Actas de las XIII Jornadas de Tiempo Real

Granada, 4-5 de Febrero de 2010

Editores:
Manuel I. Capel Tuñón
Juan Antonio Holgado Terriza

Organizado por
Grupo de Sistemas Concurrentes
Dpto. de Lenguajes y Sistemas Informáticos
Universidad de Granada

Godel Editorial

Actas de las

XIII Jornadas de Tiempo Real

(JTR 2010)

Granada, 4 y 5 de Febrero de 2010

EDITORES

Manuel I. Capel Tuñón
Juan Antonio Holgado Terriza

ORGANIZADO POR:

Grupo de Sistemas Concurrentes
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

ISBN 10: 978-84-92757-51-0

Depósito Legal: GR-414/2010

Maquetación: Juan Antonio Holgado Terriza

Fotocomposición: Ediciones Copicentro Granada SL

Impreso por Talleres Gráficos Copicentro Granada SL

Impreso en España - Printed in Spain

xCCM: Plataforma RT-Linux para aplicaciones de tiempo-real basadas en componentes

Ángela del Barrio, Laura Barros, Patricia López Martínez y José M. Drake

Departamento de Electrónica y Computadores, Universidad de Cantabria,
39005-Santander, SPAIN
{delbarrioa,barrosl,lopezpa,drakej}@unican.es

Resumen: Se describe la infraestructura realizada con la implementación Xenomai de RT-Linux para la ejecución de aplicaciones de tiempo real basadas en componentes. Se utiliza la tecnología RT-CCM que es una extensión de la tecnología CCM de OMG propuesta a fin de construir aplicaciones distribuidas de tiempo real estricto. Se definen tres servicios genéricos: ThreadingService, SchedulabilityService y SynchronizationService, que son implementados por el contenedor de los componentes con requisitos de tiempo real y cuya función es proporcionar los threads y los mecanismos de sincronización que el código de negocio de los componentes requiere para implementar su funcionalidad.

Palabras clave: RT-Linux, Component-based, hard real-time, scheduling design

1 Introducción[1]

La idea básica del paradigma de componentes consiste en poder construir aplicaciones mediante ensamblado de módulos software que satisfagan tres características: *aislamiento* (el componente es una unidad atómica de despliegue), *componibilidad* (un componente puede ser ensamblado con otros componentes para generar o bien un nuevo componente o una aplicación) y *opacidad* (ni el entorno, ni los otros componentes, ni el diseñador necesitan conocer el código interno del componente para manejarlo).

La aplicación del paradigma de componentes al desarrollo de aplicaciones de tiempo real ha suscitado la aparición de diferentes estrategias muy diferentes entre sí. En este trabajo, se emplea la tecnología RT-CCM (*Real-Time Container Component Model)* [1] que tiene su origen en diferentes proyectos europeos [2][3], y que resulta de aplicar la especificación CCM (*CORBA Component Model*) propuesta por OMG [4] con dos excepciones:

1. Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia del Gobierno de España dentro del proyecto TIN2008-06766-C03-03 (RT-MODEL) y por la Unión Europea bajo el proyecto FP7/NoE/214373 (ArtistDesign). Este trabajo refleja sólo el punto de vista de los autores; la UE no se responsabiliza del uso que se pueda hacer de la información contenida.

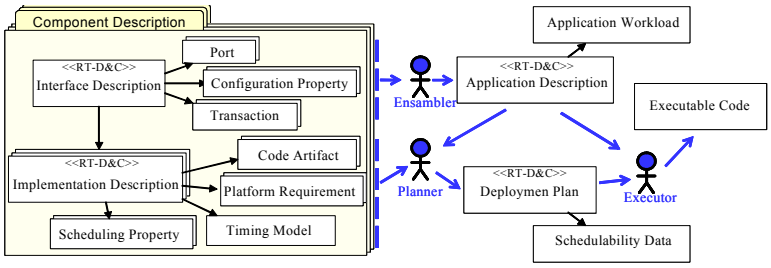

Fig. 1. Información asociada a un componente RT-CCM

- La interacción entre componentes se formula en base a unos componentes especializados denominados conectores, que incorporan en su código el mecanismo de comunicación (no necesariamente basado en CORBA, como requiere OMG).
- En la especificación del componente y de sus implementaciones, se incluyen metadatos relativos a la respuesta temporal del componente y que tienen como objeto poder predecir el comportamiento temporal de las aplicaciones en las que se utiliza el componente.

Como muestra la figura 1, un componente RT-CCM se distribuye como un paquete que contiene toda la información necesaria para utilizarlo en una aplicación:

- Descripción de la interfaz del componente: La utiliza el ensamblador de la aplicación y en base a ella decide si el componente es adecuado para la construcción de la aplicación que está diseñando, tanto desde el punto de vista funcional, como del comportamiento temporal que se requiere al componente para satisfacer los requisitos de tiempo real que tiene especificados.
- Descripción de las implementaciones del componente: La utiliza el planificador de la aplicación para definir el despliegue de la aplicación en una determinada plataforma distribuida y como vía para obtener las instancias del componente utilizadas en los nudos de procesamiento a los que hayan sido asignadas.

La especificación D&C de OMG [5] define los formatos y los contenidos de los documentos estandarizados que describen al componente, y se utiliza como guía para que los actores que diseñan la aplicación, y las herramientas que le dan soporte, puedan navegar por ella. La extensión RT-D&C [6] amplía la especificación D&C con la información de los modelos de comportamiento temporal necesarios para hacer predecible el comportamiento temporal de las aplicaciones que usan el componente.

En la tecnología RT-CCM, los componentes son módulos software de complejidad y estructura interna arbitraria, que definen la funcionalidad ofrecida y requerida, mediante la especificación de las interfaces de sus puertos. Los componentes definen su capacidad de configuración, a través de la descripción de las propiedades de configuración que tienen definidas en su interfaz, y definen su comportamiento temporal a través de modelos reactivos que describen las actividades y los recursos que se requieren cuando son invocados sus servicios servicios ofertados. En este trabajo tratamos únicamente la estrategia que se utiliza para los requisitos de tiempo real y cómo se

implementan los recursos que se requieren de la plataforma utilizando RT-Linux con Xenomai.

2 ScadaDemo: un ejemplo de aplicación RT-CCM.

En la figura 2 se muestra un ejemplo sencillo de una aplicación scada que ha sido diseñada utilizando RT-CCM. En este trabajo presentamos esta aplicación y los componentes que utiliza como medio para simplificar la exposición de los conceptos que se plantean en él. La aplicación supervisa un conjunto de variables analógicas de entorno y almacena información estadística de las valores leídos de cada variable en un registro (*Logger*). El operador desde un teclado puede elegir una de ellas de forma que los valores que se obtienen se visualicen en el monitor. La aplicación se construye utilizando cuatro tipos de componentes, organizados en una arquitectura de tres capas. La capa más baja está compuesta por los componentes genéricos *IOCard* y *Logger* que gestionan los recursos del sistema, la capa intermedia está constituida por el componente *ScadaEngine* que implementa la funcionalidad del dominio de aplicación *SCADA*, y por último, en la capa superior está el componente *ScadaManager* que implementa la funcionalidad específica de la aplicación. En la figura 2, se muestra también la funcionalidad básica de los componentes, los puertos que ofertan y requieren, y las interfaces que implementan. En la sección 3 se describe con más detalle el componente ScadaManager que se utiliza como ejemplo.

La funcionalidad de una aplicación de tiempo real se describe en RT-CCM utilizando una especificación de tipo reactivo, esto es, formulando las transacciones que describen las respuestas a los eventos externos o temporizados que gestiona el sistema. En el ejemplo ScadaDemo, existen cuatro transacciones, de las cuales tres, tienen requisitos de tiempo real:

1. Transacción *Sampling*: Con periodo *samplingPeriod*, se leen y procesan estadísticamente los valores de cada magnitud.
2. Transacción *Logging*: Con periodo *loggingPeriod* (superior a *samplingPeriod*) la información de todas las magnitudes supervisadas se registra en la base de datos.

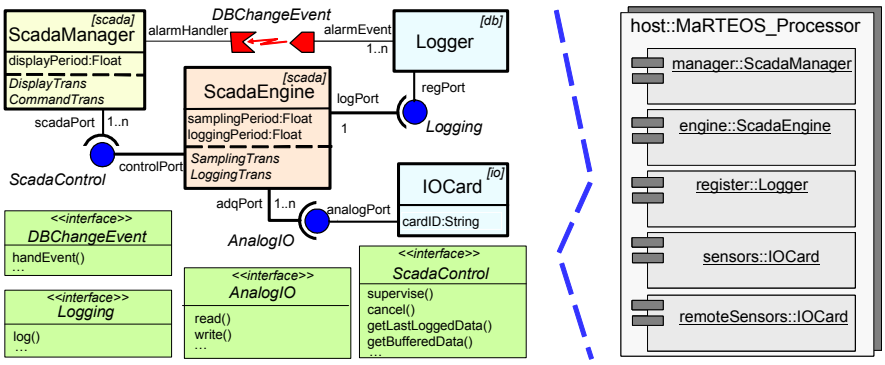

Fig. 2. Arquitectura y despliegue de la aplicación ScadaDemo

3. Transacción *Display*: Con periodo *displayPeriod*, la información relativa a una de las magnitudes supervisadas es actualizada en el monitor.
4. Transacción *Command*: Se atienden y ejecutan las instrucciones que introduce el operador por el teclado sin requisitos temporales.

3 Threads y componentes

Un componente es un módulo software que ha sido diseñado de forma que puede ser instanciado como un ente independiente y que puede ser reutilizado en diferentes aplicaciones. El principio clave del paradigma de componentes consiste en que los componentes puedan ser elegidos, instanciados, conectados, configurados y ejecutados, sin modificar ni conocer sus códigos. A tal fin, los componentes llevan asociados metadatos que describen la funcionalidad que ofrecen, y los servicios y recursos de la plataforma y de otros componentes de los que dependen para implementar su funcionalidad.

Un componente contiene el código de los servicios que ofrece, de la gestión de los recursos que crea y de las respuestas a los eventos que maneja. En el caso general, el código de un componente se ejecuta concurrentemente en múltiples threads que tienen su origen o bien en el propio componente (si éste es activo) o en otros componentes que invocan concurrentemente sus servicios. En consecuencia, los componentes utilizan mecanismos de sincronización internos para garantizar el acceso seguro a los recursos que requieren exclusión mutua y para la sincronización entre los threads que interaccionan en los componentes.

Como ejemplo, en la figura 3 se muestran los threads que concurren en el componente ScadaEngine de la aplicación ScadaDemo. En él se ejecutan concurrentemente cuatro threads. Dos son internos del componente: el thread *samplingTh* que lee periódicamente las magnitudes analógicas supervisadas, y el thread *loggingTh*, que registra periódicamente en el logger los resultados estadísticos de la información leída. Otros dos threads provienen de la invocación de servicios a través de la faceta *controlPort*: el thread *keyboardTh*, que modifica la lista de variables que se supervisan, y el thread *displayTh*, que requiere periódicamente la información de la magnitud que está siendo mostrada en el monitor. Los cuatros threads acceden a los datos que mantiene el componente, por lo que deben estar sincronizados a través de un mutex al que se denomina *dataMutex*.

El diseño de tiempo real y el análisis de planificabilidad de las aplicaciones que hacen uso de los componentes requieren que los threads, los mecanismos que sincronizan su flujos de control y las actividades que ejecutan, sean conocidos. Además, la configuración de la planificabilidad de la aplicación requiere que los parámetros que controlan la planificabilidad de estos elementos puedan ser establecidos por las herramientas de configuración y lanzamiento.

A fin de hacer compatible la opacidad inherente a los componentes con el diseño de tiempo real, en RT-CCM la creación y gestión de los threads y de los mecanismos de sincronización se han excluido del código de los componentes y se implementan en el

contenedor cuyo código es conocido y accesible.

La tecnología RT-CCM utiliza tres elementos para facilitar la planificabilidad de los threads:

- Define cuatro nuevos tipos de puertos a través de los cuales el código del componente accede a los threads y a los mecanismos de sincronización que proporciona el contenedor.
- Define el modelo de tiempo real del componente que describe de forma simplificada la asignación entre threads y actividades y los estados en los que los threads sincronizan su flujo de control.
- Utiliza los interceptores definidos en la especificación CCM para establecer los parámetros de planificación con los que se ejecuta cada servicio del componente, en función de la transacción y del punto de ésta en la que se invoca.

Los tipos de puertos que se definen en la tecnología RT-CCM, son:

- Facetas *PeriodicActivation*: Por cada puerto de este tipo que declara el componente, el contenedor (por medio del ThreadingService) crea un thread que ejecuta periódicamente el método *update*() que ofrece el puerto. La actividad que ejecuta *update*() debe tener una duración limitada (inferior al periodo de activación).
- Facetas *OneShotActivation*: Por cada puerto de este tipo que declare el componente, el contenedor (por medio del ThreadingService) crea un thread que ejecuta el método *run()* que ofrece el puerto. El thread ejecuta el método *run()* una sola vez cuando se activa el componente. El método *run()* ejecuta una actividad de duración arbitraria, que puede perdurar toda la vida activa del componente.
- Receptáculos *Mutex:* Por cada puerto de este tipo el contenedor (por medio del SynchronizationService) crea un mutex, y el componente accede a él a través de los metodos *lock()* y *unlock()* que implementa la interfaz de este tipo de puerto.
- Receptáculos *ConditionVariable*: Por cada puerto de este tipo el contenedor (por medio del SynchronizationService) crea una variable de condición y el código del componente accede a él a través de los métodos *wait()*, *notify()* y *notifyAll()* que implementa la interfaz de este tipo de puerto.

En la figura 3(b) se muestran los puertos que declara el componente ScadaEngine para requerir los thread internos *samplingTh* y *loggingTh* y el mutex *dataMutex* que requiere para implementar su funcionalidad.

La ejecución de los servicios y las respuestas a los eventos que gestionan los componentes deben ser planificadas con parámetros de planificación (prioridad, preemption level, deadline, etc.) que son función de la transacción y del estado dentro de la transacción en que se invocan. Para facilitar la planificación, cada una de las transacciones de la aplicación se suele asociar a un thread independiente si es secuencial, o a un conjunto de threads si es de naturaleza concurrente. Cuando los requisitos temporales están asociados a la finalización de la última actividad de la transacción la planificación se puede asociar a los threads, y todas las actividades se ejecutan con los mismos valores de los parámetros de planificación. Sin embargo, si los requisitos temporales

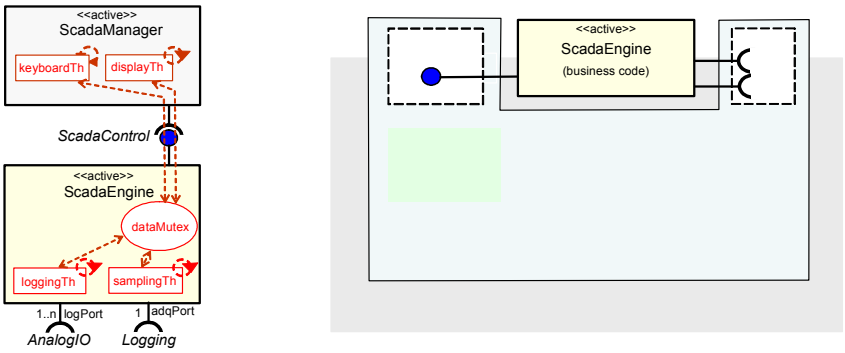

Fig. 3. Threads en el componente ScadaEngine

están asociados a estados intermedios de la transacción, hay que asociar los valores de los parámetros de planificación a cada actividad de manera independiente.

La planificación de la ejecución de los servicios y de las respuestas a los eventos que atiende un componente se realiza en RT-CCM haciendo uso de los interceptores. Éstos son mecanismos que introduce la especificación CCM, y cuya función es invocar una operación del entorno antes de que se inicie la ejecución del servicio, y otra después de que finalice su ejecución. En la tecnología RT-CCM antes de que se invoque un servicio o de que se inicie la respuesta a un evento, el interceptor invoca el método *setSchedParam()* del SchedulingService del contenedor, al cual se le pasan como parámetros el *stimId* que identifica la transacción y su estado, el puerto y la operación que se invoca, y en respuesta modifica los parámetros de planificación del thread y el valor del propio *stimId*. Cuando la ejecución del servicio del componente finaliza, el interceptor invoca el método *recoverSchedParam()*, y se restauran el valor de *stimId* y los valores de planificación del thread a los valores que tenían antes de la invocación.

En la figura 5, se muestra como ejemplo la gestión de la planificabilidad de la ejecución de las actividades de la transacción *LoggingTrans*. Cuando en esta transacción se invoca el servicio *log* del objeto *register::Logger*, se realiza con *stimId*=24, y al invocar a través del interceptor el método *setSchedParam()* del servicio SchedulingService, se establece de acuerdo con la configuración de planificabilidad establecida para la aplicación, que este servicio se ejecutará bajo *stimId*=27 y con prioridad 23. Cuando el servicio finaliza, se invoca el método recoverSchedParam() del servicio ScheduligService, y de nuevo la ejecución del servicio *update()* recupera su *stimId* 24 y la prioridad 15.

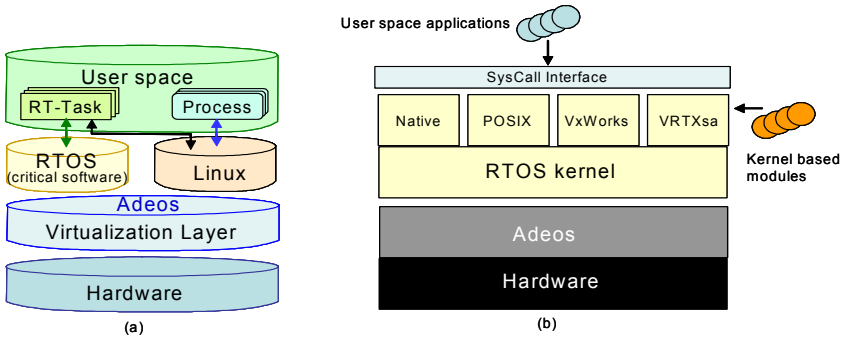

Fig. 5. (a) Arquitectura Xenomai. (b) Interfaces a RTOS legados.

4 Plataforma Xenomai

Xenomai [7][8] es una extensión de tiempo real para Linux. Se basa en una arquitectura en la que se mantienen simultáneamente en ejecución dos núcleos, el de tiempo real, que ofrece los recursos propios de un sistema operativo de tiempo real y un núcleo de Linux estándar que mantiene todos sus recursos. Entre ambos existen diferentes tipos de mecanismos de comunicación diseñados para que preserven la predictibilidad que requiere el tiempo real. En la figura 5a, se muestra los principales elementos de la arquitectura Xenomai. Ambos núcleos coexisten haciendo uso de una capa de virtualización denominada Adeos [9] que intercepta las interrupciones del hardware y las redirecciona hacia ambos núcleos. Adeos está diseñado para que la transición entre la ejecución en los dos núcleos sea muy ligera y para que la latencia de respuesta a las interrupciones hardware sea mínima.

El proyecto Xenomai estaba inicialmente integrado en el proyecto RTAI[10], pero a partir de 2005 se dividieron e incluso actualmente compiten entre ellos en el mercado. Su arquitectura es similar (por ejemplo comparten Adeos), pero sus objetivos son diferentes. RTAI cuida los aspectos de bajo nivel y ofrece latencias significativamente menores que las de Xenomai. Éste se ha orientado a facilitar la adaptación del software desde otros sistemas operativos legados como VxWorks, pSOS+ o VRTXsa. Como se muestra en la figura 5b, Xenomai integra un conjunto de interfaces especializadas (*skins*) que emulan las APIs de estos sistemas operativos comerciales. En la tecnología xCCM optamos por utilizar la intefaz nativa de Xenomai ya que es más rica y el comportamiento de sus métodos está mejor definido

La API nativa de Xenomai ofrece seis familias de servicios[11]. Cada familia define un amplio conjunto de funciones, la mayoría de las cuales pueden ser invocadas tanto desde el espacio de usuario, como desde el núcleo:

- Gestión de tareas: Define un conjunto de servicios relacionados con el ciclo de vida de los threads y el control de sus parámetros de planificación.
- Servicios de temporización: Esta familia agrupa funciones relativas a la gestión de los diferentes tipos de temporizadores: *System timers*, *Watchdogs* y *Alarms.*

- Mecanismos de sincronización: Ofrece recursos para la sincronización de las tareas y el control de acceso a recursos que requieren exclusión mutua: *Counting semaphores, Mutexes*, *Condition variables* y *Event flag groups*.
- Servicios de comunicación y transmisión de mensajes: Ofrece servicios para implementar diferentes modos de intercambio de bloques de datos entre las tareas de tiempo real, y entre éstas y los procesos Linux del espacio de usuario: *Intertask synchronous message*, *Message queues*, *Memory heaps* y *Message pipes*.
- Manejadores de dispositivos. Como la gestión de entradas/salidas es uno de los aspectos mas complejos de los RTOS, la interfaz nativa sólo ofrece mecanismos de gestión de las interrupciones, y de acceso a memoria de I/O desde el espacio de usuario.
- Soporte de registros: Es muy específica de la interfaz nativa, y ofrece funciones semejantes a las llamadas de sistema desde los diferentes espacios de ejecución.

5 Servicio ThreadingService

Es un servicio del entorno de ejecución con el que los contenedores de los componentes crean los threads que invocan los puertos de activación. En el diagrama de clases de la figura 6 se muestra la funcionalidad del servicio. Las principales características de este servicio son:

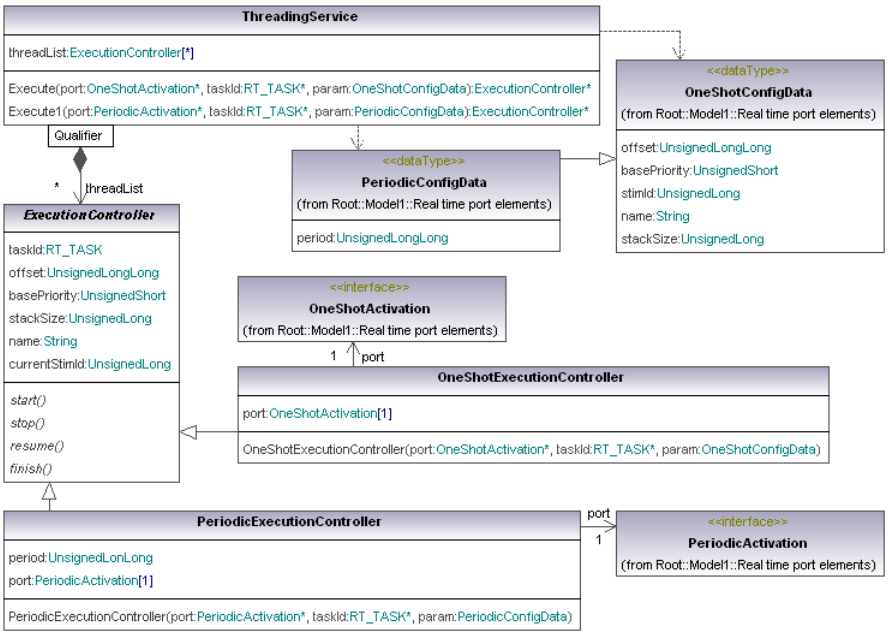

Fig. 6. Estructura y funcionalidad del ThreadigService

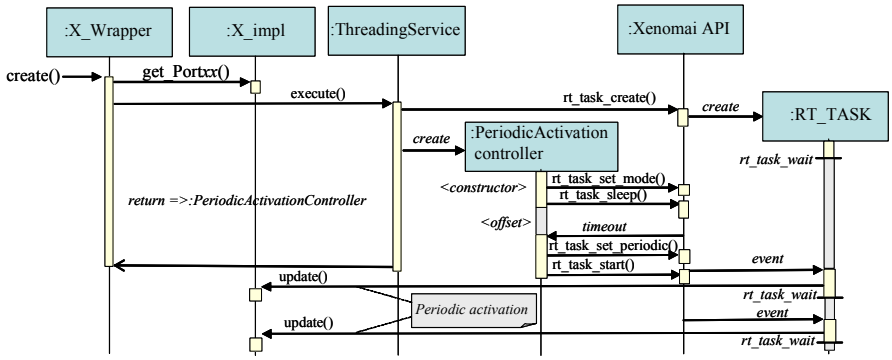

Fig. 7. Interacciones con el API de Xenomai para la gestión de threads.

- Para la activación de un componente basta con invocar, por cada puerto de activación, el método estático *execute()* del ThreadingService. Mediante esta invocación se crea un thread que activa el puerto del componente que se le pasa como parámetro.
- Por cada thread que se crea se retorna un objeto *ExecutionController* con el que el contenedor controla (*start(), stop(), resume(), finish()*) el ciclo de vida del thread, de acuerdo con el estado del componente.
- Como se muestra en el diagrama de secuencias de la figura 7, utilizando Xenomai es posible implementar el servicio de forma muy simple ya que el API de las tareas de Xenomai ofrece funciones muy próximas a las que se necesitan en RT-CCM.
- Una dificultad importante que se ha encontrado es la imposibilidad de extender la información asociada a un thread. Xenomai asocia a cada thread la estructura *RT_TASK_INFO* que es cerrada y no admite ser extendida. Por ello se ha optado por incluir en el nombre de la tarea un identificador con dos caracteres numéricos, que permiten, acceder de forma eficiente al correspondiente *ExecutionController*, y en él incluir los atributos que se asignan a la tarea. Esta estrategia se ha utilizado para incluir el atributo *StimId* que se utiliza para gestionar la planificación de la tarea.

6 Servicio Synchronization Service

El servicio del entorno de ejecución *SynchronizationService* es utilizado por los contenedores para crear los mecanismos de sincronización (*Mutex* y *ConditionVariables*) que requiere el código de los componentes para la sincronización de los threads. Cada mecanismo de sincronización es a su vez un objeto que ofrece la interfaz *Mutex* o *ConditionVariable* que espera el código del componente, por estar definida en la tecnología RT-CCM.

En el diagrama de clases de la figura 8 se muestra la estructura y la funcionalidad del

servicio:

- Para que un componente pueda disponer de un mutex basta con invocar el método estático *getMutex()* del SynchronizationService y pasar al componente la referencia del objeto que retorna. Para garantizar la seguridad en la gestión de los mutex se eleva una *OwnerMutexException* si un thread que no tiene tomado el mutex trata de liberarlo ejecutando una operación *unlock()*. La implementación del mutex utilizando Xenomai es muy simple ya que el API de los mutex coincide exactamente con la funcionalidad que se necesita.
- Para que un componente pueda disponer de una variable de condición basta con invocar el método estático *getConditionVariable()* del SynchronizationService y retornar al componente la referencia del objeto retornado.
- En la implementación de xCCM, la variable de condición se relaciona uno a uno con el mutex que se utiliza para garantizar la seguridad de la operación *wait()* del mutex. A tal fin, el mutex debe ser creado, y pasado como parámetro en la creación de la variable de condición. Si sobre la variable de condición se ejecuta alguna operación sin tener tomado el mutex, se eleva la *GuardMutexException.*
- En CCM las variables de condición se utilizan con varios objetivos. A veces, de forma convencional para sincronizar dos threads internos que interaccionan dentro de un componente, otras veces es la forma en que un componente gestiona un grupo de threads, que se generan en la creación del componente y luego se activan según los eventos externos que gestiona el componente. Cuando el componente tiene capacidad de responder a diferentes eventos externos, el código de negocio tiene capacidad para establecer el StimId que se asigna al thread que se activa en la variable condición.

7 Servicio SchedulabilityService

La tecnología RT-CCM utiliza los interceptores como base para poder establecer los parámetros de planificabilidad con los que se ejecuta cada invocación de los servicios de un componente. Un interceptor es un elemento que se introduce en el contenedor por cada método de los puertos de un componente que se declaran de tiempo real, y que introduce un mecanismo para que cuando se invoque el método, se invoque previamente una operación del entorno (*receiveRequest()*), y cuando se obtenga el retorno del método, se ejecute una nueva operación del entorno (*sendReply()*).

En la tecnología RT-CCM se utilizan los interceptores para gestionar los parámetros de planificación con los que se ejecuta el servicio que representa el método al que se asocia. En xCCM el parámetro de planificación que se gestiona es la prioridad del thread que invoca la operación del servicio.

En la figura 9 se muestra la invocación desde un componente cliente del método *oper()* del componente que se representa. La invocación es procesada por el interceptor asociado al método *oper()*. Inicialmente invoca el método *nextStimId()* del *InterceptorSchedulingManager* asociado al interceptor, el cual de acuerdo con el estado de la transacción (*stimId* asociado al thread) evalúa el nuevo *stimId* con que se ejecutará

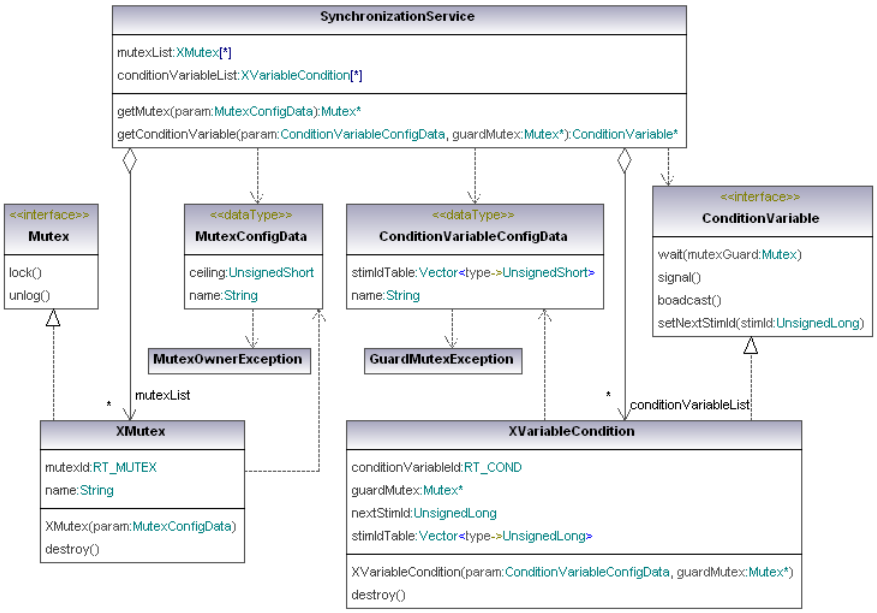

Fig. 8. Estructura y funcionalidad del SyncronizationService.

el servicio y lo establece en el thread invocante, y a través del método *setPriority()* del *SchedulingService* establece la prioridad del thread invocante con la que se ejecutará la operación invocada. Por último se invoca sobre el código del componente la operación *oper()* que era el objetivo de todo este proceso.

Cuando la operación *oper()* termina y retorna los resultados, interviene de nuevo el interceptor. El cual invoca el método *getbackStimId()* del objeto InterceptorSchedulingManager que reestablece en el *stimId* del thread el valor que estaba establecido antes de la invocación, y dentro de él se vuelve a invocar el método *setPriority()* del *SchedulingService* con el que también se restaura la prioridad que tenía el thread antes de la invocación.

En el diagrama de clases de la figura 10, se especifica la funcionalidad de los objetos relacionados con la gestión de los parámetros de planificación. El servicio SchedulingService se basa en la tabla de StimId que especifica los valores de los parámetros de planificación con los que debe ejecutarse cada servicio de los componentes en función de la transacción y del estado de ésta en que se invoca.

8 Conclusiones y trabajo futuro

En este trabajo se muestra una estrategia para implementar aplicaciones de tiempo real utilizando el paradigma de componentes, en una plataforma del tipo RT_Linux, en el que se combina la riqueza de recursos de una plataforma Linux convencional con la capacidad de ejecutar tareas con las restricciones y predictibilidad que requiere el

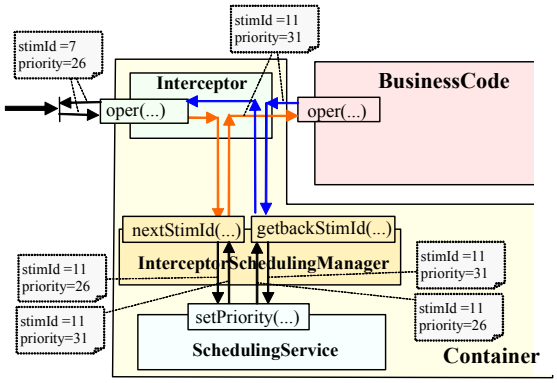

Fig. 9. Interaccion entre los interceptores y el SchedulingService.

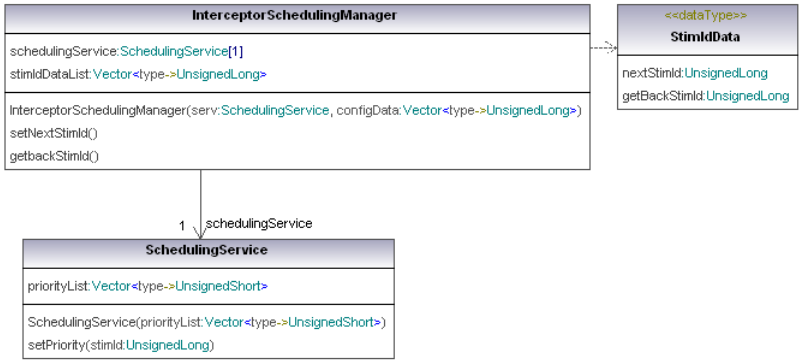

Fig. 10. Estructura y funcionalidad del ShedulingService

tiempo real estricto.

Se ha comprobado la flexibilidad que ofrece la tecnología de componentes RT-CCM para adaptarse a nuevas plataformas con un esfuerzo mínimo, y permitiendo reutilizar los componentes entre ellas. Una dificultad que se ha presentado es que, en su versión actual, sólo admite código en lenguaje C/C++, y no se han podido reusar los componentes formulados en otros lenguajes, por ejemplo Ada.

La implementación Xenomai de RT-Linux se ha mostrado muy flexible y segura, y la adaptación a ella de código desarrollado para otras plataforma resulta muy sencilla. Su documentación es buena a pesar de que carece de un sistema de soporte de usuarios dinámico.

El trabajo aun no está concluido, y presenta muchos huecos. Por un lado, se necesita el desarrollo de las estrategias de implementación de los conectores entre los componentes que corren en el núcleo de tiempo real de Xenomai y otras plataformas, y por otro, se requiere caracterizar el comportamiento temporal de los servicios Xenomai.

References

[1] P. López, J.M. Drake, P. Pacheco, and J.L. Medina: An Ada 2005 Technology for Distributed and Real-Time Component-based Applications. Proc. 13th Intl. Conf. on Reliable Software Technologies Ada-Europe, Venice, 2008.

[2] IST project COMPARE: Component-based approach for real-time and embedded systems http://www.ist-compare.org

[3] IST project FRESCOR: Framework for Real-time Embedded Systems based on Contracts http://www.frescor.org.

[4] OMG: Lightweight Corba Component Model, ptc/03-11-03, November 2003

[5] OMG: Deployment and Configuration of Component-Based Distributed Applications Specification, version 4.0, Formal/06-04-02, April 2006

[6] P. López Martínez et al.: Real-time Extensions to the OMG's Deployment and Configuration of Component-based Distributed Applications. OMG's 9th *Work*. Distributed Object Computing for Real-time and Embedded Systems, Arlington, VA, USA, 2008.

[7] Xenomai: Real-Time Framework for Linux: "http://www.xenomai.org/index.php/Main_Page"

[8] Philippe Gerum: "Xenomai - Implementing a RTOS emulation framework on GNU/Linux",<http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf>

[9] DEOS: Delaware Environmental Observing System, <http://www.deos.udel.edu/index.html>

[10] RTAI: Real Time Interface for Linux,<https://www.rtai.org/index>

[11] Native Xenomai API. <http://www.xenomai.org/documentation/xenomai-2.0/html/api/group__native.htmlNative API>