ScienceDirect
Disponible en www.sciencedirect.com
Revista Iberoamericana de Automática e Informática industrial 11 (2014) 98–108
www.elsevier.es/RIAI

Marco de componentes con soporte para reemplazo dinámico y seguro en sistemas de tiempo real

Julio Cano*, Marisol García-Valls, Pablo Basanta-Val

Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, Av. Universidad, nº30, 28911, Leganés, España

Resumen

En las últimas décadas se han aportado soluciones para el desarrollo de sistemas de tiempo real basados en componentes como base para aumentar la productividad y la fiabilidad de su desarrollo así como su posterior mantenimiento. De modo más reciente están apareciendo soluciones que permiten cierta flexibilidad en estos sistemas con miras a soportar ejecución dinámica a través de reemplazos de componentes en tiempo de ejecución. Para ello se adaptan los modelos de componentes intentando minimizar los conflictos que aparecen al integrar tiempo real y comportamiento dinámico y conseguir reemplazos de componentes en un tiempo acotado. Uno de los principales retos para esto es el cálculo de los tiempos requeridos por las diferentes operaciones necesarias para realizar un reemplazo de componente. El otro gran obstáculo es conocer los tiempos de operación de los componentes del sistema cuando la implementación de éstos puede cambiar durante la vida del sistema. En este trabajo se describe la implementación de un marco de componentes que aporta una solución parcial a estos problemas. Se proporciona un modelo de componentes junto con sus correspondientes algoritmos para asegurar que los componentes pueden ser cargados y reemplazados en tiempo de ejecución sin interferir en el cumplimiento de sus plazos de ejecución. El modelo está diseñado para evitar fallos en los reemplazos de componente. Finalmente se aporta la validación de los conceptos presentados.

Palabras Clave:
Marcos de componentes, tiempo real, sistemas dinámicos, reemplazo de componente, reconfiguración

1. Introducción

A lo largo de los años el principal objetivo del software de tiempo real ha sido garantizar la seguridad y el determinismo temporal en su ejecución. Esto se ha obtenido forzando la predictibilidad del sistema desde su diseño hasta su funcionamiento en tiempo de ejecución, evitando fuentes de indeterminismo como la presencia de comportamiento dinámico.

Evoluciones posteriores en la investigación de tiempo real han argumentado que, a pesar de ser un reto evidente, el comportamiento dinámico puede aumentar la seguridad y el rendimiento de los sistemas de tiempo real (Li, 2011)(Kramer and Magee, 1990)(Vandewoude et al., 2007)(Warren et al., 2006). La posibilidad de reemplazar partes defectuosas del sistema sin parar la ejecución de sus componentes incrementa la seguridad. La capacidad de actualizar partes software del sistema con partes nuevas, incrementando las capacidades y funcionalidad con mejor rendimiento, incrementa también la vida del sistema y permite corregir posibles fallos.

Una de las maneras más extendidas para soportar comportamiento dinámico y aumentar la reutilización del software es el uso de componentes (Canal et al., 2006)(McKinley et al., 2004). Se han realizado aportaciones que permiten soportar comportamiento dinámico en tiempo de ejecución a sistemas de componentes: desde sistemas distribuidos como el modelo de componentes de CORBA (*CORBA Component Model*, CCM) (OMG, 2006) en (Tewksbury et al., 2001)(Stankovic, 2000)(Almeida and Wegdam, 2001) hasta sistemas embebidos muy ligados a la plataforma como MARTE (OMG, 2009) en (Quadri et al., 2010).

Debido al extendido uso del lenguaje de programación Java se han diseñado muchos modelos de componentes haciendo uso de este lenguaje. Además también se han diseñado para Java especificaciones de tiempo real (*Real Time Specification for Java*,

* Autor en correspondencia:
Correo electrónico: julioangel.cano@uc3m.es
URL: http://www.it.uc3m.es/jcano/

http://dx.doi.org/10.1016/j.riai.2013.11.007

RTSJ) (Bollella and Gosling, 2000) con la intención de sobrepasar algunas de las limitaciones de la máquina virtual y el recolector de basura y aumentar su campo de uso a los sistemas más críticos.

Soportar comportamientos dinámicos en un sistema de tiempo real basado en Java presenta numerosos retos. Dicho comportamiento está parcialmente restringido en estos sistemas para que no se perjudique su predictibilidad y la fiabilidad. Sin embargo, que los componentes del sistema puedan ser actualizados sin afectar su correcta ejecución es un comportamiento deseable que permite dotar a los sistemas software de propiedades muy beneficiosas como la flexibilidad/versatilidad funcional o la corrección de errores de código detectados en la fase de ejecución o tras el despliegue del sistema.

El objetivo de este trabajo es proporcionar un marco de componentes para sistemas de tiempo real que presentan comportamiento dinámico, es decir, su funcionalidad puede ser modificada introduciendo nuevos componentes, eliminando otros o reemplazando otros en tiempo de ejecución. Por lo tanto, deben diseñarse mecanismos que permitan dicho comportamiento sin parar el sistema y sin que el resto de componentes vean afectadas sus restricciones temporales.

Actualmente la frontera entre componentes y servicios aparece, en ciertas contribuciones, algo difusa. En este trabajo hablamos de *componentes activos* (refiriéndonos a componentes cuya funcionalidad se basa en una tarea que ejecuta su código de forma periódica) y *componentes pasivos* (ofrecen su funcionalidad a través de operaciones que pueden ser invocadas por los componentes activos). No se contemplan aquí tareas aperiódicas. Debido a que la funcionalidad de unos componentes depende de la funcionalidad de otros, el tiempo total que un componente necesita para completar su ejecución depende del tiempo que requieren otros componentes para completar las operaciones invocadas. Si el sistema es estático (los componentes que integran el sistema no varían a lo largo de su vida) estos tiempos pueden ser conocidos en el momento de su diseño. En el caso de que la implementación de los componentes del sistema pueda cambiar a lo largo de su ejecución, los tiempos de cómputo de cada componente sólo pueden ser calculados en tiempo de ejecución.

En este trabajo se describe la implementación de un marco en el cual se proporciona reemplazo seguro y actualización de componentes en tiempo de ejecución preservando las propiedades temporales de la ejecución. También permite la carga y descarga de componentes en tiempo de ejecución. La carga y reemplazo de componentes sólo son aceptadas si la planificabilidad del sistema durante y después del reemplazo, puede ser garantizada.

Este trabajo se estructura como sigue. La siguiente sección describe trabajos relacionados con marcos de componentes dinámicos y de tiempo real. La sección 3 describe el modelo de componente usado y en el que se basa la implementación. Las operaciones ofrecidas por el marco, como la carga y descarga de componentes se describen en la sección 4. El reemplazo de componentes así como su implementación en el marco son descritos en la sección 5. La sección 6 describe cómo ha sido manejado el problema del recolector de basura en esta implementación. En la sección 7 se muestran los resultados de una validación empírica del marco. Finalmente la sección 8 aporta unas conclusiones sobre este trabajo.

2. Trabajos relacionados

Existen muchos marcos de componentes, algunos de ellos diseñados para ser dinámicos mientras que otros están más orientados a sistemas de tiempo real. En esta sección se describen los más relevantes, los cuales no llegan a aportar el nivel de dinamicidad que aporta nuestro trabajo.

2.1. Marcos de componentes de tiempo real

Uno de los modelos de componentes más relevantes orientados al desarrollo de sistemas de tiempo real es (Tesanovic et al., 2003), donde se hace uso de la orientación a aspectos para implementar varias de las características de los componentes y poder *tejerlas* (*weave*) entre sí. Este tejido de los diferentes aspectos de los componentes requiere calcular el tiempo de cómputo en el peor de los casos (*Worst Case Execution Time*, WCET) de cada operación de cada componente para los análisis de planificabilidad. Estos cálculos se realizan en tiempo de compilación, donde los diferentes aspectos son tejidos. No se realizan cambios en tiempo de ejecución. No se aporta un modelo o información que permita al sistema cierta dinamicidad para poder cargar, descargar o actualizar componentes en tiempo de ejecución.

VEST (Stankovic, 2000) es otro marco de componentes basado en composición orientada a aspectos. Este marco también incorpora el cálculo de tiempos de cómputo en el peor de los casos basados en la composición de aspectos para los análisis de planificabilidad. Igualmente carece de soporte para dinamicidad en tiempo de ejecución.

HOLA-QoS (Garcia-Valls et al., 2003) introduce un marco en el que se gestiona la calidad de servicio y permite realizar modificaciones en el sistema durante la ejecución del mismo. Esta propuesta se basa en cambio de modo, lo que la limita a los modos establecidos en tiempo de diseño.

Esta propuesta de HOLA-QoS se completa y mejora incluyendo un protocolo para gestión dinámica de prioridades en los cambios de modo en (García-Valls et al., 2012a). Esta contribución puede ser aplicada como base para conseguir reemplazos de bajo coste.

2.2. Marcos con capacidad de actualización/reemplazo en tiempo de ejecución

En (Sha, 1998) se propone un modelo para reemplazo seguro de componentes en tiempo de ejecución. Sha propone mantener el componente antiguo en el sistema tras ser reemplazado por uno nuevo. En caso de que la ejecución del nuevo componente falle es posible restaurar la ejecución del antiguo componente por cuestiones de seguridad. No se aporta ningún tipo de análisis de planificabilidad para asegurar que los reemplazos de componentes puedan llevarse a cabo con garantías de calidad de servicio..

En (Isovic and Lindgren, 2000) se hace referencia a la necesidad de tener en cuenta la verificación del WCET en tiempo de ejecución cuando los componentes son actualizados. Pero simplemente se supone que el WCET del nuevo componente es menor o igual al tiempo reservado para el componente a reemplazar, manteniendo los mismos valores en el resto de parámetros del componente. No se contempla ningún tipo de análisis de planificabilidad. Tampoco se contempla la reserva de recursos para poder realizar el propio reemplazo de componente.

El cálculo de WCET así como de los tiempos de ejecución extremo a extremo de componentes (*end-to-end*) de éste y otros trabajos están basados en (Cornwell and Wellings, 1996).

Rasche y Polze presentan en (Rasche and Polze, 2005) y (Rasche and Polze, 2008) una técnica para la reconfiguración dinámica de software basado en componentes en la que los componentes son bloqueados para poder aplicar tareas de gestión como una reconfiguración. Realmente la reconfiguración se aplica en una manera que la ejecución de todas las transacciones que se estén llevando a cabo en el momento entre componentes se terminen correctamente. Sin embargo, la aplicación es bloqueada hasta que la reconfiguración haya terminado. No se tienen en cuenta plazos de ejecución de tiempo real.

Otros trabajos sí que tienen en cuenta plazos de ejecución de tiempo real, como (Wahler et al., 2011) que se centra en la copia del estado del componente, la cual puede requerir varios ciclos de ejecución hasta completarse. Aunque este método no asegura que la copia del estado del componente pueda completarse en un número fijo de ciclos, dado que el componente sigue en ejecución y el estado puede verse modificado. Los datos ya copiados podrían tener que volver a ser copiados si se han modificado.

Existen más trabajos que proporcionan algún tipo de calidad de servicio durante una reconfiguración como (García-Valls et al., 2011)(Miguel et al., 2002) donde se propone reservar recursos para poder realizar reconfiguraciones de acuerdo con las necesidades del sistema. Este método está orientado a sistemas multimedia, basado en asignación de tiempos de ejecución y asignación dinámica de prioridades. Los recursos son reservados para cada componente independientemente de manera que los componentes pueden reconfigurarse para adaptarse a la calidad de servicio que necesita el sistema. Este método está centrado en asegurar una calidad de servicio para una constante reconfiguración en un sistema multimedia. Realmente el sistema no está diseñado para permitir el reemplazo de componentes, sino la reconfiguración de éstos para modificar la calidad de servicio ofrecida en cada momento.

De forma similar otro de los métodos más utilizados es el cambio de modo (García-Valls et al., 2009) donde todas las posibles configuraciones ya están predeterminadas en tiempo de diseño. Los cambios de un modo a otro y las tareas que se ejecutan en cada momento se encuentran ya predeterminadas. Uno de los ejemplos de implementación de este sistema está descrito en (García-Valls et al., 2012b). La principal limitación de la reconfiguración basada en cambio de modos es que los componentes no pueden ser actualizados o reemplazados por nuevas versiones en tiempo de ejecución. Las nuevas características temporales de los componentes darían lugar a configuraciones del sistema no previstas en tiempo de diseño.

También existen aproximaciones para soportar la ejecución dinámica en sistemas de tiempo real distribuidos basados en composición de servicios (García-Valls et al., 2013a) y ajustados al cumplimiento de requisitos de calidad de servicio. Para ello existen trabajos que caracterizan las propiedades de calidad de servicio relacionadas principalmente con el comportamiento temporal, como se describe en (García-Valls et al., 2013b).

2.3. Marcos de componentes para RTSJ

Existen múltiples marcos de componentes basados en el lenguaje Java. Uno de los más conocidos es el proporcionado por OSGi (OSGi Alliance, 2009). Éste es llamado *Declarative Services* (Servicios Declarativos) y está basado en componentes de servicios. Estos componentes declaran los servicios que proporcionan al resto del sistema y los servicios que requieren de otros componentes para poder trabajar. OSGi, por lo tanto, es una plataforma dinámica donde los componentes pueden ser cargados, descargados y reemplazados en tiempo de ejecución sin parar o bloquear la ejecución de los componentes relacionados. Esto es realizado haciendo uso de la *continuidad*, definida en los sistemas de componentes como la capacidad de mantener varias versiones de los mismos componentes a la vez en ejecución mientras se realizan los reemplazos o mientras sean referenciados por otros componentes. En ningún caso OSGi asegura la calidad de servicio en la plataforma, durante la ejecución normal del sistema o durante la ejecución de tareas de gestión.

Existen otros marcos basados en componentes y en el lenguaje Java que sí aportan características de tiempo real, haciendo uso de RTSJ. La mayoría de ellos aportan abstracciones de alto nivel para crear componentes de tiempo real o para gestionar memoria como (Bruneton et al., 2006), (Bures et al., 2006), (Clarke et al., 2001) y (Plšek et al., 2008). Sin embargo sólo Plšek hace una reseña sobre la necesidad de soportar la adaptación dinámica en (Plšek et al., 2008), donde se reconoce dicha necesidad pero tampoco se aporta ninguna solución.

En (Cano and García-Valls, 2013) se establecen las bases y los mecanismos para la planificación de tiempo real de reemplazos de componentes. En el presente trabajo se detalla el marco que englobaría las estrategias de reemplazo, proporcionando una vista más orientada a la implementación de dicho marco.

3. Modelo de componente

De acuerdo con (Crnkovic et al., 2011), los componentes pueden ser clasificados en *basados en operación* o *basados en puertos* dependiendo del tipo de interfaz ofrecida y sus conexiones. De ello dependerá la forma en la que sean conectados. En los modelos basados en puertos, los datos intercambiados entre componentes son enviados a través de una conexión. La ejecución de un componente es activada al recibir una señal de otro componente o de forma periódica a través de un reloj.

En el caso de OSGi su modelo de componentes está basado en operación debido a que ofrecen servicios mediante interfaces de Java. La ejecución de las operaciones del componente es activada cuando son invocadas desde otro componente. En Java esto representa la llamada o invocación de un método de un objeto.

3.1. Interfaces

Los métodos que proporciona un componente definen su interfaz. En este trabajo se diferencia entre dos tipos de interfaces: interfaces de configuración e interfaces de composición.

Interfaz de configuración: Es una interfaz común a todos los componentes y que permite que sean gestionados por el marco de componentes. Permite al marco desplegar los componentes en la plataforma. Los métodos definidos en esta interfaz se encuentran detallados en el Listado 1.

Esta interfaz proporciona básicamente dos métodos: uno para que el marco inicie la ejecución del componente y otro para detenerlo. Para poder conectar este componente a otros, en ejecución la plataforma ofrece el método `setDepend`. Este método, junto con `getState` y `setState` son usados en el proceso de reemplazo para copiar las dependencias del

componente así como su estado. Estos métodos serán detallados más adelante.

```
public interface Component {
   public Object getState();
   public void setState(Object state);

   public void setDepend(Dependence dep, Component
comp);

   public boolean start();
   public void stop();
}
```

Listado 1: Interfaz de configuración de componente

Interfaz de operaciones o composición: Los objetos de Java pueden ofrecer varias interfaces diferentes al mismo tiempo, pero por simplicidad (y compatibilidad con la plataforma OSGi) sólo se tiene en cuenta uno de los interfaces implementados. Esto también simplifica la composición de componentes de la misma manera que en los *servicios dinámicos* de OSGi. Cada componente sólo publica un servicio, aunque puede requerir varios servicios diferentes para trabajar.

3.2. Extendiendo el modelo de componente para reemplazo de tiempo real

El código de los componentes es ejecutado por tareas de tiempo real que deben ser planificadas para garantizar sus tiempos de ejecución. A continuación se incluye una descripción más detallada de los requisitos correspondientes.

El marco soporta dos tipos de componentes: pasivos y activos.

Componentes pasivos: Estos componentes reciben peticiones para ejecución de sus operaciones. No disponen de una tarea inherente al mismo que ejecute su código. Es necesario calcular el tiempo de cómputo en el peor de los casos para cada operación ofrecida, para que sea tenido en cuenta en la planificación global de tareas del marco de componentes.

Componentes activos: Se consideran componentes activos los que disponen de una tarea de tiempo real que ejecuta su código. La tarea de un componente activo se ejecuta de forma periódica y sus parámetros temporales utilizados por la plataforma para el análisis de planificabilidad son:

- Frecuencia o periodo de activación del componente (T_i).
- Tiempo de cómputo en el peor caso (C_i).
- Plazo de ejecución para cada periodo de activación del componente (D_i).

Donde *i* es el número de tarea o componente activo dentro del marco. Se considera que sólo hay una hebra de ejecución por componente activo.

Este modelo se corresponde con el modelo de planificación de prioridades monótonas en frecuencia (*Rate Monotonic Scheduling*, RMS) (Lehoczky et al., 1989) para el análisis de planificabilidad.

Los componentes activos hacen uso de los componentes pasivos. En un entorno dinámico en el que los componentes pueden ser cargados, descargados y reemplazados, el tiempo para completar la ejecución de una tarea puede variar (Isovic and Lindgren, 2000). Las condiciones de ejecución cambian durante la vida del sistema. Por tanto los tiempos de cómputo no sólo dependen de la plataforma física de ejecución, sino que también dependen del resto de componentes en ejecución en la plataforma. El tiempo necesario para completar la ejecución de operaciones de componentes pasivos afecta a los tiempos de ejecución de los componentes activos. Por lo tanto es necesario un método para calcular el tiempo de cómputo en el peor caso (extremo a extremo) para la tarea de un componente activo en tiempo de ejecución, es decir, antes de que el componente sea cargado y activado para poder pasar los análisis de planificabilidad.

3.3. Modelo de tiempo de cómputo en el peor caso

Tal y como se ha descrito en la sección anterior el tiempo total de ejecución extremo a extremo de la invocación de una tarea de componente depende del tiempo de ejecución de las operaciones utilizadas por dicho componente. El tiempo de cómputo de la tarea de un componente activo consta del coste de ejecutar el código de su propio componente más el coste de las operaciones de componentes pasivos que invoca. En este trabajo consideramos que estos tiempos pueden no ser conocidos en tiempo de diseño, sólo cuando los componentes son desplegados sobre la plataforma y conectados para su ejecución. Únicamente el tiempo mínimo de ejecución de la tarea del componente es conocido en tiempo de diseño. El tiempo total de ejecución de la tarea de un componente activo vendrá dado, por lo tanto, por su tiempo mínimo de ejecución (C_i^{act}) más los tiempos de ejecución de las operaciones invocadas ($C_{j,l}^{pass}$). Esto viene descrito en la siguiente ecuación:

$$C_i = C_i^{act} + \sum_{j=1}^{m} \left(C_{j,l}^{pass} * n_{j,l} \right) \quad (1)$$

Donde:

- C_i es el tiempo de cómputo en el peor caso de la tarea del componente activo.
- C_i^{act} es el tiempo mínimo de ejecución de la tarea (sin tener en cuenta las operaciones invocadas de otros componentes).
- m es el número de operaciones de otros componentes que son invocados por la tarea.
- $C_{j,l}^{pass}$ es el tiempo de cómputo en el peor caso de la operación l del componente j invocado por la tarea del componente i.
- $n_{j,l}$ es el número de veces que la operación l del componente j es invocado.

Los tiempos de cómputo suelen ser calculados en tiempo de diseño o de compilación (Stankovic, 2000; Tesanovic et al., 2003) pero en un entorno dinámico con reemplazo de componentes en tiempo de ejecución esta información resulta inútil tras una modificación del sistema. Por lo tanto esta información tiene que calcularse en tiempo de ejecución para poder aplicar correctamente los análisis de planificabilidad. Por lo tanto cada componente deberá incluir la información correspondiente a los tiempos de ejecución, como el tiempo mínimo de ejecución, las operaciones que son invocadas de otros componentes y el número de veces que son invocados. Esta información es usada por el marco en tiempo de ejecución para aplicar la Ecuación (1) y calcular el tiempo de cómputo extremo a extremo en el peor caso de cada tarea en el sistema. Este tiempo varía a lo largo de la ejecución del sistema dependiendo de los componentes que se estén ejecutando en cada momento.

3.4. Implementación del modelo de descripción de componente

Existen varias clases para representar la información sobre los componentes que necesita el marco para poder obtener el tiempo de cómputo en el peor caso de cada tarea. La Figura 1 muestra estas clases. Cada componente tiene asociada una clase que la describe (`ComponentDesc`). Esta clase contiene las dependencias del componente (`Dependency`), esto son las operaciones de las que depende para proporcionar su funcionalidad. También contiene la información sobre las operaciones que proporciona (`Operation`), así como las invocaciones realizadas por estas operaciones (`ServiceInvocation`).

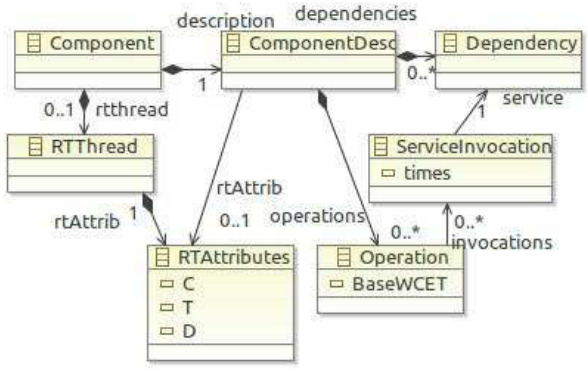

Figura 1: Clases de descripción del componente

La tarea de los componentes activos es de tipo `RTThread`, que deriva de la clase `RealtimeThread` de RTSJ. Estos componentes también deben incluir un objeto de tipo `RTAttributes` que incluye los parámetros temporales que deberán ser aplicados en los análisis de planificabilidad del marco.

`RTThread` se proporciona como una clase abstracta que ya se encarga de manejar ciertas características de la clase `RealtimeThread` de RTSJ. De esta manera proporciona una interfaz simplificada para que un componente sólo tenga que incluir el código que deba ser ejecutado en cada invocación de la tarea. También se proporciona un método (`wrapup`) de cierre para cuando el componente sea parado y la tarea sea eliminada del planificador (ver Listado 2).

```
public abstract class RTThread extends
FRealtimeThread {

   private boolean active = true;

   public void run() {

      while (active == true) {
         execute();
         waitForNextPeriod();
      }
      wrapup();
   }

   protected abstract void wrapup();
   protected abstract void execute();
}
```

Listado 2: Implementación de RTThread

Los componentes activos sólo requieren implementar una clase interna que extienda de `RTThread` con los métodos `execute` y `wrapup` implementados. Aunque los componentes activos no están restringidos al uso de esta clase la implementación de los componentes está diseñada para hacer uso del método `setActive` para detener la ejecución de la tarea cuando el componente es descargado.

3.5. Cálculo de WCET

Antes de poder instanciar cualquier componente en el sistema es necesario calcular el tiempo de cómputo en el peor caso (WCET) de dicho componente. Este WCET es desconocido hasta que una implementación concreta es cargada en el sistema. De la misma manera, cuando un componente es reemplazado, el tiempo asignado para la ejecución del componente depende de las necesidades de la nueva instancia. Hay que recalcular el WCET para la nueva instancia.

La información necesaria para realizar este cálculo está descrita y modelada para el marco en las secciones anteriores. El Listado 3 muestra cómo es calculado el WCET. En él se hace uso de dependencias inversas. Ésta es una lista en la que por cada componente se hace referencia a los que hacen uso de él. Las dependencias inversas se describen con más detalle en la siguiente sección.

```
CalculateWCET(Component):
   operations =
registry.getComponent(Component).getDescription().g
etOperations()
   for all O in operation do
      invocations = O.getInvocations()
      WCET = O.getBaseWCET()
      for all S in invocations do
         WCET = WCET + S.getTimes() *
S.getServiceOperation().getWCET()
      end for
      O.setWCET(WCET)
   end for
   rev_dependencies =
registry.getRev_Dependencies(Component)
   for all R in rev_dependencies do
      calculateWCET(R.getComponent)
   end for
```

Listado 3: Cálculo de WCET

Algunas partes del código mostrado en el Listado 3 han sido simplificadas para facilitar su comprensión. Para los componentes que dependan de éste no es necesario recalcular el WCET para todas sus operaciones, sino sólo para las operaciones específicas que dependan del componente a reemplazar. De manera similar la información contenida en la clase `Dependency` y su gestión es simplificada por razones de presentación.

4. Registro de componentes

El registro de componentes tiene un papel principal en el marco y está diseñado para manejar las instancias de componentes en ejecución, así como sus dependencias. También gestiona las clases de cada componente incluso si no hay ninguna implementación ejecutándose en ese momento. Aunque la clase `Framework` ofrece una interfaz para el manejo de componentes, es

la clase del registro (`ComponentReg`) la encargada del manejo de instancias (ver Figura 2).

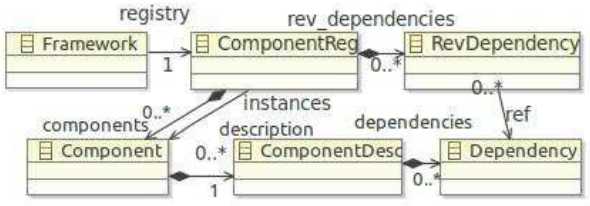

Figura 2: Modelo de registro de componentes

La clase `ComponentReg` implementa la mayor parte de la funcionalidad del registro. Además de la lista de componentes disponibles en el marco y de la lista de instancias también contiene una lista inversa de dependencias (`rev_dependencies`). Esta lista es la utilizada en el cálculo del WCET para calcular el peor tiempo de cómputo de cada componente en tiempo de ejecución. Para la aplicación de este algoritmo las dependencias de los componentes tienen que resolverse en orden inverso, del componente reemplazado o cargado a aquellos que hacen uso de él. Así no es necesario hacer una búsqueda de componentes que se vean afectados por el uso de la nueva implementación.

4.1. Actividades gestionadas por el registro

Las principales actividades de este marco son la carga, descarga y reemplazo de componentes en tiempo de ejecución. Debido a la complejidad en el manejo de los cargadores de clases de Java (*classloaders*) y el manejo de ficheros de empaquetado de clases (ficheros tipo *.jar*) sólo se consideran en el marco las clases que ya existen el sistema en tiempo de ejecución. En futuros trabajos este marco será integrado con OSGi, el cual permite la carga y descarga de código Java mediante ficheros de tipo *.jar*. A continuación se detallan las diferentes actividades que proporciona el marco. El reemplazo de componentes está descrito en la sección siguiente debido a su complejidad y al ser el principal objetivo de este trabajo.

Carga e instanciación de componentes

Dadas las restricciones anteriormente descritas la carga de un componente en el marco consiste en añadir la clase del componente al registro con su correspondiente descripción. Una vez que el componente ha sido registrado, entonces puede ser instanciado. Otros componentes serán instanciados recursivamente si han sido registrados pero todavía no han sido instanciados y éste componente depende de ellos.

El algoritmo en el Listado 4 muestra como las dependencias de los componentes son instanciadas si es necesario. Sólo si se han cubierto todas las dependencias del componente, entonces éste es arrancado.

```
Instantiate (Component):
   if (Component in registry.getInstances())
      return registry.getInstance(Component)
   end if
   if (Acceptance_test(Component) == false)
      return null;
   end if
   comp = new Component
   dependencies =
registry.getDependencies(Component)
   for all D in dependencies do
      inst = Instantiate(D)
      if (inst == null)
         return null;
      else
         comp.setDepend(D, inst)
   end for
   if (comp.start() == true)
      registry.addInstance(comp)
      return comp
   else
      return null
   end if
```

Listado 4: Proceso de instanciación de un componente

Otro punto importante en la instanciación de un componente es el test de aceptación. El sistema debe cumplir los requisitos descritos más adelante en la sección 5. Estos requisitos no son sólo de planificabilidad de los componentes sino también de la actividad de reemplazo, asegurando que el reemplazo de un componente es posible sin afectar a la planificabilidad del sistema. El test de aceptación consiste en actualizar el WCET de los componentes afectados por la carga o reemplazo del componente y modificar las características de la tarea de reemplazo de acuerdo con ello. Entonces se aplica un análisis de planificabilidad por parte del planificador. Si las modificaciones son correctamente planificables, entonces la instanciación o reemplazo de componente es aceptado.

Descarga de componente

La descarga de un componente consiste en parar la ejecución del componente, eliminar su tarea de ejecución de la lista de tareas del planificador si es un componente activo y eliminar su instancia de la lista de instancias del registro.

La implementación actual no elimina los componentes que dependen de éste. Éstos deben ser descargados de forma específica. Otros componentes podrían depender también de ellos, además del descargado.

5. Reemplazo de componentes

En esta sección se describen el modelo y la implementación del reemplazo de componentes del marco, gestionado por el registro de componentes. A partir de aquí se hace referencia principalmente a los componentes activos, que son los que representan mayor complejidad. Pero todo lo descrito es igualmente aplicable al reemplazo de componentes pasivos. Por cada componente pasivo reemplazado es necesario tener en cuenta sus conexiones y cómo modifica los tiempos de ejecución de los componentes activos que dependen de él.

Realizar el reemplazo del componente de forma segura es indispensable para mantener la seguridad en la ejecución del sistema. Reservar recursos es necesario tanto para las tareas de los componentes como para las actividades de reemplazo. El hecho de que las actividades de reemplazo se ejecuten siempre dentro de un tiempo reservado específicamente para este fin permite al marco asegurar los plazos temporales en todo momento. El cálculo de dicho tiempo lo realiza el marco en tiempo de ejecución.

Los pasos del proceso de reemplazo son los siguientes:

- Copiar el estado de la anterior implementación del componente a la nueva implementación.
- Reemplazar las conexiones del componente.

El número de conexiones viene dado por el número de dependencias del componente más el número de componentes que dependen del que se va a reemplazar. El tiempo mínimo que se requiere para completar el reemplazo del componente viene dado por la siguiente ecuación:

$$C^r_i = t_{state} + n*t_{bind} \quad (2)$$

Los componentes de la misma son:

C^r_i –tiempo total requerido en el peor caso para reemplazar el componente *i*.

t_{state} –tiempo requerido para transferir el estado de la antigua implementación del componente a la nueva.

t_{bind} –tiempo requerido para crear o reconfigurar cada conexión del componente (siendo *n* el número total de conexiones del componente).

El componente proporciona métodos específicos para ser utilizados durante el reemplazo. `getState` y `setState` permiten obtener el estado actual del componente y asignarlo a la nueva instancia. `setDepend` permite asignar a la nueva instancia las referencias a los componentes de los que depende. Para ello se le proporciona el objeto `Dependency` correspondiente y la referencia al componente del que depende.

El tiempo requerido para transferir el estado del componente puede ser conocido previamente o calculado basándose en el tamaño máximo de datos manejados por el componente. El tiempo requerido para reemplazar cada componente también puede ser conocido previamente y depende de la implementación del marco. Por lo tanto, siguiendo este protocolo de reemplazo de componente, el tiempo total requerido para completar la parte crítica del reemplazo de componente es conocido y acotado. Se reserva tiempo de procesador para la ejecución de esta tarea al igual que para el resto de componentes del sistema. Algunas propuestas, como (Rasche and Polze, 2005) se basan en parar la ejecución de la aplicación para realizar reconfiguraciones. En un entorno de tiempo real bloquear la ejecución de la aplicación no es aceptable por ello el marco propuesto mantiene una tarea de actualizaciones encargada de aplicar los reemplazos de componentes de forma segura en la cuota de procesador que tiene reservada para ello.

Haciendo uso de RMS (Lehoczky et al., 1989) el tiempo de procesador disponible para completar la actualización de un componente puede ser conocido. En el caso de una única tarea periódica el tiempo de procesador disponible comenzaría cuando terminase la ejecución del componente a reemplazar, hasta que el mismo componente vuelva a ser activado. Este tiempo libre tiene que ser igual o mayor al tiempo necesario para reemplazar el componente. La siguiente ecuación representa esta restricción:

$$C_i + C^r_i \leq T_i \quad (3)$$

Reservar tiempo de procesador para cada tarea de componente de manera que se pueda asegurar un reemplazo de componente cada vez que éste se ejecuta requiere añadir el tiempo necesario para el reemplazo del componente al tiempo de ejecución de dicha tarea. Algo a tener en cuenta es que el plazo de ejecución de la tarea del componente no se vería afectado ya que el reemplazo se aplica justo después de que la hebra termine su ejecución y antes de que vuelva a ser ejecutada. La prioridad asignada a estos reemplazos puede ser la misma del componente para asegurar que se lleve a cabo en tiempo. Modificando la ecuación de tiempo de respuesta dada por (Lehoczky et al., 1989) añadiendo el tiempo de reemplazo a cada hebra tenemos:

$$R_i = C_i + \sum_{j=\mathrm{hp(i)}} \left\lceil \frac{R_i}{T_j} \right\rceil \left(C_j + C^r_j\right) \quad (4)$$

La Ecuación (4) representa el tiempo de respuesta de la hebra *i* donde el tiempo requerido por cada componente de mayor prioridad para ser reemplazado (C^r_j) es añadido a su tiempo de ejecución (C_j) para calcular su interferencia sobre las hebras de menor prioridad. El tiempo de reemplazo del componente *i* no se incluye porque no entra dentro de su plazo de ejecución.

Este método puede representar un gran sobrecoste en reserva de procesador en caso de que no se realice un reemplazo de componente cada vez que es ejecutado. Frente a este método pesimista, el marco incluye también otro método mucho más selectivo que representa una solución de compromiso. Se puede reservar un tiempo menor de procesador a cambio de reducir el número de reemplazos permitidos durante un tiempo especificado. El tiempo de procesador reservado para reemplazo es compartido por los componentes haciendo uso de una tarea común para reemplazos. Ésta es una tarea adicional del marco encargada de realizar los reemplazos solicitados.

Debería ser suficiente con asignar la misma prioridad a la tarea de reemplazo que al componente que va a reemplazar, para que no se vea interrumpida por el componente. Pero eso implicaría cambiar la prioridad de la tarea de reemplazo en cada ejecución, introduciendo mucha variación en la interferencia creada en el resto de tareas. Además, no siempre es posible variar la prioridad de una tarea en cada una de sus ejecuciones. Por ello aquí se considera como opción más aplicable y realista el asignar la prioridad máxima a la tarea de reemplazo. Esto también facilita los análisis de planificabilidad del sistema.

La tarea será ejecutada cuando el componente a reemplazar no se encuentre activo. La Ecuación (3) asegura que el tiempo de reemplazo es inferior al tiempo entre ejecuciones del componente. Esto también asegura que no hay problemas de accesos concurrentes a los datos del componente.

El primer paso para calcular el efecto de esta tarea en el análisis de planificabilidad es considerarla como una nueva tarea añadida a la lista de tareas, pero ejecutada con una prioridad superior que el resto para que no sea interrumpida. Si esta tarea es interrumpida no se cumplirá su plazo de ejecución.

Para asegurar que se reserva suficiente tiempo de procesador para reemplazar cualquier componente, el tiempo de cómputo en el peor caso de esta tarea debe ser igual al mayor de los tiempos necesarios para reemplazar cualquiera de los componentes:

$$C_r = \max(C^r_1,..,C^r_m) \quad (5)$$

El efecto de la ejecución de esta tarea con la máxima prioridad en el análisis de planificación del conjunto de tareas quedaría de la siguiente manera:

$$R_i = C_i + \sum_{j=hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + I_r \quad (6)$$

Donde I_r representa la interferencia creada por la tarea de reemplazo de componentes de la siguiente manera:

$$I_r = \left\lceil \frac{R_i}{T_r} \right\rceil C_r \quad (7)$$

Desde la inicialización del sistema debe existir una tarea para reemplazo de componentes con un periodo de ejecución T_r, un tiempo de cómputo en el peor caso de C_r y un plazo de ejecución igual al tiempo de cómputo en el peor caso de D_r. Los parámetros de esta tarea se actualizarán de acuerdo a las modificaciones que se produzcan en el sistema, como carga, descarga o reemplazo de componentes. Se establece que el plazo de ejecución sea igual al tiempo de cómputo ($C_r = D_r$) porque de otra manera representaría que la tarea de reemplazo ha sido interrumpida, condición impuesta anteriormente.

El periodo de ejecución de la tarea de reemplazo (T_r) viene determinado por el tiempo de procesador reservado o por el tiempo mínimo entre llegadas de peticiones de reemplazo. Aquí se describen dos formas de calcularlo.

Si la intención es reservar tiempo de procesador para otras tareas de mantenimiento, además de para reemplazar componentes, se puede calcular T_r basándose en el tiempo de procesador que se desea utilizar para dichas tareas. La ecuación sería la siguiente:

$$T_r = \frac{100 * C_r}{U} \qquad (8)$$

Donde *U* representa el porcentaje de procesador reservado para tareas de mantenimiento en general o específicamente para la tarea de reemplazo de componentes. El tiempo de procesador reservado podrá usarse para otras funciones cuando no se dedique a reemplazo de componentes.

Por otro lado puede esperarse un número mínimo de reemplazos en un tiempo especificado (Mínimum Inter-arrival Time, MIT). Este tiempo será el que determine el periodo de ejecución de la tarea de reemplazo (T_r = *MIT*).

En la implementación descrita en este trabajo se hace uso del modelo de reemplazo de componentes selectivo en el que la tarea de reemplazo es compartida por todos los componentes y un tiempo de procesador es reservado para la ejecución de dicha tarea. La implementación consiste en una tarea periódica que se ejecutará de acuerdo con los parámetros ya descritos descritos (C_r, T_r and D_r).

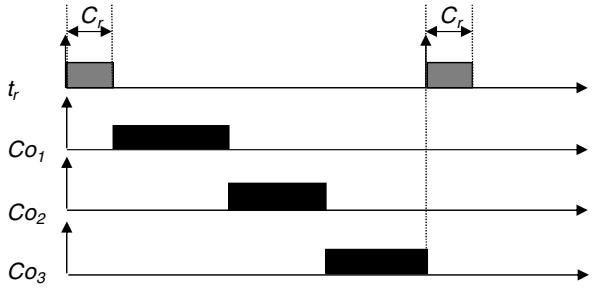

Figura 3: Modelo de reemplazo selectivo de componente

En la Figura 3 se puede apreciar el modelo implementado en el que el tiempo de la tarea de reemplazo es compartido por todos los componentes. En este caso por cada ejecución de la tarea de reemplazo sólo podrá ser reemplazado un componente. Ya que se reserva tiempo de procesador para el mayor tiempo de reemplazo requerido, como se indica en la Ecuación (5), se puede realizar el reemplazo de cualquiera de los componentes.

5.1. Modelo de tarea de reemplazo de componentes

Hace falta una infraestructura para aplicar los reemplazos de componente. Para ello es necesaria una tarea para reemplazos de componentes, como ya se ha descrito. Esta tarea recibe peticiones de reemplazo y las irá aplicando en orden. Primero debe crearse una petición de reemplazo indicando el nuevo componente y el antiguo componente a reemplazar. Esta petición se añade a la lista de reemplazos. Cuando la tarea de reemplazos es planificada para su ejecución seleccionará la primera de las peticiones y la aplicará.

Para que los reemplazos puedan ser totalmente controlados por el marco, algunas clases de RTSJ deben ser extendidas (ver Figura 4). `FScheduler` es un planificador basado en el `PriorityScheduler` aportado por la especificación de RTSJ. En él se han implementado la gestión de tareas de componentes y los test de aceptación basados en el análisis de planificabilidad de RUB (Límite superior de tiempo de respuesta, Response-Time Upper Bound) (Bini et al., 2009). `FRealtimeThread` extiende `RealtimeThread` añadiendo algunas características como la posibilidad de añadir o eliminar la tarea del conjunto de tareas del planificador.

`UpdateTask` extiende la clase `FRealtimeThread` para implementar la tarea de reemplazo. Esta tarea recibirá las peticiones de reemplazo en forma de objetos de tipo `UpdateOrder`. La petición de reemplazo hará referencia al componente a reemplazar y a la nueva instancia.

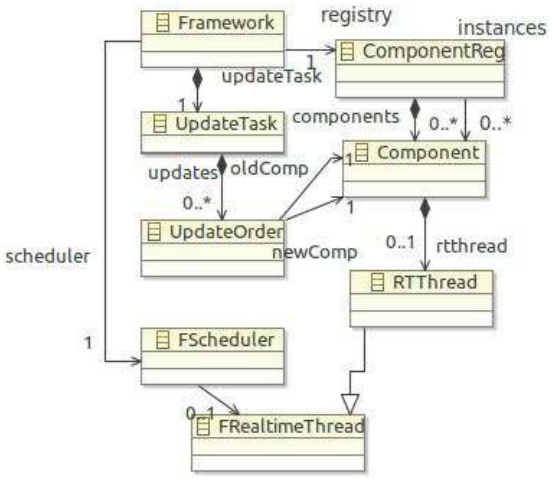

Figura 4: Modelo de tarea de reemplazo del marco

5.2. Implementación de la tarea de reemplazo

Aquí se describe más en detalle la implementación de la actividad de reemplazo. Esta implementación difiere de las actividades de carga y descarga de componentes ya que estas actividades no tienen requisitos específicos de tiempo para que su ejecución sea correcta. Esta actividad, como ya se ha descrito, tiene requisitos de tiempo para que la ejecución de los componentes sea correcta.

Los parámetros de esta tarea son actualizados cada vez que un componente es aceptado en el marco. También, de forma

diferente a la carga y descarga de componentes, los reemplazos se efectúan de forma asíncrona. Esto quiere decir que el reemplazo no se realiza en el momento en el que se solicita, sino en el momento en el que la tarea se planifica para ser ejecutada. La Figura 5 representa el proceso completo. Una vez que se pide el reemplazo de componente al marco éste crea una petición de reemplazo y se la entrega al servidor de tarea de reemplazo. Éste lo añade a la lista de reemplazos pendientes hasta la próxima ejecución de la tarea. Hasta ese momento todo el proceso se realiza en segundo plano usando tiempo libre del procesador. Esto es posible debido a que para estos pasos no hay restricciones de tiempo.

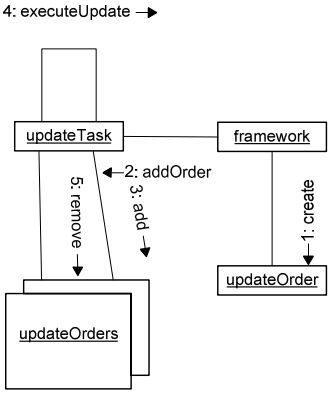

Figura 5: Proceso de gestión de reemplazos

El reemplazo en sí es realizado por la tarea de reemplazo con prioridad máxima. Hay que tener en cuenta que la ejecución del reemplazo será retrasada si el componente a ser reemplazado se encuentra en ejecución. El componente se considera como un recurso compartido. Esto asegura la correcta ejecución del reemplazo. Debido a que ha sido reservado el tiempo de procesador necesario, no se producen retrasos en la ejecución del resto de tareas.

```
Update(OldComponent, NewComponent):
   oldInstance = registry.getInstance(OldComponent)
   newInstance = new NewComponent
   newInstance.setState(oldInstance.getState())
   dependencies =
registry.getDependencies(NewComponent)
   for all D in dependencies do
      inst = registry.getInstance(D.getComponent())
      if (inst == null)
         return null;
      end if
      else
         newInstance.setDepend(D, inst)
   end for
   rev_dependencies =
registry.getRevDependencies(NewComponent)
   for all R in rev_dependencies do
      dep_component =
registry.getInstance(R.getComponent())
      dep_component.setDepend(R.getDepend(),
newInstance)
      newInstance.setState(oldInstance.getState())
   end for
```

Listado 5: Reemplazo de componente

El Listado 5 describe el proceso de reemplazo de un componente. El nuevo componente es instanciado y el estado es copiado de la anterior instancia a la nueva. Entonces se configuran las conexiones. Primero se configuran los componentes de los que depende éste. Por simplicidad las dependencias se suponen ya instanciadas en el listado. Después la lista de dependencias inversas es usada para configurar las conexiones con los componentes que dependen de éste. En este momento el nuevo componente ha reemplazado al anterior.

6. Recolector de basura de Java

Uno de los mayores problemas en cuanto a las implementaciones de sistemas de tiempo real con la Máquina Virtual de Java es el uso de un recolector de basura (Nilsen, 1996) para la gestión automática de memoria. La ejecución inesperada del recolector de basura rompe la ejecución planificada de las tareas de tiempo real. Se han aportado soluciones para minimizar el efecto de la ejecución del recolector de basura para sistemas de tiempo real en Java (Bacon et al., 2003; Nilsen, 1998; Pizlo and Vitek, 2008).

RTSJ aporta algunas alternativas para evitar o disminuir el uso del recolector de basura. Tanto el uso de hebras uso de memoria recolectable (*no heap threads*) como el uso de memoria de ámbito (*scoped memory*) eliminan el uso de memoria dinámica y obligan al programador a gestionar la memoria usada. De esta manera se puede reducir al mínimo la necesidad de un recolector de memoria.

Para poder mantener la compatibilidad de código ya existente aquí se ha optado por otra opción. Ésta consiste en el uso de una máquina virtual cuya implementación incluye un recolector de basura con características de tiempo real. En este caso se ha usado JamaicaVM (Siebert, 2007), que proporciona un recolector de basura cuya interferencia en la ejecución del código, aunque existente, está acotada.

7. Validación empírica

Las pruebas realizadas se han diseñado de manera que se ha reducido al mínimo la interferencia creada por el recolector de basura en los tiempos de ejecución. Para todas las pruebas de reemplazo se ha dividido la ejecución de las pruebas en dos partes. La primera parte está dedicada a la creación, registro e instanciación de todos los componentes. De esta manera durante las pruebas de reemplazo no es necesario reservar y liberar memoria, evitando al mínimo el uso del recolector de basura.

El entorno de pruebas consiste en un Core 2 Duo a 2.2Ghz, con 4Gb de memoria RAM. El sistema operativo es una distribución Ubuntu 12.04.1 con una versión del kernel 3.2.0 modificada para incluir características de tiempo real (Wiki 2012). La máquiva virtual de Java utilizada, como se ha descrito, es la JamaivaVM versión 6.1.2.

La segunda parte de las pruebas consiste básicamente en realizar reemplazos de componentes. Dada una configuración previa de componentes ya en funcionamiento se selecciona de forma aleatoria una implementación diferente de las disponibles para reemplazar a un componente de la configuración actual.

Para poder medir el tiempo necesario para reemplazar un componente, los componentes para el reemplazo son seleccionados de forma aleatoria. Los métodos usados para copiar el estado del componente son los descritos en la sección 5. Para estas pruebas los datos manejados por el componente son

modelados como un objeto, por lo que la copia consistirá básicamente en copiar dicho objeto (su referencia) a la nueva instancia del componente.

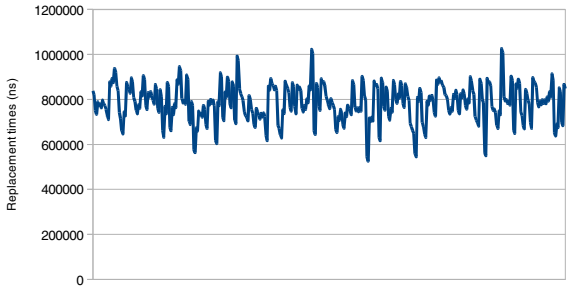

Figura 6: Media de tiempo de ejecución de reemplazo

En la Figura 6 se muestra la media de los tiempos de la secuencia de reemplazos de todas las ejecuciones realizadas. Se puede apreciar que, al seguir usando memoria recolectada se producen interferencias en la ejecución. También se puede apreciar que estas interferencias se encuentran dentro de unos límites.

Tabla 1: Tiempos de reemplazo de componentes (ms)

Tiempo medio de reemplazo	0,803
Tiempo máximo de reemplazo	1,477

La Tabla 1 muestra el tiempo medio necesario para completar un reemplazo de componente, que en este caso es de 0,803 milisegundos. El tiempo máximo requerido para realizar un reemplazo durante todas las pruebas ejecutadas es de 1,477 milisegundos. Estos tiempos podrán variar en el futuro debido a que dependen de otras complejidades del marco, como el método de la copia de estado o el tamaño de éste.

Estas mismas pruebas fueron utilizadas para medir el tiempo requerido para completar el test de aceptación. Además del análisis de tiempo de respuesta exacto (*Response Time Analysis, RTA*) basado en (Lehoczky et al., 1989), descrito en este caso por las ecuaciones (6) y (7), también ha sido implementado RUB (Bini et al., 2009). RUB reduce los tiempos de computación de los análisis de planificabilidad de forma significativa a costa de aceptar un menor número de componentes. Dado que la mayor parte del tiempo se debe al test de aceptación el uso de RUB reduce significativamente el tiempo de ejecución de este proceso. La Tabla 2 muestra la diferencia de tiempos en el test de aceptación haciendo uso de RTA y de RUB.

Tabla 2: Tiempos de test de aceptación (ms)

RTA – Tiempo medio	552
RUB – Tiempo medio	71

El tiempo medio del test de aceptación para RTA son 552 milisegundos, mientras que para RUB es de 71 milisegundos. Esto supone una cantidad de tiempo significativa ahorrada mediante el uso de RUB debido a su menor coste computacional. Aunque un número menor de componentes serían aceptados (Bini et al., 2009) ésta reducción no es significativa.

Los algoritmos aquí implementados pueden ser optimizados en varios aspectos, pero estas optimizaciones dependerán de las características finales de la implementación del marco.

Como también se ha indicado, la implementación de este marco de componentes en Java hace posible una futura integración con OSGi para hacer uso de su capacidad de reemplazo de paquetes y apreciar los beneficios del reemplazo de componentes propuesto en este trabajo.

8. Conclusiones

Soportar dinamismo en los sistemas de tiempo real requiere el empleo de mecanismos que determinen el máximo coste temporal de las tareas en tiempo de ejecución. En entornos de componentes, el cálculo del tiempo de reemplazo de componente requiere la adecuada caracterización temporal de los componentes. En este trabajo se aporta una solución para conseguir reemplazos seguros en tiempo de ejecución ofreciendo un modelo de componente simple que integra las propiedades temporales necesarias para realizar un análisis de planificabilidad. El tiempo necesario para los reemplazos es integrado en la planificación de tiempos del sistema de manera que se reserva tiempo de procesador para su realizaciónn.

Se ha implementado el modelo de componentes que de forma segura con operaciones atómicas para realizar las acciones básicas como el cálculo de tiempos de reemplazos y la realización del reemplazo mismo. Existe una tarea de alta prioridad (mayor que la del resto de componentes) cuyo tiempo asignado se basa en los requisitos de tiempo de reemplazo de los componentes del sistema. La frecuencia de ejecución de dicha tarea deberá es calculada previamente mediante los métodos aportados.

Este trabajo prueba que es posible implementar un marco de componentes en Java con capacidades dinámicas. Además del cálculo del WCET en tiempo de ejecución de los componentes activos, la principal aportación es la capacidad de reemplazar los componentes que hacen uso de este marco en tiempo de ejecución de forma segura y transparente, sin romper su ejecución.

Como trabajo futuro se integrará este marco con la plataforma OSGi, teniendo en cuenta las limitaciones de RTSJ y del recolector de basura.

English Summary

Component Framework for supporting safe and dynamic replacement in real-time systems

Abstract

In the last decades solutions have been provided for the real-time component-based systems development as a base to increase productivity and reliability of their development as well as their maintenance. Solutions are increasingly appearing that allow controlled flexibility in these systems, aiming to support dynamic execution through the component replacement at run-time. So, component models are adapted trying to minimize conflicts integrating real-time and dynamic behaviors, and achieving components replacements in a bounded time. One of the main challenges for this is to calculate the required times by the different operations needed in a component replacement. The other issue is to know the operating times of the component in the

system when their implementations change along the life of the system. In this work the implementation of a component framework implementation is described providing a partial solution for these problems. A component model is provided together with the corresponding algorithms to assure that components can be loaded and replaced at run-time without interfering in their execution deadlines. The model is designed to avoid failures during component replacements. Finally a validation of the presented concepts is provided.

Keywords:

Components frameworks, real-time, dynamic systems, component replacement, reconfiguration

Referencias

Almeida, J., Wegdam, M., 2001. Transparent dynamic reconfiguration for CORBA, in: Blair, G., Schmidt, D., Tar, Z. (Eds.), 3rd International Symposium on Distributed Objects and Applications. Rome, pp. 197–207.

Bacon, D.F., Cheng, P., Rajan, V., 2003. A real-time garbage collector with low overhead and consistent utilization, in: ACM SIGPLAN Notices. ACM, pp. 285–298.

Bini, E., Cha, T.H., Richard, P., Baruah, S.K., 2009. A Response-Time Bound in Fixed-Priority Scheduling with Arbitrary Deadlines. IEEE Transactions on Computers 58, 279–286.

Bollella, G., Gosling, J., 2000. The real-time specification for Java. IEEE Computer 33, 47–54.

Bruneton, E., Coupaye, T., Leclercq, M., 2006. The fractal component model and its support in Java. Software: Practice and Experience 36, 1257–1284.

Bures, T., Hnetynka, P., Plasil, F., 2006. Sofa 2.0: Balancing advanced features in a hierarchical component model, in: Software Engineering Research, Management and Applications, 2006. Fourth International Conference On. IEEE, pp. 40–48.

Canal, C., Murillo, J., Poizat, P., 2006. Software adaptation. L'objet 12, 9–31.

Cano, J., García-Valls, M., 2013. Scheduling component replacement for timely execution in dynamic systems. Software: Practice and Experience.

Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N., 2001. An efficient component model for the construction of adaptive middleware. Middleware 2001 1–15.

Cornwell, P., Wellings, A., 1996. Transaction integration for reusable hard real-time components. Proceedings. IEEE High-Assurance Systems Engineering Workshop (Cat. No.96TB100076) 166–175.

Crnkovic, I., Sentilles, S., Aneta, V., 2011. A classification framework for software component models. IEEE Transactions on Software Engineering 37, 593–615.

Garcia-Valls, M., Alonso Muñoz, A., Ruiz, J., Groba, A., 2003. An Architecture of a Quality of Service Resource Manager for Flexible Multimedia Embedded Systems. Proceedings of 3rd International Workshop on Software Engineering and Middleware. LNCS 2596, 36–55.

García-Valls, M., Alonso, A., De la Puente, J., 2009. Mode change protocols for predictable contract-based resource management in embedded multimedia systems, in: Embedded Software and Systems, 2009. ICESS'09. International Conference On. IEEE, HangZhou, pp. 221–230.

García-Valls, M., Alonso, A., De la Puente, J.A., 2012a. A dual-band priority assignment algorithm for dynamic QoS resource management. Future Generation Computer Systems 28, 902–912.

García-Valls, M., Basanta-Val, P., Estevez-Ayres, I., 2011. Real-time reconfiguration in multimedia embedded systems. Consumer Electronics, IEEE Transactions on 57, 1280–1287.

García-Valls, M., Rodriguez Lopez, I., Fernández Villar, L., 2012b. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time System. IEEE Transactions on Industrial Informatics 9, 228–236.

García-Valls, M., Basanta-Val, P., 2013a. A real-time perspective of service composition: Key concepts and some contributions. Journal of Systems Architecture. Elsevier, http://dx.doi.org/10.1016/j.sysarc.2013.06.008

García-Valls, M., Basanta-Val, P., Marcos, M., Estévez, E., 2013b. A bi-dimensional QoS model for SOA and real-time middleware. International Journal of Computer Systems Science and Engineering, CLR Publishing, ISSN 0267-6192. (To appear)

Isovic, D., Lindgren, M., 2000. System development with real-time components. Proceedings of ECOOP Workshop - Pervasive Component-Based Systems.

Kramer, J., Magee, J., 1990. The evolving philosophers problem: Dynamic change management. Software Engineering, IEEE Transactions on 16, 1293–1306.

Lehoczky, J., Sha, L., Ding, Y., 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior, in: Real Time Systems Symposium. IEEE Comput. Soc. Press, Santa Monica, California, USA, pp. 166–171.

Li, W., 2011. QoS Assurance for Dynamic Reconfiguration of Component Based Software Systems. IEEE Transactions on Software Engineering 38, 658–676.

McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C., 2004. Composing adaptive software. Computer 37, 56–64.

Miguel, M. De, Ruiz, J., García-Valls, M., 2002. QoS-aware component frameworks. 10th IEEE Int'l Workshop on Quality of Service 161–169.

Nilsen, K., 1996. Issues in the design and implementation of real-time Java.

Nilsen, K., 1998. Adding real-time capabilities to Java. Communications of the ACM 41, 49–56.

OMG, 2006. CORBA Component Model Specification, Management.

OMG, 2009. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems.

OSGi Alliance, 2009. OSGi Service Platform Release 4 Service Compendium Version 4.2, Access.

Pizlo, F., Vitek, J., 2008. Memory management for real-time java: State of the art, in: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium On. IEEE, pp. 248–254.

Plšek, A., Loiret, F., Merle, P., Seinturier, L., 2008. A component framework for java-based real-time embedded systems, in: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware. Springer-Verlag New York, Inc., Leuven (Belgium), pp. 124–143.

Quadri, I.R., Muller, A., Meftali, S., Dekeyser, J., 2010. MARTE based design flow for Partially Reconfigurable Systems-on-Chips, in: 17th IFIP/IEEE International Conference on Very Large Scale Integration. Florianapolis, Brazil.

Rasche, A., Polze, A., 2005. Dynamic reconfiguration of component-based real-time software, in: 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. WORDS 2005. IEEE, Sedona, Arizona, pp. 347–354.

Rasche, A., Polze, A., 2008. ReDAC–Dynamic Reconfiguration of Distributed Component-Based Applications with Cyclic Dependencies, in: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium On. IEEE, Orlando, Florida, pp. 322–330.

Sha, L., 1998. Dependable system upgrade, in: Real-Time Systems Symposium,. Madrid, Spain, pp. 440–449.

Siebert, F., 2007. Realtime garbage collection in the JamaicaVM 3.0. Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems - JTRES '07 94.

Stankovic, J., 2000. VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems. University of Virginia TRCS-2000-19.

Tesanovic, A., Hansson, J., Te Usanovic, A., Nyström, D., Norström, C., 2003. Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components. Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03).

Tewksbury, L.A., Moser, L.E., Melliar-Smith, P.M., 2001. Live upgrades of CORBA applications using object replication. ICSM '01 Proceedings of the IEEE International Conference on Software Maintenance 488–497.

Vandewoude, Y., Ebraert, P., Berbers, Y., Hondt, T.D., 2007. Tranquillity: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. Software Engineering, IEEE Transactions on 33, 856–868.

Wahler, M., Richter, S., Kumar, S., Oriol, M., 2011. Non-disruptive large-scale component updates for real-time controllers, in: International Conference on Data Engineering Workshops (ICDEW), 2011 IEEE 27th. IEEE, Hannover (Germany), pp. 174–178.

Warren, I., Sun, J., Krishnamohan, S., 2006. An automated formal approach to managing dynamic reconfiguration, in: IEEE International Conference on Automated Software Engineering.

Wiki, https://rt.wiki.kernel.org/index.php/Main_Page, February, 2013