Disponible en www.sciencedirect.com
ScienceDirect
Revista Iberoamericana de Automática e Informática industrial 9 (2012) 170–181
www.elsevier.es/RIAI

Generación Automática de Software para Sistemas de Tiempo Real: Un Enfoque basado en Componentes, Modelos y Frameworks

Diego Alonso*, Juan Ángel Pastor, Pedro Sánchez, Bárbara Álvarez, Cristina Vicente-Chicote

División de Sistemas e Ingeniería Electrónica (DSIE), Universidad Politécnica de Cartagena, Hospital de Marina, 30202, Cartagena (España)

Resumen

Los *Sistemas de Tiempo-Real* poseen características que los hacen particularmente sensibles a las decisiones arquitectónicas que se adopten. El uso de *Frameworks* y *Componentes* ha demostrado ser eficaz en la mejora de la productividad y calidad del software, sobre todo si se combina con enfoques de *Líneas de Productos*. Sin embargo, los resultados en cuanto a reutilización y estandarización dejan patente la ausencia de portabilidad tanto de los diseños como las implementaciones basadas en componentes. Este artículo, fundamentado en el *Desarrollo de Software Dirigido por Modelos*, presenta un enfoque que separa la descripción de aplicaciones de tiempo–real basadas en componentes de sus posibles implementaciones para distintas plataformas. Esta separación viene soportada por la integración automática del código obtenido a partir de los modelos de entrada en frameworks implementados usando tecnología orientada a objetos. Asimismo, se detallan las decisiones arquitectónicas adoptadas en la implementación de uno de estos frameworks, que se utilizará como caso de estudio para ilustrar los beneficios derivados del enfoque propuesto. Por último, se realiza una comparativa en términos de coste de desarrollo con otros enfoques alternativos.

Palabras Clave: Ingeniería del Software, Desarrollo de Software Basado en Componentes, Desarrollo de Software Dirigido por Modelos, Frameworks, Patrones de Diseño Software, Tiempo-Real.

1. Introducción

Los *Sistemas de Tiempo-Real* (**RTS** en sus siglas inglesas) son por su naturaleza sistemas heterogéneos que involucran distintas plataformas de ejecución. Estas plataformas deben adaptarse a distintos tipos de restricciones (temporales, energéticas, de interacción con el entorno, etc.) e integrarse con otros programas, infraestructura de comunicaciones (*middleware*), etc. En alguno de los dominios específicos de los RTS, como es el caso de la Robótica, el desarrollo de software es a día de hoy un arte más que una disciplina sistemática. Algunas de las causas subyacentes son (Chella et al., 2010): la *variabilidad* en el tipo de aplicaciones y componentes (tanto hardware como software) que se utilizan en el dominio; la *dificultad* en la reutilización al no estar claras las fronteras entre elementos arquitectónicos de distinto tipo (manejadores de dispositivos, algoritmia, middleware, etc.) y, por último la *falta de interoperabilidad entre herramientas* que utilizan en las diferentes fases de desarrollo.

En este contexto, el *Desarrollo de Software Basado en Componentes* (**CBSD** en sus siglas inglesas) (Szyperski, 2002) es una alternativa de diseño que favorece la reutilización de elementos software y facilita el desarrollo de sistemas a partir de elementos preexistentes. Un *Componente Software* es una unidad de composición con interfaces bien definidas y un contexto de uso explícito (Szyperski, 2002). Como se establece en (Lau and Wang, 2007), los principios del CBSD puede aplicarse en la práctica o bien considerando que los componentes son objetos, o bien considerando que son unidades arquitectónicas (Shaw and Clements, 2006). *CORBA Component Model* (OMG, 2006) es un ejemplo del primer caso, mientras que los *Lenguajes de Descripción de Arquitecturas* (**ADL**s en sus siglas inglesas) (Medvidovic and Taylor, 2000) son un ejemplos del segundo. Los beneficios de adoptar el primer enfoque (componentes como objetos) son los derivados de la amplia difusión de esta tecnología (herramientas, entornos de ejecución, etc.). Sin embargo, la orientación a objetos (**OO**) define unos mecanismos de interacción (invocación de método) que tienden a acoplar los objetos y a dificultar su reutilización. En contraposición, considerar los componentes como unidades arquitectónicas solventa las desventajas anteriores, ya que sólo interaccionan a través de sus puertos, reduciéndose el acoplamiento y facilitándose su reutilización. Por contra, se carece de herramientas de desarrollo y entornos de ejecución.

*Autor en correspondencia.
Correo electrónico: diego.alonso@upct.es (Diego Alonso)

doi:10.1016/j.riai.2012.02.010

Éste es el enfoque de diseño seguido por los autores de este trabajo en los último años (Iborra et al., 2009).

El desarrollo de aplicaciones basado en un enfoque CBSD, donde los componentes se consideren unidades arquitectónicas, requiere aún que se solucionen dos problemas: (**P1**) disponer de un conjunto de *técnicas, herramientas y métodos* que den soporte a dicho enfoque y (**P2**) definir e implementar una serie de mecanismos para *transformar los diseños CBSD en programas ejecutables* (es decir, un compilador para un lenguaje basado en componentes). Los autores de este artículo sostienen que el enfoque de *Desarrollo de Software Dirigido por Modelos* (en inglés **MDSD**) (Stahl and Völter, 2006) puede proporcionar los fundamentos teóricos y las herramientas que permiten solucionar los dos problemas anteriores. A pesar de su relativa novedad, MDSD ha demostrado resultados prometedores en dominios de aplicación como la automoción, la aviónica y la electrónica, entre otros (OMG, 2008). Estos resultados se han cuantificado en mejoras en cuanto al nivel de reutilización, la calidad del software obtenido, y en la reducción del tiempo de desarrollo de los productos.

Siguiendo el enfoque MDSD, un sistema puede ser representado a un cierto nivel de abstracción utilizando *Modelos* (Bézivin, 2005). Mediante una correcta selección de niveles de abstracción es posible separar los artefactos software independientes de la plataforma de ejecución de aquellos que no lo son. Los modelos son por tanto los artefactos principales que dirigen el proceso de desarrollo. Según (Bézivin, 2005), estos modelos son representaciones simplificadas de la realidad en los que los detalles no relevantes se abstraen, favoreciendo así tanto la comprensión como la comunicación de la realidad subyacente a dichos modelos. Los modelos se definen conformes a *Meta-Modelos*, que capturan los conceptos relevantes a un dominio de aplicación particular, así como las relaciones existentes entre ellos (es decir, representan la sintaxis abstracta del lenguaje). Las transformaciones entre modelos (Mens and van Gorp, 2006) son igualmente un mecanismo clave en MDSD, puesto que definen cómo dichos modelos serán interpretados y trasladados a otras representaciones en el mismo o diferente nivel de abstracción, hasta llegar finalmente a la generación de código en un lenguaje de implementación.

El problema P1 puede resolverse, al menos en parte, mediante la definición de meta-modelos con los que construir modelos de las aplicaciones usando el enfoque CBSD. Las transformaciones entre modelos proporcionan el mecanismo a partir del cual solucionar el problema P2. De cara a mantener la generalidad del enfoque, la solución adoptada para transformar los modelos de componentes a código ejecutable debería ser válida para un conjunto de aplicaciones que compartieran unas características mínimas. A nivel de implementación, los *Frameworks*[1] son los artefactos que proporcionan un mayor grado de reutilización de código y mayor flexibilidad, puesto que pueden considerarse aplicaciones semi-completas que se especializan para producir aplicaciones concretas (Fayad et al., 1999). Un framework captura las características comunes a muchas aplicaciones del dominio de interés y ofrece, al mismo tiempo, mecanismos de especialización y puntos de variabilidad con los que especificar las diferencias. El diseño de estos frameworks se hace utilizando *Patrones de Diseño*, que proporcionan soluciones recurrentes ampliamente probadas así como un vocabulario común que facilita la documentación y el uso de dichos frameworks. Por todo esto, en este artículo se propone el uso de frameworks para solucionar el problema P2.

Este artículo describe un enfoque de diseño que combina el uso de componentes, modelos y frameworks para el desarrollo de aplicaciones RTS, que es extrapolable a otros sistemas y dominios. En este sentido se cubre una deficiencia del enfoque CBSD derivada de la falta de herramientas de soporte y entornos de ejecución. Para ilustrar estas ideas se detalla la implementación de un framework y su aplicación al caso de estudio de un sistema robótico teleoperado. Con todo ello, el artículo se estructura como sigue. La Sección 2 resume el estado de la técnica y los trabajos más significativos, mientras la Sección 3 resume las principales aportaciones del trabajo. La Sección 4 detalla enfoque para el desarrollo de aplicaciones RTS, que es ilustrado en la Sección 5 con la implementación de un framework para el caso de estudio robótico. La Sección 6 compara el enfoque del artículo frente a otros similares, y por último la Sección 7 presenta las conclusiones.

2. Estado de la Técnica

En los siguientes apartados se describen y comparan brevemente algunos de los trabajos relacionados con las disciplinas de la Ingeniería del Software empleadas en este artículo, según se ha descrito en la Introducción.

2.1. Desarrollo Software Basado en Componentes

A pesar de los beneficios esperados de la adopción del enfoque CBSD en el desarrollo de software, no existe una evidencia plena de su aplicación a gran escala. Aunque CBSD fue concebido inicialmente para ser independiente de la plataforma de ejecución subyacente, finalmente esto no se cumple en la práctica. De manera general, es posible destacar los siguientes aspectos como susceptibles de mejora:

- No existe un consenso unánime sobre los conceptos, definiciones y reglas de construcción de sistemas software utilizando CBSD (Crnkovic et al., 2005). Una consecuencia directa es que los modelos de componentes software no son portables entre plataformas y menos aún el software derivado de dichos modelos. Estos problemas quedan patentes en el estudio realizado en (Lau and Wang, 2007), donde se presenta un resumen de los trece modelos de componentes más representativos y se concluye que "*en general, ninguno de los modelos [de componentes] actuales son idealmente adecuados para cumplir las promesas hechas desde CBSD*".

[1] Utilizaremos el término anglosajón "framework" dadas las connotaciones inherentes al término que habría que justificar artificialmente en el caso de haber utilizando un equivalente en castellano.

- Cada uno de estos modelos de componentes está ligado a una implementación específica, con lo que seleccionar uno de ellos implica ligarse a una implementación. Esto no es malo en sí mismo, siempre que la implementación exhiba las características que se esperan del diseño basado en componentes. Sin embargo, la implementación incluye características que no aparecen de forma explícita en el modelo de componentes (como gestión de la concurrencia) y que generalmente son desconocidas para el desarrollador de aplicaciones CBSD, que pierde así el control sobre algunas características de la aplicación.

Dentro de los modelos de componentes de propósito general cabe destacar Fractal (Blair et al., 2009), el modelo de componentes de CORBA (OMG, 2006), KobrA (Atkinson et al., 2001), SOFA 2.0 (Bures et al., 2007), SaveCCM (Carlson and Håkansson, 2006), y Koala (van Ommering et al., 2000), entre otros. Fractal proporciona un ADL para describir la arquitectura de las aplicaciones y un marco de implementación de componentes en Java y C/C++. SOFA proporciona un enfoque similar que permite la distribución y actualización de componentes, aunque limitado al lenguaje Java. El modelo CORBA CCM fue desarrollado para construir aplicaciones basadas en componentes sobre el middleware de comunicación CORBA, y proporciona un IDL para generar la estructura externa de los componentes y facilitar su integración en el middleware. Koala es el primer modelo de componentes aplicado al sector electrónico, desarrollado por Phillips y utilizado como base para desarrollar el software de sus productos de consumo. KobrA es una de las propuestas más populares, en la que se definen un conjunto de principios para describir y descomponer un sistema software siguiendo un enfoque descendente basado en componentes arquitectónicos. Pero en todos estos casos, la implementación del código del componente y su estructura interna sigue dependiendo completamente del desarrollador.

Un enfoque ligeramente distinto a los anteriores es el proporcionado por Cadena/CALM (Childs et al., 2006), que es un entorno Eclipse para el diseño de *Líneas de Producto Software* (**SPL** en inglés) basadas en componentes. Con Cadena, los diseñadores seleccionan primero el modelo de componentes destino y posteriormente el entorno facilita la definición de la SPL a partir de él. La selección tan temprana de la plataforma objetivo supone, a nuestro juicio, una desventaja del enfoque, puesto que obliga al usuario a adoptar desde el principio una tecnología de componentes. En nuestra propuesta, la selección de la plataforma se pospone lo máximo posible.

La mayoría de estos modelos contemplan únicamente el modelado estructural de la aplicación, dejando la implementación de la lógica de los componentes (y por tanto, su comportamiento) a la fase de codificación manual. SaveCCM supone una excepción, ya que incluye el modelado del comportamiento, si bien con restricciones derivadas de las características del dominio de los sistemas embebidos.

En el ámbito de los RTS se están obteniendo resultados muy prometedores con la aplicación del enfoque MDSD. Ejemplos significativos son la Red de Excelencia en Diseño de Sistemas Embebidos ArtistDesign (Artist-ESD, 2008-2011) para el diseño de sistemas empotrados, y el proyecto OpenEmbeDD (OpenEmbeDD, 2008-2011). Por otro lado, en el ámbito de la automoción la industria ha estandarizado AUTOSAR (Autosar, 2008-2011) para el desarrollo de vehículos siguiendo este enfoque.

2.2. *Frameworks de Componentes para Robótica de Servicio*

De manera más particular, en el contexto de la Robótica, ORCA (Brooks et al., 2007) define un framework de código abierto para el desarrollo CBSD de aplicaciones robóticas, que incluye un repositorio de componentes que proporcionan la funcionalidad típica del dominio. Los componentes ORCA son, sin embargo, librerías dinámicas, con lo que son dependientes de la plataforma de ejecución asociado al enfoque.

SmartSoft (Schlegel, 2006) integra en su modelo de componentes un enfoque de desarrollo dirigido por patrones de comunicación, que según los autores son suficientes para cubrir las interacciones típicas entre componentes. Además, es posible asociar información temporal a dichas interacciones, con lo que el despliegue de los componentes puede verificarse con respecto de las capacidades de la plataforma destino elegida.

GenoM (Bensalem et al., 2009; Dominguez-Brito et al., 2004) es un paquete de desarrollo basado en componentes en los que se encapsulan dispositivos hardware o algoritmos habituales en robótica. La comunicación entre componentes se realiza a través de regiones compartidas de memoria.

En el marco del proyecto Open RObot COntrol Software (OROCOS) (Bruyninckx, 2001) se definió un framework modular para el desarrollo de software también para sistemas robóticos. En OROCOS coexisten tres tipos de librerías que proporcionan la infraestructura y funcionalidad para desarrollar aplicaciones basadas en componentes, la cinemática y un conjunto de métodos Bayesianos reutilizables para el desarrollo de sistemas RTS. OROCOS permite comunicación síncrona y asíncrona, así como interfaces para comunicación distribuida.

Más recientemente, en (Hongxing et al., 2009) se define un framework basado en componentes para el diseño de arquitecturas robóticas con escasos recursos computacionales. Este framework es adecuado para situaciones en las que se requieran mecanismos de comunicación sencillos entre componentes, facilitando así el despliegue de la aplicación. Este trabajo representa un cambio significativo frente a las propuestas mencionadas anteriormente, todas ellas pensadas para sistemas con muchos recursos Hardware/Software.

2.3. *Tendencias en el Desarrollo de Frameworks*

Los frameworks son uno de los artefactos software que alcanzan mayores tasas de reutilización, cuyo desarrollo ha sido ampliamente estudiado (Fayad et al., 1999). Recientemente, han aparecido en la bibliografía nuevas propuestas más generales e innovadoras, enfocadas al desarrollo y uso de frameworks para el desarrollo de sistemas software en general (Fairbanks et al., 2006; Antkiewicz and Stephan, 2009).

En (Fairbanks et al., 2006), los autores proponen un método para la especialización de frameworks OO utilizando patrones

de diseño, que proporciona un *fragmento de diseño* al sistema en conjunto. Así, un fragmento de diseño es una solución probada a cómo el programa debe interactuar con el framework de cara a cumplir un objetivo. La idea es que cada framework disponga de su catálogo particular de fragmentos de diseño, recogiendo soluciones convencionales a problemas conocidos. La propuesta se valida con una herramienta basada en Eclipse que integra más de cincuenta patrones.

En (Antkiewicz and Stephan, 2009) se da un paso adicional al proporcionarse un marco conceptual y metodológico para la definición y uso de *lenguajes para el modelado específico de frameworks* (FSML, en sus siglas inglesas). Un FSML es una representación explícita de los conceptos específicos de un dominio tal y como los ofrece el framework asociado. Así, los FSMLs pueden ser utilizados para expresar modelos de aplicación "*específicos de frameworks*". Estos modelos describen las instancias de los conceptos (*features*) proporcionados por el framework que son finalmente implementadas en el código de la aplicación. Estos dos últimos trabajos señalan de manera muy clara la tendencia en el desarrollo de aplicaciones basadas en frameworks para los próximos años, por lo que deberán tenerse en cuenta en el planteamiento que se haga.

A modo de resumen, aquellos frameworks sostenidos sobre plataformas y lenguajes OO están limitados en escalabilidad y extensibilidad debido a las restricciones derivadas del uso de los objetos (Parsons et al., 2006). Por otro lado, los frameworks basados en componentes mejoran dicha flexibilidad y extensibilidad, pues ofrecen mecanismos que facilitan la reconfiguración dinámica de componentes incluso en tiempo de ejecución. Con esto, los frameworks basados en componentes facilitan la integración e interoperabilidad, si se construyen de manera que tengan en cuenta futuras extensiones con la adición de nuevos componentes (propios o de terceros).

2.4. Metodologías de Desarrollo de RTS

Existe una gran variedad de metodologías de análisis y diseño, desde las basadas en los principios de la programación estructurada hasta las más actuales orientadas a objetos. Entre ellas destacan las metodologías de propósito general definidas por Booch (Grady Booch et al., 2007), Jacobson (Jacobson, 1992) y Catalysis (D'Souza and Wills, 1998). Más concretamente, y centradas en el diseño de RTS, pueden mencionarse *Hierarchic Object-Oriented Design* (HOOD/HRT-HOOD) (Burns and Wellings, 1995), y las definidas por Gomaa, método *Concurrent Object Modeling and Architectural Design Method* (COMET) Gomaa (2000), y Douglass, método *Rapid Object-Oriented Process for Embedded Systems* (ROPES) (Douglass, 2004). En este punto merece la pena volver a mencionar KobrA (Atkinson et al., 2001), puesto que, a diferencia de las anteriores, propone una metodología centrada en el diseño con componentes arquitectónicos bien documentada. En la mayoría de ellas, el proceso está dirigido por casos de uso y el diseño sigue un desarrollo iterativo e incremental.

El enfoque propuesto en este artículo no impone una metodología de diseño, pudiendo utilizarse cualquiera de las anteriores adaptando el paradigma OO al CBSD. De hecho, estas metodologías podrían integrarse en dicho enfoque puesto que no son incompatibles con él, sino más bien ortogonales. Un componente arquitectónico tiene una interfaz más elaborada que un objeto, y en cierto sentido más restrictiva, ya que los componentes sólo pueden interaccionar a través de sus puertos. El beneficio que se obtiene de aplicar estas restricciones de diseño son un menor acoplamiento entre componentes y una mayor cohesión de la estructura interna de los mismos. Pero a parte de esto, las metodologías que se aplican en OO pueden también aplicarse en CBSD, siempre que tengan en cuenta estas restricciones a la hora de realizar la fase de diseño de detalle.

Por ejemplo, los criterios de COMET para realizar el diagrama de contexto de una aplicación son perfectamente aplicables a un diseño a nivel de componentes arquitectónicos como los descritos en (Lau and Wang, 2007) en lugar de objetos. Los criterios de COMET para agrupación de procesos puede aplicarse para desplegar el código de las aplicaciones realizadas a partir del framework. ROPES es una metodología basada en el empleo de patrones de diseño OO que, al igual que COMET, puede utilizarse tanto para el desarrollo de las aplicaciones basadas en componentes como para el diseño de los frameworks que les proporcionan el soporte a la ejecución. Por otro lado, RT-HOOD es una metodología más específica de RTS y más centrada en los procesos que forman la aplicación.

3. Aportaciones del Trabajo

El trabajo que se presenta en este artículo contribuye al estado de la técnica detallado en la sección anterior fundamentalmente por los siguientes aspectos:

- Estando basado en MDSD, los modelos son los artefactos que soportan la propuesta. El uso de modelos, además de aumentar el nivel de abstracción, permite integrar otros modelos para realizar actividades de V&V temprana, así como retrasar la elección de las características de la plataforma dadas las posibilidades de generación código de manera automática.
- El modelado de aplicaciones es puramente CBSD, es decir, el modelador no tiene por qué conocer detalles de implementación del framework sobre el que finalmente se soporte la plataforma.
- El desarrollador percibe cada framework desde un punto de vista meramente conceptual, a través del conjunto de características que lo definen. Estas características podrán instanciarse como parte del proceso de especialización del framework. Así, la elección de un framework u otro vendrá condicionada por el conjunto de características que se ofrezcan desde el mismo (por ejemplo, distribución de componentes frente a un esquema centralizado, esquemas de comunicación, etc.).
- La transformación oculta la complejidad de framework al desarrollador. Esto supone una gran ventaja, puesto que en general los frameworks tienen asociado una curva de aprendizaje larga.

- Se realiza una estimación del coste de desarrollo y evolución de aplicaciones siguiendo el enfoque propuesto, comparándolo con otros enfoques similares.

4. Descripción del Enfoque de Desarrollo

En un enfoque de MDSD, la generación de código es normalmente pospuesta hasta las etapas finales del proceso, en las que se ejecuta una transformación modelo-a-texto sobre un modelo refinado que contiene gran parte de los detalles de implementación. Esta transformación debe, por tanto, generar prácticamente todo el código de la aplicación, lo que la convierte en un artefacto software de cierta complejidad, difícil de entender, de mantener y de hacer evolucionar. Además, las transformaciones tienen una estructura no modular que impide su reutilización (total o parcial) en sistemas que pudieran tener requisitos similares. Éste es el enfoque que generalmente se sigue al aplicar MDSD y que siguieron los autores de este artículo en el trabajo descrito en (Alonso et al., 2010), que si bien resultó interesante para la puesta en práctica de MDSD en el ámbito robótico, ha dejado patente la necesidad de investigar nuevas soluciones que promuevan la reutilización. Es por tanto necesario un enfoque:

- Que ofrezca facilidades para el desarrollo de aplicaciones que, por pertenecer al mismo dominio, comparten características similares.
- Que reduzca la complejidad de las transformaciones modelo-a-texto, de manera que se favorezca el desarrollo, la reutilización y evolución de las mismas.
- Que minimice el acoplamiento entre la interpretación OO de los componentes y su infraestructura de ejecución.
- Que proporcione una implementación en código que sea fiel al modelo CBSD original, tanto en su estructura como en su comportamiento.

La Fig. 1 muestra una propuesta para solucionar estos problemas, que engloba e integra los enfoques CBSD y MDSD en un contexto de desarrollo basado en el uso de frameworks. En ella pueden diferenciarse dos tipos de rol, el *desarrollador de frameworks* y el *desarrollador de aplicaciones*:

Desarrollador de frameworks. A partir de un enfoque de ingeniería de dominio, extrae y organiza un conjunto de características comunes y específicas de las aplicaciones del dominio de interés. A partir de estas características, diseña e implementa un framework que proporciona (1) la infraestructura de ejecución para soportar estas aplicaciones, (2) una interpretación de los conceptos CBSD en tecnología OO (puesto que es la tecnología de implementación seleccionada) y (3) un conjunto de *'hot-spots'* para dar soporte a la variabilidad de las aplicaciones y su evolución. Opcionalmente, puede dotar al framework de una infraestructura de configuración para que el usuario del mismo modifique, dentro de ciertos límites, la aplicación final.

Desarrollador de aplicaciones. A partir de un conjunto de requisitos (funcionales y no funcionales) diseña la aplicación específica mediante un lenguaje de modelado orientado a componentes arquitectónicos que . Este lenguaje debe permitir definir tanto la estructura como el comportamiento de los componentes. Lenguajes con la capacidad expresiva suficiente son UML 2, y particularmente el profile MARTE (OMG, 2009). En cualquier caso, debe elegirse un lenguaje que permita modelar la estructura y el comportamiento de los componentes. Cada estado de dicha máquina puede tener asociado un algoritmo, descrito mediante un diagrama de actividad. Una vez modelada la aplicación, el desarrollador selecciona el framework que aporta las características requeridas por el dominio, rellenando los *'hot-spots'* del framework y utiliza opcionalmente la infraestructura de configuración final de la aplicación.

En este punto, debe tenerse en cuenta que el modelo de componentes construido por el desarrollador de aplicaciones considera aquellos requisitos no funcionales que pueden representarse explícitamente en la arquitectura. Por ejemplo, la fiabilidad se puede conseguir mediante componentes redundantes, que aparecen explícitamente en el modelo. Por otro lado, el framework debe soportar todos los requisitos que pueden expresarse en el modelo arquitectónico más aquellos que no pueden representarse explícitamente en él. Por ejemplo, suponiendo que hubiera requisitos de tiempo real, aunque a nivel arquitectónico no se definen hilos de ejecución, se asume que los hilos que proporciona el framework van a poder analizarse para verificar la planificabilidad de la aplicación OO generada. Ésta y otras características serán las que guíen al desarrollador en la elección de un framework frente a otros.

Con este propósito, se organizó del código en tres grupos (véase la Fig. 1): (**C1**) código que proporciona el soporte de ejecución conforme a los requisitos del dominio; (**C2**) código que proporciona una interpretación OO de los conceptos CBSD así como los 'hot-spots' del framework, y (**C3**) código específico de cada aplicación.

La idea es que C1 y C2 sean implementados manualmente por el desarrollador de frameworks, mientras que el código C3 sea generado e integrado automáticamente mediante transformaciones de modelos. Al generarse C3 a partir de modelos independientes de la plataforma de ejecución se acuña la idea de que "*los modelos rellenan los hot-spots de los frameworks*". Es decir, el desarrollador de aplicaciones percibe al framework desde un punto de vista conceptual e independiente de la plataforma como un conjunto de 'hot-spots' que van a ser rellenados a partir del modelo de componentes arquitectónicos de la aplicación.

El código C2, además de proporcionar la interpretación OO de los conceptos CBSD, reduce el acoplamiento existente entre C1 y C3, permitiendo la evolución separada de ambos. De esta forma será posible reutilizar el mismo soporte de ejecución C1 en distintas aplicaciones, siempre que C2 se mantenga constante. Al mismo tiempo, se facilita la reutilización de la misma aplicación (modelo de componentes)

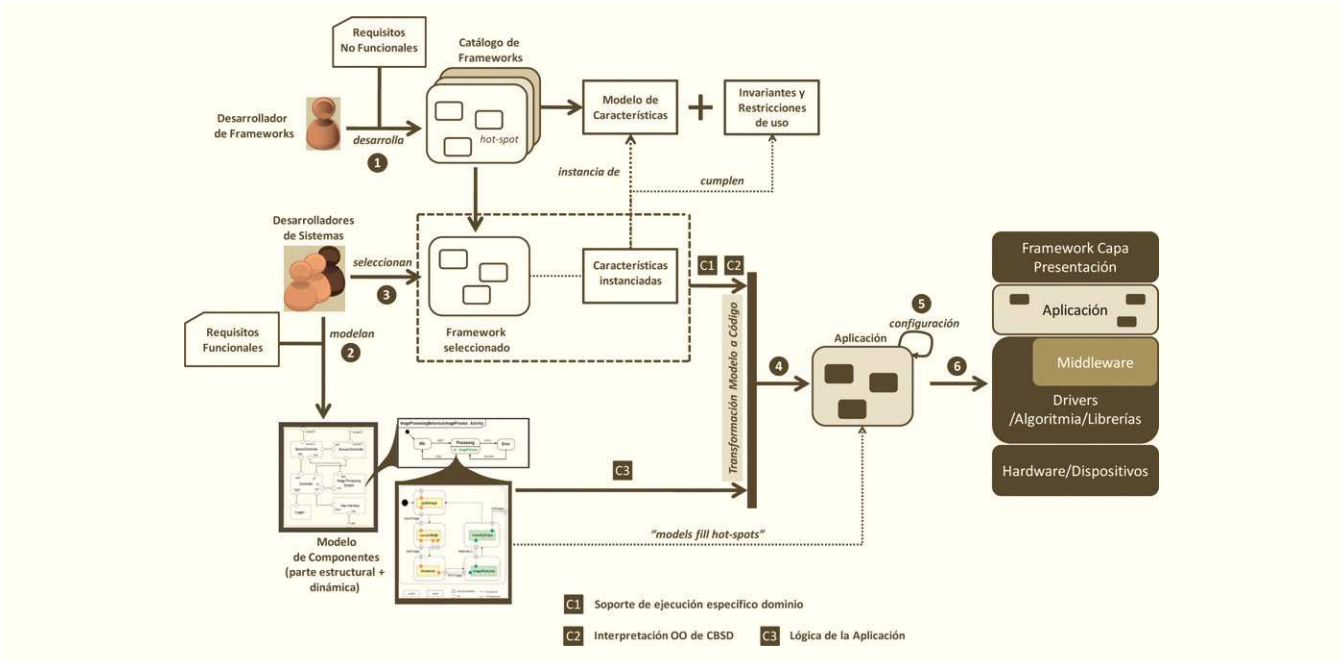

Figura 1: Enfoque de desarrollo de CBSD propuesto, que se apoya en MDSD para realizar el diseño y la validación de las aplicaciones, y en frameworks para la generación de código. Se aprecian dos tipos de usuarios: el desarrollador de aplicaciones y el desarrollador de frameworks.

con otros requisitos de ejecución (mismo C3, distinto C1). Esta flexibilidad se obtiene a costa de introducir cierto acoplamiento entre C1 y C2.

El enfoque propuesto considera la posibilidad de integrar la aplicación obtenida en un 'front-end' (paso 5 en la Fig. 1) que permita la configuración y parametrización de aquellos aspectos que no se pueda o no convenga definir a nivel arquitectónico. En este ajuste fino, se considera, por ejemplo, variar la distribución de la carga de ejecución de las actividades entre los distintos hilos concurrentes desplegadas, analizar la planificabilidad de la aplicación resultante, etc.

Con el propósito de poder integrar y reutilizar software pre-existente, este enfoque posibilita las siguientes extensiones:

- La integración de frameworks desarrollados por terceros (por ejemplo, para proporcionar una capa de presentación gráfica al estilo de Java-Swing, para tener soporte a la comunicación distribuida, etc.). En este caso, el desarrollador de aplicaciones deberá incluir en el modelo CBSD de la aplicación un componente 'wrapper' que represente la interconexión con dicho framework. Este 'wrapper' encapsula la funcionalidad de terceros y explicita al mismo tiempo las interconexiones con los componentes propios de la aplicación.
- La invocación de librerías y 'drivers' específicos del dominio (motores, controladores, sensores, etc.). Esto se consigue a partir de llamadas explícitas referenciadas en las actividades del modelo de componentes. La transformación modelo-a-texto las tendrá en cuenta y enlazará con las funciones referenciadas.

5. Caso de Estudio Demostrativo

Esta sección describe paso a paso e ilustra con un ejemplo la aplicación del enfoque descrito anteriormente. Para ello, primeramente se describe un framework desarrollado conforme al enfoque y posteriormente se demuestra su uso en la implementación de un sistema robótico representativo.

5.1. Diseño de un Framework para RTS

En esta sección se describen brevemente las principales características del primer framework desarrollado según la propuesta descrita. Este framework se desarrolló para ser aplicado en el dominio de los RTS. Una versión preliminar del diseño se describe en (Pastor et al., 2010).

Características del framework. Las características fundamentales que permiten catalogar este framework son: (1) control sobre la política de concurrencia (por ejemplo, una actividad por hilo, un único hilo para todas las actividades, o cualquier otra distribución), (2) control sobre la política de distribución de los componentes en nodos computacionales, (3) control sobre los mecanismos de comunicación entre componentes (síncrona o asíncrona). De todos ellos, el requisito principal que guía el diseño del framework es que debe proporcionar al usuario gran control sobre la concurrencia de la aplicación (tanto en el número de hilos como en sus características temporales), de forma que se pueda verificar el cumplimiento de los requisitos temporales. Para ello, se decidió realizar un diseño que permitiera asignar arbitrariamente las actividades que ejecutan los componentes que forman la aplicación a un conjunto cualquiera de hilos, tal y como se muestra en la Fig. 2. Hay que destacar en este punto que el

framework no proporciona ninguna guía sobre el número de hilos que se deben crear o la forma de realizar la asignación de actividades a hilos, sino que proporciona los mecanismos necesarios para que sea el usuario el que elija los heurísticos apropiados, por ejemplo los que se definen en (Gomaa, 2000).

Diseño del framework. El diseño y la documentación del framework se ha llevado a cabo mediante el uso de patrones de diseño, lo cual viene siendo una práctica habitual (Brugali and Menga, 2000) en la *Ingeniería del Software*. Cada patrón de diseño describe un punto de decisión en el diseño de una aplicación. Las relaciones de dependencia entre todos los patrones involucrados en el diseño de un framework pueden representarse mediante un grafo (Buschmann et al., 2007b). Este grafo contribuye a documentar su diseño y facilitar su uso. La Fig. 3 muestra la secuencia de patrones aplicados al diseño del framework aquí descrito.

De actividades a hilos. Las actividades asociadas a los estados de la máquina de estados de un componente se traducen a objetos siguiendo el patrón COMMAND. Estas actividades, que tienen asociadas unas características temporales, tienen que asignarse a hilos, de forma que se cumplan dichas características. Estos hilos se modelan mediante el patrón arquitectónico COMMAND PROCESSOR (Buschmann et al., 2007a), que separa la petición de un servicio de su ejecución. Esta separación proporciona el desacoplamiento necesario para poder asignar las actividades de los estados a diferentes hilos de ejecución. El procesador de comandos ha sido restringido para facilitar el análisis de los RTS, evitando que en tiempo de ejecución se pueda (1) crear nuevos hilos, (2) modificar la distribución de las actividades en hilos, y (3) modificar los periodos o las prioridades de los hilos. Este patrón arquitectónico utiliza otros patrones de menor entidad, como COMMAND (para modelar la petición de un servicio de un componente y la ejecución de la actividad asociada a ella) y STRATEGY (para definir el heurístico de despacho de las peticiones de servicio encoladas). Como caso especial se utiliza también el patrón NULL OBJECT para modelar estados en los que no se ejecuta ninguna acción, por ejemplo, cuando se modela un protocolo.

Máquinas de estados. El siguiente problema que se resolvió fue el modelado de la estructura de las máquinas de estado que describen el comportamiento de los componentes y la semántica de ejecución de las mismas. La estructura se ha modelado mediante el patrón COMPOSITE, que representa la relación existente entre regiones ortogonales y estados hoja. En cuanto a la semántica de ejecución, se sigue la que define el estándar UML 2, "*ejecución-hasta-compleción*". Esta semántica está implementada en una subclase dentro de la jerarquía que define el patrón COMMAND descrito anteriormente, que gestiona las regiones ortogonales de la máquina de estados. Esta clase utiliza el patrón TEMPLATE para definir la secuencia de acciones que implementan la semántica de ejecución. En caso de que se quiera cambiar dicha semántica, bastará con cambiar la implementación de dicha clase.

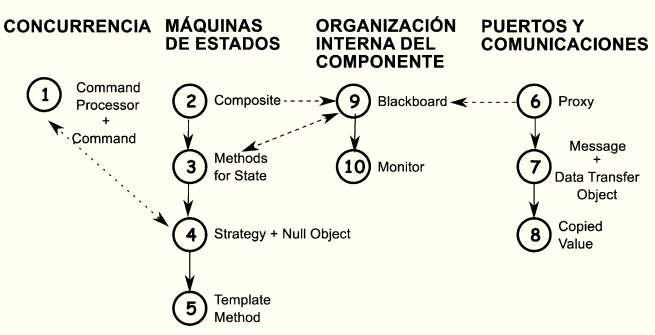

Figura 3: Grafo de patrones involucrados en el diseño del framework. Los nodos representan patrones de diseño (puntos de decisión) y los arcos representan la secuencia temporal en que fueron considerados.

Estado del componente. El estado global del componente está almacenado en una estructura tipo pizarra utilizando el patrón BLACKBOARD, local para cada componente. Esta pizarra almacena la estructura interna del componente y los datos relacionados con su estado, como la estructura de la máquina de estados (según el patrón METHODS FOR STATE), las colas de eventos recibidos por el componente, el estado de los puertos, las colas de mensajes recibidos y mensajes que quedan por enviar, etc. Puesto que las actividades de un componente pueden estar "repartidas" entre varios hilos, la pizarra está realmente formada por un conjunto de monitores que agrupan los datos relacionados y que proporcionan exclusión mutua en el acceso concurrente a los mismos. De esta forma, se evita la corrupción de dichos datos, y por tanto, del estado del componente. Se ha elegido crear una pizarra de monitores de datos y no un monitor para una pizarra de datos con la objetivo de evitar bloqueos excesivos cuando distintos hilos intentan acceder a datos que no están estrechamente relacionados entre sí, por ejemplo, la lectura/escritura de eventos y la consulta o el cambio de estado.

Puertos del componente. Los puertos se implementan utilizando el patrón PROXY usando dos clases genéricas: puerto de entrada (interfaces ofrecidas) y puerto de salida (interfaces requeridas), cuyo parámetro genérico es el tipo de mensaje que intercambian. En caso de que el componente incluya un puerto con interfaces ofrecidas y requeridas en el diseño CBSD, éste se traduce en dos puertos, uno de entrada y otro de salida. Las interfaces presentes en el modelo CBSD y sus combinaciones se traducen en los tipos de datos que se utilizan como parámetros genéricos en la instanciación de los puertos. La comunicación entre puertos se realiza mediante intercambio de mensajes, según el patrón MESSAGE, ya que facilita la distribución de componentes. Los mensajes entrantes y salientes se almacenan en colas de mensajes, contenidas en la pizarra mencionada anteriormente. La semántica de comunicación por defecto entre los puertos es comunicación asíncrona sin respuesta ya que, (1) es la que proporciona menor acoplamiento entre componentes (los componentes nunca ejecutan código de otro componente ni se quedan bloqueados esperando una respuesta), y (2) es posible implementar llamadas síncronas a partir de llamadas asíncronas. Esta semántica está implementada en una subclase dentro de la jerarquía que define el patrón COMMAND descrito anteriormente, que gestiona la comunicación de los puertos. Por último, también se utilizan los patrones COPIED

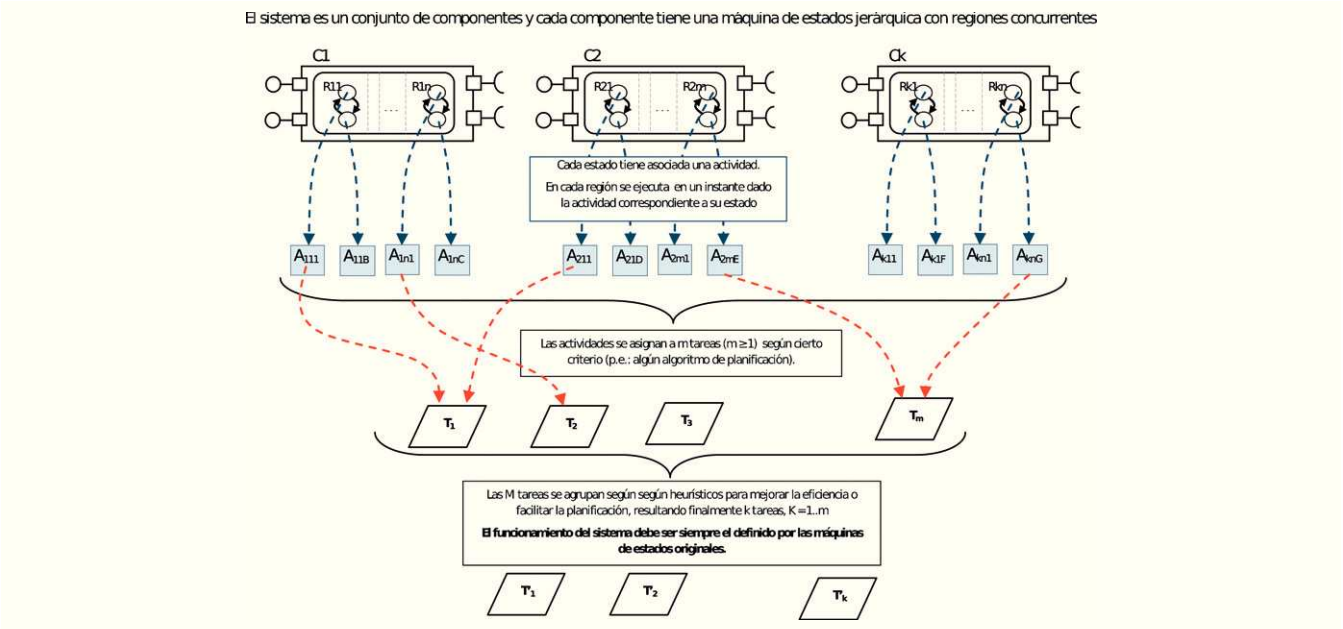

Figura 2: Control de la política de concurrencia: asignación de actividades a hilos.

VALUE para pasar copias de un mensaje en lugar del original, así como DATA TRANSFER OBJECT para reducir el número de llamadas entre objetos empaquetando grupos de atributos en un solo objeto que se pasa o se retorna en una única llamada.

Considerar las actividades de gestión de las regiones ortogonales de la máquina de estados y de gestión de los puertos como actividades normales dota de regularidad a la estructura global del sistema y las hace explícitas, permitiendo al usuario elegir los hilos con las características temporales adecuadas a los requisitos de dichas actividades. Por ejemplo, se pueden asignar las actividades de gestión de los puertos asociados a un controlador tipo PID y de la región que modela su estado a un estado de alta prioridad para asegurar el cumplimiento de sus requisitos temporales, mientras que la gestión de regiones y puertos asociados a servicios "menos prioritarios" pueden asignarse a hilos de mayor periodo.

En la versión actual del framework, sólo se pueden añadir a un Command Processor las actividades de gestión de una región ortogonal, las cuales delegan en la actividad asociada al estado hoja activo para ejecutar el código correspondiente. En términos prácticos no es una limitación, ya que por definición no se pueden ejecutar de forma concurrente las actividades correspondientes a dos estados contenidos en una misma región ortogonal. Con este esquema se puede conseguir un grado máximo de concurrencia de un hilo por región ortogonal, mientras que el mínimo es un hilo para ejecutar todo el código.

Este diseño se ha implementado de forma manual en el lenguaje Ada 2005. Para dar una idea de su complejidad, C1 está formado por dos clases mientras que C2 integra un total de treinta elementos, entre clases e interfaces. Una vez implementado, este framework se incorpora al catálogo de frameworks, estando disponible para futuros desarrollos.

5.2. *Caso de Estudio de Aplicación del Framework*

Para demostrar la aplicación del enfoque se considera una aplicación de teleoperación para un robot Pioneer. Por razones de espacio y para no introducir complejidad innecesaria, se reduce la funcionalidad de la misma a los componentes arquitectónicos indicados en la Fig. 4: una interfaz gráfica de usuario (*GUI*) desde la que se envían comandos y que se actualiza con información de estado, un componente intermedio (*PathPlanner*) que calcula la trayectoria que debe seguir el vehículo y un componente que hace de interfaz con los accionamientos y sensores del vehículo (*HAL*).

En el ejemplo, la GUI debe refrescarse a una tasa aceptable para un operador humano y transmitir los comandos al componente PathPlanner en un tiempo menor que el tiempo de reacción típico de un operador humano. Se estima que una actualización cada 150 ms es aceptable. El PathPlanner debe actualizar las acciones de control de los motores cada 10 ms. Finalmente, las actividades del HAL, relacionadas con los hilos de control de más bajo nivel, deben ejecutarse cada 1 ms.

El desarrollador de aplicaciones, a partir de los requisitos anteriores y de la particularidad de las aplicaciones del dominio, busca en el catálogo un framework que proporcione las siguientes características:

- Control sobre los hilos del framework y sus prioridades.
- Posibilidad de analizar la planificabilidad de los hilos.
- Posibilidad de distribuir los componentes.
- Comunicación asíncrona sin respuesta.

El framework descrito en la sección anterior cumple con estos requisitos y se selecciona como plataforma de destino.

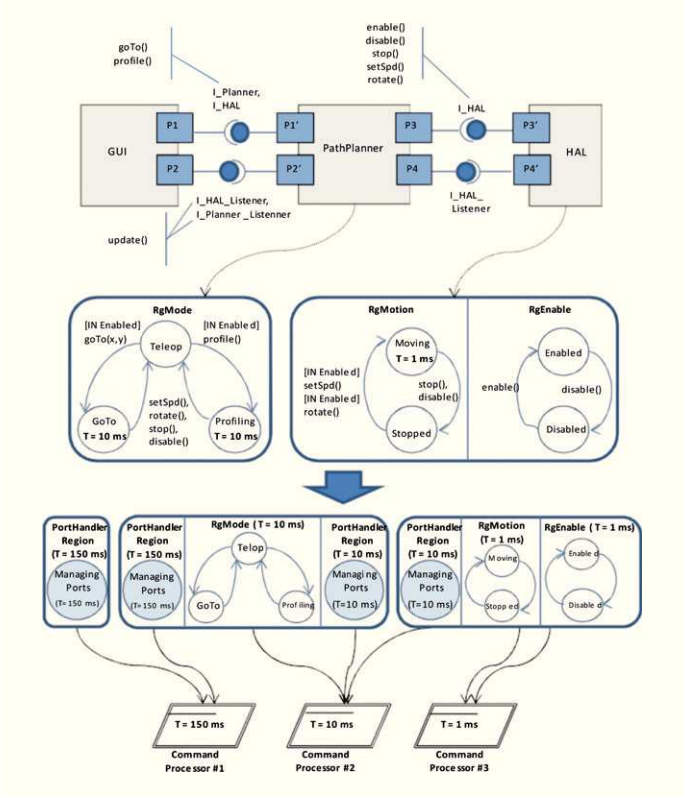

Figura 4: Caso de estudio de aplicación del enfoque para un robot Pioneer.

El comportamiento reactivo de los componentes se define mediante máquinas de estados con regiones ortogonales. No hay ningún comportamiento reactivo asociado a la GUI pues se limita a solicitar los servicios del PathPlanner y a actualizarse cuando éste le envía datos. El comportamiento del PathPlanner contempla tres casos, o bien teleoperación pura (estado Teleop, en el que se reenvían los comandos de la GUI al HAL), o seguir una trayectoria (estado GoTo), o seguir el contorno de un objeto (estado Profiling). La llegada de cualquier comando de la GUI, o de un evento de colisión o uno de deshabilitación desde el HAL provoca el paso a teleoperación. No hay ninguna actividad asociada a la teleoperación, mientras que las actividades asociadas a los otros estados (no incluidas por simplificar el ejemplo) se limitan a comprobar que el robot sigue la consigna y a corregir la trayectoria en caso contrario.

El comportamiento del HAL se modela mediante dos regiones ortogonales. La región RgEnable describe el estado de habilitación del hardware, mientras que la región RgMotion describe el estado de movimiento del robot. De los cuatro estados hoja incluidos en dichas regiones, sólo el estado Moving tiene asociado una actividad de control del movimiento. Las actividades asociadas a los estados se definen mediante diagramas de actividad, en los que pueden insertarse llamadas a librerías software. Los estados que ejecutan actividades periódicas han sido marcados con el periodo correspondiente.

La transformación genera, a partir del modelo de componentes, el código C3 partiendo de los 'hot-spots' definidos en C2. Conviene destacar que la transformación genera los siguientes elementos, que no están presentes en el modelo inicial (véase Fig. 4): (1) regiones ortogonales adicionales para gestionar los puertos con (2) un estado con una actividad asociada de gestión de los puertos que actualiza los datos internos del componente, y (3) una actividad de gestión por cada región ortogonal, incluidas las regiones creadas en (1), que implementa la semántica de ejecución de las máquinas de estados. Es relevante destacar que las regiones, estados y actividades añadidos por la transformación se van a tratar exactamente de la misma manera que aquellos generadas a partir del modelo. Esto facilita, por la regularidad del diseño, determinar la influencia que tienen en la planificabilidad de la aplicación final. El periodo de las actividades de gestión de los puertos debe configurarse una vez concluida la transformación, y debe ser igual o inferior al menor periodo de todas las actividades que pueden ser ejecutadas como consecuencia de la recepción de un mensaje a través del puerto. La transformación puede parametrizarse para crear más o menos regiones de este tipo en función del periodo con que los puertos deban leerse (entradas) o actualizarse (salidas). Por otro lado, el periodo asignado a la actividad de gestión de una región ortogonal debe ser menor o igual que el menor de los periodos de cualquiera de las actividades de sus estados.

Una vez ejecutada la transformación se dispone de una aplicación que por defecto crea un único hilo (Command Processor), al que se asignan todas las actividades, aunque puede ser modificada manualmente a criterio del desarrollador. La Fig. 4 muestra la asignación final, en la que se consideran tres hilos. Como puede observarse, las actividades que gestionan los puertos que conectan la GUI con el PathPlanner se asignan a un hilo, la actividad que gestiona el funcionamiento de la región del PathPlanner y las que gestionan los puertos que conectan el PathPlanner con el HAL a otro hilo, y el resto de actividades de gestión de las regiones del HAL al tercer hilo.

Para dar una idea del tamaño del código resultante tras la ejecución de la transformación, el conjunto C3 para el caso del componente HAL está formado por siete clases (especializaciones de los 'hot-spots') y por once instancias de clases definidas por el framework en C2. Además, se crean también tres instancias correspondientes a los tres hilos (Command Processor) y una clase que representa la aplicación.

En el ejemplo se puede apreciar cómo las actividades de regiones de diferentes componentes pueden asignarse a un mismo hilo, y que actividades de un mismo componente pueden asignarse a diferentes hilos. Será el desarrollador de aplicaciones el que finalmente elija el número de hilos y la distribución de actividades que cumplan los requisitos temporales expresados en el modelo arquitectónico.

6. Análisis Comparativo Basado en el Coste de Desarrollo

Esta sección compara el enfoque propuesto en este artículo con otros dos enfoques alternativos de desarrollo CBSD que también incluyen el uso de MDSD, puesto que MDSD aporta características extra más allá de servir de tecnología de soporte para CBSD. Concretamente, se van a comparar:

Enfoque 1: enfoque en el que primero se selecciona la plataforma y posteriormente se realiza el diseño CBSD (por ejemplo Cadena/CALM (Childs et al., 2006)).

Enfoque 2: enfoque basado en un ADL, en el que la transformación de modelos genera todo el código de implementación. Es decir, un enfoque en el que la arquitectura de la aplicación, y por tanto todos sus atributos de calidad están implícitos en dicha transformación. Este enfoque ha sido utilizado por los autores en los últimos años (Iborra et al., 2009).

Enfoque 3: enfoque propuesto en este artículo, también basado en un ADL, pero en el que existe un framework que proporciona la plataforma y el soporte de ejecución.

El enfoque (2) es típico en la aplicación de MDSD, en el que se parte de un lenguaje con cierto nivel de abstracción y posteriormente se definen una serie de transformaciones para generar el código de implementación. El enfoque (3) comparte características con los enfoques (1) y (2): como el (1) asume un entorno de ejecución pre-existente (el framework), y como el (2) pospone su elección a las etapas finales del proceso (selección de un framework y una transformación concretos).

Cadena, enfoque (1), presenta bastantes similitudes con el enfoque aquí descrito. De hecho, podría utilizar el framework que se describe en este artículo como una plataforma destino más, siempre que soportara el modelado del comportamiento de los componentes. Sin embargo, hasta donde saben los autores, Cadena se limita al modelado de los aspectos estructurales y a la generación de los esqueletos de código correspondientes. En caso de que Cadena soportara también el modelado del comportamiento y la generación del código, el coste de desarrollo de aplicaciones sería similar al de la propuesta descrita en este artículo. Por consiguiente, el resto de la sección se centra en la comparación entre los enfoques (2) y (3).

Existen varias formas de estimar los costes de desarrollo del software, según se recoge en (Sommerville, 2010). En este caso, se ha seleccionado el *Modelo Constructivo de Costes* (o COCOMO, por su acrónimo del inglés) (Boehm et al., 2000). COCOMO es un modelo matemático de base empírica utilizado para estimación de costes de desarrollo de software que ofrece tres niveles de detalle. En esta sección se utiliza el modelo básico, que estima el esfuerzo de desarrollo de software (`E`) en personas–mes según la ecuación $E = a * S^b$, donde:

- `a` y `b` son dos parámetros empíricos que dependen del tamaño del software desarrollado y de la estructura de la propia organización. A efectos de esta comparativa se consideran iguales en ambos enfoques.
- `S` es una medida del tamaño de los artefactos software involucrados en el proceso de generación de las aplicaciones. Éste es el parámetro que más varía entre los enfoques (2) y (3), y por tanto en el que se van a centrar los razonamientos. Debido a que el código final se genera de forma automática a partir de los modelos de partida, se considera que `S` depende del tamaño de las transformaciones y del framework (medidos por ejemplo en líneas de código equivalentes), no del tamaño del software que se genera a partir de ellos.

En el enfoque (2), el término `S` es equivalente al tamaño de la transformación ($E_2 = a * S_{T2}^b$), mientras que en el enfoque (3) se puede dividir en el tamaño del framework más el de la transformación ($E_3 = a * (S_{T3} + S_F)^b$). El enfoque (2) implica el desarrollo de transformaciones que tienen que conseguir que el código resultante sea conforme tanto al modelo de partida como a los requisitos no funcionales de la aplicación. En el enfoque (3), una vez elegido un framework destino, la transformación sólo tiene que especializarlo para obtener la aplicación conforme al modelo de partida, puesto que los requisitos no funcionales son aportados por el framework. El enfoque (3) requiere, por tanto, transformaciones más pequeñas y menos complejas.

El coste de *desarrollo inicial* es similar, en cuanto a orden de magnitud, en ambos casos, ya que tanto la transformación S_{T2} como el desarrollo del framework S_F son tareas exigentes en cuanto a diseño, implementación y pruebas. Aún así, el enfoque (3) tiene la ventaja respecto del (2) de permitir la división del trabajo de desarrollo entre personal experto en transformaciones y personal experto en la tecnología de implementación del framework, posibilidad no presente en el enfoque (2). Es más, si desde un principio se dispusiera de un framework que proporcionara el soporte de ejecución compatible con los requisitos de la aplicación, el coste de desarrollo inicial sería mucho menor adoptando el enfoque (3).

Sin embargo, el enfoque (3) muestra realmente su potencial cuando se realizan tareas de *mantenimiento y evolución*. Esto es debido a que, con la tecnología actual, modificar o extender un framework requiere un menor esfuerzo de desarrollo que modificar o extender transformaciones de modelos. Existe una gran experiencia bien documentada sobre patrones de diseño aplicables para proporcionar flexibilidad a un framework respecto a multitud de variables (Buschmann et al., 2007a,b). Sin embargo, por el momento no existe una experiencia equivalente y aplicable a las transformaciones de modelos. Éstas están basadas todavía en técnicas de desarrollo que no favorecen el diseño modular, creando transformaciones cuyas reglas suelen exhibir un alto grado de acoplamiento entre sí. Todo esto da lugar a que las transformaciones de modelos sean, en la mayoría de los casos, artefactos monolíticos.

Dicho esto, los cambios en los requisitos de partida de una aplicación CBSD pueden ser categorizados en uno de los siguientes grupos:

- *Cambios en la funcionalidad*. Afectan únicamente al diseño arquitectónico de la aplicación. Requerirán el diseño de nuevos componentes a nivel CBSD, pero las herramientas de generación de código no se verán afectadas. No producen variaciones en el parámetro `S`.
- *Cambios en requisitos no funcionales que fueron tenidos en cuenta en el diseño e implementación del framework*. Aquellos cambios que puedan ser solucionados utilizando los mecanismos de extensión

y especialización del framework tendrán un impacto pequeño en el esfuerzo necesario para acometerlos. Por ejemplo, el framework descrito en la sección 5.1 define una super–clase que modela la gestión de las máquinas de estados. De esta forma, un cambio en la gestión de las máquinas de estados requiere únicamente crear una nueva sub–clase y modificar la transformación para instanciarla convenientemente. En el caso del enfoque (2), este cambio requerirá modificar algunas reglas de la transformación, que seguramente estarán acopladas con otras, dando lugar a una cadena de modificaciones. En este caso se puede establecer que $\Delta S_{T2} \simeq (\Delta S_{T3} + \Delta S_F)^n$, con $'n' \geq 1$.

- *Cambios en requisitos no funcionales que no fueron tenidos en cuenta en el diseño del framework*. Bajo ciertas circunstancias es posible modificar el diseño del mismo para acomodar estos cambios. Por ejemplo, el cambio del mecanismo de comunicación del framework (descrito en la sección 5.1) de asíncrono a síncrono puede ser acomodado en el diseño actual modificando un conjunto significativo de clases base del mismo. Pero de manera general requerirá realizar un nuevo diseño en ambos enfoques.

Por nuestra experiencia, la ocurrencia de cambios en los requisitos sigue una proporción 70 % cambios funcionales, 20 % cambios considerados en el framework y 10 % cambios no considerados en el diseño del framework. Esto significa que en el doble de los casos en que se producen modificaciones importantes en los requisitos no funcionales, estos podrán ser acomodados en el framework, enfoque (3), con un esfuerzo varios órdenes de magnitud menor que en el enfoque (2). Esto hace que el enfoque (3) sea claramente más favorable que el (2), lo cual justifica suficientemente el interés de la propuesta de este artículo.

7. Conclusiones y Trabajos Futuros

En este artículo se ha presentado un enfoque que da soporte al desarrollo de aplicaciones basadas en componentes con restricciones de tiempo–real, que ha sido ilustrado con el diseño de un framework y su aplicación a un caso de estudio robótico. Una diferencia sustancial de esta propuesta frente a otras analizadas en el estado de la técnica es la posibilidad de modelar aplicaciones usando únicamente los conceptos y construcciones ofrecidos por el enfoque CBSD. Para ello, es necesario desarrollar y reutilizar frameworks (implementados usando tecnología OO) que constituyan el soporte en ejecución a dichas aplicaciones. MDSD es el enfoque que da soporte conceptual y tecnológico tanto a los niveles de abstracción considerados como a los mecanismos de transformación entre ellos. En suma, la propuesta descrita ofrece al desarrollador de aplicaciones un enfoque basado en frameworks en el que el artefacto primario del desarrollo son modelos de componentes, en contraposición a otros enfoques orientados a lenguajes de programación. Finalmente, se ha realizado una estimación de los costes de desarrollo de dicha propuesta, comparándola con otras propuestas similares. Cabe destacar que el desarrollo de una propuesta tan ambiciosa ha sido posible gracias a la madurez tecnológica de frameworks y de MDSD.

Entre los beneficios obtenidos por la aplicación del enfoque pueden destacarse: (1) la mejora en la gestión de la complejidad de los sistemas RTS gracias a los mecanismos de abstracción y el uso de modelos; (2) la minimización de los riesgos inherentes al desarrollo del software dado que se favorece el análisis en etapas tempranas, al tiempo que se permite el estudio adelantado de soluciones alternativas; (3) la mejora en la comunicación entre desarrolladores ya que se proporcionan mecanismos explícitos para compartir modelos; y (4) la reutilización tanto de los modelos de diseño como de la infraestructura necesaria para darles soporte de ejecución en un computador (frameworks).

Entre las líneas de investigación en marcha como continuación del trabajo se incluyen: desarrollar e integrar heurísticos para la distribución de actividades en hilos y el agrupamiento de las mismas, integrar herramientas de análisis de RTS así como reutilizar, en la medida de lo posible, los recursos proporcionados por frameworks robóticos como los descritos en la Sección 2, implementar un asistente para la selección y configuración de las características de los frameworks, extender el catálogo de frameworks disponible y, por último, desarrollar nuevos casos de estudio para demostrar la integración de frameworks complementarios, como interfaces gráficas, comunicaciones, etc.

English Summary

Automatic Code Generation for Real–Time Systems: a Development Approach based on Components, Models, and Frameworks.

Abstract

Real-Time Systems have some characteristics that make them particularly sensitive to architectural decisions. The use of Frameworks and Components has proven effective in improving productivity and software quality, especially when combined with Software Product Line approaches. However, the results in terms of software reuse and standardization make the lack of portability of both the design and component-based implementations clear. This article, based on the Model-Driven Software Development paradigm, presents an approach that separates the component-based description of real-time applications from their possible implementations on different platforms. This separation is supported by the automatic integration of the code obtained from the input models into object-oriented frameworks. The article also details the architectural decisions taken in the implementation of one of such frameworks, which is used as a case study to illustrate the proposed approach. Finally, a comparison with other alternative approaches is made in terms of development cost.

Keywords:

Software Engineering, Component-Based Software Develop-

ment, Model-Driven Software Development, Framework, Software Design Patterns, Real-Time.

Agradecimientos

Este trabajo ha sido parcialmente financiado por los proyectos EXPLORE (CICYT ref. TIN2009-08572) y MISSION-SICUVA (Fundación Séneca ref. 15374/PI/10).

Referencias

Alonso, D., Vicente-Chicote, C., Ortiz, F., Pastor, J., Álvarez, B., Jan. 2010. V^3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. Journal of Software Engineering for Robotics (JOSER) 1 (1), 3–17.

Antkiewicz, M.and Czarnecki, K., Stephan, M., Dec. 2009. Engineering of framework-specific modeling languages. IEEE Trans. Software Eng. 35 (6), 795–824.

Artist-ESD, 2008-2011. ArtistDesign - European Network of Excellence on Embedded Systems Design.
URL: http://www.artist-embedded.org/

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J., 2001. Component-based product line engineering with UML. A-W Prof.

Autosar, 2008-2011. AUTOSAR: Automotive Open System Architecture.
URL: http://www.autosar.org/

Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., Thanh-Hung, N., Mar. 2009. Designing autonomous robots. IEEE Robot. Automat. Mag. 16 (1), 67 – 77.

Blair, G., Coupaye, T., Stefani, J. (Eds.), 2009. Annals of Telecommunication. Component-based architecture: the Fractal initiative. Vol. 64. Springer-Verlag.

Boehm, B., Abts, C., Brown, A., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., Steece, B., 2000. Software Cost Estimation with Cocomo II. Prentice Hall.

Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A., 2007. Software Engineering for Experimental Robotics. Vol. 30 of STAR. Springer-Verlag, Ch. ORCA: A component model and repository, pp. 231 – 252.

Brugali, D., Menga, G., Mar. 2000. Frameworks and pattern languages: an intriguing relationship. ACM Computing Surveys 32 (2), 1–6.

Bruyninckx, H., 2001. Open Robot Control Software: the OROCOS project. In: Proc. of the IEEE International Conference on Robotics and Automation. Vol. 3. IEEE, pp. 2523–2528.

Bures, T., Hnetynka, P., Plasil, F., Sep. 2007. Runtime concepts of hierarchical software components. International Journal of Computer & Information Science Special (8), 454–463.

Burns, A., Wellings, A., 1995. HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. Elsevier Science.

Buschmann, F., Henney, K., C. Schmidt, D., 2007a. Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing. John Wiley and Sons Ltd.

Buschmann, F., Henney, K., Schmidt, D., 2007b. Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages. John Wiley and Sons Ltd.

Bézivin, J., May 2005. On the unification power of models. Journal of Systems and Software 4 (2), 171–188.

Carlson, J., Håkansson, J.and Petterssonb, P., Aug. 2006. SaveCCM: An analysable component model for real-time systems. Electronic Notes in Theoretical Computer Science 160, 127–140.

Chella, A., Cossentino, M., Gaglio, S., Sabatucci, L., Seidita, V., 2010. Agent-oriented software patterns for rapid and affordable robot programming. Journal of Systems and Software 83 (4), 557 – 573.

Childs, A., Greenwald, J., Jung, G., Hoosier, M., Hatcliff, J., 2006. CALM and Cadena: metamodeling for component-based product-line development. IEEE Computer 39 (2), 42–50.

Crnkovic, I., Chaudron, M., Larsson, S., Nov. 2005. Component-based development process and component lifecycle. Journal of Computing and Information Technology 13 (4), 321–327.

Dominguez-Brito, A.and Hernandez-Sosa, D., Isern-Gonzalez, J., Cabrera-Gamez, J., 2004. Integrating robotics software. In: Proc. of the IEEE Intl. Conference on Robotics and Automation, ICRA 2004. Vol. 4. IEEE, pp. 3423 – 3428.

Douglass, B., Feb. 2004. Real Time UML: Advances in the UML for Real-Time Systems. A-W Prof.

D'Souza, D., Wills, A., 1998. Objects, Components and Frameworks With UML: The Catalysis Approach. A-W.

Fairbanks, G., Garlan, D., Scherlis, W., 2006. Design fragments make using frameworks easier. In: Proc. of the 21^{st} annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA 2006. ACM, pp. 75–88.

Fayad, M., Schmidt, D., Johnson, R., 1999. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons.

Gomaa, H., 2000. Designing Concurrent, Distributed, and Real-Time Applications with UML. Object Technology. Addison-Wesley, iSBN: 0-201-65793-7.

Grady Booch, G., Maksimchuk, R., Engel, M., Young, B., Conallen, J., Houston, K., 2007. Object-Oriented Analysis and Design with Applications, 3rd Edition. A-W.

Hongxing, W., Xinming, D., Shiyi, L., Guofeng, T., Tianmiao, W., 2009. A component based design framework for robot software architecture. In: Proc. of the 2009 IEEE/RSJ international conference on Intelligent robots and systems, IROS 2009. IEEE, pp. 3429 – 3434.

Iborra, A., Alonso, D., Ortiz, F., Franco, J., Sánchez, P., Álvarez, B., Mar. 2009. Design of service robots. IEEE Robot. Automat. Mag., Special Issue on Software Engineering for Robotics 16 (1), 24–33.

Jacobson, I., 1992. Object Oriented Software Engineering: A Use Case Driven Approach. A-W.

Lau, K., Wang, Z., Oct. 2007. Software component models. IEEE Trans. Software Eng. 33 (10), 709–724.

Medvidovic, N., Taylor, R., Jun. 2000. A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng. 26 (1), 70–93.

Mens, T., van Gorp, P., 2006. A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142.

OMG, Apr. 2006. CORBA Component Model formal/06-04-01 Specification.
URL: http://www.omg.org/docs/formal/06-04-01.pdf

OMG, Jun. 2008. MDA success stories.
URL: http://www.omg.org/mda/products_success.htm

OMG, 2009. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, formal/2009-11-02.
URL: http://www.omg.org/spec/MARTE/1.0

OpenEmbeDD, 2008-2011. OpenEmbeDD project, Model Driven Engineering open-source platform for Real-Time & Embedded systems.
URL: http://openembedd.org/home_html

Parsons, D., Rashid, A., Telea, A., Speck, A., Feb. 2006. An architectural pattern for designing component-based application frameworks. Software: Practice and Experience 36 (2), 157–190.

Pastor, J., Alonso, D., Sánchez, P., Álvarez, B., Jun. 2010. Towards the definition of a pattern sequence for real-time applications using a model-driven engineering approach. In: Proc. of the 15^{th} Ada-Europe International Conference on Reliable Software Technologies, Ada Europe 2010. LNCS. Springer-Verlag, pp. 167–180.

Schlegel, C., 2006. Communication patterns as key towards component-based robotics. International Journal on Advanced Robotics Systems 3 (1), 49–54.

Shaw, M., Clements, P., Mar. 2006. The golden age of software architecture. IEEE Softw. 23 (2), 31–39.

Sommerville, I., 2010. Software Engineering, 9th Edition. A-W.

Stahl, T., Völter, M., 2006. Model-Driven Software Development: Technology, Engineering, Management. Wiley.

Szyperski, C., 2002. Component software: beyond object-oriented programming, 2nd Edition. A-W.

van Ommering, R., van der Linden, F., Kramer, J., Magee, J., 2000. The koala component model for consumer electronics software. IEEE Computer 33 (3), 78–85.