforzar la idea de que contienen conceptos auxiliares para definir los componentes del sistema. Como en el caso anterior, el contenido de ambos paquetes depende del contenido del modelo de entrada V³Studio. El primero de ellos contiene únicamente la definición de los tipos de datos tal y como se realiza en V³Studio, pero utilizando ahora la clase UML `DataType`. El segundo contiene las definiciones de las señales (clase `Signal` de UML) que se utilizan en la máquina de estados para modelar el suceso de eventos. Estas clases se describen más adelante, en la sección que detalla la transformación de la vista de comportamiento (sección 7.4).

Descripción de los componentes (naranja): son los que están representados, de forma genérica, por los componentes denominados `XXX_Component`. La transformación crea un componente UML por cada una de las definiciones de componente (`ComponentDefinition`) presentes en el modelo de entrada, que contiene el resto de elementos que modelan el componente y mantienen la cohesión y la encapsulación de los mismos. En el apartado 7.3.2 se realiza una descripción más detallada del contenido de estos componentes y se justifica la traducción que se ha realizado de los conceptos CBD a un lenguaje que describe software siguiendo el paradigma de la orientación a objetos.

Clase principal (gris): esta clase, representada por un `ComponentRealization`, se corresponde con el `CompositeComponentDefinition` de mayor nivel, el que describe la aplicación. Esta clase es la encargada de crear, arrancar e inicializar la aplicación. Para ello se encarga de crear una instancia del componente complejo de mayor nivel que, como se describe en el apartado 7.3.4, va instanciando cada uno de los componentes (simples o complejos) que contiene. Tras crear el primer componente, el procedimiento principal invoca el método `portLink` (consultar apartado 7.3.5.4) para proceder a conectar los puertos de los componentes.

Un detalle que llamará la atención del lector es que en la figura 7.6 no se han representado todas las relaciones de «uso» entre los distintos paquetes que se han descrito. En concreto, se han omitido las relaciones repetidas solamente en el caso de los paquetes que definen los componentes para evitar la proliferación de líneas que se entrecruzan en el diagrama y que dificultan la interpretación del mismo. En la figura puede verse que sólo el paquete `SC_Component` tiene dibujadas todas las relaciones con el resto de paquetes del sistema. Los otros dos paquetes, que representan los componentes `XYZ_Robot` y `HAL`, tienen las mismas relaciones que el paquete `SC_Component` aunque no se muestren. En el caso del componente complejo `XYZ_Robot` se muestran, además, las relaciones de uso con los paquetes que definen los componentes simples que contiene este componente complejo.

7.3 V^{3}Studio → UML: Vista Arquitectónica

En esta sección se describe la generación de los paquetes y las clases que describen la estructura de los componentes a partir de la vista arquitectónica. Antes de comenzar a describir el proceso de transformación, conviene recordar que, como ya se dijo en la sección 6.4, V^{3}Studio utiliza los conceptos `ComponentDefinition` y `Component` para hacer referencia, respectivamente, a la definición y la instancia de un componente. Esta separación entre definición e instancia se plasma explícitamente en la

parte de la transformación que describe el apartado 7.3.2, y permite reutilizar la misma definición en diversas instancias. Al igual que sucede en la orientación a objetos, únicamente los `Component` representan a verdaderas instancias en tiempo de ejecución.

En la parte de la transformación que se describe en esta sección se generan: (1) la estructura de paquetes en que se ha organizado la transformación (tal y como se ha descrito en el apartado 7.2.4); (2) los diagramas de clases que describen la transformación de los conceptos de CBD, concretamente puerto y componente, a UML y (3) las operaciones, junto con los respectivos diagramas de actividades que describen su funcionamiento, derivadas de la semántica asociada al meta-modelo V^{3}Studio, tal y como describe la sección 7.2. La generación de los diagramas de clases sigue las siguientes convenciones:

- No se distingue entre las relaciones de asociación con composición o con agregación; todas son con composición. El verbo «contener» se utiliza, en el contexto de la transformación, para expresar que existe una relación de composición en el sentido UML: «una relación de composición implica que el objeto compuesto es responsable de la existencia y almacenamiento de cada una de sus partes». Por tanto, es labor de la clase contenedora crear y controlar las instancias de cada uno de los objetos que contiene durante todo su tiempo de vida.
- Se ha adoptado la decisión de proteger las variables de instancia de las clases públicas con métodos accesores, pero dejar directamente accesibles las de las clases privadas, para de esta forma reducir el número de indirecciones generadas.
- Las relaciones de asociación establecen que un objeto sólo tiene referencia a otro. El objeto que contiene esta referencia no tiene, por tanto, ningún control sobre la creación o destrucción del objeto referenciado. Dependiendo de la política de protección de variables que sigan los objetos, la referencia podrá fijarse por acceso directo a la misma o mediante el uso de operaciones accesoras.
- De forma general, existen dos formas en que se pueden inicializar cada una de estas referencias: o bien se inicializan a la vez que se crea el objeto, o bien existe una operación encargada de ello. Para diferenciar entre estas dos formas de inicialización se utiliza el atributo `aggregation` de la clase UML `Property` (clase que modela las propiedades de un objeto). Los valores de este atributo, de tipo enumerado, se utilizan para distinguir cuándo se va a utilizar la primera forma de inicialización (valor `shared`) y cuándo la segunda (valor `none`). Además, el atributo `isReadOnly` de la clase `Property` se utiliza para controlar el acceso a estas propiedades y la generación de las operaciones correspondientes.

Esta sección se divide en seis apartados, en los que se presentan las distintas fases o partes en que se ha organizado esta parte de la transformación. El primero de estos

apartados describe las clases bases del framework, `Port` y `Component`, de las que heredan todas las clases que representan los puertos y los componentes en el modelo UML de salida. Posteriormente, se dedican sendos apartados a describir en detalle la transformación de los conceptos `Component` y `Port` de V^{3}Studio a un lenguaje orientado a objetos, utilizando para ello el patrón *ActiveObject*, descrito en el apartado 7.2.1. Los apartados restantes se dedican a describir, mediante diagramas de actividad, las operaciones cuya implementación se genera automáticamente gracias a la semántica asociada a los conceptos «puerto» y «componente». Concretamente, se enumeran las operaciones relacionadas con la construcción de los objetos (constructores) de puertos y componentes, las operaciones relacionadas con la conexión de los puertos y las relacionadas con la petición de servicio.

7.3.1 Descripción de las clases base del framework

Como se ha comentado anteriormente, la transformación de los conceptos «puerto» y «componente» V^{3}Studio a UML se apoya en la existencia de dos clases abstractas que contienen las operaciones comunes a todas estas clases. Estas clases abstractas comunes base son las siguientes:

`Component`. De esta clase heredan todas las clases que representan al componente real (`Component_Def_XXX`, consultar siguiente apartado). El propósito de esta clase es recoger todos los servicios básicos del framework que soporta el sistema basado en componentes (consultar sección 2.4).

`Port`. De esta clase heredan todas las clases que representan a cada uno de los puertos de un componente V^{3}Studio. La clase abstracta `Port` define tres operaciones para controlar el funcionamiento general de los puertos: `connect` (consultar apartado 7.3.5.2), `disconnect` (consultar apartado 7.3.5.3), `receiveSignal` y `sendSignal` (consultar apartado 7.3.6). Además de estas operaciones, cada puerto concreto implementa las operaciones correspondientes a los servicios que ofrece o requiere.

7.3.2 Transformación de los componentes V^{3}Studio

Como se describió en la exposición de la vista arquitectónica de V^{3}Studio (consultar apartado 6.4.2), existen dos tipos de `ComponentDefinition`: la definición de un componente simple (atómico) y la definición de un componente complejo (formado por instancias de otros componente, ya sean simples o complejas a su vez). En ambos casos, y de forma genérica, cada `ComponentDefinition` presente en el modelo de entrada

V3Studio se transforma en un componente UML que contiene todas las clases que permiten implementar la funcionalidad del componente (incluyendo los puertos y la máquina de estados), lo que permite mantener fácilmente la connotación de encapsulación y cohesión asociada al concepto «componente».

De cara a implementar el componente según el patrón *Active Object* (consultar apartado 7.2.1), la transformación de un `ComponentDefinition` comienza, independientemente de si se define un componente simple o complejo, creando un componente UML que contiene la dos clases que se describen a continuación. Como se describirá más abajo, con esta decisión se consigue (1) un diseño minimalista (conforme al principio KISS), (2) desacoplar los distintos roles que puede desempeñar un componente, esto es encapsulación de su contenido y ejecución de su funcionalidad y (3) un diseño que puede evolucionar fácilmente en un futuro. En el siguiente enumerado, `XXX` hace referencia al nombre del componente V3Studio.

`Component_Def_XXX`: clase pública que hereda de la clase base del framework `Component`. Esta clase utiliza el patrón de diseño *Façade*[3] [96] para limitar y proteger el acceso al resto de elementos que forman la descripción del componente. Esta clase fachada contiene (relación de composición) un objeto de la clase `XXX_Real` (descrita en el siguiente punto), que es la que realmente implementa la funcionalidad del componente. Además, también se generan, de forma automática, las operaciones necesarias para obtener referencias a los puertos del componente, tal y como se detalla en el apartado 7.3.3. Usando estas operaciones, la clase principal es capaz de obtener los puertos de cada componente y conectarlos entre sí según indica el modelo de entrada. Esta clase, por tanto, dota al diseño de la connotación de «encapsulación del contenido» asociada al concepto «componente».

`XXX_Real`: clase con visibilidad de paquete. Esta clase es la que realmente implementa la funcionalidad del componente y controla su funcionamiento. El comportamiento del componente es asociado a la clase `XXX_Real` mediante la referencia `classifierBehavior` (consultar apartado 7.2.2) de esta última una vez realizada la correspondiente transformación de la vista de comportamiento (ver apartado 7.4). La clase `XXX_Real` contiene, además, los objetos de las clases en que se transforman los puertos definidos en el componente V3Studio (ver apartado 7.3.3). La decisión de proteger el acceso a la clase `XXX_Real` responde al hecho de que un componente *encapsula* su funcionalidad y utiliza sus puertos para ofrecer y requerir servicios de otros componentes del sistema. Además, la política de concurrencia establecida para el componente (consultar el apartado 6.4.1) determina el valor de la propiedad

[3]El patrón de diseño *Façade* proporciona una interfaz unificada sencilla que hace de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas. Esta labor de intermediario permite, además, que la fachada encapsule parte de la funcionalidad de las partes, limitando el acceso a las mismas.

`isActive`, que se utiliza posteriormente para determinar si el componente se ejecuta en su propio hilo o no.

La figura 7.7 muestra el diagrama de clases UML resultantes de aplicar la transformación de modelos al diagrama de componentes de ejemplo presentado en la figura 7.1. La figura 7.7 muestra un diagrama de clases «aplanado», en el que se han eliminado todos los paquetes que se generan para facilitar su comprensión, y al que se le ha añadido un código de colores para identificar fácilmente las clases que pertenecen a un mismo paquete. También se han incluido las clases base del framework (en verde) y algunas de las clases de los paquetes de definición de los puertos (en azul). Las clases resaltadas en distintos matices de naranja son las que conforman cada uno de los tres componentes que se han generado. El borde rojo que tienen algunas de ellas indica que son clases con visibilidad «de paquete», mientras que el borde negro indica que son clases con visibilidad «pública».

Observando la figura se pueden identificar ya las clases y las relaciones que se generan para los componentes simples *HAL* (clases *Component_Def_Hal* y *Hal_Real*, contenidas en el paquete *Hal_Component*) y *SC* (clases *Component_Def_SC* y *SC_Real*, contenidas en el paquete *SC_Component*). También se pueden identificar las clases que se generan en el caso del componente complejo XYZ_Robot: *Component_Def_XYZ_Robot* y *XYZ_Robot_Real*, ambas contenidas en el paquete *XYZ_Robot_Component*. Sin embargo, el componente complejo exhibe algunas relaciones de asociación que no se han descrito hasta el momento.

7.3.2.1 TRANSFORMACIÓN DE UN COMPONENTE COMPLEJO

Como se dijo en la sección 6.4, la definición de un componente complejo, representada por medio del concepto `ComplexComponentDefinition`, se distingue de la de uno simple fundamentalmente en que contiene *instancias* (representadas por el concepto `Component`) de cualquier tipo de componente (simple o complejo). Por tanto, y como muestra la figura 7.7, la transformación de un componente complejo es similar a la de un componente simple, ya que de la misma manera se genera un componente UML que contiene la definición del componente (visible desde el exterior) y la clase que implementa la funcionalidad real del componente (oculta en el componente UML). También se generan todas las clases necesarias para crear los puertos del componente complejo y las relaciones de composición entre la definición y el componente real y entre este último y todos los puertos, exactamente igual que sucede con la generación de un componente simple (consultar el apartado 7.3.3). Los elementos extra que se generan en caso de transformar un `ComplexComponentDefinition` son los siguientes:

- La clase que implementa la funcionalidad real del componente complejo (clase con

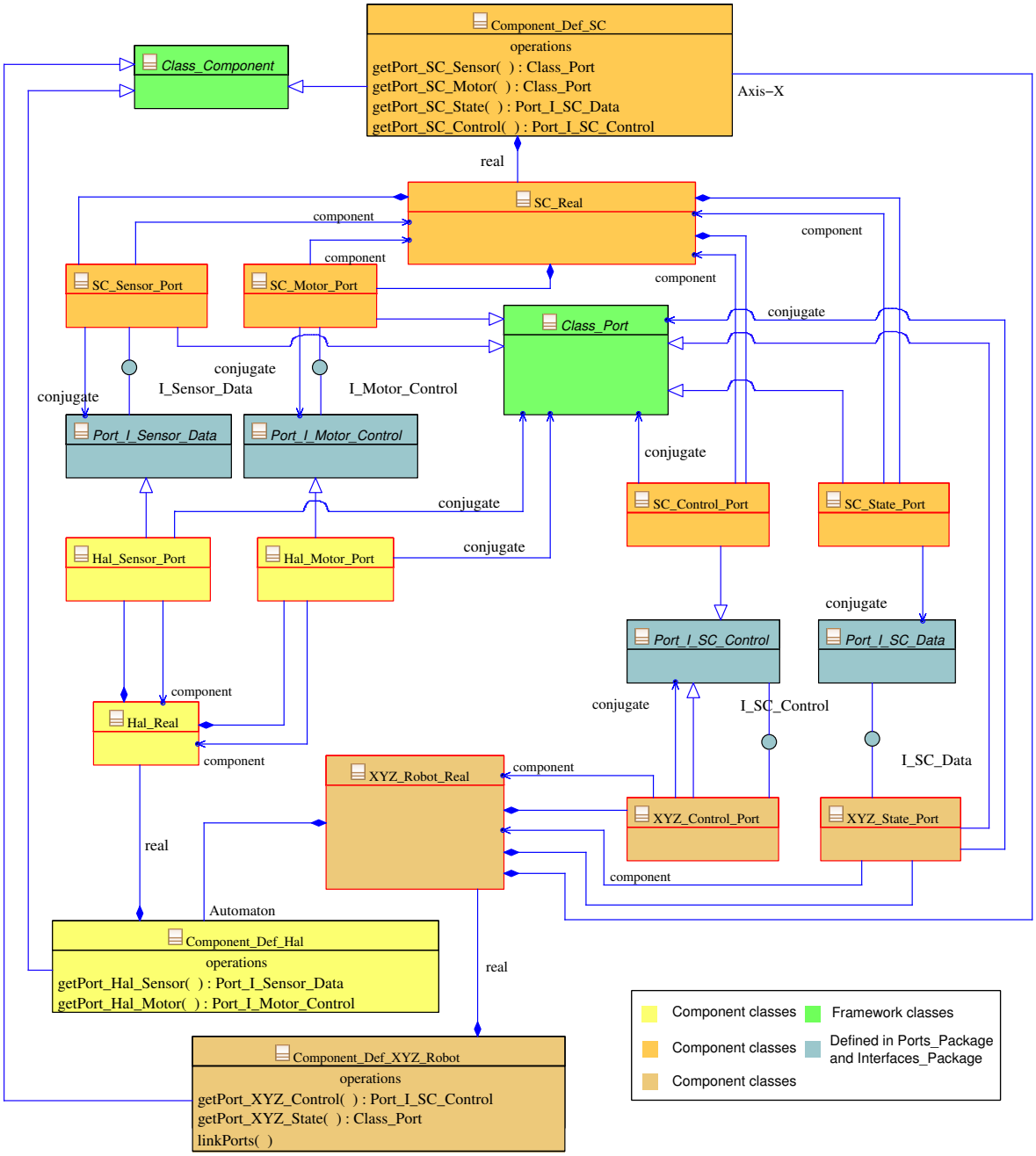


Figura 7.7: Diagrama de clases resultado de la transformación V³Studio → UML. El borde rojo en las clases indica que tienen visibilidad «paquete», mientras que el borde negro indica visibilidad «pública».

visibilidad «paquete» *XXX_Real*) contiene una instancia de la definición (clase fachada con visibilidad «pública» *Component_Def_XXX*) de cada uno de los componentes por los que está formado el componente complejo. Cada una de estas instancias tiene asignada, al igual que la clase que la contiene, una visibilidad «de paquete» para conseguir que los componentes internos de un componente complejo estén completamente encapsulados en su interior y permanezcan, de este modo, inaccesibles desde el exterior del componente (en este caso, del paquete que lo define).

- Se añade una nueva operación a la clase fachada (*Component_Def_XXX*) del componente. Esta operación se denomina *linkPorts* y su objetivo es enlazar todos los puertos de los componentes (`Component`) que contiene la definición de componente complejo de acuerdo al modelo de entrada V^{3}Studio. Todas las operaciones relacionadas con la gestión y la conexión de los puertos son generadas automáticamente y se describen con mayor detalle en el apartado 7.3.5.

En la figura 7.7 puede identificarse, en naranja, las clases generadas para el componente complejo *XYZ_Robot*. Puede observarse que la única diferencia representativa (a parte de la diferencia de tipos) reside en que se ha generado una relación de composición entre la clase *XYZ_Robot_Real* y cada una de las clases que definen la fachada de los componentes que contiene el componente complejo *XYZ_Robot*, i.e. entre la clase *XYZ_Robot_Real* y las clases *Component_Def_Hal* y *Component_Def_SC*, respectivamente. El establecimiento de la relación entre componentes mediante el uso de la clase fachada (*Component_Def_XXX*) refuerza el sentido de encapsulación de la solución y dota a los componentes de la independencia respecto al resto de elementos del sistema que se espera de un diseño CBD.

7.3.2.2 OTRAS TRANSFORMACIONES ALTERNATIVAS

Aunque se contemplaron otras posibles transformaciones del concepto «componente» (desde una única clase a varias más), la traducción que se ha presentado es la que, aún siendo sencilla, sigue fielmente la definición de componente que se ha adoptado: la de UML 2. Además, esta estructura es flexible y puede evolucionar fácilmente. La sencillez de la transformación evita la proliferación de clases que se produciría en caso de que cada componente se transformara en tres o más clases, por lo que directamente se desecharon las alternativas que generaran más clases.

Por otro lado, la que sería la «mejor» solución, i.e. generar una única clase por componente, obliga a fusionar, en una única clase, la funcionalidad de la clase base del framework (clase `Component`) con la funcionalidad concreta del componente. Este

hecho se evaluó como una posible falta de flexibilidad en la evolución del diseño, por lo que se decidió que el componente V³Studio sería transformado en dos clases, lo que facilita la evolución por separado de cada una de ellas, aunque sigue existiendo un fuerte acoplamiento entre ambas.

7.3.3 TRANSFORMACIÓN DE LOS PUERTOS V³STUDIO

Los puertos son los puntos por los que un componente interactúa con el resto de componentes del sistema. Es decir, el componente invoca los servicios que requiere y recibe las peticiones de servicio del exterior siempre a través de ellos. En la figura 7.7 puede observarse que cada uno de los puertos de un componente V³Studio se transforma en una única clase en el modelo UML de destino porque (1) como se mencionó en el apartado 7.2.1 sobre la implementación de los componentes siguiendo el patrón *Active Object*, el puerto desempeña el rol del *proxy* del sistema y (2) se persigue que la transformación genere un modelo sencillo y flexible, por lo que se intenta evitar, en la medida de lo posible, la proliferación de clases.

La clase en que se transforma un puerto está contenida en el mismo paquete que define el componente al que pertenece dicho puerto. Se genera con visibilidad «de paquete» por lo que, al igual que sucede con la clase que contiene la funcionalidad del componente (clase *XXX_Real*), no es visible desde el exterior del paquete en que está definido el componente. Esto contribuye a mantener la encapsulación de la estructura interna del componente.

Para mantenerse fiel a la definición de componente, se decidió que la clase en que se transforma cada uno de los puertos presentes en el componente V³Studio se iba a comunicar únicamente con la clase que representa la funcionalidad del componente (clase *XXX_Real*). En la figura 7.7 puede observarse que esta comunicación se lleva a cabo mediante la creación de varias relaciones de composición entre la clase real y cada una de las clases que representan cada uno de los puertos del componente. De esta forma, se crea la infraestructura necesaria para que el componente (representado por la clase `XXX_Real`) interactúe siempre con el exterior a través de sus puertos.

En el apartado 7.3.2 se comentó que la clase que representa la fachada del componente (clase *Component_Def_XXX*) dispone de operaciones para obtener referencias a los puertos del componente (operaciones `getPort`, consultar apartado 7.3.5). Estas operaciones retornan una referencia a la clase que implementa el puerto, cuyo tipo coincide con el tipo del que hereda el puerto (que está definido en el paquete común `Ports_Package`). Con esta medida se consigue mantener encapsulado el tipo real del puerto, de forma que no se depende de una implementación particular.

La clase en que se transforma un puerto V^{3}Studio se relaciona con la clase que implementa la funcionalidad real del componente (clase *XXX_Real*) mediante una relación de composición. Durante la transformación se crean en la clase real tantas instancias de cada clase puerto como puertos de ese tipo tenga el componente V^{3}Studio, ya que nada impide que un componente tenga varios puertos del mismo tipo. Esta relación se mantiene oculta en el interior del paquete, para evitar que se pueda acceder indirectamente a estas clases desde el exterior del paquete.

A continuación se exponen las dos formas en que se puede transformar un puerto V^{3}Studio ya que, como se describió en el apartado 6.4.2, los puertos de un componentes desempeñan un papel diferente dependiendo del tipo de componente que los contenga (definición de componente simple o de componente complejo). Aunque se trate en dos sub-apartados separados, la transformación de un puerto V^{3}Studio a UML es básicamente similar en ambos casos; tan sólo existen pequeñas diferencias, derivadas del hecho de que el puerto desempeña un rol distinto en cada caso. De cualquier forma, la forma de transformar los puertos V^{3}Studio que se expone a continuación persigue mantener la encapsulación y la cohesión de todos los elementos en que se transforma un componente, a la vez que convierte a los puertos en los únicos puntos de interacción de un componente con el resto de componentes que forman la aplicación.

Además de los dos sub-apartados en los que se describe la transformación del puerto V^{3}Studio, este apartado contiene otros dos sub-apartados más, en los que se muestra un catálogo que muestra algunas de las transformaciones más comunes de puertos con distinto número de interfaces ofrecidas o requeridas. Para terminar este apartado se exponen algunos diseños alternativos que fueron desechados por unas circunstancias u otras.

7.3.3.1 PUERTOS EN UN SIMPLECOMPONENTDEFINITION

Los puertos contenidos en un `SimpleComponentDefinition` cumplen estrictamente con el cometido asignado a los puertos según la definición de componente adoptada por V^{3}Studio, es decir, controlan todas las formas de comunicación del componentes (entrante y saliente, servicios síncronos y asíncronos). Los puertos denominados *Hal_XXX_Port* y *SC_XXX_Port* que aparecen en la figura 7.7 se corresponden con la transformación que se describe en el siguiente enumerado:

- Hereda o bien directamente de la clase abstracta `Port` (en caso de que el puerto no ofrezca ningún servicio) o bien de la clase abstracta definida en el paquete `Ports_Package` (y que a su vez hereda de la clase base `Port`, ver sección 7.2) que implementa todas las interfaces que *ofrece* el puerto. Esta decisión de diseño responde

al requisito de encapsulación del contenido de un componente, ya que de esta forma el tipo real del puerto se mantiene oculto en el interior del paquete. Este diseño proporciona, a la vez, una solución para poder conectar y establecer relaciones entre los distintos puertos de los componentes que se conectan sin tener que utilizar la clase real del puerto (y de este modo romper la encapsulación).

Los métodos que se crean en la clase fachada (clase *Component_Def_XXX*) para acceder a los puertos de un componente desde el exterior del mismo devuelven una referencia de este tipo abstracto común. De esta forma se sigue manteniendo la encapsulación del contenido de un componente. Estas son las razones que justifican la creación del paquete `Ports_Package` junto con todas las clases que contiene.

- Implementa todas las interfaces que aparecen como *requeridas* en el puerto V^{3}Studio. De este modo, la clase *XXX_Real* puede invocar cualquiera de los servicios que aparecen como requeridos en el puerto V^{3}Studio siempre que lo necesite. Este diseño plasma, además, la labor de mediador de las comunicaciones que ejerce el puerto y evita que los servicios requeridos puedan ser invocados desde el exterior del componente. Aunque en realidad no sería necesario que el puerto implementara todas estas interfaces, el hecho de hacerlo facilita la labor del compilador y ayuda en la detección de posibles errores en la transformación final a código. La invocación real del servicio requerido la realiza el componente a través de la referencia que contiene el puerto de salida al puerto conjugado (ver siguiente punto), lo que añade como punto negativo un nivel de indirección a la secuencia de la invocación del servicio.

- Contiene una referencia (relación de navegación en una única dirección) sin inicializar, denominada `conjugate`, al tipo de puerto al que debe conectarse el puerto que se está generando. Esta referencia es, como se ha dicho en el punto anterior, o bien del tipo `Port` (en caso de que el puerto no requiera ningún servicio) o bien del tipo definido en el paquete `Ports_Package` que implementa todas las interfaces que son requeridas por el puerto en cuestión. Esta referencia es inicializada mediante la operación `connect`, que se describe en el apartado 7.3.5.2, y borrada, en caso de que se quiera desconectar este puerto, por la operación `disconnect` (consultar apartado 7.3.5.3). De esta forma se consigue que las comunicaciones entre componente y el exterior se produzcan únicamente a través del puerto correspondiente.

  Obsérvese cómo, en este caso, el tipo de la referencia al puerto conjugado del puerto cuyo código se está generando se elige en función de las interfaces *requeridas*, mientras que la selección de la clase padre de la que hereda se realiza en función de las interfaces *ofrecidas*. De nuevo se hacen patentes las ventajas de disponer del paquete `Ports_Package` para definir los tipos de datos comunes y para mantener la encapsulación de los componentes.

- Contiene una referencia a la clase real (clase *XXX_Real*) que implementa la funcionalidad del componente, denominada `component`. Esta referencia es utilizada para redirigir las peticiones de los servicios ofrecidos por el puerto del componente V^{3}Studio a la clase que realmente los implementa, tal y como exige la definición de «componente» adoptada. De esta forma se consigue que la comunicación con un componente se produzcan únicamente a través del puerto correspondiente.

7.3.3.2 PUERTOS EN UN COMPLEXCOMPONENTDEFINITION

Los puertos contenidos en un `ComplexComponentDefinition` no cumplen estrictamente con el cometido asignado a los puertos, ya que sólo encapsulan las comunicaciones entrantes asíncronas (las síncronas siempre están disponibles y se redirigen automáticamente al puerto correspondiente del componente concreto que ejecuta este tipo de servicio). Este tipo de puertos no controla las comunicaciones salientes, sino que se permite la comunicación directa de los puertos internos con el exterior. Este hecho no sólo no rompe la encapsulación del sistema, ya que la petición de servicios al componente sigue estando controlada por el puerto del componente complejo, sino que mejora la eficiencia del mismo, al evitar una cadena de redirecciones de llamadas a método que se produciría en caso contrario. Los puertos denominados *XYZ_XXX_Port* en la figura 7.7 se corresponden con la transformación que se describe a continuación:

- La regla para determinar la clase base de la que hereda este puerto es la misma que en el caso anterior.
- También se mantiene la regla para determinar las interfaces que implementa la clase.
- El tipo de la referencia `conjugate` se elige en función de las interfaces *ofrecidas* en lugar de las *requeridas*, por lo que en este caso el tipo de `conjugate` y la clase base de la que hereda el puerto coincide. En el contexto de estos puertos, la referencia `conjugate` se utiliza para redirigir directamente al puerto en que se delega las peticiones de servicio síncronas y las operaciones de conexión, desconexión y envío de señales (consultar apartado 7.3.6). Sólo los servicios asíncronos son enviados a la clase `XXX_Real` que define el componente complejo.
- Mantiene la referencia a la clase `XXX_Real`, aunque sólo para enviarle las peticiones de servicios asíncronos.
- Las operaciones `connect` y `disconnect` no actúan sobre el propio puerto del componente complejo sino que redirigen la invocación de la operaciones hacia el puerto correspondiente del componente interno.

- Contiene una operación extra, denominada `setDelegationPort` (consultar apartado 7.3.5.2), con visibilidad «privada» y que admite un parámetro, denominado «to» y de tipo `Class_Port`. Esta operación sirve, como se ha mencionado previamente, para fijar la referencia `conjugate` al valor adecuado (el puerto en que se delega) y es invocada por la operación `portLink` (consultar apartado 7.3.5.4) durante la etapa de conexión de los puertos internos de un componente complejo.

7.3.3.3 CATÁLOGO DE PUERTOS

En este punto se va a proceder a mostrar algunos ejemplos de la transformación de distintos tipos de puertos para ilustrar el proceso descrito hasta ahora. Las figuras 7.8 y 7.9 muestran los diagramas de clases resultantes de aplicar la parte de transformación de puertos V³Studio descrita en este apartado para puertos contenidos, respectivamente, en un `SimpleComponentDefinition` y un `ComplexComponentDefinition`, y que exhiben las siguientes interfaces:

1. Una interfaz ofrecida.
2. Una interfaz requerida.
3. Dos interfaces ofrecidas.
4. Dos interfaces requeridas.
5. Una interfaz requerida y dos ofrecidas.
6. Dos interfaces requeridas y una ofrecida.
7. Una interfaz requerida y otra ofrecida.
8. Dos interfaces requeridas y dos ofrecidas.

Para facilitar la comprensión de los diagramas se han colocado, en verde, las clases bases del framework y en azul las clases e interfaces creadas en los paquetes `Ports_Package` e `Interfaces_Package`, respectivamente. Los ejemplos se han realizado utilizando dos interfaces que figuran como ofrecidas (*Provided_1* y *Provided_2*) y dos interfaces que se utilizan como requeridas (*Required_1* y *Required_2*). En dicha figura se muestra, de arriba a abajo, la transformación de diversos puertos con:

7.3.3.4 EXPOSICIÓN DE ALGUNOS DISEÑOS ALTERNATIVOS

Durante el proceso de diseño de la transformación se consideraron diversas alternativas, algunas de las cuales se van a comentar en este apartado. La primera y, tal vez, más importante decisión de diseño fue convertir los puertos en clases. En un primer diseño se probó a que el componente que se generaba implementara todas las interfaces ofrecidas y tuviera referencias al resto de componentes que completaban las interfaces requeridas, pero rápidamente se descartó esta idea debido fundamentalmente a las siguientes razones: (1) este enfoque dificultaba el diseño cuando un componente exportaba a través de sus puertos varias veces la misma interfaz; (2) el concepto básico «puerto» de CBD se perdía, quedaba disuelto y sin entidad propia en el mar de clases generadas; y (3) dado que se iba

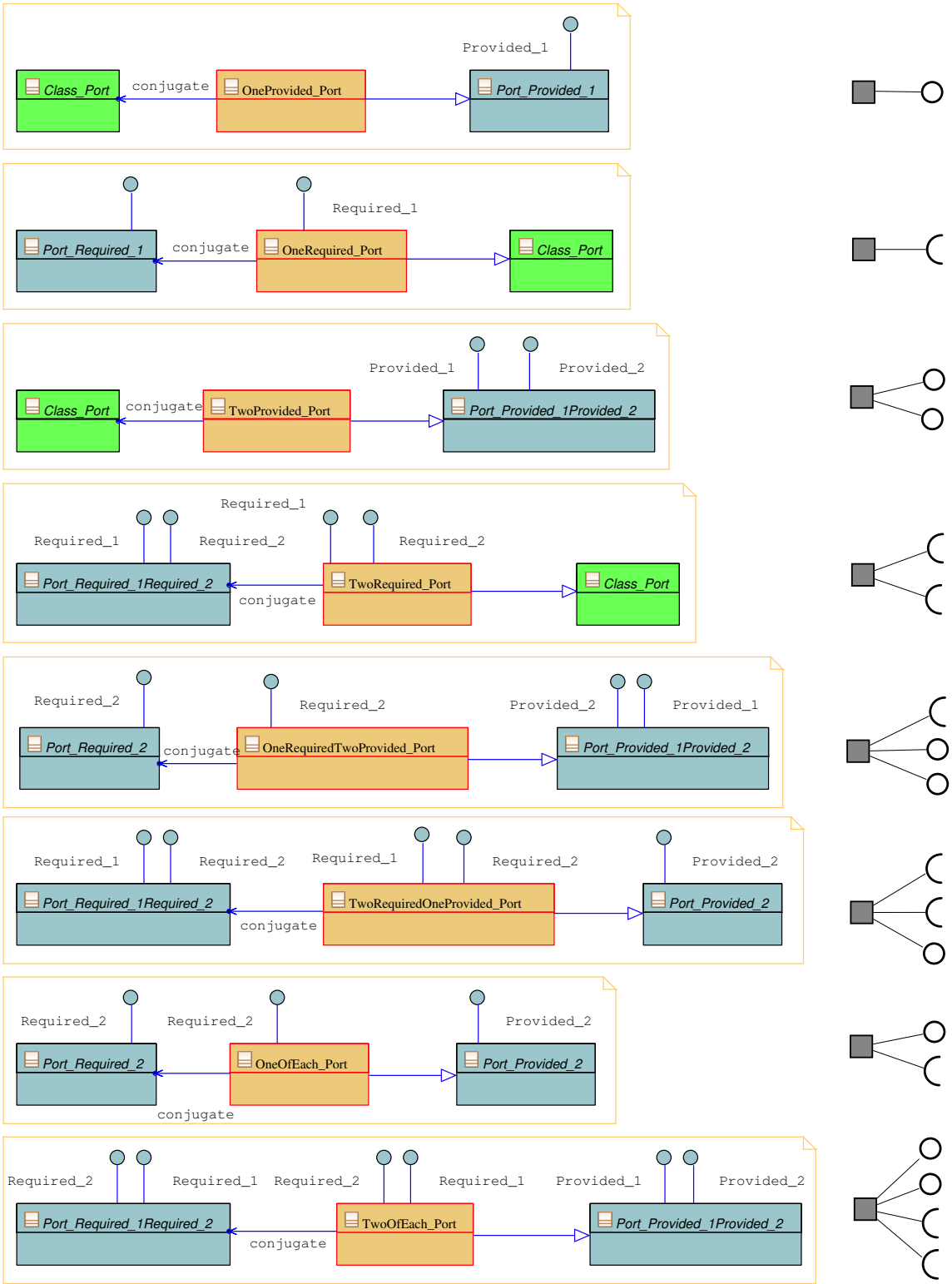


Figura 7.8: Ejemplos de transformación de distintos puertos V³Studio → UML cuando están contenidos en un componente simple. A la derecha se muestra la representación del puerto tal y como aparecería en un diagrama UML.

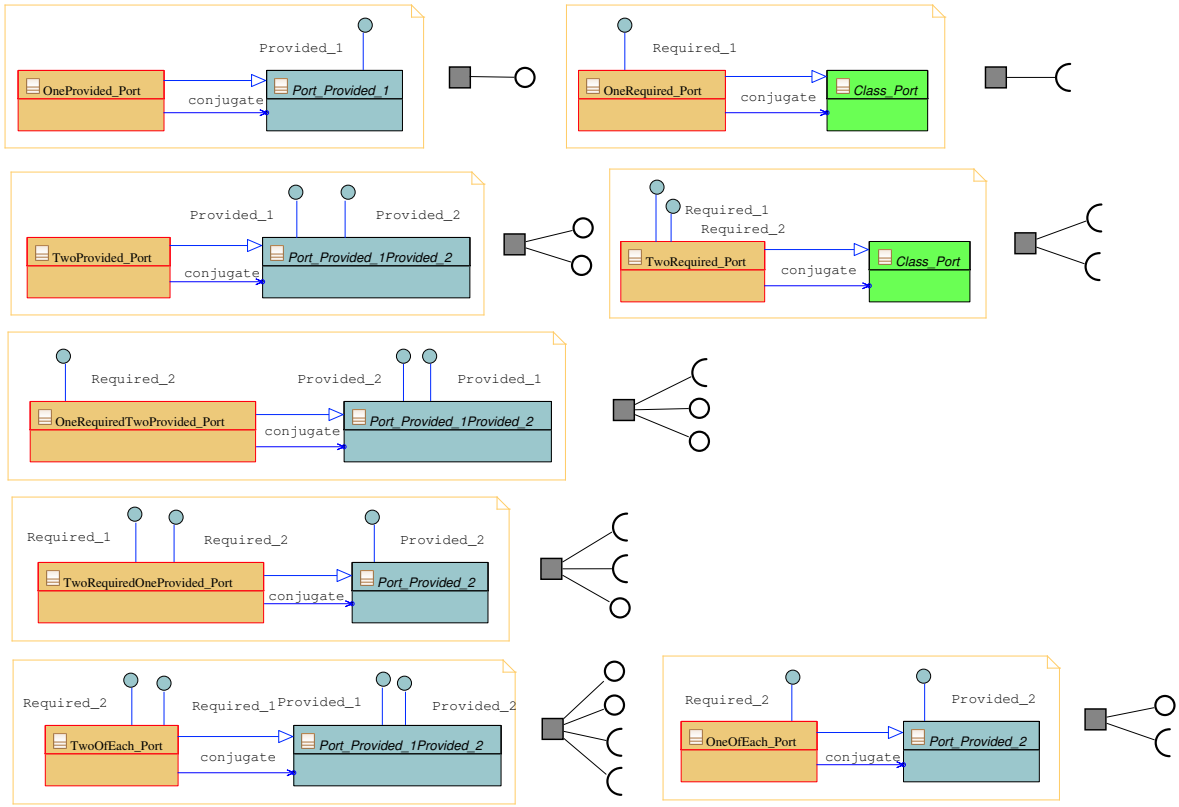


Figura 7.9: Ejemplos de transformación de distintos puertos V³Studio → UML cuando están contenidos en un componente complejo.

a utilizar un lenguaje orientado a objetos, parecía lógico que un concepto tan importante tuviera la misma entidad propia que tenía en el modelo V³Studio. Además, el convertir los puertos en clases da lugar a un diseño más flexible y abierto a futuros cambios.

Tomada esta decisión, tan sólo restaba por aclarar cuántas clases se generarían. Para evitar una excesiva proliferación de clases (la transformación actual ya genera bastantes) se decidió convertir el puerto en una única clase. Otra posible implementación que se estuvo probando fue utilizar genéricos. La principal desventaja de los genéricos es que no son soportados por todos los lenguajes de programación (aunque sí por los más utilizados) y que, realmente, la genericidad en este nivel viene dada por la transformación (que puede ser ejecutada tantas veces como sea necesario). Sin embargo, esta última opción merece un estudio pormenorizado para evaluar si los genéricos proporcionan realmente una implementación más eficiente que la que se consigue actualmente. No se descarta que futuras versiones de V³Studio utilicen genéricos para crear los puertos e incluso los componentes.

Otra decisión de diseño discutida es la aparente violación de la encapsulación de las comunicaciones que se produce en los puertos contenidos en componentes complejos,

ya que éstos sólo controlan la invocación de servicios ofrecidos (los de tipo asíncrono son enviados a la máquina de estados que define el comportamiento del componente complejo, mientras que los síncronos son directamente reenviados al puerto en que delega el puerto). Estos puertos pierden, en todo caso, el control sobre las comunicaciones salientes. Ésto se ha hecho así principalmente (1) porque un componente complejo sólo controla las comunicaciones entrantes (que son las únicas que pueden alterar su estado o el de algunos de sus componentes internos); (2) para reducir el número de indirecciones en la invocación de servicios requeridos y (3) para evitar que la máquina de estados tenga que sobrecargarse controlando si los componentes internos pueden invocar un servicio requerido o no (lo que, por otro lado, tampoco tiene mucho sentido).

Por último, resta por comentar la idea de que la clase en que se transforma un puerto V^{3}Studio podría haberse creado con la referencia al puerto conjugado inicializada desde un principio, puesto que esta información está disponible en el modelo V^{3}Studio. Esta decisión descargaría a la clase principal de la parte de código que realiza esta conexión y evitaría la generación de las operaciones para llevarla a cabo. Pero se decidió optar por el primer diseño para (1) obtener un sistema más flexible de cara a futuras mejoras; (2) permitir que los puertos de los componentes se pudieran reconectar a otros puertos de otros componentes; y (3) para plasmar de la forma más fiel posible la definición de componente, que es independiente del resto de componentes que puedan formar el sistema.

7.3.4 CONSTRUCTORES DE COMPONENTES Y PUERTOS

Este apartado presenta los diagramas de actividades que implementan los constructores de todas las clases que se han presentado hasta el momento en la vista arquitectónica, i.e. componentes (clase real y clase fachada) y puertos. Siguiendo las pautas de diseño software que se indicaron al principio de esta sección, los constructores son los encargados de crear no sólo el propio objeto sino también todos los objetos que éste pueda contener mediante una relación de composición. Además, en el constructor se añaden todos los parámetros necesarios para inicializar todas las referencias cuyo atributo `aggregation` se haya fijado al valor `shared`. En todos los diagramas de este apartado se puede observar que el objeto finalmente creado se retorna explícitamente, por medio de un `ActivityParameterNode`, a la operación que invoca el constructor. Se recuerda que en el apéndice B se puede encontrar un descripción de las acciones utilizadas en los diagramas de actividad que se muestran a lo largo del presente capítulo.

La figura 7.10 muestra los diagramas de actividad de la implementación del constructor de la clase fachada para el componente simple *SC* (figura superior) y para el componente complejo *XYZ_Robot* (figura inferior), mientras que la figura 7.11 muestra una

captura del entorno Eclipse en la que puede observarse todas las clases que conforman el diagrama de actividad que describe el constructor del componente *SC*. Los diagramas de actividad que muestran la implementación de los constructores de la clase real y la clase fachada plasman únicamente la descripción de la transformación de los componentes expuesta en el apartado 7.3.2. En estas figuras se puede observar la secuencia de creación de la clase fachada:

1. Creación del objeto de la clase fachada mediante la acción *CObA*.
2. Invocación de la operación (acción *COpA*) que representa el constructor de cada una de las propiedades contenidas en la fachada. En este caso únicamente se necesita crear un objeto, que se corresponde con la clase real que representa al componente.
3. Asignación de los objetos creados a la propiedad correspondiente del objeto contenedor mediante la acción *ASFVA*. En este caso, sólo se asigna la propiedad denominada `real` (consultar figura 7.7).
4. En la generación del diagrama de actividades que describe el constructor de la clase fachada no existen propiedades modeladas como `shared`, por lo que estos constructores no tienen parámetros de entrada. El constructor del puerto (ver figura 7.13) muestra un ejemplo en el que se pasan parámetros al constructor.

Por otro lado, la figura 7.12 muestra los diagramas de actividad correspondientes a los constructores de la clase real de estos mismos componentes V^{3}Studio. En este caso se puede observar que el diagrama es más complejo, aunque se siguen los mismos pasos que en el caso anterior. Es decir primero se crea el objeto de la clase real (mediante la acción *CObA*), posteriormente se invocan los constructores de todas las propiedades que contiene (mediante la acción *COpA*) y, por último, se asigna cada uno de los objetos creados a la referencia correspondiente mediante la acción *ASFVA* y se retorna el objeto resultante. En el caso del componente simple (parte superior de la figura 7.12) se crean únicamente los objetos que representan los puertos que contiene, mientras que en el componente complejo (parte inferior de la figura 7.12) se crean, además, los objetos que representan los componentes (`Component` V^{3}Studio) que forman este componente complejo.

Finalmente, la figura 7.13 muestra el diagrama de actividades correspondiente al constructor de uno de los puertos del sistema según lo expuesto en el apartado 7.3.3. En este caso, el diagrama de actividades muestra sólo la creación del objeto de la clase puerto correspondiente y la asignación del valor que se pasa como parámetro al constructor (`comp`) a la propiedad `component` de la clase del puerto. En este caso, el puerto no contiene ningún objeto, por lo que en la implementación del diagrama no se invoca ningún constructor adicional.

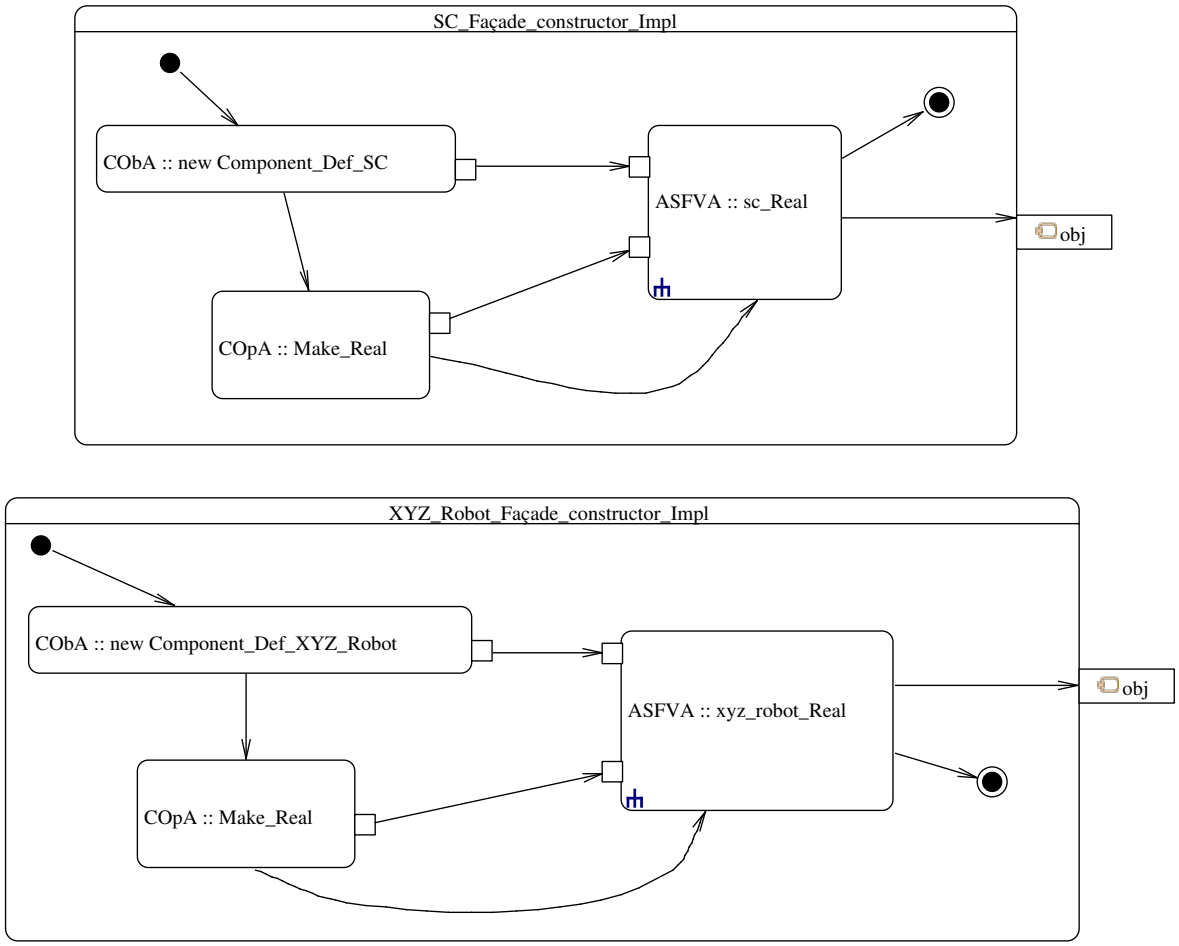


Figura 7.10: Diagramas de actividades que muestra la implementación del constructor de la clase fachada de un componente (`Component_Def_XXX`). El diagrama superior muestra el constructor de un componente simple mientras que el inferior se corresponde a un componente complejo.

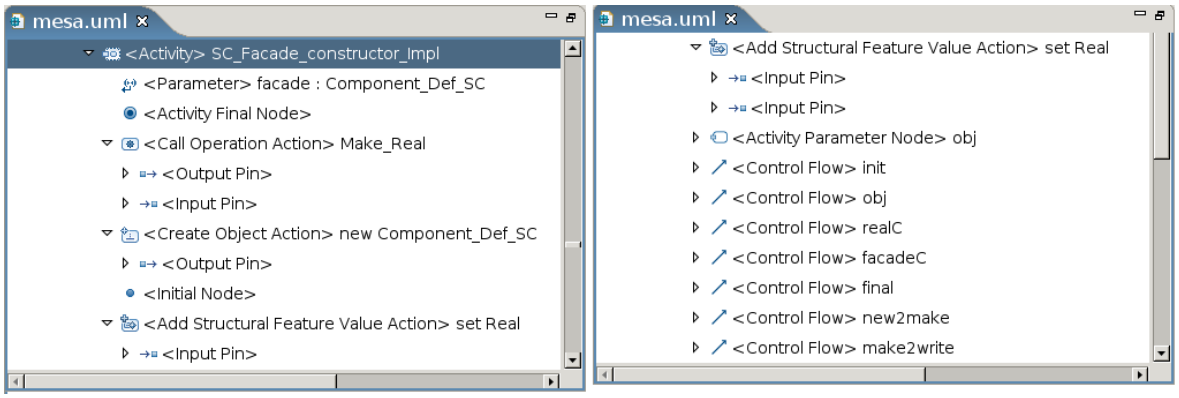


Figura 7.11: Captura del entorno Eclipse que muestra la estructura de clases UML que describe la operación `SC_constructor_Impl`.

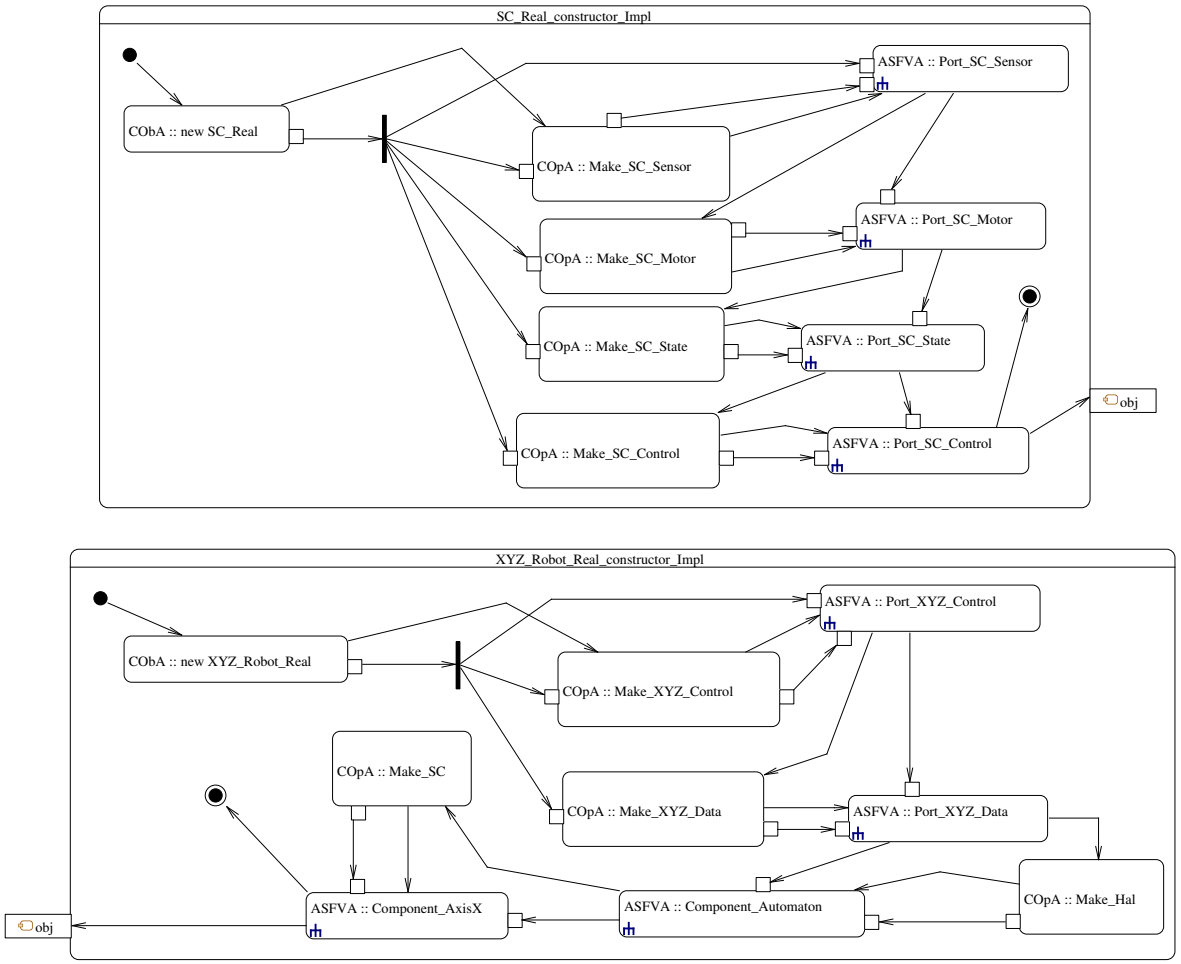


Figura 7.12: Diagramas de actividades que muestra la implementación del constructor de la clase real de un componente (`XXX_Real`). El diagrama superior muestra el constructor de un componente simple mientras que el inferior se corresponde a un componente complejo.

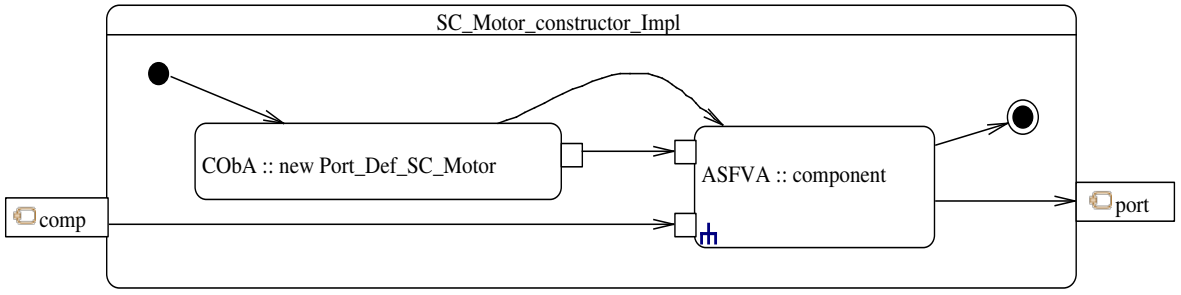


Figura 7.13: Diagrama de actividades que muestra la implementación del constructor de un puerto.

7.3.5 Operaciones de conexión de los puertos V³Studio

En este apartado se describen las operaciones relacionadas con la gestión de la conexión de los puertos que se crean durante la transformación de la vista arquitectónica. Estas operaciones son generadas automáticamente gracias a la información disponible en el modelo y a la semántica asociada a los conceptos del meta-modelo (consultar apartado 7.2). De todas las operaciones que se generan durante la transformación, en este apartado se describe la implementación de las operaciones `getPort`, `connect`, `disconnect`, `setDelegationPort` y `portLink`.

En la generación de estos diagramas de actividades se hace uso de las características estructurales de las clases que definen un componente: clase fachada de componente y sus atributos; clase componente real y sus atributos; clases definición de los puertos reales del componente y clases que definen los tipos comunes para los puertos, conceptos que ya han sido descritos en secciones anteriores. De nuevo se hará uso del ejemplo presentado al principio de la sección (figura 7.1) para ilustrar el resultado de esta parte de la transformación. Se recomienda al lector que consulte el diagrama de clases generado para el ejemplo (figura 7.7, página 184), ya que en los siguientes ejemplos se hará uso intensivo de él.

7.3.5.1 Operación getPort

Esta operación se genera en la clase `Component_Def_XXX` (la que representa la fachada del componente) y su misión es retornar una referencia, obtenida mediante su atributo `real_XXX`, al puerto en cuestión. Por tanto, en la clase fachada se generan tantas operaciones de este tipo como clases `Port` contenga el componente V³Studio. Además, al formar parte de la interfaz de la fachada, estas operaciones pueden ser invocadas desde el exterior del paquete en que se define el componente. Concretamente, esta operación es invocada por la operación `portLink` (descrita en el apartado 7.3.5.4) para llevar a cabo la conexión entre los puertos de un componente complejo.

La figura 7.14 muestra dos diagramas de actividad tipo de la implementación de esta operación, obtenidos como resultado de la aplicación de la transformación al modelo de componentes de ejemplo. La operación `getPort` retorna una referencia al puerto en cuestión cuyo tipo se corresponde o bien con uno de los tipos comunes definidos en el paquete `Ports_Package`, o bien con la clase abstracta `Port`. En este caso es necesario realizar una conversión de tipos, ya que la clase real del puerto está oculta en el interior del paquete y, por tanto, no puede ser directamente utilizada por el resto de componentes del sistema. La secuencia de acciones ejecutadas en el diagrama es la siguiente:

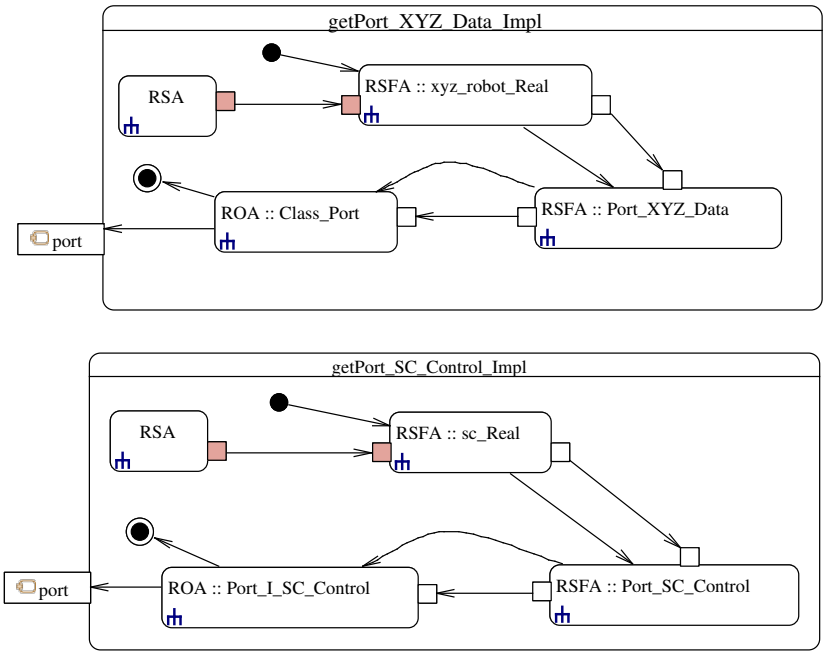


Figura 7.14: Diagramas de actividad tipo que describen la operación `getPort`

1. Obtención de la clase real que implementa la funcionalidad del componente. Para ello se utiliza la acción *RSFA* para obtener la propiedad `xxx_real` del objeto «self», que es obtenido mediante la ejecución de la acción *RSA* asociada a uno de los `ActionInputPin` (representado en rojo) de la acción *RSFA*..

2. Obtención del puerto requerido de la clase real mediante la segunda acción *RSFA*.

3. Ejecución de la acción de conversión (*ROA*) para convertir el tipo del puerto a uno de los tipos comunes creados en el paquete `Ports_Package`.

En este caso también se muestra una captura de pantalla (ver figura 7.15) de las clases UML que representan esta actividad porque se quiere resaltar la utilización, por primera vez, de la clase `ActionInputPin` (recuadrada en la figura). En esta figura se puede observar que este pin contiene una acción, que no tiene ningún pin de entrada y con un único pin de salida, cuyo valor se asocia posteriormente al *token* de entrada del `ActionInputPin`.

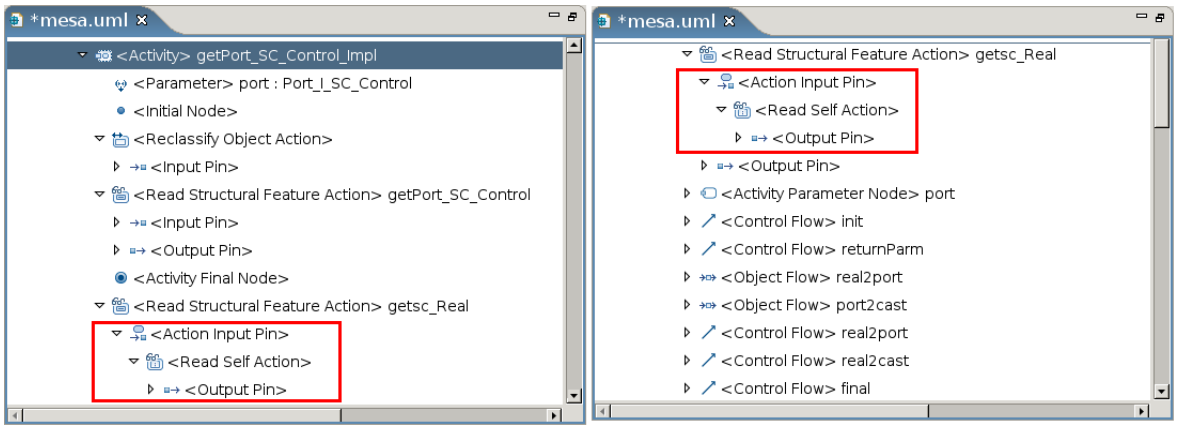


Figura 7.15: Captura de pantalla del entorno Eclipse que muestra las clases UML que implementan el diagrama de actividad de la operación `getPort`. Esta captura hace hincapié en el uso de la clase `ActionInputPin`, que aparece recuadrado.

7.3.5.2 Operación connect y setDelegationPort

Como ya se mencionó en el apartado 7.3.1, esta operación está definida en la clase abstracta `Port`, por lo que forma parte de la interfaz de todos los puertos que se crean durante la transformación. Esta operación admite un parámetro (denominado «`to`» y de tipo `Port`) que sirve para inicializar la referencia al puerto conjugado (`conjugate`) que contiene cada puerto (consultar el apartado 7.3.3). La operación `connect` está definida en la clase base `Port` con visibilidad «pública», por lo que puede ser invocada desde el exterior del componente. Concretamente, esta operación es utilizada por la operación `portLink` para llevar a cabo la conexión entre los puertos de los componentes internos de un componente complejo.

La figura 7.16 muestra cuatro diagramas de actividad tipo obtenidos como resultado de aplicar la transformación al modelo de componentes del ejemplo que se muestra en la figura 7.1. En este caso la variabilidad es doble, ya que (1) estas operaciones se comportan de forma distintas dependiendo del tipo de definición de componente en que se encuentra el puerto (consultar apartado 7.3.3) y (2) al igual que sucedió en la operación `getPort`, existen dos tipos de implementación de la operación `connect`: o bien el puerto espera que su conjugado sea del tipo `Class_Port` o bien se realiza un *cast* al tipo que espera el puerto (definidos, como siempre, en el paquete de tipos `Ports_Package`). En estos diagramas de actividad se puede observar el uso localizado que se realiza del `ActionInputPin` para obtener la referencia a «self» (acción *RSA* y pines rojos), así como la asignación del valor del parámetro del puerto conjugado (parámetro «to») a la correspondiente referencia `conjugate`. También puede observarse que la acción que realiza el *cast* (acción *ROA*) sólo es añadida en caso necesario.

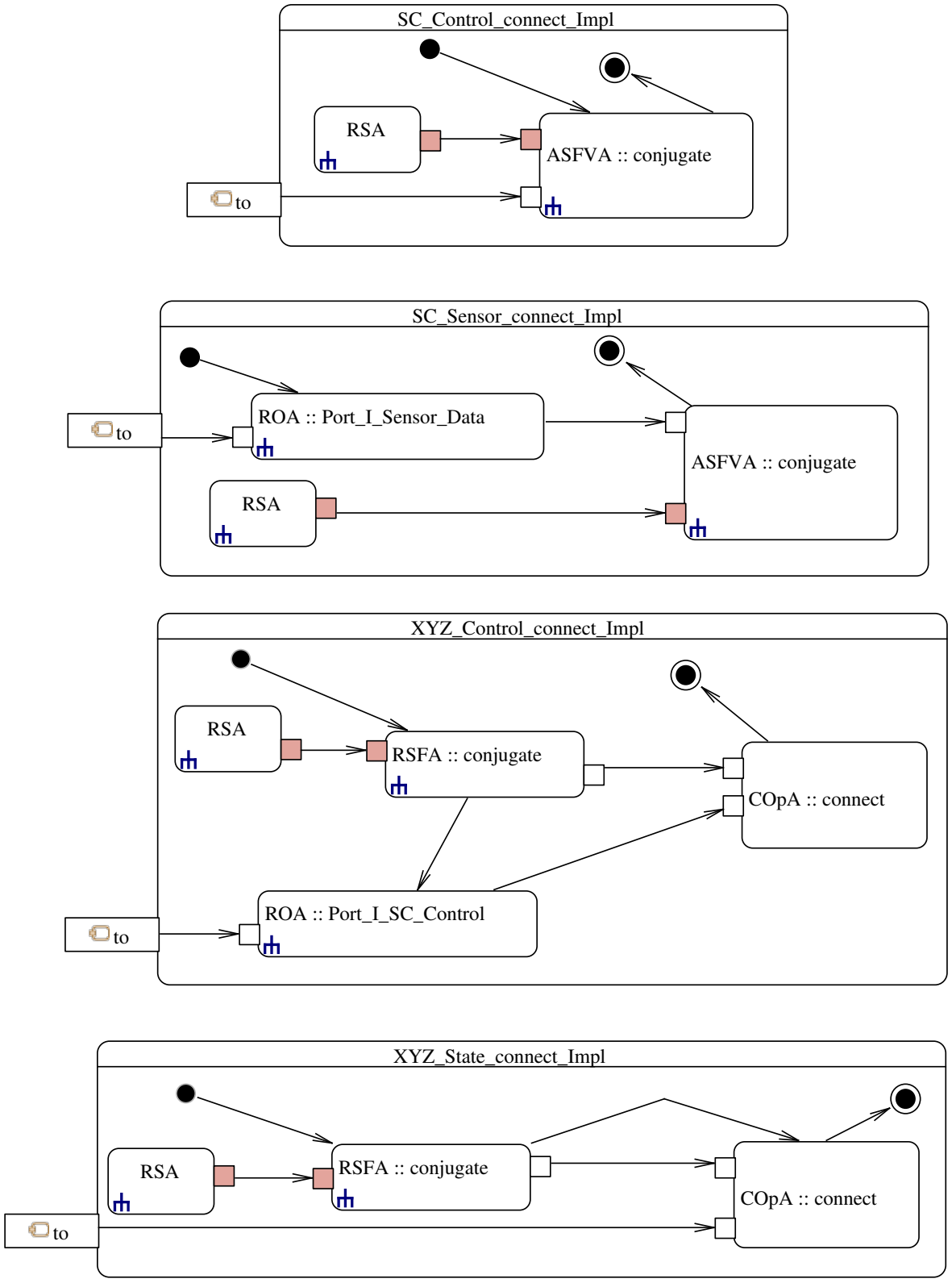


Figura 7.16: Diagramas de actividad tipo que describen la operación `connect`. Las dos figuras superiores representan las operaciones sobre puertos contenidos en componentes simples, mientras que las dos inferiores se corresponden a puertos contenidos en componentes complejos. En ambos casos se muestran los escenarios en que es necesario realizar un «cast».

Por otro lado, el diagrama de actividades que describe la implementación de la operación `setDelegationPort` (que es generada cuando se transforma un puerto contenido en un componente complejo) es similar a los que se generan en el caso de la operación `connect`, ya que el propósito es el mismo y lo único que realmente cambia es el nombre y la visibilidad de la operación, que pasa a ser privada para evitar que pueda ser invocada desde el exterior del componente.

7.3.5.3 Operación disconnect

Esta operación está definida en la clase abstracta `Port`, por lo que forma parte de la interfaz de todos los puertos que se crean durante la transformación, con visibilidad «pública», por lo que puede ser invocada desde el exterior del componente. Esta operación permite desconectar los puertos conectados de dos componentes.

La figura 7.17 muestra dos diagramas de actividad obtenidos como resultado de aplicar la transformación al modelo de componentes del ejemplo y muestra los dos casos posibles: cuando el puerto está contenido en la definición de un componente simple y cuando está contenido en la definición de un componente complejo.

7.3.5.4 Operación portLink

Esta operación aparece únicamente en la clase que define la fachada de un componente complejo. La actividad que describe la implementación de esta operación invoca las operaciones anteriormente descritas (`getPort`, `setDelegationPort` y `connect`) secuencialmente para inicializar cada una de las referencias al puerto conjugado de cada puerto, dependiendo del tipo de componente que lo contenga. La generación del diagrama utiliza la información de conexión contenida en el concepto `PortLink` del modelo V³Studio para enlazar correctamente la ejecución de las operaciones. El diagrama de actividades no procesa aquellos puertos que no están conectados (y que por tanto no tienen interfaces requeridas) e invoca, antes de comenzar la conexión de sus propios componentes, la operación `portLink` en aquellos componentes que son de tipo complejo (`ComplexComponentDefinition`). La figura 7.18 muestra el diagrama de actividad asociado a la operación portLink que se genera para el componente complejo *XYZ_Robot* de ejemplo. La secuencia de acciones que se ejecuta es la siguiente:

1. Obtención del objeto que representa el componente real mediante la acción *RSFA*. En este caso se utiliza un `ActionInputPin` para obtener la referencia a «self» sobre la que ejecutar esta acción.

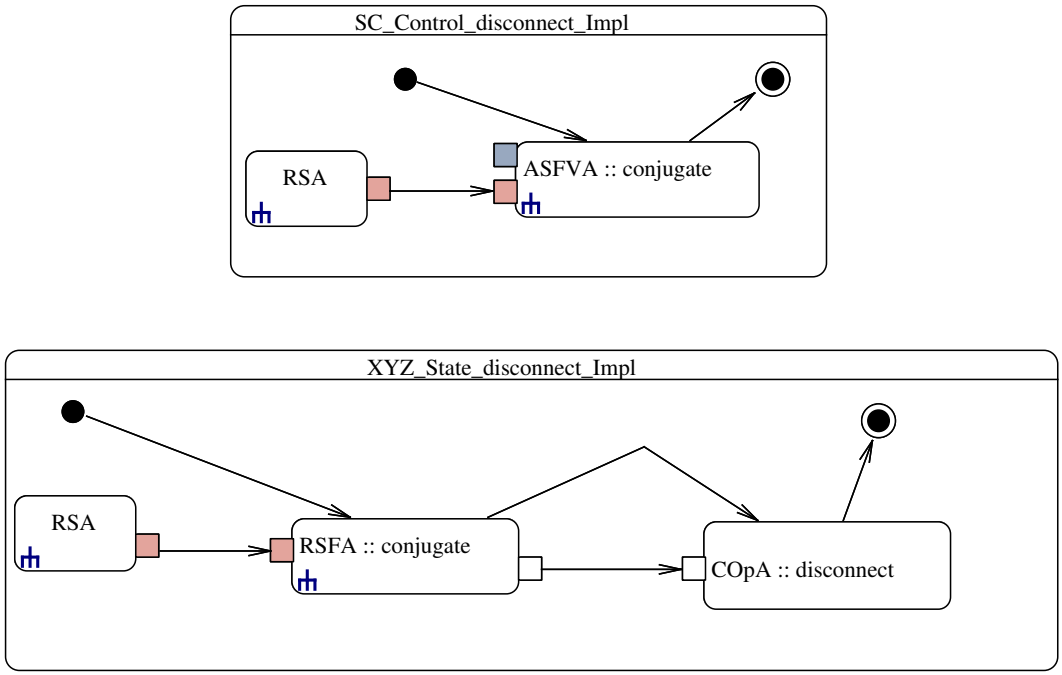


Figura 7.17: Diagramas de actividad que describen la operación `disconnect`. El pin de color azul marca un `ValuePin`, que en este caso suministra el valor *null*, con el que se reasigna el valor de la propiedad `conjugate`.

2. Se utiliza el `ForkNode` para clonar el *token* que representa el componente real y enviarlo a un conjunto de acciones *RSFA*, en las que se van a obtener cada uno de los componentes que forman parte del componente complejo.

3. Los *tokens* que representan los componentes de la definición del componente complejo son clonados de nuevo, mediante un `ForkNode`, para obtener cada uno de los puertos que tiene el componente. Para ello se utiliza la acción *COpA* para invocar la correspondiente operación `getPort` (consultar apartado 7.3.5.1).

4. Por último, un nuevo `ForkNode` permite clonar los *tokens* que representan cada uno de los puertos y relacionarlos correctamente con la acción *COpA* que invoca la operación `connect` para terminar la ejecución de la operación `portLink`.

5. En el caso de los puertos que están conectados con los puertos de un componente complejo (en el caso del ejemplo, los puertos denominados *SC_Control* y *SC_State*), el último `ForkNode` se conecta con la acción *COpA* que invoca la operación `setDelegationPort`, para de esta forma «unir» los puertos del componente complejo con los puertos en que va a delegar las peticiones de servicio en tiempo de ejecución.

En la figura 7.18 se observa que el uso del `ForkNode`, en contraposición al uso de la clase `ActionInputPin` (ver apéndice B.3) facilita la lectura y comprensión del diagrama

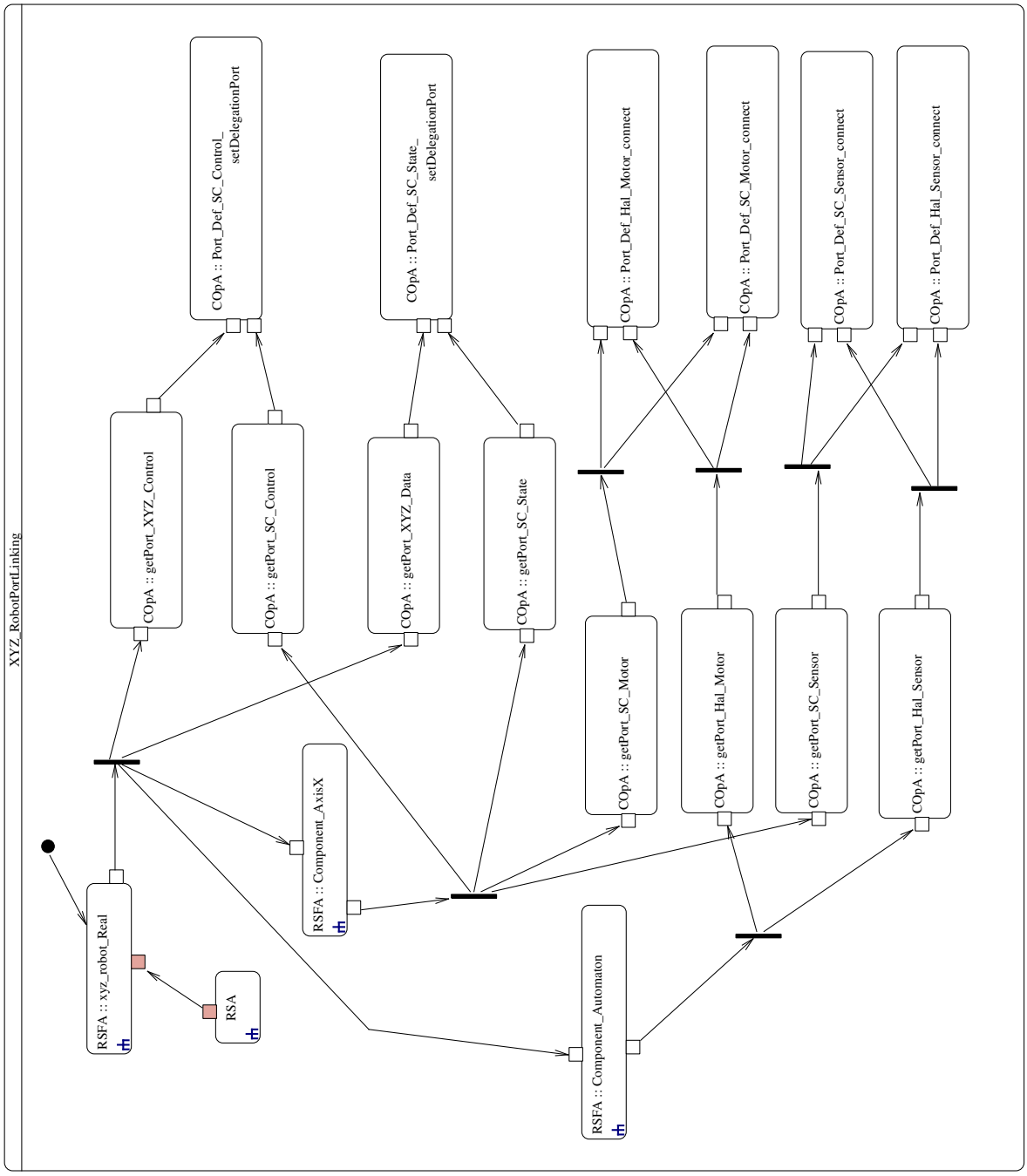


Figura 7.18: Diagrama de actividad que describe la operación `portLink`. Se ha omitido el flujo de control para facilitar el entendimiento de este diagrama de actividades.

al permitir la existencia de un único flujo de control que se va ramificando. Como ya se mencionó en la sección 6.4, la interconexión de los puertos compatibles (i.e. aquellos cuyos servicios provistos aparecen como requeridos en el conjugado y viceversa) de dos componentes se lleva a cabo mediante el uso de dos conceptos: `PortLink` y `BindingEnd`. La información que aportan estos dos conceptos es utilizada únicamente en el diagrama de actividades que describe esta operación y sólo para inicializar la conexión de los puertos de los componentes.

En este punto se va a comentar una alternativa de diseño que se desechó en su momento. Existen dos posibilidades para la generación de la actividad `portLink` cuando el componente complejo tiene varias instancias de una misma definición de componente. La primera posibilidad consiste en generar las acciones `getPort` y `connect` para una definición de componente y luego seleccionar, mediante la acción `ReadIsClassified` y el `ConditionalNode`, los puertos de los *tokens* que identifican a los componentes cuyos puertos tienen que ser conectados. La segunda posibilidad (y la que se ha elegido finalmente) consiste en la regeneración de todas las acciones `getPort` y `connect` para cada componente, independientemente de que ya hayan sido previamente generadas para otra instancia de la misma definición. Esta elección provoca que el diagrama de actividades sea mucho mayor que en el caso anterior, pero facilita la etapa posterior de generación de código, ya que el diagrama está completamente desarrollado y las acciones que se realizan son simples y siempre afectan a un único objeto (*token*).

7.3.6 OPERACIONES DE COMUNICACIÓN ENTRE COMPONENTES

Finalmente, en esta sección se presentan los diagramas de actividad que describen la implementación de diversas operaciones de comunicación entre componentes. Como ya se ha comentado, la implementación de estas operaciones se desprende de los patrones de implementación y las decisiones de diseño que se describieron previamente en el apartado 7.2.1. Concretamente, en este apartado se describen los diagramas de actividades que se generan como implementación de las operaciones que representan:

Invocación de un servicio síncrono. La figura 7.19 muestra los dos diagramas de actividad que se generan. Los servicios síncronos se ejecutan inmediatamente, en el hilo de ejecución del componente que lo solicita y sin control por parte de la máquina de estados del componente que lo proporciona. Por estas razones, la invocación de un servicio síncrono se puede ver como una cadena de invocaciones que va atravesando todos los puertos por los que pasa, que tan sólo redirigen la llamada, hasta llegar finalmente al componente que lo implementa (clase `XXX_Real`). En este punto, el componente real tiene finalmente que resolver la petición. Para ello se han añadido

a dicha clase tantos elementos `OpaqueBehavior` (para que generen operaciones a rellenar por el desarrollador) como servicios síncronos implementa el componente.

Invocación de un servicio asíncrono. La figura 7.20 muestran los diagramas de actividad que se generan para describir la implementación de un servicio asíncrono. Los servicios asíncronos son encolados en la máquina de estados del componente que proporciona el servicio, para ser ejecutados dependiendo del estado del componente. En este caso, no se realiza distinción alguna entre componentes simples y componentes complejos, ya que en ambos casos la petición de servicio asíncrono acaba siendo encolada en la máquina de estados correspondiente.

Las señales que se crean cuando se invocan estos servicios son finalmente implementadas por medio del patrón de diseño *Command*. De esta forma, se crea un objeto cuyo tipo se corresponden con el de la señal, tal y como se define en el paquete `Signals_Package` (ver apartado 7.2.4).

Operaciones de notificación. Las figuras 7.21 y 7.22 muestran el diagrama de actividad que describe la implementación de estas actividades, cuyo objetivo es permitir el intercambio de señales entre componentes. Estas señales se utilizan principalmente para notificar al componente interesado la finalización de alguna de las peticiones de servicios asíncronos, y puede contener valores producidos tras su ejecución. Aunque en un futuro se podría ampliar su alcance.

7.4 V^{3}Studio → UML: Vista Comportamiento

En esta sección se describe la parte de la transformación encargada de transformar la vista de definición de comportamiento a su equivalente en UML y relacionarla con las clases que se han generado hasta el momento. Esta es una sección breve, ya que la transformación de la vista de comportamiento respeta la forma UML de la máquina de estados y no realiza, como sería de esperar, una transformación concreta a un diagrama de clases y actividades que describa la implementación de la misma en un lenguaje orientado a objetos. En este caso se ha optado por mantener la forma de la máquina de estados en la transformación, para de esta forma ser capaces en un futuro de realizar distintas transformaciones de la máquina de estados (por ejemplo, las que enumera Gamma al describir el patrón de diseño *State* [96]). Debido a que la vista de comportamiento de V^{3}Studio está ampliamente basada en el diagrama UML de máquinas de estados, la transformación entre ambos diagramas, como se describe en esta sección, es prácticamente directa.

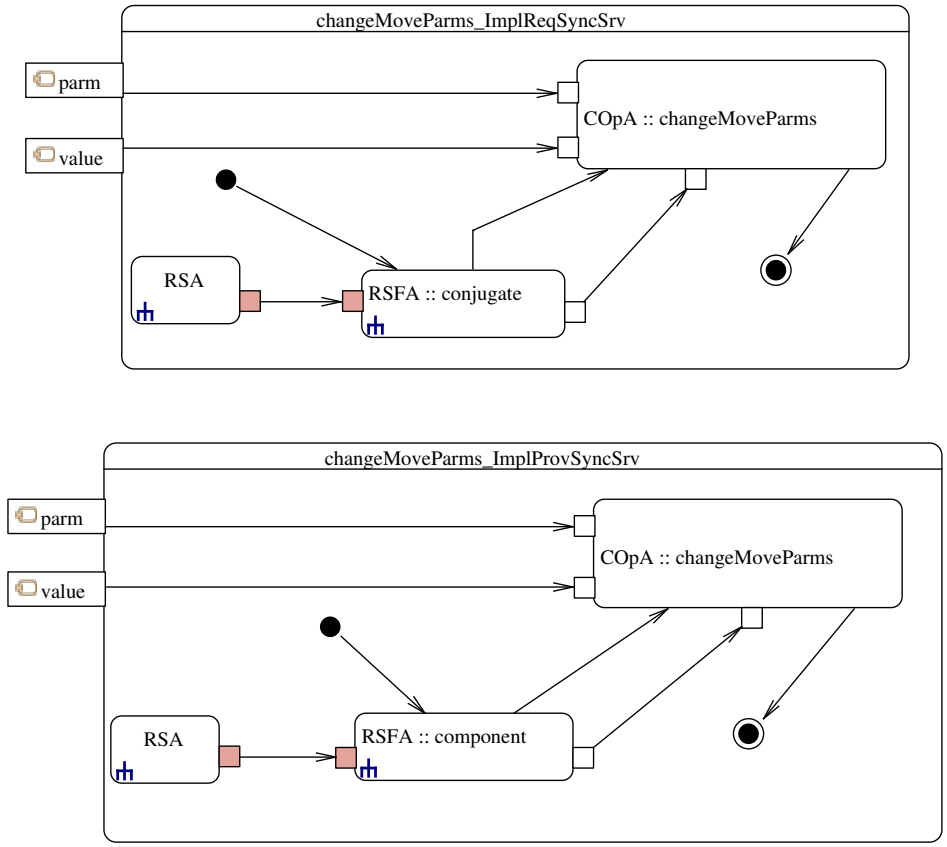


Figura 7.19: Diagramas de actividad que describen la petición de un servicio síncrono. El diagrama superior corresponde a la petición de servicio a través del puerto conjugado, mientras que el inferior representa la invocación de la operación del componente (clase `XXX_Real`) que implementa el servicio. Se recuerda que los servicios síncronos están siempre disponibles y no pasan por la máquina de estados del componente. En el caso de que el componente que implementa el servicio síncrono sea complejo, el puerto que recibe la petición la redirige al puerto interno correspondiente, utilizando un diagrama de actividades similar al mostrado en la parte superior.

En esta parte de la transformación de modelos se genera la versión UML de la máquina de estados V³Studio. Tras esta conversión la máquina de estados resultantes se asocia a la clase que describe la funcionalidad del componente (clase `XXX_Real`) mediante la relación de asociación *classifierBehavior*, como se describió en el apartado 7.2.2.

La figura 7.23 muestra una captura de pantalla del entorno Eclipse en la que se pueden observar la estructura de clases V³Studio que conforman el modelo de ejemplo de la máquina de estados que describe, en este caso, el comportamiento del componente *SC*. En esta captura puede observarse también que el modelo de máquina de estados V³Studio hace uso de la definición de las interfaces y servicios que se modelaron en las primeras etapas del desarrollo. Este detalle es importante porque plasma la estrecha relación que existe entre todas las vistas/modelos ya que, aunque se pueden utilizar por separado, todas modelan el mismo sistema. En el caso de las máquinas de estados, los servicios de las

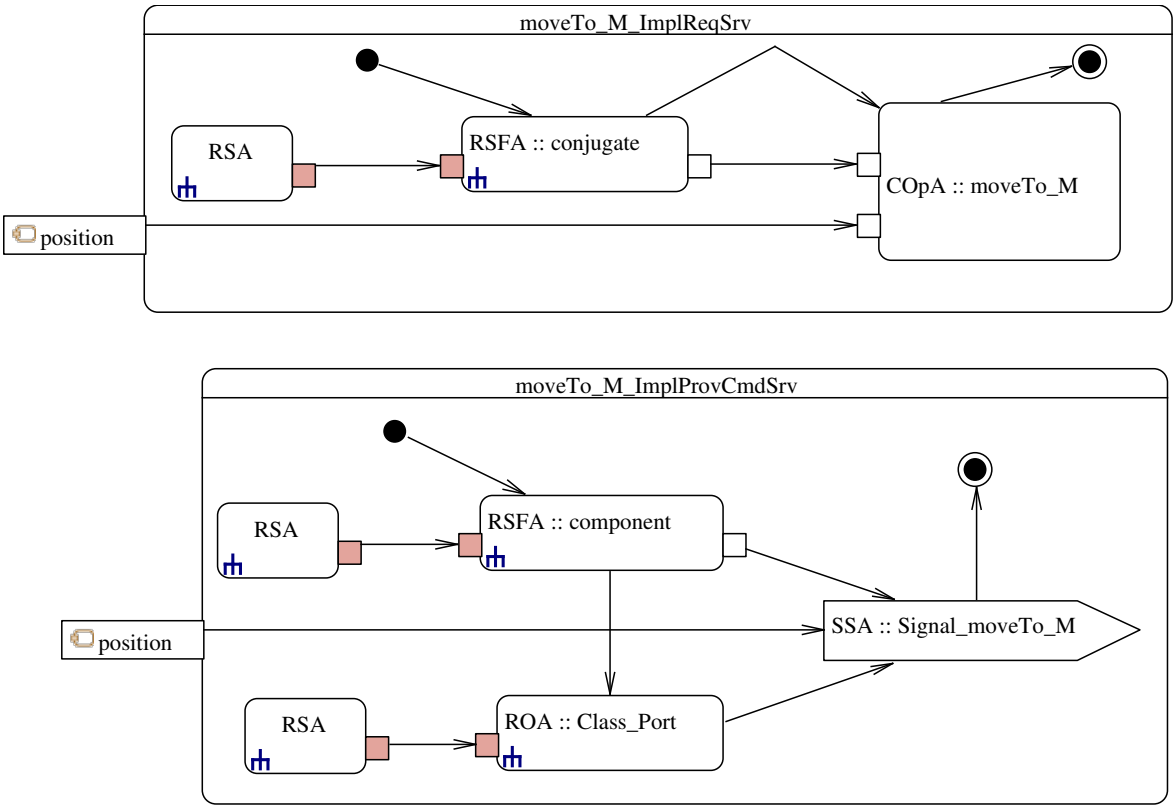


Figura 7.20: Diagramas de actividad que describen la petición de un servicio asíncrono. El diagrama superior corresponde a la petición de servicio a través del puerto conjugado, mientras que el inferior representa el envío de la correspondiente señal a la máquina de estados del componente (consultar sección 7.4). En este caso, los dos tipos de definición de componente se comportan exactamente igual. Debido a limitaciones en la herramienta UML Tools, la acción `SendSignalAction` no muestra los pines de entrada, a través de los cuáles se suministra el puerto por el que ha llegado la petición y los posibles valores.

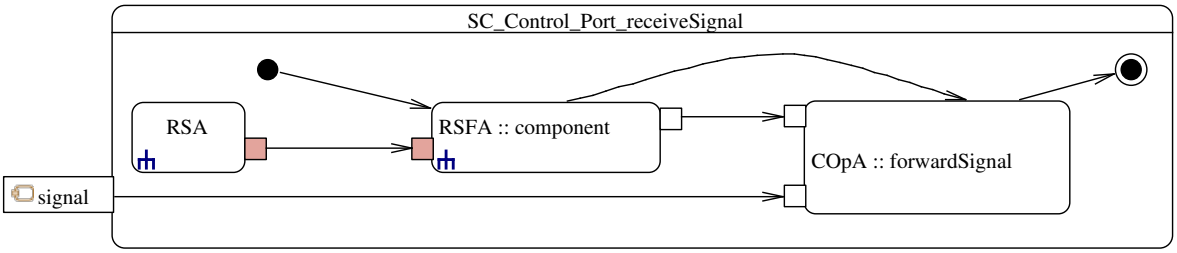


Figura 7.21: Diagrama de actividad que describe la operación `receiveSignal` de un puerto

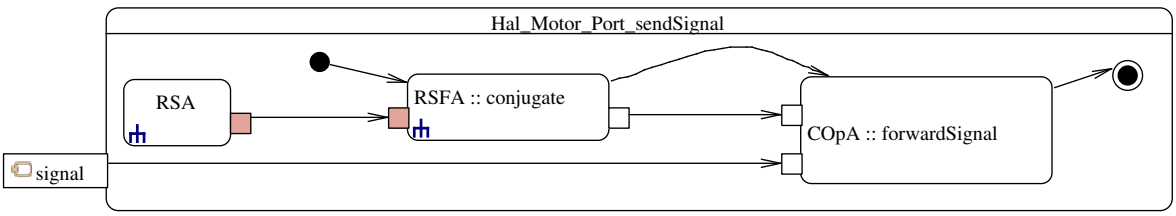


Figura 7.22: Diagrama de actividad que describe la operación `sendSignal` de un puerto

interfaces (provistas) se utilizan como disparador de las transiciones que hacen evolucionar el componente en el tiempo. La figura 7.24 muestra de forma gráfica la máquina de estados, una vez se ha completado la transformación (que, como puede verse en dicha figura, es bastante directa).

Aunque no aparecen en ninguna de las dos figuras, los disparadores (EClass `Trigger`) y las guardas (`Constraint`) asociadas a las transiciones (`Transition`) en V^{3}Studio también se transforman para generar el modelo UML final. El caso de la guarda es directo, ya que en UML también es posible asociar guardas a las transiciones y fijar un texto libre (clase `StringExpression` en UML) como descripción de la misma. Sin embargo, en el caso de los disparadores es necesario crear los eventos y las señales que se van a utilizar para establecer el flujo de estados en la máquina de estados y relacionar las peticiones de servicio con la activación de las transiciones. En este punto, la clase `Trigger` desempeña un papel fundamental, ya que modela la activación de una `Transition`. Concretamente, la transformación de un `Trigger` V^{3}Studio genera los elementos en el modelo UML que se enumeran a continuación:

- Como se comentó en el apartado 7.2.4, durante la transformación de modelos se genera una paquete denominado `Signals_Package` que contiene todas las señales y eventos (representados por las clases UML `Signal` y `ReceiveSignalEvent` respectivamente) que se utilizan en las máquinas de estados que describen el comportamiento de los componentes.

- Se genera una señal por cada servicio de tipo *asíncrono* que aparece en la aplicación, ya que únicamente los servicios asíncronos pueden provocar el cambio de estado en la máquina de estados. Estas señales representan cada una de las operaciones del componente, y su recepción por parte de la clase que representa al componente real provoca la generación y el encolado del correspondiente evento (ver siguiente punto) en la máquina de estados del componente. Posteriormente, de forma asíncrona, el procesamiento de este evento puede producir el cambio de estado en la máquina de estados o ser simplemente desechado.
  La clase `Signal` en UML hereda de `Classifier`, por lo que es posible añadir atributos a las señales. En concreto, todas las clases `Signal` que se generan tienen un atributo denominado `incomingPort` que sirve para almacenar el puerto por el que llegó la petición de servicio. De esta forma, el componente puede utilizar posteriormente las facilidades de notificación de señales embebidas en la transformación (operaciones `sendSignal` y `receiveSignal` de los puertos, consultar apartado 7.3.6) para notificar al componente adecuado la ejecución del servicio. Para poder enviar también los posibles parámetros que se hayan podido generar tras ella (y que aparecen en la especificación del servicio que ha dado

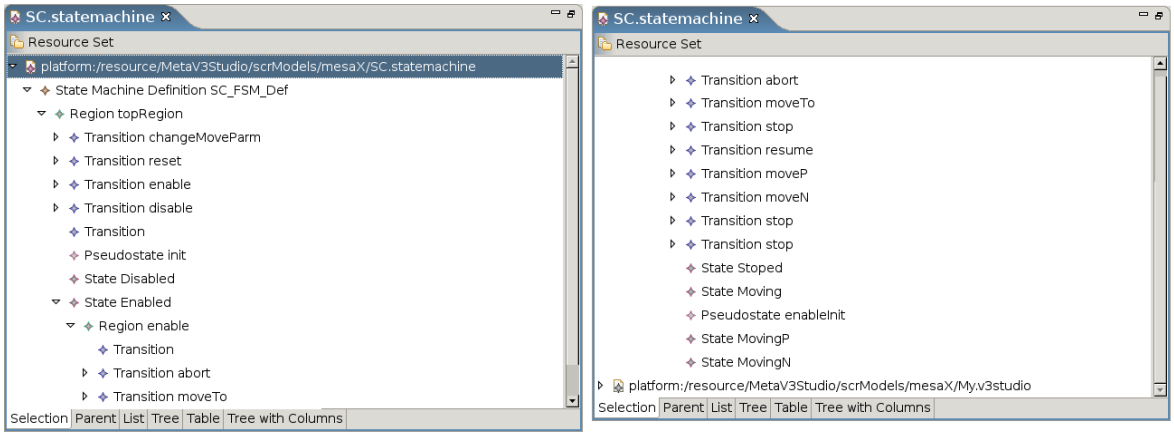


Figura 7.23: Captura de pantalla del entorno Eclipse en que se muestra la estructura de clases V³Studio que describen la máquina de estados del ejemplo de la figura 7.1. Obsérvese como cada uno de los componentes y las interfaces que forman el sistema se han definido por separado.

origen a la señal), en cada `Signal` se generan las propiedades correspondientes para almacenar tanto los valores de entrada como los de salida de la operación. Por tanto, el uso de señales se asemeja a la utilización del patrón de diseño *Command*, ya que convierten llamadas a procedimiento en objetos-eventos que son encolados en el buffer de eventos a procesar por parte de la máquina de estados.

- Se genera un evento, asociado al procesamiento de la señal por parte de la máquina de estados. En las máquinas de estados, concretamente, los disparadores (`Trigger`) de las transiciones están asociados a la ocurrencia de un evento. En el caso de esta transformación, la clase `ReceiveSignalEvent` está asociada a la recepción de una de las señales (`Signal`) generadas por los puertos del componente tras recibir una petición de servicio *asíncrono* (los servicios síncronos están siempre disponibles y no tienen que pasar el filtro de la máquina de estados, consultar apartado 7.3.6). De esta forma es posible relacionar la petición de un servicio asíncrono con el disparo de una transición que provoca el cambio de estado en el componente.
  Los eventos son una de las clases que más controversia genera dentro del propio lenguaje UML, ya que no se especifica en ningún caso quién o cómo se generan, se envían o se procesan. Sin embargo, son clases que aparecen en multitud de diagramas, sobre todo en los relacionados con la especificación del comportamiento de un sistema.

Para concluir, en esta sección se ha descrito la transformación de las máquinas de estados V³Studio en su correspondiente representación hermana en UML. Esta transformación es prácticamente directa, debido al parecido existente entre estas dos partes de los meta-modelos. Así como en el caso de la transformación de la vista arquitectónica realizada en la sección anterior, en este caso no existen posibles diseños alternativos para

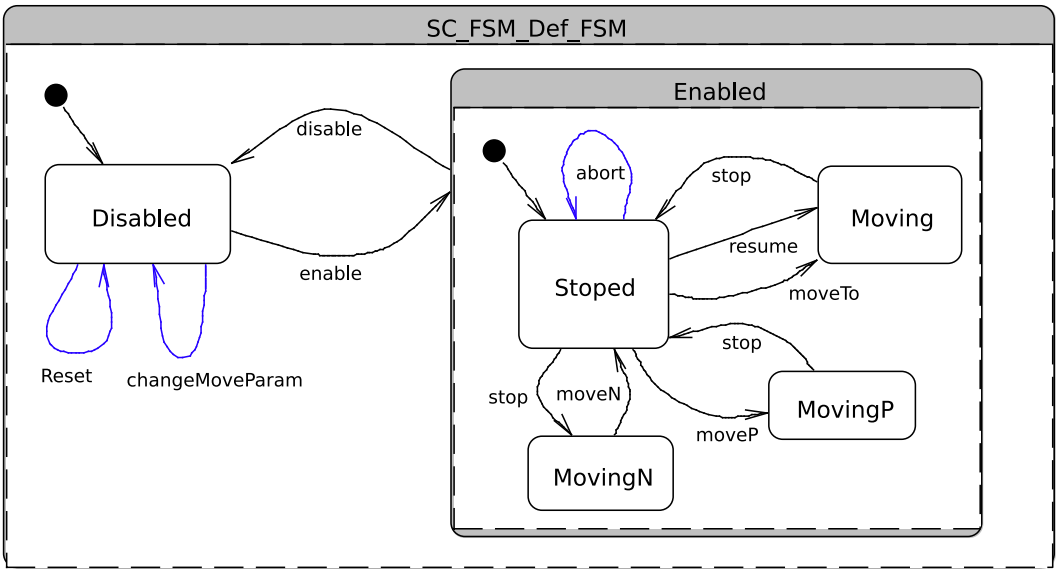


Figura 7.24: Máquina de estados del componente *SC* una vez transformada. Las transiciones de color azul denotan transiciones de tipo interno, mientras que los cuadrados a trazo discontinuo representan regiones.

llevar a cabo la transformación, ya que éstas posibilidades no aparecen en el modelo UML de máquinas de estados. Las posibles alternativas aparecerán en la fase de implementación de la lógica de la máquina de estados en código fuente, como se describe en la sección 8.2.

7.5 V³STUDIO → UML: VISTA ALGORÍTMICA

ESTA última sección describe la parte de la transformación encargada de la vista algorítmica. Como sucede con la vista de comportamiento, esta vista también está basada, aunque en menor medida y con más cambios, en el diagrama homónimo de UML. Y como en el caso del diagrama de máquinas de estados, la transformación entre el modelo V³Studio y el de UML es también bastante directa.

En este punto conviene recordar que los diagramas de actividad de UML 2 contienen los elementos atómicos más próximos a la implementación en código, ya que permiten describir detalles de tan bajo nivel como la secuencia de instrucciones que se ejecuta en una operación. V³Studio no llega a este nivel, ya que para ello sería necesario desarrollar un meta-modelo tan completo y complejo como UML. En vez de popular la vista algorítmica con un (amplio) conjunto de clases que permitieran describir el comportamiento con gran detalle, se tomó la decisión de suministrar un conjunto mínimo de actividades que cubrieran las necesidades básicas del desarrollador: invocación de un servicio requerido, de una función de una librería (dada la profusión de librerías en el dominio de la robótica) y de

redirección de peticiones (en el caso de componentes complejos). Cualquier otra actividad que se saliera de este espectro de actividades básicas se modela como «código a completar por el usuario tras la transformación». Sin embargo, la transformación, aunque es directa, no es sencilla, ya que UML 2 distingue entre actividades y acciones en su meta-modelo (cosa que no sucede en V^{3}Studio), además de que en UML existen más de veinticinco clases de acciones y otras tantas de actividades.

A diferencia de las dos secciones anteriores, esta sección sobre la transformación de la vista algorítmica va a describir, de forma genérica, la transformación de cada uno de los tipos de actividad que se enumeraron a lo largo de la sección 6.6, ya que, en general, son todas bastante sencillas y las transformaciones son independientes unas de otras.

7.5.1 TRANSFORMACIÓN DE LAS ACTIVIDADES SIMPLES

Tal y como se expuso en el apartado 6.6.2, V^{3}Studio modela 5 tipos de actividades simples (atómicas) que constituyen los bloques básicos para especificar la secuencia de instrucciones que describen las operaciones `doActivity`, `entry`, `exit` y `effect` que aparecen, respectivamente, en las clases `State` y `Transition` de V^{3}Studio. Antes de mostrar los diagramas de actividad tipo en que se transforman cada una de las cinco actividades simples de V^{3}Studio, hay que resaltar que todos los diagramas UML de actividad que se generan están asociados a la máquina de estados que describe el comportamiento de la clase que implementa el componente (`XXX_Real`), y que, por tanto, cuando se utiliza la acción `ReadSelfAction` se está obteniendo una referencia a un objeto de dicha clase. Esta aclaración es importante para comprender las transformaciones que se describen a continuación:

ServiceCall: modela la invocación de uno de los servicios requeridos por el componente por uno de sus puertos. Por tanto, una actividad de este tipo tiene sendas referencias al servicio que debe invocar y al puerto en que se encuentra. En la figura 7.25 puede observarse el diagrama de actividad resultante de la transformación de esta actividad. En este caso, puede observarse que se hace uso de la acción `ReadStructuralFeatureAction` para obtener la referencia al puerto por el que se tiene que realizar la petición de servicio y de la acción `CallOperationAction` para invocar dicho servicio sobre el puerto adecuado. En la figura se muestra la transformación de un `ServiceCall` que invocan un servicio síncrono y otro asíncrono, respectivamente.

LibraryCall: esta actividad modela la invocación de una función o procedimiento definido en una librería o *API*, ya que es común que en dominios tan asentados como la robótica

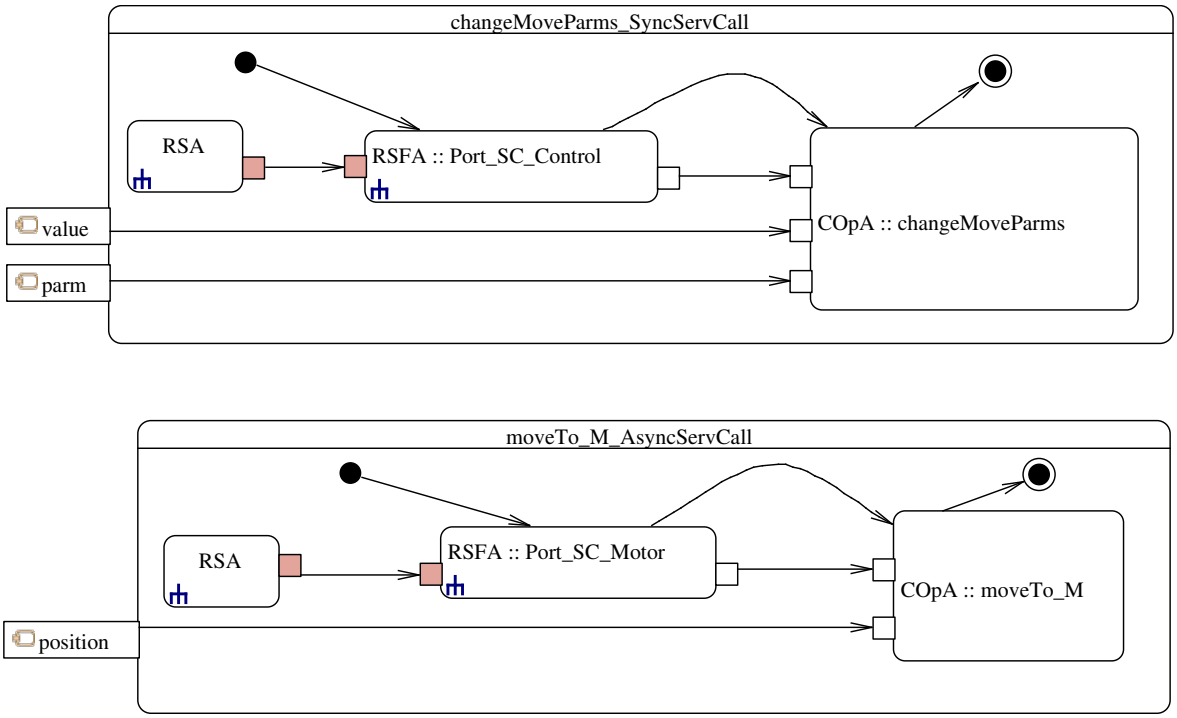


Figura 7.25: Transformación de la actividad simple `ServiceCall`. La figura superior muestra la invocación del servicio síncrono *changeMoveParms* (definido en la interfaz *I_SC_Control* del ejemplo, consultar tabla 7.1 en la página 167), mientras que la figura inferior se muestra el servicio asíncrono *moveTo_M* (definido en la interfaz *I_Motor_Control*), junto con los respectivos parámetros. En este caso puede observarse que las transformaciones generan diagramas parecidos, ya que la forma en que se invoca un servicio es independiente de su tipo.

existan librerías comerciales o de libre distribución que provean las funciones más utilizadas en el dominio. En este caso se genera siempre un diagrama de actividad que contiene una `CallOperationAction` en su interior, que representa la función de la librería que se quiere invocar. Los parámetros de la actividad `LibraryCall` se convierten en parámetros (`ActivityParameterNode`) del diagrama UML, mientras que los `Pin` se transforman en la correspondiente clase UML (`InputPin` u `OutputPin`) y se añaden al `CallOperationAction` que representa la función de la librería. Por último, cada `PinParamLink` se transforma en un `ControlFlow` que relaciona los datos de entrada y salida de la actividad/wrapper con los de la función de la librería.

ForwardService: esta actividad modela el reenvío, por parte de la máquina de estados que define el comportamiento de un componente complejo, de una petición de servicio al puerto correspondiente del componente interno que va a procesarla finalmente. En la figura 7.26 puede observarse que el componente accede a la referencia `conjugate` del puerto (que en el caso de puertos contenidos en componentes complejos apunta

al puerto interno en que se delega la ejecución del servicio) para invocar directamente la operación `receiveSignal`, que reenvía la petición de servicio. Esta petición de servicio ya se convirtió en señal en el momento en que se encoló por primera vez en el componente complejo, por lo que la invocación de la operación se realiza de forma directa.

UserDefinedActivity: esta actividad modela, de forma general, un bloque de código que va a ser rellenado por el usuario y que no se corresponde con ninguna de las actividades definidas en los puntos anteriores. En este caso se ha optado por utilizar la clase UML `OpaqueAction`, que modela un tipo de acciones en las que el usuario introduce directamente el código fuente. Por tanto, una actividad simple de tipo `UserDefinedActivity` se transforma directamente en una acción `OpaqueAction`. Como en los casos anteriores, también se transforman los posibles parámetros que se hayan podido definir.

ConstantActivity: una actividad constante modela una actividad que únicamente produce un valor de salida preestablecido por el desarrollador. En este caso, como sucede con la actividad anterior, la traducción es directa, ya que UML define la clase `ValueSpecificationAction`, que genera un valor constante del tipo dado.

7.5.2 TRANSFORMACIÓN DE LAS ACTIVIDADES COMPLEJAS

Una actividad compleja (`ComplexActivity`) permite establecer establecer flujos de ejecución secuencial de cualquier otro tipo de actividades (simples o complejas) y reutilizar actividades previamente modeladas. Las actividades complejas permiten describir el comportamiento de las actividades de los estados y transiciones de la máquina de estados que describe el comportamiento de un componente, partiendo de otras actividades más simples. Además de la clase general `ComplexActivity`, V^{3}Studio modela la existencia de dos tipos especiales de actividades complejas: los bucles (`LoopActivity`) y las actividades condicionales (`ConditionalActivity`). Las clases V^{3}Studio que permiten modelar estas actividades complejas se transforman de la siguiente forma:

- Las actividades complejas se transforman en una `Activity` UML que contiene en su interior todas las clases que se producen como resultado de transformar todas las actividades simples contenidas en la actividad compleja V^{3}Studio, tal y como se indicó en la sección 7.5.1.
- Los `ActivityParameter` de V^{3}Studio se convierten en pines de entrada (`InputPin`) o de salida (`OutputPin`), dependiendo del nombre de la asociación en que se encuentren contenidos (consultar figura 6.10 en la página 152).

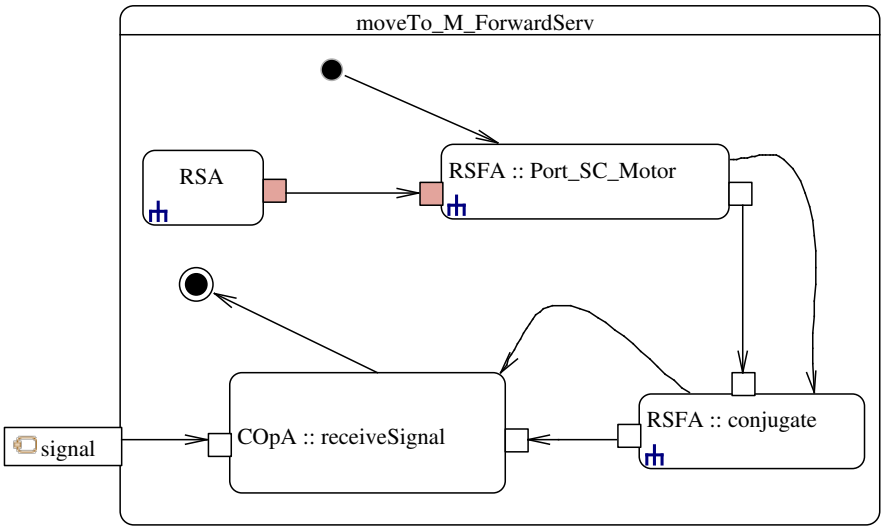


Figura 7.26: Transformación de la actividad simple `ForwardService`

- Las clases `ActivityEdge` que controlan el flujo de datos y de control entre las distintas actividades en el modelo V³Studio se convierten, respectivamente, en las clases UML homólogas `DataFlow` y `ControlFlow`.
- Las clases de tipo `PseudoActivity` se transforman también en sus homólogas UML.
- Las actividades complejas «especiales» `LoopActivity` y `ConditionalActivity` se transforman en las clases UML `LoopNode` y `ConditionalNode`, respectivamente. La transformación es de nuevo directa en este caso, ya que V³Studio adoptó también la definición de este tipo especial de actividades complejas de la superestructura de UML.

7.6 RESUMEN DE LA TRANSFORMACIÓN

EN ESTE capítulo se ha descrito una transformación de modelos, entre V³Studio y UML, que cumple con dos objetivos principales: (1) reducir la distancia semántica existente entre los conceptos presentes en V³Studio (principalmente los conceptos de CBD componente, conector y puerto) y los conceptos presentes en los lenguajes de programación orientados a objetos (clases, herencia, composición, polimorfismo, etc), y (2) describir en un lenguaje orientado a objetos e independiente de la plataforma como es UML la estructura software que representa los conceptos CBD, de forma que sea fácilmente traducible a otros lenguajes de implementación. Además, UML es un lenguaje muy difundido y utilizado por la comunidad de desarrollo software, lo cuál ha propiciado el desarrollo de numerosas herramientas para analizar los diagramas (por ejemplo, cálculo de

métricas a partir del diagrama de clases) y simular el comportamiento (por ejemplo, análisis del comportamiento temporal). Gracias a la transformación a un modelo UML se podrá hacer uso de estas herramientas tanto para mejorar el diseño de la transformación como para estudiar el comportamiento y las características de un modelo V^{3}Studio en particular.

A modo de resumen final, a continuación se presenta un recuento de las clases UML que se han generado tras aplicar la transformación al modelo V^{3}Studio de ejemplo (consultar apartado 7.1.1). Esta estadística es aproximada, sobre todo porque el número de clases que se generan en los diagramas de actividades es muy variable y porque hay algunas clases secundarias que no se han contado. Esta sección trata de aportar al lector una idea de la magnitud del modelo UML de salida.

- Siete paquetes (clase `Package` de UML) para almacenar las clases básicas de apoyo del framework (cuatro paquetes) y los componentes (tres paquetes, uno por componente), tal y como se expuso en el apartado 7.2.4.

- Cuatro interfaces (clase UML `Interface`), una por cada interfaz presente en el ejemplo, contenidas en el paquete `Interfaces_Package`.

- Veinte clases (clase UML `Class`), repartidas prácticamente entre todos los paquetes. Concretamente, hay cuatro clases en el paquete `Ports_Package` (definen los tipos comunes, consultar apartado 7.3.3), dos clases corresponden a las clases base del framework, y el resto se encuentran en los componentes: cuatro clases para *HAL_Component* y para *XYZ_Robot* (dos para implementar el componente según lo expuesto en el apartado 7.3.2 y una por cada uno de los dos puertos que tienen cada uno) y seis para *SC_Component* (dos para implementar el componente y una por cada uno de los cuatro puertos que contiene el componente).

- Ciento veintinueve operaciones (clase UML `Operation`) repartidas también entre casi todos los paquetes. Estas operaciones engloban tanto a las que están definidas en las clases base del framework como las que se generan por cada servicio presente en la aplicación.

- Veintiuna relaciones de asociación (clase UML `Association`), repartidas entre los tres paquetes que definen los tres componentes presentes en el sistema. Estas relaciones de asociación agrupan también a las relaciones de contención.

- Veintinueve atributos (clase UML `Property`), contenidos por las clases que implementan la transformación de los componentes y los puertos presentes en el modelo V^{3}Studio, que están a su vez contenidas en los paquetes que contienen la transformación de cada uno de los componentes.

- Cuatro funciones (clase UML `OpaqueBehavior`) sin implementación para que el usuario introduzca el código que finalmente tiene que ejecutar el componente que ofrece un servicio síncrono en alguno de sus puertos. Estas funciones están contenidas en las correspondientes clases `XXX_Real` que describen el componente.

- Setenta y cinco actividades (clase UML `Activity`), repartidas entre las clases que forman la implementación de los componentes y los puertos. Estas actividades implementan parte de la semántica asociada a la transformación utilizando el patrón *Active Object* (consultar el apartado 7.2.1) así como la definición de componente adoptada para llevarla a cabo. Cada una de estas actividades representa, como aclara el punto siguiente, un diagrama de actividad de los que se han presentado a lo largo de la transformación.

- Una media de dieciséis clases, de las comentadas en el anexo B, para describir la implementación de cada una de las actividades comentadas en el punto anterior.

- Ocho eventos (clase UML `ReceiveSignalEvent`) y once señales (clase UML `Signal`), contenidas en el paquete `Signals_Package`. Como se menciona en la sección 7.4, estas clases son utilizadas para controlar el funcionamiento de la máquina de estados y para notificar al cliente que el servicio de tipo asíncrono ha sido completado.

- Cuarenta clases para describir la máquina de estados asociada al componente *SC*.

- Otras tantas clases, necesarias para representar (1) los tipos de datos definidos en la aplicación, (2) los parámetros de las operaciones, (3) la implementación de interfaces, (4) la herencia de una clase, (5) las propiedades que definen los extremos de las relaciones de asociación (con y sin contención), etc.

Todas las clases enumeradas anteriormente suman un total de unas 1.550 clases UML, generadas a partir de un conjunto de modelos V^{3}Studio que contienen 91 clases y que están divididos en 5 ficheros (modelos) reutilizables por separado (siempre que se reutilicen también sus dependencias).

CAPÍTULO 8

APLICACIÓN DE V^{3}STUDIO AL CONTROL DE UNA MESA XYZ

ESTE último capítulo de la parte en que se describe el desarrollo de V^{3}Studio aborda una posible traducción del modelo UML generado tras la ejecutar la transformación de de modelos descrita en el capítulo anterior. Para desarrrollar la transformación a código y acabar de demostrar la viabilidad del enfoque de desarrollo basado en modelos para sistemas basados en componentes propuesto en esta Tesis Doctoral, se ha elegido el lenguaje de programación Ada en su revisión del año 2005 [1]. La razón que justifica esta elección es sencilla: Ada es el único lenguaje especialmente diseñado para el desarrollo de sistemas con restricciones de tiempo real. En este punto se contempló también la posibilidad de utilizar Real-Time Java [2], pero fue desechada debido a que no es tan madura como Ada.

Una de las características fundamentales de todos los sistemas que interaccionan con su entorno físico, entre los que se cuentan los sistemas robóticos, es el carácter eminentemente concurrente y de tiempo-real de la aplicación. y la necesidad de adoptar los mecanismos de protección frente a los clásicos problemas de concurrencia (exclusión mutua, región crítica, variables compartidas, etc). En este campo, el lenguaje Ada sigue sin tener rival entre los lenguajes de programación actuales y es uno de los más seguros y fiables.

8.1 GENERACIÓN DE LA ESTRUCTURA DEL CÓDIGO

EN ESTA sección se describe la generación de la estructura del código Ada a partir del modelo UML que se obtiene tras la transformación del modelo V³Studio mostrado en la sección 7.1.1. Esta estructura de paquetes Ada es similar a la que se describió en el apartado 7.2.4 (consultar página 177), aunque con algunas ligeras modificaciones para ajustarse a las características particulares del lenguaje elegido. Antes de comenzar la descripción de la estructura de paquetes se van a reproducir la definición de los principales tipos de «paquetes» del lenguaje Ada 2005 [1]:

Package: un paquete es una unidad de programa que permite definir un grupo de entidades lógicamente relacionadas entre sí. Normalmente, un paquete contiene la declaración de un tipo (generalmente un tipo privado o una extensión privada) junto con la declaración de los subprogramas primitivos del mismo. Los paquetes proporcionan el mecanismo básico de encapsulación del lenguaje.

Child Package: un paquete hijo es un paquete que conceptualmente forma parte de su paquete padre, pero que está definido en un fichero distinto. De esta forma se facilita el diseño incremental del software, ya que los paquetes hijo se pueden recompilar por separado sin romper las dependencias del paquete padre. Los paquetes hijos proporcionan otro mecanismo adicional para descomponer un sistema complejo en partes más sencillas.

Private Child Package: un paquete hijo privado es similar a un paquete hijo normal, salvo que sólo puede utilizarse (sólo son visibles) en el contexto definido por otros paquetes incluidos en la jerarquía del paquete padre.

Haciendo uso de las características especiales de cada uno de los tipos de paquetes que define Ada 2005 se ha realizado una traducción del diagrama de paquetes y componentes anteriormente mencionado a una estructura de paquetes Ada. Esta estructura de paquetes mantiene, como se muestra a continuación, la encapsulación del contenido de los componentes originalmente definidos con V³Studio. La transformación genera un esqueleto de código Ada en el que se utilizan algunas de las nuevas mejoras que se han añadido al lenguaje en su última revisión, como son los tipos interfaz (`interface`), la especificación de relaciones cíclicas entre tipos y operaciones (`[private] limited with`), especificación de operaciones de sobrescritura de los subprogramas heredados (`overriding`), las nuevas características de orientación a objetos y sentencias `return` extendidas, entre otras.

Esta sección está organizada en tres apartados, en los que se describen la generación de los paquetes que contienen los tipos de datos y operaciones que se corresponden con la transformación de la vista arquitectónica de V^{3}Studio. Concretamente, los siguientes dos apartados describen la generación de los paquetes y las clases Ada que forman parte de esta infraestructura auxiliar, mientras que el último apartado describe la generación del código asociado a la transformación de los componentes.

8.1.1 Generación de la infraestructura de apoyo

En este apartado se describe la parte de la transformación modelo-a-texto que genera lo que se ha denominado «infraestructura de apoyo» de la aplicación (consultar apartado 7.2.4). Esta infraestructura está formada por aquellos paquetes y clases que definen las partes básicas del framework y los tipos de datos auxiliares que serán utilizados posteriormente en la generación del código de los componentes, tal y como se describe en el apartado 8.1.3. Concretamente, en este apartado se describen, y por este orden, la generación de las clases básicas del framework (`Port` y `Component`), las clases relacionadas con los tipos de puertos abstractos y comunes para toda la aplicación, la generación de los tipos de datos y, por último, la generación del subprograma principal de la aplicación, por el que comienza la ejecución de la misma.

Clases básicas: las clases básicas del framework (`Port` y `Component`) se generan en un mismo paquete Ada, ya que ambas clases son abstractas y no proporcionan, de momento, ninguna operación concreta. La clase `Port` define una serie de operaciones abstractas comunes (consultar apartados 7.3.5 y 7.3.6), mientras que la clase `Component` se genera actualmente sin ninguna operación primitiva. Por el momento, se deja la especificación de las operaciones básicas comunes del framework a los usuarios de V^{3}Studio, ya que cada aplicación puede requerir operaciones específicas que no pueden ser previstas con antelación.

Definición de los puertos: la definición de las interfaces de la aplicación y de los tipos de puerto abstractos comunes se realiza en los paquetes `Interfaces_Package` y `Ports_Package`, respectivamente. En ambos casos, los paquetes contienen únicamente la especificación de los tipos de datos junto con las operaciones primitivas, ya que las interfaces no tienen implementación y los tipos que definen los puertos comunes sólo se utilizan para llevar a cabo las operaciones de petición de servicios y enlazado de puertos (consultar el apartado 7.3.3) y, además, son clases abstractas sin subprogramas concretos.

Definición de los tipos de datos. Ada es un lenguaje fuertemente tipado y requiere que todos los tipos de datos (salvo los tipos primitivos) se declaren previamente. En este caso, se define el paquete `Datatypes_Package` como un paquete base a partir del cuál el usuario debe generar, posteriormente, los paquetes hijos que definen los tipos de datos, según estime necesario. Estos tipos de datos se corresponden con la definición de los tipos de datos que se utilizan en los *servicios que se invocan entre componentes* únicamente, ya que cualquier tipo de dato que sea de uso interno del componente se puede añadir posteriormente por el usuario al código que se genera para el componente.

Subprograma principal. Es el punto por el que comienza la ejecución de la aplicación definida en base a los modelos creados con V^{3}Studio que, como exige el lenguaje, es un procedimiento. En este procedimiento tan sólo

1. Se crea una instancia del componentes complejo de mayor nivel de la aplicación (consultar apartado 6.4.2), ya que el constructor de dicho componente se encarga de instanciar, jerárquicamente, el resto de componentes del sistema (consultar apartado 7.3.4).
2. Se invoca la operación `portLink` (consultar apartado 7.3.5) para que el componente de mayor nivel proceda a enlazar los puertos compatibles de los componentes que contiene e invoque, de nuevo jerárquicamente, esta operación en todos los componentes complejos que contenga

Una vez finalizada esta segunda operación la aplicación está lista para comenzar su ejecución, que a partir de este momento evoluciona de forma autónoma, dependiendo del comportamiento de los propios componentes.

8.1.2 TRANSFORMACIÓN DE LAS SEÑALES Y EVENTOS UML

Las señales, modeladas por la clase UML `Signal`, representan la señalización de la ocurrencia de algún evento en la aplicación. Estas clases son utilizadas en el contexto de la transformación de la vista de comportamiento (máquinas de estado, consultar sección 7.4) como las clases que generan los eventos (representados por la clase UML `ReceiveSignalEvent`), que a su vez provocan la activación y reacción de la máquina de estados del componente frente a la invocación de alguno de los servicios ofrecidos por el mismo.

Cada vez que un componente recibe una petición de un servicio *asíncrono* (los síncronos están siempre disponibles y no tienen que pasar el filtro de la máquina de estados, consultar apartado 7.3.6) por el puerto correspondiente se crea un objeto del tipo «señal»,

que simboliza la señalización de la recepción de la petición de servicio. En el modelo UML, esta señal provoca la creación de un evento `ReceiveSignalEvent`, que es encolado en el buffer de eventos de la máquina de estados, para ser posteriormente procesado por el componente en el momento en que tenga asignada la CPU. Estos `ReceiveSignalEvent` se asocian durante la transformación V^{3}Studio→UML (consultar sección 7.4) al disparador de la transición correspondiente, y de este modo se puede relacionar la invocación de un servicio asíncrono con el cambio de estado del componente.

Sin embargo, en la transformación a código no se generan tantas clases como señales y eventos aparecen en el modelo UML, sino que se aprovecha el hecho de que la clase `Signal` hereda de la clase `Classifier` para almacenar todos los parámetros de la petición de servicio en dicha señal. Por otro lado, los eventos y su relación con el disparo de las transiciones quedan embebidos en el código que se genera para traducir la lógica de la máquina de estados (consultar sección 8.2). En este contexto, el patrón de diseño *Command* aparece como la solución de diseño ideal para (1) encolar las peticiones de servicio asíncrono en el buffer de eventos (que tras la transformación pasaría a denominarse «de señales», aunque se mantiene la nomenclatura UML) de la máquina de estados, y (2) llevar a cabo el intercambio de señales entre puertos, según las operaciones descritas en el apartado 7.3.6. Tras realizar estas consideraciones, se pasa a describir los elementos que se generan en esta parte de la transformación a código Ada:

- Se crea un paquete, denominado `Signals_Package`, que contiene la definición de todas las clases en que se transforman las señales, así como la clase padre abstracta de la que heredan todas ellas. Esta clase padre, denominada `Root_Signal`, tiene únicamente un atributo de instancia en el que se almacena el puerto por el que llegó la petición de servicio, de forma que sea posible retornar cualquier resultado de la ejecución del servicio al componente que originalmente solicitó su ejecución.

- Por cada una de las señales (clase UML `Signal`) presentes en el modelo UML se crea una clase que hereda de la clase padre abstracta `Root_Signal` y que contiene tantas variables de instancia como parámetros (tanto de entrada como de salida) tenga el servicio cuya ejecución se está solicitando. Los objetos de estas clases son creados y encolados en el buffer de eventos de la máquina de estados por el puerto que recibe la petición de servicio, correspondiéndose con el uso del patrón de diseño *Command*. Todas estas clases se generan en el mismo paquete Ada ya que, en este momento, las señales sólo se utilizan para almacenar los valores de entrada y salida de las operaciones. Aunque los tipos de variable *record* de Ada son un mecanismo «más ligero» que el uso de objetos para almacenar de forma cohesiva un conjunto de datos relacionados, se decidió utilizar objetos para facilitar posibles ampliaciones de esta parte de la transformación.

8.1.3 TRANSFORMACIÓN DE COMPONENTES Y PUERTOS

La traducción de los componentes V³Studio a un conjunto de clases UML siguiendo el patrón de diseño *Active Object* [190] (consultar apartado 7.2.1) se expuso en el apartado 7.3.2. Según se indicó en dicho apartado, un componente V³Studio se convierte en dos clases: (1) una clase fachada que define únicamente las operaciones para obtener las referencias a los puertos del componente y no tiene comportamiento (salvo el que herede de la clase base del framework `Component`), y (2) una clase que contiene los objetos que definen los puertos reales del componente (consultar apartado 7.3.3), así como la máquina de estados que define el comportamiento de el mismo. Así se consigue una estructura flexible que cumple con el requisito de encapsulación del contenido del componente. Esta estructura se plasma en el código Ada generado siguiendo el siguiente esquema:

- Se crea un paquete padre, denominado de forma genérica `XXX_Component` (donde `XXX` es el nombre del componente V³Studio) que establece la base de la jerarquía a partir de la cual se crean el resto de paquetes hijo que se describen a continuación. En este paquete padre se define la fachada de la implementación del componente, que hereda a su vez de la clase base abstracta `Component` (consultar apartado 8.1.1). En este paquete se definen, además, (1) las operaciones para obtener las referencias a los puertos del componente (obtenidas como valores del tipo de puerto común y abstracto correspondiente, consultar de nuevo el apartado 8.1.1), y (2) una función constructor, según describe el apartado 7.3.4, para crear e inicializar un componente de este tipo.

- Se crea un paquete hijo privado en el que se define la clase que implementa la funcionalidad real del componente y que contiene los objetos que definen los puertos del mismo. Esta clase define e implementa (1) todas las operaciones básicas para llevar a cabo la comunicación entre puertos (definidas en el apartado 7.3.6), (2) todos los operaciones asociadas a los servicios marcados como *síncronos* en el modelo V³Studio (cuyo código tiene que suministrar posteriormente el usuario), (3) una función constructor en la que crea e inicializa la clase y los objetos que representan los puertos del componente (ver punto siguiente). La utilización de un paquete hijo privado asegura que esta clase es únicamente visible y accesible por la clase que está definida en el paquete padre, y de esta forma se mantiene la encapsulación del contenido del componente.
- Se crea un paquete hijo privado por cada uno de los tipos de puertos que contiene el componente V³Studio. Estos paquetes contienen la definición de las clases de implementación de los puertos junto con todas operaciones asociadas a las mismas. En este caso se generan estructuras ligeramente distintas, ya que un puerto desempeña

labores ligeramente distintas dependiendo del tipo de definición de componente que lo contenga (consultar apartados 7.3.3.1 y 7.3.3.2).
En el caso de un puerto contenido en un `SimpleComponentDefinition`, las operaciones que se generan en los puertos son: (1) una función para crear e inicializar el objeto que representa el puerto y sus atributos, (2) las operaciones básicas de comunicación y enlazado de puertos (consultar apartados 7.3.6 y 7.3.5) y (3) tantas operaciones como servicios ofrece y requiere el puerto en cuestión.
En las operaciones generadas a partir de los servicios requeridos se invoca el servicio correspondiente en el puerto conjugado, mientras que en los servicios ofrecidos se distinguen dos casos: (1) si el servicio es síncrono, el puerto que recibe la petición invoca directamente la operación correspondiente en la clase que implementa el componente real, mientras que (2) si el servicio es asíncrono, el puerto que recibe la petición instancia la correspondiente clase señal (descrita en el apartado 8.1.2) y la encola en la máquina de estado del componente.

En el caso de un componente complejo (EClass `ComplexComponentDefinition` en V^{3}Studio), la transformación se amplía ligeramente para incorporar las características propias de un componente complejo (consultar apartado 7.3.3). En este caso, los cambios son los siguientes:

- En el paquete que contiene la clase fachada se añade la operación `portLink`, que indica al componente complejo que debe proceder a enlazar los puertos de sus componentes internos. La generación del diagrama de actividades de esta operación se detalla en el apartado 7.3.5.4.
- Como ya se ha mencionado, la estructura interna de un puerto contenido en un componente complejo es ligeramente distinta a la de un puerto contenido en uno simple. En este caso, la variable de instancia `conjugate` tiene un tipo distinto, los cuerpos de los servicios síncronos redirigen la petición al puerto correspondiente del componente interno que implementa dicha funcionalidad (en vez de a la clase que implementa el comportamiento de el componente) y se crea una operación nueva, denominada `setDelegationPort`, para inicializar la referencia conjugate al valor adecuado. Todas estas modificaciones aparecen detalladas en el apartado 7.3.3.2.
- En el paquete hijo privado que contiene la implementación del comportamiento del componente se añaden tantas variables de instancia como componentes contiene el componente complejo. El constructor de la clase se modifica también para invocar los respectivos constructores de los nuevos atributos que se han añadido a la clase.

En la transformación de los componentes se ha hecho uso intensivo de las nuevas características de orientación a objetos y otras mejoras que se han añadido al lenguaje Ada

en la revisión del 2.005. Concretamente, han resultado especialmente valiosas y útiles la inclusión de las sentencias *limited with* y *limited private with* para romper las relaciones cíclicas entre los tipos de datos, ya que tanto la clase que desempeña labores de fachada del componente, como la que implementa su funcionalidad como la de los puertos están fuertemente acopladas (como era de esperar, por otro lado). Estas nueva sentencias han permitido partir y definir por separado cada una de las clases descritas anteriormente, lo cual redunda en la obtención de un diseño más claro, ampliable y modular.

Para terminar esta sección, en la que se ha descrito la generación de código Ada a partir del modelo UML obtenido tras la ejecución de la transformación V^{3}Studio → UML, se van a presentar unas estadísticas del código generado. A modo de resumen, para el ejemplo del componente complejo mostrado en la sección 7.1.1 se han generado (1) un total de 46 ficheros, en los que se definen 25 paquetes y el procedimiento principal con el que comienza la ejecución de la aplicación; (2) la estructura de los componentes y la implementación de las operaciones expuestas en los apartados 7.3.4, 7.3.5 y 7.3.6 generan unas 950 SLOC (*Source Lines of Code*).

8.2 GENERACIÓN DE LAS MÁQUINAS DE ESTADOS

EN ESTA sección se describen algunas de las posibles alternativas para la implementación de máquinas de estados y parte del trabajo que se está desarrollando actualmente. En la transformación de la máquina de estados, que se corresponden con la vista de comportamiento, tienen una gran influencia las características de concurrencia del componente. La concurrencia es un característica intrínseca de las aplicaciones que tienen que controlar o desenvolverse en un entorno físico. Además, en cierto modo, los componentes tienen también una naturaleza concurrente, no necesariamente en el sentido de ejecución, sino en el sentido de que un componente es independiente del resto de componentes que forman un sistema, y en cierto modo se puede suponer que todos ellos se ejecutan de forma «concurrente». La naturaleza concurrente de los componentes está presente en todas las etapas del desarrollo de una aplicación basada en componentes con V^{3}Studio:

1. En la creación de los modelos V^{3}Studio, en los que se puede establecer la política de concurrencia del componente-instancia (consultar apartado 6.4.1).

2. En el marcado de la clase que define un componente como activa o no (consultar apartado 7.3.2) y en la utilización del patrón *Active Object* como medio para describir y traducir los componentes (consultar apartado 7.2.1).

3. En las decisiones de diseño que se describen a continuación y que atañen a la generación de código Ada.

La concurrencia en los componentes aparece, fundamentalmente, en su máquina de estados. La máquina de estados recibe todas las peticiones de servicio que el componente recibe a través de sus puertos, las almacena en el buffer de eventos como se indicó en el apartado 8.1.2, y las va sirviendo, una a una según indica el patrón *Active Object*. La primera medida, por tanto, consiste en proteger este buffer de eventos, de forma que sea posible acceder al mismo de forma concurrente. El lenguaje Ada proporciona los *tipos protegidos* como un medio para definir un tipo de datos cuyo acceso se realiza en régimen de exclusión mútua, por la que la generación del buffer de eventos es directa una vez que se utiliza este mecanismo del lenguaje.

Tras proteger la máquina de estados frente al acceso concurrente de peticiones de servicio *asíncrono* por parte del resto de compononentes del sistema, tan sólo resta por describir la forma en que se ha realizado la transformación a código. Como ya se mencionó en la sección 7.4, el modelo UML origen de la transformación a código Ada que se está describiendo a lo largo del presente capítulo respeta la notación y los conceptos de las máquinas de estado. De esta forma se posibilita la generación de distintas implementaciones de la lógica de la máquina de estados, por ejemplo, según se indica en el patrón *State* [96]:

- Siguiendo el propio patrón *State*, Gamma identifica dos posibles diseños, dependiendo de si la lógica de el cambio de estado está centralizada en una única clase o dispersa entre todas los objetos que forman la máquina de estados.
- Creación de una tabla estados-transiciones que indique el estado que se puede alcanzar a partir de la activación de una transición.
- Centralizar la lógica del cambio de estado en grandes bucles de selección, de forma que los estados contienen únicamente el código de las actividades que tienen que ejecutar.

Actualmente se está trabajando en el desarrollo y valoración de estas posibles traducciones a código. Concretamente, se ha desarrollado un Proyecto Fin de Carrera en el que se comparan tres posibles implementaciónes de la lógica de la máquina de estados. Asimismo se han desarrrollado dos herramientas [10, 213] con las que trabajar con máquinas de estados aislada e independientemente de V^{3}Studio. El desarrollo de estas herramientas nos permitirá analizar y comprobar el funcionamiento de distintas implementaciones de las máquinas de estados, ya que, al controlar y coordinar el comportamiento del componente, son una de las partes críticas de la aplicación.

En cierto modo, la máquina de estados son artefactos propios e internos de un componente que no afectan ni a la estructura ni al funcionamiento global de la aplicación

como tal, sino más bien al propio componente. Además, y a diferencia de lo que sucede con los componentes, la transformación de la máquina de estados sólo afecta de forma local al componente, por lo que se decidió respetar la notación para experimentar con distintas implementaciones. De hecho, sería incluso posible que la máquina de estados de distintos componentes estuviera implementada de manera distinta.

8.3 APLICACIÓN AL CONTROL DE UNA MESA XYZ

PARA concluir este capítulo se va a describir el caso de estudio en el que se ha aplicado V^{3}Studio y con el que se pretende demostrar la viabilidad, capacidad de modelado y potencial de esta herramienta y del enfoque que subyace bajo su desarrollo. El caso de estudio que se ha elegido utiliza uno de los robots desarrollados en el contexto del proyecto EFTCoR, concretamente en el robot cartesiano que soporta el cabezal de limpieza.

El objetivo del proyecto EFTCoR[1] [180] es el desarrollo de un sistema de limpieza automatizada de la superficie de un buque y del sistema de recogida y reciclaje de residuos. Este sistema tiene que alcanzar, al menos, la misma calidad de acabado superficial que se consigue con las técnicas actuales pero reduciendo la emisión de productos tóxicos y contaminantes al entorno.

Esta sección está organizada en dos apartados, en los que se describe someramente la problemática que aborda el proyecto EFTCoR junto con la solución propuesta por el grupo DSIE, y la arquitectura de control y el desarrollo del software llevado a cabo mediante V^{3}Studio, respectivamente.

8.3.1 EL PROYECTO EFTCOR. DESCRIPCIÓN DE LA MESA XYZ

Debido a las características especiales del entorno de trabajo, el proyecto EFTCoR requirió el desarrollo de distintos sistemas robóticos. Estos sistemas se desarrollaron para adaptarse a las dos operaciones de limpieza que son requeridas por los astilleros (limpieza completa de un área de cierto tamaño del casco del barco y limpieza de pequeñas áreas dispersas) y a las distintas formas que adopta el casco de un barco (ver figura 8.1). Para hacer frente a esta variabilidad, se desarrollaron dos sistemas robóticos:

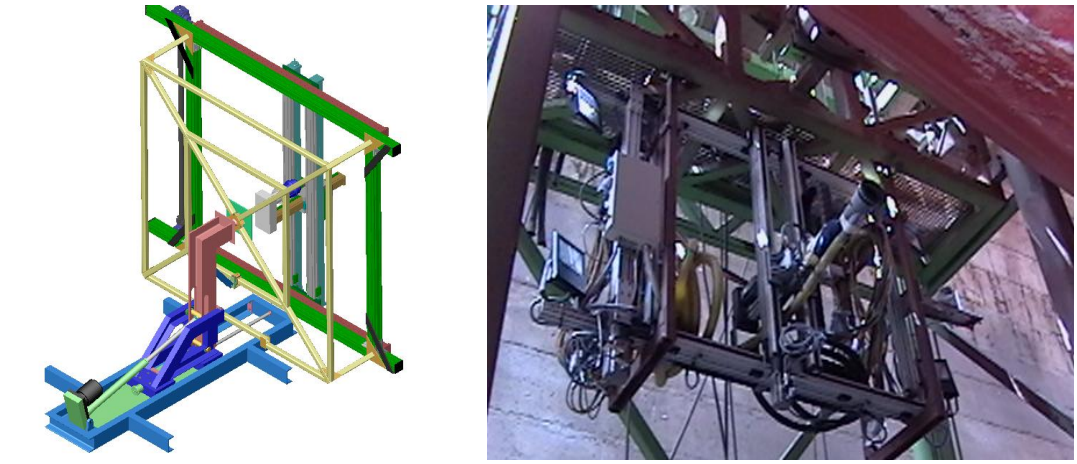
[1] *Environmental friendly and cost-effective technology for coating removal*, Proyecto GROWTH del V Programa Marco de la Unión Europea G3RD-CT-2002-00794 y CICYT DPI2002-11583-E, 2002-2005

Figura 8.1: Distintas formas del casco de un barco: proa (izquierda), banda (central) y bajos (derecha)

- Un robot trepador pequeño, capaz de alcanzar las zonas más complicadas del casco del barco, para llevar a cabo las tareas de limpieza localizada.
- Un robot cartesiano capaz de limpiar grandes áreas del casco de un barco y que podía ser acoplado a distintos sistemas de posicionamiento primario. Por ejemplo, una grúa o un vehículo tipo carretilla elevadora.

Esta sección se centra en el segundo de estos robots, el robot cartesiano XZY que muestra la figura 8.2. En respuesta a los requisitos industriales especiales del proyecto EFTCoR, se decidió utilizar componentes industriales de gran fiablidad de la compañía Siemens para desarrollarlo. Concretamente, se decidió implementar la unidad de control en un PLC (*Programmable Logic Controler*) Simatic S7-300 y utilizar un bus de campo Profibus-DP para comunicar la unidad de control con el resto de sistemas que integran el robot. Todos los componentes utilizados en el robot son componentes industriales estándar de amplia disponibilidad, robustos y fácilmente mantenibles que facilitan la integrabilidad, interoperabilidad y mantenibilidad de todo el sistema. La figura 8.3 muestra, de forma esquemática, la estructura de control del robot cartesiano, en la que:

1. Unidad de programación PG, un PC conectado vía Profibus con el PLC.
2. Bus de campo Profibus-DP. Bus de campo especialmente adaptado a la comunicación entre sistemas de automatización y periferia descentralizada.
3. Siemens Mobile Panel 170, panel de operador industrial.
4. PLC Simatic S7-314C-2DP. El programa de control se ejecuta en esta CPU. Integra también un interfaz DP para Profibus, actúa como maestro del bus Profibus-DP.
5. Modulos de entrada/salida para la conexión de sensores y actuadores. Uno de estos módulos está directamente integrado en el PLC, mientras que el segundo se encuentra



(a) Modelo 3D de la mesa XYZ (b) Mesa XYZ real

Figura 8.2: Robot cartesiano

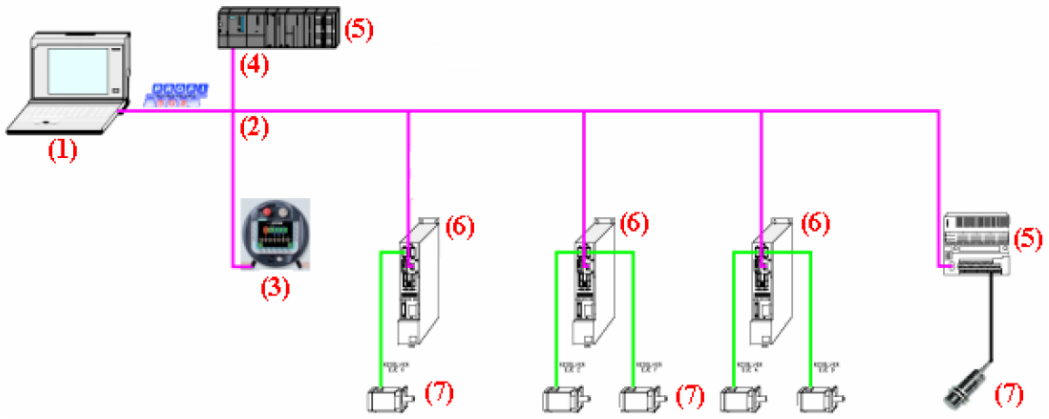


Figura 8.3: Arquitectura hardware de la mesa XYZ.

situado en el propio robot (y conectado por Profibus con el PLC), lo que reduce considerablemente el cableado y facilita la adición de nuevos sensores al sistema.

6. Módulo de regulación Simodrive 611U, elementos de control de los servomotores del sistema. Cada módulo consiste en una tarjeta para el control de dos ejes de manera independiente.

7. Diversos sensores (encoders, finales de carrera, etc.) conectados a los distintos módulos que conforman el robot.

8.3.2 IMPLEMENTACIÓN EN V^{3}STUDIO

El desarrollo de la aplicación de control para el robot cartesiano puede dividirse en dos partes, claramente diferenciadas: la programación del *driver* de comunicación con el PLC de Siemens a través del Profibus-DP, y el desarrollo de la arquitectura y la aplicación de control. El *driver* de comunicación con el Profibus-DP tuvo que desarrollarse de forma tradicional, ya que Siemens sólo proporciona una implementación en C de las librerías de comunicación. Una vez programado el *driver* de comunicación se pasó a desarrollar la arquitectura del sistema de control del robot cartesiano, mostrada en la figura 8.4. En la Tesis Doctoral del doctor Ortiz [181] se realiza una discusión pormenorizada de la arquitectura y de posibles variaciones, especialmente las que conciernen a la inclusión de componentes de tipo SC dentro de componentes MC. Como muestra la figura, la arquitectura del robot cartesiano está formada por los siguientes componentes:

- El componente `Driver_PLC` es un componente sencillo, diseñado *ad-hoc*, que contiene la lógica de comunicación con el bus Profibus-DP. Este «componente» no tiene máquina de estados debido a que la lógica de comunicación con el PLC es muy sencilla, ya que el programa que en él reside tan sólo redirige los comandos de movimiento al controlador correspondiente.

- El componente simple `Axis_X:SC` modela un controlador simple de uno de los ejes del robot. Las interfaces y los puertos coinciden con los que se expusieron en la sección 7.1.1. En este caso, los SC no controlan directamente los servomotores, sino que la labor de seguimiento de la posición y velocidad de los motores de cada eje es llevada a cabo por los Simodrive. La máquina de estados de este componente se corresponde con la que se describió en la sección 7.4.

- El componente `XZYtable:MC` es un componente complejo que modela el control sobre los tres ejes de la mesa. Para ello, contiene en su interior un controlador de tipo `Axis:SC` por cada uno de los tres ejes. Las interfaces de este componente coinciden, como muestra la figura 8.4, con las del componente `Driver_PLC` y con las de los tres controladores de los servomotores. En este caso, la máquina de estados de esta definición de componente complejo tan sólo redirige las peticiones de servicio al puerto correspondiente de cada SC.

- El componente simple `User Interface` controla el acceso a la funcionalidad del robot, representada por el componente `XZYtable:MC`. Este componente crea e inicializa un servidor para recibir peticiones de servicio a través de una conexión por *socket*, y posteriormente redirige todas estas peticiones hacia el componente MC.

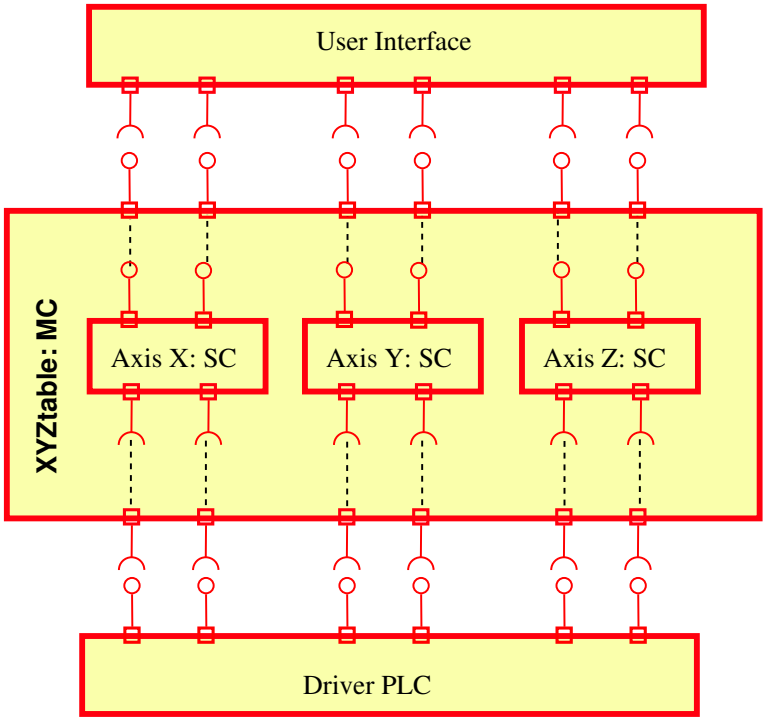


Figura 8.4: Arquitectura de control del robot cartesiano.

Cada uno de estos componentes se han traducido en las correspondientes definiciones de componente V³Studio, tal y como se ha descrito en la enumeración anterior. Los componentes (clase `Component` de V³Studio) que se corresponden con estas definiciones de componente se han configurado para ejecutarse en el mismo hilo de ejecución que su contenedor (consultar apartado 6.4.1), ya que la única actividad (no en el sentido UML de la palabra) que requiere concurrencia es realizada por los controladores de los servomotores.

La figura 8.5 muestra la interfaz gráfica de usuario, desarrollada en Java, con la que se puede teleoperar el robot cartesiano. Esta interfaz gráfica se ha programado de forma tradicional, pero está basada en las interfaces que se han utilizado a lo largo de este documento. Esta interfaz gráfica se comunica con el componente `User Interface` mediante el uso de *sockets*. En la figura 8.5 se pueden ver tres fila horizontales de botones con los comandos de teleoperación descritos en las interfaces del sistema, una fila por eje del robot, así como sendas gráficas en las que se representan la posición y la velocidad de cada eje, suministrados por los propios controladores de los servomotores.

A partir del modelo V³Studio, creado según se ha descrito en este apartado, se genera un modelo UML a partir de la transformación V³Studio → UML que se describió en el capítulo 7. Este modelo coincide, en gran parte, con la estructura de paquetes y clases que se describió en el ejemplo adelantado y simplificado del robot cartesiano que se utilizó en dicho capítulo. El modelo UML generado presenta las siguientes modificaciones respecto a los

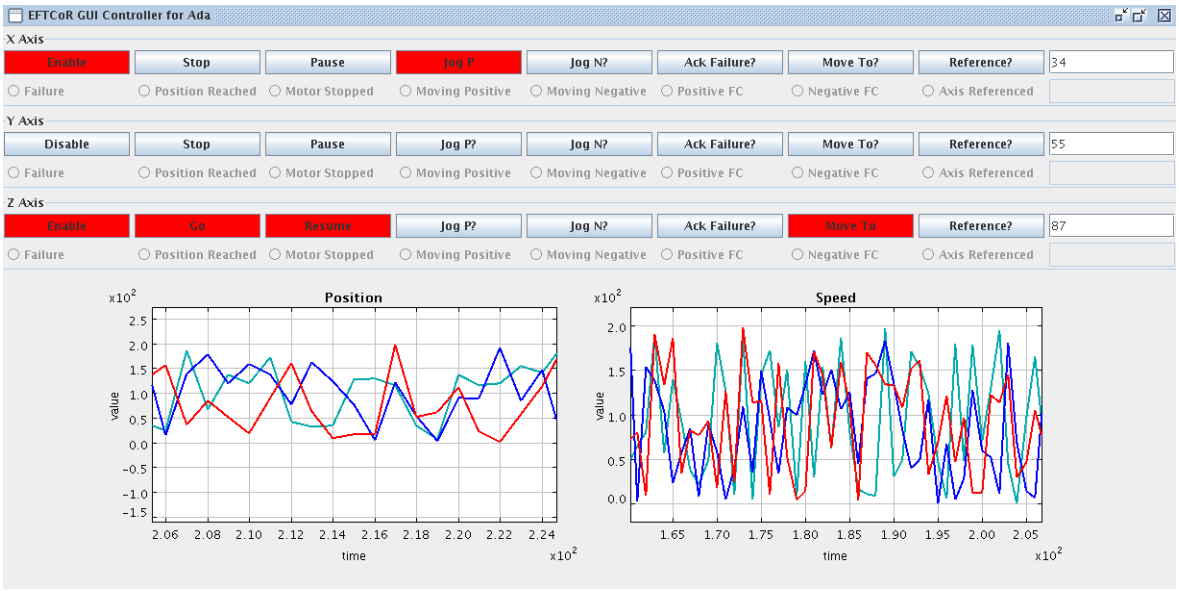


Figura 8.5: Interfaz gráfica de usuario desarrollada en Java para controlar el robot cartesiano.

modelos que se presentaron en el capítulo 7: (1) el componente `Driver_PLC` (`HAL` en el caso del ejemplo del capítulo 7) contiene tres instancias de los distintos tipos de puertos definidos en la arquitectura, una por eje a controlar; (2) el componente `XZYtable:MC` (`XYZ_Robot` en el caso del ejemplo del capítulo 7) contiene tres instancias del componente `Axis:SC`, una por cada eje; (3) se ha añadido el componente `User_Interface` (tanto en el diagrama de paquetes como en el de clases), junto con las operaciones asociadas al mismo. A partir de este modelo UML la generación del esqueleto de código Ada es directa, tal y como se ha expuesto al principio de este capítulo.

CAPÍTULO 9

CONCLUSIONES Y TRABAJOS FUTUROS

9.1 CONCLUSIONES

EL DESARROLLO de la arquitectura software de un sistema es una de las primeras y más importantes fases del proceso de desarrollo de una aplicación. Estudiada desde hace más de veinte años, la arquitectura software de un sistema determina gran parte de las propiedades y características que exhibirá el sistema, tanto para ser ampliado y mantenido como en tiempo de ejecución. Por tanto, el diseño arquitectónico puede marcar, ya desde un principio, la capacidad de adaptación de la aplicación a sus requisitos y, por tanto, las posibilidades de éxito de la misma. Prácticamente desde el principio de su existencia, esta rama de la Ingeniería Software adoptó los conceptos «componente», «puerto» y «conector» del paradigma de *Desarrollo Software Basado en Componentes* no sólo para diseñar la arquitectura de la aplicación sino también para analizar sus cualidades.

Sin embargo, y a pesar de la aparición y el desarrollo de UML, existe una gran diferencia entre el diseño arquitectónico de una aplicación y su posterior implementación en código. Este paso se suele realizar de manera tradicional por un programador que interpreta y traduce los diagramas UML al lenguaje de programación seleccionado. Con esta forma de proceder, el diseño arquitectónico queda reducido, en el mejor de los casos, a mera documentación gráfica de la aplicación, cuando no a un diseño que nada tiene que ver con el código que finalmente se ha escrito. El grupo de investigación DSIE se enfrentó a este mismo problema tras el desarrollo de la arquitectura ACRoSeT: el diseño arquitectónico en base a

componentes abstractos se quedaba como mera «documentación», ya que posteriormente había que implementarla de forma manual en el lenguaje de programación elegido.

En este contexto se enmarca el desarrollo de la presente Tesis Doctoral. El paradigma de *Desarrollo Software Dirigido por Modelos* se vislumbró como la tecnología adecuada para automatizar el proceso de desarrollo de las aplicaciones, en principio robóticas, que se desarrollan en el seno del grupo. La aplicación de la tecnología MDE ha permitido no sólo aumentar el nivel de abstracción y reutilización de los diseños (que ahora se realizan utilizando los conceptos CBD), sino también automatizar gran parte de la transformación de estos modelos hasta obtener un esqueleto del código de la aplicación.

Una parte del núcleo de esta Tesis Doctoral está constituido por el desarrollo de V^3Studio, un meta-modelo inspirado en UML pero que contiene únicamente los conceptos necesarios para modelar las características fundamentales de los sistemas basados en componentes que diseña el grupo de investigación DSIE. La utilización de la información específica del dominio y las características especiales del mismo han hecho posible el desarrollo de un meta-modelo adaptado a las necesidades del DSIE, fácil de comprender, fácil de utilizar y con la suficiente capacidad de modelado para ser aplicado incluso en otros dominios. Actualmente, se están realizando trabajos para utilizar V^3Studio en el marco de la definición de sistemas domóticos y redes de sensores.

La decisión de desarrollar un meta-modelo en lugar de utilizar directamente UML 2 o un perfil del mismo ha sido una decisión muy meditada en el seno del grupo y el autor es consciente de que genera cierta controversia en algunos círculos. Sin embargo, esta decisión se ha mostrado como la adecuada. Tras comenzar a utilizar V^3Studio, se ha constatado que la mayoría de usuarios del grupo han sido capaces de comprenderlo y utilizarlo sin grandes dificultades (a lo que ha contribuido sobremanera que previamente conocieran UML). Esta afirmación no sería realizable si se hubiera utilizado UML 2, debido a la gran cantidad de conceptos que contiene (más de 500 conceptos frente a los apenas 100 que tiene V^3Studio). Además, la gran cantidad de diagramas existentes y la mezcla de niveles de abstracción no facilitan la comprensión y utilización de UML.

La transformación entre el modelo V^3Studio y UML 2 constituye la otra parte del núcleo de esta Tesis Doctoral. Aunque se acaba de comentar que se prefirió desarrollar un meta-modelo inspirado en UML 2 en vez de utilizar el propio UML directamente, muchos de los conceptos presentes en V^3Studio (especialmente el concepto «componente») tienen un nivel de abstracción demasiado alejado de la implementación final en código, por lo que era necesario desarrollar una transformación adicional que creara modelos más cercanos a la plataforma. Para llevar a cabo esta traducción fue necesario estudiar las características primordiales de los componentes y realizar una búsqueda entre diversos patrones de diseños hasta encontrar un diseño que reflejara dichas características.

Además, la transformación de los modelos de componentes abstractos a distintas tecnologías de implementación (lenguajes y frameworks robóticos) es otro de los objetivos principales de esta Tesis Doctoral. Por tanto, se decidió utilizar UML 2 como el lenguaje que permite describir, de manera independiente de la plataforma final de ejecución, la estructura del sistema utilizando los conceptos de los lenguajes orientados a objetos, de forma que las transformaciones posteriores a código fueran lo más sencillas posible.

Otra de las ventajas de generar un modelo UML reside en el hecho de que se puede hacer uso del gran número de herramientas CASE que existen, no sólo para generar código sino también para llevar a cabo análisis sobre los propios modelos. Este trabajo de análisis redundará en una mejora de la transformación en sí misma. Las transformaciones de modelos, y concretamente la transformación modelo-a-modelo, es uno de los mecanismos más importantes y poderosos de MDE. Gracias a ellas es posible crear varios meta-modelos con el grado de detalle y abstracción requeridos por las personas que los van a utilizar y mantener, hasta cierto punto, la coherencia y la sincronización entre todos ellos, evitando la aparición de modelos obsoletos que no reflejan la estructura de la aplicación final.

Por último, se ha esbozado una posible transformación a código Ada del modelo UML obtenido tras la ejecución de la transformación de modelos V^3Studio a UML. Esta transformación a código es relativamente sencilla y directa, puesto que la complicación se encuentra embebida en la transformación al modelo UML, en la que se generan las implementaciones de los componentes del sistema, de parte de los servicios auxiliares y de algunas operaciones. El esqueleto de la aplicación final que se genera (1) contiene la definición de los componentes y los puertos que forman la aplicación y la implementación de los mismo, (2) establece el esquema de concurrencia en función de los modelos creados por el usuario, (3) controla el comportamiento de los componentes mediante el uso del patrón *Active Object* y (4) propone diversas implementaciones de la lógica de la máquina de estados asociada al comportamiento del componente.

Aunque la presente Tesis Doctoral se desarrolla en el marco de los sistemas robóticos, el enfoque y las herramientas utilizadas y desarrolladas pueden ser igualmente aplicable en otros entornos y otros proyectos. El *Desarrollo Software Dirigido por Modelos* se vislumbra como una tecnología realmente evolucionaria, que va a permitir la creación de programas cada vez más fácil y rápidamente. Además, estos programas generados automáticamente en base a transformaciones de modelos serán cada vez más seguros y fiables conforme aumenten el conjunto de herramientas y tecnologías disponibles. Ahora sí, la Ingeniería del Software avanza con paso firme hacia la obtención de un verdadero cuerpo de conocimiento que le permita afrontar con garantías de éxito el desarrollo de aplicaciones cada vez más complejas. Aunque todavía queda mucho camino que recorrer y muchos retos por superar para que el enfoque sea ampliamente adoptado y aceptado.

En cuanto a las herramientas utilizadas (concretamente el entorno Eclipse), si bien es la única herramientas que proporciona el soporte necesario para realizar un diseño dirigido por modelos, se encuentra todavía en fase de desarrollo. La gran cantidad de plug-ins que existen, junto con la dificultad añadida por las diversas incompatibilidades que existen entre las diversas versiones, dificultan la utilización del entorno, resultando bastante complicado y trabajoso obtener un versión «estable», en la que funcionen todas las versiones correctamente. Sin embargo, sin duda alguna, Eclipse es una plataforma con un gran potencial, cuyo uso en la comunidad software se va a ver incrementado en los próximos años.

9.2 APORTACIONES DE ESTA TESIS DOCTORAL

A CONTINUACIÓN se enumeran las principales aportaciones y características más sobresalientes de la presente Tesis Doctoral. En opinión del autor, el trabajo que se presenta cumple con los objetivos y las expectativas marcados en el capítulo de introducción, aunque es inevitable que se hayan producido ligeros cambios según se iba desarrollando y evolucionando el trabajo de investigación.

- Se ha propuesto un meta-modelo para modelar aplicaciones software basadas en componentes, denominado V^{3}Studio. V^{3}Studio propone un meta-modelo inspirado en UML 2, ya que se decidió reutilizar al máximo el conocimiento embebido en este lenguaje, para modelar tres vistas de una aplicación:
  - **Vista arquitectónica**, en la que el usuario puede describir los servicios, interfaces y tipos de datos globales de la aplicación, así como los componentes que describen la arquitectura del sistema.
  - **Vista de comportamiento**, en la que el usuario describe, mediante una máquina de estados, el comportamiento propio del componente y su reacción ante las peticiones de servicio del resto de componentes que integran la aplicación.
  - **Vista algorítmica**, en la que el usuario describe, mediante diagramas de actividades, el algoritmo que define el comportamiento asociado a cada uno de los estados y transiciones de la máquina de estados que describe el comportamiento de un componente.
- V^{3}Studio ha sido diseñado para permitir la reutilización de los modelos creados en cada una de las vistas y para permitir la generación de editores separados que faciliten el trabajo del usuario, que sólo se tiene que concentrar en un conjunto reducido de conceptos. Para aumentar el número de posibles escenarios en que se pueden reutilizar

los modelos, V^{3}Studio distingue los conceptos «definición» e «instancia», y permite configurar y parametrizar las definiciones para ampliar el rango de utilización de las instancias.

- Se ha propuesto una traducción de los conceptos básicos del desarrollo basado en componentes a los conceptos de los lenguajes orientados a objetos. Tras realizar un estudio de los principales patrones de diseño software, finalmente se optó por la utilización de los patrones de diseño *Active Object* y *Façade* para implementar los componentes y el patrón *Command* para los servicios. El mantenimiento de la característica de encapsulación del contenido de los componentes y la propuesta de un traducción sencilla pero flexible han sido las dos guías principales de esta transformación.

- Se ha desarrollado una transformación de modelos para generar un modelo UML a partir del modelo V^{3}Studio. Este modelo de implementación UML contiene no sólo el diagrama de clases y paquetes derivado de la traducción de los conceptos CBD descrita en el punto anterior, sino que también contiene la implementación de gran parte de las operaciones y la estructura del framework de soporte para la ejecución de los componentes.

- Se ha esbozado una transformación modelo-a-texto para la generación de código Ada a partir del modelo UML de la aplicación de componentes. El esqueleto que genera la transformación demuestra la viabilidad del enfoque, ya que se crea la infraestructura de clases y operaciones necesaria para que el usuario pueda concentrarse en completar el código de los servicios en vez de en programar la estructura de los componentes y el cuerpo de las operaciones.

9.3 Divulgación de Resultados

Los resultados obtenidos como parte de la realización de esta Tesis Doctoral se han publicado en las revistas y congresos que se enumeran a continuación. Al final de este apartado se describen someramente los principales proyectos de investigación que han financiado este trabajo y en cuyo marco se ha desarrollado esta Tesis Doctoral.

CONGRESOS INTERNACIONALES

- Ortiz, F.; Alonso, D.; Álvarez, B. y Pastor, J.A.: «A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle». En: 10^{th} *International Conference on Reliable Software Technologies, volumen 3555 de Lecture Notes on Computer Science*, pp. 13–24. ISSN 0302-9743, 2005.

- Ortiz, F.; Pastor, J.A.; Alonso, D.; Losilla, F. y de Jódar, E.: «A reference architecture for managing variability among teleoperated service robots». En: 2^{nd} *International Conference on Informatics in Control, Automation and Robotics*, pp. 322–328. ISBN 9728865309, 2005.

- Alonso, D.; Sánchez, P.; Álvarez, B. y Pastor, J.A.: «A systematic approach to developing safe tele-operated robots». En: 11^{th} *International Conference on Reliable Software Technologies, volumen 4006 de Lecture Notes on Computer Science*, pp. 119–130. ISSN 0302-9743, 2006.

- Alonso, D.; Vicente-Chicote, C.; Sánchez, P.; Álvarez, B. y Losilla, F.: «Automatic Ada Code Generation Using a Model-Driven Engineering Approach». En: 12^{th} *International Conference on Reliable Software Technologies, volumen 4498 de Lecture Notes on Computer Science*, pp. 168–179. ISSN 0302-9743, 2007.

- Ortiz, F.; Pastor, J.A.; Alonso, D.; Álvarez, B. y Sánchez, P.: «Experiences using a component-oriented architectural framework for tobots and its improvement with a MDE approach». En: *Software architecture (*1^{st} *European Conference, ECSA 2007), volumen 4758 de Lecture Notes on Computer Science*, pp. 335–338. ISSN 0302-9743, 2007.

- Vicente-Chicote, C.; Alonso, D. y Chauvel, F.: «V^{3}Studio: a component based architecture description meta-model - extensions to model component behaviour variability». En: 2^{nd} *International Conference on Software and Data Technologies*, pp. 437–440. ISBN 9789898111067, 2007.

- Alonso, D.; Vicente-Chicote, C. y Barais, O.: «V^{3}Studio: a component-based modelling language». Aceptado en 15^{th} IEEE International Conference on Engineering of Computer-Based Systems, 2008.

Congresos Nacionales

- Vicente-Chicote, C.; Alonso, D. y Álvarez, B.: «StateML: modelado gráfico de máquinas de estados y generación de código siguiendo un enfoque MDE». En: *Actas de las XII Jornadas de Ingeniería Software y Bases de Datos*, pp. 401–402. ISBN 9788497325950, 2007.
- Vicente-Chicote, C.; Alonso, D. y Barais, O.: «V^{3}Studio: un entorno gráfico para el diseño de sistemas basados en componentes siguiendo un enfoque dirigido por modelos». En: *Actas de las XII Jornadas de Ingeniería del Software y Bases de Datos*, pp. 403–404. ISBN 9788497325950, 2007.

Otras Publicaciones de Interés

- Vicente-Chicote, C. y Alonso, D.: Tutorial titulado «Herramientas Eclipse para el Desarrollo Software Dirigido por Modelos», 2007, 1(8), Sistedes. ISSN 1988-3455.
- Ortiz, F.; Álvarez, B.; Sánchez, P. y Alonso, D.: «Arquitectura para control de robots de servicio teleoperados». Revista Telecoforum III, pp. 3–8. ISSN 1698-2924, 2005.
- Ortiz, F.; Alonso, D.; Pastor, J.A.; Álvarez, B. e Iborra, A.: «A Reference Control Architecture for Service Robots as applied to a Climbing Vehicle». Capítulo del libro *Bioinspiration and Robotics Walking and Climbing Robots*, pp. 187-208. Ed. Advanced Robotic Systems International and I-Tech. ISBN 9783902613158, 2007.

Marco de Desarrollo

Además de las publicaciones que han sido enumeradas, esta Tesis Doctoral se ha desarrollado en el marco de los siguientes proyectos financiados por el Ministerio de Ciencia y Tecnología:

- **DYNAMICA** — subproyecto **ANCLA** (*Arquitecturas Dinámicas para Sistemas de Teleoperacion*). TIC 2003–07804–C05–02.
  Entidades participantes: Universidad Politécnica de Cartagena, Universidad Carlos III, Universidad Politécnica de Valencia, Universidad de Castilla-La Mancha, Universidad de Murcia.
  Duración: desde 01/12/2003 hasta 30/11/2006.
- **META** — subproyecto **MEDWSA** (*Marco conceptual y tecnológico para el desarrollo de software de sistemas reactivos*). TIN 2006–15175–C05–02.

Entidades participantes: Universidad Politécnica de Cartagena, European Software Institute (ESI), Universidad Politécnica de Valencia, Universidad de Castilla-La Mancha, Universidad de Murcia.
Duración: desde 01/01/2007 hasta 31/12/2009.

9.4 TRABAJOS FUTUROS

FINALMENTE, este último apartado recoge las principales líneas de investigación, algunas de las cuáles ya se encuentran en marcha en el marco de MEDWSA, en las que se va a seguir trabajando en el seno del grupo DSIE. Estas líneas, en algunas de las cuales se está ya trabajando, se han clasificado en tres apartados, que coinciden con cada una de las etapas del desarrollo software dirigido por modelos: mejoras en el modelado de los sistemas, mejoras en la transformación de modelos y mejoras en la transformación a código.

AMPLIACIÓN DE LA CAPACIDAD DE MODELADO DE V^{3}STUDIO

- Como ya se ha comentado, en el contexto del proyecto MEDWSA V^{3}Studio aporta un modelos de componentes intermedio e independiente de la plataforma (PIM), sobre el que se pueden transformar los distintos modelos creados con los DSLs que se están desarrollando para especificar aplicaciones domóticas y de redes de sensores (ver figura 9.1). Al ampliar el campo de uso es posible que la versión de V^{3}Studio descrita en esta Tesis Doctoral tenga que ser ampliada para adaptarse a alguno de los requisitos especiales de estos dominios.

- Dentro también del proyecto MEDWSA, pero de forma independiente a lo que se ha resaltado en el punto anterior, también será preciso ampliar el meta-modelo para modelar líneas de producto software. Para ello será necesario dotar a V^{3}Studio de conceptos para modelar la arquitectura genérica de la línea, clasificar componentes y establecer relaciones entre ellos, tanto de obligatoriedad como de opcionalidad.

- Desarrollo e integración del concepto «conector» en V^{3}Studio, acompañándolo de una serie de conectores tipo. Aunque finalmente este «conector» se traduciría en uno o varios componentes para llevar a cabo su funcionalidad, la especificación de distintos tipos de conector liberaría al diseñador de la tarea de tener que crear e inicializar los puertos correctos con las interfaces adecuadas a los componentes que se quieren conectar por medio de dicho conector. El uso de «conectores» permitiría, además, especificar y modelar la creación de sistemas distribuidos.

Figura 9.1: Esquema de desarrollo del proyecto MEDWSA, en el que puede observarse el papel central que desempeña V^{3}Studio.

- Creación de un editor gráfico para desarrollar los modelos de cada una de las vistas. También será necesario establecer un repositorio de modelos para centralizar su almacenamiento y ser capaces efectivamente de reutilizarlos.

- La capacidad de crear y validar los modelos de las distintas vistas por separado y de reutilizarlos en distintos contextos y aplicaciones presenta como contrapartida que no siempre es posible validar completamente los modelos, ya que hay algunas restricciones OCL que no pueden verificarse hasta el paso en que se relacionan finalmente los distintos modelos entre sí.

Mejoras en la Transformación a un Modelo UML

- Utilización de otras herramientas CASE de análisis de modelos UML. Este tipo de herramientas permitirán mejorar algunos apartados de la transformación de modelos y servirán para analizar la transformación en conjunto y proponer posibles mejoras.

- Generación de modelos para realizar el análisis de planificabilidad del sistema. A partir de la especificación del número de tareas presentes en el sistema se podría solicitar al diseñador una estimación temporal de la ejecución de las distintas actividades. Con esta información sería posteriormente posible realizar un análisis y simulación temporal del sistema.

- Desarrollo de traducciones alternativas de los conceptos «componente» y «puerto». Concretamente, se estudiarán otras posibles transformaciones que utilicen patrones alternativos al *Active Object* o incluso otras técnicas más cercanas al desarrollo de middleware, como por ejemplo el canal de eventos.
- Definición de un conjunto de servicios básicos que debería proporcionar el framework de componentes. Estos servicios formarían parte de la interfaz de la clase componente, que actualmente se genera vacía para que el diseñador adapte el código generado a los requisitos de su aplicación.
- Validación y verificación de la transformación. Aunque para ello primero se deben desarrollar las técnicas adecuadas.

MEJORAS EN LA TRANSFORMACIÓN A CÓDIGO

- Desarrollo de la transformación a otros lenguajes de programación, concretamente C/C++, ya que son los más utilizados en la comunidad del software embebido. De forma paralela, y dentro del proyecto MEDWSA, también será necesario desarrollar transformaciones a los lenguajes propios de dominios tan cerrados como la domótica o las redes de sensores, en las que el lenguaje dominante es *NesC*.
- Utilización de herramientas CASE para el análisis del código obtenido tras la transformación, ya que este tipo de análisis ayudaría a mejorar el comportamiento global del sistema en ejecución e incluso ayudaría a modificar la transformación de modelos V^{3}Studio a UML para mejorar la eficiencia del código final.
- Utilización de herramientas CASE para la generación automática de código a partir de modelos UML, como por ejemplo Telelogic Rhapsody®. El uso de herramientas comerciales para llevar a cabo esta última etapa del desarrollo aceleraría la generación y la confianza en el código generado en distintos lenguajes, ya que en el futuro estas herramientas serán cada vez mejores y permitirán generar código rápida y fácilmente para cualquier lenguaje.

APÉNDICE A

RESTRICCIONES OCL ADICIONALES

ESTE apéndice agrupa las restricciones OCL [174] que se han definido en el contexto de V^{3}Studio y que permiten asegurar la coherencia semántica de los modelos que con él se realicen. OCL es un lenguaje de especificación de restricciones que permiten completar la sintaxis definida por un meta-modelo. Un meta-modelo, por sí mismo, sólo define el conjunto de elementos del dominio y las relaciones entre ellos, pero no puede establecer restricciones extras sobre ninguno de sus elementos. Estas restricciones extra, del tipo «sólo se pueden conectar los puertos compatibles de dos componentes», tienen que ser añadidas y comprobadas posteriormente.

El anexo está organizado de la misma manera que se ha organizado el capítulo 6, ya que este capítulo contiene la descripción de V^{3}Studio, que se completa con la adición de las restricciones OCL que se muestran en este apéndice. Concretamente, este apéndice presenta tres secciones en las que se resumen las principales restricciones que completan la vista arquitectónica, la vista de comportamiento y la vista algorítmica.

A.1 VISTA ARQUITECTÓNICA

ESTA sección recoge las restricciones OCL que están relacionadas con la vista arquitectónica definida por V^{3}Studio, descrita en el capítulo 6.4. La mayor parte de las restricciones que aquí se recogen están relacionadas con la EClass `Component` (que representa una «instancia») y sus numerosas relaciones y atributos, aunque también es necesario asegurar la corrección de la definición de interfaces y servicios así comprobar que las conexiones entre puertos son correctas. Todas estas restricciones están recogidas en el siguiente enumerado.

Restricción 1: En un puerto (EClass `Port`, ver apartado 6.4.3) no puede aparecer la misma interfaz repetida en ninguno de los conjuntos de interfaces ofrecidas ni requeridas. La misma interfaz tampoco puede ser ofrecida y requerida a la vez.

```
1  context Port
2    inv repeatedRequiredInterface:
3      self.requiredInterface -> forAll (i : Interface |
4        let tmp : Bag(Interface) = self.requiredInterface -> select (j : Interface | j = i) in
5        tmp.size = 1
6      )
7    inv repeatedProvidedInterface:
8      self.providedInterface -> forAll (i : Interface |
9        let tmp : Bag(Interface) = self.providedInterface -> select (j : Interface | j = i) in
10       tmp.size = 1
11     )
12   inv requiringAndProvidingSameInterface:
13     self.providedInterface -> forAll (i : Interface |
14       not self.requiredInterface -> exists (j : Interface | j = i)
15     )
```

Restricción 2: En una interfaz no se puede repetir un mismo servicio. Dos servicios son iguales si tienen el mismo nombre y los mismos parámetros. Ver apartado 6.4.3.

```
1  context Interface
2    inv repeatedService:
3      self.service -> forAll (s1, s2 : Service |
4        s1 <> s2 implies s1.name <> s2.name and s1.parameter -> forAll (j : ServiceParameter |
5          s2.parameter->forAll (k : ServiceParameter | k.name <> j.name or k.type <> j.type
6          or k.direction <> j.direction)
7        )
8      )
```

Restricción 3: Un `DelegationLink` sólo puede conectar puertos que ofrecen y requieren las mismas interfaces, ya que un `ComplexComponentDefinition` delega la ejecución de la petición de servicio a los `Component` contenidos en éste último.

```
1  context DelegationLink
2    inv compatiblePorts_required_endA:
3      self.endA.port.requiredInterface -> forAll (i : Interface |
4        self.endB.port.requiredInterface -> exists (j : Interface | j = i)
5      )
6    inv compatiblePorts_provided_endA:
7      self.endA.port.providedInterface -> forAll (i : Interface |
8        self.endB.port.providedInterface -> exists (j : Interface | j = i)
9      )
10   inv compatiblePorts_required_endB:
11     self.endB.port.requiredInterface -> forAll (i : Interface |
12       self.endA.port.requiredInterface -> exists (j : Interface | j = i)
13     )
14   inv compatiblePorts_provided_endB:
15     self.endB.port.providedInterface -> forAll (i : Interface |
16       self.endA.port.providedInterface -> exists (j : Interface | j = i)
17     )
```

Restricción 4: Un `AssemblyLink` sólo puede conectar puertos *compatibles*, i.e. puertos en los que todos los servicios requeridos son ofrecidos por el conjugado y viceversa. Más detalles en el apartado 6.4.4.

```
1  context AssemblyLink
2    inv compatiblePorts_required_endA:
3      self.endA.port.requiredInterface -> forAll (i : Interface |
4        self.endB.port.providedInterface -> exists (j : Interface | j = i)
5      )
6    inv compatiblePorts_provided_endA:
7      self.endA.port.providedInterface -> forAll (i : Interface |
8        self.endB.port.requiredInterface -> exists (j : Interface | j = i)
9      )
10   inv compatiblePorts_required_endB:
11     self.endB.port.requiredInterface -> forAll (i : Interface |
12       self.endA.port.providedInterface -> exists (j : Interface | j = i)
13     )
14   inv compatiblePorts_provided_endB:
15     self.endB.port.providedInterface -> forAll (i : Interface |
16       self.endA.port.requiredInterface -> exists (j : Interface | j = i)
17     )
```

Restricción 5: Un `AssemblyLink` sólo puede conectar puertos que pertenezcan a distintos componentes que estén contenidos en una misma definición de componente complejo (`ComplexComponentDefinition`). Consultar el apartado 6.4.4.

```
1  context AssemblyLink
2    inv selfComponentPortConnectionNotAllowed_AssemblyLink:
3      self.endA.component <> self.endB.component
4    inv componentsContainedInSameComplexComponent_AssemblyLink:
5      self.endA.component.owner = self.endB.component.owner
```

Restricción 6: El requisito (`ServiceRequirement`) que un componente (`Component`) impone a cada uno de los servicios que requiere tiene que ser acorde al tipo de servicio (`ServiceKind`) definido en la interfaz. Consultar los apartados 6.4.3 y 6.4.1.

```
1  context ServiceRequirement
2    inv incorrectServiceRequirement:
3      if (self.service.kind = ServiceKind :: synchronous) then
4        self.requirementKind = ServiceRequirementKind :: polling or
5          self.requirementKind = ServiceRequirementKind :: subscribe
6      else
7        if (self.service.kind = ServiceKind :: asynchronous) then
8          self.requirementKind = ServiceRequirementKind :: woNotification or
9            self.requirementKind = ServiceRequirementKind :: wNotification
10       endif
11     endif
```

Restricción 7: No se puede repetir varias veces la misma interfaz. Dos interfaces son iguales si tienen los mismos servicios.

```
1  context Architecture
2    inv repeatedInterface:
3      self.interface -> forAll (i1, i2 : Interface |
4        i1 <> i2 implies i1.name <> i2.name and i1.parameter -> forAll (j : ServiceParameter |
5          i2.parameter->forAll (k : ServiceParameter | k.name <> j.name and k.type <> j.type)
6        )
7      )
```

Restricción 8: El puerto del servicio sobre el que se especifica un `ServiceRequirement` tiene que pertenecer a la definición del componente que contiene este requisito, tal y como se describe en el apartado 6.4.1.

```
1  context Component
2    inv portReferedByServiceRequirementMustBelongToComponentDefinition:
3      self.serviceRequirement -> forAll (s : ServiceRequirement |
4        self.type.port -> exists (p : Port | p = s.port)
5      )
```

Restricción 9: Un `DelegationLink` sólo puede conectar el puerto contenido en un `ComplexComponentDefinition` con el de un `Component` contenido en aquel. Además, la referencia `ccdPort` de un `DelegationLink` debe apuntar a un puerto contenido por el mismo `ComplexComponentDefinition` que contiene a este `DelegationLink`. Consultar el apartado 6.4.4.

```
1 context DelegationLink
2   inv componentContainedInSameComplexComponent_DelegationLink:
3     self.endA.component.owner = ccdPort.owner
4   inv ccdPortContainedInSameComplexComponent_DelegationLink:
5     self.ccdPort.owner = self.owner
```

Restricción 10: El puerto de conexión que especifica un `BindingEnd` tiene que pertenecer a la definición del componente que especifica el mismo `BindingEnd`.

```
1 context BindingEnd
2   inv portReferedByBindingMustBelongToComponentDefinition:
3     self.component.type.port -> exists (p : Port | p = self.port)
```

Restricción 11: Las conexiones entre puertos sólo se pueden realizar punto-a-punto. No se permiten conexiones uno-a-muchos o muchos-a-uno en la versión actual de V^{3}Studio. Consultar apartado 6.4.4.

```
1  context ComplexComponentDefinition
2    inv oneAndOnlyOneConnectionAmongPorts_AssemblyLink:
3      self.component -> forAll (i : Component |
4        let tmp : Bag (Port) = i.type.port -> collect (j : Port |
5          self.binding -> select (k : AssemblyLink |
6            (k.endA.component = i and k.endA.port = j) or
7            (k.endB.component = i and k.endB.port = j)
8          )
9        ) in
10       tmp.size = 1
11     )
12   inv oneAndOnlyOneConnectionAmongPorts_DelegationLink:
13     self.component -> forAll (i : Component |
14       let tmp : Bag (Port) = i.type.port -> collect (j : Port |
15         self.binding -> select (k : DelegationLink |
16           (k.endA.component = i and k.endA.port = j) or
17           (k.ccdPort = j)
18         )
19       ) in
20       tmp.size = 1
21     )
```

Restricción 12: No se puede repetir el mismo tipo de dato (`DataType`).

```
context Architecture
  inv repeatedDataType:
    self.dataType -> forAll (d1, d2 : DataType | d1 <> d2 implies d1.name <> d2.name)
```

Restricción 13: Todos los servicios requeridos por un componente tienen que estar definidos por uno y sólo un `ServiceRequirement`. Ver apartado 6.4.3.

```
context Component
  inv oneAndOnlyServiceRequirementPerService:
    self.type.port -> forAll (p : Port |
      let tmp : Bag (ServiceRequirement) = self.serviceRequirement ->
        select (i : ServiceRequirement | i.port = p) in
      p.requiredInterface -> forAll (i : Interface |
        i.service -> forAll (s : Service |
          let tmp2 : Bag (ServiceRequirement) = tmp ->
            select (k : ServiceRequirement | k.service = s) in
          tmp2.size = 1
        )
      )
    )
```

Restricción 14: Todos los puertos de todos los componentes (`Component`) de una definición de componente complejo tienen que estar conectados.

```
context ComplexComponentDefinition
  inv allPortsConnected:
    self.component -> forAll (i : Component |
      i.type.port -> forAll (j : Port |
        j.connected.size = 1
      )
    )
```

Restricción 15: Todos los puertos de todos los componentes (`Component`) de una definición de componente complejo tienen que estar conectados.

```
context ComplexComponentDefinition
  inv allPortsConnected:
    self.component -> forAll (i : Component |
      i.type.port -> forAll (j : Port |
        j.connected.size = 1
      )
    )
```

Restricción 16: Todas las actividades asociadas a la máquina de estados de un componente complejo tienen que ser de tipo `ForwareService`. Además, el servicio redirigido debe aparecer como ofrecido por el componente complejo.

```
1 context ComplexComponentDefinition
2   inv allActivitiesOfStateMachineDefinitionAreForwardService:
3     Activity.allInstances() -> forAll (a : Activity | a.oclIsTypeOf(ForwardService))
4   inv forwardedServiceMustBeOwnedByComplexComponent:
5     let tmp : Bag(Service) = self.port -> collect (p : Port | p.providedService ->
6       collect (s : Service)) in
7     Activity.allInstances() -> forAll (a : Activity | tmp -> exists (s : Service |
8       s = a.service
9     )
```

A.2 VISTA DE COMPORTAMIENTO

ESTA sección recoge las restricciones OCL que están relacionadas con la vista de comportamiento (consultar capítulo 6.5), vista que está basada en las máquinas de estados definidas por UML. La mayor parte de las restricciones que aquí se recogen comprueban que las transiciones están correctamente definidas y que los pseudo-estados son se utilizan adecuadamente. El otro gran bloque de restricciones está relacionado con las clases que permiten parametrizar las máquinas de estados y establecer actividades que pueden variar entre distintas «instancias». El siguiente enumerado resume las restricciones OCL que se han añadido a esta vista de comportamiento.

Restricción 17: Un pseudo-estado de tipo `joinState` sólo tiene una transición saliente. Además, todas las transiciones entrantes tienen que proceder de estados contenidos en regiones ortogonales contenidas en un mismo macro-estado. En el apartado 6.5.4 se describe este pseudo-estado en detalle.

```
1 context Pseudostate
2   inv joinStateWith1OutgoingTransition:
3     if (self.kind = PseudostateKind :: joinState) then
4       self.outgoing.size = 1
5     endif
6   inv joinTransitionsComeFromOrthogonalRegionsInSameMacroState:
7     if (self.kind = PseudostateKind :: joinState) then
8       self.incoming -> forAll (s1, s2 : Transition |
9         s1 <> s2 implies s1.owner <> s2.owner and s1.owner.owner = s2.owner.owner
10      )
11    endif
```

Restricción 18: Los pseudo-estados de tipo inicial o histórico sólo tienen una transición saliente (y sin guarda) y no tienen transiciones entrantes. Más información en el apartado 6.5.3.

```
1  context Pseudostate
2    inv initialOrHistoryPseudostateWithIncomingTransition:
3      if ((self.kind = PseudostateKind :: initialState ) or
4      (self.kind = PseudostateKind :: historyState)) then
5        self.incoming.size = 0
6      endif
7    inv initialOrHistoryPseudostateWithoutOutgoingTransition:
8      if ((self.kind = PseudostateKind :: initialState ) or
9      (self.kind = PseudostateKind :: historyState)) then
10       self.outgoing.size = 1 and self.outgoing -> first(). guard.size = 0 and
11       self.outgoing -> first(). trigger.kind = TriggerKind :: activityCompletion and
12       self.outgoing -> first(). kind = TransitionKind :: external
13     endif
```

Restricción 19: Los atributos del disparador (`Trigger`) de una transición tienen los valores adecuados al tipo de fuente que activa al disparador (ver apartado 6.5.2).

```
1  context Trigger
2    inv incorrectTrigger:
3      if (self.kind = TriggerKind :: service) then
4        self.service.size = 1
5      else
6        if (self.kind = TriggerKind :: timeout) then
7          self.timeout > 0
8        endif
9      endif
```

Restricción 20: Un `joinState` no puede ser alcanzado por más transiciones que regiones ortogonales tiene el macro-estado de partida de las transiciones. Además, no puede salir más de una transición de la misma región ortogonal.

```
1  context Pseudostate
2    inv noMoreIncomingTransitionsThanRegions:
3      if (self.kind = PseudostateKind :: joinState) then
4        self.incoming.size = self.incoming -> first().source.owner.owner.region.size
5      endif
6    inv noMoreThanOneTransitionPerRegionToJoin:
7      if (self.kind = PseudostateKind :: joinState) then
8        self.incoming -> forAll (t1,t2 : Transition |
9          t1 <> t2 implies t1.source.owner <> t2.source.owner)
10     endif
```

Restricción 21: Las actividades `entry`, `doActivity` y `exit`, realizadas por un estado, y el `effect` de una transición, siempre que estén definidos, no pueden ser de tipo `PseudoActivity`, ver apartado 6.5.2. Esta restricción está a medio camino entre la vista de comportamiento y la vista algorítmica, pero aparece aquí ya que los elementos de contexto pertencen a la vista de comportamiento.

```
1  context Transition
2    inv activityIsNotPseudoActivity:
3      if (self.effect.size = 1) then
4        not self.effect.oclIsTypeOf (PseudoActivity)
5      endif
6  context State
7    inv entryIsNotPseudoActivity:
8      if (self.entry.size = 1) then
9        not self.entry.oclIsTypeOf (PseudoActivity)
10     endif
11   inv exitIsNotPseudoActivity:
12     if (self.exit.size = 1) then
13       not self.exit.oclIsTypeOf (PseudoActivity)
14     endif
15   inv doActivityIsNotPseudoActivity:
16     if (self.doActivity.size = 1) then
17       not self.doActivity.oclIsTypeOf (PseudoActivity)
18     endif
```

Restricción 22: Una transición de tipo (`TransitionKind`) interna tiene el mismo origen y destino. Consultar apartado 6.5.2.

```
1  context Transition
2    inv incorrectInternalTransition :
3      if (self.kind = TransitionKind :: internal) then
4        self.source = self.target
5      endif
```

Restricción 23: En cada región tiene que haber uno y sólo un pseudo-estado de tipo inicial o histórico. Más información en el apartado 6.5.3.

```
1  context Region
2    inv noInitialOrHistoricPseudostateInRegion:
3      let tmp : Bag (Pseudostate) = self.vertex -> select (v : Vertex |
4          v.oclIsTypeOf (Pseudostate)) in
5      tmp -> select (v : Pseudostate | v.kind = PseudostateKind :: initialState ).size = 1 xor
6      tmp -> select (v : Pseudostate | v.kind = PseudostateKind :: historyState ).size = 1)
7    )
```

Restricción 24: De un `forkState` no pueden salir más transiciones que regiones ortogonales tiene el macro-estado que se alcanza. Además, ninguna región ortogonal puede ser alcanzada más de una vez.

```
1  context Pseudostate
2    inv noMoreOutgoingTransitionsThanRegions:
3      if (self.kind = PseudostateKind :: forkState) then
4        self.outgoing.size = self.outgoing -> first (). target.owner.owner.region.size
5      endif
6    inv regionCannotBeTargetedMoreThanOnceByFork:
7      if (self.kind = PseudostateKind :: forkState) then
8        self.outgoing -> forAll (t1,t2 : Transition |
9          t1 <> t2 implies t1.target.owner <> t2.target.owner)
10     endif
```

Restricción 25: Todos los puntos de variación (`ActivityVariabilityPoint`) de la definición de una máquina de estados tienen que estar completados en la correspondiente instancia por uno y sólo un `ActivityBinding`. La parametrización de las máquinas de estados se trata en el apartado 6.5.1.

```
1  context StateMachine
2    inv variabilityPointsCannotBeCompletedMoreThanOnce:
3      self.binding -> forAll (b1, b2 : ActivityBinding |
4        b1 <> b2 implies b1.from <> b2.from)
5    inv sameNumberOfVariabilityPointsAndBindings:
6      self.binding.size = self.definition.instanceActivity.size
7    inv variabilityPointsOwnedByCorrectDefinitionAndInstance:
8      self.from -> forAll (a : ActivityVariabilityPoint |
9        a.owner = self.definition
10     )
```

Restricción 26: Todos los vértices (EClass abstracta `Vertex`) de una máquina de estados tienen que estar conectados.

```
1  context Vertex
2    inv connected:
3      self.outgoing.size > 0 or self.incoming.size > 0
```

Restricción 27: La definición y la instancia de un componente y de su respectiva máquina de estados tienen que ser acordes. Consultar apartado 6.5.1.

```
1  context Component
2    inv incoherentFSM:
3      self.statemachine.definition = self.type.behavior
```

Restricción 28: Un estado final (EClass `FinalState`) no tiene transiciones de salida, ya que marca el final de la máquina de estados. Por tanto, tampoco debe tener definida la actividad `exit`. Más información sobre el estado final en el apartado 6.5.2.

```
1 context FinalState
2   inv finalStateWithOutgoingTransition:
3     self.outgoing.size <> 0
4   inv noExitActivityInFinalState:
5     self.exit.size= 0
```

Restricción 29: Las transiciones que llegan a un pseudo-estado `joinState` no tienen guarda. Sólo la transición de salida puede tenerla en este caso.

```
1 context Pseudostate
2   inv noGuardInJoinIncomingTransitions:
3     if (self.kind = PseudostateKind::joinState) then
4       self.incoming -> forAll (t : Transition | t.guard.size = 0)
5     endif
```

Restricción 30: Un pseudo-estado de tipo `forkState` sólo tiene una transición entrante. Además, todas las transiciones salientes tienen que alcanzar estados contenidos en regiones ortogonales contenidas en un mismo macro-estado.En el apartado 6.5.4 se describe este pseudo-estado en detalle.

```
1 context Pseudostate
2   inv forkStateWith1IncomingTransition:
3     if (self.kind = PseudostateKind::forkState) then
4       self.incoming.size = 1
5     endif
6   inv forkTransitionsGoToOrthogonalRegionsInSameMacroState:
7     if (self.kind = PseudostateKind::forkState) then
8       self.outgoing -> forAll (s1, s2 : Transition |
9         s1 <> s2 implies s1.owner <> s2.owner and s1.owner.owner = s2.owner.owner
10      )
11    endif
```

Restricción 31: Las transiciones que salen de un pseudo-estado `forkState` no tienen guarda. Sólo la transición de llegada puede tenerla en este caso.

```
1 context Pseudostate
2   inv noGuardInForkOutgoingTransitions:
3     if (self.kind = PseudostateKind::forkState) then
4       self.outgoing -> forAll (t : Transition | t.guard.size = 0)
5     endif
```

A.3 VISTA ALGORÍTMICA

ESTA sección recoge las restricciones OCL que están relacionadas con la vista de comportamiento. Tal y como se explicó en la sección 6.5, esta vista está basada en los diagramas de actividades definidos por UML. La mayor parte de las restricciones que aquí se recogen están relacionadas con la EClass `Component` (que representa una «instancia») y sus numerosas relaciones y atributos, aunque también es necesario asegurar la corrección de la definición de interfaces y servicios así comprobar que las conexiones entre puertos son correctas. Todas estas restricciones están recogidas en el siguiente enumerado.

Restricción 32: Una pseudo-actividad de tipo `final` no tiene arcos de salida. Todos los arcos que le llegan son de tipo `ControlEdge`. Consular el apartado 6.6.4.

```
1  context PseudoActivity
2    inv noOutgoingEdgeFromFinalPseudoActivity:
3      if (self.kind = PseudoActivityKind :: final) then
4        self.outgoing.size = 0
5      endif
6    inv outgoingEdgeFromInitialPseudoActivityOfTypeControlEdge:
7      if (self.kind = PseudoActivityKind :: final) then
8        self.incoming -> forAll (e : ActivityEdge | e.oclIsTypeOf(ControlEdge))
9      endif
```

Restricción 33: Una pseudo-actividad de tipo `fork` sólo es enlazada por arcos de tipo dato (`DataEdge`). Además, sólo tiene un arco entrante y tiene que tener más de uno saliente.

```
1  context PseudoActivity
2    inv forkUsesOnlyDataEdge:
3      if (self.kind = PseudoActivityKind :: fork) then
4        self.incoming -> forAll (i : ActivityEdge | i.oclIsTypeOf(DataEdge)) and
5        self.outgoing -> forAll (i : ActivityEdge | i.oclIsTypeOf(DataEdge))
6      endif
7    inv forkHasOneIncomingEdge:
8      if (self.kind = PseudoActivityKind :: fork) then
9        self.incoming.size = 1
10     endif
11   inv forkHasManyOutgoingEdges:
12     if (self.kind = PseudoActivityKind :: fork) then
13       self.outgoing.size > 1
14     endif
```

Restricción 34: Las actividades referenciadas por un `LoopActivity` tienen que estar contenidas en la propia actividad compleja.

```
1  context LoopActivity
2    inv initContainedInLoopAct:
3      self.activity ->exists (a : Activity | a = self.init)
4    inv bodyContainedInLoopAct:
5      self.activity ->exists (a : Activity | a = self.body)
6    inv testContainedInLoopAct:
7      self.activity ->exists (a : Activity | a = self.test)
```

Restricción 35: En una pseudo-actividad de tipo `fork`, los arcos de entrada tienen que provenir de `ActivityParameter` de tipo salida, mientras que los arcos de salida tienen que llegar a parámetros de tipo entrada.

```
1  context PseudoActivity
2    inv forkIncomingEdgeOfTypeOut:
3      if (self.kind = PseudoActivityKind::fork) then
4        (self.incoming ->first().source.direction = ParameterKind::out) or
5        (self.incoming ->first().source.direction = ParameterKind::inOut)
6      endif
7    inv forkOutgoingEdgesOfTypeIn:
8      if (self.kind = PseudoActivityKind::fork) then
9        self.outgoing ->forAll(e : DataEdge |
10         (e.target.direction = ParameterKind::in) or
11         (e.target.direction = ParameterKind::inOut)
12       )
13     endif
```

Restricción 36: Una pseudo-actividad de tipo `initial` no tiene arcos de entrada y tiene sólo un arco de salida, de tipo `ControlEdge`.

```
1  context PseudoActivity
2    inv noIncomingEdgeToInitialPseudoActivity:
3      if (self.kind = PseudoActivityKind::initial) then
4        self.incoming.size = 0
5      endif
6    inv onlyOneOutgoingEdgeFromInitialPseudoActivity:
7      if (self.kind = PseudoActivityKind::initial) then
8        self.outgoing.size = 1
9      endif
10   inv outgoingEdgeFromInitialPseudoActivityOfTypeControlEdge:
11     if (self.kind = PseudoActivityKind::initial) then
12       self.outgoing ->first(). oclIsTypeOf(ControlEdge)
13     endif
```

Restricción 37: Las actividades referenciadas por un `ConditionalActivity` tienen que estar contenidas en la propia actividad compleja.

```
1 context ConditionalActivity
2   inv bodyContainedInConditionalAct:
3     self.activity ->exists (a : Activity | a = self.body)
4   inv testContainedInConditionalAct:
5     self.activity ->exists (a : Activity | a = self.test)
```

Restricción 38: Las actividades que forman una `ConditionalActivity` no pueden ser pseudo-actividades. Consultar apartado 6.6.3.

```
1 context ConditionalActivity
2   inv noPseudoActivityBody:
3     if (self.body.size = 1) then
4       not self.body.oclIsTypeOf (PseudoActivity)
5     endif
6   inv noPseudoActivityTest:
7     if (self.test.size = 1) then
8       not self.test.oclIsTypeOf (PseudoActivity)
9     endif
```

Restricción 39: Las pseudo-actividades no tienen parámetros (`ActivityParameter`) ni de entrada ni de salida. Más información en el apartado 6.6.4.

```
1 context PseudoActivity
2   inv noActivityParameters:
3     self.parameter.size = 0
```

Restricción 40: Las actividades que forman un bucle (EClass `LoopActivity`) no pueden ser pseudo-actividades. Ver apartado 6.6.3.

```
1 context LoopActivity
2   inv noPseudoActivityInit:
3     if (self.init.size = 1) then
4       not self.init.oclIsTypeOf (PseudoActivity)
5     endif
6   inv noPseudoActivityBody:
7     if (self.body.size = 1) then
8       not self.body.oclIsTypeOf (PseudoActivity)
9     endif
10  inv noPseudoActivityTest:
11    if (self.test.size = 1) then
12      not self.test.oclIsTypeOf (PseudoActivity)
13    endif
```

Restricción 41: Todos los parámetros de una actividad compleja tienen que estar conectados. Consultar apartado 6.6.3 .

```
1  context ComplexActivity
2    inv allInputParametersConnected:
3      let tmp : Bag(ActivityParameter) = self.parameter -> select (p : ActivityParameter |
4        (p.direction = ParameterKind :: in) or (p.direction = ParameterKind :: inOut)
5      ) in
6      tmp -> forAll (p : ActivityParameter | self.outgoing.size = 1)
7    inv allOutputParametersConnected:
8      let tmp : Bag(ActivityParameter) = self.parameter -> select (p : ActivityParameter |
9        (p.direction = ParameterKind :: out) or (p.direction = ParameterKind :: inOut)
10     ) in
11     tmp -> forAll (p : ActivityParameter | self.incoming.size = 1)
```

Restricción 42: Todas las actividades de una actividad compleja, salvo una pseudo-activity de tipo `fork`, tienen que estar conectadas mediante un `ControlEdge` que marque la secuencia de ejecución. Consultar apartado 6.6.3.

```
1  context ComplexActivity
2    inv allActivitiesConnectedByControlEdges:
3      let tmp : Bag(Activity) = self.activity -> reject (a : Activity |
4        (a.oclIsTypeOf(PseudoActivity)) and (a.kind = PseudoActivityKind :: fork)
5      ) in
6      tmp -> forAll (a : Activity | (a.incoming.size <> 0) or (a.outgoing.size <>0))
```

Restricción 43: Cuando se invoca un servicio requerido de un componente se tiene que asegurar que los parámetros de la actividad coinciden con los parámetros del servicio, y que todos los parámetros se relacionan entre sí uno a uno únicamente. Más información en el apartado 6.6.2.

```
1  context ServiceCall
2    inv sameNumberOfLinksAndParameters:
3      self.edge.size = self.parameter.size
4    inv actParameterCannotBeLinkedMoreThanOnce:
5      self.edge -> forAll (i1, i2 : ParameterLink |
6        i1 <> i2 implies i1.actParameter <> i2.actParameter
7      )
8    inv servParameterCannotBeLinkedMoreThanOnce:
9      self.edge -> forAll (i1, i2 : ParameterLink |
10       i1 <> i2 implies i1.serviceParameter <> i2.serviceParameter
11     )
```

Restricción 44: Los parámetros de una actividad de invocación de una función externa (`LibraryCall`) tienen que coincidir con los pines (`Pin`) de la función invocada. Todos los parámetros se relacionan con uno y sólo con un pin. Ver apartado 6.6.2.

```
1  context LibraryCall
2    inv sameNumberOfLinksAndPins:
3      self.pin.size = self.link.size
4    inv pinCannotBeLinkedMoreThanOnce:
5      self.link -> forAll (i1, i2 : PinParamLink |
6        i1 <> i2 implies i1.pin <> i2.pin
7      )
8    inv actParameterCannotBeLinkedMoreThanOnce:
9      self.link -> forAll (i1, i2 : PinParamLink |
10       i1 <> i2 implies i1.actParameter <> i2.actParameter
11     )
```

Restricción 45: El tipo de dato al que hace referencia el pin de una acción y el parámetro de una actividad enlazados por el mismo `PinParamLink` tiene que ser el mismo. Consultar el apartado 6.6.5 sobre variabilidad en la implementación.

```
1  context PinParamLink
2    inv thePinAndTheActivityParameterHaveSameType:
3      (self.pin.type = self.actParameter.type) and (self.pin.kind = self.actParameter.kind)
```

Restricción 46: El tipo de dato al que hace referencia el parámetro de una actividad y el parámetro de un servicio enlazados por el mismo `ParameterLink` tiene que ser el mismo. Consultar apartado 6.6.2.

```
1  context ParameterLink
2    inv theActivityParameterAndServiceParameterHaveSameType:
3      (self.serviceParameter.type = self.actParameter.type) and
4      (self.serviceParameter.kind = self.actParameter.kind)
```

Restricción 47: Una actividad constante (`ConstantActivity`) no tiene arcos de entrada, ya que sólo suministra valores a la salida. Además, tiene un único parámetro, cuyo tipo tiene que ser de salida.

```
1  context ConstantActivity
2    inv noIncomingEdgesToConstantActivity:
3      self.incoming.size = 0
4    inv oneAndOnlyOneParameterPerConstantActivity:
5      self.parameter.size = 1
6    inv constantActivityParameterMustBeAnOutputParameter:
7      self.parameter -> first(). direction = ParameterKind :: out
```

Restricción 48: Los `ActivityParameter` conectados por un `DataEdge` tienen que tener el mismo tipo de dato. Además, el parámetro tiene que estar marcado como de salida en el origen y de entrada en el destino. Más información sobre enlazado de actividades en el apartado 6.6.3.

```
1 context DataEdge
2   inv parametersMustHaveSameDataType:
3     self.source.type = self.target.type
4   inv sourceParameterMustHaveOutDirection:
5     (self.source.direction = ParameterKind::in) or
6     (self.source.direction = ParameterKind::inOut)
7   inv targetParameterMustHaveInDirection:
8     (self.target.direction = ParameterKind::out) or
9     (self.target.direction = ParameterKind::inOut)
```

Restricción 49: La actividad enlazada por un `ActivityBinding` no puede ser de tipo `PseudoActivity`.

```
1 context ActivityBinding
2   inv activityBindingCannotPointToPseudoActivity:
3     not self.to.oclIsTypeOf(PseudoActivity)
```

APÉNDICE B

UNA VISIÓN DETALLADA DE LOS DIAGRAMAS DE ACTIVIDAD

LA GRAN cantidad de diagramas de actividades que se generan en el desarrollo de la transformación de modelos entre V^{3}Studio y UML ha propiciado la creación de este apéndice. El objetivo de este apéndice es describir algunos detalles del diagrama de actividades UML que, debido a la profusión de clases que tiene este diagrama en UML y al poco uso que se hace de él normalmente, puedan resultar confusos o incluso desconocidos para el lector mientras lee el capítulo 6.

UML es un lenguaje de modelado que permite describir distintas vistas del software que tienen distinto nivel de abstracción, e.g. desde los posibles escenarios en que se va a utilizar el programa (casos de uso) hasta el detalle fino de implementación de un algoritmo (actividades y acciones). Sin embargo, y en todos los casos, UML asegura la unicidad de todos las clases que se utilicen, independientemente del diagrama en que aparezcan o del nivel de abstracción que tengan, y que existe una o más formas de navegar entre estas clases, ya que todas están, en cierta forma, relacionadas entre sí, ya que describen la misma realidad: una misma aplicación software.

Es justamente esta información la que no aparece de forma explícita en la mayoría de libros sobre UML. En ellos se enseña cómo realizar los diagramas, pero no se explica realmente cómo se utilizan las clases de UML que subyacen al diagrama ni las relaciones que existen entre ellas.

B.1 ACTIVIDADES Y ACCIONES EN UML

EL MODELADO del comportamiento mediante un diagrama de actividades ha sufrido una profunda modificación entre las versiones 1.5 y 2.0 de UML. Estos diagramas han pasado de ser una mera representación alternativa a las máquinas de estados a tener su propia semántica, ya que actualmente están basados en redes de Petri y en el paso de *tokens* entre actividades. Como ya se describió en el apartado 6.6, las actividades que se generan en el transcurso de la transformación de modelos se mantienen fieles a la nueva semántica del diagrama de actividididades de cara a que los modelos generados puedan ser fácilmente utilizados en herramientas que trabajan con redes de Petri, como simuladores o demostradores de propiedades.

Los diagramas de actividades en UML están formados por dos tipos de clases: *acciones* y *actividades*. Una acción es «la unidad fundamental de especificación de comportamiento. Una acción toma un conjunto de entrada y lo convierte en un conjunto de salida, aunque cualquiera o incluso los dos conjuntos pueden estar vacíos». Las actividades, por el contrario «enfatizan la secuencia y las condiciones para coordinar el comportamiento de bajo nivel, en vez de clasificarlo. Estas secuencias son normalmente conocidas como flujo de control y flujo de datos. Las acciones que coordina una actividad pueden ser iniciadas debido a que otras acciones han terminado su ejecución, a que un objeto o un dato es accesible o porque ha ocurrido un evento externo al flujo». Se puede concluir, por tanto, que las actividades son agrupaciones de acciones (y, posiblemente, de otras actividades).

Los diagramas de actividades de UML hacen distinción entre flujo de control, que guía la ejecución secuencial de las acciones, y el flujo de datos, que guía las relaciones entre los datos que se intercambian acciones y actividades. Tradicionalmente, los antecesores de los diagramas de actividad, los *workflow*, sólo mostraban el flujo de control entre las distintas actividades que formaban parte del diagrama, mientras que el flujo de datos entre ellos se relegaba a un segundo plano. A menudo, era labor de la persona que interpretaba el workflow deducir qué datos se intercambiaban, entre qué actividades y cómo. UML 2.0 ha contribuido definitivamente a establecer la necesidad de definir explícitamente ambos flujos, aunque esto suponga enmarañar el diagrama de actividades.

Como sucede en la mayoría de los diagramas UML, también en el caso del diseño de un diagrama de actividades existen diversas formas equivalentes de describir un mismo comportamiento. En este caso las posibilidades son aún mayores, ya que UML define más de 20 tipos de acciones, los elementos fundamentales que forman una actividad y más de 10 tipos de actividades, que pueden contener acciones u otras actividades y que se pueden conectar entre sí de diversas formas.

There are potentially many ways of implementing the same specification, and any implementation that preserves the information content and behavior of the specification is acceptable. Because the implementation can have a different structure from that of the specification, there is a mapping between the specification and its implementation. This mapping need not be one-to-one [...]

— UML 2.0 Superstructure [177]

B.2 DESCRIPCIÓN DE LAS CLASES UTILIZADAS

EN ESTA sección se describen todas las clases relacionadas con la descripción de acciones y actividades que se han utilizado para realizar la transformación entre los modelos de V^{3}Studio y UML. Esta clases no tienen que se confundidas, en ningún caso, con las clases que forman la vista algorítmica de V^{3}Studio (consultar apartado 6.6). Esta sección describe, en apartados separados, las clases que se han utilizado de paquete de actividades de UML y las que están definidas en el paquete acciones. Las figuras B.1 y B.2 muestran, respectivamente, sendos extractos del meta-modelo de UML para facilitar la comprensión de los detalles que se exponen en los siguientes apartados.

Para facilitar la identificación de estas clases en los diagramas de actividad que se muestran a lo largo del capítulo 6 se ha utilizado un abreviatura de la clase (mostrada entre paréntesis) en caso de que la representación gráfica resulte confusa. Esto sucede, principalmente, con las clases que representan acciones (consultar el apartado B.2.2).

B.2.1 CLASES DEL PAQUETE ACTIVIDADES

`ActivityParameterNode`: modela un parámetro de entrada o salida a una actividad. Esta clase tiene un atributo para especificar el tipo del parámetro, mientras que la dirección se obtiene indirectamente a partir de la siguiente restricción: «un `ActivityParameterNode` no puede tener arcos de entrada y salida a la vez». Se representa gráficamente como un cuadrado sobre el borde que define la actividad.

`ControlFlow`: modela el flujo de control entre dos `ActivityNode`. Como puede verse en la figura B.1, UML establece que un `ControlFlow` no puede conectarse con ningún `ObjectNode`.

`ObjectFlow`: modela el flujo de datos entre dos `ActivityNode`. UML establece que un `ObjectFlow` sólo puede conectarse con `ObjectNode`, ya que son los únicos que permiten el flujo de datos a su través.

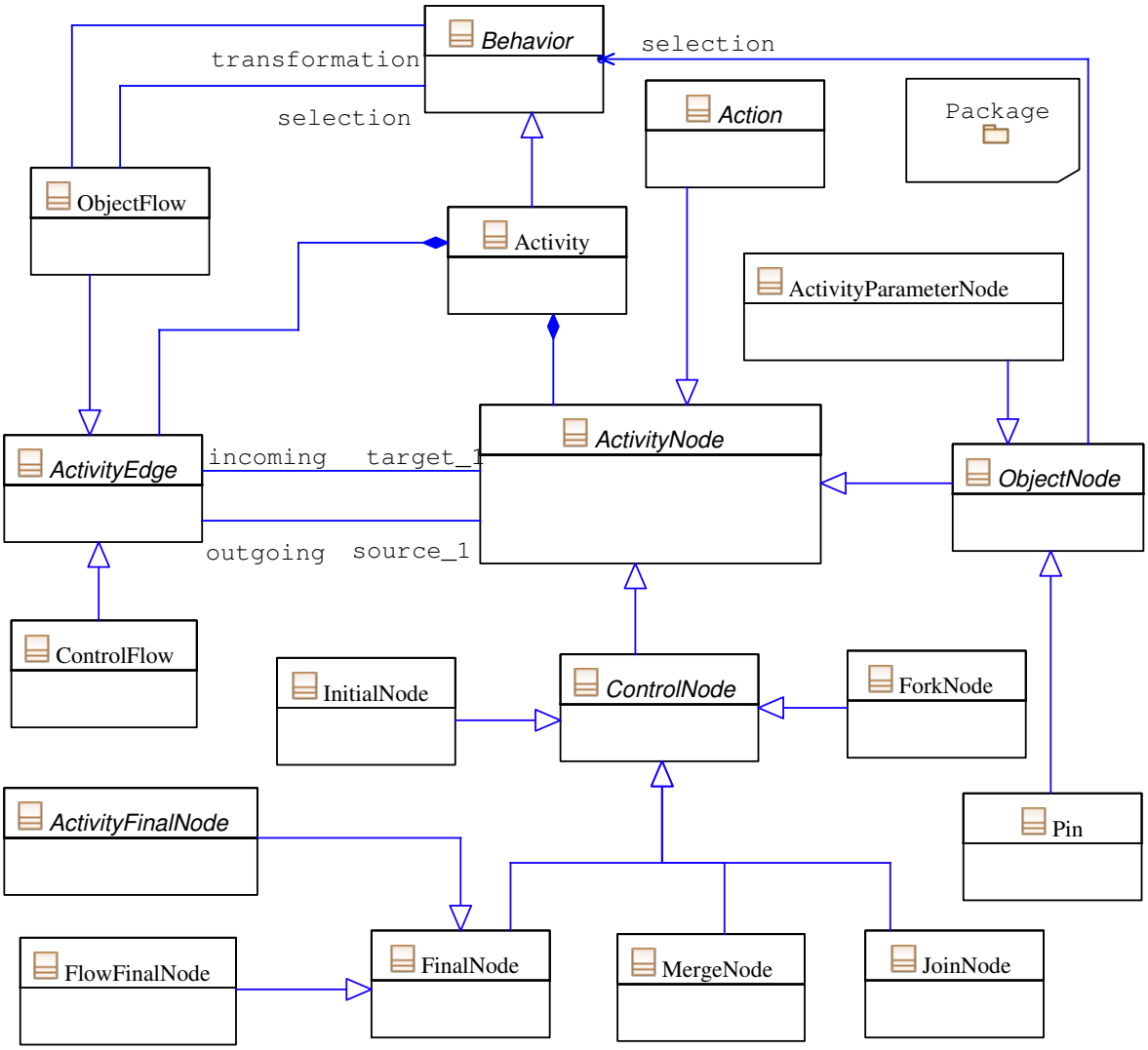


Figura B.1: Meta-modelo simplificado del paquete de actividades de UML (extraído de [177])

`ForkNode`: es un `ControlNode` que divide un flujo (de control o datos) en múltiples flujos concurrentes. Un `ForkNode` tiene un único arco de entrada y múltiples flujos de salida. Se representa gráficamente como una línea vertical corta y de trazo grueso.

`InitialNode`: es un `ControlNode` que marca el comienzo del flujo cuando una actividad es invocada. Un `InitialNode` no tiene arcos de entrada, y los arcos de salida son de tipo `ControlFlow`. Curiosamente, UML establece que un actividad puede tener más de un nodo de este tipo. Se representa gráficamente como un círculo de poco radio rellenado de negro.

`ActivityFinalNode`: es un `FinalNode` que marca el final de todos los flujos de una actividad. Se representa gráficamente como dos círculos concéntricos, estándo relleno de negro el círculo interior.

B.2.2 CLASES DEL PAQUETE ACCIONES

`InputPin`: son `ObjectNode` que proporcionan a la acción que los contiene el *token* que reciben de otros `ObjectNode` a través del `ObjectFlow` que los une. La representación gráfica de todos los pines es la misma: un pequeño cuadrado en el borde de la acción en la que están contenidos. En los diagramas de actividades que se presentan a lo largo del capítulo 6 no se distingue entre los distintos tipos de pines (entrada o salida). Sólo el `ActionInputPin` y el `ValuePin` aparecen marcados, respectivamente, en rojo y azul, para resaltar que no son pines comunes.

`ValuePin`: es un `InputPin` que proporciona un token de valor fijo. Este tipo de pin aparece representado como un pequeño cuadrado de color azul en el borde del cuadrado que representa la acción que lo contiene.

son `ObjectNode` que proporcionan el *token* que produce la acción que los contiene a otros `ObjectNode`, con los que se conecta a través del `ObjectFlow` que los une.

`OutputPin`: son `ObjectNode` que proporcionan el *token* que produce la acción que los contiene a otros `ObjectNode`, con los que se conecta a través del `ObjectFlow` que los une.

`ActionInputPin`: es un tipo de `InputPin` que ejecuta una acción para determinar el valor que se va a suministrar a la acción que lo contiene. Como no existe representación gráfica para este tipo de pin, en los diagramas se muestra como un cuadrado pequeño relleno en rojo. La acción que sirve de origen al `ObjectFlow` que realiza la conexión es la que está realmente contenida en el `ActionInputPin` en el modelo UML.

`CreateObjectAction` (CObA): es una acción que crea un objeto del tipo que se especifica en uno de sus atributos. Este objeto (*token*) se ofrece en un pin de salida denominado `result`.

`AddStructuralFeatureValueAction` (ASFVA): es una acción que sirve para modificar el valor de una característica o propiedad del *token* que se pasa a la acción a través del pin denominado `objectPin`. La propiedad que se debe modificar en el token de entrada es establecida a través de un atributo del `AddStructuralFeatureValueAction`, mientras que el valor de esta propiedad se establece mediante el resto de pines de entrada que pudiera contener la acción.

`CallOperationAction` (COpA): es una acción que invoca una operación en el *token* que se le pasa en el pin denominado `object`. Los parámetros de entrada para invocar esta operación se encuentran en los pines de entrada a la acción, mientras que el resultado de la operación se deposita en los pines de salida de la acción.

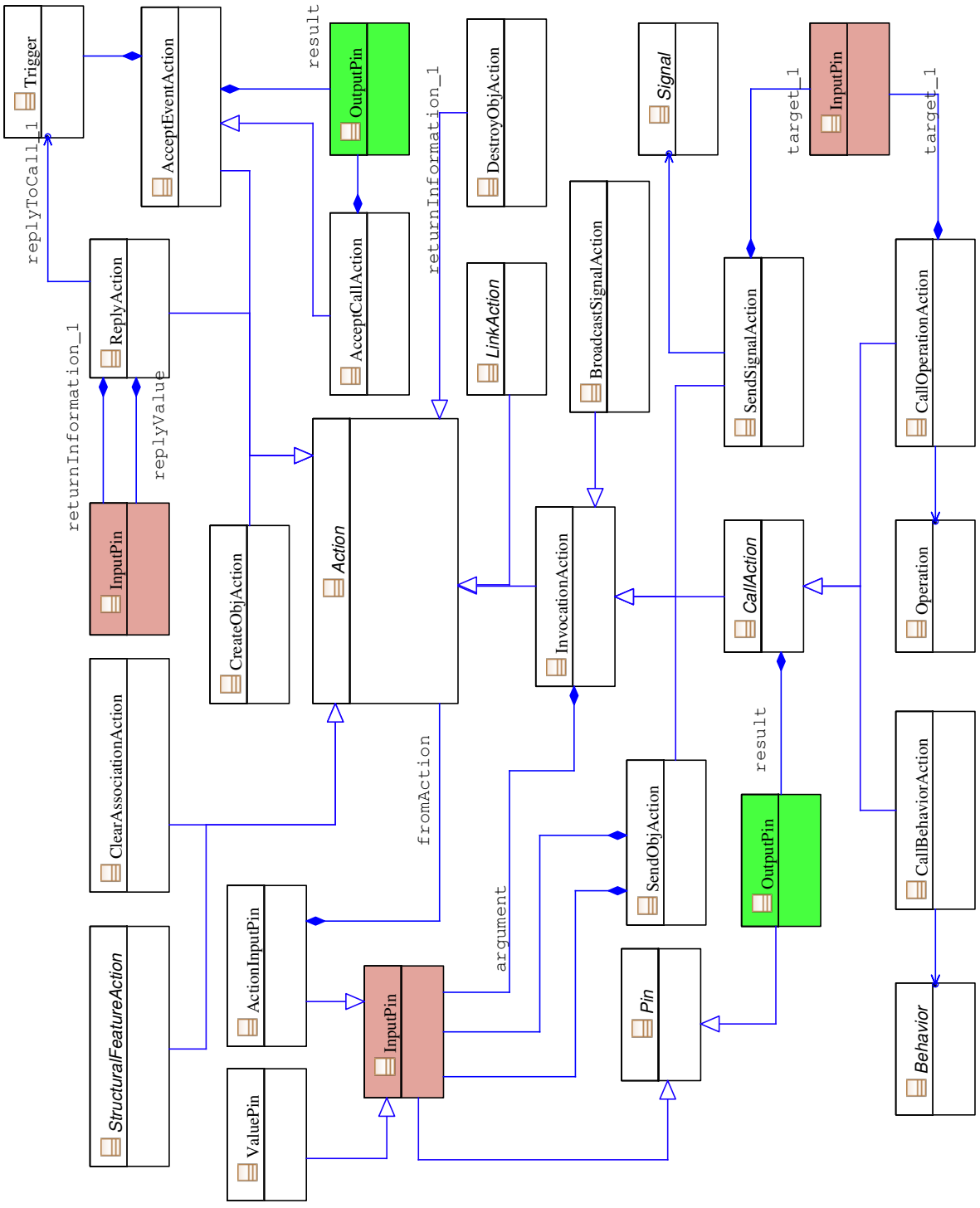


Figura B.2: Meta-modelo simplificado del paquete de acciones de UML (extraído de [177]). Las clases con el mismo color representan la misma clase, que se ha repetido para evitar que se entrecrucen demasiadas líneas.

`ReclassifyObjectAction` (ROA): es una acción que cambia el clasificador que clasifica el token que fluye en la acción a través del pin de entrada denominado `object`.

`ReadStructuralFeatureAction` (RSFA): esta acción lee una propiedad del *token* que se le pasa en el pin de entrada `object` y genera un *token* en el pin de salida con su valor.

`SendSignalAction` (SSA): es una acción que crea un señal de un tipo determinado y lo envía al objeto (*token*) que se le pasa a través del pin de entrada denominado `targetObject`. La representación gráfica es la misma que tiene en UML: un pentágono no regular.

`ReadSelfAction` (RSA): es una acción que coloca el objeto que ejecuta esta operación como un *token* en el pin de salida denominado `object`.

Todas las acciones cuya representación gráfica no haya sido descrita son representadas mediante un cuadrado de bordes redondeados. Para evitar la confusión que puede provocar el uso de la misma representación gráfica, el nombre de la acción utiliza una de las siglas descritas en este apartado.

B.3 FORKNODE CONTRA ACTIONINPUTPIN

COMO sucede en la mayoría de los diagramas UML, también existen diversas formas equivalentes de describir una misma actividad. En este caso, es posible diseñar diagramas de actividades semánticamente equivalentes pero utilizando distintos elementos. Concretamente, en el desarrollo de la transformación se detectó que era posible utilizar tanto un `ForkNode` como un `ActionInputPin` cuando se requería que el resultado de una acción se utilizara como parámetro de entrada de dos acciones distintas. Teniendo en cuenta la semántica de consumo de *tokens* asociada a las redes de Petri, la solución con los dos tipos de nodos difiere ligeramente (aunque el resultado es el mismo):

`ForkNode`: este nodo permite clonar el *token* que recibe como entrada y enviarlo a todas las acciones con las que se conecta. Su uso es imprescindible si se quiere que los diagramas de actividades se ajusten al modelo de las redes de Petri, de forma que puedan ser posteriormente analizados por cualquier herramienta basada en este formalismo.

`ActionInputPin`: este pin es distinto a los otros dos tipo (`InputPin` y `OutputPin`) en el sentido de que no es pasivo, por lo que puede romper el flujo de datos del diagrama

de actividad. En este caso, el token cuyo valor se quiere enviar a varias acciones no se clona, sino que se recalcula cada vez.

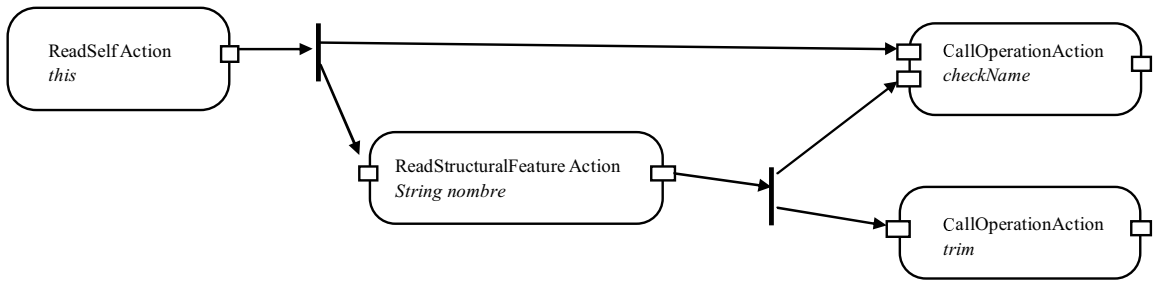
La ventaja de utilizar el primero de ellos es que los diagramas son más fáciles de entender y mantener, ya que el flujo de la actividad queda expresado de forma clara. La ventaja del segundo es que requiere el uso de menos elementos, con lo que se simplifica la transformación de modelos. Sin embargo, un `ActionInputPin` no puede ser representado gráficamente en un diagrama de actividades y, además, la lógica de su ejecución no permite reflejar de forma explicita el flujo de control en el diagrama. Además, la acción de la que obtiene el *token* el `ActionInputPin` sólo puede tener un pin de salida. La figura B.3 muestra un pequeño ejemplo comparativo, en el que aparece un trozo de código en java y su traducción a dos diagramas de actividades que utilizan, respectivamente, `ForkNode` y `ActionInputPin`.

Finalmente, se decidió mezclar ambos modelos aunque predomina el uso del `ForkNode`, ya que consigue que el diagrama final sea más fácil de entender al dejar patente el flujo que guía la actividad. Los `ActionInputPin` se han utilizado únicamente y exclusivamente en las acciones de tipo `ReadStructuralFeatureAction` que actúan sobre el objeto *self*, ya que en este caso se consiguen crear menos elementos y el diagrama sigue siendo perfectamente inteligible.

Código Java

```
public class prueba {
        String nombre;
        public void checkName (String arg) {
                // … (no mostrado)
        }
        public void operation () {
                this.checkName(this.nombre);
                this.nombre.trim();
        }
}
```

Uso de Fork



Uso de ActionInputPin

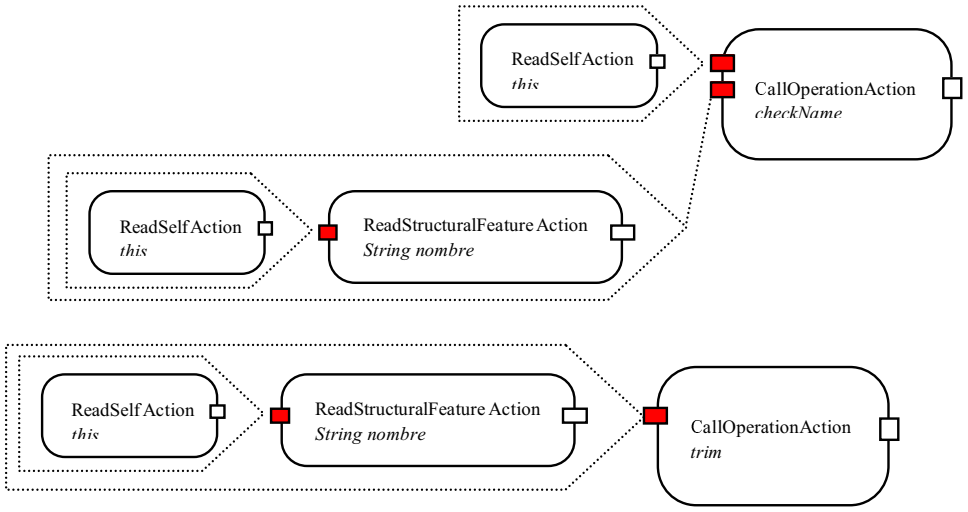


Figura B.3: Ejemplo comparativo entre el uso de `ForkNode` y `ActionInputPin` (en rojo en el diagrama)

GLOSARIO DE ACRÓNIMOS

A

Architecture Description Language (ADL) Lenguaje de descripción arquitectónica. Los ADLs florecieron en la década de los noventa con el auge de la investigación en arquitectura software. Un ADL se centra en la estructura de alto nivel global de la aplicación en vez de en los detalles de implementación de cualquier módulo de código.

ATLAS Transformation Language (ATL) Lenguaje de transformación del grupo ATLAS. Desarrollado en el año 2.004 por el grupo de investigación ATLAS del INRIA francés, ATL es uno de los primeros lenguajes de transformación de modelos que se han desarrollado. ATL es un lenguaje declarativo/imperativo, que permite especificar cómo se transforman los elementos de un meta-modelo origen en uno o varios elementos de un meta-modelo destino. ATL permite transformaciones con múltiples modelos de entrada y múltiples modelos de salida.

C

Commercial Off-The-Shelf (COTS) Componente comercial, literalmente «tomado de la estantería». COTS es un adjetivo que se aplica en el mundo del software y del hardware, generalmente a productos tecnológicos o informáticos, que están listos para ser vendidos o alquilados a posibles clientes.

Common Warehouse Metamodel (CWM) CWM es una especificación del OMG para modelar los meta-datos, ya sean relacionales, no relacionales o multi-dimensionales de la mayoría de los datos que almacena una empresa. La última versión de CWM es del año 2.004.

Component-Based Development (CBD) Desarrollo basado en componente. Paradigma de desarrollo software que utiliza componentes, como unidades autocontenidas y reemplazables en un contexto, como unidad fundamental de diseño y construcción de aplicaciones. Este término genérico engloba las dos visiones particulares: CBSE y COSE.

Component-Based Software Engineering (CBSE) Ingeniería software basada en componentes. Esta rama de desarrollo software basado en components CBD centra su atención en el

desarrollo de los componentes propiamente dichos y en el de la infraestructura, tanto de ejecución como de creación, que necesitan. El desarrollo de modelos de componentes y frameworks centran el ámbito de especificación de CBSE.

Component-Oriented Software Engineering (COSE) Ingeniería software orientada a componentes. Esta rama de desarrollo software basado en components CBD centra su atención en el desarrollo de aplicaciones mediante el ensamblado de componentes ya existentes. La elaboración de catálogos de componentes, selección de los componentes adecuados y la evaluación del comportamiento global de la aplicación son algunas de las principales áreas de investigación de COSE.

Computation Independent Model (CIM) Modelo independiente de computación. En el contexto definido por MDA, un CIM representa el modelo de mayor nivel de abstracción, que contiene únicamente conceptos y datos del dominio de aplicación. Un CIM sitúa y relaciona al sistema con el entorno en que opera y permite no sólo ayudar a entender la problemática sino que también proporciona el vocabulario que van a utilizar el resto de familias de modelos MDA.

D

Domain-Specific Language (DSL) Los lenguajes específicos de dominio son lenguajes de programación o especificaciones ejecutables que ofrecen, gracias a una notación y a un nivel de abstracción adecuados, un gran poder expresivo centrado, y generalmente limitado, a un dominio de aplicación concreto. Los DSLs han sido utilizados desde hace mucho tiempo, si bien su uso y desarrollo ha experimentado un tremendo auge gracias a MDE.

E

Eclipse Modelling Framework (EMF) Marco de modelado de Eclipse. EMF es un plug-in del entorno de desarrollo Eclipse que proporciona una implementación del estándar del OMG MOF, concretamente de EMOF. Como sucede con MOF en el marco MDA, EMF es el plug-in básico de soporte del enfoque MDE dentro de la plataforma Eclipse, de la que dependen el resto de herramientas MDE. EMF proporciona no sólo la implementación de EMOF, sino también código para guardar y cargar modelos y meta-modelos en repositorios, así como generadores de código para traducir los meta-modelos a un conjunto de clases Java.

Essential-MOF (EMOF) MOF esencial. La especificación de MOF establece la existencia de dos niveles de compatibilidad con MOF: EMOF y CMOF. EMOF representa el nivel básico, que contiene los elementos necesarios para facilitar la descripción de meta-modelos y el intercambio de estos meta-modelos entre distintas herramientas.

G

Generative Programming (GP) Programación generativa. Enunciada por Czarnecki en el año 2.000, la programación generativa es un paradigma que permite la creación bajo demanda y de forma personalizada de un producto intermedio o final altamente optimizado a partir de componentes elementales y reutilizables.

K

Kernel Metamodelling (Kermeta) Kermeta es un lenguaje de modelado y programación conforme a la especificación EMOF del OMG. Kermeta es un lenguaje desarrollado por grupo de investigación Triskell del IRISA francés en el año 2.005. Kermeta ha sido diseñado para escribir programas que también son modelos, para describir transformaciones de modelos (que también son programas), para añadir restricciones a estos modelos y para ejecutar los modelos (que son programas).

M

Meta-Object Facility (MOF) MOF es un estándar desarrollado por el OMG, cuyo objetivo inicial era proporcionar un meta-modelo para describir UML y servir como pieza central de la iniciativa MDA. En la arquitectura en cuatro capas sintácticas definida por el OMG, MOF ocupa el nivel M3, el más alto de todos. El resto de lenguajes y meta-modelos definidos en distintos estándares del OMG (como UML, SPEM, CWM, etc.) son definidos utilizando MOF. MOF es la herramienta central primordial tanto en del enfoque MDA como del más general MDE. La última versión de estándar disponible en estos momentos es del año 2.004.

Model-Driven Architecture (MDA) Arquitectura basada en modelos. Visión particular del OMG del desarrollo basado en modelos (MDE). La primera versión de MDA fue liberada por el OMG en el año 2.001. MDA se basa en una serie de estándares que también han sido definidos por el propio OMG, como son MOF, OCL, QVT, UML, etc. El diseño siguiendo un proceso MDA comienza con el desarrollo de un modelo que contiene únicamente información del proceso de negocio (CIM), para luego obtener otro modelo más detallado pero independiente de la plataforma de ejecución (PIM), que finalmente es completado en un tercer modelo que contiene los datos necesarios para realizar una traducción a código (PSM).

Model-Driven Development (MDD) Desarrollo guíado por modelos. Término registrado por el OMG para referirse al proceso de desarrollo utilizando modelos MDA.

Model-Driven Engineering (MDE) Desarrollo basado en modelos. Enfoque de desarrollo software basado en el uso sistemático de modelos, como representación simplificada de la realidad, a lo largo de todas las etapas del diseño software. De esta forma se aumenta el nivel de abstracción con que se realiza el diseño. MDE promete superar las limitaciones

de los lenguajes de programación para expresar los conceptos del dominio de forma efectiva.

Model-To-Model (M2M) Transformación modelo-a-modelo. Dentro del enfoque MDE, estas transformaciones se utilizan con varios fines: refinamiento (aumenta el nivel de detalle de un modelo), entretejido (distintos modelos que representan distintas vistas se funden en uno solo), concreción (añade detalles de la plataforma de ejecución), etc.

Model-To-Text (M2T) Transformación modelo-a-texto. Estas transformaciones son las últimas que se realizan cuando se sigue un desarrollo MDE, ya que no producen otro modelo, sino una representación textual. Las transformaciones a texto se utilizan principalmente para generar código en un lenguaje de programación concreto, aunque es posible generar cualquier otro tipo de representación textual.

O

Object Constraint Language (OCL) Lenguaje de especificación de restricciones sobre objetos. OCL completa el enfoque MDA proporcionando un lenguaje mixto (declarativo/imperativo) para especificar restricciones adicionales sobre los elementos de un modelo. OCL es un lenguaje potente que permite detectar condiciones permitidas por el metamodelo que generan modelos correctos sintácticamente pero no semánticamente. La última versión de este estándar es del año 2.006.

Object Management Group (OMG) Consorcio de empresas dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización no lucrativa que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para tecnologías orientadas a objetos. El grupo está formado por diversas compañías y organizaciones de software de reconocido prestigio internacional, como IBM, Sun Microsystems o HP.

P

Platform-Independant Model (PIM) Modelo independiente de plataforma. Dentro del contexto MDA, un modelo PIM ocupa el nivel de abstracción intermedio, entre el CIM y un PSM. Un modelo PIM proporciona una vista más detallada del problema que el modelo CIM en que se basa. No contiene elementos específicos de ninguna plataforma o tecnología concreta, aunque sí contiene elementos genéricos que no están presentes en el CIM, e.g. puede hacer referencia a una base de datos. Un modelo único modelo CIM puede transformarse en distintos modelos PIM, y pueden existir diversos modelos PIM con distintos niveles de detalle, aunque siempre independientes de plataforma y tecnología de implementación.

Platform-Specific Model (PSM) Modelo específico de plataforma. Un modelo PSM ocupa el último nivel de abstracción en el desarrollo MDA, por debajo del PSM. Un PSM es un modelo de un subsistema que incluye información sobre la tecnología específica

que se utiliza para la realización de ese subsistema en una plataforma específica. Un modelo PIM puede, a su vez, dar origen a distintos modelos PSM, e.g. traducciones a distintos lenguajes de programación. Además, pueden existir diversos modelos PSM que proporcionen cada vez más detalles de la plataforma de ejecución.

Q

Query, View and Transformation (QVT) Lenguaje de consulta, vista y transformación. QVT es un estándar definido por el OMG en el contexto de su inciativa MDA. QVT completa el enfoque MDA proporcionando un lenguaje mixto (declarativo/imperativo) para especificar cómo se realizan las transformaciones entre modelos, cómo se pueden realizar consultas a un modelo y cómo se puede generar vistas de los elementos que interesan al desarrollador. La última versión de este estándar es del año 2.005.

S

Software Enginnering Institute (SEI) Instituto de Ingeniería del Software. El SEI es un centro de investigación y desarrollo fundado en 1.984 con fondos federales en la Universidad Carnegie Mellon, en Estados Unidos, aunque tiene varias oficinas repartidas por el mundo. La misión principal del SEI es desarrollar proyectos financiados por el DoD (Department of Defense). Además, el SEI ha contribuido al desarrollo de la Ingeniería Software realizando diversos estudios en las ramas más punteras, como arquitectura software, proceso de desarrollo software, líneas de producto, interoperabilidad de sistemas, etc.

Software Factories (SF) Las factorías software son la visión particular de MDE de Microsoft. Una SF contiene todos los elementos de configuración necesarios para adaptar un entorno de desarrollo concreto para desarrollar un tipo de aplicación concreto. Una SF aglutina todas las herramientas de desarrollo basado en modelos, su configuración y la del entorno en un único fichero.

Software Process Engineering Metamodel (SPEM) Metamodelo para el proceso de ingeniería software. SPEM es un estándar del OMG cuyo objetivo es desarrollar un meta-modelo para describir un proceso concreto de desarrollo software o una familia de procesos de desarrollo relacionados. La última versión de este estándar es del año 2.005.

System Modelling Language (SysML) Lenguaje de modelado de sistemas. SysML es un estándar definido por el OMG para modelar sistemas ingenieriles. SysML pretende, de esta forma, subsanar uno de los mayores defectos de UML: UML permite describir únicamente sistemas software, dejando de lado el modelado del resto delementos hardware, también muy importantes. SysML ha sido desarrollado como un profile de UML. La última versión de este estándar es del año 2.006.

T

Technological Spaces (TS) Espacios tecnológicos. Un TS es un marco de trabajo con un conjunto de conceptos asociados, un cuerpo de conocimiento, herramientas, habilidades requeridas y posibilidades. Un TS está comúnmente asociado a una comunidad de usuarios, que comparten una determinada forma de realizar su trabajo, una literatura común y que celebran incluso reuniones y conferencias regularmente sobre el tema. Algunos TS fácilmente identificables son el TS de XML (intercambio de información), el TS de DBMS (almacenamiento de información), el TS de sintaxis abstracta (definición de lenguajes), el TS de meta-modelado (iniciativa MDA del OMG), etc.

U

Unified Modelling Language (UML) Lenguaje unificado de modelado. UML es un lenguaje gráfico estandarizado por el OMG para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir un «plano» del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables. La última versión es del año 2.007.

X

XML Metadata Interchange (XMI) Intercambio de metadatos en XML. XMI es un estándar del OMG para facilitar el intercambio de meta-datos basado en XML. XMI es utilizado por el resto de estándares del OMG para salvar e intercambiar sus modelos. Es el formato principal utilizado en el desarrollo basado por modelos. La última versión del estándar XMI fue liberada por el OMG en el año 2.005.

BIBLIOGRAFÍA

[1] *Ada Reference Manual ISO/IEC 8652:2007(E)*. International Standards Organization, 2006. Disponible en `http://www.adaic.org/standards/05rm/RM-Final.pdf`

[2] *Real-time Specification for Java (Final Release 3)*. Java Community Process, 2006. Disponible en `http://jcp.org/aboutJava/communityprocess/mrel/jsr001/index2.html`

[3] Abouzahra, A.; Bézivin, J.; Didonet, M. y Jouault, F.: «A Practical Approach to Bridging Domain Specific Languages with UML profiles». En: *Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development*, , 2005.

[4] Akyildiz, I.F.; Su, W.; Sankarasubramaniam, Y. y Cayirci, E.: «Wireless sensor networks: a survey». *Computer Networks*, 2002, **38(4)**, Elsevier. ISSN 1389-1286. doi: 10.1016/S1389-1286(01)00302-4.

[5] Albert, C. y Brownsword, L.: «Evolutionary Process for Integrating COTS-Based Systems (EPIC): An Overview». *Informe técnico CMU/SEI-2002-TR-009*, Software Engineering Institute (SEI), Carnegie Mellon University, 2002. Disponible en `http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr009.pdf`

[6] Albus, J.; Quintero, R. y Lumia, R: «An Overview of NASREM: The NASA/NBS Standard Reference Model for Telerobot Control System Architecture,». *Informe técnico 5412*, National Institute of Standards and Technology, 1994.

[7] Aldrich, Jonathan: *Using Types to Enforce Architectural Structure*. Tesis doctoral, University of Washington, 2003. Disponible en `http://archjava.fluid.cs.cmu.edu/papers/aldrich-dissertation.pdf`

[8] Alonso, A.; Álvarez, B.; Pastor, J.A.; de la Puente, J.A. y Iborra, A.: «Software Architecture for a Robot Teleoperation System». En: *Proceedings of the IV IFAC Workshop on Algorithms and Architectures for Real-Time Control*, Elsevier Science. ISBN 0080429300, 1997.

[9] Alonso, D.; Sánchez, P.; Álvarez, B. y Pastor, J.A.: «A systematic approach to developing safe tele-operated robots». En: 11^{th} *International Conference on Reliable Software Technologies*, volumen 4006 de *Lecture Notes on Computer Science*, pp. 119–130. ISBN 3-540-34663-5. ISSN 0302-9743, 2006.

[10] Alonso, D.; Vicente-Chicote, C.; Sánchez, P.; Álvarez, B. y Losilla, F.: «Automatic Ada Code Generation Using a Model-Driven Engineering Approach». En: 12^{th} *International Conference on Reliable Software Technologies*, volumen 4498 de *Lecture Notes on Computer Science*, pp. 168–179. ISSN 0302-9743, 2007.

[11] Andrade, L. F. y Fiadeiro, J. L.: «Interconnecting Objects via Contracts». En: *UML'99 - The Unified Modeling Language: Beyond the Standard, Second International Conference*, volumen 1723 de *Lecture Notes on Computer Science*, pp. 566–583. Springer-Verlag. ISSN 0302-9743, 1999.

[12] Andrade, L.F. y Fiadeiro, J.L.: «Architecture Based Evolution of Software Systems». En: *Formal Methods for Software Architectures: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003*, volumen 2804 de *Lecture Notes on Computer Science*, pp. 148–181. Springer-Verlag. ISSN 0302-9743, 2003. doi: 10.1007/b13225.

[13] Arbaoui, S.; Derniame, J.C.; Oquendo, F. y Verjus, H.: «A Comparative Review of Process-Centered Software Engineering Environments». *Ann. Softw. Eng.*, 2002, **14(1–4)**, pp. 311–340. ISSN 1022-7091.

[14] Arkin, R.: «Motor Schema — Based Mobile Robot Navigation». *International Journal of Robotics Research*, 1989, **8(4)**, SAGE publications. ISSN 0278-3649. doi: 10.1177/02783649890080040.

[15] Arkin, R.: *Behavior-Based Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 1998. ISBN 0262011654.

[16] Atkinson, C. y Kühne, T.: «Model-Driven Development: a Metamodelling Foundation». *IEEE Software*, 2003, **20(5)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2003.1231145.

[17] Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R. y Wallnau, K.: «Technical Concepts of Component-Based Software Engineering». *Informe técnico CMU/SEI-2000-TR-008*, Software Engineering Institute (SEI), Carnegie Mellon University, 2000. Disponible en `http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html`

[18] Bachmann, F.; Bass, L. y Klein, M.: «Preliminary Design of ArchE: A Software Architecture Design Assistant». *Informe técnico CMU/SEI-2003-TR-021*, Software Engineering Institute (SEI), Carnegie Mellon University, 2003. Disponible en `http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tr021.pdf`

[19] Bachmann, F.; Bass, L.; Klein, M. y Shelton, C.: «Designing software architectures to achieve quality attribute requirements». *Proceedings of IEE Software*, 2005, **152(4)**, pp. 153–165. ISSN 1462-5970. doi: 10.1049/ip-sen:20045037.

[20] Balsamo, S.; Di Marco, A.; Inverardi, P. y Simeoni, M.: «Model-Based Performance Prediction in Software Development: A Survey», 2004, **30(5)**, IEEE Computer Society. ISSN 0098-5589. doi: 10.1109/TSE.2004.9.

[21] Barbanov, M.: *A Linux-based Real-Time Operating System*. Tesis doctoral, New Mexico Institute of Mining and Technology, 1997.

[22] Bass, L.; Clements, P. y Kazman, R.: *Software architecture in practice*. Addison-Wesley Professional, 1998. ISBN 0-201-19930-0.

[23] Bass, L.; Clements, P. y Kazman, R.: *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003. ISBN 0321154959.

[24] Bass, L.; Ivers, J.; Klein, M. y Merson, P.: «Reasoning Frameworks». *Informe técnico CMU/SEI-2005-TR-007*, Software Engineering Institute (SEI), Carnegie Mellon University, 2005. Disponible en `http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tr007.pdf`

[25] Baudry, B.; Dinh, T.; Mottu, J.M.; Simmonds, D.; France, R.; Ghosh, S.; Fleurey, F. y Le Traon, Y.: «Model transformation testing challenges». En: *Proceedings of the 2006 workshop on Integration of Model Driven Development and Model Driven Testing*, , 2006.

[26] Baudry, B.; Fleurey, F.; Jézéquel, J.M. y Le Traon, Y.: «Automatic test cases optimization: a bacteriologic algorithm». *IEEE Software*, 2005, **22(2)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2005.30.

[27] Beneken, G.; Hammerschall, U.; Broy, M.; Cengarle, M.; Jürjens, J.; Pretschner, A.; Rumpe, B. y Schoenmakers, M.: «Componentware - State of the Art 2003». *Informe técnico*, Institut für Informatik, Technische Universität München, 2003. Disponible en `http://www4.in.tum.de/~beneken/papers/ComponentWare.pdf`

[28] Benini, L.; Farella, E. y Guiducci, C.: «Wireless sensor networks: Enabling technology for ambient intelligence». *Microelectronics Journal*, 2006, **37(12)**, pp. 1639–1649. ISSN 0026-2692. doi: http://dx.doi.org/10.1016/j.mejo.2006.04.021.

[29] Beugnard, A.; Jézéquel, J.M.; Plouzeau, N. y Watkins, D.: «Making Components Contract Aware». *IEEE Computer*, 1999, **32(7)**, pp. 38–45. ISSN 0018-9162. doi: 10.1109/2.774917.

[30] Björkander, M. y Kobryn, C.: «Architecting Systems with UML 2.0». *IEEE Software*, 2003, **20(4)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2003.1207456.

[31] Blum, S.: «From a CORBA-Based Software Framework to a Component-Based System Architecture for Controlling a Mobile Robot». En: *Proceedings of the* 3^{rd} *International Conference on Computer Vision Systems*, volumen 2626 de *Lecture Notes on Computer Science*, pp. 333–344. Springer-Verlag. ISSN 0302-9743, 2003.

[32] Boehm, B. y In, H.: «Identifying Quality-Requirement Conflicts». *IEEE Software*, 1996, **13(2)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/52.506460.

[33] Bonasso, R.; Firby, J.; Gat, E.; Kortenkamp, D.; Miller, D. y Slack, M.: «Experiences with an Architecture for Intelligent, Reactive Agents». *Journal of Experimental & Theoretical Artificial Intelligence*, 1997, **9(2/3)**, Taylor and Francis Ltd. ISSN 0952-813X. doi: 10.1080/095281397147103.

[34] Booch, G.: *Software Components With Ada: Structures, Tools, and Subsystems*. Benjamin-Cummings, 1987. ISBN 0805306099.

[35] Bosch, J.: *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley Longman Publishing Co., Inc., 1ª edición, 2000. ISBN 0-201-67494-7.

[36] Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S. y Oreback, A.: «Towards component-based robotics». En: *Proceedings of the International Conference on Intelligent Robots and Systems*, pp. 163–168. IEEE Computer Society. ISBN 0-7803-8912-3, 2005. doi: 10.1109/IROS.2005.1545523.

[37] Brooks, C., R. an Breazeal; Marjanovic, M.; Scassellati, B. y Williamson, M.: «The Cog Project: Building a Humanoid Robot». En: *Proceedings of the International Workshop on Computation for Metaphors, Analogy and Agents*, volumen 1562 de *Lecture Notes on Computer Science*, pp. 52–87. Springer-Verlag. ISSN 0302-9743, 1999.

[38] Brooks, F., Jr.: *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201835959.

[39] Brooks, R.: «A robust layered control system for a mobile robot». *IEEE Journal of Robotics and Automation*, 1986, **2(1)**, IEEE Computer Society. ISSN 0882-4967.

[40] Brown, A.: *Building Systems from Pieces with Component-Based Software Engineering*. volumen Constructing Superior Software, capítulo 6. Sams, 1999. Disponible en `http://www.cbd-hq.com/PDFs/cbdhq_991215ab_superiorsoftware_exerpt.pdf`

[41] Brown, A. y Wallnau, K.: «The Current State of CBSE». *IEEE Software*, 1998, **15(5)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/52.714622.

[42] Bruyninckx, H.: «Open Robot Control Software: the OROCOS project». En: *Proceedings of the IEEE International Conference on Robotics and Automation*, volumen 3, pp. 2523–2528. IEEE Computer Society. ISBN 0-7803-6578-X, 2001. doi: 10.1109/ROBOT.2001.933002.

[43] Budinsky, F.; Steinberg, D.; Merks, E.; Ellersick, R. y Grose, T.: *Eclipse Modeling Framework*. Eclipse series. Addison-Wesley Professional, 2003. ISBN 0131425420.

[44] Bures, T. y Plasil, F.: «Communication Style Driven Connector Configurations». En: *Proceedings of the* 1^{st} *International Conference on Software Engineering Research and Applications*, volumen 3026 de *Lecture Notes on Computer Science*, pp. 102–116. Springer-Verlag. ISSN 0302-9743, 2004. doi: 10.1007/b97161.

[45] Buschmann, F.; Henney, K. y C. Schmidt, D.: *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. John Wiley and Sons Ltd, 2007. ISBN 0471486485.

[46] Buschmann, F.; Henney, K. y Schmidt, D.: *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. John Wiley and Sons Ltd., 2007. ISBN 0471486485.

[47] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P. y Stal, M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996. ISBN 0471958697.

[48] Bálek, D. y Plasil, F.: «Software Connectors and their Role in Component Deployment». En: *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pp. 69–84. Kluwer, B.V. ISBN 0-7923-7481-9, 2001. Disponible en `http://nenya.ms.mff.cuni.cz/publications/DAIS01.pdf`

[49] Bézivin, J.: «On the unification power of models». *Journal of Software and Systems Modeling*, 2005, **4(2)**, pp. 171–188. doi: 10.1007/s10270-005-0079-0.

[50] Bézivin, J. y Jouault, F.: «Using ATL for Checking Models». *Electronic Notes in Theoretical Computer Science*, 2006, **152**, Elsevier Science Inc. ISSN 1571-0661. doi: 10.1016/j.entcs.2006.01.015.

[51] Bézivin, J.; Jouault, F. y Touzet, D.: *Model engineering: from principles to platforms*. capítulo Model Driven Engineering for Distributed Real-Time Embedded Systems, pp. 15–30. Hermes Science Publishing Ltd. ISBN 1905209320, 2005.

[52] Carney, D.: «Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities». *SEI Monographs on the Use of Commercial Software in Government Systems 1*, Software Engineering Institute (SEI), 1997. Disponible en `http://www.sei.cmu.edu/cbs/monographs/assembling-systems/assembling.systems.pdf`

[53] Carney, D. y Long, F.: «What do you mean by COTS? Finally, a usefull answer». *IEEE Software*, 2000, **17(2)**, pp. 83–86. ISSN 0740-7459. doi: 10.1109/52.841700.

[54] Carney, D.; Morris, E. y Place, P.: «Identifying Commercial Off-the-Shelf (COTS) Product Risks: The COTS Usage Risk Evaluation». *Informe técnico CMU/SEI-2003-TR-023*, Software Engineering Institute (SEI), Carnegie Mellon University, 2003. Disponible en `http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tr023.pdf`

[55] Chaimowicz, A., L. andCowley; Sabella, V. y Taylor, C.J.: «ROCI: A Distributed Framework for Multi-Robot Perception and Control». En: *Proc. of the 2003 IEEE/RSJ, Intl. Conference on Intelligent Robots and Systems*, pp. 266–271. ISBN 0-7803-7860-1/03, 2003.

[56] Chambers, C.: «Towards reusable, extensible components». *ACM Computing Surveys*, 1996, **28(4es)**, p. 192. ISSN 0360-0300. doi: 10.1145/242224.242473.

[57] Cheesman, J. y Daniels, J.: *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley Professional, 1ª edición, 2000. ISBN 0201708515.

[58] Childs, A.; Greenwald, J.; Jung, G.; Hoosier, M. y Hatcliff, J.: «CALM and Cadena: Metamodeling for Component-Based Product-Line Development». *IEEE Computer*, 2006, **39(2)**, IEEE Computer Society. ISSN 0018-9162. doi: 10.1109/MC.2006.51.

[59] Clements, P.: «Comparing the SEI's Views and Beyond Approach for Documenting Software Architectures with ANSI-IEEE 1471-2000». *Informe técnico CMU/SEI-2005-TN-017*, Software Engineering Institute (SEI), Carnegie Mellon University, 2005. Disponible en `http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tn017.pdf`

[60] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R. y Stafford, J.: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002. ISBN 0201703726.

[61] Clements, P.; Kazman, R. y Klein, M.: *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2002. ISBN 020170482X.

[62] Clements, P. y Northrop, L.: *Software Product Lines : Practices and Patterns*. Software Engineering (SEI). Addison-Wesley Professional, 3ª edición, 2001. ISBN 0201703327.

[63] Collett, T.; MacDonald, B. y Gerkey, B.: «Player 2.0: Toward a Practical Robot Programming Framework». En: *Proceedings of the Australasian Conference on Robotics and Automation*, , 2005. Disponible en `http://www.araa.asn.au/acra/acra2005/papers/collet.pdf`

[64] Coste-Manière, E. y Simmons, R.: «Architecture, the Backbone of Robotic Systems». En: *Proceedings of the 2000 IEEE International Conference on Robotics & Automation*, pp. 67–72. IEEE, 2000. Disponible en `http://www-2.cs.cmu.edu/~reids/papers/backbone.pdf`

[65] Cowley, A.; Chaimowicz, L. y Taylor, C.J.: «Design minimalism in robotics programming». *International Journal of Advanced Robotic Systems*, 2006, **3(1)**, pp. 31–36. ISSN 1729–8806.

[66] Crnkovic, I. y Larsson, M.: *Building Reliable Component-Based Software Systems*. Artech House Publisher, 2002. ISBN 1580533272.

[67] Czarnecki, K.: «Overview of Generative Software Development». En: *Proceedings of the International Workshop on Unconventional Programming Paradigms*, volumen 3566 de *Lecture Notes on Computer Science*, pp. 326–341. Springer-Verlag. ISSN 0302-9743, 2005. doi: 10.1007/11527800_25.

[68] Czarnecki, K.; Eisenecker, U. y Czarnecki, K.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000. ISBN 0201309777.

[69] Czarnecki, K. y Helsen, S.: «Classification of Model Transformation Approaches». En: *Proceedings of the* 2^{nd} *OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, , 2003.

[70] Dashofy, E.; van der Hoek, A. y Taylor, R.: «An infrastructure for the rapid development of XML-based architecture description languages». En: *Proceedings of the* 24^{th} *International Conference on Software Engineering*, pp. 266–276. ISBN 1-58113-472-X, 2002. doi: 10.1145/581339.581374.

[71] Dashofy, E.; van der Hoek, A. y Taylor, R.: «A comprehensive approach for the development of modular software architecture description languages». *ACM Trans. Softw. Eng. Methodol.*, 2005, **14(2)**, ACM Press. ISSN 1049-331X. doi: 10.1145/1061254.1061258.

[72] DeMichiel, L. y Keith, M.: «JSR 220: Enterprise JavaBeans,Version 3.0». Online, 2006. Disponible en `http://java.sun.com/products/ejb/docs.html`

[73] Dijkstra, E.: «The humble programmer». *Commun. ACM*, 1972, **15(10)**, pp. 859–866.

[74] Dobrica, L. y Niemelä, E.: «A survey on software architecture analysis methods». *IEEE Transactions on Software Engineering*, 2002, **28(7)**, IEEE Computer Society. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019479.

[75] Dogru, A.H. y Tanik, M.M.: «A Process Model for Component-Oriented Software Engineering». *IEEE Software*, 2003, **20(2)**, pp. 34–41. ISSN 0740-7459. doi: 10.1109/MS.2003.1184164.

[76] Douglass, B.: *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley Professional, 2004. ISBN 0321160762.

[77] Douglass, B.P.: *Doing hard time: developing real-time systems with UML, objects, frameworks and patterns*. Object Technology. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-49837-5.

[78] Douglass, B.P.: *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Object Technology. Addison-Wesley Professional, 2002. ISBN: 0-201-69956-7.

[79] D'Souza, D. y Wills, A.: *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0201310120.

[80] Easton, M. y King, J.: *Mono, Portable.Net, and .Net: Cross-Platform .Net Coding*. APress, 2004. ISBN 1590593308.

[81] Estublier, J.; Vega, G. y Daniela, A.: «Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications». En: *Model Driven Engineering Languages and Systems, 8^th International Conference, MoDELS 2005*, volumen 3713 de *Lecture Notes on Computer Science*, pp. 69–83. Springer-Verlag. ISBN 3-540-29010-9. ISSN 0302-9743, 2005. doi: 10.1007/11557432_6.

[82] Favre, Jean-Marie:. «Foundations of Meta-Pyramids: Languages vs. Metamodels. Episode II: Story of Thotus the Baboon». Online.

[83] Favre, J.M.: «Towards a Basic Theory to Model Driven Engineering». En: *Proceedings of the 3^rd Workshop in Software Model Engineering*, volumen 3297 de *Lecture Notes on Computer Science*. Springer-Verlag. ISSN 0302-9743, 2004. Disponible en `http://www.metamodel.com/wisme-2004/present/22.pdf`

[84] Favre, J.M. y NGuyen, T.: «Towards a Megamodel to Model Software Evolution Through Transformations». *Electronic Notes in Theoretical Computer Science*, 2005, **137(3)**, Elsevier Science Inc. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.08.034.

[85] Fayad, M.; Schmidt, D. y Johnson, R.: *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1999. ISBN 0471248754.

[86] Feiler, P.; Gluch, D. y Hudak, J.: «The Architecture Analysis & Design Language (AADL): An Introduction». *Informe técnico CMU/SEI-2006-TN-011*, Software Engineering Institute (SEI), Carnegie Mellon University, 2006. Disponible en `http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn011.pdf`

[87] Feiler, P.; Gluch, D.; Hudak, J. y Lewis, B.: «Embedded Systems Architecture Analysis Using SAE AADL». *Informe técnico CMU/SEI-2004-TN-005*, Software Engineering Institute (SEI), Carnegie Mellon University, 2004. Disponible en `http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tn005.pdf`

[88] Felix Bachmann, F.; Bass, L. y Klein, M.: «Illuminating the Fundamental Contributors to Software Architecture Quality». *Informe técnico CMU/SEI-2002-TR-025*, Software Engineering Institute (SEI), Carnegie Mellon University, 2002. Disponible en `http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr025.pdf`

[89] Fernández, C.; Iborra, A.; Álvarez, B.; Pastor, J.A.; Sánchez, P.; Fernández, J.M. y Ortega, N.: «Co-operative Robots for Hull Blasting in European Shiprepair Industry». *IEEE Journal of Robotics and Automation*, 2004. ISSN: 1070-9932.

[90] Fiadeiro, J.L.: «Designing for Software's Social Complexity». *IEEE Computer*, 2007, **40(1)**, pp. 34–39. ISSN 0018-9162. doi: 10.1109/MC.2007.16.

[91] Firby, James: *Adaptive Execution in Complex Dynamic Worlds*. Tesis doctoral, Yale University, 1989. Disponible en `http://people.cs.uchicago.edu/users/firby/thesis/thesis.ps.Z`

[92] Franch, X. y Torchiano, M.: «Towards a reference framework for COTS-based development: a proposal». En: 2^{nd} *International workshop on models and processes for the evaluation of off-the-shelf components*, pp. 1–4. ACM Press. ISBN 1-59593-129-5, 2005. doi: 10.1145/1082948.1082952.

[93] Fröhlich, P.H. y Franz, M.: «Component-Oriented Programming in Object-Oriented Languages». *Informe técnico 99-49*, Department of Information and Computer Science, University of California, Irvine, 1999. Disponible en `http://www.ics.uci.edu/~franz/publications/COPinOOL-ics-tr-99-49.pdf`

[94] Fröhlich, P.H. y Franz, M.: «Stand-Alone Messages: A Step Towards Component-Oriented Programming Languages». En: *Modular Programming Languages: Proceedings of the Fifth Joint Modular Languages Conference (JMLC 2000)*, Número 1891 en Lecture Notes on Computer Science, pp. 90–103. Springer-Verlag. ISSN 0302-9743, 2000. Disponible en `http://www.ics.uci.edu/~franz/Site/pubs-pdf/C13.pdf`

[95] Fröhlich, P.H.; Gal, A. y Franz, M.: «Supporting software composition at the programming language level». *Science of Computer Programming*, 2004, **56(1-2)**, pp. 41–57. doi: 10.1016/j.scico.2004.11.004.

[96] Gamma, E.; Helm, R.; Johnson, R. y Vlissides, J.: *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995. ISBN 0201633612.

[97] Gao, J.; Gupta, K.; Gupta, S. y Shim, S.: «On Building Testable Software Components». En: *COTS-Based Software Systems*, volumen 2255 de *Lecture Notes on Computer Science*, pp. 108–121. Springer-Verlag. ISSN 0302-9743, 2002.

[98] García, F.J.; Barras, J.A.; Laguna, M.A. y Marqués, J.M.: «Líneas de Productos, Componentes, Frameworks y Mecanos». *Informe técnico DPTOIA-IT-2002-004*, Universidad de Salamanca, 2002. Disponible en `http://tejo.usal.es/inftec/2002/DPTOIA-IT-2002-004.pdf`

[99] Gat, E.: *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. capítulo Three-layer architectures, pp. 195–210. MIT Press. ISBN 0-262-61137-6, 1998.

[100] Gay, D.; Levis, P.; von Behren, R.; Welsh, M.; Brewer, E. y Culler, D.: «The nesC Language: A Holistic Approach to Networked Embedded Systems». En: *Programming Language Design and Implementation (PLDI)*, pp. 1–11. ACM Press. ISBN 1-58113-662-5, 2003. doi: 10.1145/781131.781133.

[101] Gerkey, B.; Vaughan, R.; Stoy, K.; Howard, A.; Sukhatme, G. y Mataric, M.: «Most valuable player: a robot device server for distributed control». En: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volumen 3, pp. 1226–1231. IEEE Computer Society. ISBN 0-7803-6612-3, 2001. doi: 10.1109/IROS.2001.977150.

[102] Gomaa, H.: *Designing Software Product Lines with UML*. Addison-Wesley Professional, 2004. ISBN 0201775956.

[103] Gomaa, H. y Menascé, D.: «Performance Engineering of Component-Based Distributed Software Systems». En: *Performance Engineering: State of the Art and Current Trends*, volumen 2047 de *Lecture Notes on Computer Science*, pp. 40–55. Springer-Verlag. ISSN 0302-9743, 2001.

[104] Grabriel, R.: «Objects have failed». OOPSLA'02 debate (Seattle), 2002. Disponible en `http://www.dreamsongs.com/Files/ObjectsHaveFailed.pdf`

[105] Greenfield, J.; Short, K.; Cook, S.; Kent, S. y Crupi, J.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 1ª edición, 2004. ISBN 0471202843.

[106] Guarino, N. y Welty, C.: «Towards a Methodology for Ontology Based Model Engineering». En: *Proc. of the ECOOP Workshop on Model Engineering*, , 2000.

[107] Hamilton, G.: «Java Beans Specification 1.01». Online, 1997. Disponible en `http://java.sun.com/products/javabeans/docs/spec.html`

[108] Harel, D.: «Statecharts: A Visual Formalism for Complex Systems». *Science of Computer Programming*, 1987, **8(3)**, Elsevier. ISSN 0167-6423. doi: 10.1016/0167-6423(87)90035-9.

[109] Hayes-Roth, B.: «Blackboard for Control». *Artificial Intelligence*, 1985, **26(3)**, Elsevier Science Inc. ISSN 0004-3702. doi: 10.1016/0004-3702(85)90063-3.

[110] Heineman, G. y Councill, W.: *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 2001. ISBN 0201704854.

[111] Henning, M.: «A New Approach to Object-Oriented Middleware». *IEEE Internet Computing*, 2004, **8(1)**, IEEE Computer Society. ISSN 1089-7801. doi: 10.1109/MIC.2004.1260706.

[112] Hissam, S.; Ivers, J.; Plakosh, D. y Wallnau, K.: «Pin Component Technology (V1.0) and Its C Interface». *Informe técnico CMU/SEI-2005-TN-001*, Software Engineering Institute (SEI), 2005. Disponible en `http://www.sei.cmu.edu/publications/documents/05.reports/05tn001.html`

[113] Hissam, S.; Klein, M.; Lehoczky, J.; Merson, P.; Moreno, G. y Wallnau, K.: «Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)». *Informe técnico CMU/SEI-2004-TR-017*, Software Engineering Institute (SEI), Carnegie Mellon University, 2004. Disponible en `http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tr017.pdf`

[114] Hoare, C.: *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN 0131532898.

[115] Hofmeister, C.; Nord, R. y Soni, D.: *Applied Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0-201-32571-3.

[116] Hopkins, J.: «Component primer». *Commun. ACM*, 2000, **43(10)**, pp. 27–30. ISSN 0001-0782. doi: 10.1145/352183.352198.

[117] Iborra, A.; Pastor, J.A.; Álvarez, B.; Fernández, C. y Fernández-Meroño, J.M.: «Robots in Radioactive Environments». *IEEE Journal of Robotics and Automation*, 2003, **10(4)**, IEEE Computer Society. ISSN 1070-9932. doi: 10.1109/MRA.2003.1256294.

[118] Iborra, A.; Álvarez, B.; Pastor, J.A. y Fernández, J.M.: «Robotized system for retrieving fallen objects within the reactor vessel of a nuclear power plant (PWR)». En: *IEEE International Symposium of Industrial Electronics (ISIE'2000)*, ISBN 0780366069, 2000.

[119] International Standard ISO/IEC 9126: «Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use», 1991.

[120] Iribarne, L.; Troya, J. y Vallecillo, A.: «Selecting Software Components with Multiple Interfaces». En: 28^{th} *EUROMICRO Conference*, pp. 26–32. IEEE Computer Society. ISBN 0-7695-1787-0. ISSN 1089-6503, 2002. doi: 10.1109/EURMIC.2002.1046129.

[121] Iribarne, L.; Troya, J.M. y Vallecillo, A.: «A Trading Service for COTS Components». *The Computer Journal*, 2004, **47(3)**, Oxford University Press. ISSN 0010-4620. doi: 10.1093/comjnl/47.3.342.

[122] Jacobson, I.; Booch, G. y Rumbaugh, J.: *El proceso unificado de desarrollo software (RUP)*. Object Technology. Addison-Wesley Longman Publishing Co., Inc., 2ª edición, 2000. ISBN 8478290363.

[123] Jonathan Aldrich, J.; Chambers, C. y Notkin, D.: «Architectural Reasoning in ArchJava». *Informe técnico UW-CSE-02-04-01*, University of Washington, 2002. Disponible en `http://archjava.fluid.cs.cmu.edu/papers/UW-CSE-02-04-01.pdf`

[124] Jouault, F. y Kurtev, I.: «On the architectural alignment of ATL and QVT». En: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1188–1195. ACM Press. ISBN 1-59593-108-2, 2006. doi: 1141277.1141561.

[125] Kang, K.; Kim, J., S.and Lee; Kim, K.; Shin, E. y Huh, M.: «FORM: A feature-oriented reuse method with domain-specific reference architectures». *Annals of Software Engineering*, 1998, **5(0)**, Springer-Verlag. ISSN 1022-7091. doi: 10.1023/A:1018980625587.

[126] Kazman, R.; Bass, L. y Klein, M.: «The essential components of software architecture design and analysis». *Journal of Systems and Software*, 2006, **79(8)**, pp. 1207–1216. doi: 10.1016/j.jss.2006.05.001.

[127] Kazman, R.; Bass, L.; Klein, M.; Lattanze, T. y Northrop, L.: «A Basis for Analyzing Software Architecture Analysis Methods». *Software Quality Control*, 2005, **13(4)**, Kluwer Academic Publishers. ISSN 0963-9314. doi: 10.1007/s11219-005-4250-1.

[128] Kent, S.: «Model Driven Engineering». En: *Integrated Formal Methods, 3^{rd} International Conference, IFM 2002*, volumen 2335 de *Lecture Notes on Computer Science*, pp. 286–298. Springer-Verlag. ISBN 3-540-43703-7. ISSN 0302-9743, 2002.

[129] Kircher, M. y Prashant Jain, P.: *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource*. John Wiley and Sons Ltd., 2004. ISBN 0470845252.

[130] Klein, M. y Kazman, R.: «Attribute-Based Architectural Styles». *Informe técnico CMU/SEI-99-TR-022*, Software Engineering Institute (SEI), Carnegie Mellon University, 1999. Disponible en `http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr022.pdf`

[131] Kleppe, A.; Warmer, J. y Bast, W.: *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional, 2003. ISBN 032119442X.

[132] Konolige, K. y Myer, K.: *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. capítulo The Saphira Architecture for Autonomous Mobile Robots, pp. 211–242. MIT Press. ISBN 0-262-61137-6, 1998.

[133] Krasner, G. y Pope, S.: «A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80». *Journal of Object-Oriented Programming*, 1988, **1(3)**, pp. 26–49. ISSN 0896-8438.

[134] Kruchten, P.: «The 4+1 View Model of Architecture». *IEEE Software*, 1995, **12(6)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/52.469759.

[135] Kruchten, P.; Obbink, J. y Stafford, J.: «The Past, Present, and Future for Software Architecture». *IEEE Software*, 2006, **23(2)**, pp. 22–30. ISSN 0740-7459. doi: 10.1109/MS.2006.59.

[136] Kurtev, I.; Bézivin, J. y Aksit, M.: «Technological Spaces: an Initial Appraisal», 2002. Disponible en `www.sciences.univ-nantes.fr/lina/atl/www/papers/PositionPaperKurtev.pdf`

[137] Lange, C.; Chaudron, M. y Muskens, J.: «In Practice: UML Software Architecture and Design Description». *IEEE Software*, 2006, **23(2)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2006.50.

[138] Lee, J. y Bass, L.: «Elements of a Usability Reasoning Framework». *Informe técnico CMU/SEI-2005-TN-030*, Software Engineering Institute (SEI), Carnegie Mellon University, 2005. Disponible en `http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tn030.pdf`

[139] Len Bass, L.; Klein, M. y Bachmann, F.: «Quality Attribute Design Primitives». *Informe técnico CMU/SEI-2000-TN-017*, Software Engineering Institute (SEI), Carnegie Mellon University, 2000. Disponible en `http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tn017.pdf`

[140] Makarenko, A.; Brooks, A. y Kaupp, T.: «Orca: Components for Robotics». En: *Proceedings of the International Conference on Intelligent Robots and Systems*, IEEE Computer Society, 2006.

[141] Martin Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. y Don Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672.

[142] McIlroy, M.: «Mass produced software components». En: *Proceedings of NATO Software Engineering Conference*, volumen 1, pp. 138–150. NATO Science Committee, 1968.

[143] Medvidovic, N.; Rosenblum, D.; Redmiles, D. y Robbins, J.: «Modeling software architectures in the Unified Modeling Language». *ACM Trans. Softw. Eng. Methodol.*, 2002, **11(1)**, pp. 2–57. ISSN 1049-331X. doi: 10.1145/504087.504088.

[144] Medvidovic, N. y Taylor, R.: «A Classification and Comparison Framework for Software Architecture Description Languages». *IEEE Transactions on Software Engineering*, 2000, **26(1)**, IEEE Computer Society. ISSN 0098-5589. doi: 10.1109/32.825767.

[145] Mehta, N. y Medvidovic, N.: «Understanding Software Connector Compatibilities Using A Connector Taxonomy». *Informe técnico USC-CSE-2002-511*, Center for Software Engineering (USC), 2002. Disponible en `http://csse.usc.edu/publications/TECHRPTS/2002/usccse2002-511/usccse2002-511.pdf`

[146] Mehta, N.; Medvidovic, N. y Phadke, S.: «Towards a taxonomy of software connectors». En: *Proceedings of the* 22^{nd} *international conference on Software Engineering*, pp. 178–187. ACM Press. ISBN 1-58113-206-9, 2000. doi: 10.1145/337180.337201.

[147] Mellor, S. y Balcer, M.: *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., 1ª edición, 2002. ISBN 0201748045.

[148] Mellor, S.; Scott, K.; Uhl, A. y Weise, D.: *MDA Distilled*. Object Technology. Addison-Wesley Professional, 1ª edición, 2004. ISBN 0-201-78891-8.

[149] Mens, T. y van Gorp, P.: «A Taxonomy of Model Transformation». *Electronic Notes in Theoretical Computer Science*, 2006, **152**, Elsevier Science Inc. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.10.021.

[150] Mernik, M.; Heering, J. y Sloane, A.: «When and how to develop domain-specific languages». *ACM Comput. Surv.*, 2005, **37(4)**, ACM Press. ISSN 0360-0300. doi: 10.1145/1118890.1118892.

[151] Meyer, B.: «Applying 'design by contract'». *IEEE Computer*, 1992, **25(10)**, pp. 40–51. ISSN 0018-9162. Disponible en `http://se.ethz.ch/~meyer/publications/computer/contract.pdf`

[152] Meyer, B.: «The grand challenge of Trusted Components». En: *Proceedings of the* 25^{th} *International Conference on Software Engineering*, pp. 660–667. IEEE Computer Society. ISBN 0-7695-1877-X, 2003. doi: 10.1109/ICSE.2003.1201252.

[153] Meyers, B. y Oberndorf, P.: *Managing Software Acquisition: Open Systems and COTS Products*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70454-4.

[154] MITRE Corp: «A Guide to Total Software Quality Control. Volume 1». *Informe técnico ADA263881*, Defense Technical Information Center (DoD), 1992.

[155] Mogul, J.: «Emergent (mis)behavior vs. complex software systems». En: *Proceedings of the 2006 EuroSys conference*, pp. 293–304. ACM Press. ISBN 1-59593-322-0, 2006. doi: 10.1145/1217935.1217964.

[156] Montemerlo, M.; Roy, N. y Thrun, S.: «Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit». En: *Proceedings of the Intelligent Robots and Systems*, volumen 3, pp. 2436– 2441. IEEE Computer Society. ISBN 0-7803-7860-1, 2003. doi: 10.1109/IROS.2003.1249235.

[157] Muller, P.A.; Fleurey, F. y Jézéquel, J.M.: «Weaving executability into object-oriented meta-languages». En: *Proceedings of MODELS/UML'2005*, volumen 3713 de *Lecture Notes on Computer Science*, pp. 264–278. Springer-Verlag. ISBN 3-540-29010-9. ISSN 0302-9743, 2005. doi: 10.1007/11557432_19.

[158] Musliner, David: *CIRCA: the cooperative intelligent real-time control architecture*. Tesis doctoral, University of Michigan, 1993.

[159] Nesnas, I.; R. Simmons, R.; Gaines, D.; Kunz, C.; Diaz-Calderon, A.; Estlin, T.; Madison, R.; Guineau, J.; McHenry, M.; Shu, I. y Apfelbaum, D.: «CLARAty: Challenges and Steps Toward Reusable Robotic Software». *International Journal of Advanced Robotic Systems*, 2006, **3(1)**, pp. 23–30. ISSN 1729-8806.

[160] Nesnas, I.; Wright, A.; Bajracharya, M.; Simmons, R.; Estlin, T. y Kim, W.: «CLARAty: An architecture for reusable robotic software». En: *Proceedings of the SPIE Aerosense Conference*, volumen 5083, pp. 253–264. ISBN 0-8194-4942-3. ISSN 0277-786X, 2003. Disponible en `http://claraty.jpl.nasa.gov/main/overview/publications/03_nesnas_claraty_spie.pdf`

[161] Nesnas, R., I.and Volpe; Estlin, T.; Das, H.; Petras, R. y Mutz, D.: «Toward developing reusable software components for robotic applications». En: *Proceedings of the 2001 IEEE/RJS International Conference on Intelligent Robots and Systems*, pp. 2375–2383. ISBN 0-7803-6612-3, 2001. Disponible en `http://claraty.jpl.nasa.gov/main/overview/publications/01_nesnas_reusable_iros.pdf`

[162] Nilsson, N.: *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, 1980. ISBN 0934613109.

[163] Nistor, E.; Erenkrantz, J.; Hendrickson, S. y van der Hoek, A.: «ArchEvol: versioning architectural-implementation relationships». En: *Proceedings of the* 12^{th} *international workshop on Software configuration management*, pp. 99–111. ACM Press. ISBN 1-59593-310-7, 2005. doi: 10.1145/1109128.1109136.

[164] Object Management Group (OMG): *Common Warehouse Metamodel (CWM) Specification v1.1, formal/2003-03-02*, 2003. Disponible en `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-02.pdf`

[165] Object Management Group (OMG): *Model Driven Architecture Guide Version v1.0.1, omg/2003-06-01*, 2003. Disponible en `http://www.omg.org/docs/omg/03-06-01.pdf`

[166] Object Management Group (OMG): *Common Object Request Broker Architecture (CORBA/IIOP) formal/04-03-12 Specification*, 2004. Disponible en http://www.omg.org/docs/formal/04-03-12.pdf

[167] Object Management Group (OMG): *Meta-Object Facility (MOF) Specification v2.0, ptc/04-10-15*, 2004. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf

[168] Object Management Group (OMG): *Meta-Object Facility (MOF) v2.0 Query/View/Transformation Specification, ptc/05-11-01*, 2005. Disponible en http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf

[169] Object Management Group (OMG): *Software Process Engineering Metamodel (SPEM) Specification v1.1, formal/2005-01-06*, 2005. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/05-01-06.pdf

[170] Object Management Group (OMG): *Unified Modeling Language (UML) Superstructure Specification v2.0, formal/05-07-04*, 2005. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf

[171] Object Management Group (OMG): *XML Metadata Interchange (XMI) Specification v2.1, formal/2005-09-01*, 2005. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf

[172] Object Management Group (OMG): *CORBA Component Model formal/06-04-01 Specification*, 2006. Disponible en http://www.omg.org/docs/formal/06-04-01.pdf

[173] Object Management Group (OMG): *MOF Models to Text Transformation Language Specification, ptc/06-11-01*, 2006. Disponible en http://www.omg.org/cgi-bin/apps/doc?ptc/06-11-01.pdf

[174] Object Management Group (OMG): *Object Constraint Language (OCL) Specification v2.0, formal/06-05-01*, 2006. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf

[175] Object Management Group (OMG): *OMG Systems Modeling Language (OMG SysML) Specification v1.0, ptc/06-05-04*, 2006. Disponible en http://www.sysml.org/docs/specs/OMGSysML-FAS-06-05-04.pdf

[176] Object Management Group (OMG): *Unified Modeling Language (UML) Infrastructure Specification v2.1.1, formal/07-02-06*, 2007. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/07-02-06.pdf

[177] Object Management Group (OMG): *Unified Modeling Language (UML) Superstructure Specification v2.1.1, formal/2007-02-05*, 2007. Disponible en http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf

[178] Ortiz, F.; Alonso, D.; Álvarez, B. y Pastor, J.A.: «A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle». En: 10^{th} *International Conference on Reliable*

Software Technologies, volumen 3555 de *Lecture Notes on Computer Science*, pp. 13–24. ISSN 0302-9743, 2005.

[179] Ortiz, F.; Martínez, A.S.; Álvarez, B.; Iborra, A. y Fernández, J.M.: «Development of a Control System for Teleoperated Robots Using UML and Ada'95». En: *VII Ada-Europe International Conference on Reliable Software Technologies*, , 2002.

[180] Ortiz, F.; Pastor, J.; Álvarez, B.; Iborra, A.; Ortega, N.; Rodríguez, D. y Conesa, C.: «Robots for hull ship cleaning». En: *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE'07)*, IEEE Computer Society, 2007.

[181] Ortiz Zaragoza, Francisco José: *Arquitectura de Referencia para Unidades de Control de Robots de Servicio Teleoperados*. Tesis doctoral, Dpto. Tecnología Electrónica, Universidad Politécnica de Cartagena (Spain), 2005.

[182] Page-Jones, M.: *Practical Guide to Structured Systems Design (2nd Edition)*. Prentice Hall, 1988. ISBN 0136907695.

[183] Parsons, R.: «Components and the World of Chaos». *IEEE Software*, 2003, **20(3)**, IEEE Computer Society Press. ISSN 0740-7459. doi: 10.1109/MS.2003.1196326.

[184] Pastor, J.A.; Álvarez, B.; Iborra, A. y Fernádez, J.M.: «An underwater teleoperated vehicle for inspection and retrieving». En: *CLAWAR'98, I International Symposium CLAWAR*, , 1998.

[185] Pastor Franco, Juan Ángel: *Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales*. Tesis doctoral, ETSIT, Universidad Politécnica de Cartagena (Spain), 2002.

[186] Pfister, C. y Szyperski, C.: «Why Objects are Not Enough». En: *Proceedings, International Component Users Conference*, pp. 141–147. SIGS. ISBN 0-521-64821-1, 1996.

[187] Pressman, R.: *Ingenieria del Software - Un Enfoque Practico*. McGraw-Hill, 4ª edición, 1998. ISBN 8448111869.

[188] Prieto-Diaz, R. y Neighbors, J.: «Module interconnection languages». *Journal of Systems and Software*, 1986, **6(4)**, Elsevier Science Inc. ISSN 0164-1212. doi: 10.1016/0164-1212(86)90002-6.

[189] Roshandel, R. y Medvidovic, N.: «Relating Software Component Models». *Informe técnico USC-CSE-2003-504*, Center for Software Engineering (USC), 2003. Disponible en `http://sunset.usc.edu/publications/TECHRPTS/2003/usccse2003-504/usccse2003-504.pdf`

[190] Schmidt, D.; Stal, M.; Rohnert, H. y Buschmann, F.: *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000. ISBN 0471606952.

[191] Schmidt, D.C.: «Model-Driven Engineering». *IEEE Computer*, 2006, **39(2)**, IEEE Computer Society. ISSN 0018-9162. doi: 10.1109/MC.2006.58.

[192] Scholl, K.; Albiez, J. y Gassmann, B.: «MCA – An Expandable Modular Controller Architecture». En: *Proceedings of the* 3^{rd} *Real-Time Linux Workshop*, , 2001.

[193] Seidewitz, E.: «What Models Mean». *IEEE Software*, 2003, **20(5)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2003.1231147.

[194] Selic, B.: «The Pragmatics of Model-Driven Development». *IEEE Transactions on Software Engineering*, 2003, **20(5)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2003.1231146.

[195] Selic, B.; Gullekson, G. y Ward, P.T.: *Real-Time Object-Oriented Modelling (ROOM)*. John Wiley and Sons, 1994. ISBN: 0-471-59917-4.

[196] Sendall, S. y Kozaczynski, W.: «Model Transformation: The Heart and Soul of Model-Driven S ftware Development». *IEEE Software*, 2003, **20(5)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2003.1231150.

[197] Shaw, M. y Clements, P.: «The Golden Age of Software Architecture.» *IEEE Software*, 2006, **23(2)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2006.58.

[198] Shaw, M. y Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. ISBN 0131829572.

[199] Simmons, R.: «Structured control for autonomous robots». *IEEE Journal of Robotics and Automation*, 1994, **10(1)**, IEEE Computer Society. ISSN 0882-4967. doi: 10.1109/70.285583.

[200] Spinellis, D.: «Notable Design Patterns for Domain Specific Languages». *Journal of Systems and Software*, 2001, **56(1)**, pp. 91–99. ISSN 0164-1212. doi: 10.1016/S0164-1212(00)00089-3.

[201] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2ª edición, 2002. ISBN 0201745720.

[202] Teschke, T. y Jörg, J.: «Towards a Foundation of Component-Oriented Software Reference Models». En: *Generative and Component-Based Software Engineering*, volumen 2177 de *Lecture Notes on Computer Science*, pp. 70–84. Springer-Verlag. ISSN 0302-9743, 2000.

[203] The ANSI-IEEE 1471-2000 Standard: «IEEE Recommended Practice for Architectural Description of Software-Intensive Systems». The Institute of Electrical and Electronics Engineers (IEEE), 2000.

[204] The IEEE 1061-1998 Standard: «IEEE Standard for a Software Quality Metrics Methodology». The Institute of Electrical and Electronics Engineers (IEEE), 1998.

[205] The ISO/IEC 9126-1:2001 Standard: «Software engineering – Product quality – Part 1: Quality model». International Organization for Standarization, 2001.

[206] Torchiano, M. y Morisio, M.: «Overlooked Aspects of COTS-Based Development». *IEEE Software*, 2004, **21(2)**, IEEE Computer Society. ISSN 0740-7459. doi: 10.1109/MS.2004.1270770.

[207] Uddel, J.: «ComponentWare». *Byte Magazine*, 1994, **19(5)**, pp. 46–56. Disponible en `http://www.byte.com/art/9405/sec5/art1.htm`

[208] van Deursen, A.; Klint, P. y Visser, J.: «Domain-specific languages: an annotated bibliography». *SIGPLAN Not.*, 2000, **35(6)**, pp. 26–36. ISSN 0362-1340. doi: 10.1145/352029.352035.

[209] van Ommering, R.: «Building product populations with software components». En: *Proceedings of the 24^{th} International Conference on Software Engineering*, pp. 255–265. ACM Press. ISBN 1-58113-472-X, 2002. doi: 10.1145/581339.581373.

[210] van Ommering, R.; van der Linden, F.; Kramer, J. y Magee, J.: «The Koala Component Model for Consumer Electronics Software». *IEEE Computer*, 2000, **33(3)**, IEEE Computer Society. ISSN 0018-9162. doi: 10.1109/2.825699.

[211] Vassilopoulos, D.; Pilioura, T. y Tsalgatidou, A.: «Distributed Technologies CORBA, Enterprise JavaBeans, Web Services – A Comparative Presentation». En: *Proceedings of the 14^{th} Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pp. 280–284. IEEE Computer Society. ISBN 0-7695-2513-X, 2006. doi: 10.1109/PDP.2006.29.

[212] Vestal, S.: «A Cursory Overview and Comparison of Four Architecture Description Languages». *Informe técnico*, Honeywell Technology Center, 1993. Disponible en `http://www.htc.honeywell.com/projects/dssa/ftp/papers/four_adl.ps`

[213] Vicente-Chicote, C.; Alonso, D. y Álvarez, B.: «StateML: modelado gráfico de máquinas de estados y generación de código siguiendo un enfoque MDE». En: *Actas de las XII Jornadas de Ingeniería Software y Bases de Datos*, pp. 401–402. ISBN 9788497325950, 2007.

[214] Vincenzi, A.M.; Maldonado, J.C.; Delamaro, M.E.; Spoto, E.S. y Wong, W.E.: «Component-Based Software: An Overview of Testing». En: *Component-Based Software Quality: Methods and Techniques*, volumen 2693 de *Lecture Notes on Computer Science*, pp. 99–127. Springer-Verlag. ISSN 0302-9743, 2003. doi: 10.1007/b11721.

[215] von der Beeck, M.: «A Comparison of Statecharts Variants». En: *Proceedings of the 3^{rd} International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volumen 863 de *Lecture Notes on Computer Science*, pp. 128–148. Springer-Verlag. ISBN 3-540-58468-4. ISSN 0302-9743, 1994. doi: 10.1007/3-540-58468-4_163.

[216] Wallnau, K.: «Software Component Certification: 10 Useful Distinctions». *Informe técnico CMU/SEI-2004-TN-031*, Software Engineering Institute (SEI), Carnegie Mellon University, 2004. Disponible en `http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tn031.pdf`

[217] Wallnau, K.; Hissam, S. y Seacord, R.: *Building Systems from Commercial Components*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70064-6.

[218] Warford, J. y Hug, K.: *Computing Fundamentals: The Theory and Practice of Software Design with BlackBox Component Builder*, 2003. ISBN 3528058285.

[219] Wojcik, R.; Bachmann, F.; Bass, L.; Clements, P.; Merson, P.; Nord, R. y Wood, B.: «Attribute-Driven Design (ADD), Version 2.0». *Informe técnico CMU/SEI-2006-TR-023*, Software Engineering Institute (SEI), Carnegie Mellon University, 2006. Disponible en `http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tr023.pdf`

[220] Zadeh, L.: «Fuzzy algorithms». *Information and Control*, 1968, **12(2)**, Elsevier Science Inc. doi: 10.1016/S0019-9958(68)90211-8.

[221] Álvarez, B.; Alonso, A. y de la Puente, J.A.: «Timing Analysis of a Generic Robot Teleoperation Software Architecture». En: *Proceedings of the XXII IFAC Workshop on Real-Time Programming*, , 1997.

[222] Álvarez, B.; Iborra, A.; Alonso, A. y de la Puente, J.A.: «Reference architecture for robot teleoperation: Development details and practical use». *Control Engineering Practice*, 2001.

[223] Álvarez, B.; Iborra, A.; Alonso, A.; de la Puente, J.A. y Pastor, J.A.: «Developing multi-application remote systems». *Nuclear Engineering International*, 2000, **45(548)**.

[224] Álvarez, B.; Ortiz, F.; Martínez, A.S.; Sánchez, P.; Pastor, J.A. y Iborra, A.: «Towards a Generic Software Architecture for a Service Robot Controller». En: *XV IFAC World Congress*, , 2002.

[225] Álvarez, B.; Sánchez, P.; Pastor, J.A. y Ortiz, F.: «An architectural framework for modeling teleoperated service robots». *Robotica*, 2006, **24(4)**, Cambridge University Press. ISSN 0263-5747. doi: 10.1017/S0263574705002407.

[226] Álvarez Torres, Bárbara: *Arquitectura Software de Referencia para Sistemas de Teleoperación*. Tesis doctoral, Departamento de Ingeniería de Sistemas Telemáticos, Univ. Politécnica de Madrid (Spain), 1997.