Trabajo Fin de Máster

HERRAMIENTA PARA EL MODELADO Y ANÁLISIS DE PLANIFICABILIDAD DE APLICACIONES RT-JAVA SOBRE RT-LINUX

(Modeling and schedulability analysis tool for RT-Java applications on RT-Linux)

Para acceder al Título de

MÁSTER EN COMPUTACIÓN

Autor: Jose María Martínez Lanza

Mayo - 2013

Índice

Índice de figuras

Índice de tablas

Resumen

Actualmente, buena parte de la tecnología y la industria utiliza el lenguaje de programación Java en el desarrollo de sistemas industriales. Estos sistemas, en ocasiones, se implementan bajo la especificación RTSJ (Real-time System Java), que dota a este lenguaje de los recursos necesarios para el desarrollo de aplicaciones de tiempo real, bien sea laxo o estricto. Pero el desarrollo de una aplicación de tiempo real, además de formular su código, requiere de la configuración de su planificabilidad para garantizar el cumplimiento de sus requisitos.

En este trabajo se desarrolla un conjunto de herramientas en RT-Java cuyo objetivo final ha sido la construcción de un modelo MAST 2.0 que permita analizar aplicaciones de tiempo real basadas en la metodología de la orientación a objetos y, más concretamente, en los patrones de diseño impuestos por el modelo de referencia OORT. Para ello se desarrollan herramientas que evalúan tanto el modelo de la plataforma en la que va a ser desplegada la aplicación (prioridades, tiempos de cambio de contexto, etc.), como el modelo lógico y reactivo de las aplicaciones diseñadas e implementadas bajo este modelo de referencia (evaluación de los tiempos de ejecución de thread y métodos), una herramienta que instrumenta el código de la aplicación de tiempo real para las medidas de tiempos y, finalmente, una herramienta que dados todos los datos e información recogidos construya el modelo MAST correspondiente a la aplicación ejecutada.

Así mismo, se desarrolla una aplicación de prueba con el fin de validad el proceso propuesto en el trabajo.

I Aplicaciones RT-Java y modelo de referencia

I.1 Aplicaciones de tiempo real con RT-Java

La reducción del costo de la tecnología permite que actualmente los sistemas embebidos puedan estar dotados de procesadores, memoria y sistemas operativos que soportan paradigmas de programación avanzados. En el diseño de estos sistemas, son más relevantes los aspectos de I/O interfacing con el entorno físico y los requerimientos de tiempo real estricto, que los aspectos tradicionales de limitación de capacidad y de recursos. *La metodología de diseño de subsistemas de tiempo real que se describe en esta memoria, ha sido desarrollada para el proyecto Energos* [1] *que ha abordado el desarrollo de tecnología Smart Grid para la gestión de la producción, transmisión y consumo en redes eléctricas utilizando un middleware DDS* [2]. En estos casos, la plataforma en la que se despliegan las aplicaciones es típicamente heterogénea y está constituida tanto por potentes servidores situados en los centros de control, como de un gran número de pequeños sistemas embarcados en las unidades remotas de teleoperación, en los sistemas de protección, en los puntos de integración de contadores, etc.. En estos sistemas embebidos, el desafío tecnológico está en poder aplicar tecnologías software compatible con los niveles superiores, y en la capacidad de componentización para el despliegue y reconfiguración dinámico en los equipos. Todo ello sin perder la capacidad de acceso al entorno físico y de ofrecer servicios y respuestas con los requisitos temporales que requieren.

Actualmente la tecnología Java se ha impuesto en amplios sectores del desarrollo de sistemas embebidos, porque embarcar en el procesador del sistema embebido una máquina virtual Java (JVM) y desarrollar las aplicaciones utilizando el lenguaje de programación Java es muy atractivo ya que:

1) Incrementa la productividad de los programadores, al inducir la utilización de programación orientada a objetos, e incorporar a través del propio lenguaje la implementación de un gran número de patrones de diseño [1] [2].

2) Facilita la portabilidad del software a cualquier plataforma, para la que existe una máquina virtual Java (JVM). El principio básico del lenguaje Java es “Write Once, Run Anywhere”.

3) Facilita la componentización. Java adopto desde su inicio la reutilización del código a través de la capacidad de construir componentes. Inicialmente con los JavaBeans[3] para construir aplicaciones, y posteriormente con los EJB (Enterprise JavaBeans) [4] para la construcción servidores distribuidos que pueden ser desplegados independientemente.

4) Tiene una alta aceptación industrial porque se adapta muy bien al desarrollo inter-empresarial.

5) Tiene una actualización muy dinámica que incorpora ágilmente las innovaciones que aparecen en ingeniería de software.

La especificación RTSJ (Real-Time System Java) que recientemente ha sido formulada JSR-1 [5] y JSR-282 [6] y las implementaciones que de ellas están actualmente disponibles [7], [8] y [9] incorporan a la JVM y al lenguaje de programación RT-Java los recursos que se necesitan

para el desarrollo eficiente de aplicaciones de tiempo real dentro de un amplio espectro de aplicaciones industriales. Algunos de estos recursos son:

1) Threads de tiempo real con planificadores de tiempo real basado en prioridades fijas.
2) Gestión de memoria con tiempos de acceso acotados.
3) Mecanismos de sincronización entre threads que evitan la inversión de prioridad.
4) Gestión eficiente de eventos externos y mecanismos de transferencia de control asíncronos.
5) Recursos para acceder de forma sencilla a la memoria física.
6) La incorporación al lenguaje de un amplio conjunto de patrones de diseño de tiempo real [8].

Con todo esto, RTSJ ofrece capacidad para desarrollar aplicaciones de tiempo real estricto, manteniendo las ventajas que son tradicionales en Java. Sin embargo, el diseño de una aplicación de tiempo real no sólo requiere formular su código, sino que debe ser configurada su planificabilidad para garantizar el cumplimiento de los requisitos de tiempo real. Evaluar esta configuración es una tarea compleja ya que depende de la capacidad de procesamiento que requiere la aplicación, de la capacidad que ofrece la plataforma en que se ejecuta, y de la carga de trabajo que requiere el entorno físico sobre el que opera.

La configuración de la planificabilidad de la aplicación requiere formular un modelo de comportamiento temporal que sirva de base a las técnicas de análisis de planificabilidad. Esta tarea es compleja en cualquier sistema de tiempo real, pero en el caso de aplicaciones diseñadas en base al paradigma de orientación a objetos (OO), como son las que resultan cuando se utiliza RTSJ, presenta dos dificultades añadidas:

1) El comportamiento reactivo en que se basa el modelo de tiempo real, está oculto en el código de las clases de la aplicación. Se requeriría realizar un análisis detallado del códigos de muchas clases de la aplicación, y secuencializar las invocaciones de los métodos entre objetos que se realizan, para identificar la cadena de actividades que implementan una respuesta.
2) Así mismo, en cualquier sistema de tiempo real, al evaluar los tiempos de ejecución de peor caso del código de las respuestas y del uso de los recursos es una tarea muy costosa, pero en el caso de las aplicaciones orientadas a objetos es mucho más costosa ya que depende de las múltiples invocaciones entre métodos que se ocultan en el código y que varían en función del despliegue y configuración que se haga de la aplicación.

En este trabajo se parte del modelo de referencia OORT para el diseño de las aplicaciones de tiempo real basadas en RT-Java que hace uso de los patrones implementados por el propio lenguaje, pero sistematiza su uso a fin de que se facilite la generación automática del modelo de tiempo real en base a la ejecución de la aplicación sobre la plataforma final en la que va a ser instalada.

Como se muestra en la Figura I.1, una aplicación desarrollada de acuerdo con este modelo de referencia se diseña de acuerdo con unos patrones sistemáticos, que pueden ser

instrumentados mediante un generador de código automático, de forma que ejecutada sobre el procesador embebido final y ejecutada en un entorno de prueba suficientemente rico permite generar automáticamente el modelo de planificabilidad que, una vez procesado, proporciona la configuración que debe aplicarse a la aplicación cuando se ejecuta sobre el sistema final.

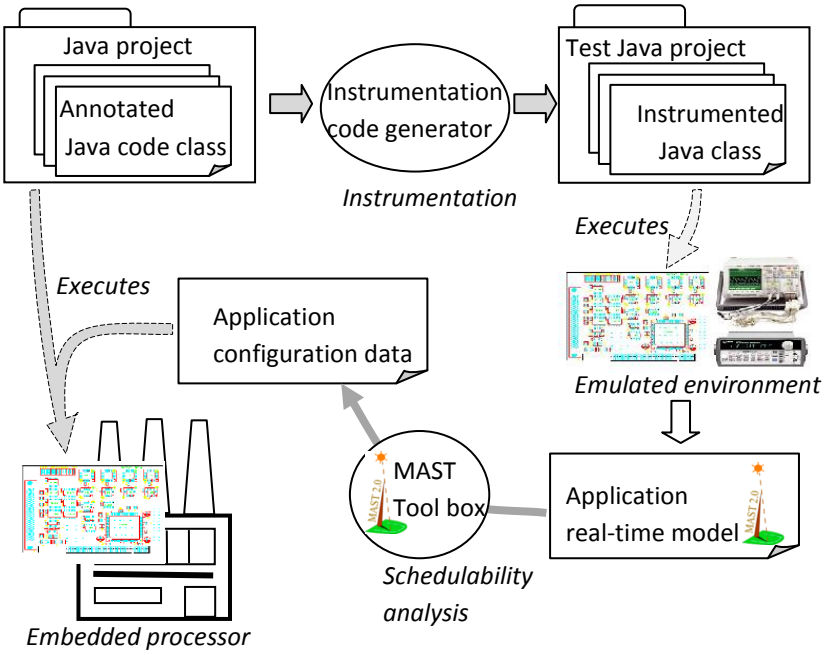


Figura I.1: Proceso de configuración de la planificabilidad de una aplicación

Esta estrategia no puede aplicarse a sistemas críticos de tiempo real estricto, ya que los tiempos de ejecución de peor caso no se basan en el análisis de cobertura de todas las posibilidades de ejecución del código, sino en las medidas hechas sobre la ejecución el sistema cuando este opera en el entorno de configuración, que es una emulación del entorno real pero enriquecido con todas las alternativas posibles que admite su especificación. Por ello, la metodología utilizada es aplicable a sistemas de tiempo real laxo o "firm real-time system" ("*a task is said to be firm if missing a timing constraint does not jeopardize the correct system behavior but it is completely useless for the system*" [10]) en los que no se requieren certificar la imposibilidad de ocurrencias de fallos. Esta estrategia de modelado es admitida para la mayoría de las aplicaciones industriales si excluimos algunas muy críticas como la industria aviónica, aeroespacial o nuclear. El gran beneficio que se obtiene con la estrategia propuesta es que se aplicar a sistemas muy complejos y que utilizan las tecnologías más avanzadas (como RTSJ) aunque no sea certificable.

I.2 Objetivos del trabajo

El objetivo de este trabajo ha sido el desarrollo del entorno que permite construir el modelo MAST 2 [11] para analizar y configurar las aplicaciones de tiempo real diseñadas de acuerdo con el modelo de referencia OORT, en base a la ejecución de la aplicación en un procesador igual que al que va a existir en el entorno de ejecución industrial, solo en un entorno de laboratorio lo suficientemente rico en ocurrencias de eventos para que todos los

tipos especificados en el entorno de ejecución real ocurran de forma repetida durante el tiempo de construcción del modelo, a fin de que sus tiempos puedan se cuantificados estadísticamente.

Los objetivos específicos de este trabajo han sido:

1. Desarrollo de un programa de evaluación del modelo de la plataforma que ejecutado sobre el procesador del entorno de configuración, permita estimar la capacidad del procesador y los tiempos de peor, mejor y caso promedio del planificador y de los relojes.
   1.1. Identificación de los elementos relevantes del modelo MAST 2 de la plataforma RT-Java sobre Ubuntu Linux 2.6.X.rt [12].
   1.2. Evaluación de los rangos de prioridad de la plataforma RTSJ.
   1.3. Evaluación de los tiempos de cambio de contexto que ofrece el planificador de prioridades fijas de la plataforma RTSJ.
   1.4. Evaluación de la precisión y de los tiempos de atención de los relojes de la plataforma RTSJ.
2. Desarrollo de un proceso para la evaluación del modelo lógico y del modelo reactivo de cualquier aplicación RT-Java que haya sido diseñada e implementada de acuerdo con el modelo de referencia OORT. Este proceso incluye:
   2.1. La especificación de las anotaciones que deben ser incluidas en el código de la aplicación para identificar los elementos relevantes de la aplicación.
   2.2. Diseño e implementación de la estrategia de instrumentación de las aplicaciones para la medida de los tiempos de ejecución. Así como la especificación y diseño de la clase TimeMntr que contiene los recursos comunes que se requieren para la estimación de los tiempos y la construcción del modelo.
   2.3. El desarrollo de la aplicación de procesamiento de texto que procesa el código fuente (Java) de cualquier aplicación con las anotaciones OORT y genera la aplicación instrumentada que una vez ejecutada construye el modelo lógico y reactivo MAST 2.
   2.4. Desarrollo del programa Java (incluido como el método createMastModel() de la clase TimeMntr) que construye el modelo MAST de la aplicación que se configura, en base a las anotaciones del código y de la información recogida durante la ejecución en el entorno de configuración.
3. Desarrollo de una aplicación de prueba denominada BURTA (Aplicación de una unidad remota y teleoperada básica) a fin de validar la metodología y el proceso propuesto.

I.3 Modelo de referencia OORT orientado al modelado

Una aplicación de tiempo real se especifica y diseña en base al conjunto de eventos que son atendidos por ella, así como por las respuestas que ejecuta concurrentemente para su atención.

En el modelo de referencia OORT:

1) Los eventos que atiende una aplicación pueden ser de dos tipos:

- Eventos temporizados periódicos.
- Eventos esporádicos generados en el entorno bien por la propia plataforma de ejecución o a través de algún dispositivo de I/O.

2) Las respuestas a un evento son siempre lineales y conducidas por una única línea de flujo de control que se inicia con la ocurrencia del evento de disparo, y finaliza cuando la respuesta concluye.

3) Entre las respuestas a dos ocurrencias de evento diferentes (ya sean de eventos de diferentes tipos o eventos del mismo tipo o fuente de eventos) no existen ninguna relación de flujo de control.

4) Entre las respuestas a dos ocurrencias diferentes de eventos puede requerirse sincronización bien porque:

- Las respuestas a dos eventos diferentes requieran utilizar un recurso protegido (recurso que dada la concurrencia inherente a las respuestas ha de ser usado utilizando un mutex que garantice el acceso al mismo con exclusión mutua).
- Las respuestas a dos ocurrencias de un mismo evento puedan requerir sincronización porque ambas respuestas utilizan los mismos thread para su planificación.

5) Las secuencias de actividades que constituyen la respuesta a un evento se planifican por un conjunto de threads con prioridades fijas creados estáticamente y dedicados a ella. El número de threads de la respuesta a un evento es función del número de requisitos intermedios que tenga especificados:

- En el caso de que la respuesta tenga como requisito temporal un único plazo, este sea al final de la respuesta, y su valor sea inferior al mínimo tiempo entre eventos que generan la respuesta, se utiliza un único thread.
- Si en los requisitos temporales de una respuesta hay especificados más de un plazo (incluyendo en esta una sección final sin plazo), o hay plazo superiores al tiempo mínimo entre eventos, la respuesta es planificada por varios threads, todos ellos (salvo el inicial) están sincronizado cada uno con el siguiente por la única línea de flujo de control que conduce la respuesta, y la sincronización se implementa a través de un mecanismo de sincronización cruzada (tipo variable de condición).

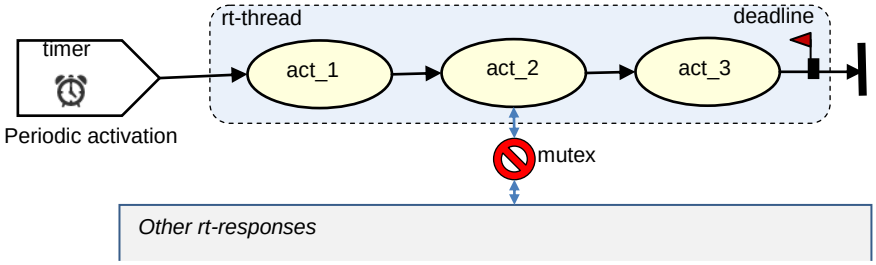
6) En una respuesta a un evento temporizado periódico, un requisito de jitter acotado se implementa con un thread con prioridad variable y un timer que introduce un offset (respecto al instante de activación) que define el instante respecto al que se mide el jitter. No se requiere el timer si el jitter acotado es relativo al instante de activación (jitter de entrada).

7) Las respuestas a eventos de tiempo real no pueden nunca bloquearse en espera a que un thread de no tiempo real libere el recurso. Por ello los recursos de comunicación entre threads de tiempo real y threads de no tiempo real se realizan a través de un nuevo tipo de recurso que ofrece la opción de acceder a él sólo si está libre, en caso contrario retorna sin bloquearse notificando que el acceso no se ha realizado. Este

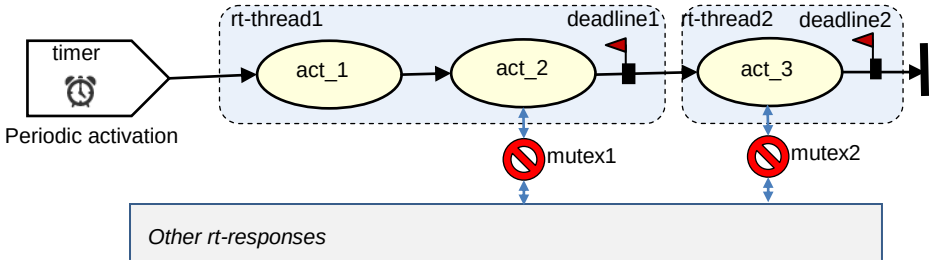
acceso no bloqueante es el utilizado por los thread de tiempo real para acceder al recurso sin necesitar esperar a que el thread de no tiempo real finalice.

De acuerdo con estos criterios de implementación de las respuestas a los eventos, se pueden identificar los siguientes casos representativos:

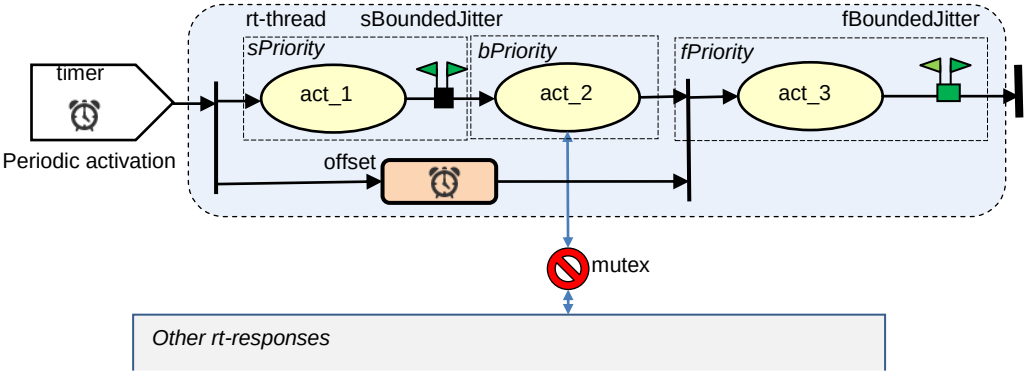
- Respuesta periódica con requisito temporal final



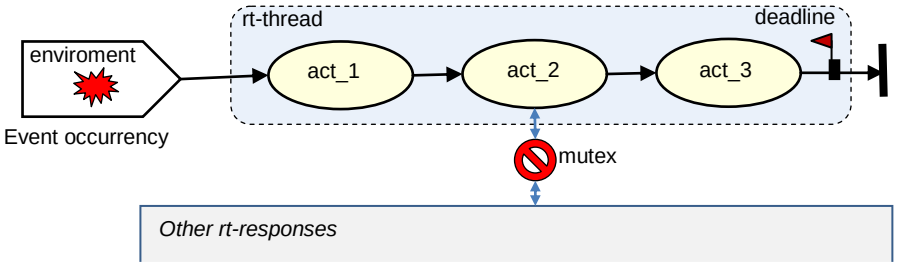
- Respuesta periódica con requisitos temporales intermedios



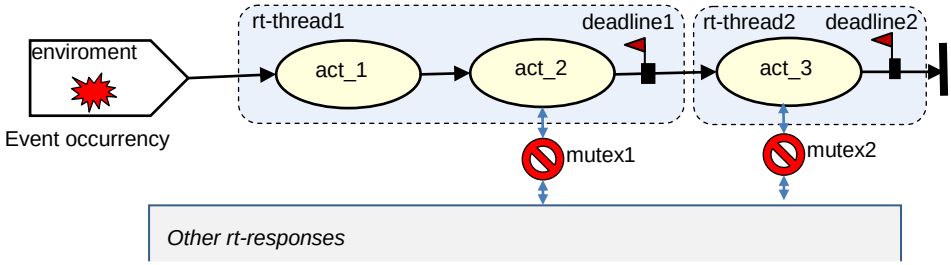
- Respuesta con jitter acotados de entrada y salida



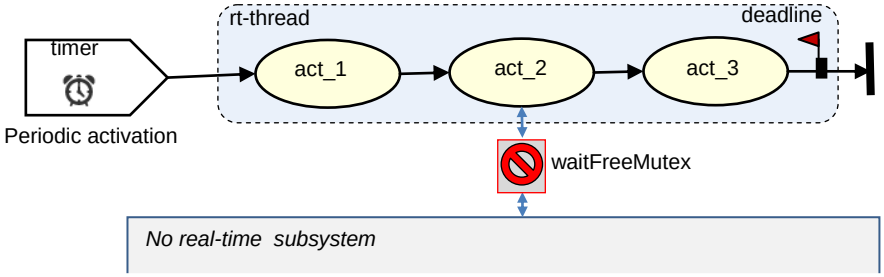
- Respuesta a un evento del entorno con requisito temporal final



- Respuesta a un evento del entorno con requisitos temporales intermedios.



- Respuesta que se comunica con un subsistema de no tiempo real.



I.4 Patrones para el diseño de aplicaciones de tiempo real

Al estar las aplicaciones implementadas utilizando RT-Java que es un lenguaje orientado a objetos (OO), no está definida la correspondencia entre el modelo lógico y el modelo reactivo. Uno de los objetivos del modelo de referencia OORT es establecer una correspondencia estricta entre ambos:

1) Cada thread que se requiere en la aplicación se implementa por una instancia de una clase activa que extiende a la clase RT-Java *NoHeapRealtimeThread*:
   - El thread que planifica la respuesta a un evento temporizado periódico se construye con las instancias de una clase de estereotipo *«periodicExecutor»* cuyo thread interno se activa con el periodo especificado e invoca la tarea que constituye la respuesta.

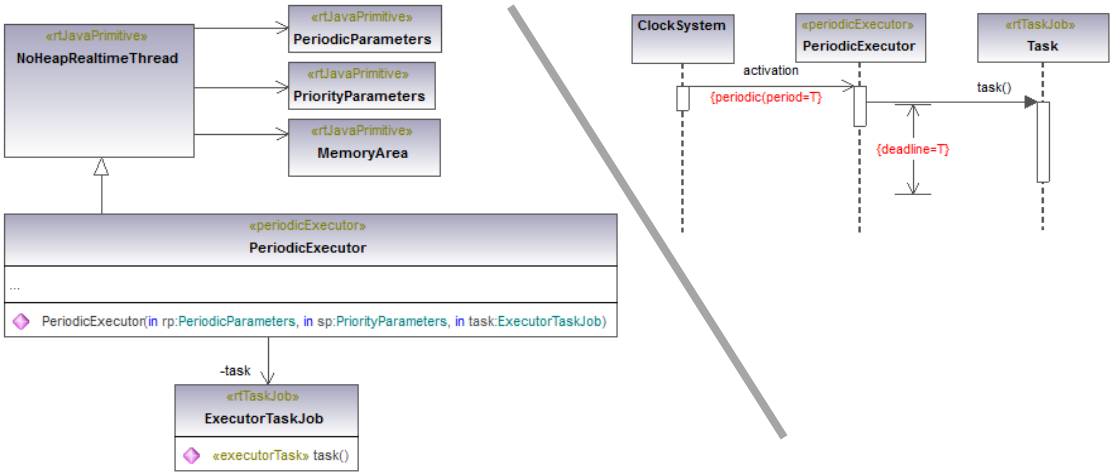


Figura I.2: Clases del patrón Periodic Executor

- El thread que planifica la respuesta a un evento esporádico del entorno se construye con la instancia de una clase de estereotipo *«eventHandler»* cuyo thread interno

itera un bucle continuo en el que se ejecuta sucesivamente la invocación de una instancia en el que se espera a la ocurrencia del evento, y posteriormente se invoca la tarea que constituye la respuesta al evento.

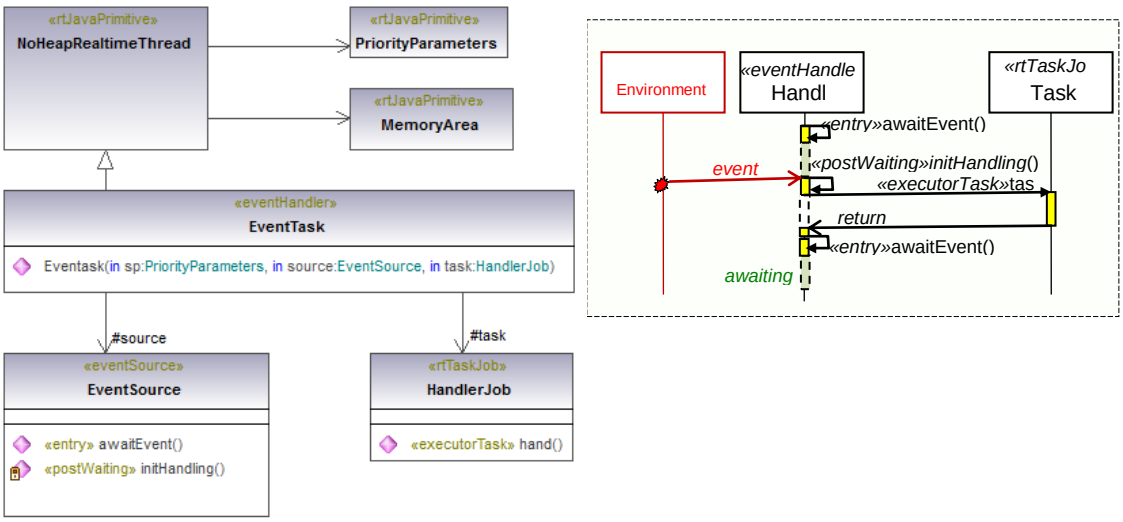


Figura I.3: Elementos del patrón EventHandler y atención a eventos del entorno

- El thread que planifica un segmento intermedio o final de una respuesta se construye también con la instancia de una clase de estereotipo *«eventHandler»* cuyo thread interno itera un bucle continuo en el que se ejecuta sucesivamente la invocación de una instancia en el que se espera a que la actividad que le precede en la respuesta le pase el flujo de control, y posteriormente se invoca la tarea que constituye la ejecución del segmento de respuesta que planifica.

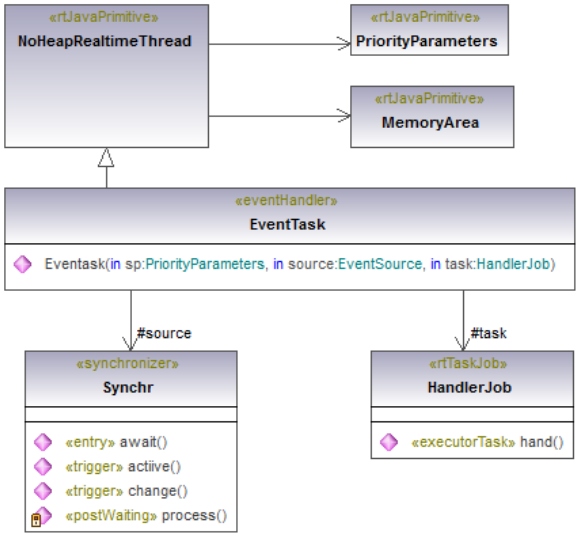


Figura I.4: Elementos del patrón EventHandler y transferencia de flujo de control

- El thread que planifica una respuesta temporizada periódica con jitter acotado se construye con la instancia de una clase de estereotipo *«boundedJitterPeriodicExecutor»* cuyo thread interno se activa con el periodo especificado e invoca secuencialmente las siguientes operaciones:

- La tarea que constituye la fase inicial con jitter acotado (si existe). Al finalizar la misma establece al thread a la prioridad base.
- La tarea que constituye la fase intermedia (si existe). Al finalizar la misma se establece el thread a la prioridad que corresponde a la fase final.
- Se suspende el thread hasta que se alcance el instante del offset que corresponde a la fase final.

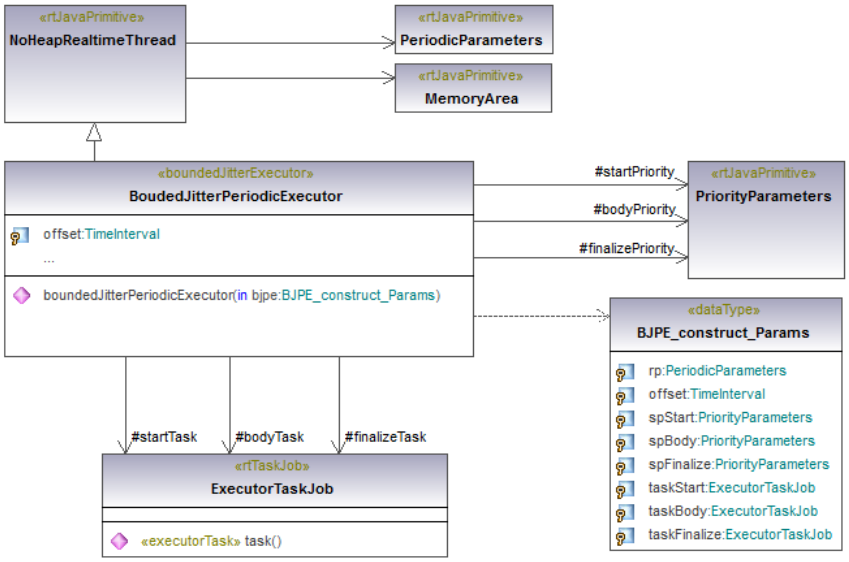


Figura I.5: Elementos del patrón Bounded Jitter task

- Cuando se activa por alcanzarse el offset, se invoca la tarea que constituye la fase final (si existe), y finalizar la misma establece el thread a la prioridad que corresponde a la fase inicial.
- El thread se suspende hasta el próximo periodo de activación.

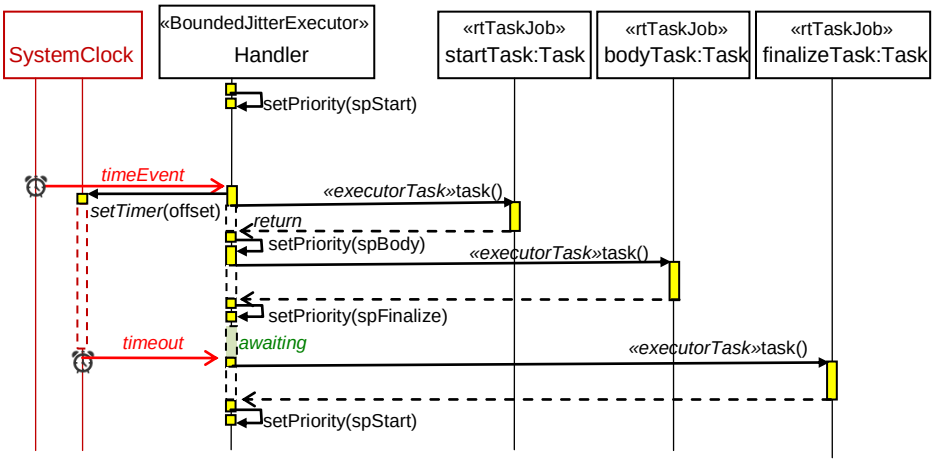


Figura I.6: Elementos del patrón Bounded Jitter task

2) La respuesta a cualquier tipo de evento (activación periódica procedente del reloj, evento procedente del entorno, activación por flujo de control al finalizar el flujo previo) siempre corresponde a la ejecución de un único método de estereotipo *«executorTask»* que está definido en una clase de estereotipo *«rtTaskJob»*. En la ejecución de este método se ejecutan todas las secuencias de actividades que correspondan a la respuesta, la cuales

pueden consistir en la invocación de otros métodos definidos en las instancias de la aplicación.

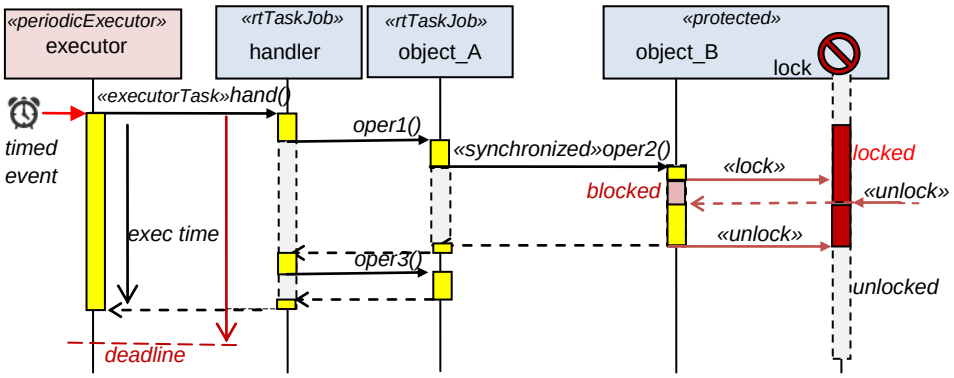


Figura I.7: Respuesta a un evento periódico temporizado

3) Un recurso compartido por varias respuestas de la aplicación y accedido con exclusión mutua siempre esta encapsulado en una instancia de la aplicación que corresponde a una clase de estereotipo *«protected»* cuyos métodos tienen el estereotipo *«synchronized»*. Previamente a la ejecución Java requiere tomar el mutex (lock) asociado a la instancia, y por tanto, se garantiza la exclusión mutua. Igualmente un recurso compartido puede construirse en base a una clase (no a una instancia como antes), invocando en este caso métodos con estereotipo *«staticSynchronized»*. En Java esto método requieren tomar el mutex asociado a la clase antes de ejecutarse, y por tanto garantiza la exclusión mutua entre ellos (el mutex de cada instancia y el mutex de clase son independientes y representan recursos diferentes).

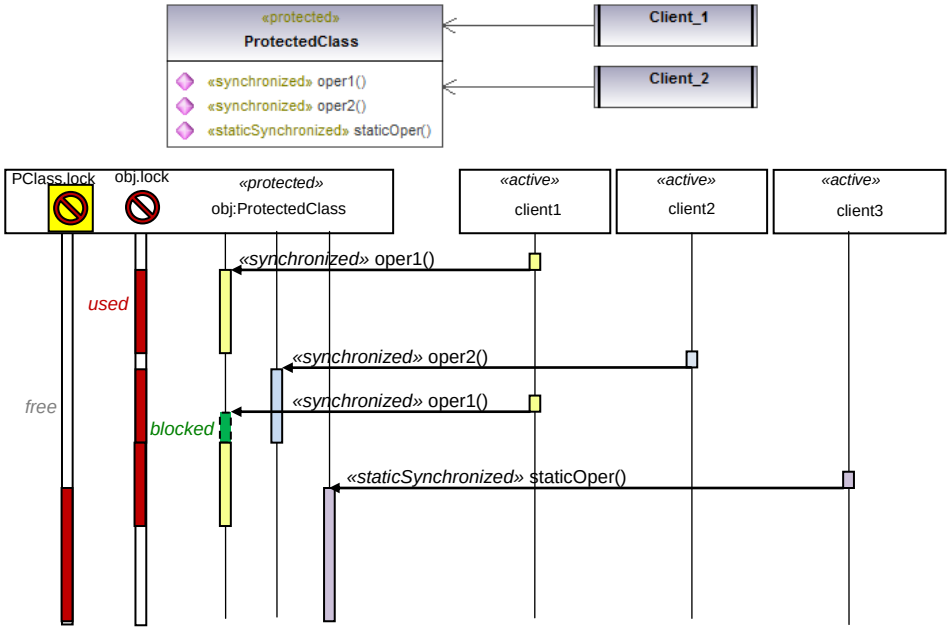


Figura I.8: *Elementos del patrón Protected*

La gestión de un evento del entorno (de la plataforma o de los dispositivos externos) se realiza a través de una instancia de una clase protegida *«eventSource»* cuyo estado interno cambia cuando el evento se recibe. El mecanismo por el que el thread que planifica su

respuesta se activa con su ocurrencia, es en base a invocar un método de estereotipo *«entry»* que es ofrecido por la clase. Este método suspende en un mecanismo de sincronización interno a la clase (tipo variable de condición) el thread en espera al cambio de estado que produce el evento. Cuando el evento ocurre, el thread ejecuta un método opcional de estereotipo *«postWaiting»*, la invocación del método *«entry»* finaliza y el thread ejecuta la tarea de respuesta al evento. Las instancias de las clases *«eventSource»* son protegidas y los métodos *«entry»* y *«postWaiting»* son extensiones de *«synchronized»*, por lo que se ejecutan con exclusión mutua respecto a cualquier thread interno que tiene definida la clase.

4) La gestión de sincronización cruzada entre dos thread que planifican una misma respuesta y que se transfieren el flujo de control se realiza a través de una instancia de la clase protegida de estereotipo *«synchronizer»*. El thread que espera el flujo de control invoca el método de estereotipo *«entry»* que es ofrecido por la clase. Este método suspende en un mecanismo de sincronización interno a la clase (tipo variable de condición) al thread en espera al cambio de estado que produce el thread que planifica el segmento previo de la respuesta, y que transfiere el control invocando el método *«trigger»* que también ofrece *«synchronizer»*. Cuando la activación ocurre, el thread ejecuta un método opcional de estereotipo *«postWaiting»*, y la invocación del método *«entry»* finaliza y el thread ejecuta la tarea que corresponde al segmento de la respuesta que planifica. Las instancias de las clases *«synchronizer»* son protegidas y los métodos *«entry»*, *«postWaiting»* y *«trigger»* son extensiones de *«synchronized»*, por lo que se ejecutan con exclusión mutua respecto a cualquier thread que acceda a la clase.

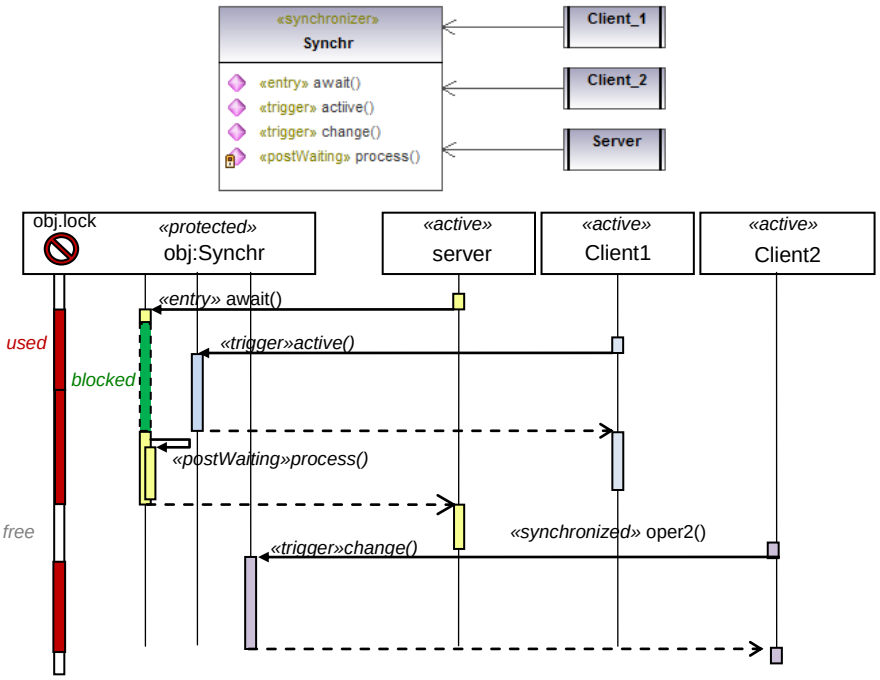


Figura I.9: Elementos del patrón Synchronizer

5) La interacción entre thread de tiempo real y thread de no tiempo real se realiza a través de instancias de clases protegidas de estereotipo *«waitFreeProtected»*. Son una extensión de

las clases *«protected»* que además de ofrecer métodos de estereotipo *«synchronized»* ofrecen métodos de estereotipo *«waitFreeSynchronized»* que cuando son invocados, si el mutex de la instancia está libre, es tomado y el método se ejecuta e informa que se ha ejecutado. Por el contrario, si el mutex de las instancias está tomado por otro thread, retorna informando que el método no se ha ejecutado.

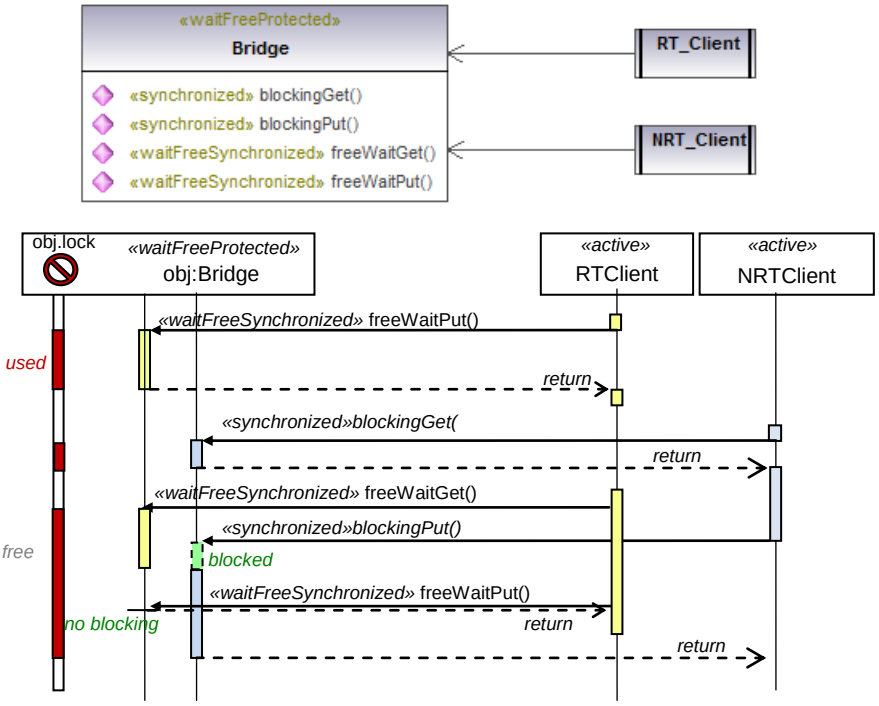


Figura I.10: Elementos del patrón WaitFreeProtected

Estereotipos definidos en el modelo de referencia OORT

Las clases que constituyen una aplicación de tiempo real se construyen con clases RT-Java que implementan los patrones que se han descrito. Se han dejado libres los nombres de las clases y de los métodos para que se correspondan con la semántica de su funcionalidad. Se han definido un conjunto de estereotipos de clases y de métodos para describir el papel que juegan desde el punto de vista del modelo de referencia de tiempo real. Los estereotipos definidos han sido:

En la Figura I.11 se muestran los estereotipos definidos para las clases:

Estereotipo **rtActive:** Es un estereotipo abstracto que se asocia a las clases activas que hereda de *NoHeapRealtimeThread* un thread de tiempo real, y que tiene que estar instanciada en una zona de memoria externa al heap (*Immortal Memory* o *Scoped Memory*).

Estereotipo **periodicExecutor:** Es un estereotipo concreto que se asocia a una clase rtActive que tiene asociado un mecanismo de activación temporizado periódico (*PeriodicParameters*) y un parámetro de planificación (*PriorityParameters*) que utiliza durante toda su actividad.

Estereotipo **boundedJitterPeriodicExecutor:** Es un estereotipo concreto que se asocia a una clase «periodicExecutor» que tiene asociado tres parámetros de planificación diferenciadas (*PriorityParameters*) y un intervalo de tiempo definido como offset. En cada activación ejecuta las tareas taskStart, taskBody y taskFinalize con una prioridad diferentes. La tarea taskFinalize se ejecuta siempre que haya ocurrido un tiempo igual al offset (deadline) desde su activación.

Estereotipo **eventHandler:** Es un estereotipo concreto que se asocia a una clase rtActive que tiene asociado un parámetro de planificación (*PriorityParameters*) que utiliza durante toda su actividad. Su actividad es un bucle indefinido de ejecución de la secuencia invocación a elemento con estereotipo *«eventSource»* a la espera del evento de activación seguida de la invocación de la tarea de ejecución de la tarea de respuesta al evento asociada.

Estereotipo **rtTask:** Es un estereotipo abstracto que se asocia a las clases pasivas de tiempo real que van a ser utilizadas como tareas de un «rtActive». Tienen que estar instanciadas en una zona de memoria que no pertenezca al Heap (*Immortal Memory* o *Scoped Memory*), ya que sus métodos van a ser invocados por un thread de tiempo real.

Estereotipo **rtTaskJob:** Es un estereotipo concreto que se asocia a las clases pasivas «rtTask» que ofrecen métodos (con estereotipos «executorTask») que van a ser asignados como tareas de un «rtActive». Tienen que estar instanciadas en una zona de memoria que no pertenezca al Heap (*Immortal Memory* o *Scoped Memory*), ya que sus métodos van a ser invocados por thread de tiempo real.

Estereotipo **protected:** Es un estereotipo concreto que se asocia a las clases pasivas del tipo *rtTask* cuyos métodos van a ser invocados por threads concurrentes de la aplicación, y para que el acceso sea seguro se requiere que sean invocados con exclusión mutua. Por cada instancia de una clase protegida se define un mutex que debe ser invocado antes de su ejecución. Si el método es estático, el mutex se define a nivel de la clase y se garantiza que entre todos los métodos estáticos exista exclusión mutua.

Estereotipo **eventSource:** Es un estereotipo concreto que se asocia a clases protegidas *protected* y a las que se añade un mecanismo de sincronización con capacidad de suspender un thread que invoca ciertos métodos específicos en base al estado interno del objeto en ese instante. El estado interno del objeto puede cambiar en base a eventos en el sistema o en el entorno, con lo que se puede activar el thread. El hecho de que herede de objeto protegido garantiza que todos los thread que invocan métodos del objeto se ejecutan con exclusión mutua.

Estereotipo **synchronizer:** Es un estereotipo concreto que se asocia a clases protegidas *eventSource* y a las que se añade la capacidad de que el cambio de estado que activa el thread suspendido sea otro thread de la aplicación que invocan método del objeto.

Estereotipo **waitFreeProtected:** Es un estereotipo concreto que se asocia a las clases protegidas *protected* y a las que el mutex asociado es un tipo extendido que permite que un thread que lo invoca cuando el mutex esté tomado no suspenda al thread que invoca sino que termina retornando una información de acceso no realizado.

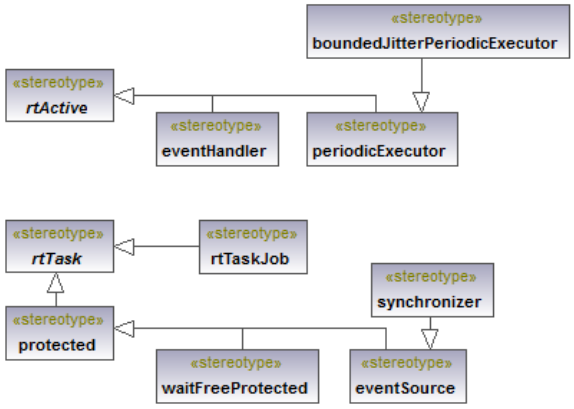


Figura I.11: Estereotipos de clases definidos para los modelos de tiempo real

Otros estereotipos de clase utilizados en la memoria pero no directamente relacionados con los patrones de tiempo real son:

Estereotipo **active:** Es un estereotipo concreto que se asocia a las clases que derivan de clase Java Thread, y por tanto tienen asociado un thread. Como no es de tiempo real, habitualmente se instancian en la memoria Heap.

Estereotipo **main:** Es un estereotipo concreto que se asocia a las clases que contienen el método main() con el que se lanza la ejecución de la aplicación. El thread que ejecuta main() es procedente del entorno y no es de tiempo real.

Estereotipo **rtJavaPrimitive:** Es el estereotipo que se asocia a las clases definida por la especificación RT-Java.

Estereotipo **neutral**: Es el estereotipo que se asocia a clases cuyos métodos son operaciones basadas con información de stack, y por tanto pueden ser ejecutados concurrentemente y de forma segura sin que se requiera sincronización entre los thread que los invocan. Cuando una clase no tiene definido estereotipo, es neutral.

En la Figura I.12 se muestran los estereotipos definidos para las operaciones:

Estereotipo **executorTask:** Es el estereotipo que se asocia a la operación de una clase *«rtTaskJob»* que define la actividad que inicia y finaliza una respuesta de tiempo real.

Estereotipo **synchronized:** Es el estereotipo que se asocia a la operación de una clase *«protected»* para establecer que se ha de ejecutar con exclusión mutua a cualquier operación de la misma instancia que se invoque concurrentemente con su invocación. Al iniciarse trata de tomar el mutex (instance.lock) de la instancia. Si lo consigue lo toma y se ejecuta. Si el mutex está ocupado se suspende en el mutex hasta que este queda libre.

Estereotipo **entry:** Es el estereotipo que se asocia a la operación de una clase *«eventSource»* que tiene capacidad de suspender el thread que invoca en base al estado de la instancia. En una clase *«eventSource»* sólo existe un método con este estereotipo.

Estereotipo **postWaiting:** Es el estereotipo que se asocia a la operación interna de una clase *«eventSource»* que se ejecuta por el thread que ha invocado una operación *«entry»* tras ser activada. Aunque es privada (de hecho a nivel de código es la parte de la operación *«entry»* que se ejecuta tras la activación) es una operación *«synchronized»* que tiene que esperar a que el thread que provocado la activación libere el mutex. Este tipo de operación es opcional en una clase *«eventSource».*

Estereotipo **trigger:** Es el estereotipo que se asocia a la operación de una clase *«synchronizer»* y cuya invocación puede producir un cambio de estado de la instancia que active el thread suspendido en la *«entry»*. Una clase *«synchronizer*» puede tener varias operaciones con este estereotipo, y todas ellas hacen referencia a la única operación *«entry».*

Estereotipo **waitFreeSynchronized:** Es el estereotipo que se asocia a la operación de una clase *«waitFreeProtected»* que no lleva a cabo una invocación no bloqueante, esto es, si encuentra el mutex tomado, no se suspende sino que retorna con la información de acceso no realizado.

Estereotipo **staticSynchronized**: Es el estereotipo que se asocia a la operación estática de una clase «protected» para establecer que se ha de ejecutar con exclusión mutua a cualquier otra operación estática de la misma clase que se invoque concurrentemente con su invocación. Al iniciarse trata de tomar el mutex de la clase (class.lock) de la instancia. Si lo consigue lo toma y se ejecuta. Si el mutex está ocupado por otro thread, se suspende hasta que quede libre.

Estereotipo **staticEntry:** Es el estereotipo que se asocia a la operación estática de una clase *«eventSource»* que tiene capacidad de suspender el thread que invoca en base al estado de las variables estáticas de la clase. En una clase *«eventSource»* sólo existe un método con este estereotipo.

Estereotipo **staticPostWaiting:** Es el estereotipo que se asocia a la operación interna estática de una clase *«eventSource»* que se ejecuta por el thread que ha invocado una operación *«entry»* tras ser activada. Aunque es privada (de hecho a nivel de código es la parte de la operación *«staticEntry»* que se ejecuta tras la activación) es una operación *«staticSynchronized»* que tiene que esperar a que el thread que ha provocado la activación libere el mutex de la clase. Este tipo de operación es opcional en una clase *«eventSource».*

Estereotipo **staticTrigger:** Es el estereotipo que se asocia a una operación estática de una clase *«synchronizer»* y cuya invocación puede producir un cambio de estado de la instancia que active el thread suspendido en la *«staticEntry»*. Una clase *«synchronizer*»

puede tener varias operaciones con este estereotipo, y todas ellas hacen referencia a la única operación *«staticEntry».*

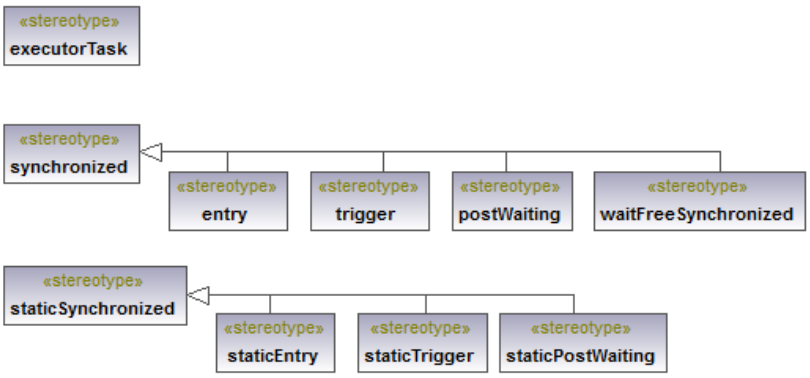


Figura I.12: Estereotipos de operaciones definidos para los patrones de tiempo real

I.5 Estructura de los modelos de planificabilidad

El análisis y la configuración de planificabilidad de un sistema de tiempo real requieren construir un modelo de comportamiento temporal. En este trabajo utilizamos la metodología MAST 2 para formular los modelos de tiempo real. Los modelos son sencillos ya que las aplicaciones se ejecutan sobre un único procesador y se han diseñado utilizando el modelo de referencia OORT.

El modelo MAST de un sistema de tiempo real se compone de 3 secciones complementarias: El modelo de la plataforma, el modelo lógico y el modelo reactivo. En la Figura I.13 se muestran las dependencias entre estas secciones.

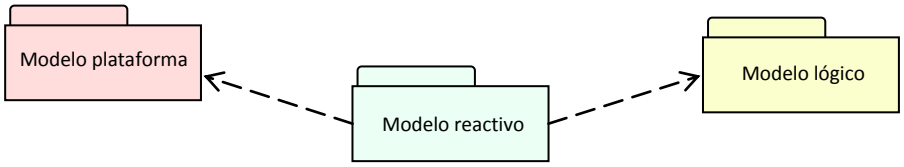


Figura I.13: Dependencias entres las secciones del modelo de tiempo real

- Modelo de la Plataforma: Describe la capacidad del procesador, y la parte de ella que es consumida por los recursos que existen en la plataforma con independencia de la aplicación que se ejecute. En el caso de una plataforma RTSJ sobre Linux de tiempo real los recursos son el planificador y los dos relojes que se definen en RTSJ: el reloj de no tiempo real (*nrtTimer*) que aunque no se utiliza por su baja resolución en las aplicaciones de tiempo real continua instalado y requiere periódicamente la atención de la CPU, y el reloj de tiempo real (rtTimer) que se utiliza en las aplicaciones de tiempo real.

  En la Figura I.14 se muestran los elementos que constituyen el modelo de la plataforma para estos sistemas.

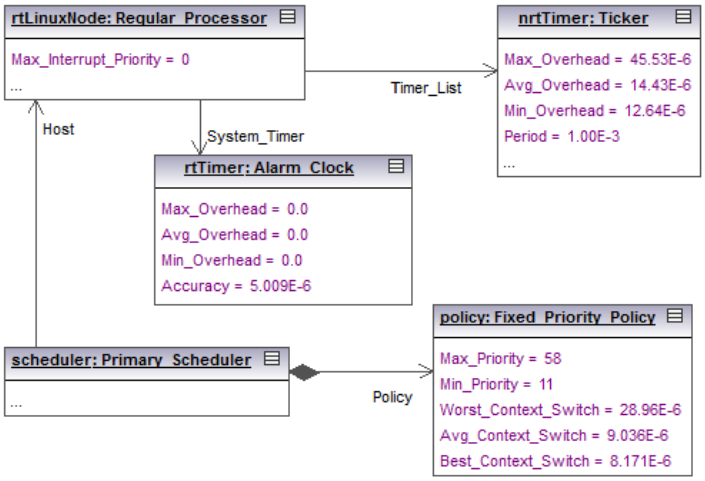


Figura I.14: Elementos del modelo de la plataforma de RTSJ sobre Ubuntu Linux 2.6.X.rt

rtLinuxNode:Regular_Processor: describe el comportamiento de la CPU y la gestión de la interrupciones. Speed_Factor toma siempre su valor por defecto 1.0 (por ello no se muestra), ya que los tiempos de ejecución del código se miden sobre el procesador de la plataforma de ejecución. La plataforma Linux de tiempo real sólo permite un nivel de interrupción hardware y arbitrariamente a las prioridades de interrupción, se le asigna el valor Max_Interrupt_Priority = Min_Interrupt_Priority=0. Los tiempos de cambio a modo interrupción, no se han evaluado en este trabajo y se consideran despreciables (Worst_ISR_Switch= Avg_ISR_Switch = Best_ISR_Switch=0.0, y como son el valor por defecto no se representan).

scheduler:Primary_Scheduler: Describe la política de planificación que este caso es de prioridades fijas. La política describe el rango de prioridades de la plataforma RTSJ y los tiempos de sobrecarga que introducen los cambios de contexto. Los tiempos de cambio de contexto dependen del procesador que se utiliza.

nrtTimer:Ticker: Representa el temporizador estándar de la JVM. Es de tipo ticker con un periodo de activación y una resolución de 1 ms. Los tiempos de cambio de sobrecarga dependen del procesador que se utiliza.

rtTimer:AlarmClock: Representa el temporizador de tiempo real introducido con el RTSJ, su resolución es del orden de los microsegundos, y su sobrecarga de gestión solo se realiza cuando se atiende un evento programado.

En el modelo no se han analizado los modelos de los driver de conexión con los dispositivos del entorno. Su modelo será parte de las clases *«EventSource»* que gestionan los fenómenos de interrupción hardware que producen estos dispositivos.

- Modelo lógico: Describe la capacidad de procesamiento que requiere la ejecución de las secciones de código que deben ser planificada en los thread. En el caso de aplicaciones diseñadas con el modelo de referencia OORT, el modelo lógico se compone de cinco tipos de elementos lógicos:

- Elementos de modelado *Enclossing_Operation* que describen globalmente el comportamiento temporal del código con el que se implementa las respuestas de la aplicación. Sus atributos *Worst_Case_Execution_Time*, *Avg_Case_Execution_Time* y *Best_Case_Execution_Time* describen estadísticamente el tiempo que dedica la CPU a ejecutar el código de la respuesta sin considerar tiempos de bloqueos en recursos compartidos, tiempos en los que la respuesta ha sido expulsada por threads de mayor prioridad, o tiempos de espera a eventos (del entorno o del reloj). La lista *Operation_List*, describe todos los métodos protegidos *«synchronized»* que son invocadas desde la respuesta. Estas operaciones deben ser especificadas porque representan posibles tiempos de bloqueo que pueden ser introducidos por otras respuestas de menor prioridad.

- Elementos de modelado *Priority_Inheritance_Mutex* que representan los mecanismos de exclusión mutua de las instancias de clases *«protected*» que son utilizados por sus métodos *«synchronized».*

- Elementos de modelado *Priority_Inheritance_Mutex* que representan los mutexes de las clases «*protected*» y que son utilizados por los métodos *«staticSynchronized».*

- Elementos de modelado *Simple_Operation* que describen los tiempos de ejecución de los métodos *«synchronized», «trigger», «entry»* y *«postWaiting*» de cada instancias de una clase *«protected*» que sea invocados como parte de una respuesta de la aplicación. Sus atributos *Worst_Case_Execution_Time*, *Avg_Case_Execution_Time* y *Best_Case_ Execution_Time* representa el tiempo que dedica la CPU a la ejecución del código que incluye el método. Así mismo referencian en su atributo *Mutex_List* el mutex introducido con la instancia sobre la que se ha invocado el método.

- Elementos de modelado *Simple_Operation* que describen los tiempos de ejecución de los métodos *«staticSynchronized», «staticTrigger», «staticEntry»* y *«staticPostWaiting*» de las clases *«protected*» que son invocados desde la respuesta de la aplicación. Sus atributos *Worst_Case_Execution_Time*, *Avg_Case_Execution_Time* y *Best_Case_ Execution_Time* describen los tiempos que dedica la CPU a la ejecución de los métodos estáticos. Así mismo referencian en su atributo *Mutex_List* el mutex de la clase de la que es método estático.

En la Figura I.17 se muestra la sección de modelo lógico que es introducido como consecuencia de la respuesta cuya secuencia de invocaciones se muestra en el diagrama de secuencias de la Figura I.16, y que corresponde al diseño lógico que incluye las clases del diagrama que se muestra en la Figura I.15. Este diseño es una sección simplificada del ejemplo BURTA que se describe en el capítulo IV.

Obsérvese que los métodos *«waitFreeSynchronized*» no son referenciados en las Enclosing_Operations ya que no introducen bloqueos, sólo mayor variabilidad en los tiempos de respuesta que los invocan.

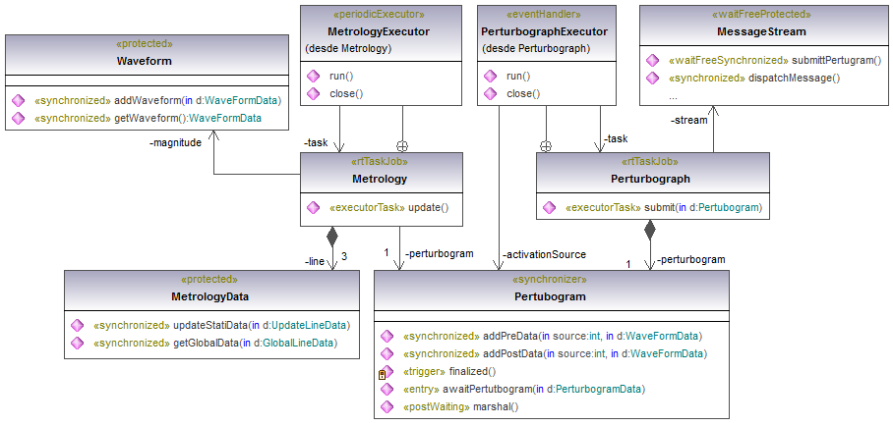


Figura I.15: Clases del diseño lógico de la aplicación

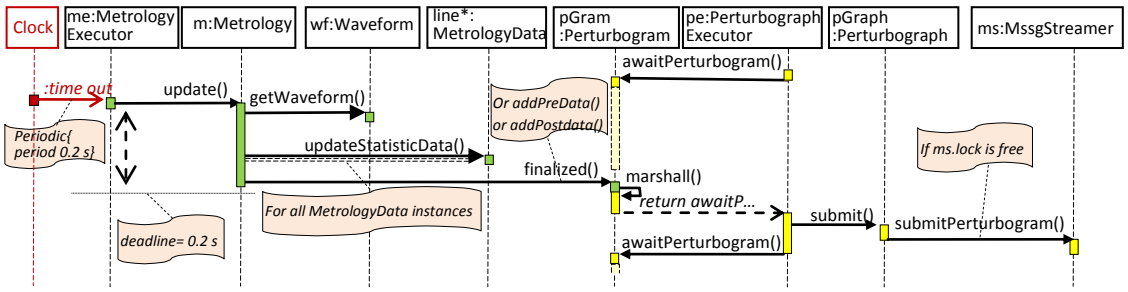


Figura I.16: Secuencia de invocaciones que constituyen la respuesta ejemplo

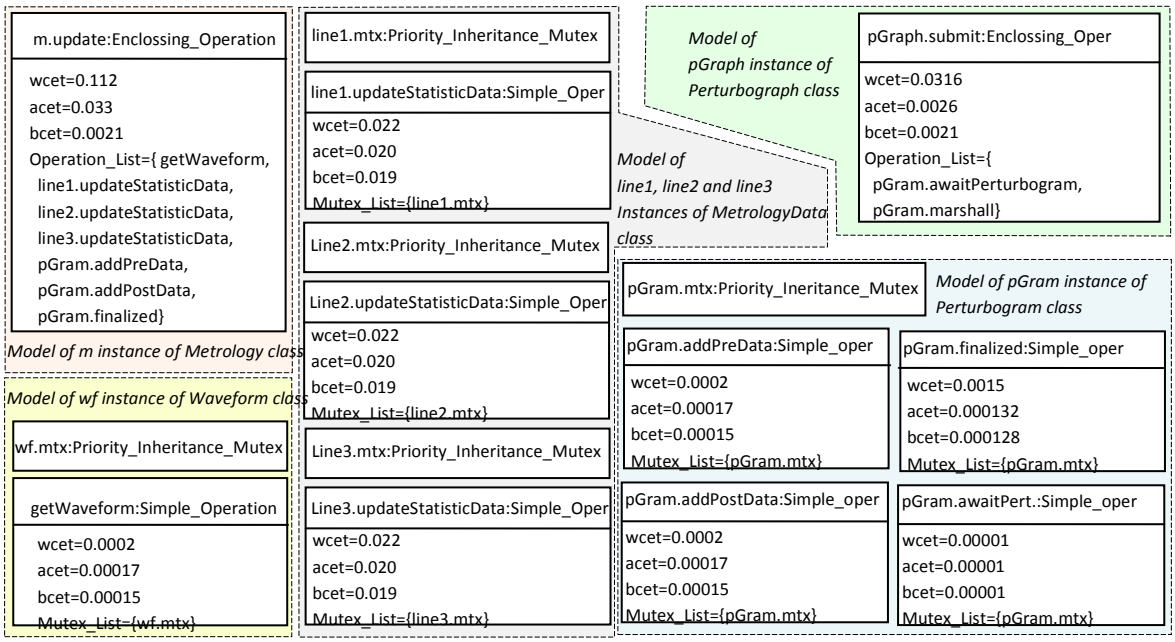


Figura I.17: Modelo lógico referenciado por la respuesta ejemplo

Modelo Reactivo: Describe el modelo de carga de la aplicación, esto es, los eventos a los que responde, los thread que a cada uno de ellos asigna para planificar la respuesta y la secuencia de actividades que ejecuta como respuesta a cada uno de ellos.

El modelo reactivo va estructurado por respuestas. Para cada respuesta el modelo contiene:

- Los elementos de modelado Thread que utiliza para planificar la respuesta. Este elemento contiene los parámetros Fixed_Priority_Params que describe la prioridad con la que debe ser planificado.
- Un elemento tipo End_To_End_Flow que describe:
    - El patrón de generación de los eventos que generan la respuesta. En el caso de un evento temporizado se modela mediante un elemento Periodic_Event en el que se especifica el periodo de generación y la referencia al timer que lo genera. Y en el caso de un evento del entorno se especifica por un elemento Sporadic_Event en el que se especifica el mínimo tiempo entre eventos.
    - La secuencia de actividades planificada en la respuesta modelada como secuencia de elementos de modelado Internal_Event y Step. Los elementos Internal_Event pueden llevar asociados los requerimientos temporales modelados con elementos Hard_Global_Deadline y cada Step hace referencia al elemento Thread en el que se planifica y el elemento Enclosing_Operation que describe la temporización de la respuesta.

En la Figura I.18 se muestra la sección del modelo reactivo que modela la respuesta descrita en el diagrama de secuencias de la Figura I.16.

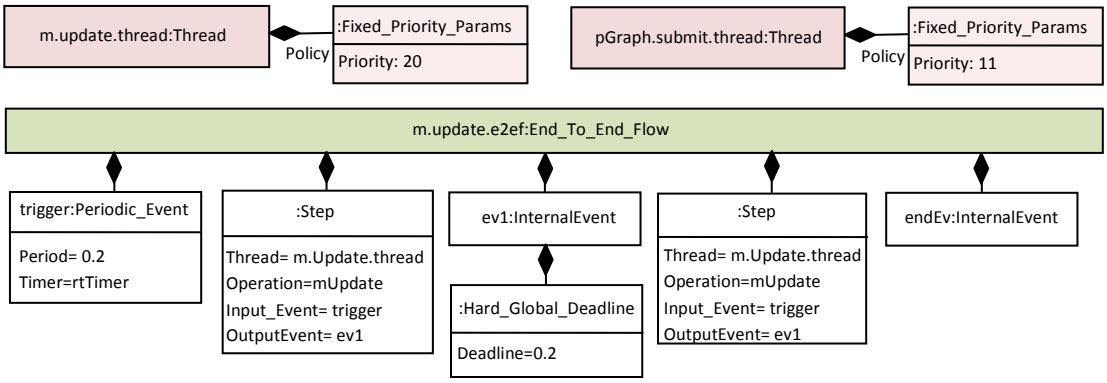


Figura I.18: Modelo reactivo de la respuesta descrita en la Figura I.16

II Herramienta para la estimación automática del modelo de la plataforma

II.1 Estructura y elementos del modelo de tiempo real de la plataforma

En la Figura I.14 se ha mostrado la estructura del modelo de la plataforma RTSJ sobre Ubuntu Linux 2.6.X.rt. En este capítulo se trata la evaluación experimental de los valores de sus parámetros. Los parámetros evaluados son:

- La evaluación de los rangos de prioridades que admiten los thread de tiempo real en RTSJ.
- Los tiempos de CPU que requiere el planificador para los cambios de contexto, esto es para pasar de planificar un thread de tiempo real a planificar otro.
- Los tiempos de CPU que requiere la atención del temporizador Java de no tiempo real que es de tipo ticker y con periodo 1ms.
- La resolución de reloj de tiempo real de RTSJ con el que se generan los eventos temporizados.

II.2 Parámetros del modelo de la plataforma y estrategia de medida

La medida de los atributos del modelo de tiempo real del procesador son procesos complejos y de naturaleza estadística, que deben ser realizados bajo condiciones de carga programada del procesador. Se ha realizado el diseño de una aplicación RT-Java que ejecutada en la plataforma que se evalúa, proporciona los parámetros de su modelo de tiempo real.

El proceso de medida se basa en todos los casos en programar diferentes conjuntos de threads de tiempo real que leen continuamente el reloj del sistema, a través de bucles con el mínimo tiempo de ciclo y los almacena en un conjunto de arrays de memoria durante el tiempo de ejecución. Finalizado el experimento que puede durar entre milisegundos y segundos, se finaliza la ejecución de los threads y por tanto las lecturas del reloj, y se realiza el análisis estadístico de los tiempos almacenados obteniendo información sobre la temporización de los fenómenos producidos. Finalizado el ciclo de medida y análisis, los resultados se pueden almacenar, y de nuevo repetir el ciclo durante tiempos tan prolongados como se necesite a fin de obtener los niveles de confianza requeridos.

El módulo de toma de medidas deberá tener estrategias para corregir y procesar las medidas de tiempo de forma fiable:

- Debe disponer de estrategias de calibrado que eliminen los términos constantes y los sesgos estadísticos que introduzca el procedimiento.
- Debe filtrar las interferencias inherentes a interrupciones hardware que siempre existen en el sistema y por encima de los threads de la máxima prioridad.
- La toma de medidas debe almacenarse en memoria, y nunca en ficheros persistentes o en los terminales que tienen unas latencias altas y no acotadas.
- Debe disponer de un módulo estadístico que calcule los valores extremos y las métricas estadísticas necesarias.

- Y por último debe poder almacenar los resultados en ficheros persistentes, o en la consola para mostrar al operador la evolución del proceso y facilitar su control.

Los parámetros que se estiman en este proceso son:

- **Rango de la prioridades:**

Determina los rangos de prioridades de tiempo real que pueden ser asignados a los thread de tiempo real. Dado que la clase *PriorityScheduler* de RT-Java ofrece los métodos estáticos para determina la mínima prioridad (*getMinPriority()*) y la máxima prioridad (*getMaxPriority()*) asignables a los thread de tiempo real.

- **Tiempos de cambios de contextos entre threads.**

Los tiempos de cambio de contexto son los tiempos de CPU que requiere el planificador para sustituir la planificación de un thread por otro más prioritario.

Existen cuatro causas que pueden conducir a un cambio de contexto y deben ser evaluados de forma diferentes. Sin embargo, MAST sólo admite unos únicos parámetros para representar todos los tipos de cambio de contexto, por ellos se deben considerar de forma pesimista que siempre será el peor de ellos.

Los cuatro tipos de cambio de contexto son:

- Cambio de contexto al activarse por temporización la suspensión de un thread de mayor prioridad que el que se está ejecutando.

  Forma de medida: Se utilizan dos thread uno de alta prioridad *hThread* que tras un cierto tiempo de ejecución T_{exec} se suspende durante un cierto tiempo T_{susp}. Un segundo thread de ejecución continua y prioridad inferior (*lThread*) que ejecuta en los tiempos que no está ejecutándose *hThread*. Los tiempos de cambio de contexto se miden en los instantes de activación del thread de alta prioridad.

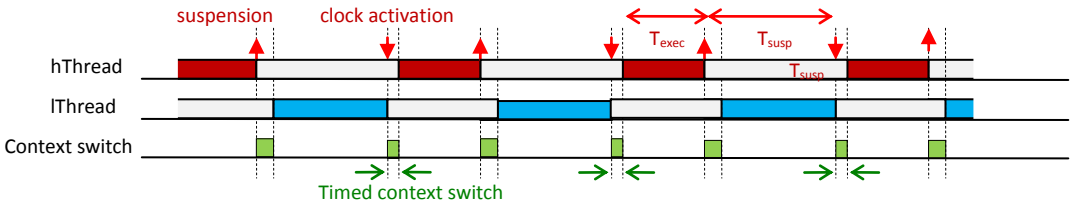


Figura II.1: Medida del cambio de contexto por suspensión temporal

- Cambio de contexto por suspensión en el lock (variable de condición) de una instancia y por activación cruzada (notify) de otro thread.

  Forma de medida: Se utilizan dos thread uno de alta prioridad *hThread* que tras un cierto tiempo de ejecución T_{exec} se suspende invocando (*wait()*) el lock de un objeto protegido. Un segundo thread de prioridad inferior (*lThread*) se ejecuta continuamente y cada cierto intervalo T_{susp}, invoca (*notify()*) el lock del objeto protegido y activa a *hThread*.

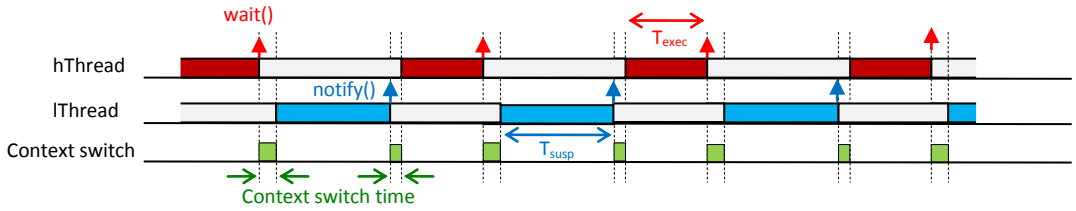


Figura II.2: Medida del cambio de contexto por suspensión en un objeto protegido

- Cambio de contexto tras el bloqueo de acceso a un mutex que está ocupado por un thread de menor prioridad.

  Forma de medida: Se utilizan dos thread *hThread* y lThread de prioridad variable. Cada thread realiza su actividad (leer consecutivamente el reloj a base de execReadingNum/ciclo) en base a tres secciones de igual actividad. La primera sección la ejecutan sin tener tomado el mutex. Las dos siguientes secciones las ejecutan dentro de la operación protegida synchrOper(). Al finalizar la segunda sección, dentro de ella, se eleva la prioridad del otro thread (a *highPrty*) y a continuación se baja la prioridad propia (a *lowPrty*). Como se muestra en el diagrama de tiempos de la figura 2.3, al finalizar de ejecutarse synchrOper el otro thread está esperando para acceder al mutex y con prioridad superior, por lo que se produce el cambio de contesto que se mide.

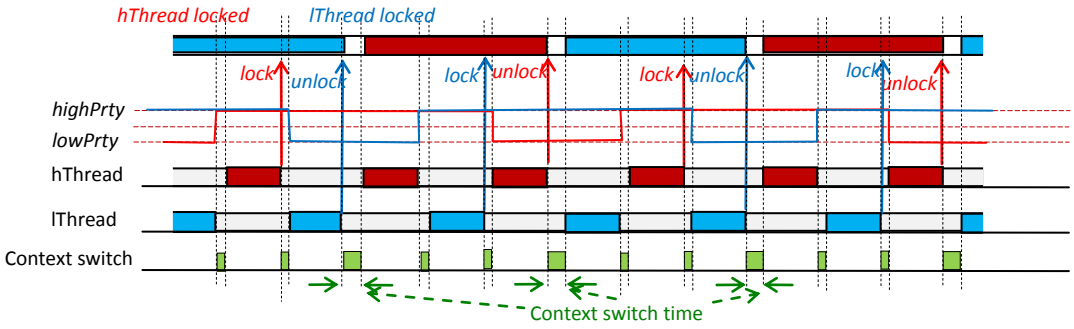


Figura II.3: Medida del cambio de contexto por suspensión en un mutex

- Cambio de contexto por asignación explícita de prioridad a un thread por encima de la prioridad del thread que se está ejecutando.

  Forma de medida: Se utilizan dos thread uno de alta prioridad *hThread* que tras un cierto tiempo de ejecución T_{exec} se reduce su prioridad por debajo de la prioridad de lThread. El segundo thread de prioridad intermedia (*lThread*) se ejecuta continuamente y cada cierto tiempo T_{susp} eleva la prioridad hThread a su valor inicial.

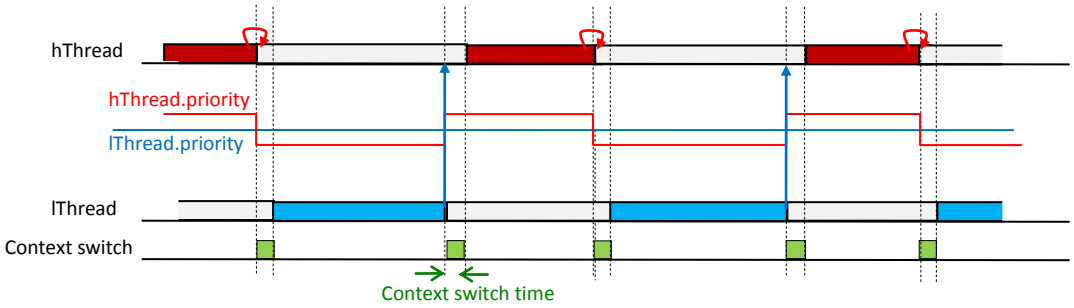


Figura II.4: Medida del cambio de contexto por cambio cruzado de la prioridad

- **Tiempo de CPU utilizado por el temporizador Java de no tiempo real y 1 ms de periodo.**

  Forma de medida: Se utilizan un thread de la máxima prioridad que lee continuamente el reloj. Posteriormente se procesa los intervalos entre lecturas con el siguiente algoritmo (Figura II.5):

  1) Se evalúa la desviación típica de los intervalos entre sucesivas lecturas del reloj, y se considera que la resolución de la medida de tiempos es de 4 veces esta desviación típica. Los intervalos entre lecturas de valor inferior a la resolución se consideran que son introducidos por el código del bucle de lectura repetida del reloj y por tanto no son considerados como tiempos dedicados por la CPU a algo diferente de la lectura del reloj.

  2) Se identifica entre los intervalos del registro del primer milisegundo, aquel que considerado como inicio contiene tras de sí una secuencia con mayor número de intervalos separados entre ellos por un intervalo de 1 ms

  3) Se establece como valores de la sobrecarga del ticker de 1 ms, los valores máximo, mínimo y promedio de los intervalos del análisis estadístico de la serie con 1 ms de periodo determinada en 2.

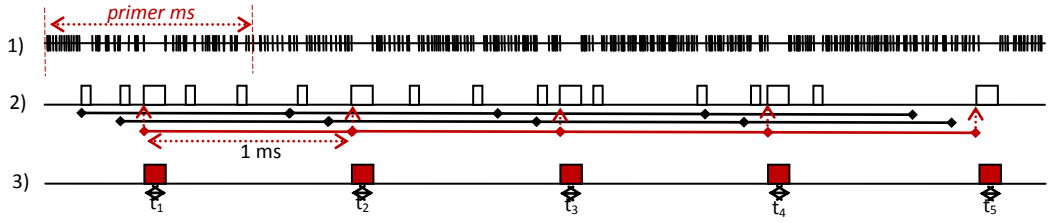


Figura II.5: Selección de los tiempos de atención del ticker de 1ms

- **Resolución del temporizador de tiempo real**

Se trata de evaluar el mínimo incremento en el tiempo de suspensión de un thread al que es capaz de responder el temporizador de tiempo real.

Forma de medida: Se utilizan un thread de la máxima prioridad que sucesivamente va suspendiendo en tiempos comprendidos entre 10 ms y 1 µs. con una distribución logarítmica en base a 10 valores por década. Así mismo se lee el tiempo antes de iniciar la suspensión e inmediatamente después de la activación. Típicamente si se representan

el tiempo de suspensión medido frente al tiempo de suspensión programado se obtiene un comportamiento como el mostrado en la figura 2.6. La recta se obtiene por regresión de los puntos medidos entre 10 ms y 100 µs (rango al que se supone que es inferior la resolución que se mide). Se considera como resolución el tiempo en el que el error frente a la recta de regresión es superior al 10%.

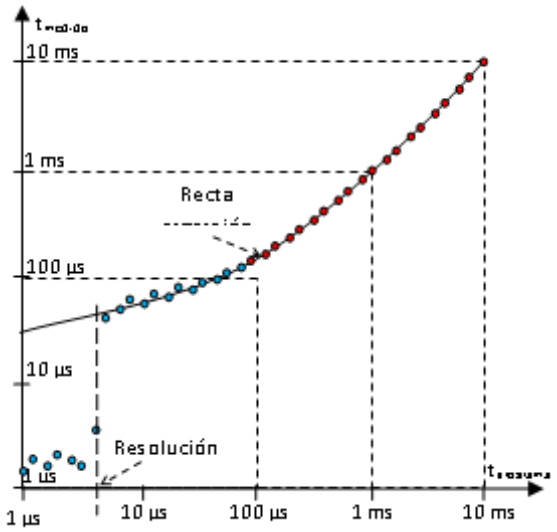


Figura II.6: Medidas para evaluar la resolución del reloj de tiempo real del RTSJ

II.3 Programa de medida del modelo de la plataforma

La herramienta RTLinuxNodeModelEstimator es un programa RT-Java que ejecutado sobre la plataforma RTSJ y sobre Ubuntu Linux 2.6.X.rt., genera un fichero con el modelo MAST 2, así como un informe de texto con información complementaria. En la figura 2.7 se muestra su estructura a través de un diagrama de clases.

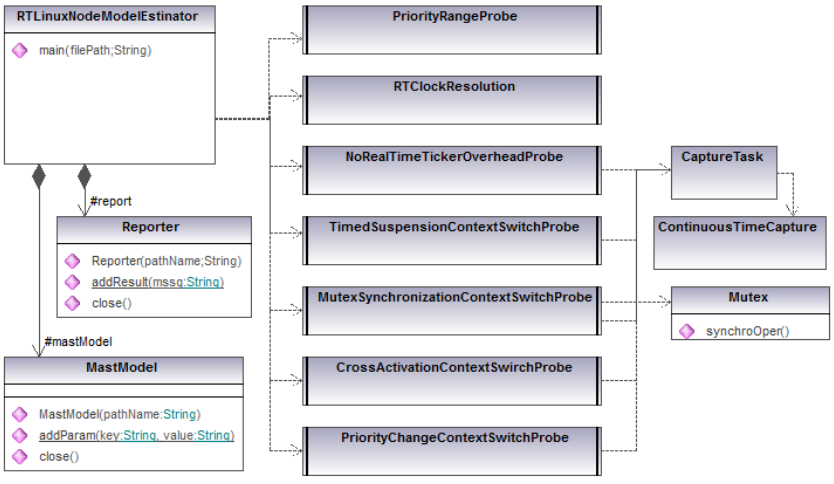


Figura II.7: Estructura del programa RTLinuxNodeModelEstimator

El método *main()* de la clase *RTLinuxNodeModelEstimator* ejecuta la aplicación de forma completa. Solo en el caso de que se presenten algún tipo de error, presentaría el correspondiente aviso a través de la consola. Su actividad es muy simple:

- Construye las instancias de las clases auxiliares *Reporter* y *MastModel.*

- A través de un thread de tiempo real efímero, instancia en memoria inmortal los recursos que van a ser utilizados por el conjunto de prueba, de forma que las estructura de datos masivas sean reutilizadas por las diferentes pruebas.
- Crea sucesivamente una instancia de cada una de las pruebas que estiman los diferentes parámetros del modelo. La creación es efímera, esto es, invocando al constructor se instancia y dentro del propio constructor se ejecuta la prueba, registrando los resultados en las clases *Reporter* y *MastModel* y finalizando su actividad tras concluir la ejecución del constructor.
- Se cierran las instancias *Reporter* y *MastModel*, con lo que se almacenan de forma persistente los resultados.

Las pruebas son clases activas con uno o varios threads de tiempo real que realizan las tareas que se han descrito en la sección II.2, y que valúan los valores de los parámetros del modelo de la plataforma:

- *PriorityRangeProbe*: Evalúa el rango de prioridades de tiempo real que admite la plataforma.
- *RTClockResolution*: Evalúa la resolución del reloj de tiempo real de la plataforma.
- *NoRealTimeTickerOverheadProbe*: Evalúa los tiempos de CPU de sobrecarga que introduce la gestión del timer de 1 ms de Java estándar.
- *TimedSuspensionContextSwitchProbe*: Evalúa los tiempos de cambio de contexto que se produce cuando se despierta por finalización de su plazo de suspensión un thread con prioridad superior al que se está ejecutando.
- *MutexSynchronizationContextSwitchProbe*: Evalúa los tiempos de cambio de contexto que se produce cuando un thread de mayor prioridad finaliza el bloqueo de acceso a un mutex que está siendo utilizado por un thread de prioridad inferior.
- *CrossActivationContextSwitchProbe*: Evalúa los tiempos de cambio de contexto que se producen cuando un thread de mayor prioridad que está suspendido (wait) en el lock de una instancia es activada (notify) por otro thread de prioridad inferior.
- *PriorityChangeContextSwitchProbe*: Evalúa los tiempos de cambio de contexto que se producen cuando como consecuencia de un cambio de prioridad explícito de un thread un nuevo thread pasa a ser planificado.

Como ejemplo, en la figura 2.8 se muestran los detalles de la prueba *PriorityChangeContextSwitchProbe*. En este caso se utilizan dos thread de alta prioridad. Uno *fixedPrtyThr* de prioridad intermedia fija, y otro *varPrtyThr* de prioridad variable que alternativamente (cada *execReadingNum* lecturas) cambian su prioridad de un valor superior a un valor inferior de la de la tarea *fixedPrtyThr*.

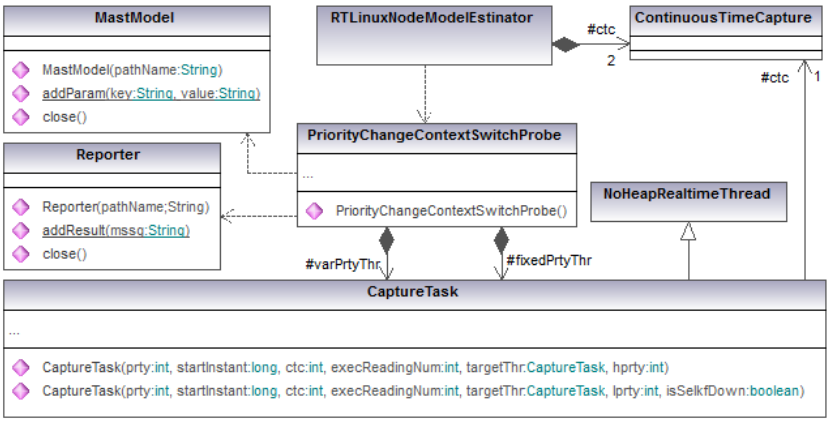


Figura II.8: Elementos de PriorityChangeContextSwitchProbe

La actividad de esta prueba, que es básicamente común con el de otras pruebas, y consiste en:

- Declara las variables locales que va a utilizar.
- Crea un thread *th* de tiempo real efímero para construir las dos tareas varPrtyThr y fixedPrtyThr de lectura continua de reloj que al operar en memoria inmortal no pueden ser creadas directamente del thread Java procedente de *main()*. Las dos tareas se crean con constructores diferentes porque su actividad interna también lo es.
- Se espera a que finalice el thread *th*, lo que representa que la lectura de los datos ha finalizado.
- Se procesa estadísticamente las lecturas del reloj realizadas para obtener los tiempos de cambio de contexto, de acuerdo con los mostrados en la figura 2.4.
- Se registran utilizando los métodos estáticos *addParams()* de MastModel y addresult() de Reporter los valores de los parámetros estimados.

La clase *Reporter* recibe de su constructor el *pathFile* del fichero en el que se debe dejar almacenado el informe con los resultados. Recibe (*addResult()*) de las diferentes pruebas los mensajes que esta generan, y cuando finaliza su actividad (close()) genera el fichero correspondiente en disco.

La clase *MastModel* recibe igualmente de su constructor el *pathFile* del fichero en el que se va generar el modelo de la plataforma. Recibe (*addParams()*) de las diferentes pruebas los datos estimados en base a parejas [*key*, *value*], que deben ser interpretados de acuerdo con alguna estrategia en los parámetros del modelo. Este es creado cuando al finalizar se invoca *close()*. En este caso la estrategia que se ha utilizado para establecer los únicos valores de cambio de contexto que tiene el modelo en base a los cuatro conjunto de valores de cambio de contexto que se han estimado ha sido:

- No se ha considerado los cambios de contexto por modificación explícita de la prioridad de los threads (*PriorityChangeContextSwitchProbe*)que son muy altas, y sólo ocurren en el patrón de diseño BoundedJitterPeriodicExecutor.

- Se ha asignado al parámetro WorstContextSwitch del modelo el mayor de los otros tipos de cambio de contexto (*TimedSuspensionContextSwitchProbe, MutexSynchronizationContextSwitchProbe y CrossActivationContextSwitch Probe).*
- Se han asignado a los parámetros *BestContextSwitch y AvgContextSwitch* del modelo los valores del tipo de cambio de contexto (excluido *Priority ChangeContextSwitchProbe*) que tenía el menor valor.

Hay que establecer en el lanzamiento de la herramienta que, en el caso de que el nudo tenga diferentes procesadores, la aplicación se ejecute únicamente en uno de ellos. Así mismo se ha inhibido que la JVM utilice el mecanismo JIT (Just-In-Time compilation) que hace que las primeras medidas estén afectadas por los procesos de compilación. La sentencia de lanzamiento de la aplicación ha sido:

```
schedtool –a 0x01 –e java-rt –XX:-JITRT RTLinuxNodeModelEstimator
```

En las Tabla II.1 y Tabla *II.2* se muestran ejemplos de los ficheros obtenidos para el informe y para el modelo, cuando se ha estimado el modelo de una determinada plataforma.

```
<?xml version="1.0" encoding="UTF-8"?>
<mast_mdl:MAST_MODEL Model_Name="RTLinuxNode"
                Model_Date="2013-04-03T23:05:18"
                xmlns:mast_mdl="http://mast.unican.es/xmlmast/model"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://mast.unican.es/xmlmast/model
Mast_2_0_v8.xsd">
        <mast_mdl:Regular_Processor Name="RTLinuxNode"
                        Max_Interrupt_Priority="0"
                        Min_Interrupt_Priority="0"
                        System_Timer="RTSJClock">
                <mast_mdl:Timer Name="nrtJavaTimer"/>
        </mast_mdl:Regular_Processor>
        <mast_mdl:Alarm_Clock Name="RTSJClock"
                        Max_Overhead="0.0"
                        Avg_Overhead="0.0"
                        Min_Overhead="0.0"
                        Precision="5.009E-6"/>
        <mast_mdl:Ticker Name="nrtJavaTimer"
                        Period="1.000E-3"
                        Precision="1.000E-3"
                        Max_Overhead="45.53E-6"
                        Avg_Overhead="14.43E-6"
                        Min_Overhead="12.64E-6"/>
        <mast_mdl:Primary_Scheduler Name="scheduler"
                        Host="RTLinuxNode">
                <mast_mdl:Fixed_Priority_Policy
                                Min_Priority="11"
                                Max_Priority="58"
                                Worst_Context_Switch="28.96E-6"
                                Avg_Context_Switch="9.036E-6"
                                Best_Context_Switch="8.171E-6"/>
        </mast_mdl:Primary_Scheduler>

</mast_mdl:MAST_MODEL>
```

Tabla II.1: Modelo MAST 2 generado por la herramienta

```
*************** RTLinuxModelEstimator application results report ****************

*** Priority Range Probe ***********************************************************
   Highest Real-time priority = 58
```

```
   Lowest Real-time priority = 11
********************************************************************************
*** No real-time ticker overhead ************************************************
   Analyzed interval: 10.00   Resolution= 1.568E-6
   *** nrtTickerOverhead ***************************
       period= 1.000E-3
       Activity: offset= 0
          Maximum ovehead time= 45.53E-6
          Minimun ovehead time= 14.43E-6
          Average ovehead time= 12.64E-6
   *** Maximal Resting activity time: 689.3E-6
********************************************************************************
*** Timed Suspension Context Switch Probe ***************************************
   Number of context switches = 5772
   Maximum context switch time = 28.96E-6
   Minimum context switch time = 10.82E-6
   Average context switch time = 12.76E-6
********************************************************************************
*** Priority Change Context Switch Probe ****************************************
   Number of context switches = 7996
   Maximum context switch time = 149.7E-6
   Minimum context switch time = 54.75E-6
   Average context switch time = 59.87E-6
********************************************************************************
*** Mutex Synchronization Change Context Switch Probe ***************************
   Number of context switches = 6664
   Maximum context switch time = 26.47E-6
   Minimum context switch time = 10.89E-6
   Average context switch time = 11.85E-6
********************************************************************************
*** CrossActivation Change Context Switch Probe *********************************
   Number of context switches = 3998
   Maximum context switch time = 23.53E-6
   Minimum context switch time = 8.241E-6
   Average context switch time = 9.223E-6
********************************************************************************
*** Real-Time Clock Resolution Probe ********************************
   Real-time Clock resolution = 5.009E-6
********************************************************************************
```

Tabla II.2: Documento con la información generada por la aplicación

III Herramienta para la generación automática del modelo de la aplicación

III.1 Estrategia de la herramienta

Hay que conocer la planificabilidad de una aplicación de tiempo real frente a la carga de trabajo que introduce un determinado entorno. Esto es, en una aplicación de tiempo real estricto, hay que poder determinar si se puede garantizar que no va a perder los plazos establecidos en su especificación cuando opera en ese entorno.

El elemento final de este proyecto es una herramienta que, dado el código (o archivos de código) de una aplicación genere de forma automática un código similar, pero instrumentado, con nuevas líneas que, posteriormente, generen el modelo de planificabilidad MAST, cuando la aplicación se ejecute en un entorno de configuración.

Estas líneas de instrumentación serán invocaciones a un módulo denominado TimeMntr, introducido en el proyecto de configuración para identificar los métodos que son invocados por la aplicación y los tiempos de CPU que requieren su ejecución y, con eso, crear el modelo MAST de la aplicación y asignar valores a los atributos de sus elementos.

El proceso que sigue la herramienta es, en cada ciclo de ejecución, registrar los tiempos de duración de cada una de las operaciones que utilizan un mutex, que han debido de ser claramente etiquetadas e instrumentadas, obteniendo unos tiempos máximos, mínimos y medios de ejecución, así como el registro del tiempo de CPU que utiliza la actividad de cada thread en sus activaciones.

Una vez instrumentada la aplicación, ésta es ejecutada en una plataforma idéntica a la plataforma de ejecución final, pero en un entorno de laboratorio (*Configuration enviroment*), específicamente diseñado para que en ella se produzcan todos los fenómenos que están especificados en el entorno físico, y con la adecuada frecuencia para que los resultados estadísticos sean significativos.

III.2 Definición de las anotaciones del código Java

A la hora de instrumentar el código de la aplicación de tiempo real es necesario poder identificar el tipo de clase que es (ejecutora, protegida…) como los diferentes tipos de métodos con papel específico que contiene (entry, trigger, protected…), de forma que el código quede asociado, de forma inequívoca, con el modelo reactivo de la aplicación.

Para instrumentar y definir dónde hay que introducir esas nuevas instrucciones y cuáles han de ser, han sido creadas unas anotaciones de estilo JAVADOC (/* *@OORT_.... */) las cuales documentan el código y ayudan a la herramienta a trabajar. Estas anotaciones se han definido a partir de los estereotipos introducidos con los patrones junto a un OORT_ para diferenciarlas de otras anotaciones de tipo JAVADOC, que pudieran ser iguales o parecidas y pudieran llevar a la confusión a la herramienta.

Definiremos las diferentes etiquetas acorde a los distintos patrones de diseño explicados anteriormente, que se colocarán, en el código, sobre la clase o método que tiene un estereotipo específico:

- Patrón executor:

```
/* * @OORT_eventHandler // @OORT_periodicExecutor */
Clase Executor {
    Constructor(){
            …
    }

    public void run(){
            while(!interrupted()){
                    …
            }
    }
}
```

Las etiquetas ***@OORT_EventHandler*** u ***@OORT_PeriodicExecutor*** se encuentran en aquellas clases ejecutoras, manejadoras de eventos que se ejecutarán en respuesta a un evento determinado o periódicas, respectivamente, y que constan de un thread de tiempo real, o son una clase que extiende a dicho tipo de thread y, por consiguiente, han de tener definido un método run() correspondiente a la actividad que realiza ese thread.

- Patrón protegido:

```
/** * @OORT_protected // @OORT_eventSource // @OORT_synchronizer */
Clase Protected {
    Constructor(){
            …
    }

    /** * @OORT_synchronized */
    synchronized(){
            …
    }

    /** * @OORT_entry */
    entry(){
            …
            wait();
            …
            return;
    }

  /** * @OORT_postWaiting */
    postWaiting(){
            …
    }

    /** * @OORT_eventTrigger */
    eventTrigger(){
            …
            notify();
            …
    }
}
```

La etiqueta ***@OORT_protected*** (o cualquiera de las relativas a sus estereotipos descendientes) indica que la clase descrita a continuación está utilizada como un objeto protegido, es decir, los accesos concurrentes a los métodos anotados con la etiqueta ***@OORT_synchronized*** (o cualquiera de sus estereotipos descendientes) se realizarán en régimen de exclusión mutua, con la primitiva synchronized ya sea en la cabecera del método o englobando una parte del código del este.

Con la anotación ***@OORT_eventSource*** se entiende que la clase descrita es una fuente de eventos, incluirá entre sus métodos un ***@OORT_entry***, que contienen entre sus líneas de código un método wait() que provoca la suspensión de aquellos threads que pasen por ella.

También se pueden encontrar métodos etiquetados como ***@OORT_postWaiting***, que serán aquellos que se realizarán de forma protegida tras salir de la suspensión provocada por el wait() de un método etiquetado como @OORT_entry.

La etiqueta ***@OORT_synchronizer*** se coloca en aquellas clases responsables de sincronizar dos threads distintos que participan en la ejecución de una misma respuesta a un evento. Para ello, además de un método etiquetado como @OORT_entry, la clase contendrá otro, etiquetado como ***@OORT_trigger***, que será el que produce el cambio de estado que activa al thread que haya quedado suspendido en un método wait(). Para ello, ha de contener en el código un método notify() que notifique aquellos threads que estén esperando.

- Main:

```
/** * @OORT_main */
Clase Main {
    Main(){
            new NoHeapRealTimeThread().run(){
                    … .start() / .close() …
            }
    }
}
```

En la clase Main, aquella que comienza la ejecución de la aplicación basada en patrones sobre la que se instrumentará el código, el etiquetado es tan sencillo como estereotiparlo con ***@OORT_main*** sobre la línea de código que comienza la clase y la da nombre.

III.3 Módulo TimeMntr

La instancia de la clase TimeMntr constituye un módulo de la aplicación instrumentada que dispone de los métodos que miden los tiempos de CPU de las respuesta de los threads de la aplicación y de los métodos anotados, almacenando en unas estructuras de datos internas la información estadística de estos y que, posteriormente, serán las que se utilizarán para crear el modelo MAST de la aplicación.

El módulo proporciona las operaciones necesarias para captar los instantes de tiempo de proceso en el que se está ejecutando una clase o un método, para calcular su tiempo de ejecución, así como las estructuras de datos internas que guardan todos los datos necesarios para crear posteriormente el modelo de aplicación.

Las clases principales, como se observan en la Figura III.1, y sus operaciones más importantes son:

Clase **TimeMntr**: Es la clase central del módulo. Contiene los métodos estáticos que generan los datos relativos a los tiempos de ejecución de la aplicación instrumentada. Además, contiene el método nativo JNI que llama a una función C, encargada de consultar los relojes de ejecución de cada thread.

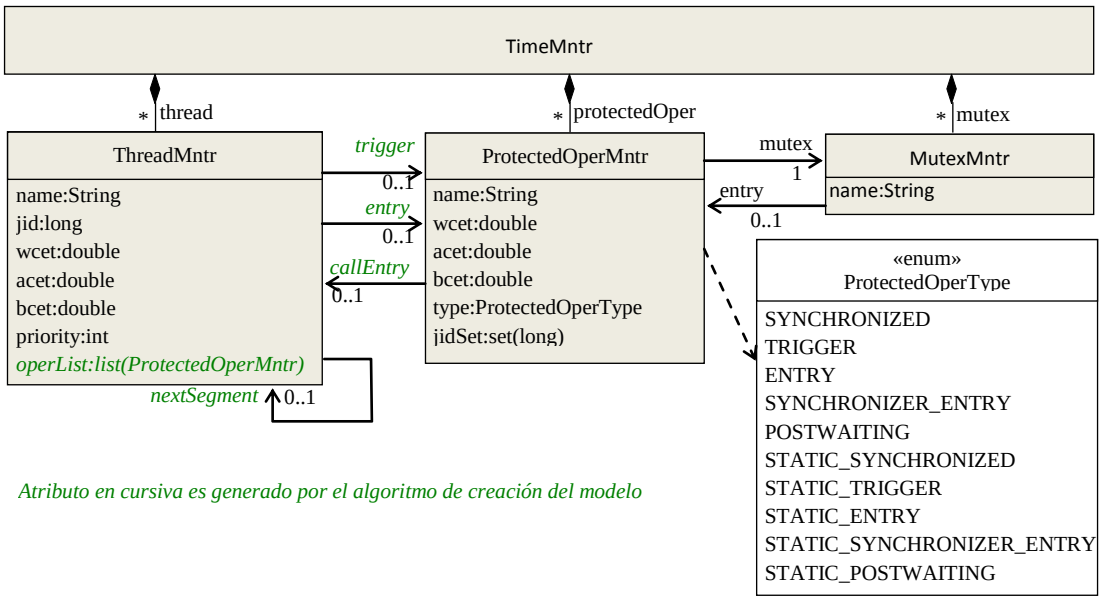


Figura III.1: Estructura del módulo TimeMntr

Los atributos más significativos de esta clase son:

- thread:List[ThreadMntr] -> Lista con las estructuras de datos asociadas a los monitores de threads registrados por la aplicación, y que posteriormente se utilizará para crear los elementos de modelado *Thread* del modelo MAST de la aplicación.

- protectedOper:List[ProtectedOperMntr] -> Lista don las estructuras de datos asociadas a los monitores de operaciones protegidas registrados por la aplicación, que en el modelo MAST darán lugar a elementos de modelado *Simple_Operation*, que referencia el uso de un determinado mutex.

- mutex:List[MutexMntr] -> Lista de las estructuras de datos asociadas a los recursos protegidos registrados por la aplicación. Cada elemento de la lista dará lugar a un elemento de modelado mutex en el modelo MAST de aplicación.

Los métodos más importantes de TimeMntr son los siguientes:

- createThreadMntr(name:String):int -> Método estático que crea un nuevo monitor de thread y lo añade a su correspondiente lista, retornando el índice de ésta en el que se ha colocado. La información necesaria para crearlo es solamente su nombre *name*, que se utilizará para verificar si el monitor existe ya en la lista o es necesario crear uno nuevo.

- createProtectedOperMntr(name: String, rsrc: ProtectedResource, operType: ProtectedOperType): int -> Método estático que crea un nuevo monitor de operación protegida y lo añade a su correspondiente lista, retornando el índice de ésta en el que se ha colocado. Para crear un nuevo monitor se necesita, además de su nombre *name*, el recurso protegido *rsrc* en el que se encuentra este método y el tipo de operación *operType* que es, con lo que posteriormente se crearán los End-to-End flows.

- createProtectedResource(name:String): ProtectedResource -> Método estático que crea un nuevo recurso protegido de nombre *name* y lo añade a su correspondiente lista. Retorna un puntero al propio recurso protegido.

- recordThrProcessInstant(source:int) -> Método estático que ejecuta un método nativo JNI que lee un instante de reloj del proceso en ejecución (cuyo índice de la lista de monitores de thread es *source*) para calcular los valores máximos, mínimos y medios de los tiempos de ejecución de ese thread.

- recordOperProcessInstant(source:int) -> Método estático que ejecuta un método nativo JNI que lee un instante de reloj del proceso en ejecución para calcular los valores máximos, mínimos y medios de los tiempos de ejecución de una operación en concreto (cuyo índice de la lista de monitores de operaciones protegidas es *source*).

- mastModel() -> Método que crea y escribe el fichero MAST de acuerdo a los resultados obtenidos por los monitores de thread y operaciones protegidas.

DataType ThreadMntr: tipo de dato que monitoriza la actuación de cada thread individualmente. Puede tener asociados un monitor de operación protegida que señale la operación *entry* del thread y otro que señale el posible *trigger*. Además, en el caso de que estemos ante una secuencia de threads, también puede asociarse con el siguiente segmento de la secuencia, en este caso otro monitor de thread.

Los atributos de este datatype son:

- name:String -> Nombre del monitor.

- acet:long = 0-> Atributo derivado de la suma de todos los tiempos de ejecución registrados entre el número de ejecuciones que se han registrado, por lo que se obtiene el tiempo medio de ejecución.

- wcet:long = Long.MIN_VALUE -> Peor tiempo de ejecución registrado.

- bcet:long = Long.MAX_VALUE -> Mejor tiempo de ejecución registrado.

- JID:long -> Identificador de thread Java correspondiente al thread monitorizado por esta instancia.

- priority:int -> Prioridad a la que ejecuta el thread monitorizado por esta instancia.

- operList:List[ProtectedOperMntr] -> Lista de todas las operaciones protegidas que ejecuta este thread. Todas estas operaciones, posteriormente, conformarán la lista de *Enclosing_Operation*, en el modelo de aplicación.

DataType ProtectedOperMntr: tipo de dato que monitoriza la actuación de cada una de las operaciones instrumentadas de las distintas clases. Hereda los atributos y el método del tipo de dato Mntr y añade el tipo de operación que es y los thread que la ejecutan. Está asociado

con el mutex al que pertenece y puede estar también asociado con un ThreadMntr, en el caso en el que sea la operación sea una entry, señalando el thread que lo inicia.

Estos son sus atributos:

- name:String -> Nombre del monitor.
- acet:long = 0-> Atributo derivado de la suma de todos los tiempos de ejecución registrados entre el número de ejecuciones que se han registrado, por lo que se obtiene el tiempo medio de ejecución.
- wcet:long = Long.MIN_VALUE -> Peor tiempo de ejecución registrado.
- bcet:long = Long.MAX_VALUE -> Mejor tiempo de ejecución registrado.
- threadJID:Set<long> -> Conjunto de los identificadores de thread que utilizan la operación protegida monitorizada.
- operType:ProtectedOperType -> Tipo de operación protegida, dado por uno de los slots del enumerado ProtectedOperType.

DataType MutexMntr: tipo de dato que instancia todos aquellos recursos protegidos utilizados en la aplicación instrumentada. En el caso de que el recurso protegido tenga un método entry, éste estará asociado con el monitor de esa operación.

Su único atributo es el siguiente:

- name:String -> Nombre del recurso protegido.

Enumeration ProtectedOperType: clase enumerada que define los diferentes tipos de operaciones protegidas que podemos encontrar en una aplicación de tiempo real.

Slots:

- ENTRY
- POSTWAITING
- TRIGGER
- SYNCHRONIZED
- SYNCHRONIZER_ENTRY
- STATIC_ENTRY
- STATIC_POSTWAITING
- STATIC_TRIGGER

- STATIC_SYNCHRONIZED
- STATIC_SYNCHRONIZER_ENTRY

El mecanismo por el cual las operaciones recordOperProcessInstant(source:int) y recordThrProcessInstant(source:int) obtienen los tiempos de ejecución, tanto de operaciones como de threads respectivamente, es una llamada a una función C. Para ello se utiliza la JNI o Interfaz nativa de Java, que utiliza librerías compiladas a través de archivos C para utilizar comandos de este lenguaje.

En este caso, utilizamos el método nativo JNIgetProcessInstant() que hace llamada a la función C descrita en la Tabla III.1*: Código de la funcion C Java_TimeMntr_JNIgetProcessInstant* y devuelve el calor del reloj en el instante que se invoca.

```
jlong Java_TimeMntr_JNIgetProcessInstant(JNIEnv* env, jobject obj)
{
    clockid_t clockid;
    struct timespec ts;
    int error;

    error = pthread_getcpuclockid(pthread_self(),&clockid);
    if (error != 0){
        perror("Error al obtener clockId\n");
        return NULL;}

    error = clock_gettime(clockId, &ts);
    if (error != 0){
        perror("Error al obtener tiempo\n");
        return NULL;}

    return (jlong)ts.tv_sec*1000000000+ts.tv_nsec;
}
```

Tabla III.1: Código de la funcion C Java_TimeMntr_JNIgetProcessInstant

La actividad de esta función es sencilla, escrito bajo el estándar POSIX de tiempo real, la función obtiene el reloj del thread que la ejecuta y, con él, posteriormente, el instante exacto de ejecución, que es el valor de retorno.

El archivo C que contiene esta, y el resto de operaciones en el caso de que las hubiera, ha de compilarse junto al su esqueleto (en un archivo H) de forma que se crease una librería cuyo nombre ha de ser libNombreLibreria.so.

Tanto en recordOperProcessInstant(source:int) como recordThrProcessInstant(source:int), la aplicación discierne si es la llamada de inicio de ejecución o de final, en cuyo caso hace las operaciones necesarias para obtener la duración total y asignarlo, si es preciso, como mejor o peor caso de tiempo de ejecución.

Por último, como se ha comentado anteriormente, el método mastModel() de cada monitor crea el modelo MAST a partir de los datos almacenados durante la ejecución de la aplicación. Este modelo MAST se crea encadenando las llamadas a este método de las diferentes instancias de monitores mediante el algoritmo recogido en la Tabla III.2:

```
1 Se crea la cabecera del modelo y se abre un elemento Mast_Model
2 FOR cada theMutex:MutexMntr en TimeMntr.mutex:
```

```
  2.1  Introduce un elemento Priority_Inritance_Mutex relativo a él
3 FOR cada theOper:ProtectedOperMntr en TimeMntr.protectedOper:
  3.1  Introduce un elemento Simple_Operation relativo a él
  3.2  FOR cada theJid:long en el set theOper.jidSet :
       3.2.1  Identifica en la lista TimeMntr.thread el theThread tal que
theThread.jid=theJid
       3.2.2 Añade theOper a la lista theThread.operList.
       3.2.3 IF (theOper.type=ENTRY | theOper.type=STATIC_ENTRY|
                 theOper.type=SYNCHRONIZER_ENTRY |
theOper.type=STATIC_SYNCHRONIZER_ENTRY|):
             3.2.3.1 theOper.callEntry= theThread
             3.2.3.2 theThread.entry=theOper
             3.2.3.3 theThread.isSegment= (theOper.type =SYNCHRONIZER_ENTRY)|
                                          theOper.type =STATIC_SYNCHRONIZER_ENTRY)
       3.2.4 IF (theOper.type=TRIGGER | theOper.type=STATIC_TRIGGER)
             3.2.4.1 theThread.trigger=theOper
4 FOR cada theThread:ThreadMntr in TimeMntr.thread
  4.1 Intriduce un elemento Thread con un Fixed_Priority_Params relativo a theThread
  4.2 IF (theThread.trigger!=null)
      4.2.1 theThread.nextSegment=(((theThread.trigger).mutex).entry).callEntry
5 FOR cada theThread:ThreadMntr in TimeMntr.thread
  5.1 IF (!theThread.isSegment)
      5.1.1 Introduce y abre un elemento End_To_End_Flow relativos a theThread
      5.1.2 IF (theThread.entry=null)
            5.1.2.1 Introduce un elemento Periodic_Event
      5.1.3 IF(theThread.entry!=null)
            5.1.3.1 Introduce un element Sporadic_Event
      5.1.4 Introduce un element Internal_Event con un elemento Hard_Global_Deadline
      5.1.5 Introduce un element Step
      5.1.6 WHILE (theThread.nextSegment!=null)
            5.1.6.1 theThread=theThread.nextSegment
            5.1.6.2 Introduce un element Internal_Event con un element
Hard_Global_Deadline
            5.1.6.3 Introduce un element Step
      5.1.7 Cierra elemento End_To_End_Flow
6  Cierra el elemento Mast_Model
```

Tabla III.2: Pseudocódigo de la función MastModel

III.4 Instrumentación de la aplicación en base a las anotaciones

En este apartado se describen las líneas de código que son añadidas al código de la aplicación para constituir el código de la aplicación instrumentada. Son líneas que invocan métodos del módulo TimeMntr y que hacen que se registre en el la información temporal que se utilizará para generar el modelo MAST de la aplicación.

Todos los archivos que vayan a ser tratados por la aplicación TimeMntr para conseguir su correspondiente especificación MAST han de importar el paquete correspondiente a este módulo, insertando al principio del código la línea `import TimeMntr.TimeMntr` o, de forma genérica, `import TimeMntr.*`.

Dependiendo de las anotaciones que etiqueten las diferentes clases y métodos del código a analizar, este se verá afectado de distinta forma. Se explicará, de la misma forma que en el apartado III.2, atendiendo a los diferentes patrones explicados:

- Patrón executor:

```
import TimeMntr.*;

 /* * @OORT_eventHandler // @OORT_periodicExecutor */
 Clase Executor {
     private int monitores;
     Constructor(){
            …
     }

     public void run(){
```

```
                createThreadMnrt();
                recordThrProcessInstant();
                while(!interrupted()){
                        …
                        recordThrProcessInstant();
                }
        }
}
```

Como es un objeto activo en la aplicación, en el constructor de la clase anotada con la etiqueta @OORT_EventHandler o @OORT_periodicExecutor ha de crearse en TimeMntr un elemento monitor de thread, correspondiente al que se inicia en esta clase. Este ha de hacerse al comienzo del método run(), que es ejecutado por el thread en activo, con sus parámetros de planificación correspondientes y cuyo índice se guardará en un atributo especialmente creado para ello.

A este monitor hay que añadir las líneas de código necesarias para anotar la duración de la respuesta, que se resuelve con operaciones que leen el tiempo de thread antes de iniciar el bucle y al final de este, también dentro del propio método run().

- Patrón protegido:

```
import TimeMntr.*;

 /** * @OORT_protected // @OORT_eventSource // @OORT_synchronizer */
 Clase Protected {
     private int monitores;
     Constructor(){
             …
             createProtectedResource();
             createProtectedOperMntr();
     }

     /** * @OORT_synchronized */
     synchronized(){
             recordOperProcessInstant();
             …
             recordOperProcessInstant();
     }

     /** * @OORT_entry */
     entry(){
             recordOperProcessInstant();
             …
             wait();
             …
             recordOperProcessInstant();
             return;
     }
    /** * @OORT_postWaiting */
     postWaiting(){
             recordOperProcessInstant();
             …
             recordOperProcessInstant();
     }

     /** * @OORT_eventTrigger */
     eventTrigger(){
             recordOperProcessInstant();
             …
             notify();
             …
             recordOperProcessInstant();
     }
}
```

Para las clases etiquetadas como @OORT_protected (o sus derivados), al tratarse de objetos protegidos, MAST debe crear un mutex que es accedido al ejecutar sus operaciones synchronized. Para ello, en el propio constructor de la clase (que si no está incluido en la clase

se crearía uno) al final de éste, se añade un nuevo recurso a la lista propia de recursos del módulo TimeMntr y un monitor para cada una de las operaciones protegidas (etiquetadas también con @OORT) que tenga la clase, con sus respectivos índices como atributos.

En cuanto a las operaciones protegidas de la clase, para todas ellas se ha de registrar el tiempo de ejecución de cada operación. A la hora de anotar el tiempo de ejecución de la operación, tanto en las operaciones marcadas como @OORT_protected se realizarán dos tomas de tiempo, una al comienzo de la operación y otra al finalizar la ejecución de todo el código de esta, siempre y cuando no haya un valor de retorno, en cuyo caso la línea se colocará justo antes del return. Una excepción a este caso son los métodos @OORT_entry, cuya segunda línea de toma de tiempo se realizará justo después del método wait(), propio de este tipo de operaciones.

- Main:

```
Importar paquete TimeMntr
 /** * @OORT_main */
 Clase Main {
     private static TimeMntr mntr;
     Main(){
             new NoHeapRealTimeThread().run(){
                     mntr = new TimeMntr();
                     … .start() / .close() …
                     mntr.stopRecordingData();
                     mntr.createMastModel();
             }
     }
}
```

En la clase Main, etiquetada como @OORT_main, la más importante dado que es la que comienza la ejecución de todos los threads de la aplicación, aparte de la importación del módulo TimeMntr, ha de instanciarse éste para que pueda comenzar la recogida de los datos.

Una vez comenzada la ejecución de la aplicación, las clases ejecutoras y protegidas irán registrando los tiempos como se ha explicado anteriormente, hasta que, se finalice esta, mediante los close() de los diferentes threads en activo. Una vez terminados, se para la recogida de datos y se crea el modelo de planificabilidad.

Como se ha comentado, todo este proceso de instrumentación del código se hace de forma automática, por lo tanto, una vez implementada la aplicación de tiempo real mediante los diferentes tipos de patrones que se han explicado, y etiquetados correctamente según la función que corresponda tanto en clases como en métodos, ha de instrumentarse de forma correcta para que TimeMntr cree el modelo MAST correspondiente a dicha aplicación.

La aplicación *RTCode4TimeMntr* lee las etiquetas de los archivos Java de la aplicación, y añade las líneas de código de instrumentación para generar de forma automática el código de la aplicación instrumentada.

La tarea de esta aplicación es incorporar las diferentes líneas de código de instrumentación (llamadas a las diferentes operaciones del módulo TimeMntr) al código de la aplicación de tiempo real basada en patrones. Para ello, y uno a uno, se van leyendo los diferentes archivos de código Java contenidos en el directorio que se le indica a la aplicación RTCode4TimeMntr.

Como se observa en la Figura III.2, se realizan dos fases de cada archivo: una primera lectura que extrae datos correspondientes al nombre de la clase, tipo (main, protected, periodicExecutor...), si posee constructor o no, o métodos estereotipados que contiene (todo

ello se almacena en un dataType creado específicamente para esto), y una segunda lectura que, basándose en la información obtenida en la primera y la lectura de las etiquetas correspondientes sirven como base para la escritura del archivo instrumentado.

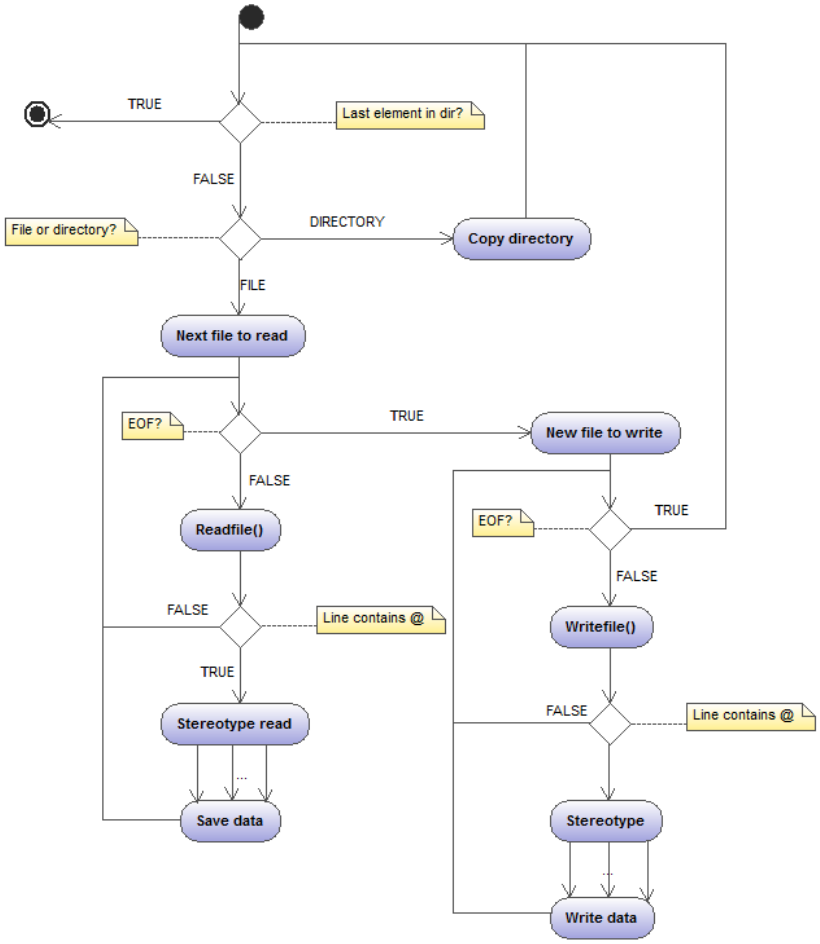


Figura III.2: Fases del autómata RTCode4TimeMntr

El dataType creado para esta aplicación contiene información sobre: nombre de la clase, tipo de clase que es, si tiene constructor, líneas de código que hay que añadir en la fase de escritura y número de métodos etiquetados que tiene la clase.

Para la fase de primera lectura, mostrada en la Figura III.3, se van extrayendo dinámicamente cada una de las líneas del archivo comprobando si contienen el símbolo @ que marca el comienzo de las etiquetas que se utilizan para estereotipar el código. En caso afirmativo, se comprueba la etiqueta y dependiendo de esta se realizan diferentes acciones:

- Para cualquier estereotipo de clase, inmediatamente se averigua el nombre de esta, que se almacena en el dataType.
- Para las clases estereotipadas como «protected», «eventSource», y «synchronizer», tras almacenar el nombre, se almacena la línea de código correspondiente a la creación de un monitor de recurso protegido en TimeMntr.

- Para las clases estereotipadas como executors («periodicExecutor» y «eventHandler»), tras guardar el nombre, se almacenan las líneas necesarias para que posteriormente se cree un monitor de thread para ese ejecutor.
- Para cualquier estereotipo de operación, se almacenan las líneas necesarias para que en la ejecución del código instrumentado se cree un monitor de operación protegida.
- En el caso de que se lea una etiqueta que no coincida con ningún estereotipo o este estereotipo esté incorrecto, se hace caso omiso y se continúa leyendo.

Una vez terminado con el archivo se realiza una nueva lectura rápida para comprobar si este tiene un método constructor o no, cuyo resultado se almacena en el dataType.

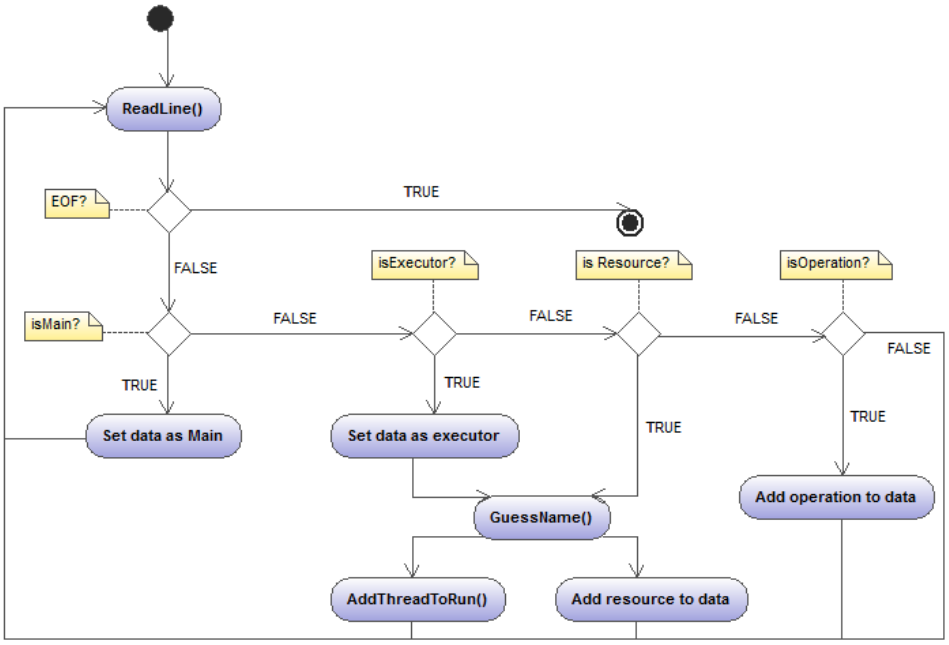


Figura III.3: Fase de lectura de RTCode4TimeMntr

En la segunda fase, de lectura-escritura descrita en la Figura III.4, la aplicación se apoya tanto en la lectura de las etiquetas como de la información recogida en la primera fase. Con todo esto dispuesto, y leyendo línea a línea el archivo, se pueden dar los siguientes casos:

Si se encuentran las etiquetas de clase («periodicExecutor», «eventHandler», «protected», «eventSource» o «synchronizer») y la clase se notifica como que no tiene constructor, se crea uno nuevo justo al comenzar la propia clase.

Si se encuentran los estereotipos de operación («synchronized», «trigger», «entry» o «postWaiting») se instrumenta el método como se ha descrito anteriormente de forma automática, es decir, al comenzar el método y, dependiendo del tipo de método que sea, después del wait(), antes del return, o al finalizar el método.

Si la clase contiene el main, se instrumenta este y no se realizan más acciones.

Si se encuentra el encabezado del constructor de la clase, se añaden los atributos correspondientes a los índices de los monitores fuera del constructor y, ya dentro de éste, justo antes de cerrar el método, las líneas correspondientes a, ya sea, creación de recursos compartidos, monitores de operaciones o de threads.

En las clases ejecutoras («periodicExecutor» y «eventHandler»), se busca el método run() correspondiente a la ejecución del thread, el cual se instrumenta en diferentes sitios: justo al inicio del método, antes de entrar en el bucle while() y justo al final de este mismo bucle.

En el caso de que se lea una etiqueta que no coincida con ningún estereotipo o este estereotipo esté incorrecto, se hace caso omiso, se escribe en el archivo de destino y se continúa leyendo.

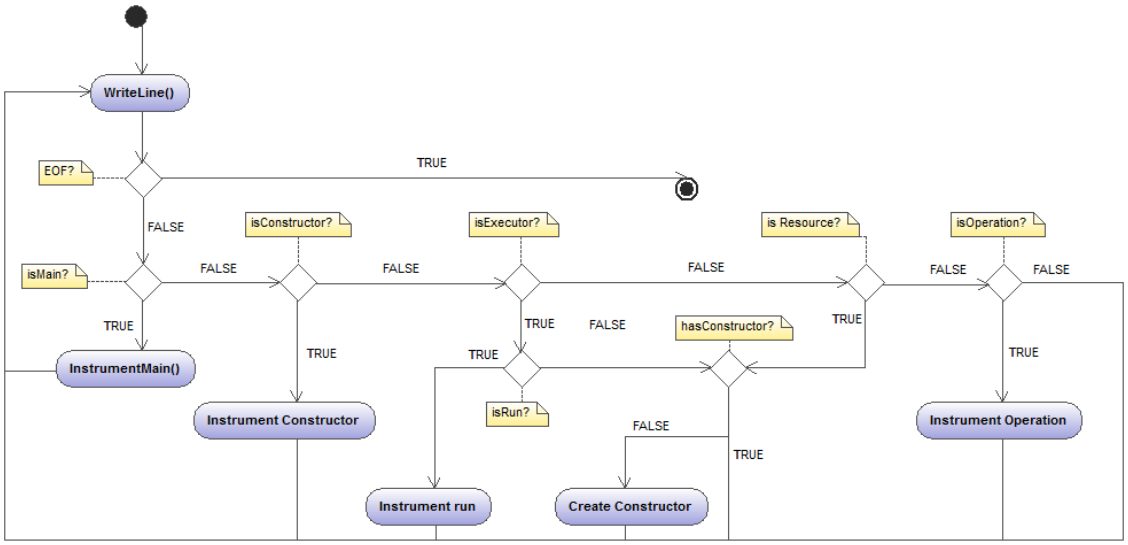


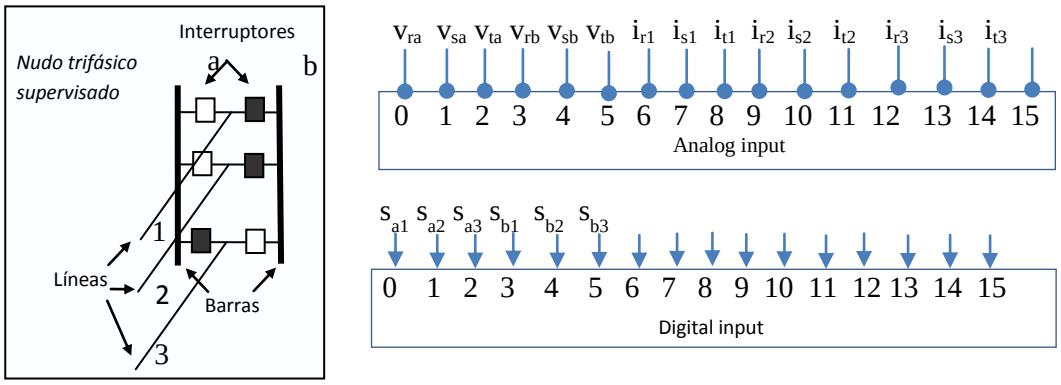
Figura III.4: Fase de escritura de RTCode4TimeMntr

IV Caso de uso

IV.1 Especificación y diseño lógico de la aplicación BURTA

Con el fin de validar la metodología y el proceso propuesto, se ha desarrollado una aplicación de prueba llamada BURTA (Aplicación de una unidad remota y teleoperada básica).

Corresponde a una aplicación con una funcionalidad básica que se ejecuta en una URT (unidad remota de teleoperación) de una subestación eléctrica a fin de gestionar la metrología, calidad de servicio, perturbometría y gestión del par de barras y de las tres líneas de interconexión trifásica que constituyen un nudo eléctrico de la subestación. La aplicación supervisa las tensiones e intensidades de un punto de conexión mediante interruptores de tres líneas trifásicas a dos nudos alternativos de tensión o barras. La aplicación muestrea las señales temporales de los dos pares de tensiones trifásicas de los nudos y las tres ternas de intensidades correspondientes a las líneas línea. Las procesa y genera información estadísticas sobre parámetros de metrología como valores eficaces, potencias, factores de desequilibrio, distorsión armónica, etc. También lee el estado lógico de los seis interruptores de conexión de las líneas a las barras. La información obtenida la publica periódicamente en el espacio de datos DDS, a fin de que pueda ser utilizada por otras aplicaciones de gestión de la red eléctrica. Así mismo, cuando se produce un evento transitorio relevante (perturbación) el sistema registra un conjunto de las señales temporales (perturbograma) que permiten visualizar o analizar los fenómenos que han ocurrido antes y después del evento.



Su funcionalidad se describe en los siguientes diagramas de módulos:

- **DAQ(RT-Driver)**: Driver Linux de tiempo real que permite el acceso al dispositivo ADQ de acceso a la tarjeta de entrada analógica y digital.
  Para evitar que se sobrescriban los bytes de memoria, la tarjeta genera el evento *HalfMemoryInterr* cuando se ha escrito la mitad de la memoria. La lectura de las señales analógicas se leen mediante un reloj interno de la tarjeta a razón de 32 muestras de cada una de las 15 señales de tensión e intensidad por ciclo de la red (50 Hz, esto es cada 20 ms).El driver permite también leer bajo demanda las líneas digitales de entrada.

- **Sampler**: Módulo que introduce el proceso que atiende los eventos *HalfMemoryInterr*, lee del driver la mitad de la memoria y las añade a la cola cíclica de Waveform formateada por variable eléctrica y en unidades físicas.

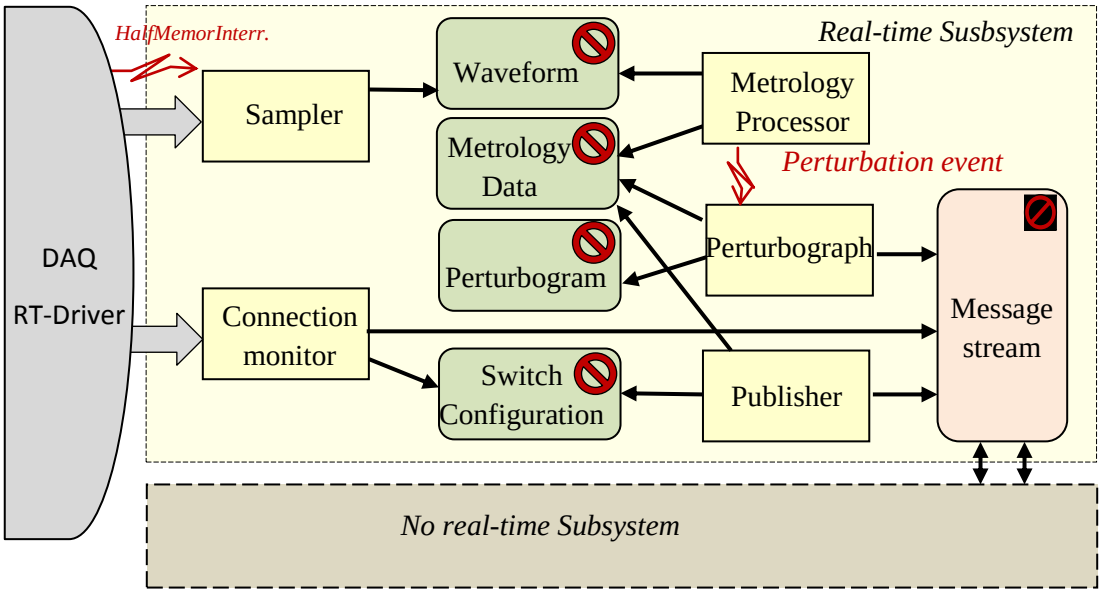


Figura IV.1: Estructura modular de la aplicación BURTA

- **Waveform**: Módulo que contiene los datos de las formas de ondas leídas y no procesadas. Está constituido por 15 estructuras de datos en forma de cola cíclica que serán accedidos concurrentemente, por lo que el acceso debe ser con exclusión mutua.

- **MetrologyProcessor**: Módulo que procesa las formas de onda disponibles en Waveform y evalúa: las magnitudes para mantener la estadística de metrología, las magnitudes para mantener la estadística de Calidad de servicio y las magnitudes para detectar y generar la información de los perturbogramas.

- **MetrologyData**: Estructura de datos compleja con las magnitudes para mantener las estadísticas de metrología y calidad de servicio. Va a ser accedido concurrentemente por los procesos de metrología y de publicación de datos, por lo que ha de ser protegida.

- **Perturbogram**: Estructura de datos con la información de una línea en la que se ha detectado una caída de tensión significativa. Como el pertubograma debe tener información de los datos previos al evento, MetrologyProcessor mantiene actualizado un buffer con las muestreas de las 15 señales, correspondientes al último segundo transcurrido. Esporádicamente cuando MetrologyProcessor detecta el estado de perturbación (PerturbationEvent), se activa el módulo Perturbograph, que recopilar y secuencializa la información completa de Perturbogram y lo publica en el espacio de datos DDS.

- **Perturbograph**: Módulo que contiene el proceso que genera el mensaje con el perturbograma registrado y que se activa esporádicamente.

- **ConnectionMonitor**: Modulo que supervisa por escrutinio periódico (polling) los cambios de estado de las conexiones de la línea. Cuando detecta un cambio transfiere un mensaje que lo publica al resto del sistema.

- **ConnectionConfiguration**: Estructura de datos que contiene la configuración actual de las conexiones, y un histórico de los hasta 100 cambios que se hayan producido sin ser publicados al resto del sistema.

- **Publisher**: Controla el proceso de publicación de los datos de metrología y calidad de servicio de una ventana temporal. No tiene requisitos temporales, pero ha de ser capaz de soportar bloqueos del MessageStreamer.

- **MessageStreamer**: Objeto de comunicación entre los subsistemas de tiempo real y de no tiempo real. Debe ofrecer mecanismo de sincronización no bloqueante a los threads de tiempo real

- **No real-time Subsystem**: Conjunto de procesos que no interaccionan con el subsistema de tiempo real por tener unas prioridades inferiores a las de los thread de tiempo real. La comunicación se realiza a través del MessageStreamer diseñado para ello.

IV.2 Especificación de tiempo real. Diseño reactivo y especificación mediante los estereotipos OORT

La aplicación se modelado bajo los patrones del modelo de referencia OORT anteriormente descrito y se ha desarrollado en RT-Java.

Para la validación de la metodología propuesta se ha simulado el driver Linux del DAQ, mediante un módulo RT-Java que se ejecuta con la prioridad más alta permitida por el lenguaje. Este módulo genera las interrupciones que activa el eventHandler que gestiona su atención.

Para describir el método de formulación del modelo reactivo, en esta memoria, sólo se incluyen los aspectos relativos a una respuesta MetrologyUpdating. En la Figura IV.2 se muestran las clases que representan la implementación de la respuesta (MetrologyUpdating) de la aplicación BURTA. Una tarea periódica procesa las formas de onda de las señales físicas capturadas de la red eléctrica y almacenadas en un conjunto de estructuras de datos (Waveform), y genera una información integrada sobre la metrología de las líneas que se almacenan en una estructuras de datos (MetrologyData) para su posterior publicación en la en espacio de datos DDS. Ocasionalmente, si se ha completado la generación de un perturbograma en respuesta a un fallo de la red, lo transfiere directamente al espacio de datos. Los módulos de publicación de información en el espacio de datos DDS no pertenecen al subsistema de tiempo real. La primera actividad tiene un requisito de plazo estricto, ya que si no se ejecuta antes de 200 ms, la información en las estructuras Waveform se sobrescribe. Por el contrario la actividad de la publicación del perturbograma en el espacio de datos no tiene requisito temporal. La clase Metrology es la clase que aporta la parte principal de la funcionalidad de esta respuesta. De hecho en el diseño de la aplicación no es una única clase sino todo un conjunto de clases pasivas que aportan la funcionalidad más específica:

DistortionAnalysis, QualityOfService, etc. La clase *«RTTaskJob»*Metrology juega un papel relevante en el framework, ya que al ofrecerse en ella el método *«ExecutorTask»*update() que es el que define globalmente la tarea que se ejecuta periódicamente y por tanto su tiempo de ejecución debe ser medido. El thread que planifica la sección de la respuesta con plazo temporal lo aporta la clase *«PeriodicExecutor»*MetrologyExecutor, y está declarado como una clase interna de la clase Metrology.

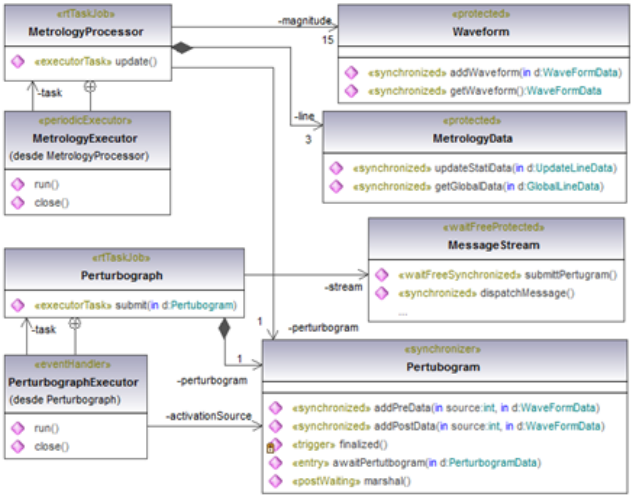


Figura IV.2: Diseño de la respuesta MetrologyUpdating

La segunda parte de la respuesta relativa a la transferencia de datos hacia el publicador DDS, se hace sin requisitos temporales y por ello se planifica por un segundo thread de baja prioridad aportado por la clase *«EventHandler»* PertubogramExecutor. La transferencia de flujo entre ambos threads se realiza a través de la clase *«Synchronizer»* Perturbogram. El thread introducido por PertubogramExecutor se suspende en su método *«Entry»* awaitPerturbogram() hasta que el thread introducido por MetrologyExecutor, invoca el método *«Trigger»* finalized() que activa el thread suspendido cuando finaliza. La interacción del segundo thread con el publicador DDS que no es de tiempo se realiza a través del *«WaitFreeSynchronized»* submitPerturbogram()que garantiza la ejecución del método con exclusión mutua, pero si el mutex está tomado por el thread de no tiempo real, no espera sino que guarda la información para el próximo intento en el que lo invoca.

En la Figura IV.3, se muestra en un diagrama de secuencias las actividades estereotipadas en el framework con constituyen la respuesta MetrologyUpdating.

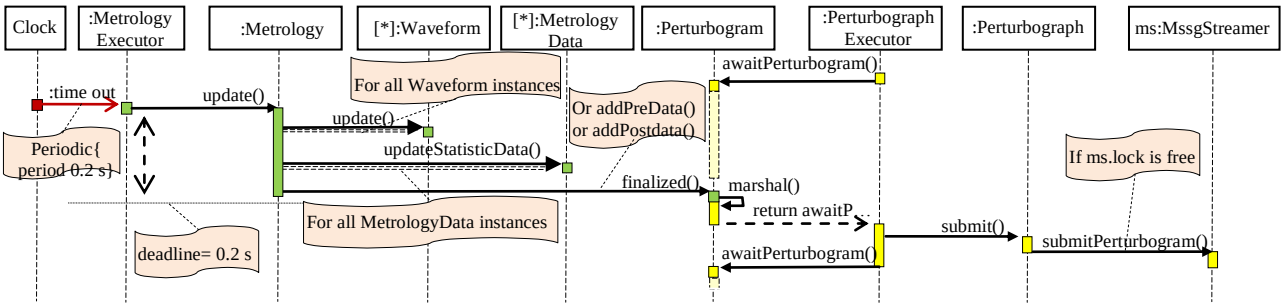


Figura IV.3: Diagrama de secuencias con la MetrologyUpdating rt-response

IV.3 Generación del modelo de tiempo real y análisis de planificabilidad

Una vez implementada la aplicación BURTA en RT-Java y comprobado su correcto funcionamiento, ha de ejecutarse la aplicación RTCode4TimeMntr para instrumentar la primera de forma que el módulo TimeMntr pueda obtener el modelo de tiempo real y realizar un análisis de la planificabilidad de la aplicación.

Continuando el ejemplo de la respuesta MetrologyUpdating que estamos desarrollando, la Tabla IV.1*: Código de la clase PertubographExecutor anotado e instrumentado* contiene el código anotado e instrumentado de la clase PerturbographExecutor, que forma parte de la respuesta anteriormente mencionada.

```
   /* *@OORT_eventHandler */
   private class PerturbographExecutor extends NoHeapRealtimeThread{
      private Perturbogram target;
      private Perturbograph task;

      private int peSrc;
      public PerturbographExecutor(…) {
           super(…);
       }

       public void run(){
          peSrc = tTimeMntr.createThreadMntr(TimeMntr.toString(this));
          PerturbogramData pData = new PerturbogramData();
          TimeMntr.recordTheadTime(peSrc);
          while(!interrupted()){
             target.awaitPerturbogram(pData);
             task.submit(pData);
             TimeMntr.recordTheadTime(peSrc);
           }
        }
}
```

Tabla IV.1: Código de la clase PertubographExecutor anotado e instrumentado

Respondiendo a la etiqueta @OORT_eventHandler, la aplicación ha instrumentado esta clase añadiendo una variable que guarda el índice del monitor del thread creado. De la misma forma, ha instrumentado el método run() de la clase (ya que ésta extiende a NoHeapRealtimeThread) de manera que se cree el monitor de thread en TimeMntr y que comience a contar el tiempo de ejecución de cada ciclo.

Es necesario compilar y ejecutar de forma especial la aplicación de forma que la herramienta que crea el modelo funcione de forma correcta, ya que necesita saber que va a utilizar métodos nativos. De esta forma, la línea de comandos que ha de escribirse para la ejecución de la aplicación es la siguiente:

```
schedtool –a 0x01 –e java-rt –XX:-JITRT –Djava.library.path=. BURTA
```

Con esta instrumentación, se obtiene el modelo de tiempo real MAST que se observa en la Tabla IV.2.

```
<!-- "Perturbograph.PerturbographExecutor_1e51060" Executor model-->
<mast_mdl:Thread Name="Perturbograph.PerturbographExecutor_1e51060.thread"
Scheduler="platformNode.sheduler">
       <mast_mdl:Fixed_Priority_Params Priority="41"/>
</mast_mdl:Thread>
<mast_mdl:Enclosing_Operation
Name="Perturbograph.PerturbographExecutor_1e51060.response"
               Worst_Case_Execution_Time="977.920241E-3"
               Avg_Case_Execution_Time="975.023895E-3"
               Best_Case_Execution_Time="861.168964E-3">
       <mast_mdl:Operation Name="Perturbogram_9ed927.awaitPerturborgram"/>
```

```
        <mast_mdl:Operation Name="Perturbogram_9ed927.marshall"/>
</mast_mdl:Enclosing_Operation>
```

Tabla IV.2: Código MAST del thread PerturbographExecutor

Se puede observar que, a pesar de ser correcta la creación del archivo MAST, está incompleta. Esto es porque es un thread de respuesta de un manejador de eventos, por lo que, su actividad está registrada en otro thread, aquel que ejecuta el trigger que despierta la entry en la que el thread de PerturbographExecutor se ha quedado dormido, es decir, el thread de Metrology, cuyo modelo de tiempo real está representado en la Tabla IV.3.

```
<!-- "Metrology.MetrologyExecutor_1a679b7" Executor model-->
<mast_mdl:Thread Name="Metrology.MetrologyExecutor_1a679b7.thread"
Scheduler="platformNode.sheduler">
        <mast_mdl:Fixed_Priority_Params Priority="47"/>
</mast_mdl:Thread>
<mast_mdl:Enclosing_Operation Name="Metrology.MetrologyExecutor_1a679b7.response"
                Worst_Case_Execution_Time="150.437166E-3"
                Avg_Case_Execution_Time="148.913968E-3"
                Best_Case_Execution_Time="99.146496E-3">
        <mast_mdl:Operation Name="Waveform_1833955.getWaveform"/>
        …
        <mast_mdl:Operation Name="Waveform_1e0be38.getWaveform"/>
        <mast_mdl:Operation Name="Perturbogram_9ed927.addPreData"/>
        <mast_mdl:Operation Name="Perturbogram_9ed927.addPostData"/>
        <mast_mdl:Operation Name="Perturbogram_9ed927.finalized"/>
        <mast_mdl:Operation Name="ConnectionConfiguration_cdfc9c.get"/>
        <mast_mdl:Operation Name="MetrologyData_183f74d.updateStateData"/>
        <mast_mdl:Operation Name="MetrologyData_e102dc.updateStateData"/>
        <mast_mdl:Operation Name="MetrologyData_82c01f.updateStateData"/>
</mast_mdl:Enclosing_Operation>
<mast_mdl:Regular_End_To_End_Flow Name="Metrology.MetrologyExecutor_1a679b7.e2ef">
        <mast_mdl:Periodic_Event Name="trigger" Period="0E0"/>
        <mast_mdl:Internal_Event Name="end">
                <mast_mdl:Hard_Global_Deadline Referenced_Event="trigger"
Deadline="0E0"/>
        </mast_mdl:Internal_Event>
        <mast_mdl:Internal_Event Name="internalEvent0"/>
        <mast_mdl:Step Input_Event="trigger" Output_Event="internalEvent0"
                Step_Schedulable_Resource="Metrology.MetrologyExecutor_1a679b7.thread"
                Step_Operation="Metrology.MetrologyExecutor_1a679b7.response"
                Hold_Schedulable_Resource="NO"/>
        <mast_mdl:Step Input_Event="internalEvent0" Output_Event="end"
        Step_Schedulable_Resource="Perturbograph.PerturbographExecutor_1e51060.thread"
                Step_Operation="Perturbograph.PerturbographExecutor_1e51060.response"
                Hold_Schedulable_Resource="NO"/>
</mast_mdl:Regular_End_To_End_Flow>
```

Tabla IV.3: Código MAST del thread MetrologyExecutor

En rojo queda resaltado el Step o respuesta del thread de Perturbograph que es adjudicado al thread de Metrology.

IV.4 Discusión de los resultados

Al ser un proceso de trabajo con varias fases, se comentarán una a una para discutir los progresos de forma ordenada.

De acuerdo con la aplicación que obtiene el modelo de la plataforma, los resultados obtenidos han sido los esperados, una cabecera de modelo MAST en el que se recogen los datos característicos de la plataforma, como son los tiempos de overhead y de cambio de contexto. Además, podemos confirmar, que los tiempos obtenidos para cada dato, son correctos, ya que se han tomado en un prototipo en la que los tiempos de ejecución eran conocidos.

Una vez desarrollada la aplicación de prueba BURTA y pasada por la aplicación que la instrumenta, los resultados que se obtuvieron también fueron los deseados, más allá de las formas y formato del texto.

Para cada clase instrumentada, el código añadido se correspondía con el esperado (ya que de otra forma la siguiente fase no se podría realizar). El único fallo que se encuentra a la hora de discutir esta aplicación es que, en vista de añadir las nuevas líneas de código, no se ha definido el formato de estas y, por ejemplo el sangrado de las líneas, se ha puesto a grosso modo.

En definitiva, a la vista de los resultados tras realizar el último paso y aplicación, se obtiene un modelo MAST, que era el punto final de este trabajo, correcto y limpio, ordenado y, que al validarlo y ejecutarlo, queda comprobado que, sin tener errores y junto al modelo de plataforma obtenido también por la primera herramienta implementada, obtenemos unos resultados de una aplicación de prueba planificable, que pese a ser de prueba, bien podría aplicarse a cualquier entorno de trabajo en tiempo real.

V Conclusiones y líneas de trabajo futuro

Para abordar la complejidad del software de los sistemas embebidos actuales que tienen especificados requisitos de tiempo real estricto, se necesita utilizar estrategias de diseño orientado a objetos y basado en componentes. Actualmente se disponen de lenguajes como Ada 2005 o RT-Java, que junto a las tecnologías de run-time asociadas hacen posible estas estrategias de programación. Sin embargo, para la operación correcta de un sistema de tiempo real se requiere que el código se ejecute con la configuración de planificabilidad correcta, lo cual requiere caracterizar el comportamiento temporal de las tareas de software cuando se instancia con una determinada configuración y se ejecuta en la plataforma embebida que se vaya a utilizar. En aplicaciones complejas orientadas a objetos, la obtención del modelo de tiempo real que sirva de base para el análisis de planificabilidad que conduzca a la configuración correcta es un trabajo muy pesado que debe ser realizado para cada configuración software que se proponga, para cada plataforma embebida en la que se vaya a instalar y para cada carga de trabajo que requiera el entorno al que se va a aplicar. En este trabajo se ha propuesto un framework de diseño de aplicación muy general que permite automatizar el proceso de configuración de la planificabilidad. Aplicaciones que han sido desarrolladas de acuerdo con este framework pueden ser configuradas en tres fases: Una herramienta proporciona el modelo de plataforma donde se ejecutará la aplicación, otra genera una versión instrumentada de la aplicación, y la ejecución de esta en la plataforma embebida proporciona automáticamente el modelo completo de análisis de planificabilidad. No suele ser aconsejable utilizar la propia plataforma final en el proceso de configuración por dos razones: en primer lugar, la seguridad del entorno sólo permite ejecutar aplicaciones configuradas, y en segundo lugar, el entorno real suele ser pobre en eventos singulares pero posibles, lo que requeriría largos tiempos de ejecución para la configuración. Lo aconsejable es utilizar la misma plataforma embebida pero operando en un entorno físico emulado en el que se controlan la ocurrencia de todos los fenómenos que están especificados para el entorno real. Un entorno de este tipo ya suele utilizarse para el desarrollo y verificación del código. Cuando se utiliza también para la configuración de planificabilidad debe diseñarse con una carga de trabajo que cubra todos los fenómenos de la especificación del entorno de tiempo real y que además cumpla su especificación temporal.

Podemos afirmar que se ha conseguido el objetivo final del trabajo. El hecho de poder obtener un modelo de aplicación, sin posibilidad de fallos humanos o erratas, de forma automática mientras esta se ejecuta es posible al encadenar las diferentes herramientas descritas e implementadas en este proyecto.

El balance del trabajo es positivo puesto que se han conseguido casi todos los objetivos que se habían planteado al comienzo del trabajo. Se han conseguido las tres herramientas que se buscaban: una que obtiene el modelo de plataforma, una que instrumenta el código a través de etiquetas de estereotipos y otra que obtiene el modelo de aplicación, todas funcionando correctamente y obteniendo lo que se pide, pero no de la forma más óptima. Si bien todas ejecutan rápidamente, en la herramienta de la obtención del modelo de aplicación nos referimos a la obtención del archivo MAST una vez ejecutada la aplicación, no ejecutan de

forma óptima y podrían mejorarse los algoritmos de obtención de los resultados o de los modelos.

Otro punto de mejora es a completar varios puntos no tratados de MAST, como por ejemplos, los términos “fork” y “merge” a la hora de hacer que los hilos de respuesta se separen en varios distintos o unifiquen en uno solo, cosa que la herramienta no hace, o, puesto que estamos frente a una orientación a objetos, tampoco se contempla la posibilidad de que se instrumente de forma correcta una clase que hereda de una superclase.

Por último, podría completarse la herramienta de la que se obtiene el modelo de plataforma con, por ejemplo, medidas de acceso a drivers, como se había pensado en un principio, pero que no tenía valor significativo a la hora de realizar este trabajo.

Referencias

[1] Mark Grand : ”Patterns in Java: Catalogue of Reusable Design Patterns Illustrated with UML”. Wiley, 2002.

[2] Doug Lea: “Concurrent Programming in Java:Design Principles and Patterns” Addison-Wesley, 1999.

[3] Oracle: “JavaBean Tutorial”. http://docs.oracle.com/javase/tutorial/javabeans/index.html

[4] Richard Monson-Haefel: ” *Enterprise JavaBeans*” O'Reilly,2004

[5] “JSR-1 Real-Time Specification for Java 1.0.2” <http://download.oracle.com/otndocs/jcp/realtime-1.0.2-mrel-evaloth-JSpec>.

[6] Peter Dibble and Andy Wellings :”JSR-282 Status Report”. JTRES '09 Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. Pp. 179-182, 2009.

[7] Filip Pizlo, Lukasz S Ziarek &J. Vitek:"Real time Java on resource-constrained platforms with Fiji VM". Proceeding JTRES '09, Pages 110-119, September 23-25 , 2009 Madrid, Spain.

[8] ORACLE: Java Real-Time System. <http://www.oracle.com/ technetwork/java/javase/tech/index-jsp-139921.html>

[9] AUSTIN ARMBRUSTER and others:"A Real-Time Java Virtual Machine with Applications in Avionics" ACM Transactions on Embedded Computing Systems, Vol. 7, No. 1, Article 5, Publication date: December 2007.

[10] IEEE TCRTS: “Technical Comitee on Real-Time Systems” http://tcrts.org/education/terminology-and-notation/

[11] MAST: “Modeling and Analysis Suite for Real-Time Applications”. <http:mast.unican.es> .

[12] P. McKenney:” A realtime preemption overview”, August, 2005. <http://lwn.net/Articles/146861/>