ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Evaluación del rendimiento de arquitecturas software para sistemas de teleoperación utilizando UML-MAST.

AUTOR: Francisco García Pérez
DIRECTOR(ES): Juan Ángel Pastor Franco
Bárbara Álvarez Torres

Julio / 2004

Autor	Fancisco García Pérez
E-mail del Autor	pacoy_gp@hotmail.com
Director(es)	Juan Ángel Pastor Franco, Bárbara Álvarez Torres
E-mail del Director	juanangel.pastor@upct.es
Codirector(es)	
Título del PFC	Evaluación del rendimiento de arquitecturas software para sistemas de teleoperación utilizando UML-MAST.
Descriptores	Evaluacion de arquitecturas software

Resúmen

Diseño y análisis de varias arquitecturas software de control de sistemas teleoperados por medio de la herramienta UML-MAST. Una vez completados los modelos, y obtenidos los datos del análisis UML-MAST, se compararán entre los modelos de igual funcionalidad y distinto tipo de arquitectura y se realizará una interpretación de los resultados, así como la realización de un esqueleto de código en Ada95 correspondiente a los modelos realizados.

Titulación	Ingeniería Técnica de Telecomunicaciones.
Intensificación	Telemática
Departamento	TIC
Fecha de Presentación	Julio- 2004

Capítulo 1
Introducción

1.1 Introducción.

En este trabajo se va a realizar un estudio sobre el comportamiento temporal de una serie de arquitecturas software para sistemas teleoperados por medio de la herramienta UML-MAST.

En el proyecto se plantean dos tipos de arquitecturas que se corresponden respectivamente con arquitectura centralizada y una arquitectura distribuida. En la arquitectura centralizada será un controlador el que lleve toda la carga del sistema, mientras en la arquitectura distribuida, esta carga se repartirá en dos controladores al mismo nivel. Se ha ido aumentando paulatinamente la complejidad del sistema, creando diferentes escenarios con una idéntica funcionalidad para ambas arquitecturas, dando así la posibilidad de estudiar los resultados obtenidos, a través de UML-MAST de manera comparativa entre las arquitecturas software. El escenario básico para la evaluación se basa en un control “On-Line” de los movimientos de un robot, en el cual el sistema va verificando la posibilidad de que haya colisión.

El sistema teleoperado que se plantea va a estar compuesto, básicamente, por una interfaz de usuario, un controlador de alto nivel, un controlador de bajo nivel, un servidor de cinemática y una representación física del robot.

En la figura 1 y la figura 2, se pueden ven ver los esquemas básicos de ambas arquitecturas software.

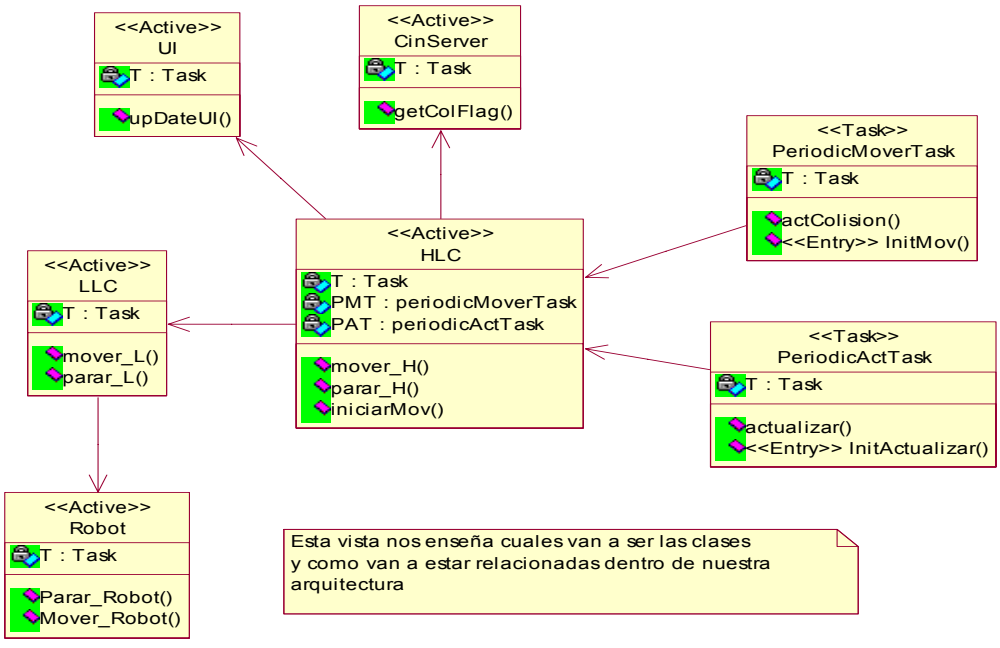


Figura 1 Arquitectura software centralizada

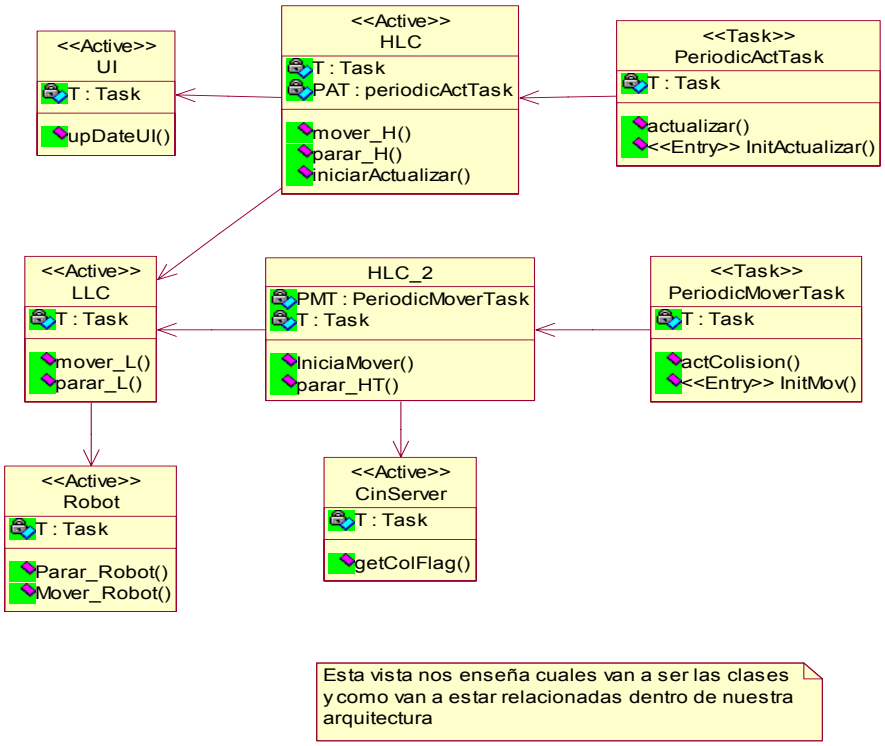


Figura 2 Arquitectura software distribuida

1.2 Objetivos del proyecto.

El objetivo de este trabajo es evaluar y comparar varias arquitecturas de control de sistemas teleoperados en función de su comportamiento temporal, así como la realización de un esqueleto de las mismas en Ada95.

La creación de los esqueletos de código deja abierto el trabajo para que posteriormente pueda continuarse el estudio, utilizando un sistema operativo de tiempo real. A partir del esqueleto se puedan comparar los resultados obtenidos en el análisis de UML-MAST, con los que se obtengan durante la ejecución del código.

Para poder llevar a cabo estos objetivos se ha partido de dos modelos muy simples, aumentando su complejidad poco a poco.

Una vez completados los modelos, y obtenidos los datos del análisis UML-MAST, es posible comparar modelos de igual funcionalidad y distinto tipo de arquitectura y obtener alguna interpretación, de los datos obtenidos que permita elegir entre una de las dos arquitecturas propuestas en el trabajo.

1.3 Herramientas para el trabajo.

Para poder llegar a los objetivos planteados para este trabajo es necesaria la utilización, de la herramienta UML-MAST y el lenguaje Ada95.

MAST es una herramienta desarrollada por el grupo de computadores y tiempo real de la Universidad de Cantabria. UML-MAST (Modeling and Analysis Suite for Rea Time Aplication) es una metodología y un conjunto de herramientas gráficas desarrolladas para modelar y analizar sistemas de tiempo real, basadas en la notación UML. Esta herramienta esta preparada para ser utilizada sobre una plataforma *CASE*, más en concreto sobre *Rational Rose 2000*. En el Capitulo 3 se dedican varios apartados a explicar UML-MAST.

Como se ha mencionado anteriormente una vez realizados los diseños en MAST, serán implementados todos los sistemas en Ada95. A la hora de elegir un lenguaje se ha optado por Ada 95, ya que se trata de un lenguaje con soporte para la concurrencia diseñado específicamente para sistemas de tiempo real. Ada es un lenguaje orientado a objetos, que desciende de Pascal, esta estructurado y muy fuertemente tipado, esta pensado para construir sistemas grandes y cambiantes. Además tienen un núcleo común para todas las implementaciones y anexos específicos para programación de sistemas de tiempo real, sistemas distribuidos, fiabilidad y seguridad. Los anexos definidos no añaden sintaxis ni vocabulario, sino que definen paquetes de biblioteca y mecanismos de implementación.

Ada es utilizado en un gran numero de sistemas como por ejemplo los ordenadores de control de vuelo de casi todos los aviones modernos y sistemas de control de tráfico aéreo, ordenadores de bastantes trenes de alta velocidad y trenes suburbanos, aplicaciones bancarias, satélites de comunicaciones y de navegación y acerías, robótica industrial, electrónica médica y telecomunicaciones.

Capítulo 2
Estado de la Técnica

2.1 Introducción

Para una mayor compresión del trabajo realizado es necesario dar una breve explicación de las técnicas, conocimientos y conceptos utilizados para llevar a cabo la realización de este proyecto. A continuación se introduce el concepto de sistemas de tiempo real y el algoritmo de planificación RMA (*Rate Monotonic Analysis*), se define el concepto de Arquitectura del Software, la notación UML, y la herramienta UML-MAST utilizada para el diseño de sistemas en tiempo real.

2.2 Sistemas de tiempo real.

2.2.1 Definición.

Existen muchas definiciones para lo que es un sistema de tiempo real. De todas ellas podríamos destacar una como la definición canónica de sistema de tiempo real.

*"**Un sistema de tiempo real es aquel en el que para que las operaciones computacionales estén correctas no depende sólo de que la lógica e implementación de los programas computacionales sean correctos, sino también en el tiempo en el que dicha operación entregó su resultado. Si las restricciones de tiempo no son respetadas el sistema se dice que ha fallado."***

Donald Gillies.

A esta definición otros han añadido:

"Por lo tanto, es esencial que las restricciones de tiempo en los sistemas sean cumplidas. El garantizar el comportamiento en el tiempo requerido necesita que el sistema sea predecible. Es también deseable que el sistema obtenga un alto grado de utilización a la vez que cumple con los requisitos de tiempo."

Contrariamente a lo que se podría pensar, su presencia se halla actualmente muy difundida. Las aplicaciones de tiempo real son muy variadas y continuamente surgen nuevos campos de utilización para las mismas, siendo los más comunes los asociados a los sistemas de telecomunicaciones, la robótica, multimedia, control de procesos industriales y sistemas espaciales, entre otros. Pensando en nuestra vida cotidiana, son sistemas de tiempo real los que ayudan a volar a los aviones, a rodar a los trenes, los que controlan el motor o los frenos de nuestro automóvil, etc.

Como ejemplo de un sistema de tiempo real podemos poner el sistema básico (ampliado poco a poco la complejidad del mismo a lo largo del estudio) con el cual se ha trabajado en este proyecto. Se trata de un robot que tiene que realizar varias tareas, entre las cuales existe una que se encarga de controlar la posibilidad de colisión durante su movimiento. Esto se realizará por medio de la activación de un flag, que se comprobará de manera periódica. Si la comprobación de dicho flag se

retrasase no habría tiempo suficiente para ejecutar una rutina para esquivar el obstáculo o parar el robot.

Todos estos sistemas tienen algo en común: están íntimamente ligados a otros sistemas con los cuales se relacionan continuamente intercambiando diversos tipos de información y efectuando sobre ellos funciones de control. En otras palabras, podemos considerar la existencia de un único sistema formado por otros dos subsistemas, a saber: el sistema controlado o entorno y el sistema de control.

El sistema controlado cuenta con una dinámica propia, es decir, con cierto esquema que describe no sólo los distintos eventos que se producen en él, sino también la forma o secuencia según la cual éstos se irán produciendo y las relaciones existentes entre cada uno de ellos. Este esquema se halla normalmente caracterizado por intervalos de tiempo más o menos bien definidos entre los eventos.

Los sistemas de control son de naturaleza más flexible pudiéndose implementar diversos esquemas sobre diversos soportes, los cuales pueden ir desde un *hardware* microcontrolador dedicado hasta un ordenador personal de aplicación general. Nuevamente, el aspecto a tener en cuenta es que no solamente importa el correcto funcionamiento lógico, en términos de proporcionar un valor correcto a partir del estado de las variables de entrada, sino además que dichos resultados sean entregados "a tiempo”.

Como resulta evidente, la dinámica de los sistemas controlados no puede ser sustancialmente modificada. Consecuentemente, otras alternativas deben ser implementadas en el sistema de control, con miras a satisfacer todos los requisitos tanto funcionales como temporales y evitar que el comportamiento global del sistema evolucione hacia valores incorrectos o indeseados.

De más está decir que la presencia de requisitos temporales hace que la construcción de los sistemas de tiempo real sea mucho más difícil y complicada, principalmente porque la mayoría de los métodos y herramientas utilizados para la construcción de sistemas convencionales no contemplan tales restricciones, características en los sistemas de tiempo real.

2.2.2 Características de los sistemas de tiempo real.

Los eventos, mencionados en párrafos anteriores y que constituyen la dinámica del sistema, desencadenan, al momento de verificarse, una serie de acciones por parte del sistema de control para la satisfacción de sus respectivos requisitos temporales. Podemos, por lo tanto, decir que cada uno de esos eventos tiene asociada una cierta rutina, tarea o actividad.

El comportamiento temporal de la totalidad del sistema puede describirse en términos de ciertas características ligadas a las mencionadas tareas. Estas características comprenden:

- El *esquema de activación* de la tarea: que básicamente describe cuando se debe ejecutar la tarea. Esta activación puede ser periódica o aperiódica.

- El *plazo de ejecución* de la actividad: indica el intervalo de tiempo máximo para la culminación de la ejecución de la tarea.

Las mencionadas características proporcionan un criterio para la clasificación de los sistemas de tiempo real. En particular, se habla de sistemas de tiempo real *acríticos* (*soft real time systems*) cuando la no culminación, dentro del plazo, de la ejecución de una tarea no produce efectos que vayan más allá que un deterioro en la calidad de los resultados proporcionados. Por el contrario, en los sistemas de tiempo

real *críticos* (*hard real time systems*), la no satisfacción estricta de los correspondientes plazos ocasiona que todo el sistema se colapse, pudiendo incluso producirse, en el caso de sistemas de seguridad *crítica* (*safety critical systems*), pérdidas de vidas humanas además de las puramente materiales.

Como ya hemos establecido, la construcción de este tipo de sistemas se ve dificultada por la inclusión del tiempo en cada una de las instancias de desarrollo y por la existencia de ciertas características en el entorno con el cual deberá interactuar el sistema de control. Estas características comprenden:

- **Concurrencia:** Normalmente los eventos en el entorno se dan en paralelo, razón por la cual el sistema debe responder a ellos mediante actividades ejecutándose en paralelo.

- **Fiabilidad y seguridad:** El sistema debe ante todo producir los resultados correctos. Dependiendo de lo crítico sea su desempeño y de los elementos con los cuales debe interactuar, vidas humanas por ejemplo, se le exigirá un mayor o menor grado de fiabilidad.

- **Determinismo:** El determinismo es una cualidad clave en los sistemas de tiempo real. Es la capacidad de determinar con una alta probabilidad, cuanto es el tiempo que se toma una tarea desde que comienza a ejecutarse. Esto es importante porque los sistemas de tiempo real necesitan que ciertas tareas se ejecuten antes de que otras se puedan iniciar.

- **Usuarios controladores**: En estos sistemas, el usuario (los procesos que corren en el sistema) tiene un control mucho más amplio del sistema.

    - El proceso es capaz de especificar su prioridad
    - El proceso es capaz de especificar el manejo de memoria que requiere (que parte estará en caché y que parte en memoria swap y que algoritmos de memoria swap usar)
    - El proceso especifica que derechos tiene sobre el sistema.

  Esto aunque parece anárquico no lo es, debido a que los sistemas de tiempo real usan TIPOS de procesos que ya incluyen estas características, y usualmente estos TIPOS de procesos son mencionados como **requisitos.** Un ejemplo es el siguiente:

  "Los procesos de mantenimiento no deberán exceder el 3% de la capacidad del procesador, a menos que en el momento que sean ejecutados el sistema se encuentre en la ventana de tiempo de menor uso".

- **Confiabilidad:** La confiabilidad en un sistema de tiempo real es otra característica clave. El sistema no debe de ser solamente libre de fallo, la calidad del servicio que presta no debe de degradarse más allá de un límite determinado. El sistema debe de seguir en funcionamiento a pesar de catástrofes, o averías mecánicas. Usualmente una degradación en el servicio en un sistema de tiempo real lleva a consecuencias catastróficas.

- **Operación a prueba de fallos (Fail soft operation):** El sistema debe de fallar de manera que: cuando ocurra un fallo, el sistema preserve la mayor parte de los datos y sus capacidades en la máxima medida posible. El sistema debe ser

estable, en el sentido de que siempre cumplirá con las tareas más críticas y de más alta prioridad.

2.2.3 Etapas de desarrollo de un sistema de tiempo real.

Las diferentes etapas de desarrollo de un sistema de tiempo real comprenden los siguientes pasos:

- **Especificación:** Antes que nada, es esencial contar con las herramientas y lenguajes apropiados para la descripción del sistema en términos funcionales y temporales. Como ya hemos mencionado, las herramientas, lenguajes y métodos convencionales no contemplan la posibilidad de introducir al tiempo como uno de los parámetros en el diseño por lo que resultan inapropiados, aunque algunas variantes han sido introducidas en algunos de estos métodos para cubrir dicha carencia.

- **Diseño:** Se debe contar con los elementos necesarios para poder describir todos los aspectos del sistema de tiempo real. Estos aspectos incluyen las distintas tareas que lo forman y sus relaciones, los recursos compartidos y los requisitos de tiempo. Actualmente, la tendencia es la de utilizar métodos de diseño orientados a objetos. Los más relevantes incluyen OCTOPUS y HRT-ROOM.

- **Realización:** Se han aprobado dos normas internacionales, las cuales definen el lenguaje de programación Ada y las interfaces de sistema operativo POSIX. Ambas están orientadas al desarrollo basado en esquemas de prioridades estáticas y definen aspectos relativos a la comunicación entre tareas, planificación de tareas basada en prioridades con expulsión (*preemption*), funciones de temporización adecuados, etc.

  En la mayoría de los casos permiten transportar el código fuente sin modificaciones entre plataformas conformes con las normas.

- **Planificación**: La planificación de los recursos es, sin lugar a dudas, uno de los problemas principales en la construcción de los sistemas de tiempo real. A los primitivos esquemas de planificación síncrona, con las tareas activándose de forma sincronizada según un reloj, de manera cíclica y según un orden prefijado, le siguieron los esquemas más modernos de planificación basados en prioridades.

  Estos últimos, se fundamentan en una asignación estática de prioridades a las distintas tareas y en un despachador con expulsión, que se encarga de ejecutar la tarea de más alta prioridad en el instante actual. Diversos estudios han sido desarrollados en los últimos años, todos ellos orientados a determinar un esquema para la asignación de las prioridades que asegure, para cada una de las tareas, un acceso oportuno al recurso compartido de modo que se satisfagan sus plazos de ejecución (*execution deadlines*).

  Entre los diversos enfoques concebidos podemos mencionar las políticas de RMS (*rate monotonic scheduling*) o DMS (*deadline monotonic scheduling)*. El primero de ellos considera una asignación de prioridades según una función monotónica de la frecuencia de las tareas, en otras palabras cuanto más corto es su período, más alta será la prioridad de la tarea. En el segundo, las prioridades se derivan según una función monotónica de sus vencimientos, es

decir, cuanto menor sea el tiempo de vencimiento de la tarea, mayor será su prioridad.

En su forma primitiva, las políticas de RMS y DMS, al igual que otros esquemas, se basan en una serie de suposiciones relativas a la naturaleza de las tareas en conflicto y sus relaciones, las cuales pueden no verificarse (estrictamente) en la realidad. Esta discrepancia hace necesarias adaptaciones o reformulaciones de las políticas, cuya complejidad no justifica su aplicación sobre todo cuando los sistemas en consideración no son muy grandes.

Aunque estas políticas ayudan a arrojar luz sobre el problema en un plano teórico, no podemos dejar de mencionar que se tratan de modelos que pueden no siempre sostenerse en la práctica, dependiendo de las características del proyecto particular.

- **Validación:** La etapa final de desarrollo de un proyecto exige someter la primera versión del producto a una serie organizada de pruebas con miras a verificar su correcto funcionamiento bajo todas las condiciones posibles de operación.

  De acuerdo con lo puntualizado en los últimos párrafos, podemos afirmar que a medida que aumentan los campos de aplicación y la complejidad de los sistemas, el análisis de todas las particularidades y requisitos temporales específicos de los mismos se vuelve extremadamente complicado. Este hecho hace necesaria la existencia de técnicas formalmente definidas y herramientas de desarrollo que permitan predecir lo más precisamente posible el comportamiento de esos sistemas, para poder determinar la posibilidad de una implementación según los esquemas de tiempo real.

  En esta línea, se debe puntualizar la carencia de algoritmos de validación que permitan la evaluación de sistemas de tiempo real, los cuales se presentan como esencialmente dinámicos, es decir imposibles de analizar con modelos y suposiciones estáticas.

Finalmente, una vez presentados los aspectos principales de los sistemas de tiempo real y sus implicancias, se considera oportuno reafirmar dos puntos esenciales:

- No es necesario que un sistema procese datos en intervalos del orden de los microsegundos para ser considerado de tiempo real; simplemente debe tener tiempos de respuesta acotados y por lo tanto predecibles.

- Los problemas no se solucionan simplemente haciendo rápido al sistema. Aunque esto ayude a cumplir los requisitos temporales, la cualidad principalmente deseada es que el sistema sea *determinístico*, es decir que su comportamiento sea el adecuado en cualquier circunstancia, incluso bajo condiciones de sobrecarga o, más formalmente, que "para cada estado posible y para cada conjunto de entradas, un conjunto único de salidas y un estado siguiente para el sistema, puedan ser determinados."

2.2.4 Repercusión de las tecnologías en tiempo real.

Considérese como ejemplo de la importancia de esta tecnología a los sistemas de control de tráfico aéreo, los cuales se presentan como sistemas particularmente

críticos, tanto en términos económicos como de seguridad. En un sistema de control del tráfico aéreo es característico el manejo continuo de grandes cantidades de datos, pero a diferencia de otros sistemas que también involucran este tipo de gestiones, los datos a ser analizados se encuentran en constante cambio. Esta particularidad, convierte a esos datos en elementos extremadamente valiosos por periodos muy cortos de tiempo, lo cual inequívocamente indica requisitos temporales estrictos que no pueden quedar insatisfechos dadas la naturaleza del sistema y el entorno con el cual opera.

Resulta claro que un sistema de este tipo es tan extenso y complejo que nuevos enfoques deben ser concebidos con miras a mejorar aún más los niveles de seguridad del mismo, así como reducir los costos del sistema y su mantenimiento, previendo siempre su continua evolución en lo que a tamaño y complejidad se refiere.

2.2.5 Evolución y crecimiento en sistemas de tiempo real.

La infraestructura de tiempo real actualmente existente, introduce formidables barreras al mejoramiento continuo de los procesos controlados. Comúnmente se observan plantas industriales montadas según una amplia y variada gama de equipamientos, cada uno de ellos provenientes de proveedores distintos y consecuentemente caracterizados por diferentes interfaces de programación, estructuras de datos, protocolos de comunicación y características temporales. En situaciones como ésta, la instalación de nuevos componentes y sobre todo, su integración efectiva al sistema no puede efectuarse sin grandes esfuerzos en forma de pruebas extensivas, que aseguren la satisfacción de todos los requisitos temporales y funcionales. Modificaciones en otras instancias del sistema, con miras a optimizar el desempeño de las mismas, son igualmente difíciles de realizar debido a la imposibilidad de determinar con certeza sus implicaciones sobre las características temporales de la totalidad del sistema, como consecuencia de la heterogeneidad anteriormente comentada.

Consecuentemente, una nueva infraestructura de tiempo real debería incluir:

- El uso extensivo de componentes basados en estándares, universalmente aceptados, siempre que sea posible (buses, sistemas operativos, redes y protocolos de comunicación).
- Un entorno conveniente y seguro para la personalización, optimización, reconfiguración, desarrollo en línea y pruebas de las distintas instancias del sistema, así como para la integración de nuevas tecnologías y productos.

Tales objetivos no serán alcanzados sin la elaboración de esquemas para la planificación y manejo de recursos en sistemas de demostrada *predictibilidad*, o como mencionamos anteriormente, de demostrado *determinismo*. Las propiedades de predictibilidad deben comprender tanto los aspectos funcionales como los de naturaleza temporal y de tolerancia a fallos.

Para concluir este punto, también es deseable el desarrollo de esquemas para la construcción de sistemas mayores a partir de subsistemas predecibles y que aseguren que las propiedades de predictibilidad de estos nuevos sistemas se deriven también de las correspondientes en los subsistemas componentes.

2.2.6 Análisis de los niveles de desempeño alcanzables.

En la mayoría de los casos, el análisis de los sistemas de tiempo real se basa fundamentalmente en simulaciones y pruebas exhaustivas. Contrariamente a esto, lo que se requiere es *una ciencia del desempeño* que permita un análisis más formal de este tipo de sistemas, los cuales se presentan como esencialmente dinámicos, cuando las condiciones del entorno no son totalmente predecibles o controlables.

Cada esquema de planificación o algoritmo de validación actualmente disponible, se basa en algún tipo de modelo para los sistemas analizados. Las conclusiones a partir de ellos obtenidas, serán válidas siempre que las suposiciones realizadas encuentren sustento en los sistemas reales.

El hecho de que algunas de las características principales de todo sistema (entre las que podemos mencionar a las velocidades del procesador, tiempos de acceso a memoria u otras dependientes de la particular plataforma escogida) no sean mensurables hasta el momento de la implementación, limita la predictibilidad del mismo, haciendo que la elaboración de modelos precisos sea una tarea imposible.

Consecuentemente, son necesarios algoritmos robustos cuyas conclusiones permanezcan válidas a pesar de que alguno de los supuestos asumidos en el correspondiente modelo no se satisfaga completamente.

Nuevos desarrollos en el área son necesarios, principalmente debido al hecho de que los enfoques actualmente disponibles no se sostienen (en otras palabras, concluyen la no factibilidad de la realización del esquema analizado aunque sea posible) para el caso de sistemas formados por procesos esporádicos, es decir procesos en los cuales los tiempos de ejecución y requisitos de recursos se presentan como esencialmente variables.

En segundo lugar, gran parte de los algoritmos actualmente existentes se basan en modelos con requisitos temporales determinísticos, lo cual los hace inaplicables a sistemas en los cuales dichos parámetros se presentan como esencialmente probabilísticos.

Finalmente, la mayoría de los algoritmos existentes se orientan a sistemas estáticamente configurados, en otras palabras, sistemas en los cuales las aplicaciones son particionadas y los procesadores y recursos son estáticamente asignados a esas particiones.

2.2.7 Confiabilidad y verificación formal.

A medida que las computadoras se vuelven una parte esencial dentro de sistemas más complejos, se hace necesario incrementar la confiabilidad de los sistemas computacionales. Existe una gran variedad de técnicas desarrolladas y utilizadas, abarcando desde análisis estáticos basados en métodos formales y teorías de planificación hasta análisis dinámicos basados en pruebas y observaciones en tiempo de ejecución. A pesar de que las mencionadas técnicas se presentan efectivas para el caso de sistemas pequeños, lo que realmente importa es contar con un marco o estructura que permita la aplicación de las mismas a sistemas más complejos.

En cuanto a su construcción, una estructura de este tipo puede ser desarrollada mediante extensiones a los formalismos actualmente existentes. Entre los mayoritariamente utilizados, podemos mencionar las máquinas de estado, las redes de

Petri o el álgebra de procesos. Cualquiera de estos formalismos podría ser integrado en las técnicas de planificación, monitorización y testeo con miras a construir el mencionado marco.

2.2.8 Lenguajes de Programación de Tiempo Real.

La mayor parte de los lenguajes de programación existentes en la actualidad sencillamente no son apropiados para el desarrollo de sistema de tiempo real. Sus limitaciones se derivan principalmente de la incapacidad para especificar requisitos temporales, con la consecuente imposibilidad de verificación de la factibilidad de ejecución del código generado, en términos de los mencionados requisitos.

Independientemente de la variedad de los enfoques aplicables, resulta claro que todo lenguaje de programación de tiempo real debe proporcionar un soporte caracterizado ante todo por un determinismo temporal. Con miras a alcanzar dicho objetivo, se debe:

- Eliminar todas aquellas construcciones caracterizadas por tiempos de ejecución indeterminados, como ser los ciclos o "loops" infinitos.
- Construir los lenguajes en conjunción con el sistema operativo.

2.3 RMA.

A continuación se va a explicar el algoritmo RMA que ha sido utilizado durante el diseño de los sistemas que se van a exponer en este trabajo, con la intención de poder garantizar los plazos de respuesta de los sistemas, siempre y cuando esto sea posible.

Uno de los problemas de los sistemas de tiempo real es la planificación del tiempo de ejecución de los procesos de forma que se garanticen los plazos de respuesta especificados. Una forma de conseguir este objetivo está basada en la asignación de prioridades a los procesos, de tal forma que los que se activan con mayor frecuencia tengan mayor prioridad (planificación con prioridades al más frecuente, *rate monotonic priority scheduling*). El interés de este tipo de planificación estriba en su realización relativamente sencilla, en particular cuando se utiliza Ada como lenguaje de implementación, y en sus propiedades de estabilidad frente a sobrecargas del sistema.

Este problema de la planificación se ha estudiado en profundidad en el marco del diseño de sistemas operativos, en el que se han definido algoritmos que permiten la ejecución de un conjunto de procesos de forma satisfactoria (turno rotatorio, primero el más corto, prioridades, etc.). Estos algoritmos, sin embargo, no se pueden utilizar para ejecutar tareas de tiempo real, ya que sólo permiten asegurar el cumplimiento de requisitos de tipo estadístico sobre los tiempos de ejecución (tiempo de respuesta medio, flujo de trabajos, etc.).

En los sistemas de tiempo real acríticos se suelen emplear métodos de planificación basados en el concepto de prioridad. La prioridad es un atributo de cada tarea, que está ligada a su importancia relativa en el sistema. La regla básica consiste en ejecutar siempre la tarea más prioritaria de las que estén preparadas para ejecutarse. Hay distintas variantes, según se admita o no que se expulse a la tarea que se está ejecutando cuando se active una más prioritaria, se pueda cambiar la prioridad de una tarea durante su ejecución, etc.

Los métodos de planificación basados en prioridades son fáciles de realizar y están basados en un concepto muy sencillo e intuitivo, lo que explica su popularidad. Sin embargo, el indeterminismo inherente al modelo de procesos concurrentes y la asignación de prioridades a las tareas a partir de conceptos poco sistemáticos, como su "importancia" relativa, han impedido en muchos casos que se puedan garantizar los requisitos temporales de las tareas de tiempo real de forma completamente determinista, lo que ha hecho que durante algún tiempo se haya evitado su utilización en sistemas críticos. En estos últimos se suele utilizar más bien la técnica del *ejecutivo cíclico* para planificar la ejecución de las tareas. Un ejecutivo cíclico activa un conjunto de tareas según un esquema fijo que se repite periódicamente, sincronizado con un reloj [Locke92]. El esquema de ejecución de las tareas se prepara manualmente cuando se diseña el sistema. Esto puede ser muy difícil de llevar a cabo, y a menudo exige modificar el comportamiento temporal de las tareas, o dividir sus actividades en segmentos de corta duración. Sin embargo, los ejecutivos cíclicos permiten garantizar de forma totalmente determinista el comportamiento temporal de las tareas de tiempo real, y por ello se han utilizado ampliamente en sistemas en los que el cumplimiento de los requisitos temporales es crítico.

Aunque el uso de un ejecutivo cíclico permite garantizar el comportamiento temporal de un sistema de tiempo real, los inconvenientes causados por su falta de flexibilidad y por la dificultad de planificar la ejecución manualmente hacen que esta técnica resulte inadecuada para sistemas de gran dimensión o con conjuntos dinámicos de tareas. En cualquier caso, este método obliga a razonar a un nivel de abstracción muy bajo sobre la forma de ejecutar las actividades de los sistemas de tiempo real, por lo que una organización basada en procesos concurrentes lógicamente independientes es más adecuada. Como ya hemos visto, sin embargo, los métodos de planificación de tareas basados en este modelo, y en particular los basados en prioridades, presentan algunos inconvenientes que es necesario resolver para emplearlos de forma efectiva en el diseño de sistemas de tiempo real crítico. La aproximación que se presenta en este tema está basada en el principio de planificación con prioridad al más frecuente (*rate monotonic priority scheduling),* introducido por Liu y Layland [Liu&73], y extendido por Sha, Goodenough y otros [Sha&90] [Goodenough&88].

Otros métodos de planificación están basados directamente en los requisitos temporales de las tareas. El más conocido es el consistente en ejecutar en cada momento la tarea que debe acabar antes (primero el más urgente, *earliest deadline first*) [Baker91]. Aunque este método tiene propiedades muy interesantes, sobre todo cuando hay muchas tareas aperiódicas, la dificultad de su puesta en práctica y su falta de estabilidad en caso de sobrecargas transitorias limitan su campo de aplicación.

2.3.1 Atributos y requisitos temporales.

Para cada tarea de tiempo real, τ_i, $i = 1,.....,n,$ podemos definir un conjunto de atributos temporales, que caracterizan su comportamiento en cada una de sus ejecuciones (figura 1):

- Período de ejecución, T_i. Sólo está definido si la tarea es periódica.
- Tiempo de la k-ésima activación, $a_i(k)$. Si la tarea es periódica, entonces:

$$a_i(k) = k * T_i + a_i(0) \tag{1}$$

donde $a_i(0)$ es un desfase o retardo inicial, que puede ser nulo.

Si la tarea es esporádica, los instantes de activación toman valores arbitrarios, que coinciden con las sucesivas ocurrencias del evento asociado a la tarea. Sin embargo, para poder garantizar el comportamiento temporal del sistema, es necesario suponer que hay una separación mínima, S_i, entre dos activaciones sucesivas de una tarea, es decir:

$$a_i(k) \geq a_i(k-1) + S_i \tag{2}$$

- Retardo, $\delta_i(k)$. Cuando una tarea se activa pasa a estar preparada para ejecutarse, pero esto no quiere decir que se ejecute inmediatamente, ya que el procesador puede estar asignado a otra tarea más prioritaria. El retardo es el tiempo que transcurre hasta que la tarea comienza efectivamente a ejecutarse.
- Tiempo de cómputo, $C_i(k)$. Es el tiempo de procesador que consume la tarea cuando se ejecuta. Con objeto de asegurar que el comportamiento del sistema es correcto en todos los casos, se suele considerar el valor máximo de este tiempo, C_i.
- Tiempo de finalización, $f_i(k)$. Es el tiempo en el que termina la ejecución de la tarea. Obviamente,

$$f_i(k) \geq a_i(k) + C_i(k) \tag{3}$$

- Tiempo de respuesta, $R_i(k)$. Es el intervalo transcurrido entre la activación y la finalización de la tarea:

$$R_i(k) = f_i(k) - a_i(k) \tag{4}$$

Los requisitos temporales de las tareas de tiempo real se definen a partir de los atributos temporales de las mismas. Aunque, en principio, se pueden especificar restricciones en los valores de todos los atributos, el requisito más común es el de tiempo de respuesta máximo o plazo de respuesta, D_i . La condición que se exige al sistema es que, para cada tarea τ_i, se cumpla:

$$\forall k, R_i(k) \leq D_i \tag{5}$$

El tiempo límite, $l_i(k)$ marca el final del plazo de respuesta:

$$l_i(k) = a_i(k) + D_i \tag{6}$$

La condición (3) puede, por tanto, expresarse también así:

$$\forall k, f_i(k) \leq l_i(k) \tag{7}$$

En general, caracterizaremos los requisitos temporales de una tarea periódica mediante una terna (T_i , C_i , D_i). Y los de una tarea esporádica, de forma similar: (S_i , C_i , D_i). Estos parámetros deben cumplir la relación:

$$0 \leq C_i \leq D_i \leq T_i \tag{8}$$

o bien:

$$0 \leq C_i \leq D_i \leq S_i \tag{9}$$

para que sea posible ejecutar la tarea dentro del plazo especificado.

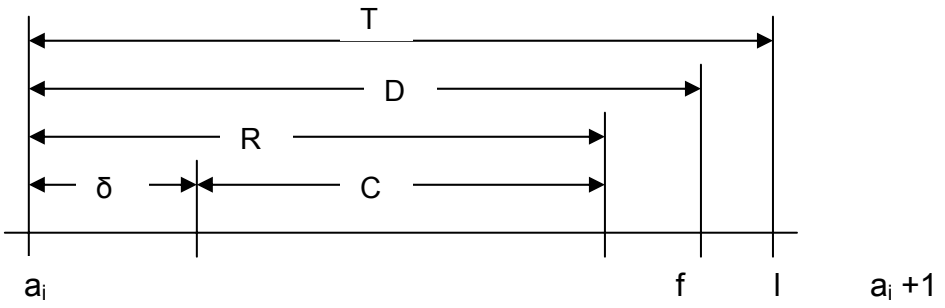


Figura 1. Atributos y requisitos temporales.

2.3.2 Modelo de tareas.

Es necesario imponer restricciones a la estructura que un programa concurrente de tiempo real pueda tener. Vamos a partir de un modelo muy simple que posteriormente generalizaremos:

- El conjunto de tareas es estático, es decir, el número de tareas es fijo y se conoce en tiempo de ejecución.
- Las tareas son periódicas, es decir, se activan a intervalos regulares.
- Las tareas son independientes entre sí, es decir no hay mecanismos de comunicación ni sincronización entre ellas.
- Los plazos de respuesta de todas las tareas son iguales a sus períodos respectivos.
- El tiempo de ejecución máximo de cada tarea es conocido.
- Las operaciones del núcleo de multiprogramación son instantáneas, es decir, no se emplea tiempo para expulsar una tarea y asignar el procesador a una tarea nueva.

Como veremos a continuación, siempre que se cumplan estas hipótesis se puede utilizar el método de planificación basado en prioridades, asignando prioridades mayores a las tareas más frecuentes.

La planificación del modelo anterior lo vamos a caracterizar por los siguientes parámetros:

N Número de tareas

T Período de activación

C Tiempo de ejecución máximo

D Plazo de respuesta

R Tiempo de respuesta máximo

P Prioridad

Para todas las tareas τ_i:

$$C_i \leq D_i = T_i \tag{10}$$

Se trata de asegurar que:

$$R_i \leq D_i \tag{11}$$

2.3.3 Utilización del procesador.

Un método de planificación tiene dos aspectos importantes:

- Un algoritmo de planificación que determina el orden de acceso de las tareas a los recursos del sistema, en particular al procesador.
- Un método de análisis que permite calcular el comportamiento temporal del sistema. En general, se estudia el peor comportamiento posible para ver si se cumplen los requisitos temporales.

Uno de los métodos de análisis de planificabilidad que se va a describir está basado en el uso del factor de utilización del procesador. La fracción de tiempo de procesador consumida por una tarea periódica τ_i es:

$$U_i = \frac{C_i}{T_i} \tag{12}$$

Análogamente, la utilización de una tarea esporádica es**:**

$$U_i = \frac{C_i}{S_i} \tag{13}$$

Y la utilización máxima del procesador por el conjunto de *N* tareas de tiempo real es, por tanto,

$$U = \sum_{i=1}^{N} U_i \tag{14}$$

Si no existieran restricciones temporales podríamos obtener una planificación admisible para cualquier conjunto de procesos que verificase la condición $U \leq 1$. Sin embargo, la exigencia de respetar los tiempos de respuesta especificados puede limitar el factor de utilización a un valor menor que la unidad.

2.3.4 Planificación con Prioridades Estáticas.

El fundamento de la planificación con prioridades estáticas consiste en asignar a cada tarea τ_i una prioridad fija, P_i, ejecutándose en cada momento la tarea de mayor prioridad de entre todas las que estén activas (preparadas para ejecutar o

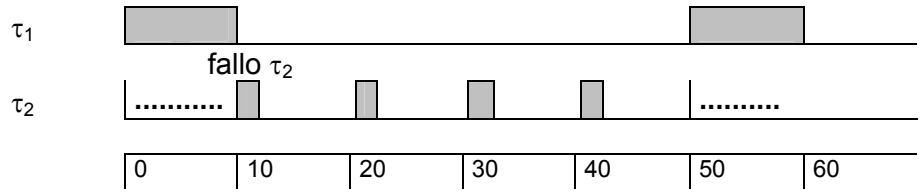
ejecutándose). Cuando hay varias tareas activas con la misma prioridad, se selecciona una de ellas de acuerdo con algún otro criterio (orden de activación, turno rotatorio, etc.). Supondremos que la planificación es expulsiva (*preemptive*), es decir que si se activa una tarea con mayor prioridad que la que está ejecutándose en un momento dado, expulsa a ésta del procesador, pasando a ejecutarse inmediatamente la más prioritaria.[1]

Ejemplo 1: Sea un sistema con dos tareas, τ_1 y τ_2, con los requisitos indicados en la tabla 1.

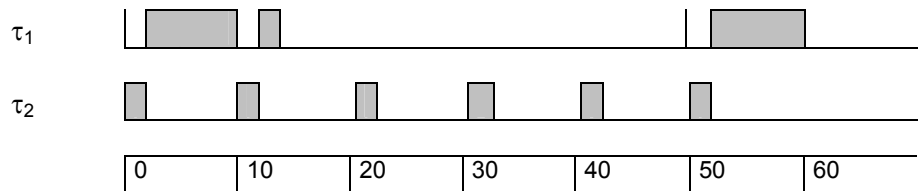
τ	*T*	*C*	*D*
τ_1	50	10	50
τ_2	10	2	10

Tabla 1. Requisitos para tareas del ejemplo 1.

Si asignamos mayor prioridad a la tarea τ_1, supuesta más importante, se produce un fallo en el tiempo de respuesta de τ_2 (Cronograma 2a). Suponemos que las dos tareas se activan a la vez. No obstante, la utilización del procesador es muy baja (*U* = 0.40), por lo que parece razonable esperar que la capacidad de cómputo disponible sea suficiente para ejecutar las dos tareas dentro de plazo. Efectivamente, si asignamos mayor prioridad a τ_2 , a pesar de ser menos "importante", las dos tareas se ejecutan de forma admisible (Cronograma 2b).



Cronograma 2a. $P(\tau_1) > P(\tau_2)$



Cronograma 2b. $P(\tau_1) < P(\tau_2)$

[1] Ésta es precisamente la forma de ejecutar las tareas en Ada cuando se usa el pragma **priority.**

2.3.5 Prioridades monotónicas en frecuencia.

El ejemplo anterior pone de manifiesto la necesidad de utilizar criterios distintos de la "importancia" de las tareas para asignarles prioridades. *Liu* y *Layland* demostraron [Liu&73] que, en las condiciones mencionadas anteriormente (tareas periódicas e independientes y plazos de respuesta iguales a los períodos), la asignación de prioridades en orden inverso al de los períodos de las tareas (mayor prioridad a las tareas más frecuentes) es óptima, en el sentido de que garantiza los plazos de respuesta siempre que ello sea posible. Esta forma de asignar prioridades a las tareas se denomina de *prioridad al más frecuente o monotónica en frecuencia (rate monotonic)*.

Condiciones de Garantía de los Plazos.

ANÁLISIS BASADO EN EL FACTOR DE UTILIZACIÓN.

Teorema (*Liu* y *Layland*): *En un sistema de n tareas periódicas independientes con prioridades asignadas en orden de frecuencia, se cumplen todos los plazos de respuesta, si*

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N * (2^{1/N} - 1) = U_o(N) \qquad (15)$$

donde $U_0(N)$ es la *utilización mínima garantizada*, y su valor tiende a ln2 ≈ 0.693 al aumentar el número de tareas. La tabla 2 contiene algunos valores de $U_o(N)$.

N	U_o
1	1,000
2	0,828
3	0,779
4	0,756
5	0.743

Tabla 2. Utilización mínima garantizada.

Para analizar el comportamiento temporal de un sistema es útil el concepto de ***instante crítico***, definido por *Liu* y *Layland*. Un instante *t* es crítico para una tarea τ_i si es un instante de activación tal que el tiempo de respuesta de la tarea es máximo con respecto a todas sus ejecuciones. Los instantes críticos de τ_i son aquellos en los que se activa simultáneamente con todas las tareas de mayor prioridad, y los del sistema en conjunto son aquellos en los que se activan todas las tareas al mismo tiempo.

Ejemplo 2: Sea un sistema con tres tareas, τ_1, τ_2, y τ_3 , con los requisitos indicados en la tabla 3.

Tarea	T	C	P	U
τ_1	30	10	3	0.333
τ_2	40	10	2	0.250
τ_3	50	12	1	0.240

				0.823

Tabla 3. Requisitos para las tareas del ejemplo 2.

La utilización total es 0.823 > U(3) = 0.779. Por tanto, el sistema no cumple la prueba de utilización. El cronograma que se muestra a continuación (Figura 3) permite comprobar que la tarea 3 falla en t = 50.

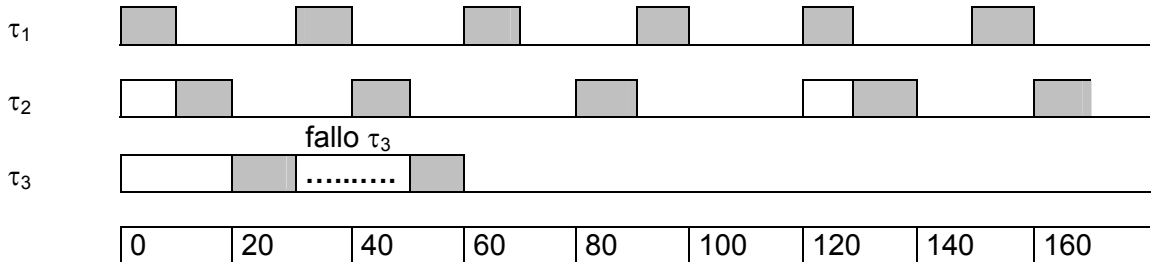


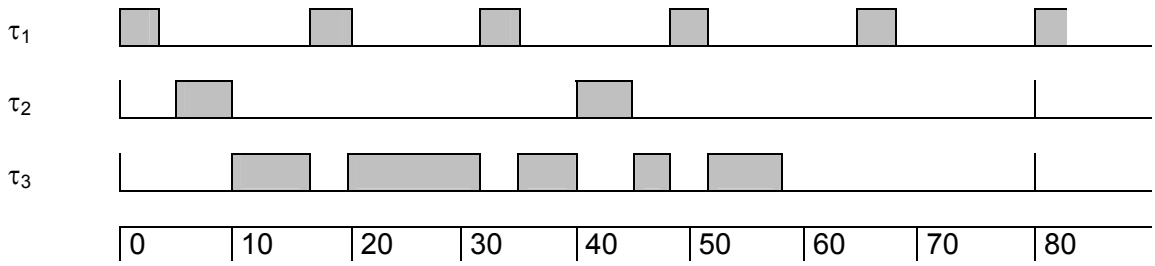
Figura 3. Incumplimiento de plazos de respuesta.

Ejemplo 3: Sea un sistema con tres tareas, τ_1, τ_2, y τ_3 , con los requisitos mostrados en la tabla 4.

Tarea	T	C	P	U
τ_1	16	4	3	0.250
τ_2	40	5	2	0.125
τ_3	80	32	1	0.400
				0.775

Tabla 4. Requisitos para las tareas del ejemplo 4.

La utilización total es 0.775 < U(3) = 0.779. Por tanto, las tres tareas están garantizadas si $P(\tau_1) > P(\tau_2) > P(\tau_3)$. El cronograma que se muestra a continuación (figura 4) permite comprobar si se cumplen los plazos de respuesta.



Cronograma 4. Cumplimiento de plazos de respuesta (ejemplo 3).

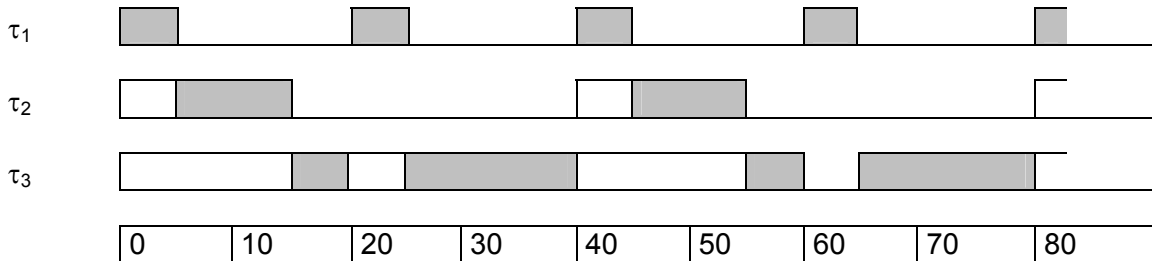
Veamos otro ejemplo en el que el sistema no pasa la prueba del factor de utilización y sin embargo se cumplen los plazos de respuesta.

Ejemplo 4: Sea un sistema con tres tareas, τ_1, τ_2, y τ_3 , con los requisitos indicados en la tabla 5.

Tarea	*T*	*C*	*P*	*U*
τ_1	20	5	3	0.250
τ_2	40	10	2	0.250
τ_3	80	40	1	0.500
				1.000

Tabla 5. Requisitos para las tareas del ejemplo 4.

La utilización total es 1.000 > U(3) = 0.779. Por tanto, las tres tareas no pasa la prueba del factor de utilización. Sin embargo, se cumplen los plazos de respuesta como se muestra en el cronograma de la figura 5.



Cronograma 5. Cumplimiento de plazos de respuesta (ejemplo 4).

Por tanto, el teorema de *Liu y Layland* define una condición suficiente para la garantía de los plazos, pero en muchos casos se pueden garantizar éstos con factores de utilización por encima de $U_o(n)$. Es una condición suficiente pero no necesaria.

2.4 Arquitectura del Software.

Hay que tener en cuenta que el concepto arquitectura es un término que tiene muchas definiciones diferentes dentro de un gran número de campos, pero para el caso que nos ocupa en Ingeniería del Software podríamos decir que: *"La arquitectura del software nos identifica los elementos mas importantes del sistema así como las relaciones existentes entre ellos, es decir, nos da una visión global del sistema".* En este trabajo el sistema es software que vamos a modelar o diseñar.

La arquitectura del software es un elemento necesario a la hora de modelar cualquier sistema ya que gracias a ella podremos entenderlo mejor, organizar su desarrollo, plantear la reutilización del software y hacerlo evolucionar. Aunque determinar los elementos que componen una arquitectura puede ser una labor compleja y complicada, actualmente existen un gran número de técnicas o metodologías de desarrollo que indican principios para identificar y diseñar dicha arquitectura, aunque la ayuda que ofrezcan actualmente sea muy limitada debido a que están basados en principios muy genéricos.

Las arquitecturas software no responden únicamente a requisitos estructurales, sino que están relacionadas con aspectos de rendimiento, usabilidad, reutilización, restricciones económicas y tecnológicas, e incluso cuestiones estéticas.

Actualmente existen muchas metodologías de desarrollo de software, desde métodos muy 'pesados' y burocráticos, métodos ajustables al proyecto y a las condiciones de desarrollo, hasta métodos 'ligeros' que surgen como respuesta a los excesos 'formales' de otros métodos.

Evidentemente, partiendo de los principios de tantas y diversas metodologías es muy difícil sacar una visión unificada sobre el diseño arquitectónico. Sin embargo sí que podemos destacar una serie de elementos comunes en aquellas que más se centran en este tema.

- La existencia de una fase en la que se establece o diseña una arquitectura base.
- La altísima dependencia que definen entre los casos de uso y la arquitectura, definiendo un caso de uso como una interacción típica entre el usuario y el sistema.

Desde un punto de vista arquitectónico, no todos los casos de uso tienen la misma importancia, destacando aquellos que nos ayudan a mitigar los riesgos más importantes y sobre todo aquellos que representan la funcionalidad básica del sistema a construir.

La arquitectura base estará especificada por diagramas que muestren subsistemas, interfaces entre los mismos, diagramas de componentes, clases, descripciones diversas, y por el conjunto de casos de uso básicos. Dichas de especificaciones nos permiten validar la arquitectura con los clientes y los desarrolladores, y asegurarnos de que es adecuada para implementar la funcionalidad básica deseada.

Una visión alternativa sería identificar el tipo de sistema que queremos construir. Todos sabemos que no hay dos aplicaciones iguales, pero que existen claros paralelismos entre las aplicaciones construidas para resolver problemas similares. El fijarnos en aplicaciones del mismo tipo tiene muchas ventajas ya que nos ayuda a entender las necesidades del cliente y las soluciones ya encontradas por otros. En I.S

a las soluciones probadas para problemas comunes se las denomina patrones de diseño [Gamma95].

La gran ventaja de la existencia de estos patrones de diseño es que se ahorra un valioso tiempo de diseño ya que permiten reutilizar soluciones.

Por tanto, se podría decir que “Construir una arquitectura” es tanto una actividad donde desarrollar ideas nuevas como una oportunidad de usar la experiencia acumulada, siendo casi siempre responsabilidad del desarrollador crear un producto de calidad y por tanto conocer el tipo de sistema a construir. Afortunadamente para esto último, los lenguajes de patrones nos pueden proporcionar una inestimable ayuda.. Estos lenguajes se podrían definir como: *"La especificación de una serie de elementos y sus relaciones (patrones de interacción) de modo que nos permiten describir buenas soluciones a los diferentes problemas que aparecen en un contexto específico".*

El objetivo de los patrones de diseño es el de capturar buenas prácticas que nos permitan mejorar la calidad del diseño de un sistema, determinando elementos que soporten roles útiles en dicho contexto, encapsulando complejidad, y haciéndolo más flexible. Los sistemas con objetivos similares comparten también una arquitectura común, unos procesos bien definidos, y un conjunto de elementos similares. Similar funcionalidad y servicio, similar estructura.

Cuando desarrollamos un sistema que se encuadra dentro de cierto tipo, es muy útil consultar lenguajes de patrones que traten el dominio en el que estamos. Un lenguaje de patrones nos sirve como referencia conceptual del dominio del problema. Además constituyen también un marco conceptual en el diseño de la arquitectura de nuestros sistemas, ya que como la función define a la forma, sintetizan por lo general soluciones arquitectónicas y estructurales probadas, muy útiles dentro del tipo de problemas que modelan.

De alguna forma, los patrones nos permiten identificar y completar los casos de uso básicos expuestos por el cliente, comprender la arquitectura del sistema a construir así como su problemática, y buscar componentes ya desarrollados que cumplan con los requisitos del tipo de sistema a construir (es decir, nos permiten obtener de una forma sencilla la arquitectura base que buscamos durante la fase de diseño arquitectónico).

Desafortunadamente los lenguajes de patrones tampoco son la panacea, y presentan muchas lagunas. Sobre todo, hay que recordar que todo este movimiento de documentación de diseño se origina a mediados de los noventa y que aún siendo mucho el trabajo realizado, no existe todavía ninguna estandarización sobre cómo abordar el desarrollo de estos lenguajes, ni ninguna clasificación que los relacione.

2.5 UML.

2.5.1 Introducción.

UML es una especificación de notación orientada a objetos. Divide cada proyecto en un número de diagramas que representan las diferentes vistas del mismo. Estos diagramas son los que representan la arquitectura del proyecto.

Con UML nos debemos olvidar del protagonismo excesivo que se le da al tradicional diagrama de clases en la programación orientada a objetos. Este representa una parte importante del sistema, pero solo una vista estática, es decir muestra al sistema parado. UML introduce nuevos diagramas que representan una visión dinámica del sistema. El diagrama de clases continua siendo muy importante, pero se debe tener en cuenta que su representación es limitada, y que ayuda a diseñar un sistema robusto con partes reutilizables, pero no a solucionar problemas de propagación de mensajes ni de sincronización o recuperación ante estados de error.

UML es un estándar de facto. Su utilización es independiente del lenguaje de programación y de las características de los proyectos, ya que UML ha sido diseñado para modelar cualquier tipo de proyectos, tanto informáticos como de arquitectura, o de cualquier otra rama.

UML permite la modificación de todos sus elementos mediante estereotipos y restricciones. Un estereotipo nos permite indicar especificaciones del lenguaje al que se refiere el diagrama de UML. Una restricción identifica un comportamiento forzado de una clase o relación, es decir mediante la restricción estamos forzando el comportamiento que debe tener el objeto al que se le aplica.

2.5.2 Tipos de Diagramas.

Se dispone de dos tipos diferentes de diagramas: los que dan una vista estática del sistema y los que dan una visión dinámica.

Los diagramas estáticos son:

- Diagrama de clases: muestra las clases, interfaces, colaboraciones y sus relaciones. Son los más comunes y dan una vista estática del proyecto.
- Diagrama de objetos: Es un diagrama de instancias de las clases mostradas en el diagrama de clases. Muestra las instancias y como se relacionan entre ellas. Se da una visión de casos reales.
- Diagrama de componentes: Muestran la organización de los componentes del sistema. Un componente se corresponde con una o varias clases, interfaces o colaboraciones.
- Diagrama de despliegue.: Muestra los nodos y sus relaciones. Un nodo es un conjunto de componentes. Se utiliza para reducir la complejidad de los diagramas de clases y componentes de un gran sistema. Sirve como resumen e índice.
- Diagrama de casos de uso: Muestran los casos de uso, actores y sus relaciones. Muestra quien puede hacer que y relaciones existen entre acciones (casos de uso). Son muy importantes para modelar y organizar el comportamiento del sistema.

Lo diagramas dinámicos son:

- Diagrama de secuencia, Diagrama de colaboración: Muestran a los diferentes objetos y las relaciones que pueden tener entre ellos, los mensajes que se envían entre ellos. Son dos diagramas diferentes, que se puede pasar de uno a otro sin perdida de información, pero que nos dan puntos de vista diferentes del sistema. En resumen, cualquiera de los dos es un Diagrama de Interacción.
- Diagrama de estados: muestra los estados, eventos, transiciones y actividades de los diferentes objetos. Son útiles en sistemas que reaccionen a eventos.
- Diagrama de actividades: Es un caso especial del diagrama de estados. Muestra el flujo entre los objetos. Se utilizan para modelar el funcionamiento del sistema y el flujo de control entre objetos.

Como podemos ver el número de diagramas es muy alto, en la mayoría de los casos excesivos, y UML permite definir solo los necesarios, ya que no todos son necesarios en todos los proyectos. En este trabajo se dará una breve explicación de los más necesarios.

Cada diagrama usa la anotación pertinente y la suma de estos diagramas crean las diferentes vistas. Las vistas existentes en UML son:

- Vista casos de uso: Se forma con los diagramas de casos de uso, colaboración, estados y actividades.
- Vista de diseño: Se forma con los diagramas de clases, objetos, colaboración, estados y actividades.
- Vista de procesos: Se forma con los diagramas de la vista de diseño. Recalcando las clases y objetos referentes a procesos.
- Vista de implementación: Se forma con los diagramas de componentes, colaboración, estados y actividades.
- Vista de despliegue: Se forma con los diagramas de despliegue, interacción, estados y actividades.

2.5.3 Diagramas de caso de uso.

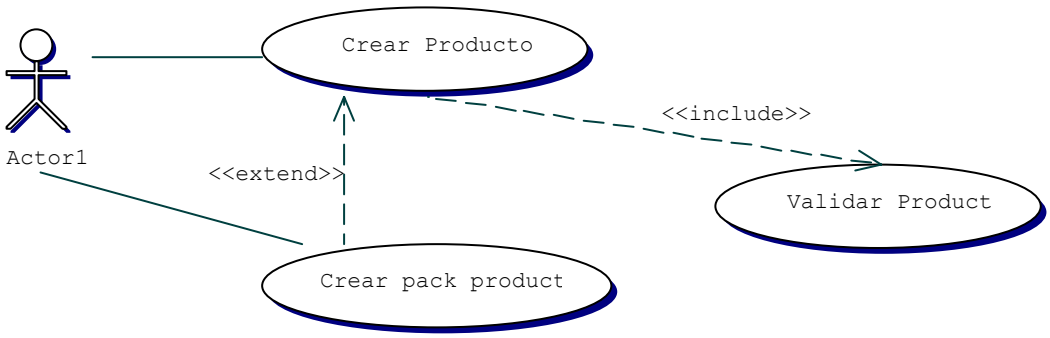
Se emplean para visualizar el comportamiento del sistema, una parte de él o de una sola clase, de forma que se pueda describir como responde esa parte del sistema. El diagrama de uso es muy útil para definir como debe ser el comportamiento del sistema. Por ello es un buen sistema para documentar partes del código que deban ser reutilizables por otros desarrolladores. El diagrama puede ser utilizado para que los expertos de dominio se comuniquen con los informáticos sin entrar en detalles técnicos.

En el diagrama nos encontramos con diferentes figuras que pueden mantener diversas relaciones:

- Casos de uso: representado por una elipse, cada caso de uso contiene un nombre, que indica su funcionalidad. Los casos de uso pueden tener relaciones con otros casos de uso. Sus relaciones son:
    - Include: un caso de uso incluye a otro.
    - Extends: Una relación de una caso de Uso A hacia un caso de uso B indica que el caso de uso B implementa la funcionalidad del caso de uso A.

- Generalization: Es la típica relación de herencia.

- Actores: se representan por un muñeco. Sus relaciones son:
  - Communicates: Comunica un actor con un caso de uso, o con otro actor.
  - Parte del sistema (System boundary): Representado por un cuadro, identifica las diferentes partes del sistema y contiene los casos de uso que la forman.



En este grafico encontramos tres casos de usos *Crear producto* utiliza *Validar producto*, y *Crear pack productos* es una especialización de *Crear productos*.

Podemos emplear el diagrama de dos formas diferentes, para modelar el contexto de un sistema, y para modelar los requisitos del sistema.

2.5.4 Diagramas de clase.

Forma parte de la vista estática del sistema. En el diagrama de clases como ya hemos comentado será donde definiremos las características de cada una de las clases, interfaces, colaboraciones y relaciones de dependencia y generalización. Es decir, es donde daremos rienda suelta a nuestros conocimientos de diseño orientado a objetos, definiendo las clases e implementando las ya típicas relaciones de herencia y agregación.

En el diagrama de clases debemos definir a éstas y a sus relaciones.

La Clase

Una clase está representada por un rectángulo que dispone de tres apartados, el primero para indicar el nombre, el segundo para los atributos y el tercero para los métodos (ver figura 1).

Cada clase debe tener un nombre único, que las diferencie de las otras.

Un atributo representa alguna propiedad de la clase que se encuentra en todas las instancias de la clase. Los atributos pueden representarse solo mostrando su nombre, mostrando su nombre y su tipo, e incluso su valor por defecto.

Un método u operación es la implementación de un servicio de la clase, que muestra un comportamiento común a todos los objetos. En resumen, es una función que le indica a las instancias de la clase que hagan algo.

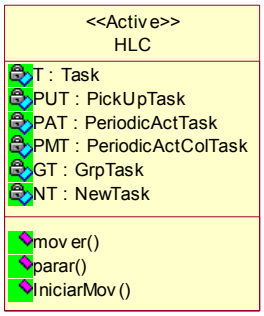


Figura 1

2.5.4.1 Relaciones entre clases.

Existen tres relaciones diferentes entre clases, Dependencias, Generalización y Asociación. En las relaciones se habla de una clase destino y de una clase origen que se representan mediante una flecha, se pueden modificar con estereotipos o con restricciones.

Dependencia

Es una relación de uso, es decir una clase usa a otra, que la necesita para su cometido. Se representa con una flecha discontinua va desde la clase utilizadora a la clase utilizada. Con la dependencia mostramos que un cambio en la clase utilizada puede afectar al funcionamiento de la clase utilizadora, pero no al contrario. Aunque las dependencias se pueden crear tal cual, es decir sin ningún estereotipo UML permite dar mas significado a las dependencias, es decir concretar más, mediante el uso de estereotipos.

Generalización.

UML soporta tanto herencia simple como herencia múltiple. Aunque la representación común es suficiente en el 99.73% de los casos UML nos permite modificar la relación de Generalización con un estereotipo y dos restricciones.

Asociación.

Especifica que los objetos de una clase están relacionados con los elementos de otra clase. Se representa mediante una línea continua, que une las dos clases. Podemos indicar el nombre, multiplicidad en los extremos, su rol, y agregación.

2.5.4.2 Diagrama de objetos.

Forma parte de la vista dinámica del sistema. En este diagrama se modelan las instancias de las clases del diagrama de clases. Muestra a los objetos y sus relaciones, en un momento concreto del sistema. Estos diagramas contienen objetos y enlaces. En los diagramas de objetos también se pueden incorporar clases, para mostrar la clase de la que es un objeto representado.

2.5.4.3 Diagrama Secuencia.

El diagrama de secuencia forma parte del modelado dinámico del sistema. Se modelan las llamadas entre clases desde un punto concreto del sistema. Es útil para observar la vida de los objetos en sistema, identificar llamadas a realizar o posibles errores del modelado estático, que imposibiliten el flujo de información o de llamadas entre los componentes del sistema.

En el diagrama de secuencia se muestra el orden de las llamadas en el sistema. Se utiliza un diagrama para cada llamada a representar. Es imposible representar en un solo diagrama de secuencia todas las secuencias posibles del sistema, por ello se escoge un punto de partida. El diagrama se forma con los objetos que forman parte de la secuencia, estos se sitúan en la parte superior de la pantalla, normalmente en la izquierda se sitúa al que inicia la acción. De estos objetos sale una línea que indica su vida en el sistema. Esta línea simple se convierte en una línea gruesa cuando representa que el objeto tiene el foco del sistema, es decir cuando el esta activo.

2.6 UML-MAST.

Dado que esta herramienta es un elemento básico para la realización de este proyecto, inicialmente es necesario un conocimiento básico sobre que es esta herramienta y para qué sirve.

UML-MAST es una metodología y un conjunto de herramientas graficas desarrolladas para analizar y modelar sistemas de tiempo real que están siendo desarrollados por la universidad de Cantabria, utilizando métodos orientados a objetos sobre herramientas CASE basadas en la notación UML (Unified Modeling Language).

De una forma menos formal se podría decir que UML-MAST es una herramienta que permite realizar un análisis temporal a partir de la definición de unas restricciones temporales aplicadas a una arquitectura software descrita sobre UML. UML-MAST trabaja sobre otra herramienta CASE, Rational Rose. Rational Rose es un entorno grafico que permite la descripción de arquitecturas de software haciendo uso de la notación UML.

MAST toma como referencia, el modelo de descripción de sistemas denominado "4+1 View" [Kruchten 1995] que se muestra en la figura2. Añadiendo una nueva vista[2] a la *"Process View"*, que será *"Mast RT View"*. Esta vista esta basada en los conceptos y componentes definidos en MAST y será la encargada de representar el comportamiento en tiempo real de nuestro sistema a definir.

[2] Tratándose de la herramienta Racional una vista puede entenderse como una plantilla sobre la que empezar a trabajar

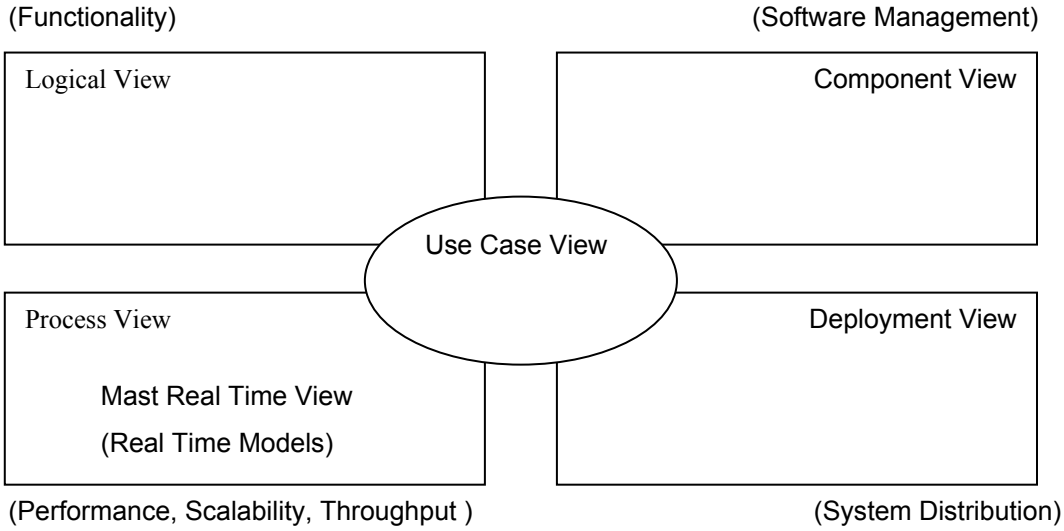


Figura 2

Así, *"Mast RT View"* es una vista complementaria de la descripción UML del sistema que modela la plataforma que se utiliza, las características de los componentes lógicos de su software y las transacciones que pueden ocurrir con los requisitos temporales que en ellas se establecen. Con esta vista el diseñador puede construir gradualmente el modelo de tiempo real del sistema que diseña, de forma paralela al desarrollo de su modelo lógico.

La descripción básica y formal de la "Mast RT View" la constituye su metamodelo, en el cual no vamos a entrar en profundidad ya que se explicará mas adelante. Pero a modo de introducción vamos a dar una vista rápida de los pasos de diseño:

1. Definición de la estructura general del modelo, estableciendo sus secciones, los aspectos del sistema que describe y los componentes que se construyen.
2. Describir los componentes que se utilizan para construir el modelo de tiempo real, definiendo su naturaleza así como los atributos que describen su comportamiento cuantitativo.
3. Describir las relaciones que se pueden establecer entre los componentes dentro del modelo.
4. En el modelo a través de estereotipos se indica el tipo de componentes UML (paquete, clase, atributo, actividad, acción, eventos, etc.) con el que sé instancia cada componente del metamodelo en el modelo concreto del sistema.

La descripción del metamodelo se realiza a través de diagramas de clases que describen gráficamente su estructura, y de texto que describe conceptualmente cada componente y sus atributos. A fin de minimizar dependencias cruzadas en la descripción de cada diagrama se incluye junto a cada uno la mayoría de las descripciones de sus clases, aunque ello implique la repetición de clases a lo largo del documento.

Capítulo 3
UML MAST

3.1 Introducción.

UML-MAST (*Modeling and Analysis Suite for Real Time Aplications*) [Referencia] es una metodología y un conjunto de herramientas gráficas desarrolladas para modelar y analizar sistemas de tiempo real, desarrollados utilizando métodos orientados a objetos basados en la notación UML [Referencia]. Los componentes de modelado y las herramientas de análisis corresponden al entorno MAST, que tiene como objetivo principal permitir el desarrollo de aplicaciones de tiempo real, comprobar el comportamiento temporal de la aplicación y realizar un análisis de planificación (*analysis schedulability*) para chequear los requerimientos temporales críticos.

Un modelo de una aplicación de tiempo real no solo debe representar las características de la arquitectura distribuida del sistema sino que además debe explicar los requisitos que tiene impuestos. Muchas de las técnicas de análisis existentes para la planificación de estos sistemas están basadas en un modelo denominado lineal. En este modelo cada tarea es activada por la llegada de un evento o mensaje, y cada mensaje es enviado por una tarea simple. Este modelo no permite interacciones complejas entre las respuestas a diferentes secuencias de eventos, excepto para la sincronización de recursos compartidos (*shared resources*). Este análisis de planificación no será aplicable a sistemas en los cuales existan estas interacciones. Hay otras muchas técnicas de análisis para sistemas de tiempo real que tienen en cuenta sincronizaciones complejas, así como interacciones entre secuencias de eventos, incluso diferentes mecanismos de planificación que fijan prioridades.

MAST permite describir una rica representación de modelos de sistemas de tiempo real, por ejemplo tareas que son activadas con la llegada de eventos o eventos que se envían a la conclusión de una tarea. Estas acciones son útiles para analizar sistemas de tiempo real que han sido diseñados utilizando metodología orientada a objetos. De hecho, las herramientas para el diseño de los modelos MAST han sido desarrolladas para que de manera automática se obtengan estos modelos de descripción a través del estándar UML.

Mast también incluye herramientas para el análisis de planificación, que utilizan las ultimas técnicas *offset-based* para intensificar los resultados del análisis. Estas técnicas de planificación son muy pesimistas respecto a otras, por lo cual han sido incluidas con la intención de completar el conjunto de herramientas de MAST. También incluye herramientas para realizar una óptima asignación prioridades y simulaciones del comportamiento temporal del sistema.

La implementación de las herramientas de MAST está abierta y es fácilmente extensible. Esto quiere decir que los creadores de MAST han dejado abierta la posibilidad que otros grupos puedan aumentarlas. La primera versión solo es valida para sistemas con prioridades fijas, los sistemas con una planificación dinámica de prioridades serán añadidos en un futuro.

MAST ha sido diseñado con la idea de poder manejar tanto sistemas con un procesador simple como sistemas multiprocesadores o sistemas distribuidos. Además

también describe el trabajo de tareas, se pueden iniciar por un evento o por una combinación de eventos, tanto internos como externos (periódicos, esporádicos, que solo ocurren una vez, etc.) o la creación de estos eventos a causa de la finalización de la tarea.

3.2 Descripción de UML-MAST

Como se ha mencionado anteriormente los modelos MAST han sido desarrollados para que se obtengan de manera automática del estándar UML. A causa de esto, para crear estos modelos se trabaja sobre una herramienta utilizada para el diseño con UML (ROSE 2000). A esta herramienta se le ha añadido un framework específico y un compilador especifico que genera un modelo Mast-File que es la formulación base del entorno MAST.

La MAST RT View (vista lógica de tiempo real) es una vista complementaria de la descripción UML de un sistema que modela su comportamiento de tiempo real. Con ella el diseñador puede construir gradualmente el modelo de tiempo real del sistema que diseña, de forma paralela al desarrollo de su modelo lógico (diagrama de clases, casos de uso, diagramas de secuencia, etc.).

La modularidad de MAST RT View coincide con la de la vista lógica, y con ello se puede disponer de un modelo de tiempo real del sistema completo, como del modelo de cada clase lógica. Con ello se consigue que los modelos de tiempo real sean reusables y que puedan construir parte de la especificación de tiempo real.

La capacidad de modelado y análisis de esta vista va a ser proporcionada por el entorno MAST. Este entorno del modelo de tiempo real se emplea de base a múltiples herramientas de análisis de planificabilidad (schedulability), análisis de rendimiento (performance), diseño, animación etc. Además, este entorno ofrece componentes conceptuales para definir recursos hardware (procesadores, redes de comunicación, equipos, etc.), mecanismos de sincronización (semáforos, monitores, etc.), recursos software (threads, procesos, drivers, etc.), componentes lógicos (clases, métodos, procedimientos, etc.), y escenarios de tiempo real que definen las situaciones de análisis y que se formulan como conjunto de transacciones concurrentes, de fuentes externas que las invocan y de las restricciones temporales que se requieren de ellas.

3.2.1 Vista UML de tiempo real

Los componentes que constituyen esta vista, son diagramas de clases y diagramas de actividad que en su conjunto representan el comportamiento dinámico de los componentes hardware y software del sistema. Los tipos de componentes que constituyen la vista de tiempo real y las relaciones que se pueden establecer entre ellos, se define a través del metamodelo “MAST_UML”.

La MAST RT View se compone de tres secciones complementarias, cada una de ellas describe un aspecto específico del modelo de tiempo real.

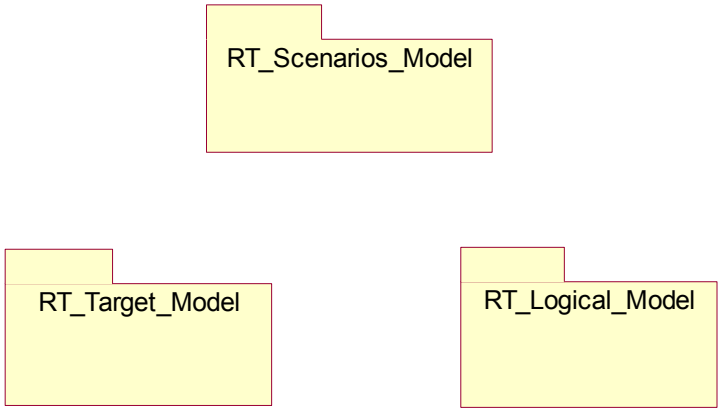


Figura 1

- **Modelo de la plataforma:** Modela la capacidad de procesamiento y las restricciones operativas de los recursos de procesamiento hardware y software que constituyen la plataforma sobre la que se ejecuta el sistema. Estos recursos son: los procesadores, threads, coprocesadores, equipos hardware específicos, redes de comunicación, etc, que tienen en común ser los agentes que ejecutan las actividades del sistema.
- **Modelo de los componentes lógicos:** Modela el tiempo procesado que requiere la ejecución de las operaciones definidas en las clases. En esta sección del modelo se declaran los recursos que necesita cada operación para llevarse a cabo, en especial aquellos, que por ser requeridos por varias operaciones concurrentes en régimen de exclusión mutua, pueden ser causa de retraso en la ejecución de las operaciones.
- **Escenarios de tiempo real:** Cada escenario se modela como un conjunto de transacciones que describen las secuencias de eventos y actividades que deben ser analizadas para que se satisfagan los requerimientos de tiempo real establecidas en ellas. Cada transacción es una descripción no iterativa de las secuencias de actividades y eventos que se desencadenan como respuesta a un patrón de eventos autónomo (procedentes del entorno exterior al sistema, de timers, de relojes, de dispositivos hardware integrados, etc.) y de sus requisitos temporales.

3.2.1.1 Modelo de la plataforma.

Modela la capacidad de procesamiento y las restricciones operativas de los recursos de procesamiento hardware y software que constituyen la plataforma sobre la que se ejecuta el sistema.

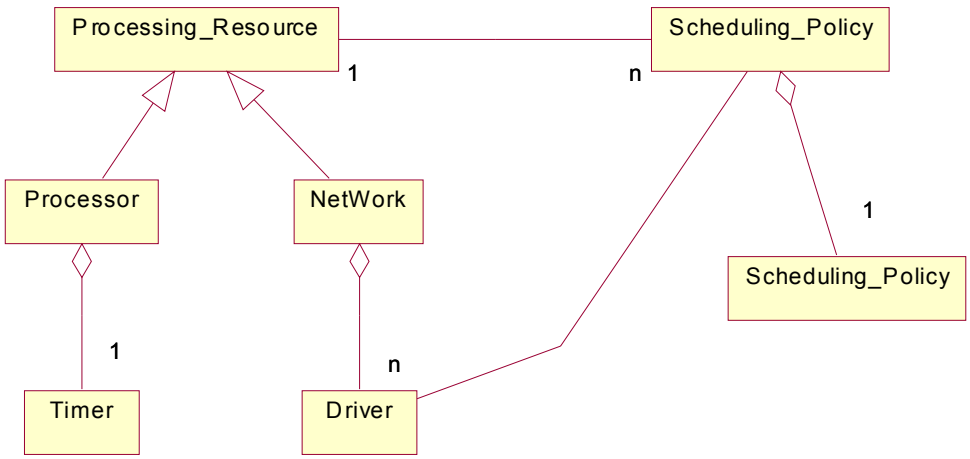


Figura 2

En la figura 2 se muestran las clases abstractas básicas del metamodelo que se utilizan para describir el modelo de la plataforma.

El *Processing_Resource* modela procesadores y redes de comunicación. De esta clase derivan dos clases especializadas, como son *Procesor* y *Network*. Processor representa procesadores, coprocesadores o equipos embarcados que ejecutan actividades. *Network* representa redes de comunicaciones cuya actividad consiste en transferencia de información entre procesadores. Los componentes *Driver* y *Timer* introducen las tareas, que se ejecutan en segundo plano, que deben ejecutar los procesadores para la gestión de los *drivers* de comunicaciones o los *timer* hardware.

En este proyecto solo se ha utilizado la clase *Processor*. Su capacidad de cómputo se distribuye entre la ejecución de código de las actividades del sistema que tiene asignado y la ejecución de las tareas de gestión, monitorización y cambio de contexto entre actividades. Hay dos modelos diferentes de *Processor*, de las cuales se va a explicar la que ha sido necesaria utilizar: *Fixed_Priority_Processor,* que es un Processor que opera bajo una prioridad fija.

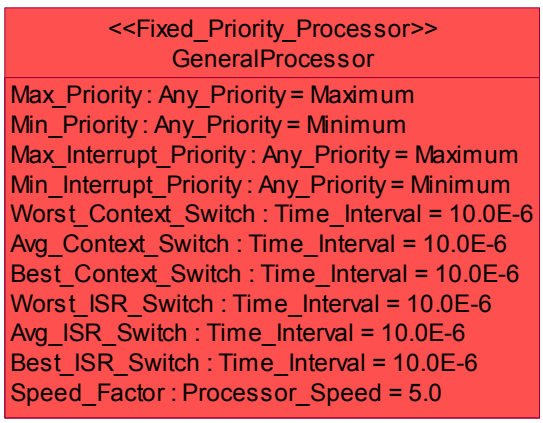


Figura 3

En la Figura 3 se puede ver un *Processor* del tipo *Fixed_Priority_Processor* que tiene una serie de atributos que deben adecuarse al procesador en el cual se va pretender ejecutar el modelo. Como en nuestro caso solo se realiza un estudio entre diferentes modelos hemos tenido en cuenta que en todos los modelos que se desean comparar sean iguales, tanto en el caso de los atributos que hacen referencia a cualidades *hardware*, como a los límites que se pueden fijar en MAST (mínima prioridad = 1, máxima prioridad = 32).

Es importante conocer que cualidades definen cada una de los atributos que podemos ver en la figura 1. Estas son:

- **Speed_Factor:** Factor de velocidad de procesamiento del recurso. El tiempo real de ejecución de cualquier actividad que se ejecute en el recurso se obtendrá dividiendo el tiempo normalizado de ejecución establecido en las operaciones que lleva a cabo por el factor de velocidad.
- **Max_Priority**: Máximo nivel de prioridad para una actividad de procesado que se ejecuta en el procesador.
- **Min_Priority**: Mínimo nivel de prioridad para una actividad de procesado que se ejecuta en el procesador.
- **Max_Interrupt_Priority**: Máximo nivel de prioridad para una rutina de interrupción que se ejecuta en el procesador.
- **Min_Interrupt_Priority**: Mínimo nivel de prioridad para una rutina de interrupción que se ejecuta en el procesador.
- **Worst_Context_Switch**: Tiempo máximo de cómputo (peor caso) que el procesador emplea en realizar un cambio de contexto entre dos actividades de diferentes hilos de ejecución (Thread).
- **Avg_Context_Switch**: Tiempo promedio de cómputo que el procesador emplea en realizar un cambio de contexto entre dos actividades de diferentes hilos de ejecución (Thread).
- **Best_Context_Switch**: Tiempo mínimo de cómputo (mejor caso) que el procesador emplea en realizar un cambio de contexto entre dos actividades de diferentes hilos de ejecución (Thread).
- **Worst_ISR_Switch**: Tiempo máximo de cómputo (peor caso) que el procesador emplea en conmutar a una rutina de interrupción.
- **Avg_ISR_Switch**: Tiempo promedio de cómputo que el procesador emplea en conmutar a la rutina de atención de una interrupción.
- **Best_ISR_Switch**: Tiempo mínimo de cómputo (mejor caso) que el procesador emplea en conmutar a una rutina de interrupción.

Como se puede observar en la figura 2, cada *Processing_Resource* puede tener asociados varios *Scheduling_Servers*, que se utilizan para modelar *threads* o tareas dentro de las que se realizan las actividades del sistema. Cada uno de estos *Scheduling_Servers* debe tener asociada una política de planificación (*Schudeling_Policy*).

Los *Scheduling_Servers* son muy importantes ya que se encargaran de simular las tareas concurrentes. Por ello, se debe de ser muy meticuloso y tener muy claro a la hora de modelar el sistema todos los hilos de ejecución existentes.

Mast solo ofrece una posibilidad para modelar los *Schudeling_Server*, la clase denominada *FP_Sched_Server* que modela una planificación basada en prioridades fijas y en la exclusión mutua.

.

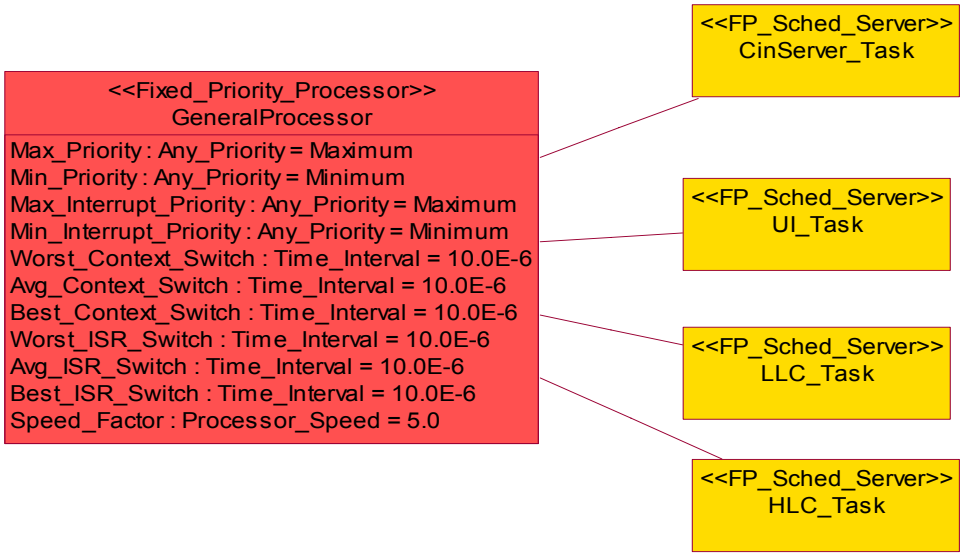


Figura 4

En la Figura 4 podemos ver varios *FP_Sched_Server* asociados a un *Fixed_Priority_Processor*, en cada uno de ellos va a estar modelado un hilo de ejecución de las tareas de los modelos diseñados en este trabajo.

Por último, cada uno de estos *FP_Sched_Server* debe tener asociado una política de planificación compatible. En nuestro caso hemos escogido una política del tipo *Fixed_Priority_Policy*, como se muestra en la figura 5.

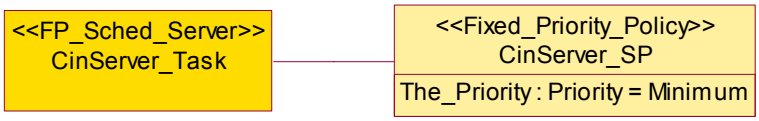


Figura 5

Al *Fixed_Priority_Policy* se le puede asignar un valor de la prioridad que va a usar el *FP_Sched_Server*, o bien dejar que sea MAST quien le asigne el valor cuando se realice el análisis de planificación del modelo, durante el cual se podrá indicar el tipo de política de asignación de prioridades que debe usar. Esto se hace como se puede observar en la Figura 5 dejando el valor *Minimun*. Con esto MAST ya sabe que tendrá que asignar el valor de la prioridad.

Por último, cabría destacar otro tipo de política de planificación que se le puede asignar al *FP_Sched_Server.* Se trataría de una política de planificación fija pero a diferencia de la anterior, se basa en el escrutinio periódico de la tabla de procesos dispuestos para su planificación. Es decir, hace un sondeo periódico del evento que lo activa.

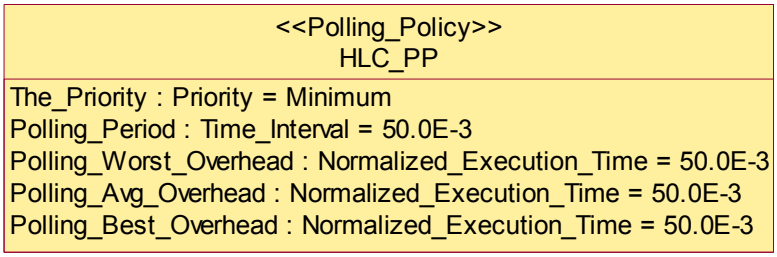


Figura 6

En la Figura 6 podemos ver un ejemplo de *Polling_Policy*. Se puede observar en la figura la existencia de varios atributos que caracterizaran la actuación de este modelo de planificación. Los atributos son los siguientes:

- Polling_Period: Período de escrutinio del planificador.
- Polling_Worst_Overhead: Tiempo normalizado máximo (peor caso) de sobrecarga debida a la planificación.
- Polling_Avg_Overhead: Tiempo normalizado promedio de sobrecarga debida a la planificación.
- Polling_Best_Overhead: Tiempo normalizado mínimo (mejor caso) de sobrecarga debida a la planificación.

3.2.1.2 Modelo de los componentes lógicos.

El modelo de tiempo real de los componentes lógicos modela el comportamiento temporal de los componentes funcionales (clases, métodos, procedimientos, operaciones, etc.) que están definidos en el sistema y cuyos tiempos de ejecución condicionan el cumplimiento de los requisitos temporales definidos en los escenarios de tiempo real que van a analizarse. El modelo de cada componente lógico describe los dos aspectos que condicionan su tiempo de ejecución: el tiempo que requiere la ejecución de su código y los bloqueos que puede sufrir su ejecución como consecuencia de que necesite acceder en régimen exclusivo a recursos compartidos (*Shared_Resource*), por otros componentes lógicos que se ejecutan concurrentemente con él.

El modelo de tiempo real de los componentes lógicos se establece con las siguientes características:

- Las temporizaciones de las operaciones se definen normalizadas, esto es, se formulan con parámetros cuyos valores son independientes de la plataforma en que se van a ejecutar.
- El modelo de interacción entre componentes lógicos se formula de forma parametrizada, identificando los recursos que son potenciales causas de bloqueo (*Shared_Resource*) y dejando hasta la descripción del escenario la declaración de los componentes lógicos concretos con los que va a interferir.

- El modelo de tiempo real de los componentes lógicos, se formula con una modularidad paralela a la modularidad que en la vista lógica ofrecen los componentes lógicos que se modelan.

También hay que decir, que dentro del modelo de los componentes lógicos se declaran un conjunto de diagramas de clase y diagramas de actividad en los que se declaran:

- Los componentes que modelan el comportamiento de las clases de la vista lógica del sistema que son relevantes a efecto de su respuesta de tiempo real.
- Operaciones predefinidas que realizan los coprocesadores, dispositivos o periféricos no programables del sistema y que influyen en la respuesta temporal del sistema.
- Recursos compartidos que requieren ser accedidos por los componentes funcionales en régimen de exclusión mutua y que describen las posibles interacciones con otros componentes que se ejecutan en concurrencia.

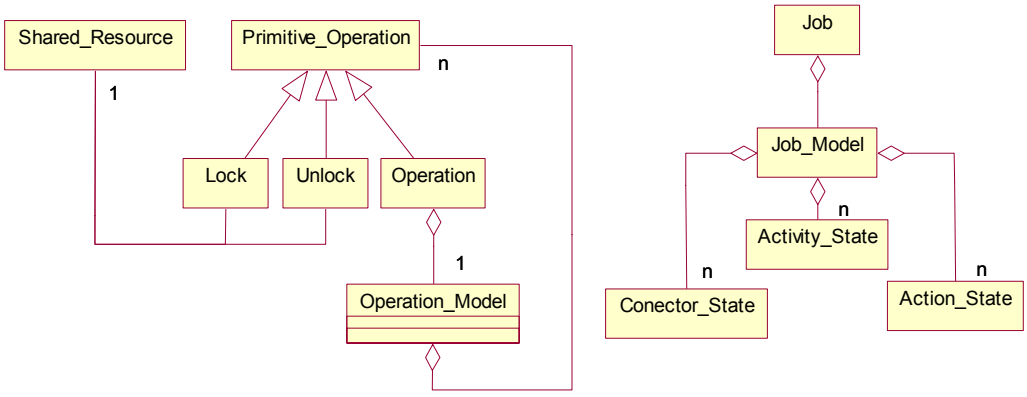


Figura 7

En la figura 7, se muestran las principales clases abstractas del metamodelo MAST con las que se formula el modelo de los componentes lógicos del sistema.

Se puede observar en la figura 7 la clase *Shared_Resource*. Esta clase modela un recurso que es requerido en exclusión mutua por diferentes operaciones concurrentes y es causa potencial de posibles bloqueos al acceder a él. A la hora de utilizar un *Shared_Resource* en MAST, éste da dos opciones de recurso compartido: *Priority_Inheritance_Resource* e *Inmediate_Ceiling_Resource.* En este trabajo solo se ha utilizado *Inmediate_Ceiling_Resource*. Con este tipo de recurso compartido, cuando una actividad accede a él hereda su techo de prioridad, que es la mayor prioridad de todos los recursos que se utilizan. Esta prioridad se calcula y establece estáticamente.

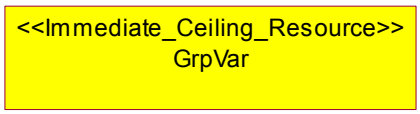


Figura 8

En la figura 8 podemos ver el modelo *Inmediate_Ceiling_Resource* al cual solo hay que asignarle un nombre para poder acceder a él desde las operaciones.

En la figura 7, también se puede ver la clase *Primitive_Operation*. Ésta representa cualquier tipo de operación que puede declararse. También se puede observar como hay tres clases que derivan de *Primitive_Operation,* que son: *Lock, Unlock, Operation.*

La clase *Lock* representa una operación primitiva que espera a que un Shared_Resource esté libre y que termina cuando accede a él en régimen exclusivo. Mientras, que la clase *Unlock* es una operación primitiva que representa la liberación de un recurso compartido previamente tomado. La clase *Operation* modela la temporización de la ejecución de un segmento de código que se ejecuta dentro de un mismo Schudeling_Server (thread). La característica específica de un componente lógico modelado por una *Operation* es que en su código no existe ningún componente de sincronización con otro thread.

A la hora de incluir las operaciones (*Operation*) en el modelo de componentes lógicos, MAST nos da tres opciones para poder definir el tipo de operación deseada. Estas son: operación-simple (*Simple_Operation)*, operación-compuesta (*Composite_Operation)* y operación-encerrada *(Enclosing_Operation)*.

La operación simple (*Simple_Operation)* describe la temporización de una operación simple. La operación que se modela se ejecuta de forma autónoma, pero compite con otras operaciones concurrentes en el uso de recursos compartidos o de recursos de procesado.

Cuando se utilice esta operación se van a tener que definir tres atributos muy importantes para el desarrollo temporal del sistema (figura 9):

- WCET: Tiempo máximo (peor caso) que tarda en ejecutarse la operación.
- ACET: Tiempo medio que tarda en ejecutarse la operación.
- BCET: Tiempo mínimo (mejor caso) que tarda en ejecutarse la operación.

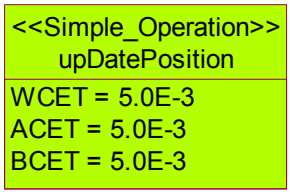


Figura 9

La operación-compuesta (*Composite_Operation*) define una secuencia ordenada de operaciones. Siempre que se define una operación-compuesta, debe añadírsele un modelo de una operación (*Operation_Model*) que se describe mediante un diagrama de actividad UML y consiste en una única activity state de UML con una secuencia do/ por cada operación primitiva que se compone.

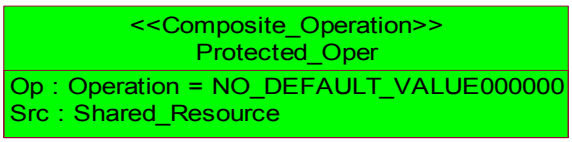


Figura 10

En la Figura 10 podemos observar un modelo de operación compuesta (*Composite_Operation*). A éstas se le pueden añadir una serie de parámetros o atributos, en este caso se le añaden dos atributos. La operación Protected_Oper que aparece en la Figura 10 se podría definir como una operación compuesta parametrizada que se utiliza para invocar una operación de cualquier objeto protegido. El parámetro OP representa a la operación que se invoca. Src representa el recurso compartido que debe ser accedido en régimen exclusivo. Al concluir la operación se liberará.

En la Figura 11 podemos ver el diagrama de actividad de Protected_Oper, en el que se puede observar como se reserva primero el recurso compartido (Src), se invoca la operación y finalmente se libera el recurso.

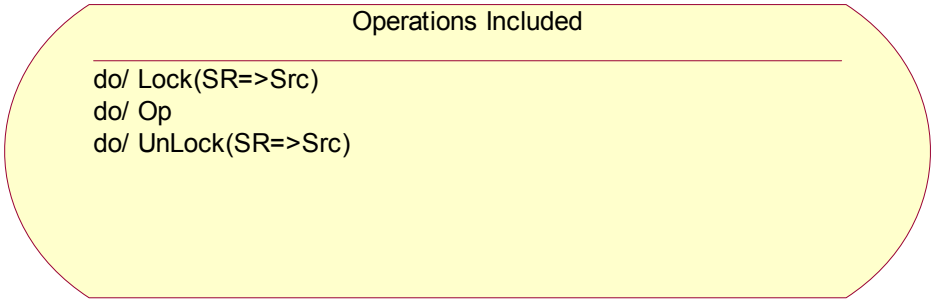


Figura 11

En la Figura 12, se puede observar otro tipo de operación compuesta (*Composite_Operation*). En este caso utilizamos la operación getGrp, en la cual viene definido el diagrama de actividad, que representa el modelo de operación, de la Figura 13. En esta operación se puede observar una sensible diferencia con la anterior, ya que realiza dos operaciones entre la que podemos destacar la llamada a la anterior operación compuesta (Protected_Oper) en la que se le asignan los valores deseados a los parámetros definidos en la operación.

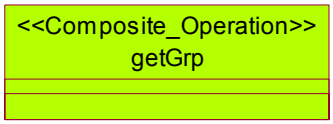


Figura 12

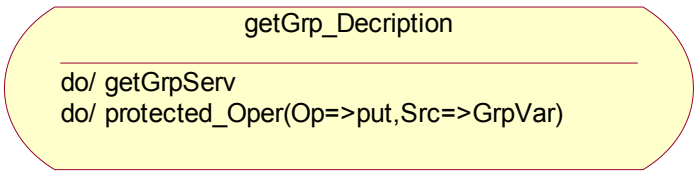


Figura 13

3.2.1.3 Escenarios de tiempo real.

Modela la Configuración o modo de operación que puede ser alcanzado por el sistema en los que están establecidos requisitos de tiempo real. Está constituido por el conjunto de los modelos de las transacciones que ocurren en él.

Una transacción describe la secuencia no iterativa de actividades que se desencadenan como respuesta a un patrón de eventos externos, que sirve de marco para definir los requerimientos temporales. Cada transacción incluye todas las actividades que se desencadenan como consecuencia del patrón de entrada y todas las actividades que requieren sincronización directa entre ellas (intercambio de eventos, sincronización por invocación, etc.).

El conjunto de transacciones que constituyen el modelo de sistema proporciona una descripción complementaria a la descripción funcional del sistema convencional. Con el modelo MAST se explicitan la temporalidad, la dependencia de flujo y la concurrencia entre las actividades de la aplicación.

Una transacción incluye todas las actividades interdependientes por transferencia de flujo o sincronización. Por lo que los conjuntos dependientes de las diferentes transacciones son disjuntos entre si y no intercambian eventos entre ellos.

Entre las diferentes transacciones de un sistema existen las relaciones y dependencias que se establecen, debidos a que las diferentes actividades de unas y de otras compiten por acceder al uso en régimen de exclusividad de los recursos comunes (*Processing_Resource* y *Shared_Resource*) que puedan existir.

Aunque las transacciones son secuencias no iterativas de actividades, pueden ser activadas periódicamente y pueden solaparse en el tiempo sucesivas ejecuciones de ellas. Esto es lo habitual en sistemas que operan en modo pipeline.

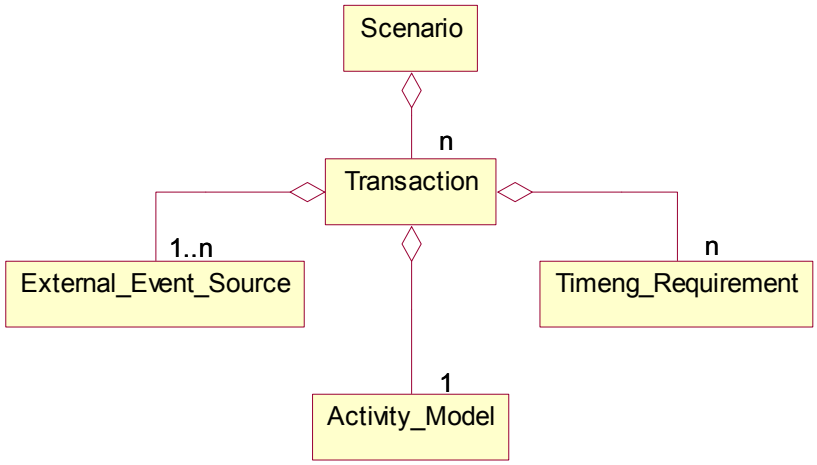


Figura 14

En la figura 14, se muestra que el modelo de un escenario se compone del conjunto de modelos de sus transacciones. El modelo de cada transacción se compone de su declaración y su descripción.

Una Transacción se declara mediante un objeto de la clase *Transaction.* Agregadas a cada objeto de la clase *Transaction* se declara la lista con la descripción de las fuentes de eventos externos (*External_Event_Source*) que constituyen su patrón de eventos externos de disparo y la lista de descripción de requisitos temporales (*Timing_Requirement*) definidos en ella. La descripción de una transacción se realiza mediante un modelo de actividad, agregado a su declaración.

En la Figura 15, vemos la definición de una Transacción en MAST, donde se pueden ver las clases que la componen:

- Mover_Trans pertenece al tipo Regular_Transaction, que es una transacción analizable mediante las herramientas MAST.
- InitMover, es un objeto de la clase *Periodic_Event_Source,* la cual es un tipo que desciende de *External_Event_Source,* y representa una secuencia de eventos externos que se originan de manera periódica cada 100 ms.
- FinMover es un objeto de la clase *Hard_Global_Deadline,* la cual desciende de la clase *Timing_Requirement,* y describe un plazo global estricto que delimita las acciones de la transacción a un deadline marcado (50 ms).

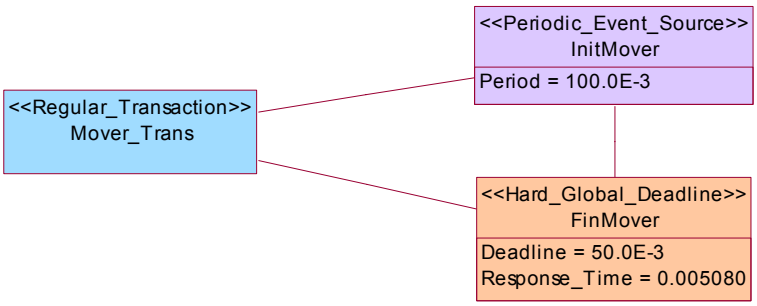


Figura 15

Como se mencionó anteriormente, una transacción se describe con un modelo de actividad (*Activity_Model)* agregado a su declaración. Este modelo de actividad va a estar compuesto por un conjunto de diagramas de actividad que describen las secuencias de actividades, estados y transiciones que se desencadenan como consecuencia de los eventos de entrada.

Dentro de los modelos de actividad vamos a encontrar diferentes modelos de estados. Entre estos estados vamos a poder distinguir varios tipos como son: *Activity_State*, *Wait_State* y *Timed_State.*

Activity_State es un estado de actividad que al ser alcanzado representa que se activa una nueva ejecución de una operación que se declara en su sentencia do/. El estado se abandona cuando se termina la ejecución de la operación, realizándose entonces la transición de salida. Sólo puede contener una sentencia do/. Dado que en general los sistemas que se modelan son concurrentes, puede activarse una Activity antes que haya concluido su ejecución anterior. Cuando esto ocurre, habrá dos instancias de la Activity dispuestas para ser planificadas en el mismo *Scheduling_Server*, las cuales compartirán una misma política de planificación.

Wait_State es un estado de espera tal que al ser alcanzado se suspende la correspondiente línea de flujo de control, a la espera de que se genere un evento

externo o de que se alcance un *Named_State* en alguna sección del modelo de la transacción. Si el *Wait_State* representa una espera a un evento externo, el identificador del *Wait_State* debe ser el mismo que el identificador del *External_Event_Source* que caracteriza la temporización de los eventos externos y que debe haber sido declarado en la transacción. Si el *Wait_State* representa una espera a que el modelo alcance un determinado estado, el identificador del *Wait_State* debe coincidir con el del *Named_State* a cuya espera se suspende. La combinación *Named_State* y *Wait_State* con el mismo identificador es el mecanismo que se dispone para transferir una línea de flujo de control concurrente de una sección de la transacción a otra. Se impone la restricción de que solo puede existir un *Wait_State* por cada *Named_State* definido en el modelo del job o transacción.

Timed_State es un estado temporal, que se declara para asignar un requerimiento temporal (Timing requirement) al estado que representa dentro del job. El nombre del *Timed_State* debe tener un identificador idéntico al del Timing_Requirement que se le asocia, el cual debe estar declarado en la transacción a la que corresponden ambos. Un *Timed_Sate* tiene siempre una transición de entrada y puede tener una o ninguna transición de salida.

Dentro de los modelos de actividad también se van a utilizar los objetos denominados *Swinlane*, los cuales se definen como una banda vertical del diagrama de actividad que representa a cada *Scheduling_Server* que participa en la ejecución del job. Cada actividad debe situarse en la Swimlane que corresponde al *Scheduling_Server* al que se asigna su ejecución. *El Scheduling_Server* en que se ejecutan las actividades declaradas en un trabajo se especifica en la propia descripción del trabajo. Las actividades que en la descripción de un trabajo se asignan a un *Swinlane* sin nombre, se ejecutan en el *Scheduling_Server* en que se encuentra la actividad que invoca al trabajo o transacción.

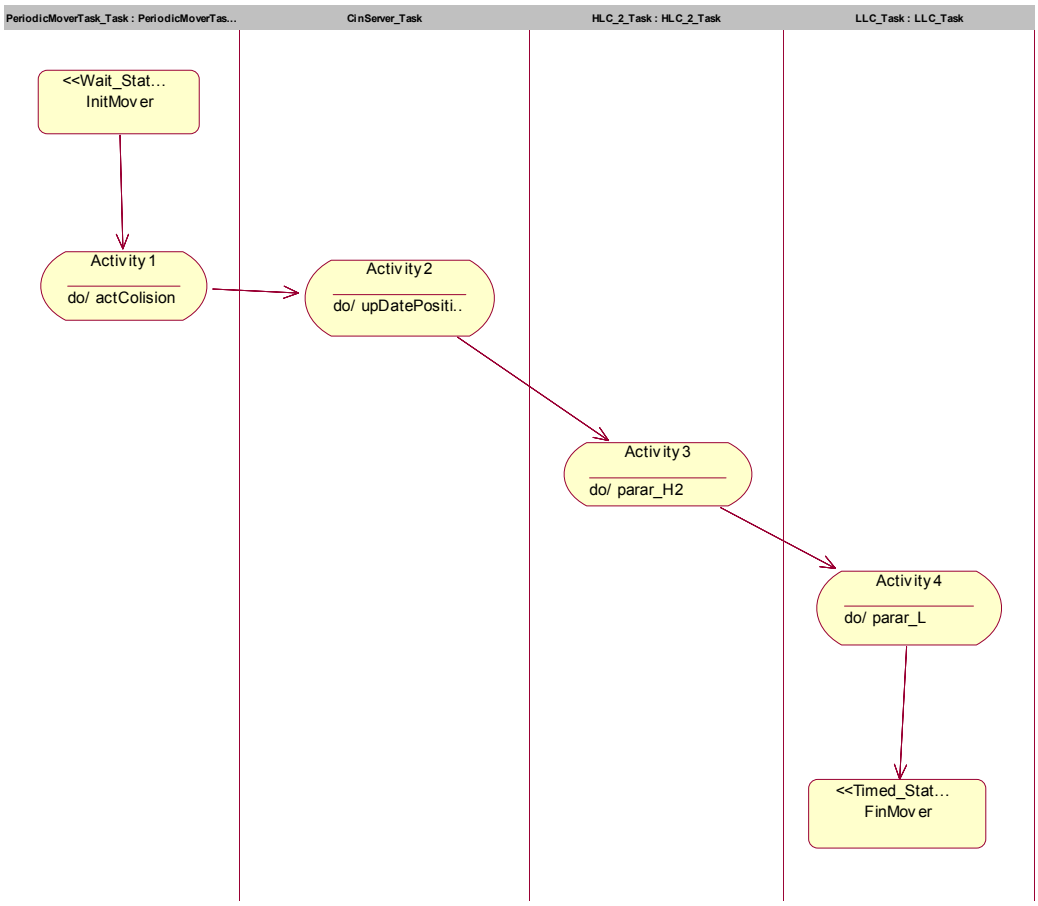


Figura 16

En la Figura 16 se muestra un diagrama de actividad, en el cual se diseña una transacción del sistema podemos ver el *Waite_State* donde comienza la transacción y el *Timed_State* donde finaliza, y los *Activity_State* donde por medio de una secuencia do/ se ejecuta las operaciones. También se indica para cada una el *swinlane* donde se van a ejecutar.

3.3 Análisis del Modelo con UML_MAST.

El análisis se realiza desde la propia herramienta Rose, A continuación se muestra la secuencia de pasos a seguir.

Paso 1:

En primer lugar hay que chequear el modelo. Este chequeo permite verificar la definición de los componentes de la Mast RT View y la consistencia de sus asociaciones que constituyen el modelo de tiempo real. Los resultados del chequeo se proporcionan en una ventana específica “Mast Real-Time View components check”. En la figura 17 se puede observar donde se realiza este chequeo.

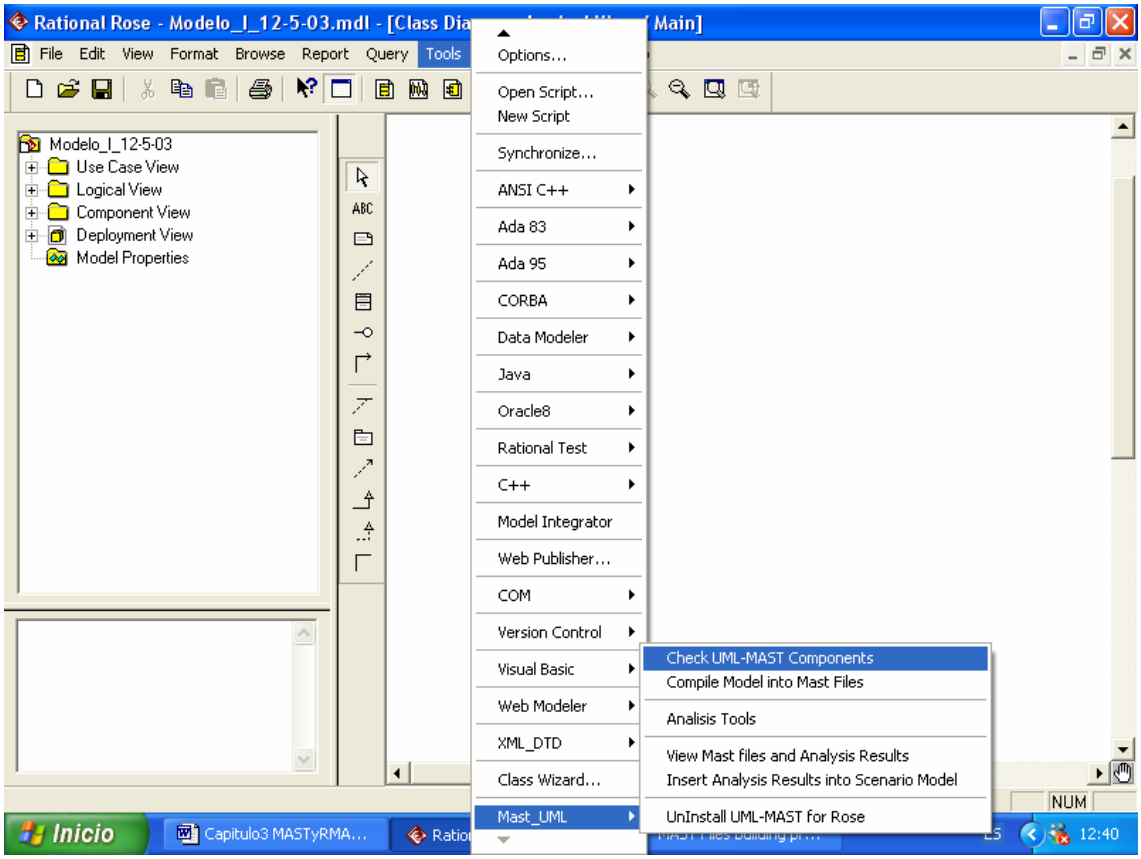


Figura 17

Paso 2:

Compilar la Mast RT View en un modelo Mast formulado mediante su correspondiente fichero textual. Por cada uno de los escenarios del modelo se genera un modelo Mast independiente. La compilación del modelo1 genera en el directorio en el que está almacenado el modelo “modelo1.mdl”, un nuevo directorio con el nombre “modelo1_Mast_Files”. En el nuevo directorio se almacena el fichero de texto MAST resultante de la compilación “modelo1.txt”, que tiene como nombre el del escenario analizado (que en este caso es el único del modelo). Podemos ver en la Figura 18 donde se realiza esta compilación.

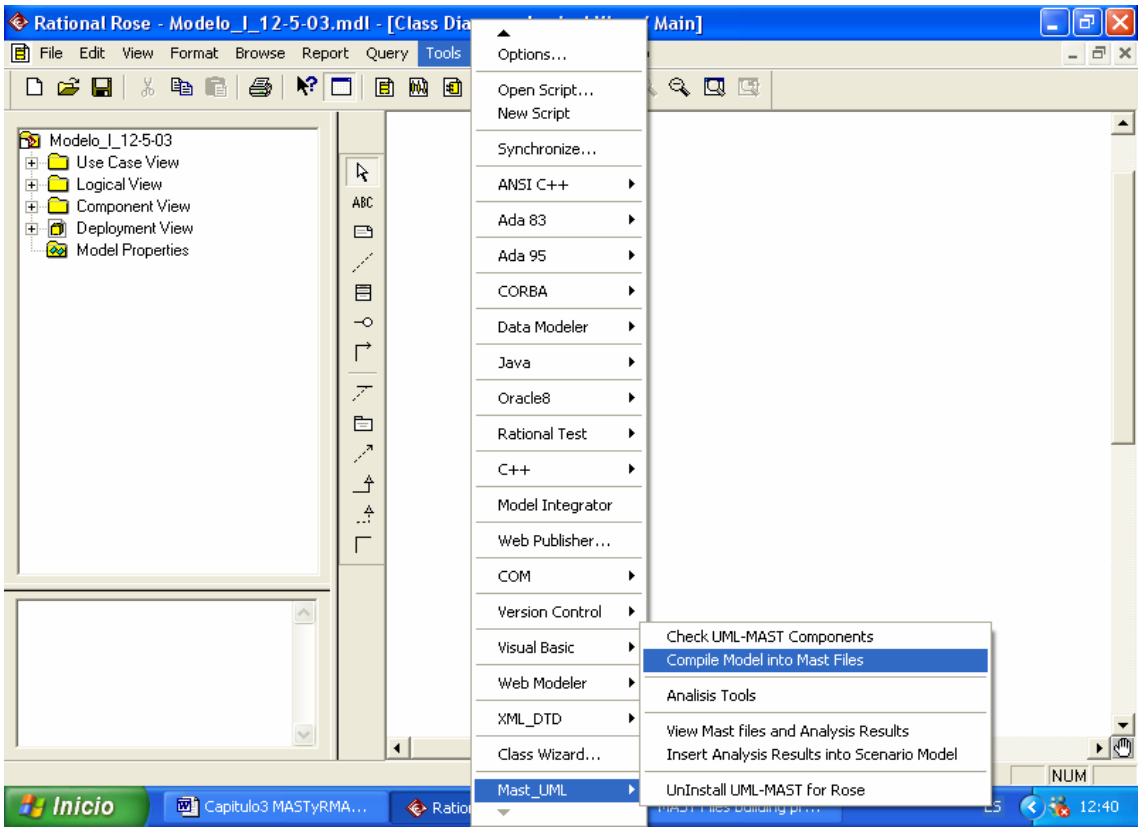


Figura 18

En el caso de que la compilación sea correcta podremos ver la ventana que se aprecia en la figura 19 por pantalla.

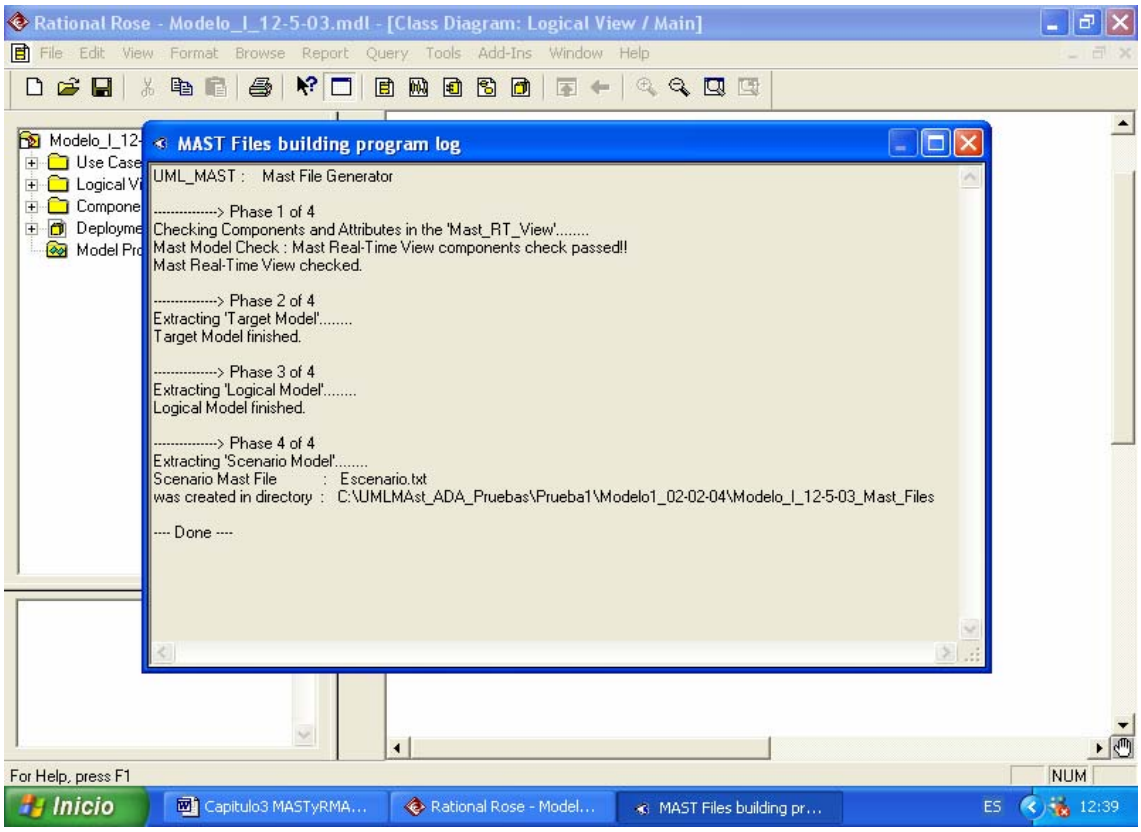


Figura 19

En caso de que la compilación no fuese correcta indicaría esta misma ventana el lugar donde se produce el error.

Paso 3:

El siguiente paso sería acceder a la ventana de invocación de las herramientas del entorno Mast.

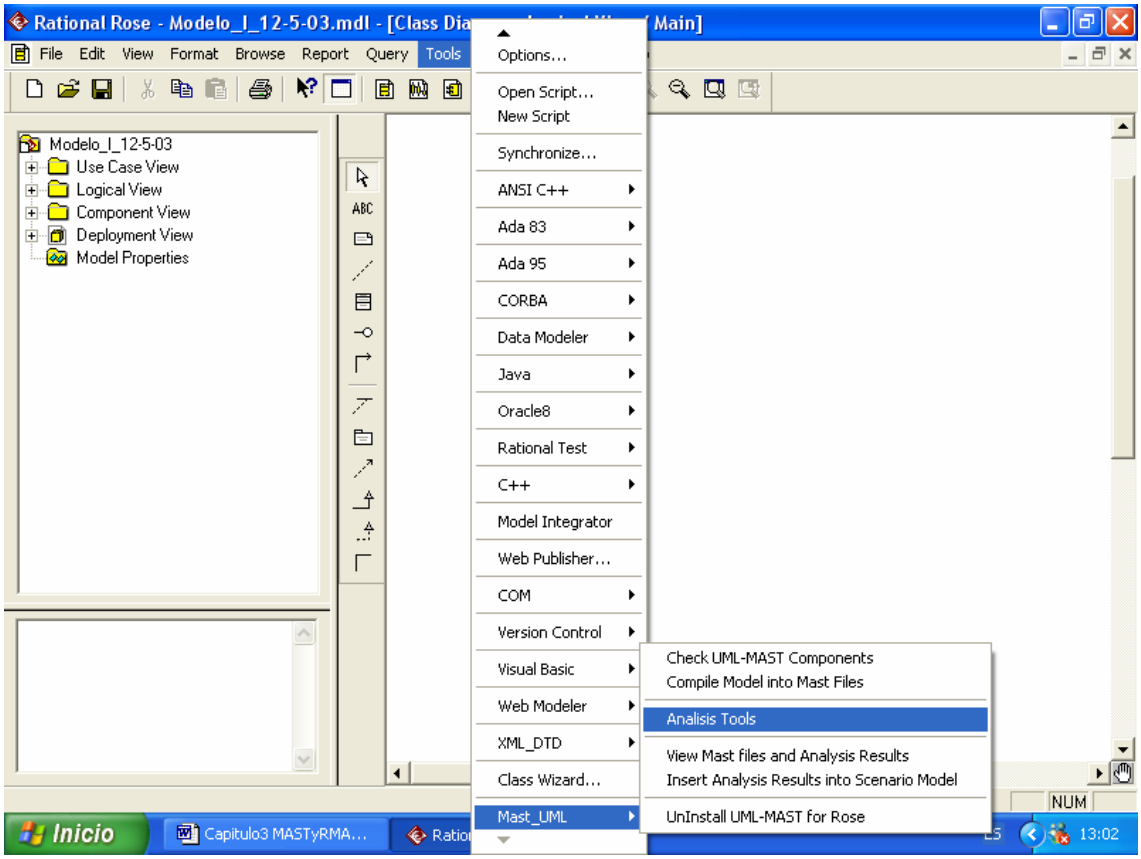


Figura 20

Desde ella se elige el tipo de herramienta, los aspectos que se desean analizar y el directorio y el fichero en que se encuentra el modelo Mast que se analiza. El resultado del análisis se almacena en el fichero modelo1.out.

El análisis del modelo1 genera como resultado el fichero ".\modelo1_Mast_Files\modelo1.out". El análisis nos dirá si el sistema es planificable (si el sistema no fuese planificable lo indicaría y el análisis finalizará), calculará los requisitos temporales del sistema y calculará las holguras, que son el tanto por ciento que se tiene que incrementar uniformemente los tiempos de las operaciones que intervienen en la operación para que deje de ser planificable. En la figura 21 podemos ver la ventana de análisis.

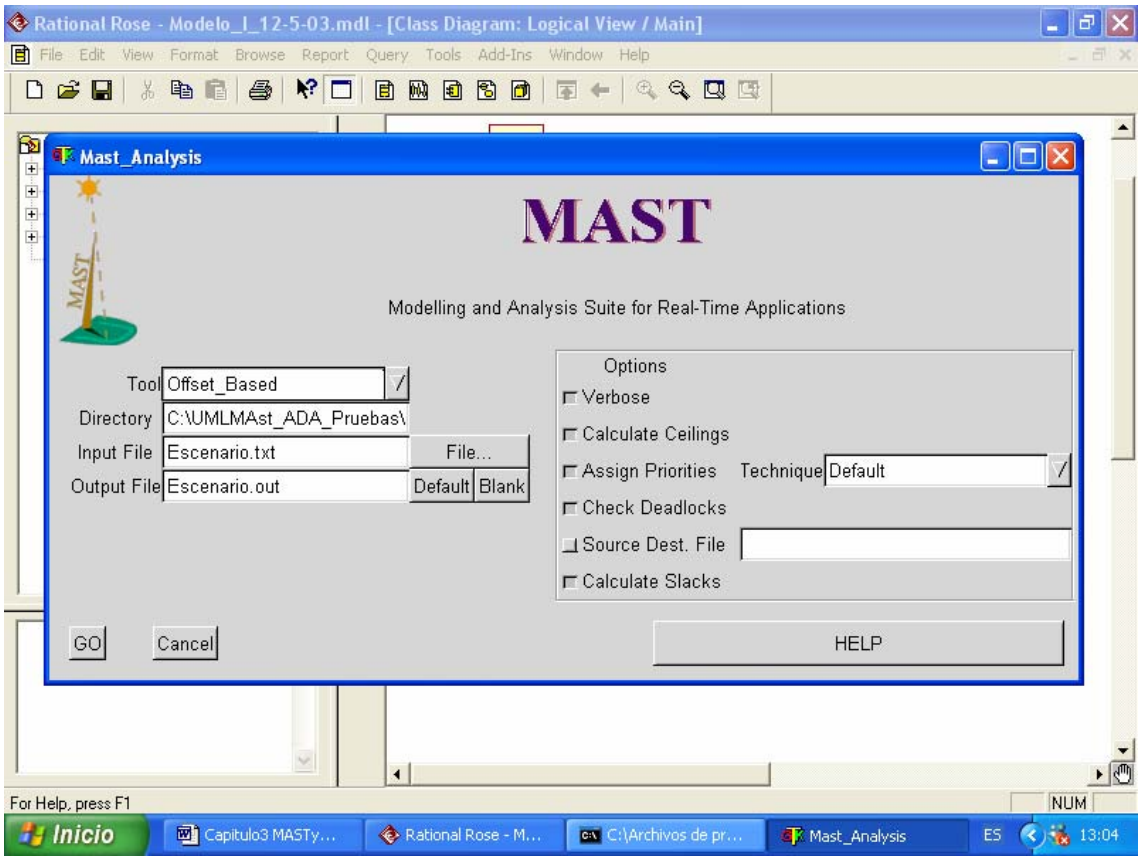


Figura 21

Paso 4:

Por ultimo una vez obtenido el fichero con el resultado del análisis, los resultados mas relevantes del análisis, los tiempos de respuesta de cada transacción, se incorporarán a la MAST_RT_View, en la figura 22 se ve la opción que se debe elegir.

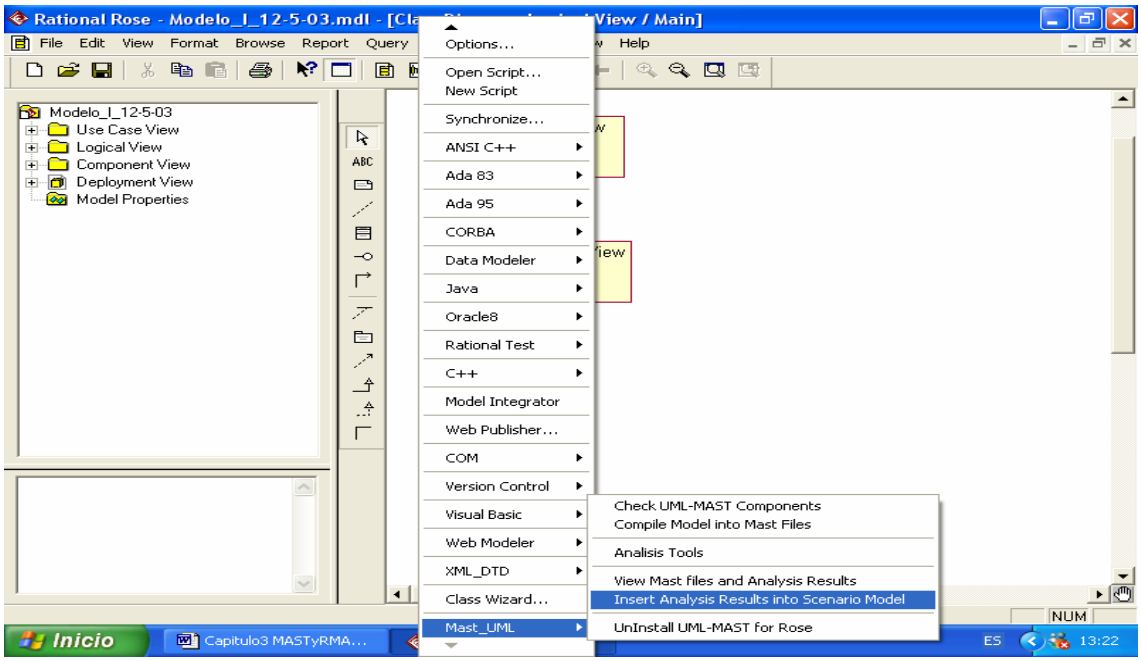


Figura 22

En la figura 23, se ve como han sido insertados estos resultados al modelo. Como se ve en la figura se muestra los tiempos de respuesta (Resoponse_Time) de cada transacción.

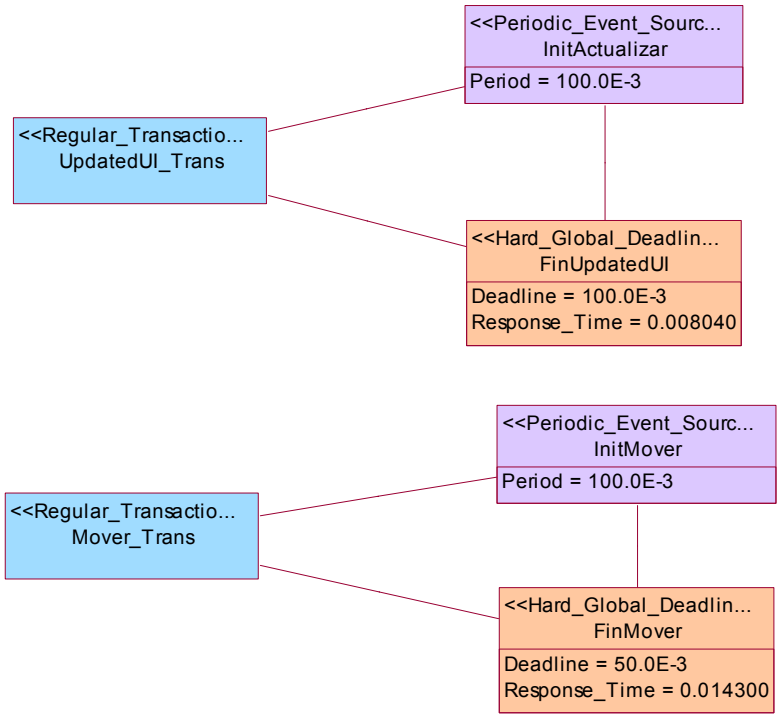


Figura 23

Así concluiría el análisis del modelo MAST que se había diseñado en el cual podremos observar los tiempos de respuesta como se ve en la figura 23, así como el resto de datos accediendo al fichero modelo1.out.

Capítulo 4
Diseño de los modelos

4.1 Introducción

Se han propuesto una amplia serie de modelos con diferentes arquitecturas y componentes para sistemas de teleoperados para someter a estudio su comportamiento temporal. Todos los modelos parten de los mismos requisitos temporales y sometidos a las mismas condiciones de ejecución (un mismo sistema operativo y un mismo algoritmo de asignación de prioridades).

El diseño de los modelos se ha realizado por medio de la herramienta UML-MAST que trabaja sobre el entorno ROSE del que ya se ha hablado en el capítulo anterior.

El tipo de sistema de tiempo real, sobre el que se ha trabajado de manera esquemática es un robot básico, el cual se compone de una interfaz de usuario, un controlador de alto nivel, un controlador de bajo nivel, un servidor de cinemática y lo que seria el robot físico.

Sobre este esquema básico se han modelado dos arquitecturas diferentes con la intención de comparar el comportamiento temporal de ambas.

El diseño de los modelos se ha realizado de una manera incremental, es decir, se ha empezado por el modelo más sencillo y con un número mínimo de operaciones y transacciones básicas. Posteriormente, este modelo ha sido modificado añadiendo nuevos módulos para aumentar la complejidad del mismo, así como comprobar la robustez del sistema al tener que responder ante unas condiciones de trabajo más exigentes, incrementando el número de tareas a ejecutar.

4.2 Prueba1.

4.2.1 Modelo1.

Se trata del modelo básico de diseño. Este modelo se ha diseñado bajo un concepto de arquitectura centralizada, una vez descritos los componentes del sistema.

4.2.1.1 Vista Lógica.

Esta vista debe ser introducida por el diseñador con el fin de facilitar la comprensión del trabajo realizado para el diseño del sistema. Dicha vista no será tenida en cuenta por MAST a la hora del análisis temporal.

En la figura 1, podemos ver el diagrama UML del sistema que representa el Modelo1.

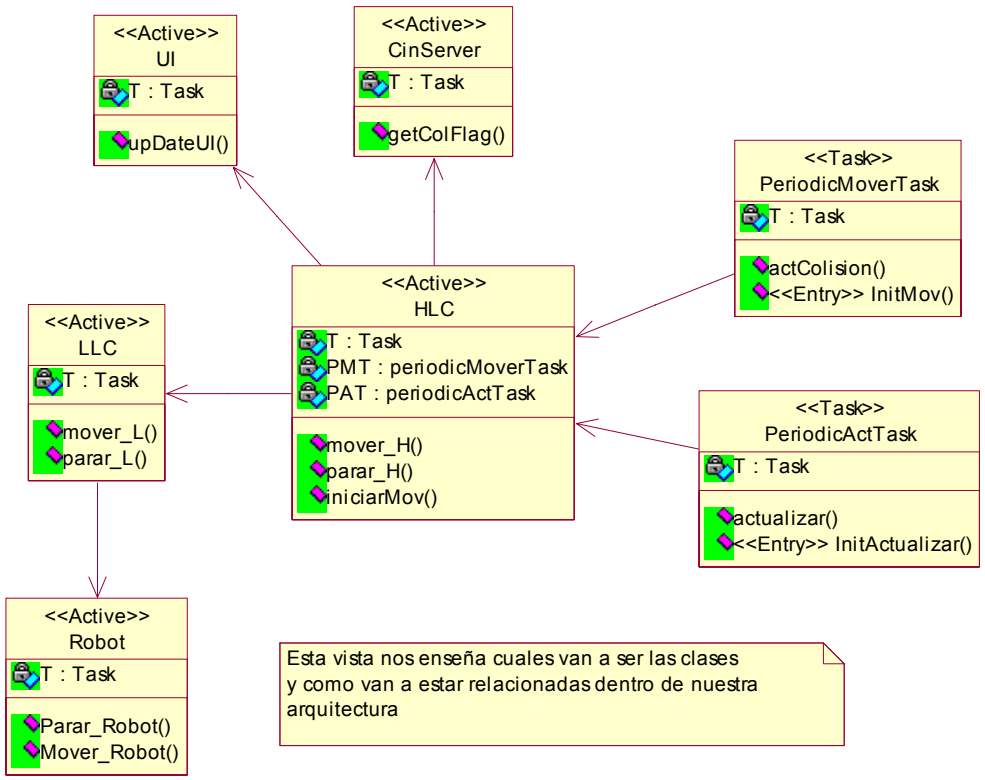


Figura 1

Cada una de estas clases representa un objeto de la realidad con una funcionalidad concreta:

- **UI**: Interfaz de Usuario (*User Interface*) Se trata de la parte de la aplicación accesible para el usuario, desde la cual éste se puede comunicar con el robot. Esta parte será un entorno gráfico mediante el cual se podrá controlar el robot.
- **Robot**: Es una manera de señalar la existencia de un ente físico de manera representativa en este modelo.

- **CinServer**: Servidor de Cinemática. Se encarga de detectar a partir de un modelo del entorno físico la posible existencia de obstáculos para el avance del robot. En caso de presencia de un obstáculo, el servidor se encargará de la activación de un *flag* que informará al controlador de alto nivel (*HLC*) del obstáculo encontrado para que éste reaccione parando el robot.
- **LLC**: Controlador de bajo nivel (*Low Level Controller*). Se va a encarga de controlar las operaciones de bajo nivel. Se podría decir que se encarga de comunicar la parte software con la parte hardware del sistema. Podría verse más que como un controlador, como un traductor de operaciones.
- **HLC**: Controlador de Alto Nivel (*High Level Controller*). Se puede definir como el objeto encargado de controlar las operaciones del Robot a alto nivel, es decir, las operaciones a nivel Software. Comunicará la parte de la aplicación de usuario con la parte física del Robot. Controlará al servidor de cinemática vigilando periódicamente el estado del *flag* que indica la existencia de obstáculo en el camino del Robot. Si este *flag* esta activado, el HLC se encargará de ordenar la detención del Robot.

Hay que señalar que dentro de cada clase como se puede observar en la Figura 1, hay definido un atributo como *Task*, que representa la existencia de una tarea o hilo de ejecución (*Thread*) perteneciente a cada objeto y que será el que ejecute las diferentes acciones u operaciones del sistema.

También hay que destacar la existencia de dos atributos dentro de la clase *HLC* que son *PeriodicActTask* y *PeriodicMoverTask* que, como luego se verá, serán tareas críticas, ya que el funcionamiento del sistema dependerá de que estas cumplan los requisitos temporales especificados para ellas.

Dentro de esta vista también se definen los diagramas de secuencia, en los cuales se especifica la secuencia de acciones que se realiza durante las transacciones que se implementarán en MAST. Hay que tener en cuenta que estos diagramas de secuencia tampoco serán considerados por MAST, simplemente se trata de una ayuda para entender el modelo.

Los diagramas de secuencia de las Figuras 2 y 3 la secuencia de llamadas muestran para movimientos del robot y actualización del estado de la interfaz de usuario.

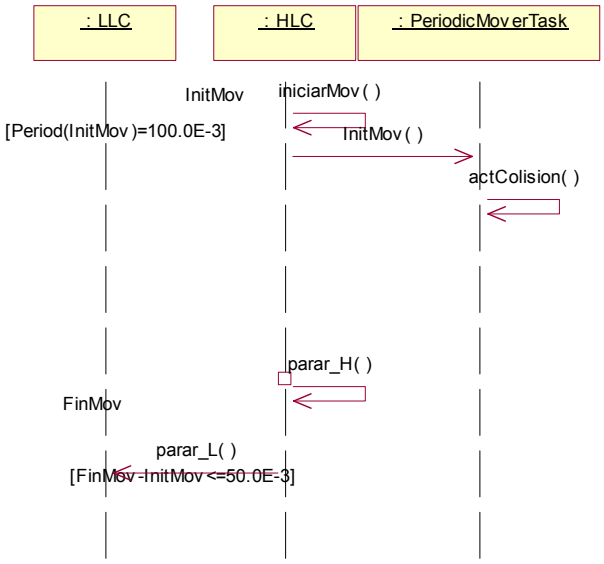


Figura 2 Diagrama de secuencia de Mover_Process

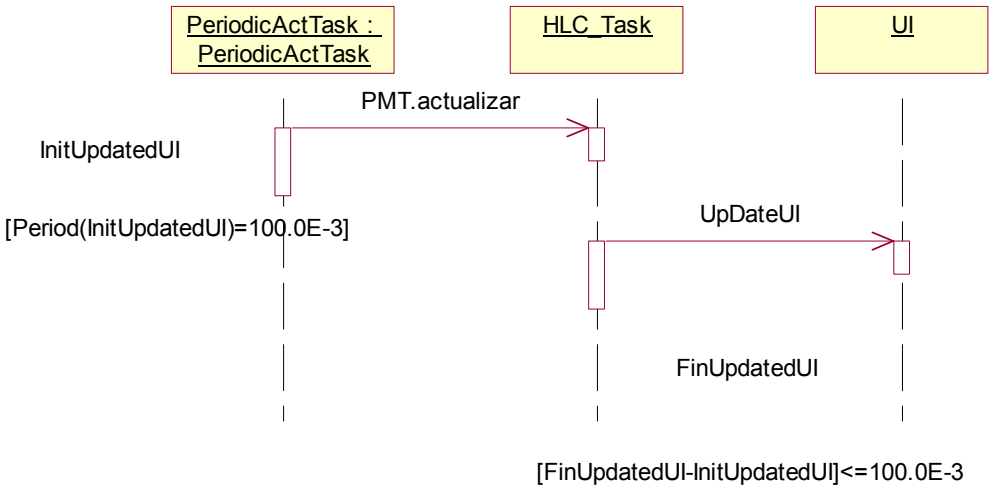


Figura 3 Diagrama de secuencia de UpDateUI_Process

4.2.1.2 Vista de tiempo real en MAST.

Esta es la vista que define el modelo de tiempo real deseado, que como ha sido explicado en el apartado de MAST, se compone de tres vistas:

- **RT_Target_Model** (Modelo de tiempo real de la plataforma), describe la capacidad de procesamiento de la plataforma hardware y software sobre la que se ejecuta la aplicación.

- **RT_Logical_Model** (Modelo de tiempo real de los componentes lógicos), describen la temporización y los requisitos de procesamiento que requieren los componentes lógicos que constituyen la aplicación.
- **RT_Scenarios_Model** (Modelo de los escenarios de tiempo real), describe la secuencia de eventos externos o temporización que conducen las transacciones que constituyen la aplicación, las secuencias de activación que se desencadenan, y los requisitos temporales que establece la especificación de tiempo real.

4.2.1.2.1 Modelo de tiempo real de la plataforma.

En la figura 4, se muestra el modelo de tiempo real de la plataforma.

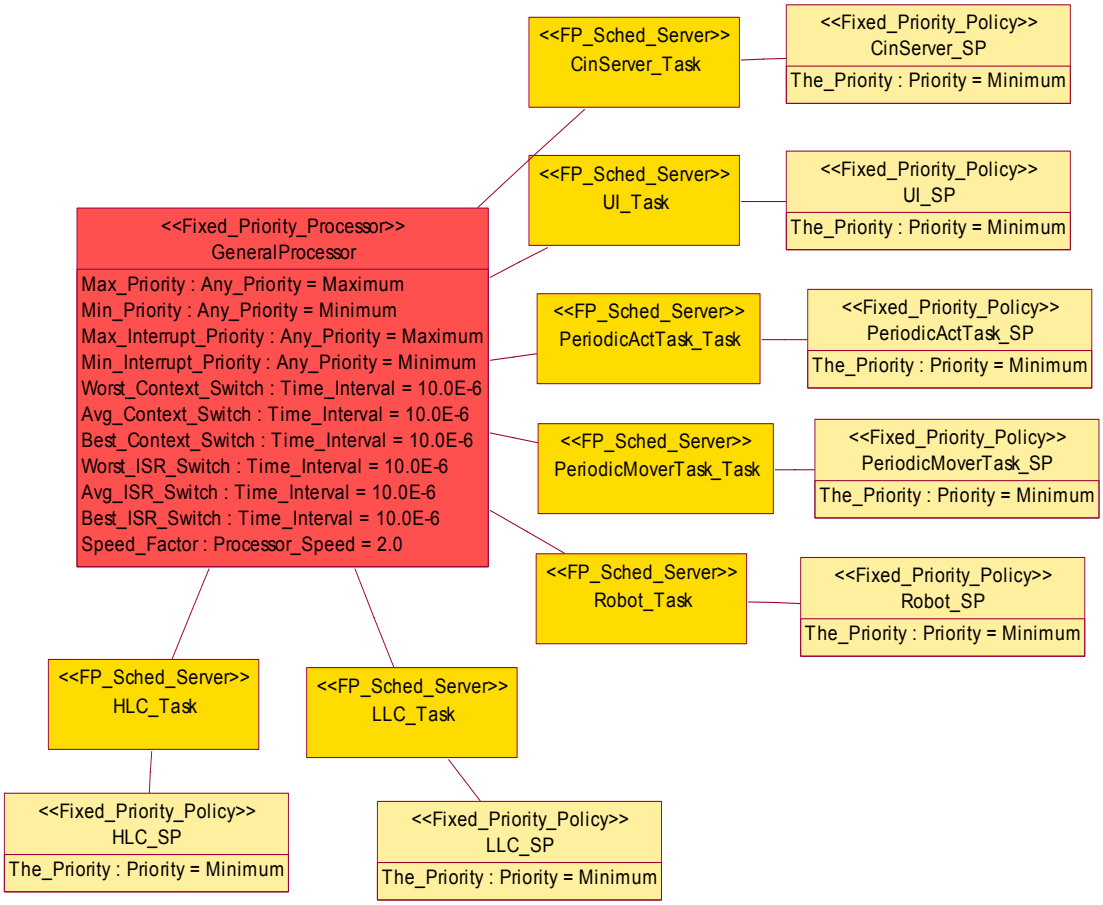


Figura 4

GeneralProcessor representa una plataforma de tipo PC sobre la que opera el sistema operativo. Todos los procesos que se ejecutan en él se hacen bajo un rango de prioridades de tiempo real para lo que se utiliza una planificación basada en prioridades fijas (Fixed_Priority_Processor).

Como se puede ver en la Figura 4 en MAST hay que definir una serie de atributos, que definirán las características del procesador:

- **Speed_Factor:=5.0**, este valor nos dice que esta aplicación debe ser ejecutada sobre un procesador de una capacidad al menos cinco veces superior al que se utilizó para probar el modelo MAST
- **Max_Priority:=Maximun**, dejando el valor *maximun* como valor de la máxima prioridad para el sistema dejamos que la herramienta sea quien le asigne este valor durante el análisis según el algoritmo que se le indique, el cual en todos los modelos que se estudiaran será RMA.
- **Min_Priority:=Minimun**, aquí se indicará el valor mínimo de prioridad con el que trabajará el sistema. De nuevo dejaremos que sea la herramienta MAST quien se lo asigne durante el análisis.
- **Max_Interrupt_Priority : Any_Priority = Maximum**, En los sistemas de tipo PC, las prioridades de interrupción son establecidas por la programación del dispositivo hardware PIC (*Programable Interrupt Controller*) y deben ser mapeadas a las prioridades del modelo MAST que son globales para todo el sistema. En este caso no existe ningún tipo de interrupción en el sistema, por eso se deja un valor por defecto que asignará la herramienta como valor máximo.
- **Min_Interrupt_Priority : Any_Priority = Minimum**, es el mismo caso que el párrafo anterior pero en esta caso se deja que la herramienta le asigne el valor mínimo.
- **Worst_Context_Switch**:=10.0E-6, el cambio de contexto entre dos *threads* del procesador se realiza en un tiempo inferior a 10 µs.
- **Avg_Context_Switch:=10.0E-6**, el cambio de contexto entre dos *threads* del procesador se realiza con un tiempo promedio de 10 µs.
- **Best_Context_Switch:=10.0E-6**, el cambio de contexto entre dos *threads* del procesador se realiza siempre en un tiempo igual o superior a 10 µs.
- **Worst_ISR_Switch:=10.0E-6**, el tiempo de cambio de contexto desde un thread de la aplicación a la rutina de atención de una interrupción se realiza en un tiempo inferior a 10 µs.
- **Avg_ISR_Switch:=10.0E-6**, el tiempo de cambio de contexto desde un *thread* de la aplicación a la rutina de atención de una interrupción se realiza con un tiempo promedio de 10 µs.
- **Best_ISR_Switch:=10.0E-6**, el tiempo de cambio de contexto desde un *thread* de la aplicación a la rutina de atención de una interrupción se realiza siempre en un tiempo igual o superior a 10 µs.

El procesador debe llevar asociados a él la representación de los *threads* de la aplicación, así como los servidores que planifican sus actividades, que en este caso seguirán una política fija.

- **UI_Task**: *Thread* que introduce la instancia de la clase UI. Este *thread* se encarga de controlar los eventos que ocurran sobre la interfaz de usuario.
- **UI_SP**: Es el *server* que planifica las actividades del UI_Task con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **CinServer_Task**: *Thread* que introduce la instancia de la clase *CinServer*. Este *thread* se encarga de controlar los eventos que ocurran sobre el servidor de cinemática.

- **CinServer_SP**: Es el *server* que planifica las actividades del *Cinserver_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **Robot_Task**: *Thread* que introduce la instancia de la clase Robot. Este thread se encarga de controlar los eventos que ocurran sobre el Robot.
- **Robot_SP**: Es el *server* que planifica las actividades del *Robot_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **LLC_Task**: *Thread* que introduce la instancia de la clase LLC. Este thread se encarga de controlar los eventos que ocurran sobre el LLC.
- **LLC_SP**: Es el *server* que planifica las actividades del *LLC_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **HLC_Task**: *Thread* que introduce la instancia de la clase HLC. Este *thread* se encarga de controlar los eventos que ocurran sobre el HLC.
- **HLC_SP**: Es el *server* que planifica las actividades del *HLC_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **PeriodicActTask_Task**: *Thread* que introduce la instancia de la clase *PeriodicActTask*. Este thread se encarga de controlar los eventos que ocurran sobre el *PeriodicActTask*.
- **PeriodicActTask_SP**: Es el *server* que planifica las actividades del *PeriodicActTask_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **PeriodicMoverTask_Task**: *Thread* que introduce la instancia de la clase *PeriodicMoverTask*. Este *thread* se encarga de controlar los eventos que ocurran sobre el *PeriodicMoverTask*.
- **PeriodicMoverTask_SP**: Es el *server* que planifica las actividades del *PeriodicMoverTask_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.

Queda así definida la plataforma en MAST sobre la cual se ejecutará la aplicación.

4.2.1.2.2 Modelo de tiempo real de los componentes lógicos.

El modelo de tiempo real de los componentes lógicos modela el comportamiento temporal de los componentes funcionales (clases, métodos, procedimientos, operaciones, etc.) que están definidos en el sistema y cuyos tiempos de ejecución condicionan el cumplimiento de los requisitos temporales definidos en los escenarios de tiempo real que van a analizarse. El modelo de cada componente lógico describe los dos aspectos que condicionan su tiempo de ejecución: el tiempo que requiere la ejecución de su código, que es función de la complejidad de los algoritmos que contiene y los bloqueos que puede sufrir su ejecución como consecuencia de que necesita acceder en régimen exclusivo a recursos que también son requeridos por otros componentes lógicos que se ejecutan concurrentemente con él.

El modelo de tiempo real de los componentes lógicos se establece con las siguientes características:

- Los tiempos de ejecución de las operaciones se definen normalizados, ésto es, se formulan con parámetros que son independientes de la plataforma en que se van a ejecutar.
- El modelo de interacción entre componentes lógicos se expresa de forma parametrizada, identificando los recursos que son potenciales causas de bloqueos (*Shared_Resource*) y dejando hasta la descripción del escenario la declaración de los componentes lógicos concretos con los que va a interferir.
- El modelo de tiempo real de los componentes lógicos, se formula con una modularidad paralela a la modularidad que en la vista lógica ofrecen los componentes lógicos que se modelan.
- En los siguientes diagramas se describen los modelos de tiempo real de los componentes lógicos de esta aplicación. A efecto meramente explicativo, se adjunta a cada diagrama del *RT_Logical_Model* la clase lógica que se modela en el diagrama.
- Cabria destacar que en este modelo lógico a la hora de asignar los tiempos de duración de cada operación sólo vamos a tener en cuenta el peor tiempo, por eso todos los tiempos de ejecución de cada operación van a ser idénticos.

UI_Model

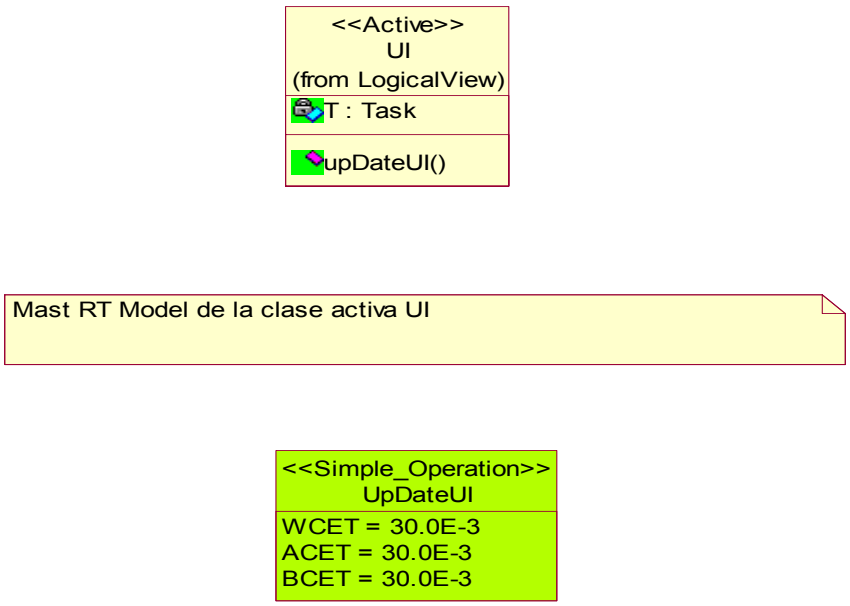


Figura 5

UpDateUI es un modelo de operación simple con un tiempo de ejecución de 30ms. Esta operación será utilizada para actualizar la interfaz de usuario.

Cinserver_Model

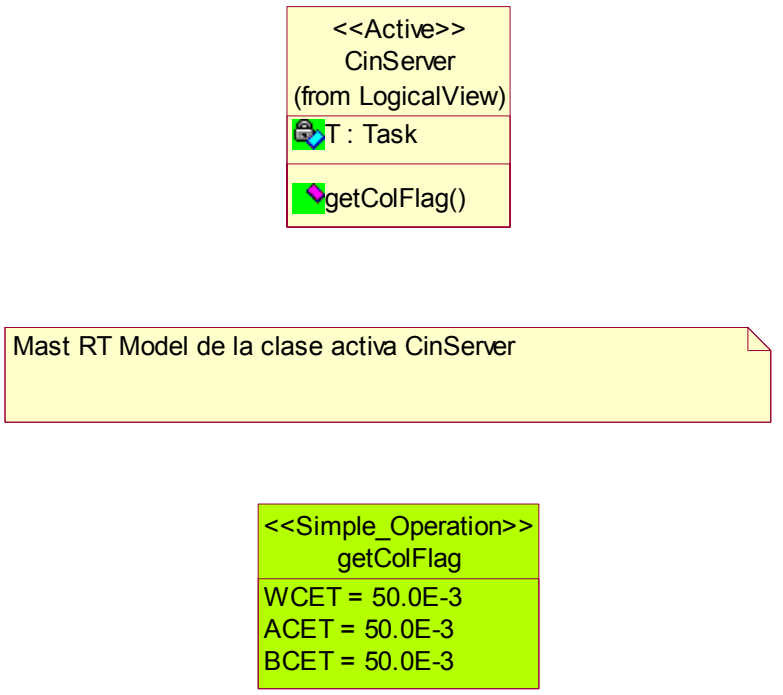


Figura 6

getColFlag es un modelo de operación simple con un tiempo de ejecución de 50ms. Es un tiempo de ejecución alto debido a que se trata de una operación de acceso a un *flag* del servidor de cinemática, esta operación permitirá conocer el estado del *flag* de colisión.

Robot_Model.

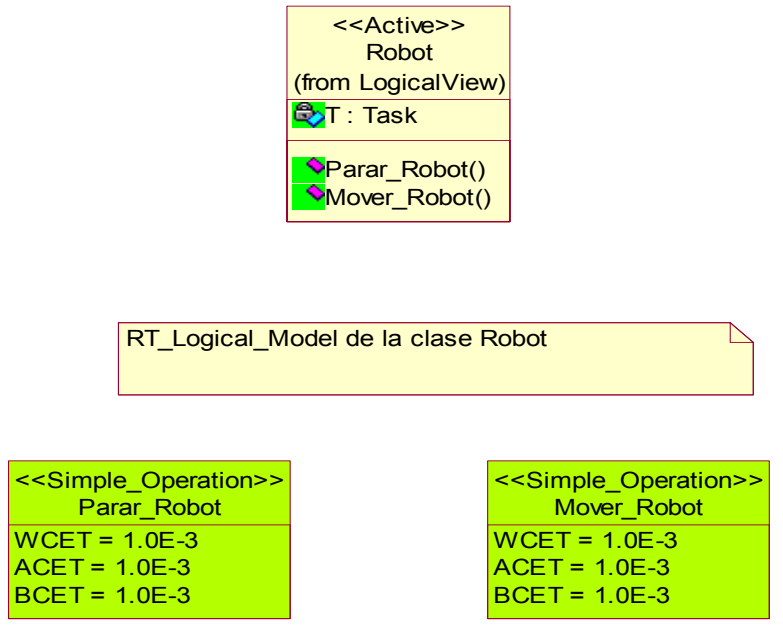


Figura 7

Parar_Robot es un modelo de operación simple con un tiempo de ejecución de 1ms. Esta operación tiene un tiempo de ejecución tan pequeño debido a que modula una operación de comunicación con el hardware con la cual enviaríamos el mensaje al robot que debe detenerse.

Mover_Robot es un modelo de operación simple con un tiempo de ejecución de 1ms. Esta operación tiene un tiempo de ejecución tan pequeño debido a que modula una operación de comunicación con el hardware con la cual enviaríamos el mensaje al robot que debe comenzar a moverse.

LLC_Model

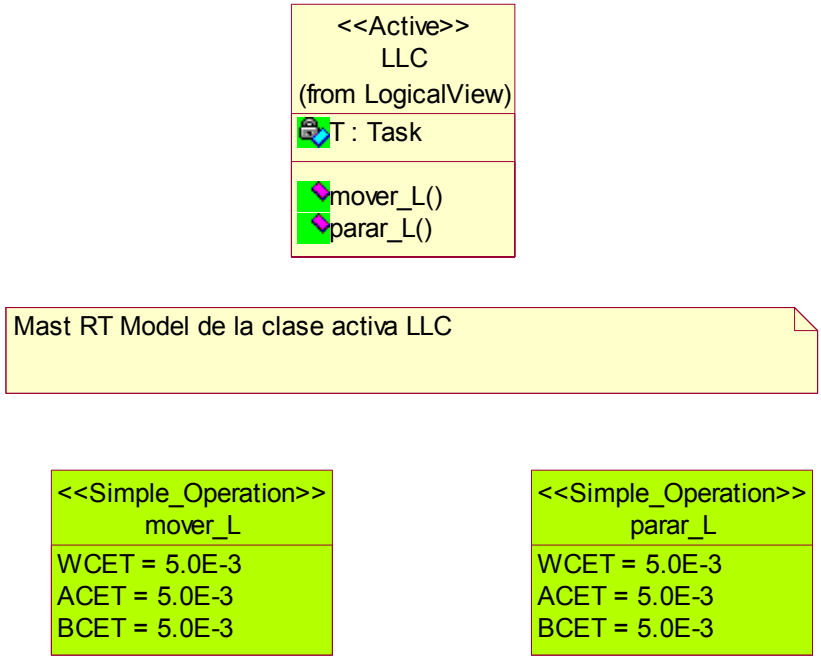


Figura 8

parar_L es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de detenerlo.

mover_L es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de comenzar a moverse.

HLC_Model

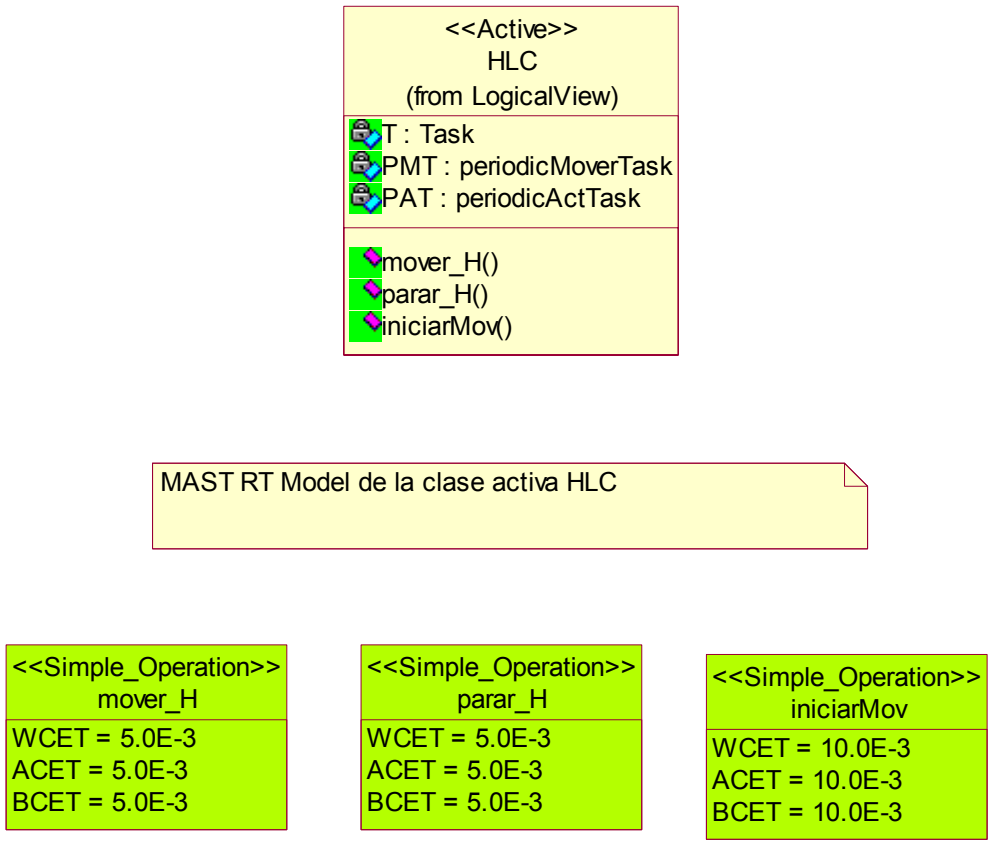


Figura 9

parar_H es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de detenerlo.

mover_H es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de comenzará a moverse.

IniciarMov es un modelo de operación simple con un tiempo de ejecución de 10 ms. Esta operación se va a encargar de iniciar las tareas que debe realizar el robot así como de controlarlas, iniciara *PeriodicActTask* y *PeriodicMoverTask*, iniciando así el movimiento

PeriodicActTask_Model

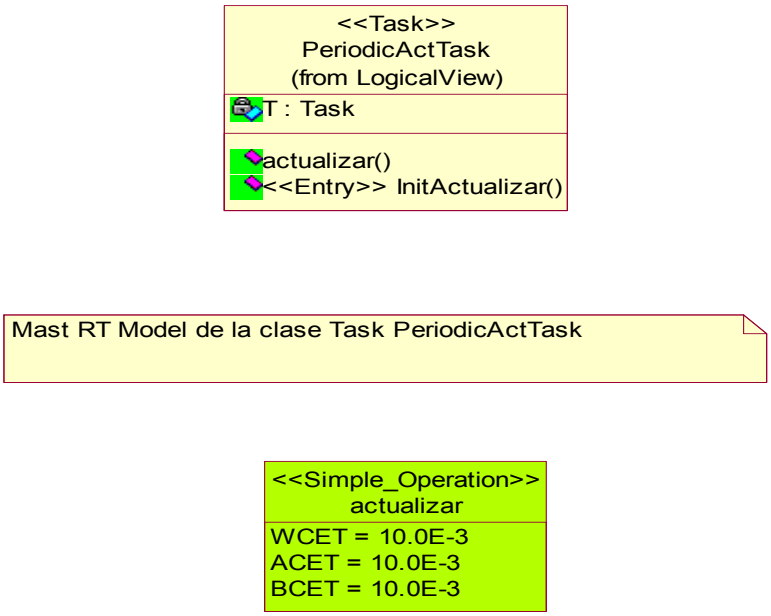


Figura 10

actualizar es un modelo de operación simple con una duración de 10ms. Esta operación está definida dentro de esta tarea la cual en su hilo de ejecución la llama de manera periódica. Esta operación se encargará de recoger datos para actualizarlos posteriormente en la interfaz de usuario.

PeriodicMoverTask

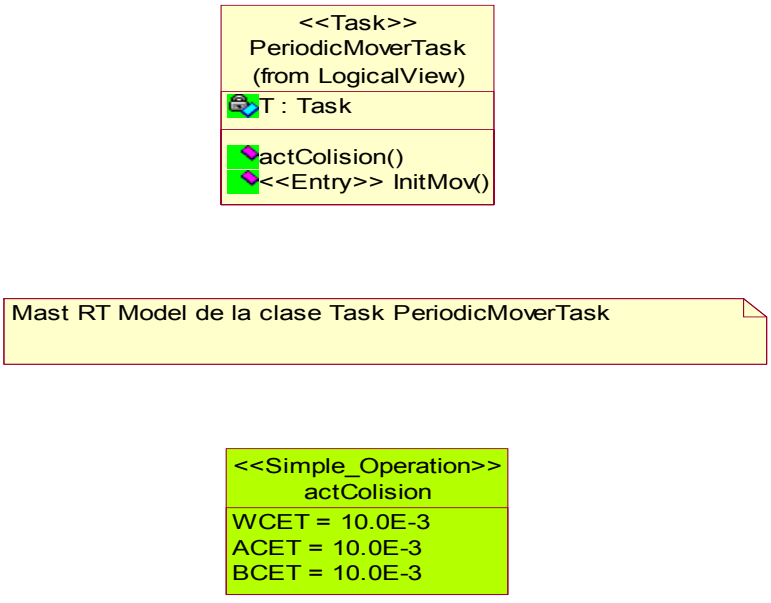


Figura 11

actColision es un modelo de operación simple con una duración de 10ms. Esta operación esta definida dentro de esta tarea, la cual en su hilo de ejecución la llama de manera periódica. Esta operación se encargara de recoger los datos del servidor de cinemática en particular el estado del *flag* de colisión para actualizarlo en el controlador de alto nivel.

4.2.1.2.3 Escenarios de tiempo real.

Contiene los escenarios de tiempo real que son los modos o configuraciones de operación hardware/software para los que el sistema tiene definidos requerimientos de tiempo real. En este sistema sólo hay definido un escenario de tiempo real denominado Escenario.

Esta carpeta contiene el único escenario del sistema en el que se trata del control realizado por el HLC del sistema, en concreto de la actualización de la interfaz grafica, así como de la rutina de la actualización del estado del *flag* de colisión y de la reacción del sistema según el valor de éste.

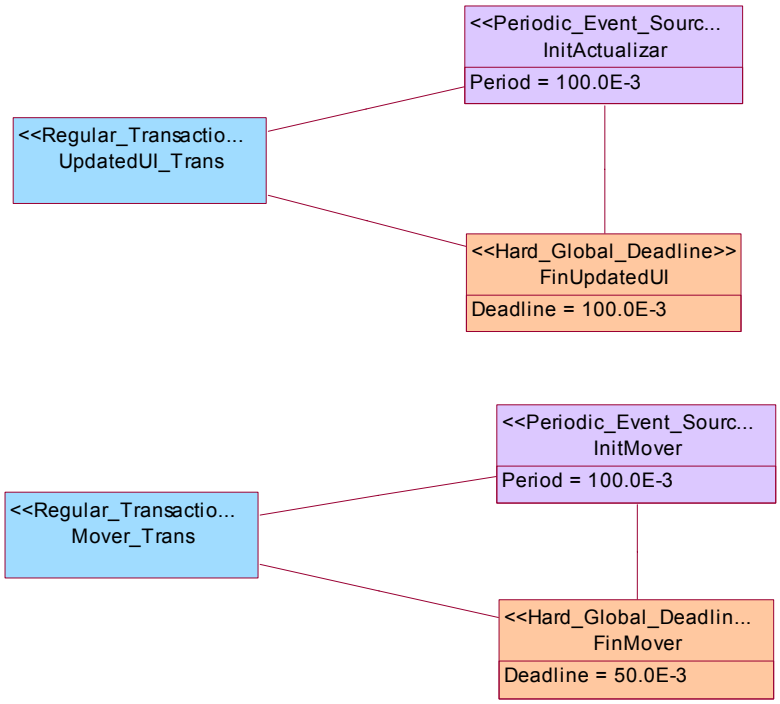


Figura 12

La figura 12 muestra dos transacciones que componen el escenario desarrollado para este modelo. Hay que tener en cuenta que estas dos transacciones son críticas ya que si una de ellas no cumpliera los requisitos temporales especificados, el sistema fallaría.

UpDateUI_Transaction.

Esta transacción se va a encargar de realizar una actualización periódica de la interfaz de usuario. Se realizará cada 100ms. Se ha considerado que este tiempo es adecuado para que el usuario del robot pueda ver los cambios efectuados en la interfaz.

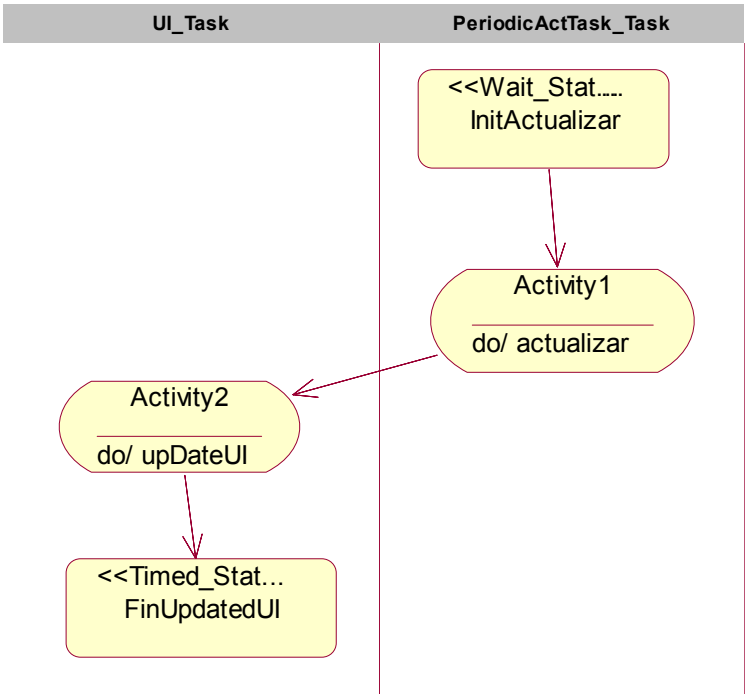


Figura 13

En esta transacción se van a actualizar periódicamente los datos de la interfaz de usuario. Será iniciada desde la tarea PariodicActTask desde un Wait_State, esto indica que la tarea estará a la espera de una señal de activación. En cuanto se produzca esta señal, se le enviaran los datos de actualización a la tarea que se encuentra en la interfaz de usuario, la cual se actualizará de manera automática con estos datos por medio de la sentencia UpDateUI. Por ultimo, se llega a un Timed_State, dándole un tiempo limite (100ms) para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo de sistema.

Esta transacción va a ser común a todos los modelos que se expondrán en este trabajo. Por ello, en el resto de los modelos sólo se mencionará, sin volver a explicarla.

Mover_Transaction.

Esta transacción se va a encargar de controlar la posibilidad de colisión durante el movimiento del robot, es decir, controlará la posibilidad de un obstáculo que imposibilite la continuidad del movimiento del robot por lo cual deberá ordenar la detención del mismo.

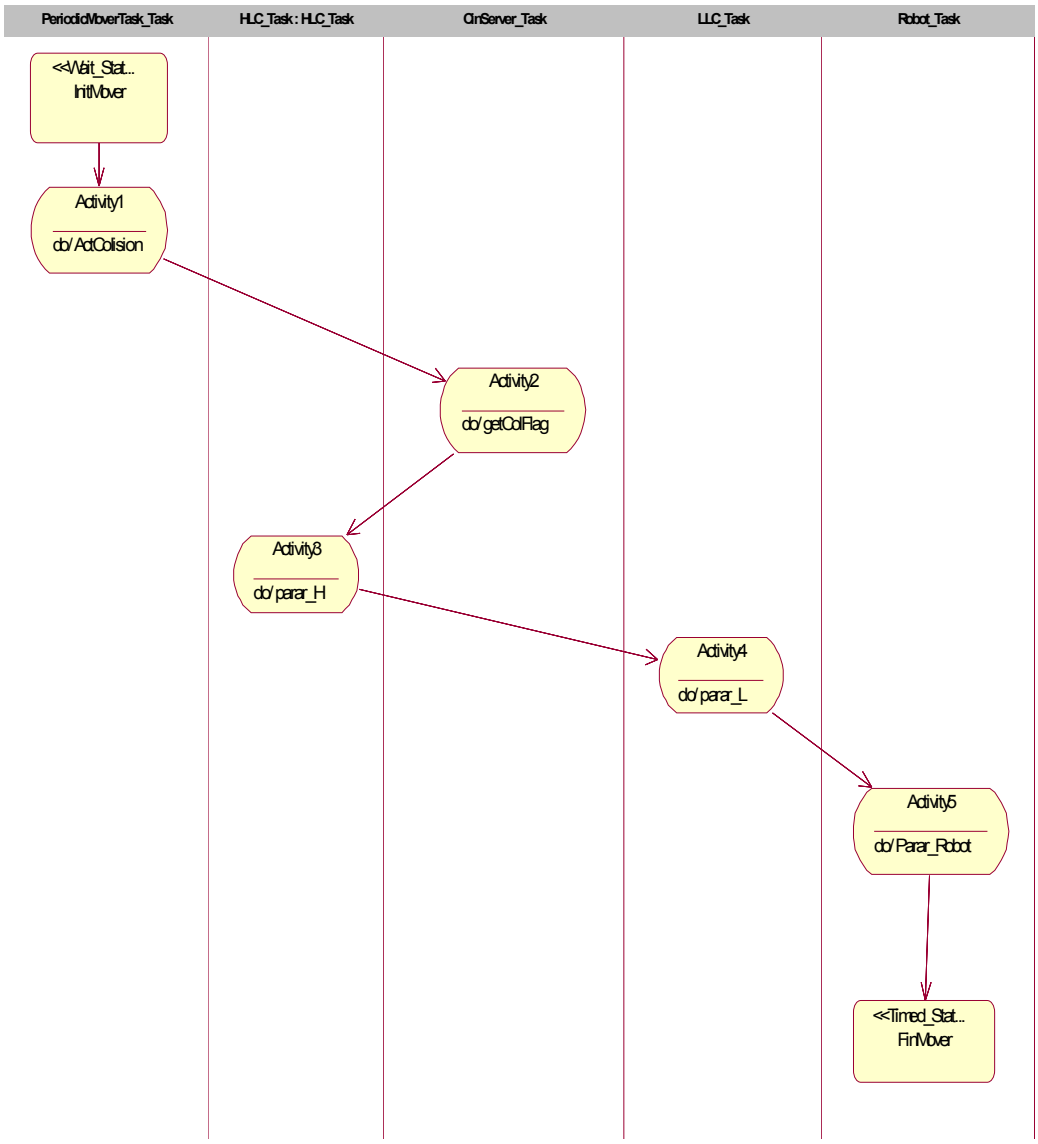


Figura 14

En esta transacción se va a controlar periódicamente el estado del *flag* de colisión. En caso de estar activado indicará la posibilidad de que se produzca una colisión por lo cual el sistema debe de actuar deteniéndose. Será iniciada desde la tarea *PeriodicMoverTask* desde un *Wait_State*, cuando se produzca una señal de activación. En cuanto se produzca esta señal, el sistema empezará el proceso de comprobar el estado del *flag* de colisión. En primer lugar conseguirá el estado del *flag*, una vez obtenido comprobará si está activado, en cuyo caso ordenará al sistema que debe detenerse. Por último, se llega a un *Timed_State.* Este estado espera una señal que será creada por el *Hard_Global_Deadline*, en este caso cada 50ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin de evitar el bloqueo del sistema.

Es importante mencionar que debido a las limitaciones aun existentes en MAST en este trabajo solo hemos tenido en cuenta el peor caso posible, refiriéndose con esto al caso más crítico para el sistema.

4.3 Prueba2

4.3.1 Modelo2

4.3.1.1 Vista lógica.

En la figura 15, se puede ver el diagrama de clases de este modelo en el cual podemos observar una pequeña diferencia respecto al primer modelo.

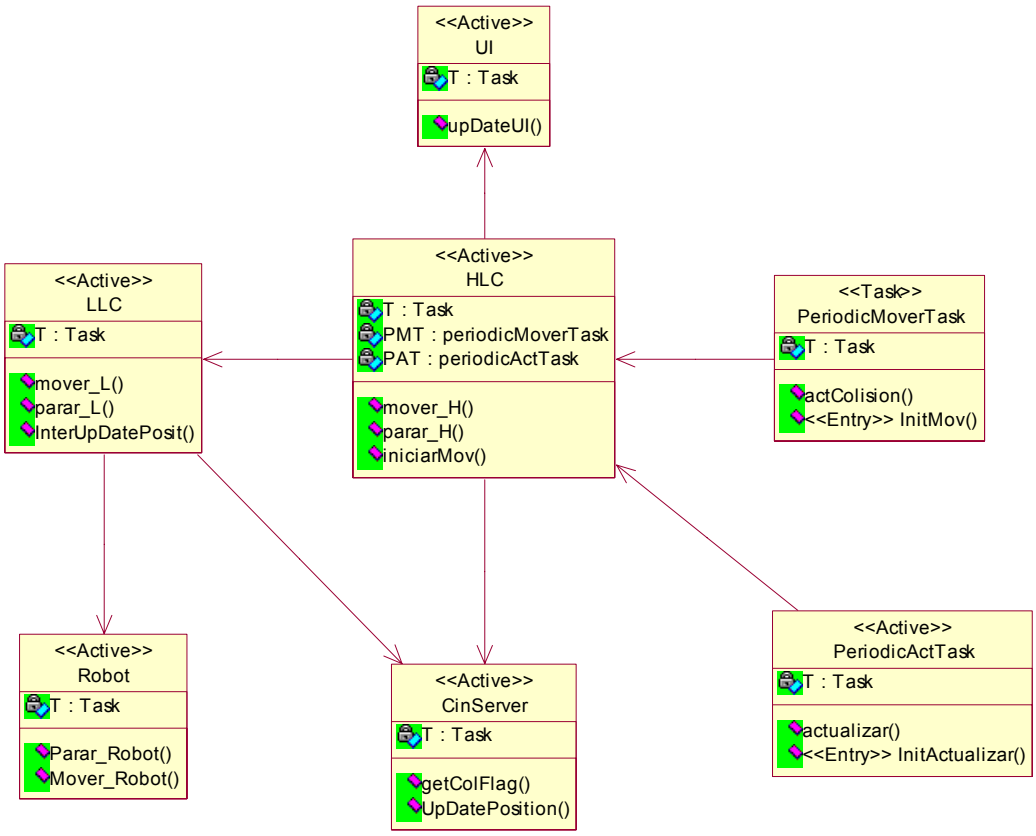


Figura 15

Se mantienen las mismas clases que en el modelo anterior y con una funcionalidad similar. En el primer modelo los datos los recogía el controlador de bajo nivel (*LLC*) y los hacia llegar hasta el servidor de cinemática (*CinServer*) a través del Controlador de alto nivel (*HLC*). Por el contrario, en este modelo no existe el paso intermedio por el HLC, es decir, el LLC le pasa los datos directamente al servidor de cinemática. Esta diferencia se puede observar más claramente en los diagramas de secuencia y posteriormente en la transacción *Mover_Transaction.*

En las figuras 16 y 17, se pueden observar los diagramas de secuencia correspondientes a este Modelo2.

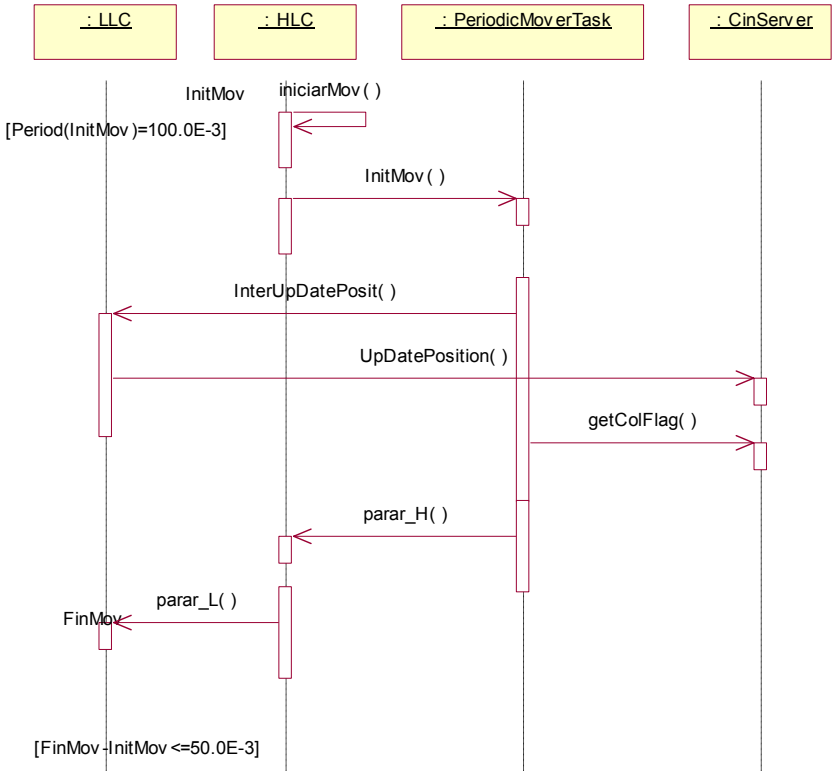


Figura 16 Diagramas de secuencia de Mover_Process

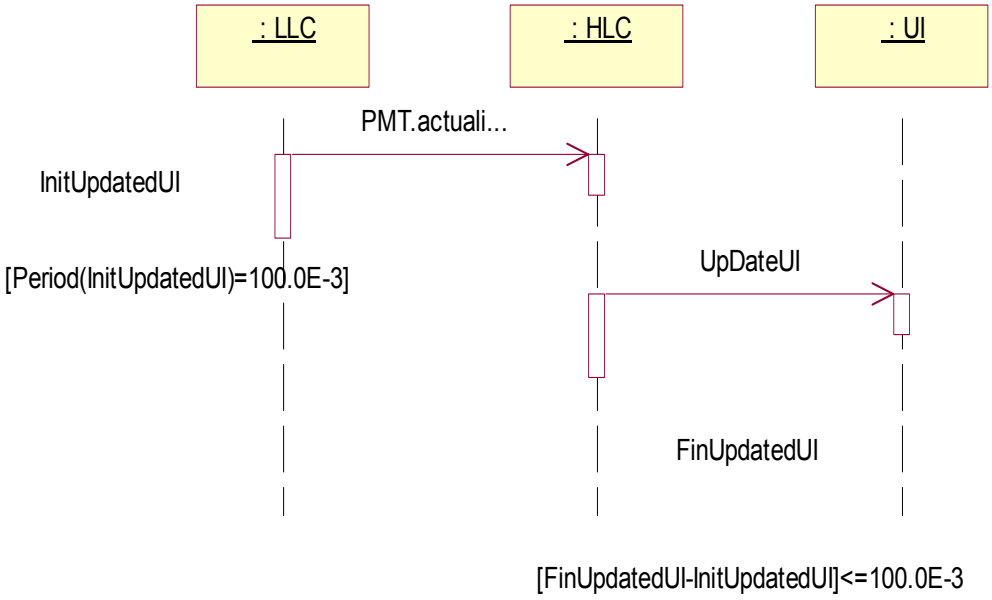


Figura 17 Diagramas de secuencia del UpDateUI_Process

4.3.1.2 Modelo de tiempo real de la plataforma.

Esta parte del modelo es idéntica para ambas pruebas, ya que ambas se ejecutan sobre los mismos recursos hardware (procesador), el mismo sistema operativo y consideran las mismas tareas sobre las que se ejecutarán las operaciones del sistema.

4.3.1.3 Modelo de los componentes lógicos.

En esta vista de MAST vamos a poder apreciar cambios en dos de los componentes lógicos del modelo, los demás serán idénticos a los del Modelo1

CinServer_Model

En este modelo lógico podemos apreciar la aparición de una nueva operación ***UpDatePosition*** del tipo *Simple_Operation,*

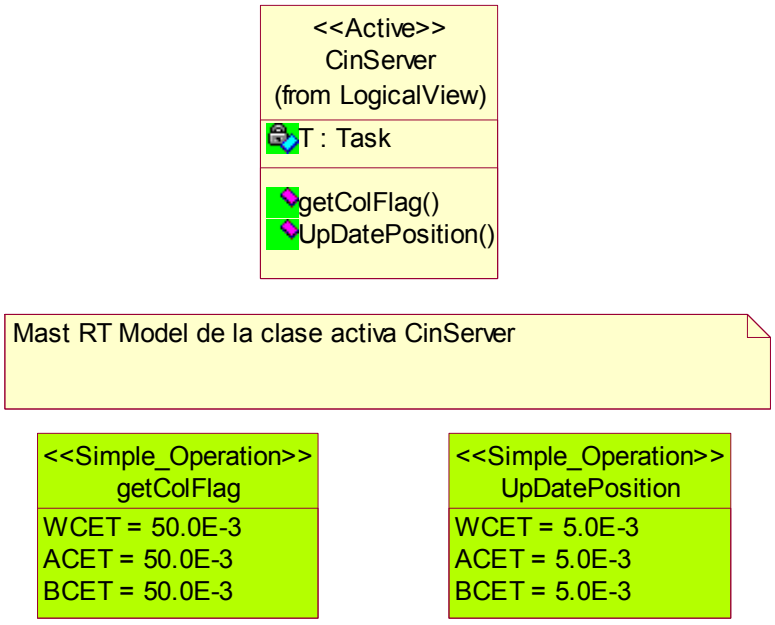


Figura 18

getColFlag es un modelo de operación simple con un tiempo de ejecución de 50 ms. Es un tiempo de ejecución alto debido a que se trata de una operación de acceso a un flag del servidor de cinemática, esta operación permitirá conocer el estado del flag de colisión.

UpDatePosition es un modelo de operación simple con un tiempo de ejecución de 5 ms. Este procedimiento permitirá que el controlador de bajo nivel actualice los datos de posición en el servidor de cinemática.

LLC_Model.

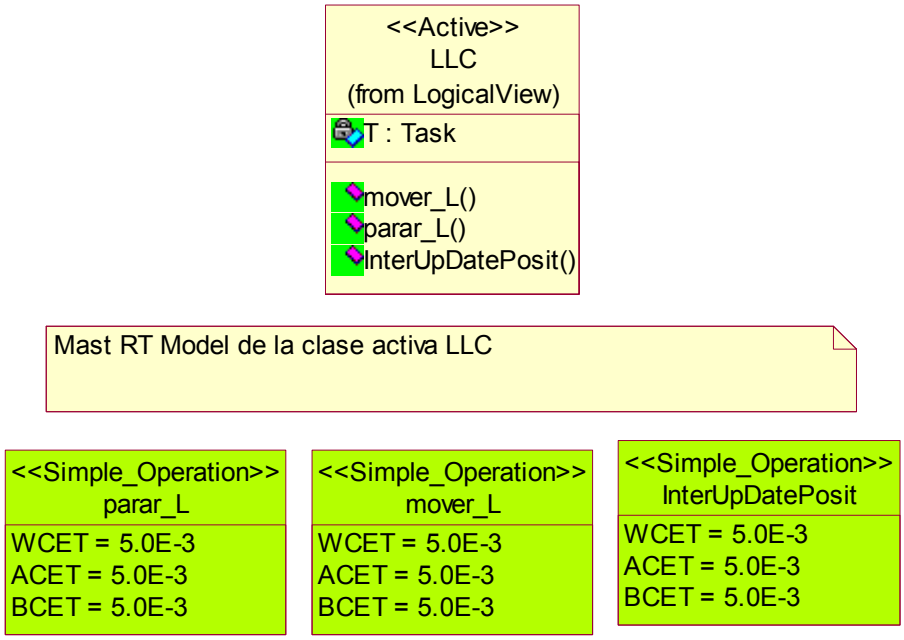


Figura 19

parar_L es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de detenerlo.

mover_L es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de comenzar a moverse.

InterUpDatePosit es un modelo de operación simple con el cual se le indica al controlador de bajo nivel que debe actualizar la posición del robot en el servidor de cinemática, se utiliza como una manera de poder sincronizar las acciones durante la transacción de mover. De esta manera, el controlador de bajo nivel sabe cual es el momento de actualizar la posición.

4.3.1.4 Modelo de los escenarios de tiempo real.

En la figura 20, se ve el escenario del Modelo2. Como en el escenario del Modelo1, se pueden apreciar dos transacciones diferentes. Estas son: *UpDateUI_Transaction* y *Mover_Transaction*.

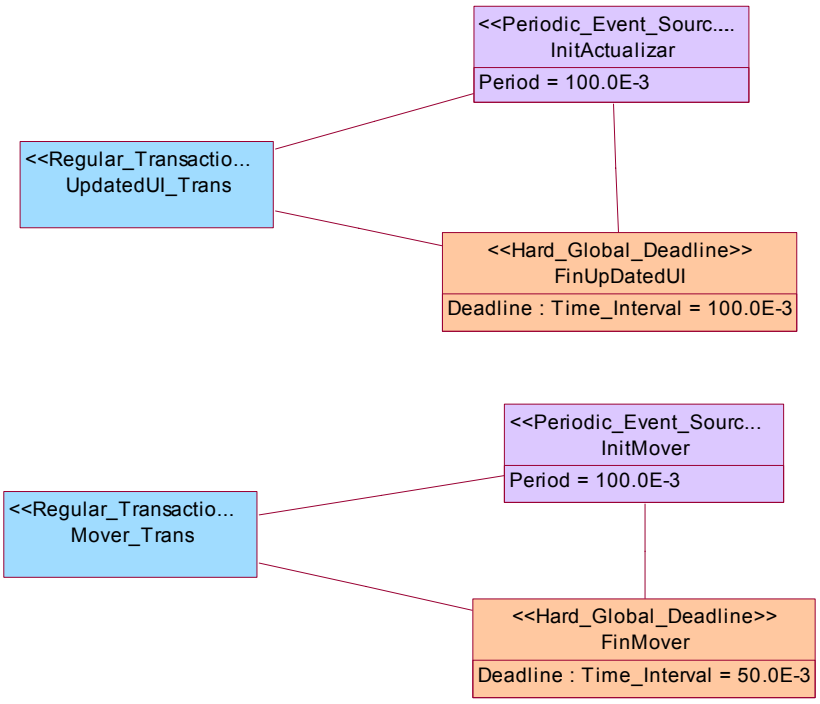
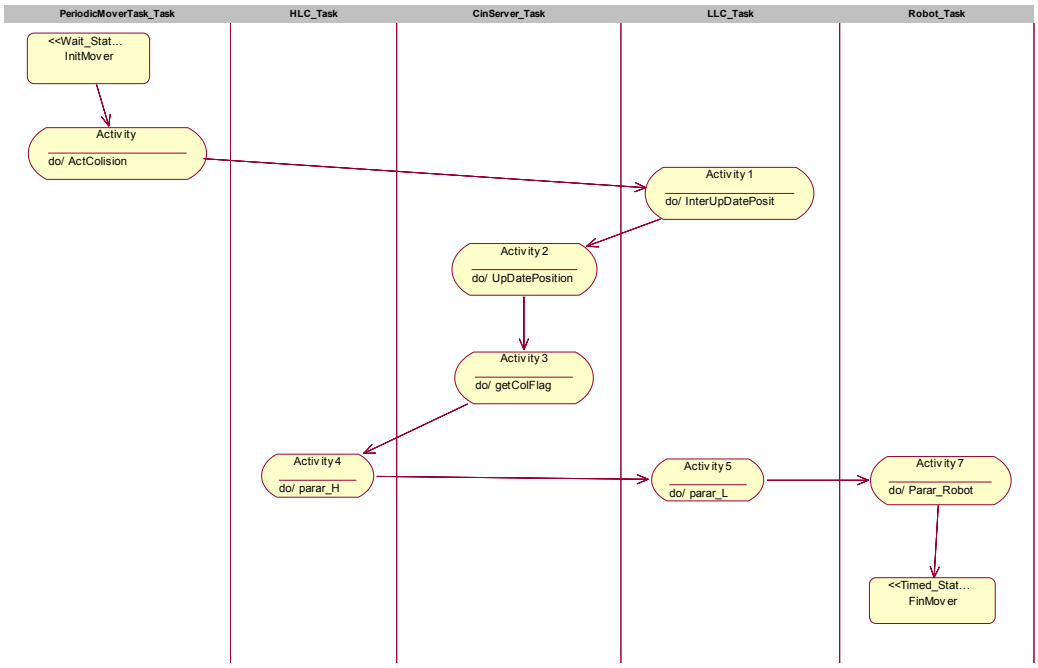


Figura 20

La transacción *UpDateUI_Transaction* va a ser idéntica a la explicada en el modelo1, por lo cual no se volverá a explicar. Pasamos directamente a explicar la transacción *Mover_Transaction*.

Mover_Transaction.

Esta transacción se va a encargar de controlar la posibilidad de colisión durante el movimiento del robot, es decir, controlará la posibilidad de un obstáculo que imposibilite la continuidad del movimiento del robot, por lo cual deberá ordenar la detención del mismo.



En esta transacción se va a controlar periódicamente el estado del *flag* de colisión. En caso de estar activado indicará la posibilidad que se produzca una colisión por lo cual el sistema debe de actuar deteniéndose. Será iniciada desde la tarea *PeriodicMoverTask* desde un *Wait_State*, esto indica que la tarea estará a la espera de una señal de activación. En cuanto se produzca esta señal, el controlador de alto nivel (*HLC*) informará al controlador de bajo nivel (*LLC*) que debe de actualizar la posición del Robot (*InterUpDatePosit*). Posteriormente el controlador de bajo nivel actualizará dicha posición en el servidor de cinemática (UpDatePosit). Finalmente el controlador de alto nivel comprobará el estado del *flag* de colisión del servidor de cinemática. Si está activado ordenará al sistema que debe detenerse. Por último, se llega a un *Timed_State*, que espera una señal de tipo temporal. Ésta será creada por el *Hard_Global_Deadline*, en este caso cada 50ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

4.4 Prueba3.

En esta prueba se parte de la **Prueba1**, aumentando la dificultad del escenario, así como los componentes que forman el sistema.

4.4.1 Modelo3.

En este nuevo modelo, se hace la inclusión de dos nuevos módulos en el sistema, esta variación se va a llevar a cabo respecto al **Modelo1**. Estos nuevos módulos van a ser un recurso compartido y una nueva tarea incluida en el controlador de alto nivel.

El añadir nuevos módulos al sistema va a variar la funcionalidad de los componentes existentes en el Modelo1. Por eso nos concentraremos en este nuevo modulo, dando por explicados todo lo que se conserve igual que en el Modelo1 original.

4.4.1.1 Vista Lógica.

En la figura 21, se puede observar el diagrama de clases que da la representación estática del Modelo3.

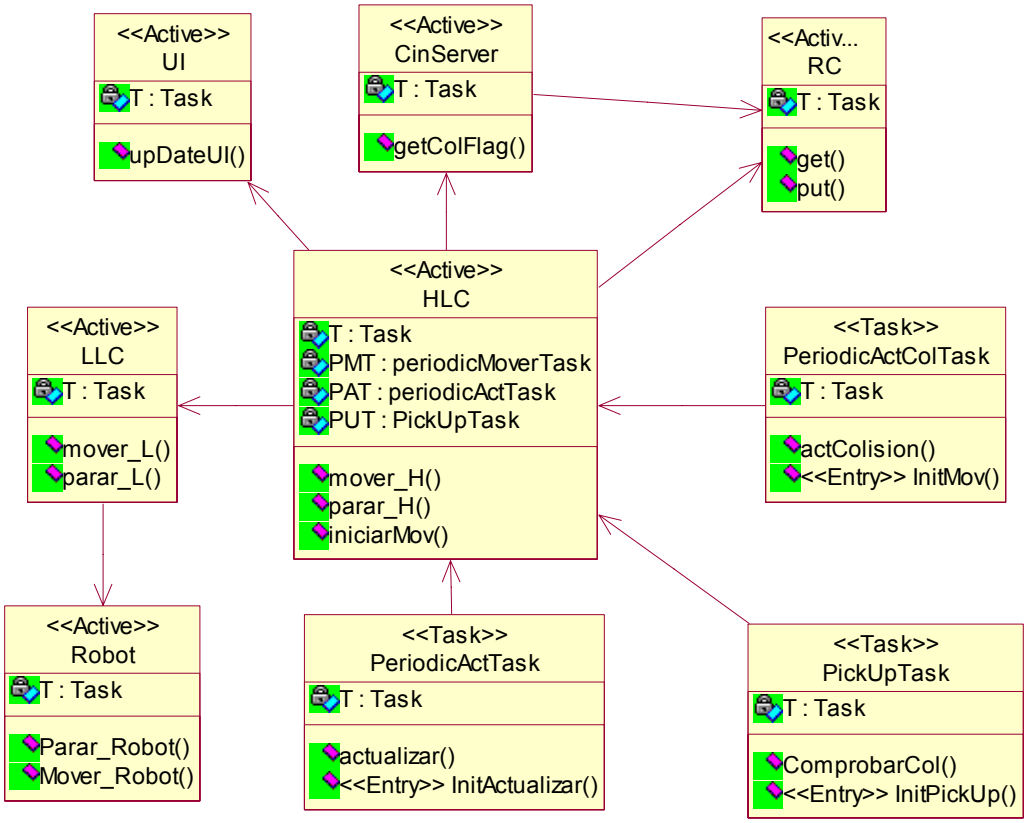


Figura 21

En este nuevo diagrama de clases, se observa la inclusión de los nuevos componentes: Un recurso compartido al cual se denominará *RC* y dos nuevas clases tipo Tarea (*Task*) nombradas como *PeriodicActColTask* y *PickUpTask*. Cada uno de ellas tiene la siguiente funcionalidad:

- **RC**: va a ser un recurso compartido, que se va a utilizar para optimizar las comunicaciones entre tareas. En este modelo, guardará el estado del *flag* de colisión, y controlará el acceso de escritura y lectura de la variable. Este recurso compartido ha sido definido en el modelo como una tarea que se encuentra en estado de espera hasta que alguien desea acceder a la variable. De esta manera, intentaremos evitar los problemas que se pueden crear cuando se tiene que acceder a variables compartidas.
- ***PeriodicActColTask*** y ***PickUpTask*** son tareas que van a tener una funcionalidad del mismo tipo que *PeriodicMoverTask* y *PeriodicActualizarTask*, es decir, serán tareas que se ejecutarán en el controlador de alto nivel y se encargarán de controlar una de las transacciones periódicas que se ejecutan en este sistema. Estas tareas se crean debido a la incorporación del recurso compartido. La tarea *PeriodicMoverTask* definida en el Modelo1, que se encargaba de controlar el estado del *flag* de colisión de manera directa preguntando el controlador de alto nivel al servidor de cinemática, en este Modelo3 se divide en dos tareas que van a ser *PeriodicActColTask* y *PeriodicPickUpTask*. *PeriodicActColTask*

va a ser una tarea que controlará la actualización de la variable de colisión por el servidor de cinemática. *PeriodicPickUpTask* se encargará de controlar el estado de la variable *colision* de manera periódica, y en caso de que esté activado, detener el Robot.

Los diagramas de secuencia del Modelo3 también van a sufrir modificaciones respecto a los expuestos en el Modelo1, excepto el que representa la secuencia que se realiza para el proceso de *UpDateUI*. La secuencia de acciones se explicará en el apartado correspondiente a la definición de los escenarios Además en este modelo habrá un nuevo diagrama de secuencia.

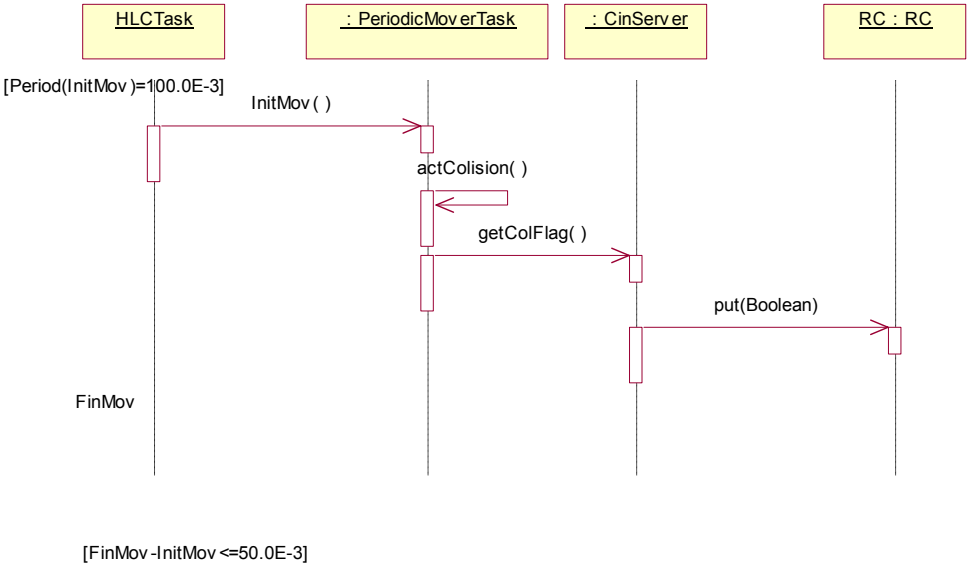


Figura 22 Diagrama de secuencia de ActCol_Process.

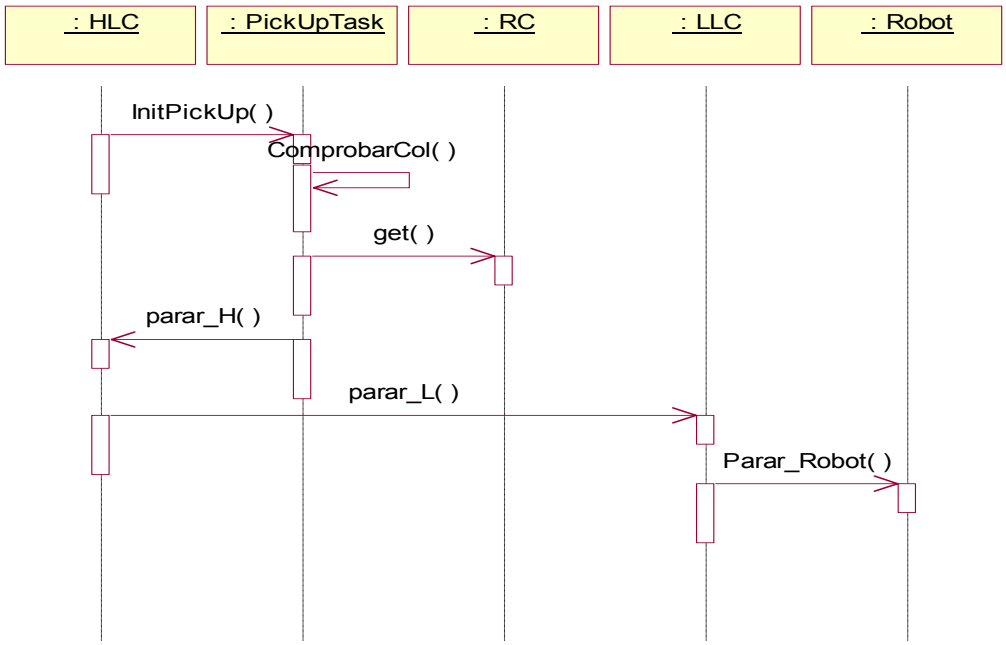


Figura 23 Diagrama de secuencia de PickUp_Process

4.4.1.2 Modelo de tiempo real de la plataforma.

En esta vista hay muy pocas variaciones respecto al Modelo1. Se mantiene el mismo procesador definido en el primer modelo. Sólo se le van a añadir tres nuevos objetos que indican los hilos de ejecución (FP_Sched_Server) asociados a los componentes añadidos, así como los objetos que le asignan sus políticas de planificación. En la figura 24 podemos ver los objetos añadidos a la vista de tiempo real de la plataforma en el Modelo3.

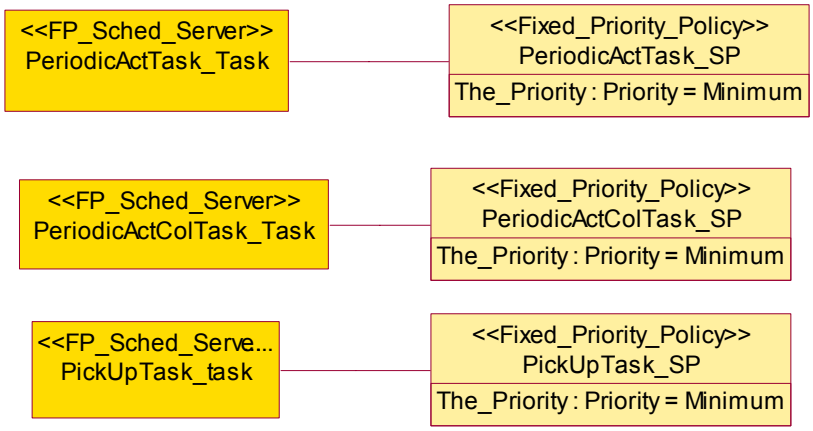


Figura 24

4.4.1.3 Modelo de tiempo real de los componentes lógicos

A esta vista se le han añadido tres nuevos modelos de componentes lógicos correspondientes a los tres nuevos componentes añadidos. Los cuales se explican a continuación.

RC_Model.

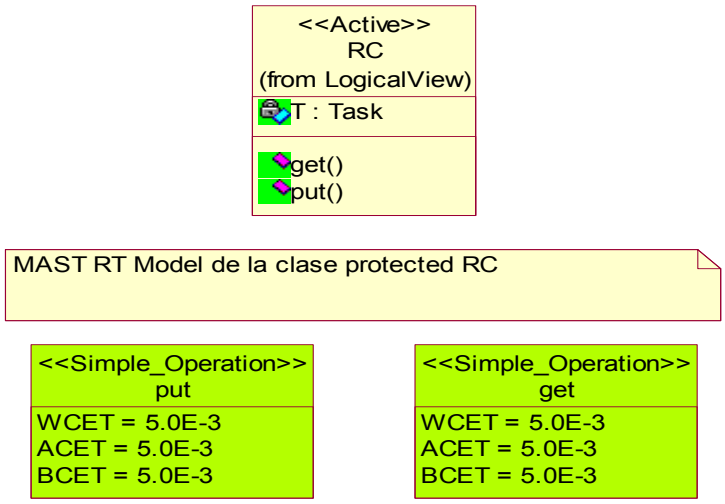


Figura 25

put es un modelo de operación simple con una duración de 5ms. Esta operación será utilizada para poder acceder en modo escritura a las variables del recurso compartido.

get es un modelo de operación simple con una duración de 5ms. Esta operación será utilizada para poder acceder en modo lectura a las variables del recurso compartido.

PeriodicActColTask

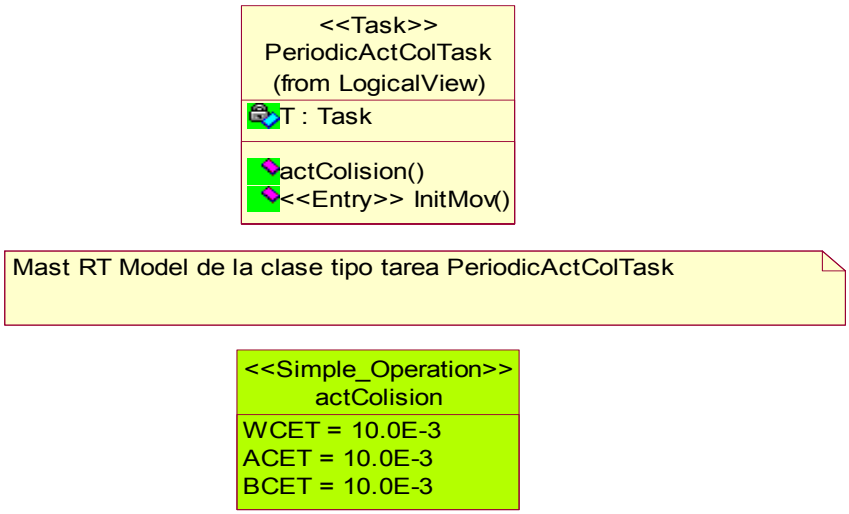


Figura 26

actColision es un modelo de operación simple con una duración de 10ms. Esta operación está definida dentro de esta tarea, la cual la llama de manera periódica. Esta operación se encargará de iniciar el proceso por el cual el servidor de cinemática actualizará en el recurso compartido el valor de la variable *colision.*

PickUpTask.

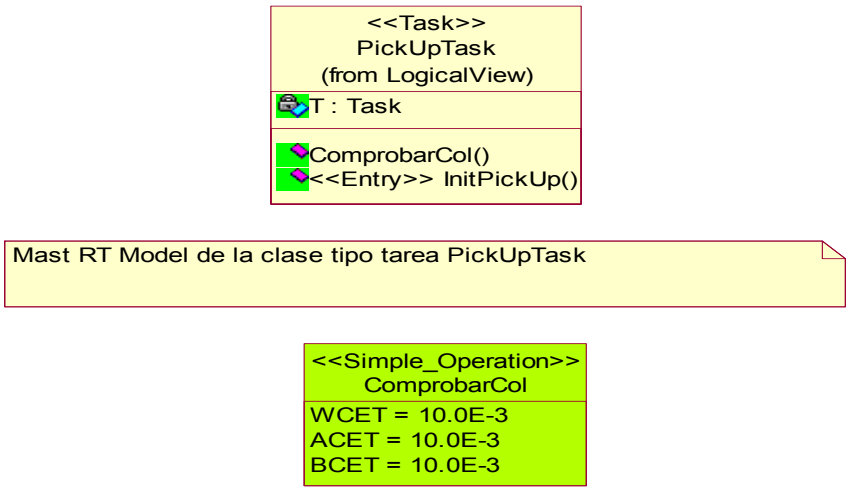


Figura 27

comprobarCol es un modelo de operación simple con una duración de 10ms. Esta operación está definida dentro de esta tarea, la cual la llama de manera periódica. Esta operación se encargará de iniciar el proceso por el cual el controlador de alto nivel comprobará en el recurso compartido el valor de la variable *colision.*

4.4.1.4 Escenarios de tiempo real.

En la figura 28, se puede ver el escenario planteado en este modelo, donde se pueden ver las tres transacciones definidas .La Transición *UpdateUI_Transaction* vuelve a ser similar a las definida en modelos anteriores y no sufre ninguna variación. Además de ésta aparecen dos nuevas transacciones que son: *PeriodicActCol_Tansaction* y *PickUp_Transaction*. Estas dos tareas van a ser el resultado de separar el proceso de acceso a variables comunes. En vez de comprobar en una sola transacción, se ha realizado una para actualizar el valor del *flag* de colisión (*PeriodicActCol_Tansaction*) y otra para ver el estado de este *flag* y hacer reaccionar al sistema según el valor de la variable consultada.

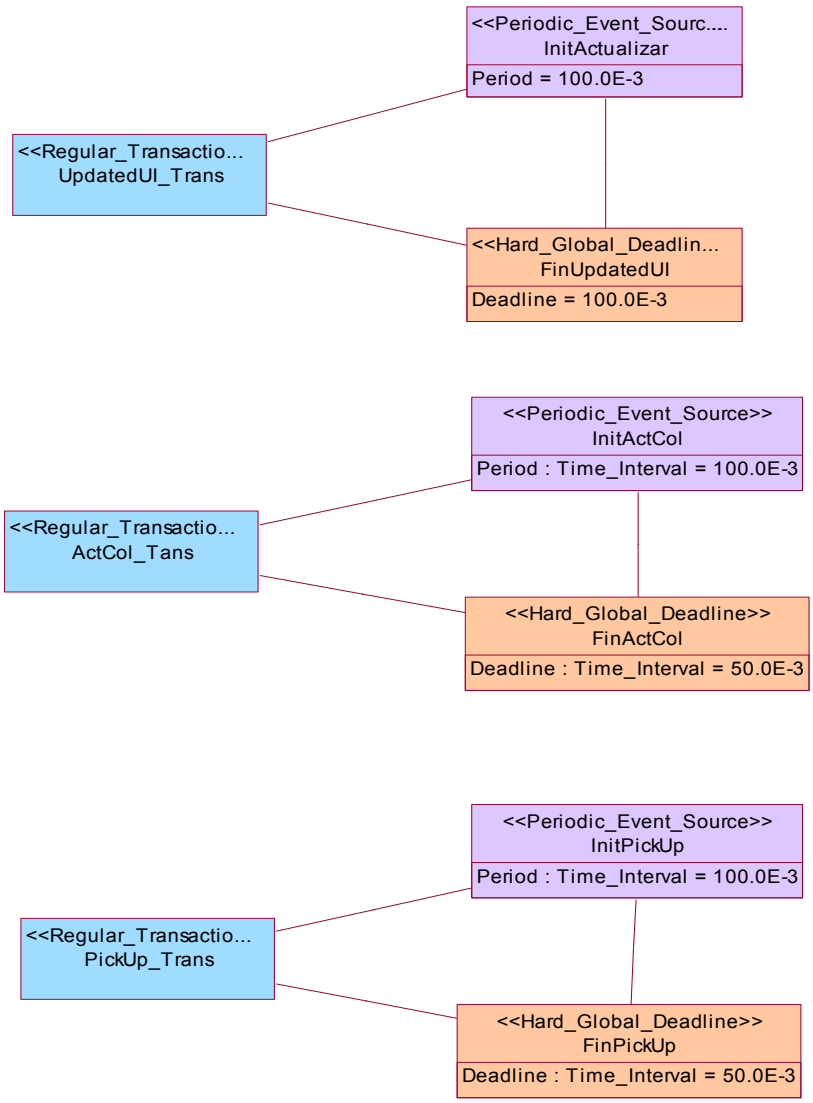


Figura 28

ActCol_Transaction.

Esta transacción se va a encargar de actualizar el valor del *flag* de colisión en la variable *colision*. Para ello deberá acceder al recurso compartido en modo de escritura.

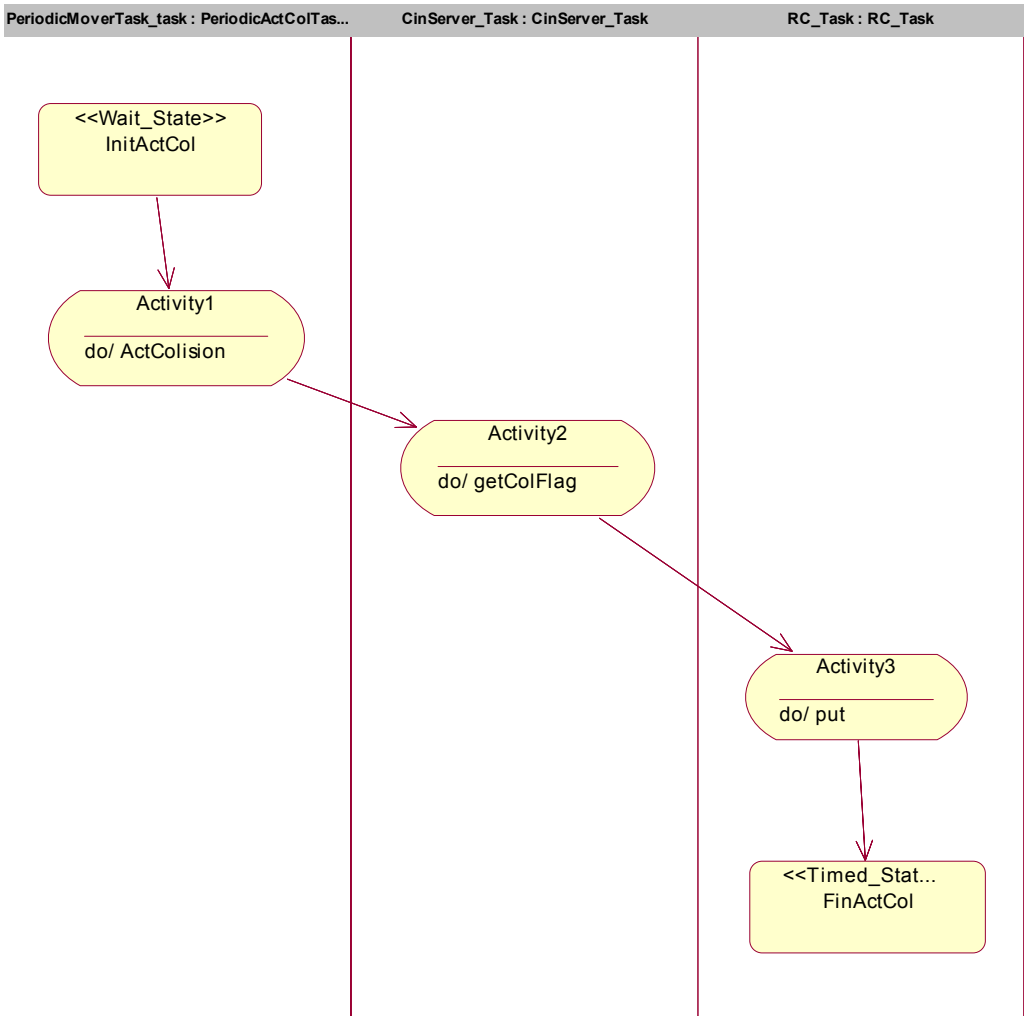


Figura 29

En la figura 29, se puede ver el diagrama de actividad de la transacción. Ésta va a actualizar periódicamente el estado del *flag* de colisión y posteriormente accederá al recurso compartido, y guardará en la variable *colision* el valor de dicho *flag* será iniciada desde la tarea PariodicActColTask desde un Wait_State. Esto indica que la tarea estará a la espera de una señal de activación (*InitActCol*). En cuanto se produzca esta señal, el sistema comprobará el estado del *flag* de colisión desde esta tarea. En primer lugar conseguirá el estado del *flan*. Una vez obtenida (getColFlag) escribirá su valor en la variable *colision* del recurso compartido (put). Por último, se llega a un Timed_State (FinActCol). Este estado espera una señal de tipo temporal creada por el Hard_Global_Deadline, en este caso cada 50ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo, este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

PickUp_Transaction.

Esta transacción se va a encargar de leer el valor de la variable *colision* situada en el recurso compartido. Para ello deberá acceder al recurso compartido en modo de lectura, ya que allí es donde se guarda la variable nombrada.

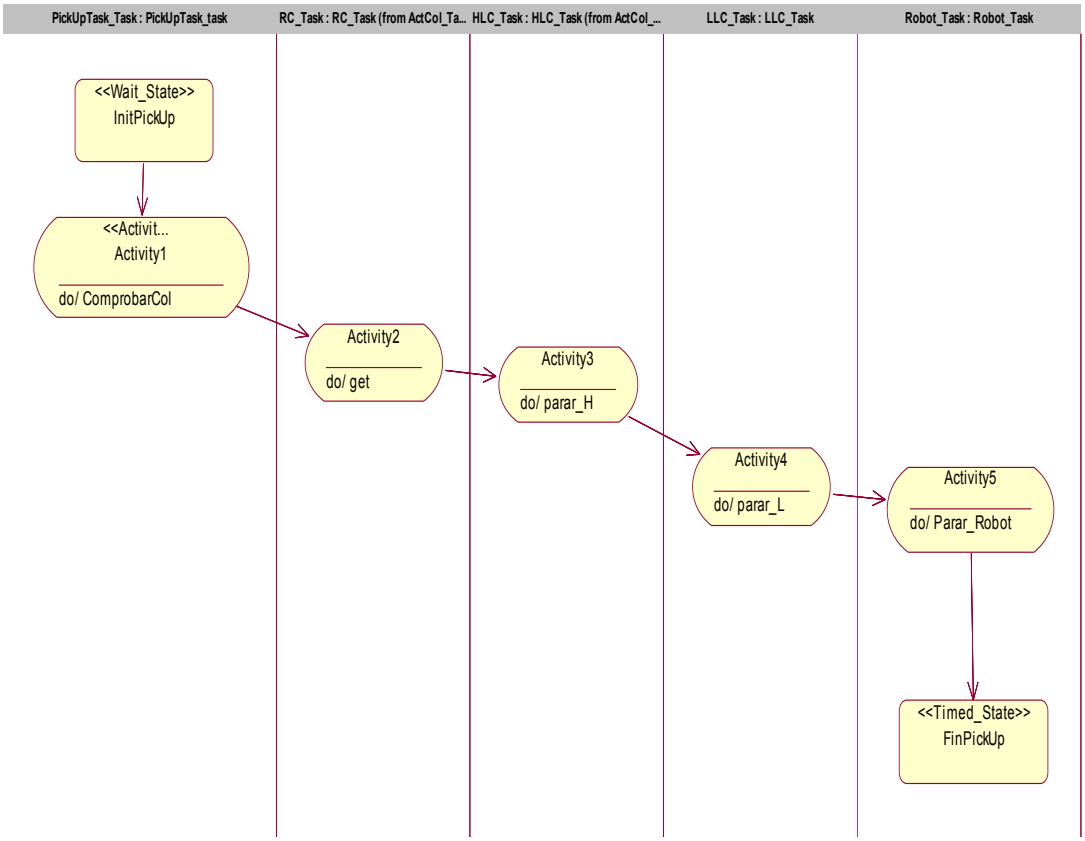


Figura 30

En la figura 30, se puede ver el diagrama de actividad de la transacción. En este diagrama se muestra la secuencia de operaciones que realiza la transacción. Ésta va a acceder periódicamente a la variable *colision,* procesará su valor y hará detenerse al sistema en caso de que esté activada. Será iniciada desde la tarea *PickUpITask,* desde un *Wait_State.* Esto indica que la tarea estará a la espera de una señal de activación (*InitPickUp*). En cuanto se produzca esta señal, el sistema empezará el proceso de lectura, obtendrá el valor de la variable *colision* del recurso compartido (*get*), y una vez lo obtenga lo procesara y en caso de estar activado empezará el proceso de detener el Robot. Por último, se llega a un *Timed_State* (*FinPickUp*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*, (en este caso cada 50ms) dándole un tiempo límite para se produzca la transacción. En caso de que se sobrepase este tiempo, este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

4.5 Prueba4

En esta prueba se va a partir de la **Prueba3**, añadiéndole un nuevo módulo y aumentando un poco la complejidad de los escenarios.

4.5.1 Modelo4.

4.5.1.1 Vista lógica.

En la figura 31, vemos el diagrama de clases definido en la vista lógica del Modelo4. En ésta se ve la aparición de un nuevo modulo: *Cindecoupler.* Este nuevo módulo va a introducir un concepto nuevo para la estructura del sistema: un desacoplador.

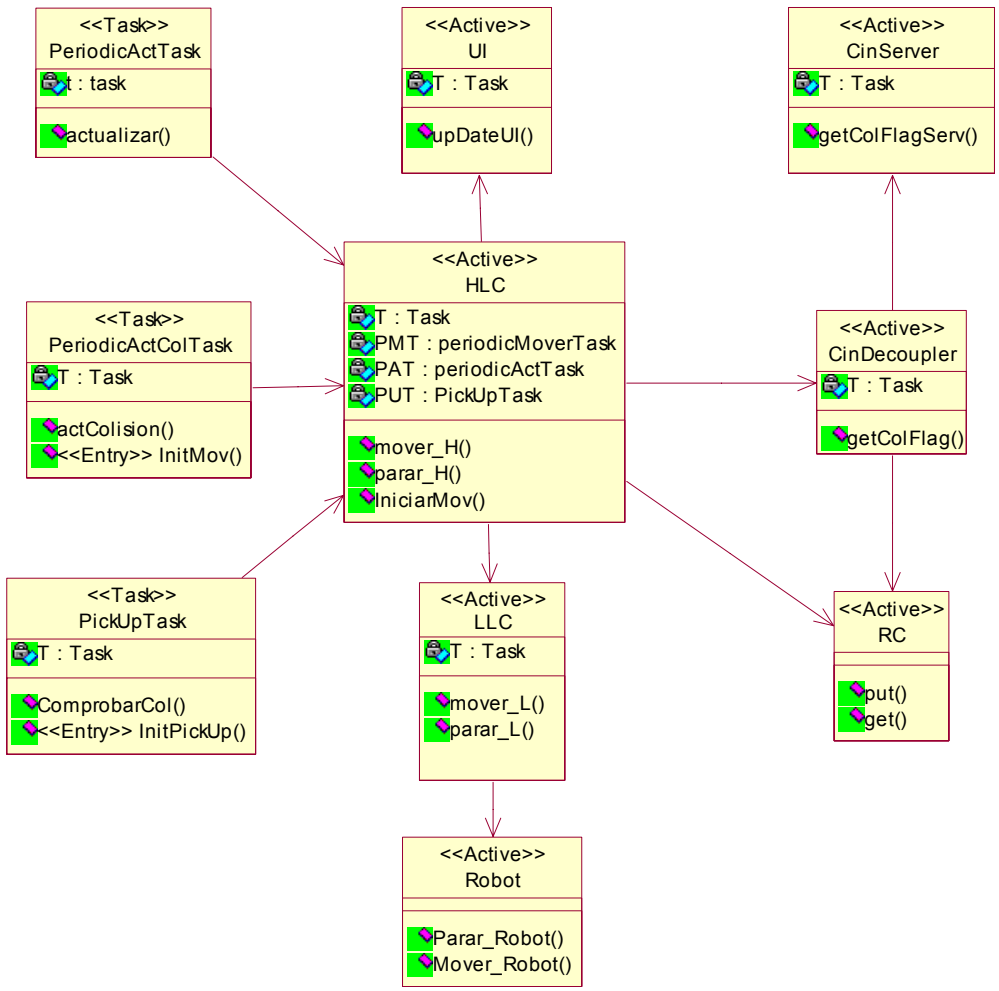


Figura 31

Cindecoupler es el nuevo módulo añadido al sistema en este modelo. Representa un módulo de desacoplo (*decoupler*). Este módulo de desacoplo se va a encargar de realizar una comunicación asíncrona, entre el controlador de alto nivel del sistema y otro subsistema, que se encuentra fuera de él, el servidor de cinemática (CinServer). Para esto, el modulo de desacoplo se va a encargar de crear una

interfaz inmutable para el acceso al resto de los subsistemas. De esta manera aíslan al controlador de cambios en sus interfaces. En este sentido, las clases de desacoplo realizan la función de adaptadores.

Debido a la inclusión del nuevo módulo explicado anteriormente, los diagramas de secuencia de *ActCol_Process* y *PickUp_Process* también van a sufrir modificación mientras que el de *UpDateUI_Process* se mantendrá igual que el del Modelo1.

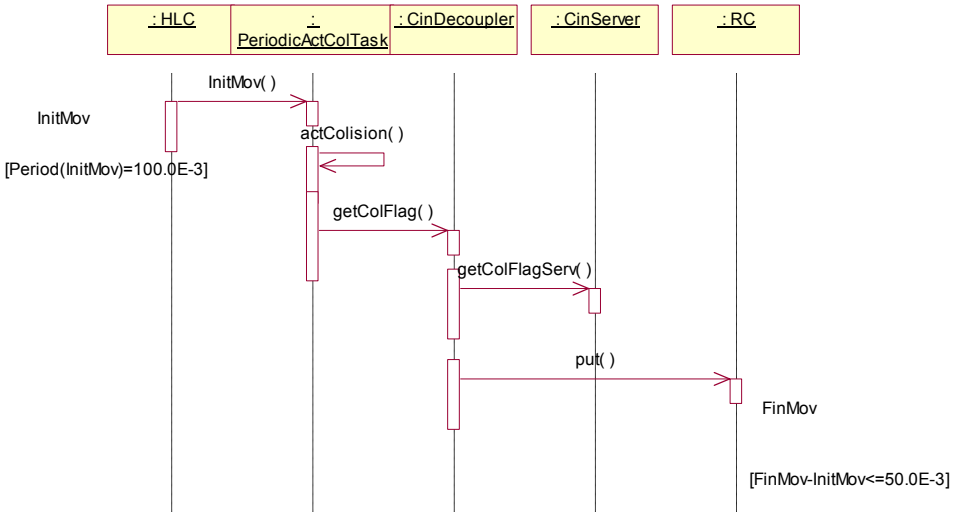


Figura 32 Diagrama de secuencia de *ActCol_Process*

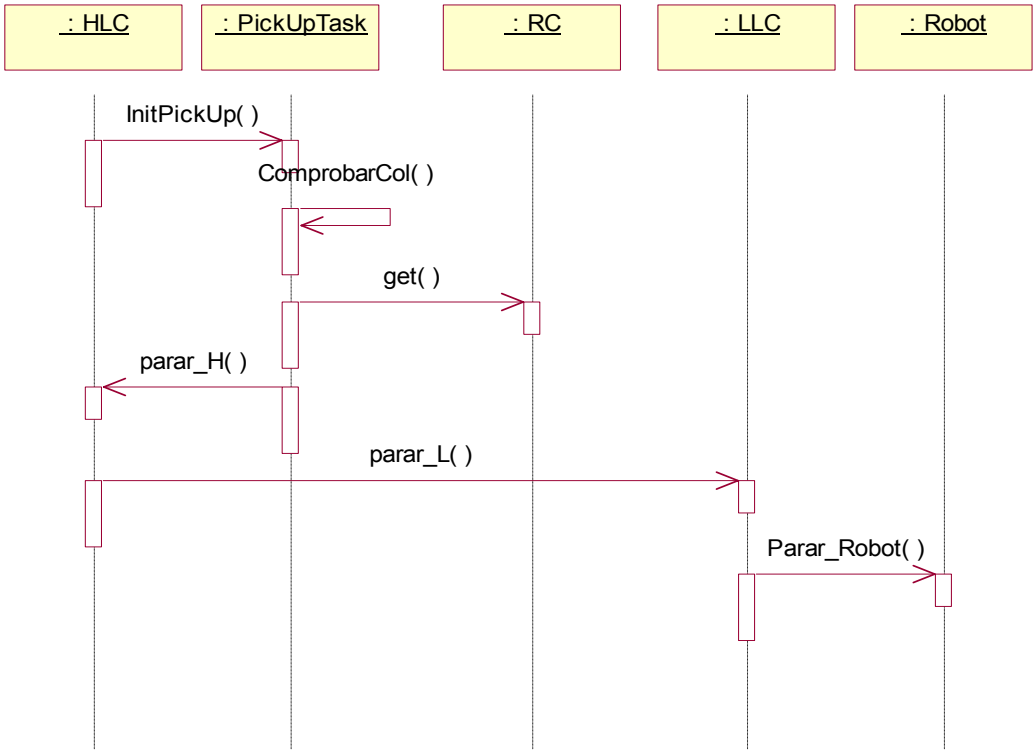


Figura 33 Diagrama de secuencia de *PickUp_Process*

4.5.1.2 Vista de tiempo real de la plataforma

Esta vista va a sufrir solamente una leve modificación, se le van a añadir los componentes *Cindecoupler_Task* y *CinDecoupler_SP,* que se observan en la figura 34.

Figura 34

- **CinDecoupler_Task**: define un *thread* que introduce la instancia a la clase *Cindecoupler*. Este *thread* se encarga de controlar los eventos que ocurran sobre el *Cindecoupler*.
- **Cindecoupler_SP**: define al *server* que planifica las actividades de *Cindecoupler_Task* con una prioridad fija que dejaremos que MAST se la asigne en el análisis según RMA.

4.5.1.3 Modelo de tiempo real de los componentes lógicos.

En esta vista va a ver una significativa variación respecto al Modelo3, debida a la función desarrollada en el sistema por el *CinDecoupler*, y para tener que realizar el menor número de cambios posibles en el sistema se ha cambiado el nombre de la operación definida en la clase *CinServer*. En el Modelo3, la operación definida en *CinServer* se le denominaba *getColFlag*. En este modelo la vamos a denominar *getColFlagServ*, aunque mantendrá la misma funcionalidad. En este caso, en la clase *CinDecoupler* definiremos una operación con el nombre *getColFlag* que va a tener la funcionalidad definida en este apartado.

A continuación se explican los dos modelos lógicos mencionados, tal como están definidos en este modelo.

Cindecoupler_Model.

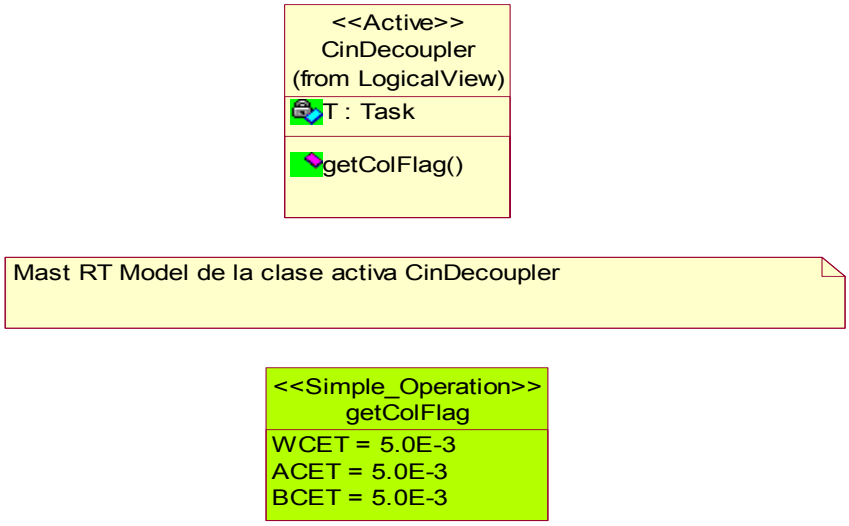


Figura 35

getColFlag es un modelo de operación simple con una duración de 10ms. Esta operación será utilizada como intermediaría para comunicar el sistema con el servidor de cinemática, para poder acceder al valor del *flag* de colisión.

CinServer_Model.

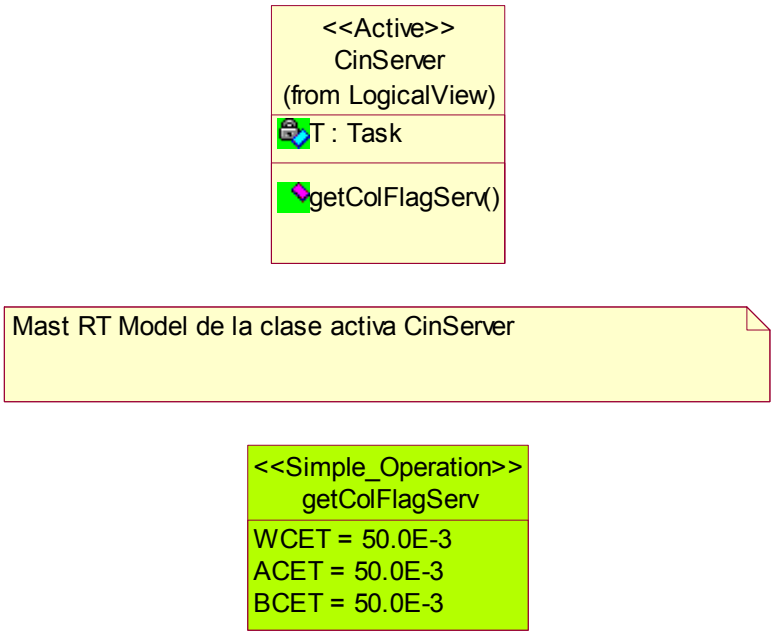


Figura 36

getColFlagServ es un modelo de operación simple con un tiempo de ejecución de 50ms. Es un tiempo de ejecución alto debido a que se trata de una operación de acceso a un *flag* del servidor de cinemática, esta operación permitirá conocer el estado del *flag* de colisión.

4.5.1.4 Escenarios de tiempo real.

En la figura 37, observamos el escenario perteneciente al Modelo4. Se puede observar que se trata del mismo escenario que en el Modelo3, pero se podrán apreciar ligeros cambios dentro del diagrama de actividad de la transaccion *ActCol_Transaction*, los cuales explicaremos a continuación, los transacciones *UpDateui_Transaction* seguirá siendo igual que en el primer modelo y *PickUp_Transaction* que será idéntica a la explicada en el Modelo3.

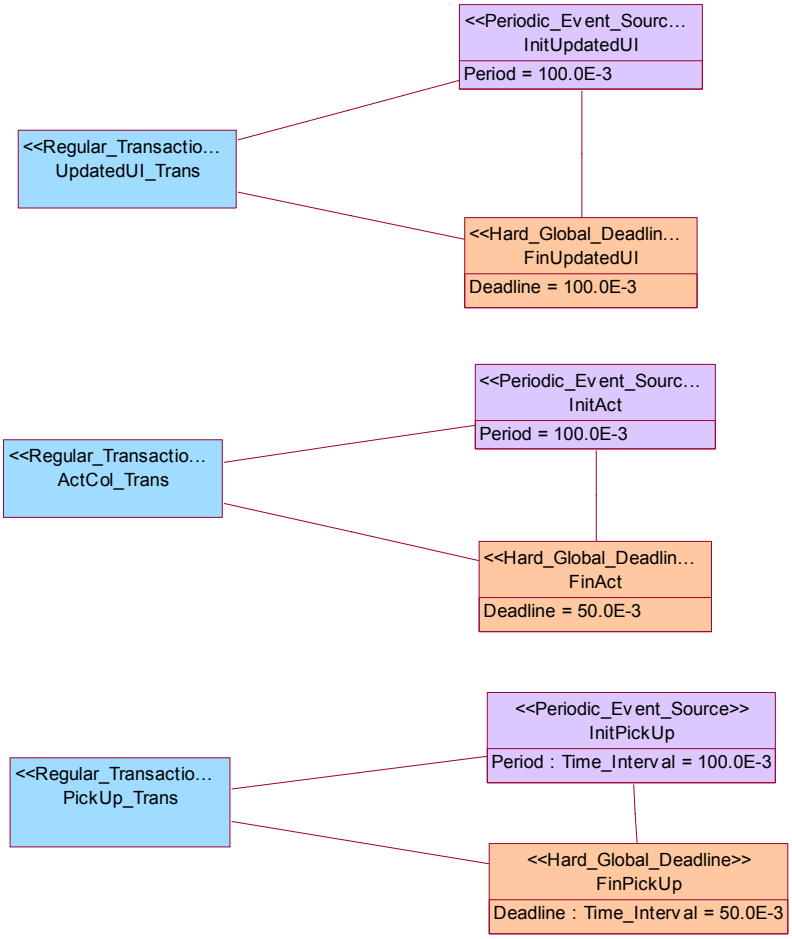


Figura 37

ActCol_Transaction.

Esta transacción se va a encargar de actualizar el valor del *flag* de colisión en la variable *colision*. En este modelo accederá al *flag* de colisión a través del desacoplador de cinemática. Después deberá acceder al recurso compartido en modo de escritura.

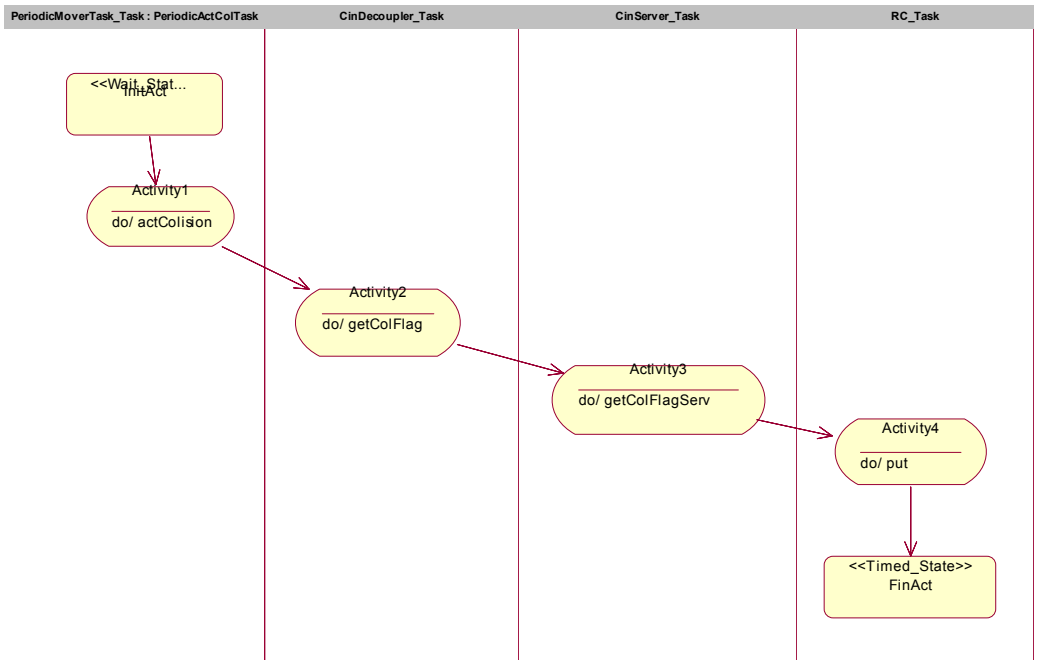


Figura 38

En la figura 38, se puede ver el diagrama de actividad de la transacción. En este diagrama se muestra la secuencia de operaciones que realiza la transacción. Ésta va a actualizar periódicamente el estado del *flag* de colisión. La transacción será iniciada desde la tarea PeriodicActColTask desde un *Wait_State*. Esto indica que la tarea estará a la espera de una señal de activación (*InitActCol*). En cuanto se produzca esta señal, el sistema empezará el proceso de comprobar el estado del *flag* de colisión. En primer lugar conseguirá el estado del *flag* a través del *Cindecoupler* (*getColFlag*) que se comunicará con el servidor de cinemática, actuando como intérprete. Una vez obtenga el valor (*getColFlagServ*), el *Cindecoupler* se lo devolverá al controlador de alto nivel, el cual escribirá este valor en la variable *colision* del recurso compartido (put). Por último, se llega a un *Timed_State* (*FinActCol*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*, en este caso cada 50ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

4.6 Prueba5.

4.6.1 Modelo5.

En este modelo se va a realizar una ampliación respecto al Modelo4, se le van a añadir nuevos componentes y se aumentará la complejidad del escenario.

4.6.1.1 Vista Lógica.

En la figura 32, se observa el diagrama de clases correspondiente al Modelo5, en cual se observa, en comparación al diagrama del modelo anterior, la inclusión de dos nuevos componentes que formarán un nuevo bloque en el sistema.

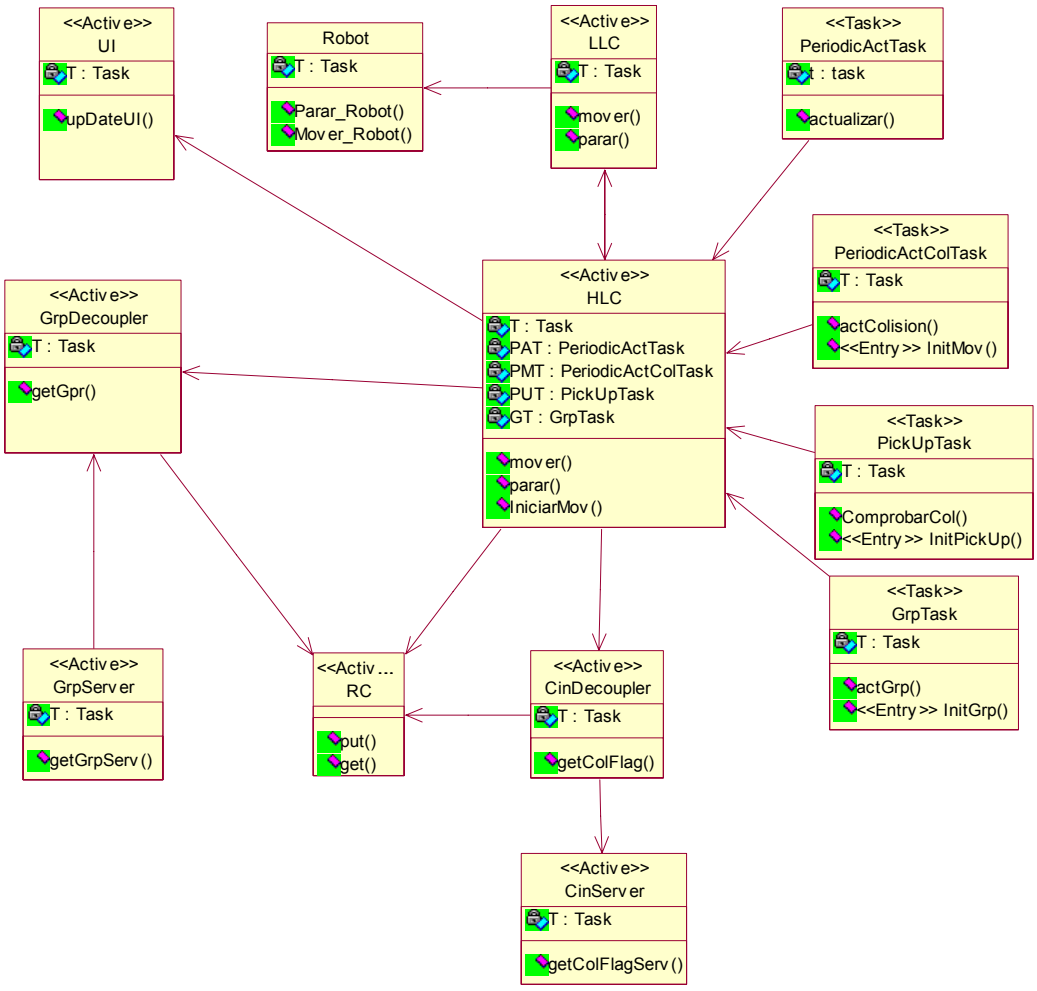


Figura 39

Al observar la figura se destaca la inclusión de un nuevo bloque compuesto por un servidor gráfico (*GrpServer*) y un módulo desacoplador, el cual realizará una función similar a la llevada a cabo por el módulo desacoplador explicado en el modelo anterior, pero respecto al servidor grafico. También se puede observar la aparición de una nueva clase tipo *Task* denominada *GrpTask*. La inclusión de este nuevo bloque

afectará al análisis temporal del sistema ya que implicará una nueva transacción, aumentando así la dificultad del escenario planteado para el mismo.

La inclusión de estos nuevos componentes, va a ir acompañada del diseño de una nueva transacción. El diagrama de secuencia que representar el nuevo proceso, denominado Grp_Process, se ilustra en la figura 40.

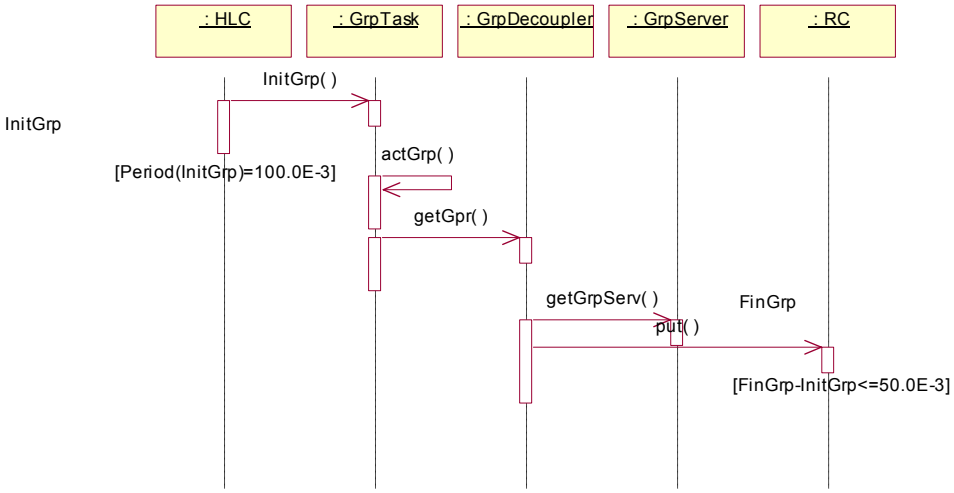


Figura 40

4.6.1.2 Vista lógica de la plataforma.

Esta vista va a sufrir al modificación, consiste en añadir los componentes *Grpdecoupler_Task*, *GrpDecoupler_SP, GrpServer_Task, GrpServer_SP, GrpTask_Task Grp* y *Task_SP* que se observan en la figura 41.

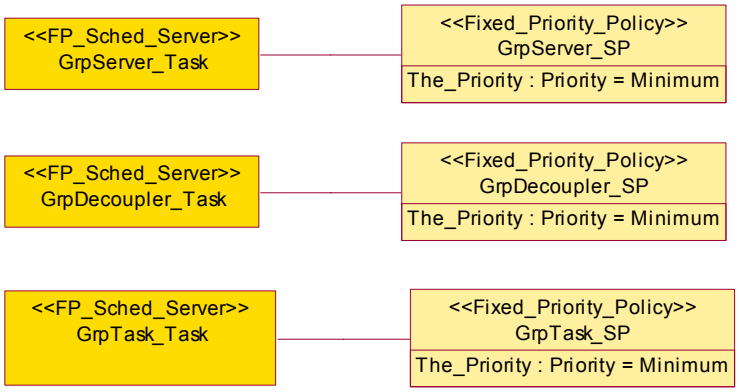


Figura 41

- **GrpDecoupler_Task**: define un *thread* que introduce la instancia a la clase *Grpdecoupler*. Este *thread* se encarga de controlar los eventos que ocurran sobre el Grp*decoupler*.

- **Grpdecoupler_SP**: define al *server* que planifica las actividades de *Grpdecoupler_Task* con una prioridad fija que dejaremos que MAST se la asigne en el análisis según RMA.
- **GrpServer_Task**: *Thread* que introduce la instancia de la clase Grp*Server*. Este *thread* se encarga de controlar los eventos que ocurran sobre el servidor de cinemática.
- **GrpServer_SP**: Es el *server* que planifica las actividades del *Grp_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **GrpTask_Task**: *Thread* que introduce la instancia de la clase Grp*Task*. Este *thread* se encarga de controlar los eventos que ocurran sobre el *GrpTask*.
- **GrpTask_SP**: Es el *server* que planifica las actividades del *GrpTask_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.

4.6.1.3 Modelo de tiempo real de los componentes lógicos.

En esta vista vamos a explicar los diferentes modelos lógicos que hemos añadido a causa de los nuevos componentes que componen este modelo del sistema.

GrpTask_Model.

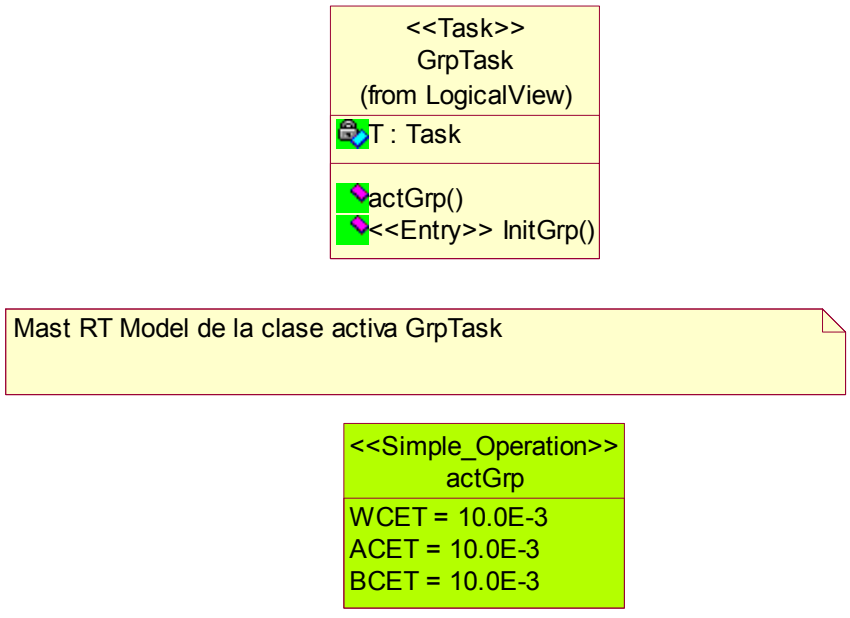


Figura 42

ActGrp es un modelo de operación simple con una duración de 10ms. Esta operación está definida dentro de esta tarea, la cual la llama de manera periódica. Esta operación se encargará de iniciar el proceso por el cual el controlador de alto nivel comprobará en el recurso compartido el valor de la variable *GrpVar*, que será con la variable que va a trabajar el servidor grafico.

GrpServer_Model.

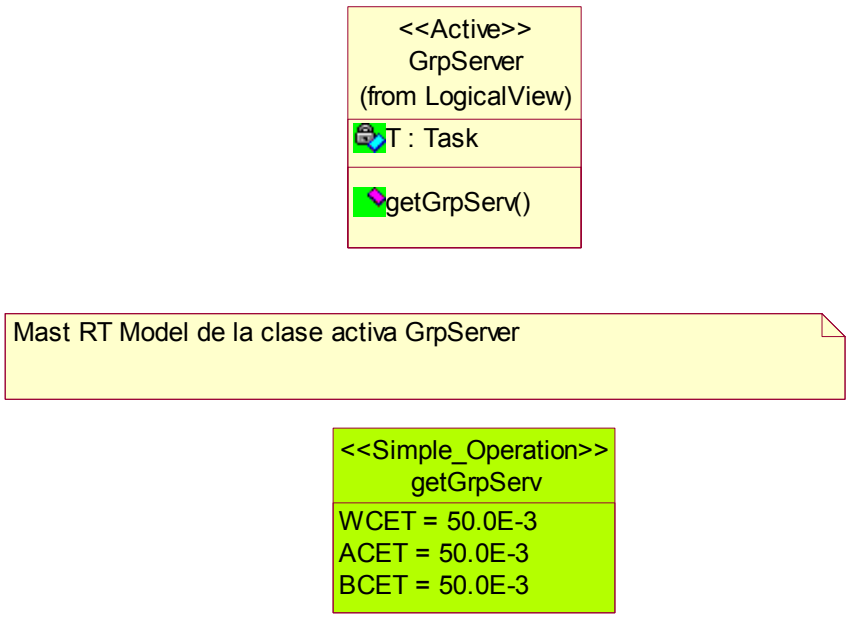


Figura 43

ActGrpServ es un modelo de operación simple con un tiempo de ejecución de 50ms. Es un tiempo de ejecución alto debido a que se trata de una operación de acceso a una variable del servidor grafico. Esta operación permitirá conocer el estado de dicha variable, *GrpVar*.

GrpDecoupler_Model.

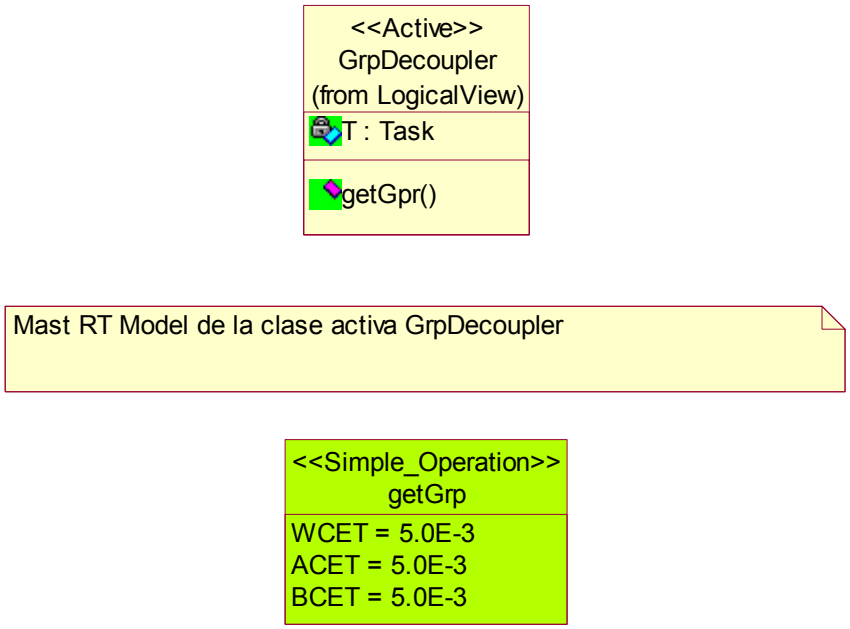


Figura 44

getGrp es un modelo de operación simple con una duración de 10ms. Esta operación será utilizada como intermediaria para comunicar el sistema con el servidor gráfico, para poder acceder al valor de la variable *GrpVar*.

4.6.1.4 Escenarios de tiempo real.

En la figura 45, se puede apreciar el escenario definido para este modelo. Se puede apreciar una diferencia clara con respecto al escenario definido en el Modelo4: la definición de una nueva transacción, *GrpTransaction*. En este apartado sólo se explicará la nueva transacción, ya que las otras tres transacciones definidas en el escenario se conservan exactamente igual a las definidas en el Modelo4.

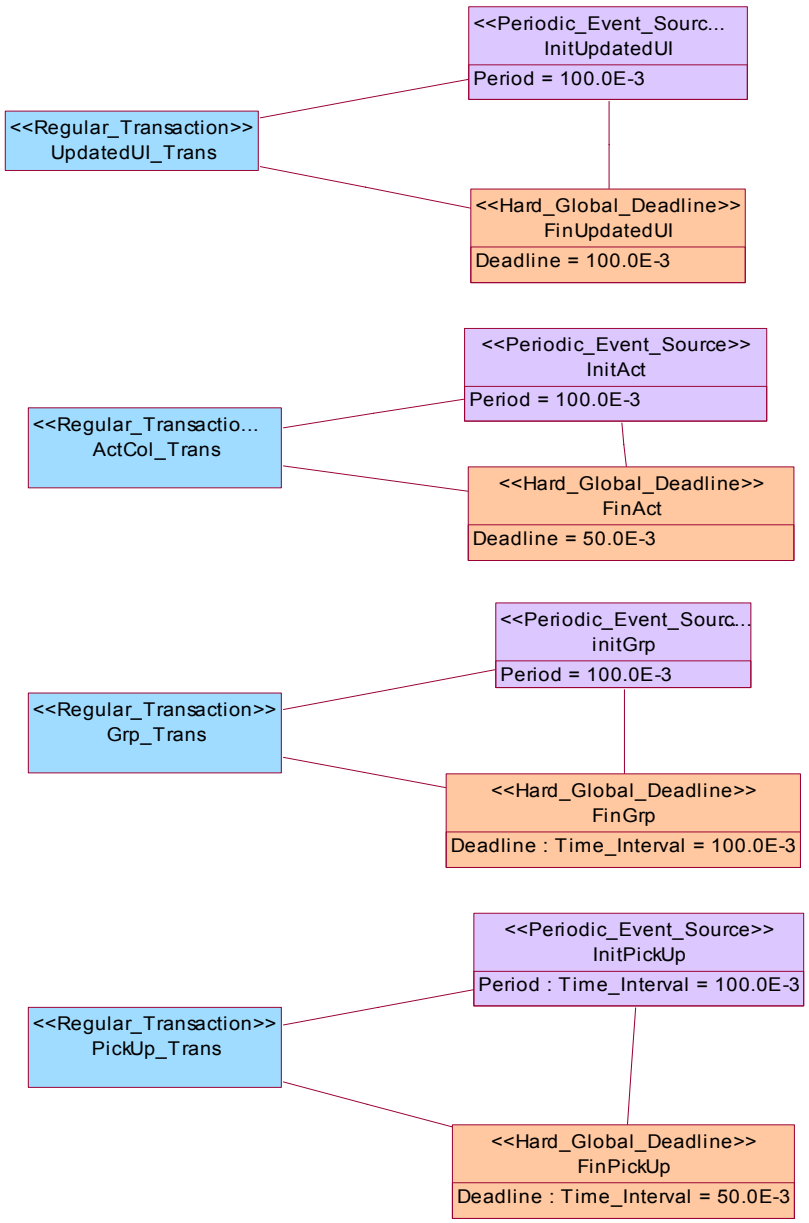


Figura 45

Grp_Transaction.

Esta transacción se va a encargar de ir recogiendo datos (en este caso una variable GrpVar) del servidor gráfico (*GrpServer*), para posteriormente guardarlos en el recurso compartido, dejándolos así a disposición del controlador de alto nivel para cuando le sean necesarios.

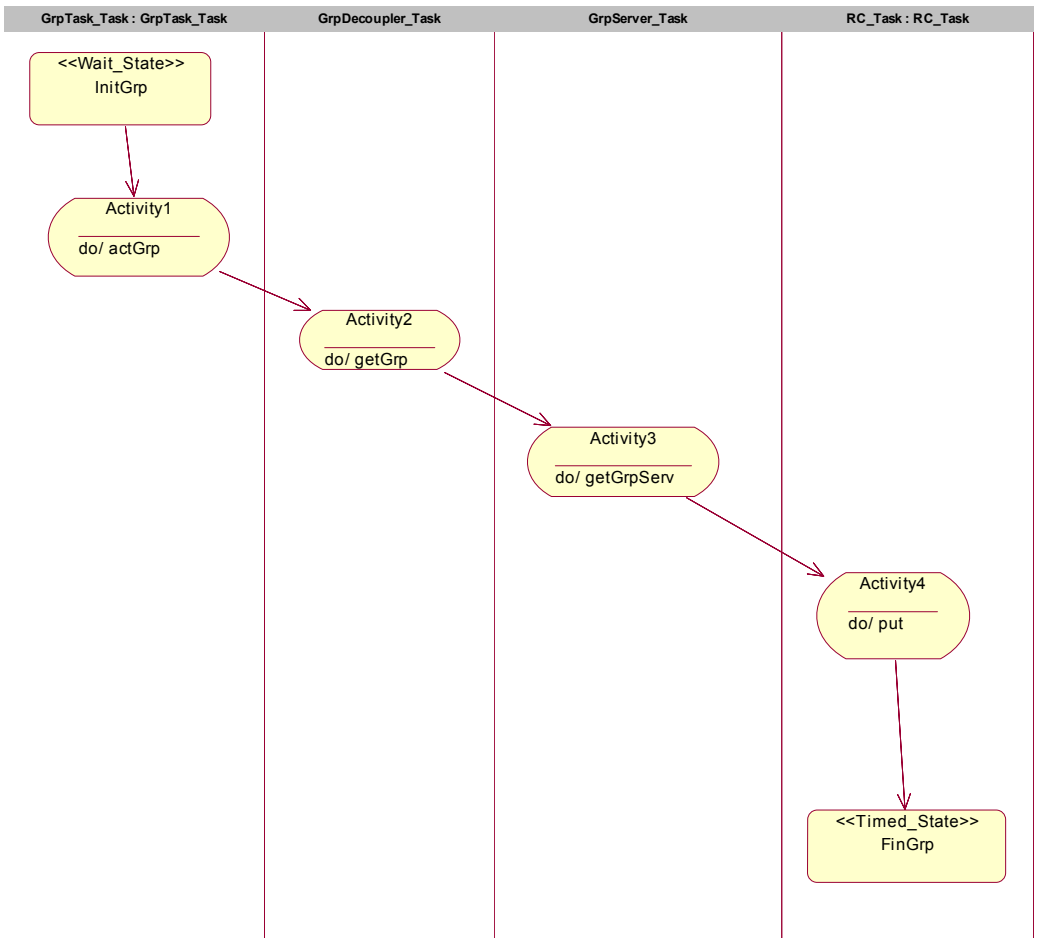


Figura 46

En la figura 46, se puede ver el diagrama de actividad de la transacción. En este diagrama se muestra la secuencia de operaciones que realiza la transacción. La transacción será iniciada desde la tarea *GrpTask* desde un *Wait_State*. Esto indica que la tarea estará a la espera de una señal de activación (*InitGrp*). En cuanto se produzca esta señal, el sistema empezará el proceso de comprobar el estado de los datos desde esta tarea. En primer lugar conseguirá los datos a través del *Grpdecoupler* (*getGrp*) que se comunicará con el servidor grafico, actuando como intérprete. Una vez obtenga el valor (*getGrpServ*), el *Grpdecoupler* escribirá este valor en la variable *GrpVar* del recurso compartido (*put*). Por último, se llega a un *Timed_State* (*FinGrp*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*, en este caso cada 100ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado

tiempo, este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistema.

4.6.2 Modelo5v2.

En este Modelo se va a introducir una variación importante respecto al Modelo5. En este caso, se definirá el recurso compartido como un objeto protegido.

4.6.2.1 Vista Lógica.

En la figura 47, se ve el diagrama de clases de este modelo y vemos que en el esta definido el recurso (*RC*) compartido como *Protected*, es decir, como un objeto protegido.

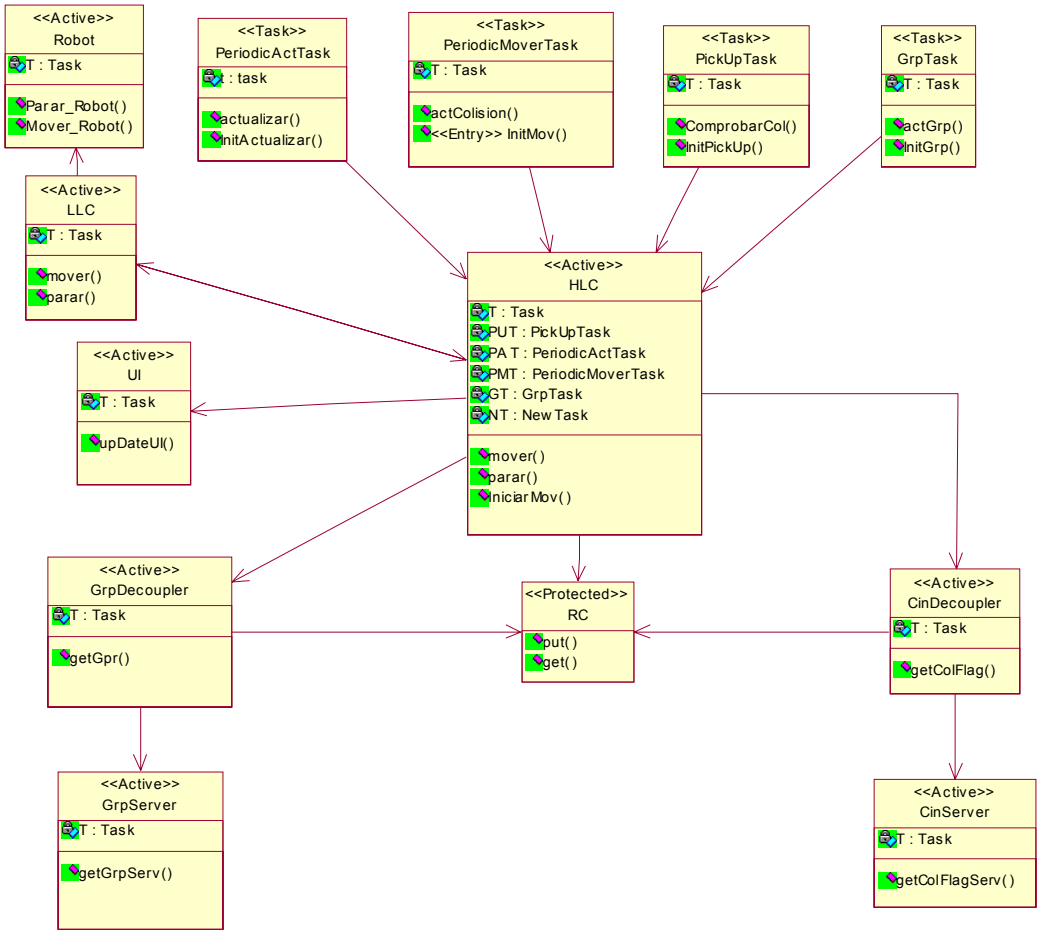


Figura 47

Los diagramas de secuencia de este modelo serán los mismos explicados en el Modelo5, ya que se realizarán los mismos procesos, sólo cambiarán el comportamiento del recurso compartido (*RC*), y luego la definición de la transacción.

4.6.2.2 Modelo de tiempo real de la plataforma.

Es el mismo modelo de plataforma definido en el Modelo5, con la salvedad de que en este caso no se definirá ningún *thread* para representar una instancia a la clase RC, ni por consiguiente su servidor de la política de planificación.

4.6.2.3 Modelo de tiempo real de los componentes lógicos.

Esta vista es la que va a sufrir una mayor variación, ya que las operaciones se definirán de acuerdo con las reglas de acceso a un objeto protegido según MAST. Los modelos lógicos que sufren variación son los siguientes.

CinDecoupler_Model.

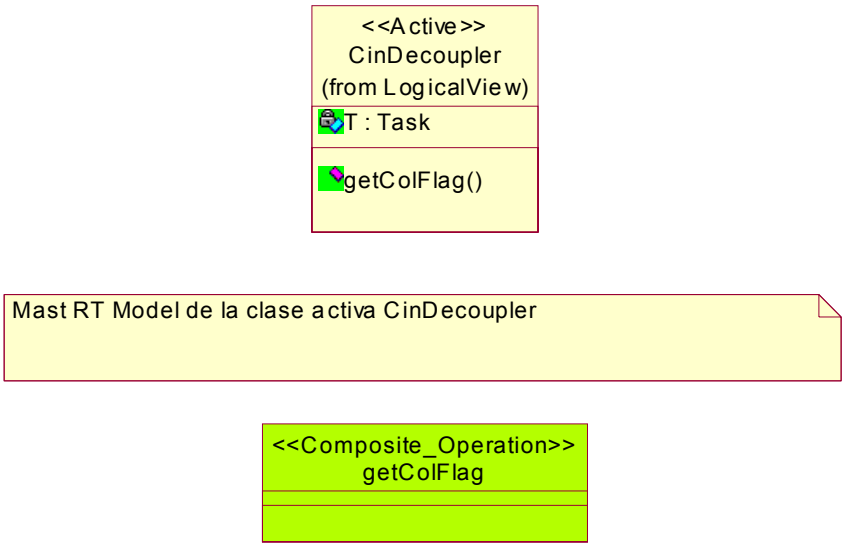


Figura 48

En este caso se va a definir la operación **getColFlag** como una operación compuesta, ya que en este caso deberá acceder al recurso compartido después de haber realizado otra operación. En la figura 49, se ve el diagrama de actividad que explica las operaciones que realiza esta operación compuesta.

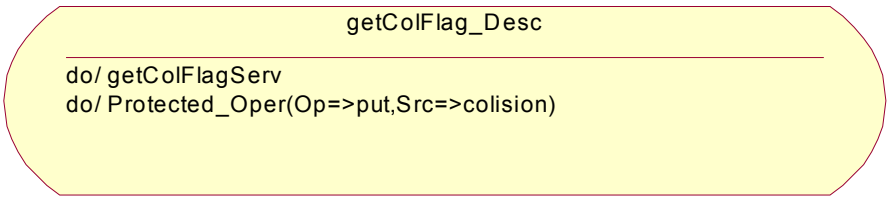


Figura 49

En primer lugar llamará al método *getColFlagServ* para poder acceder al valor del *flag* de colisión, y una vez obtenido accederá en régimen exclusivo al recurso compartido para poder actualizar allí el valor de la variable *colision*.

GrpDecoupler_Model.

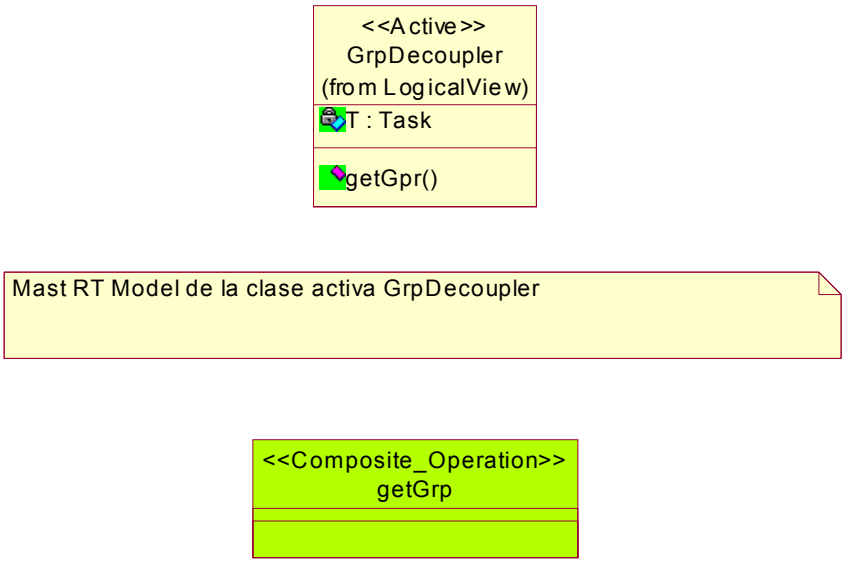


Figura 50

La operación **getGrp** se define como una operación compuesta ya que en este caso deberá acceder al recurso compartido después de haber realizado otra operación. En la figura 51, se muestran las operaciones que realiza esta operación compuesta.

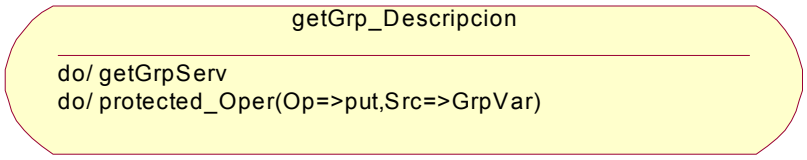


Figura 51

En primer lugar llamará al método *getGrpServ* para poder acceder al valor, que debe ser actualizado, del servidor grafico (*GrpServer*), y una vez obtenido accederá en régimen exclusivo al recurso compartido para poder actualizar allí el valor de la variable *GrpVar*.

PickUpTask_Model.

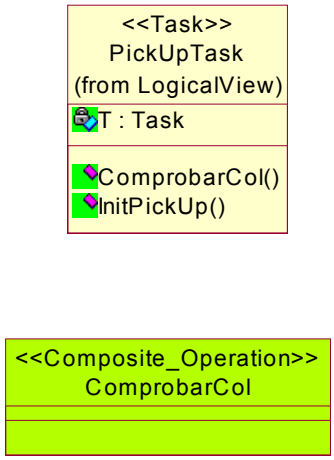


Figura 52

La operación ***ComprobarCol*** es una operación compuesta, ya que en este caso deberá acceder al recurso compartido después de haber realizado otra operación. En la figura 53, se ve el diagrama de actividad que explica la operaciones que realiza esta operación compuesta.

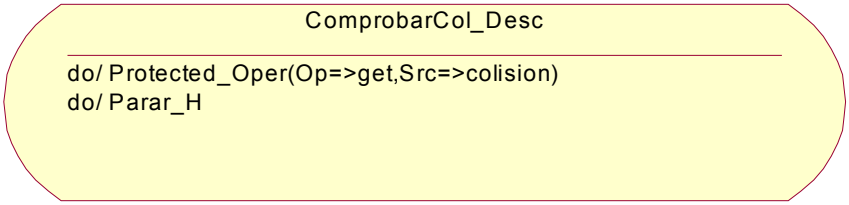


Figura 53

Accederá en régimen exclusivo al recurso compartido y comprobará el valor de la variable *colision*. Como sólo se va a estudiar el peor caso se da por supuesto que está activada, con lo cual posteriormente ordenará parar al sistema.

RC_Model.

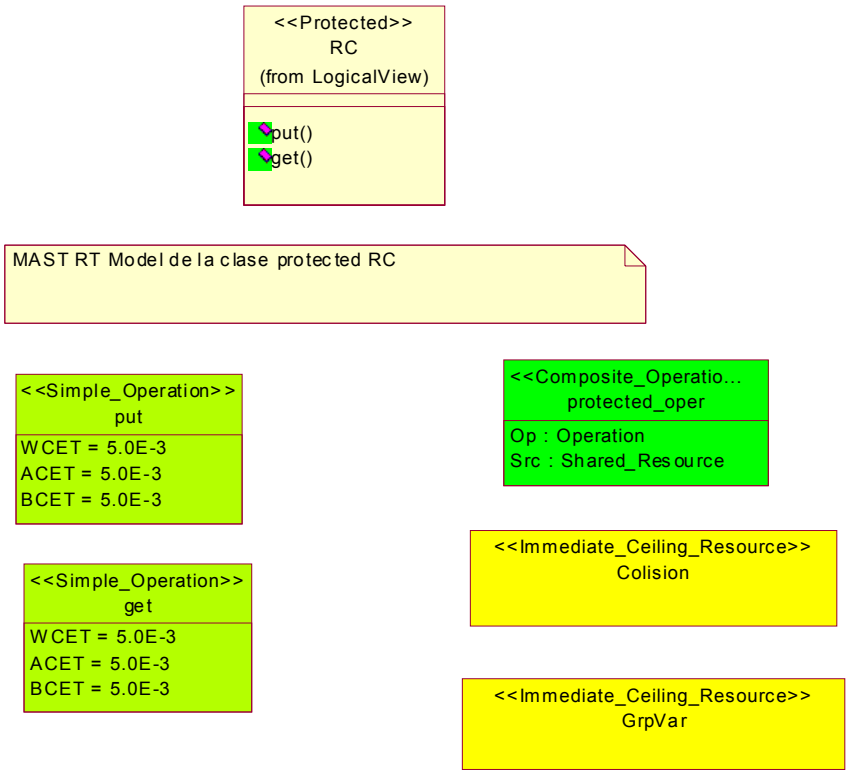


Figura 54

En la figura 54, vemos los componentes lógicos definidos para este objeto compartido. Las operaciones ***put*** y ***get*** van a realizar la misma función que está definida en el Modelo3. En la figura 54, se han definido dos objetos del tipo *Immediate_Celing_Resource.* Éstos representan los recursos compartidos a los cuales deben acceder los componentes del sistema: ***colision*** que será el recurso donde se almacene el estado del *flag* de colisión, y ***GrpVar*** que será donde se almacene los datos que provienen del servidor gráfico (*GrpServer*).

También se observa la operación compuesta **protecter_Oper**. Esta operación representará la forma de acceso al objeto protegido. En la figura 55, se ve como en primera instancia se reserva el recurso al que se desea acceder por medio de la operación primitiva ***Lock***, después se ejecutará la operación que se vaya a aplicar sobre el recurso ***Op*** y finalmente se libera el recurso ***UnLock***.

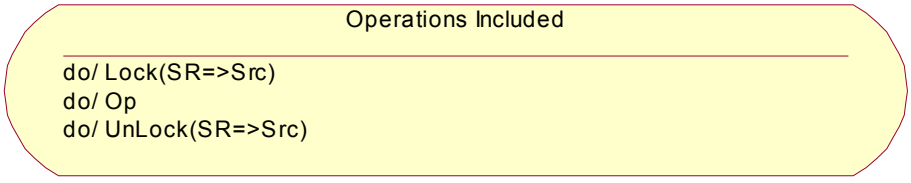


Figura 55

4.6.2.4 Escenarios de tiempo real.

En la figura 56, se ve el escenario planteado para este modelo. Este escenario se compone de tres transacciones, que son: *UpDateUI_Transaction*, *Mover_Transaction*, *PickUp_Transaction* y *Grp_Transaction*.

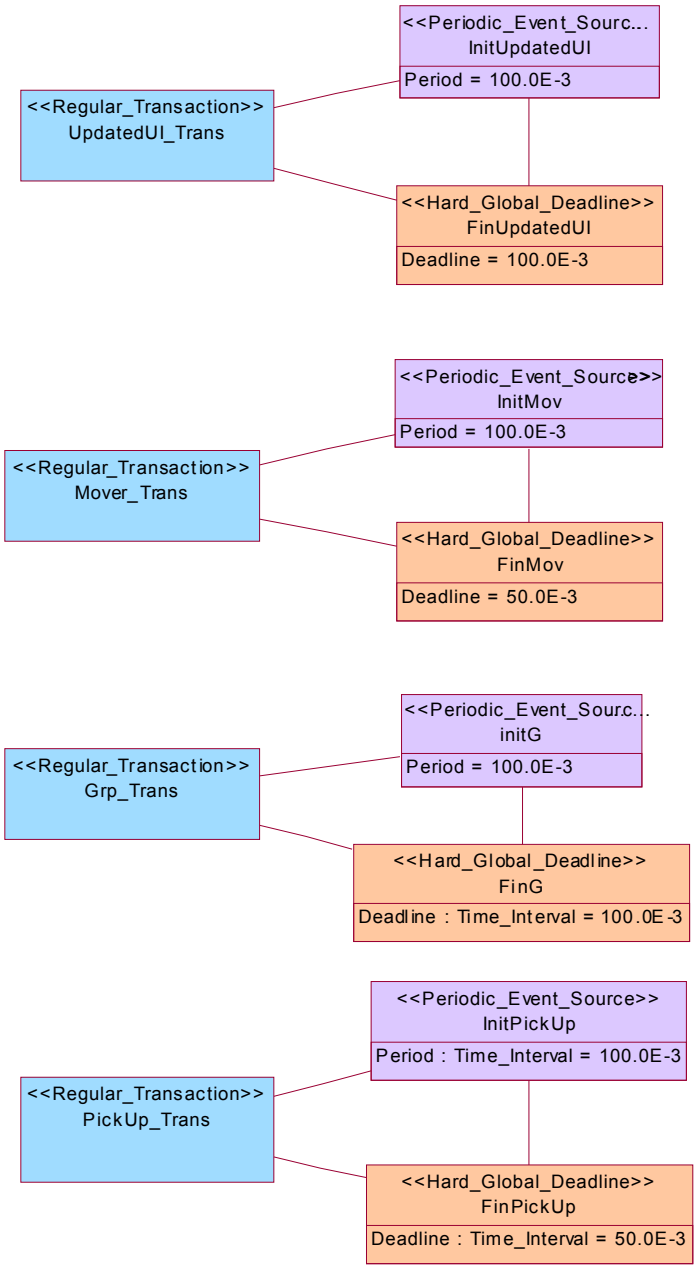


Figura 56

La transacción *UpDateUI_Transaction* sigue siendo la misma que se definió en el Modelo1. Ahora pasamos a explicar el resto de transacciones del escenario.

Mover_Trasnsaction.

Va a tener la misma funcionalidad que la transacción ActCol_Transaction, definida en el Modelo5, pero el diagrama de actividad va a ser diferente debido a los cambios de los componentes lógicos. El diagrama de actividad se puede ver en la figura 57.

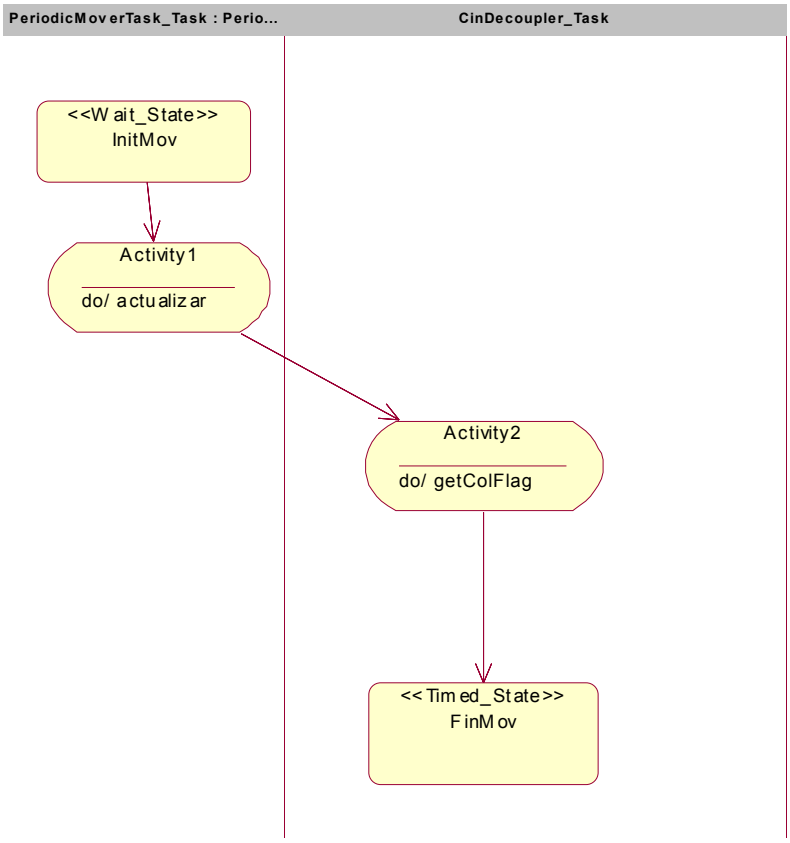


Figura 57

Esta transacción empieza en un estado de espera (*Wait_State*) denominado ***InitMover***. En cuanto se reciba esta señal, dará comienzo del proceso por medio de la operación ***actualizar*** que empezará el proceso de actualización. Posteriormente se ejecutará la actividad ***getColFlag***, la cual como se explicó en el apartado de los componentes lógicos, se encargará de acceder al estado del *flag* de colisión del servidor de cinemática (*CinServer*) para después acceder al recurso compartido y guardarlo en él. Por último, se llega a un *Timed_State* (*FinMover*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*. En este caso cada 50ms, dándole un tiempo límite para se produzca la transacción, en caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

Grp_Transaction.

Va a tener la misma funcionalidad que la transacción Grp_Transaction, definida en el Modelo5, pero el diagrama de actividad va a ser diferente debido a los cambios de los componentes lógicos. El diagrama de actividad se puede ver en la figura 58.

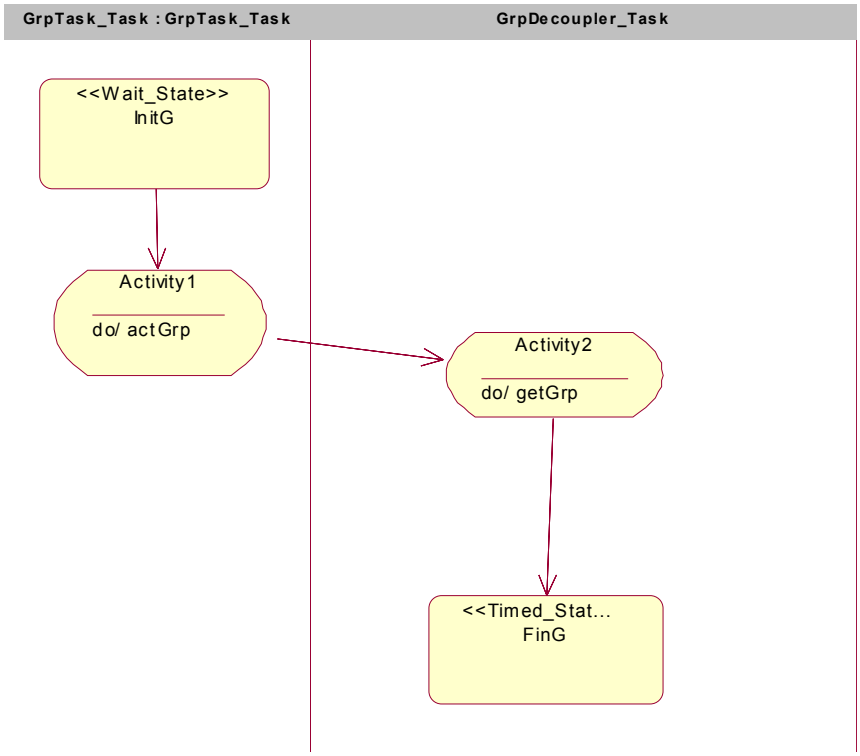


Figura 58

Esta transacción empieza en un estado de espera (*Wait_State*) denominado ***InitGrp***. En cuanto se reciba esta señal, dará comienzo del proceso por medio de la operación ***actGrp*** que empezará el proceso de actualización. Posteriormente, se ejecutará la actividad ***getGrp***, la cual como se explicó en el apartado de los componentes lógicos, se encargará de acceder a los datos del servidor gráfico (*GrpServer*) para después acceder al recurso compartido y guardarlo en él. Por último, se llega a un *Timed_State* (*FinGrp*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline,* en este caso cada 100ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

PickUp_Transaction.

Va a tener la misma funcionalidad que la transacción PickUp_Transaction, definida en el Modelo5, pero el diagrama de actividad va a ser diferente debido a los cambios de los componentes lógicos. El diagrama de actividad se puede ver en la figura 59.

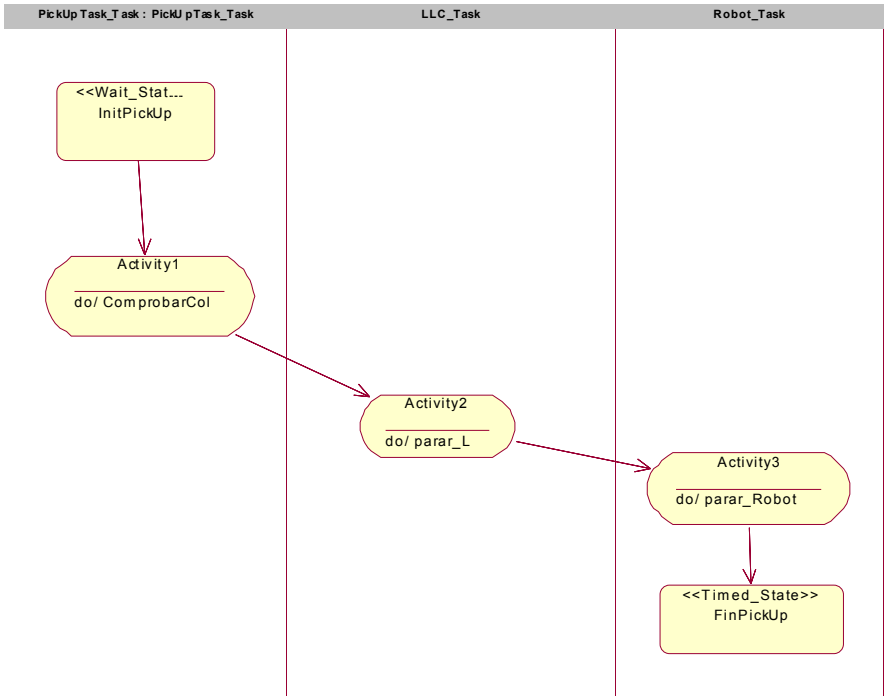


Figura 59

Esta transacción empieza en un estado de espera (*Wait_State*) denominado ***InitPickUp***. En cuanto se reciba esta señal, dará comienzo del proceso por medio de la operación ***ComprobarCol*** que empezará el proceso de comprobación. Esta operación ordenará parar el sistema en caso de estar activado el valor de *colision* mandará parar al controlador de alto nivel, y a su vez éste mandará detener al controlador de bajo nivel (*parar_L*). Finalmente la orden de parar llegará al robot (parar_Robot). Por último, se llega a un *Timed_State* (*FinGrp*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*, en este caso cada 100ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

4.7 Prueba6.

4.7.1 Modelo6.

En este modelo se va a realizar una ampliación respecto al Modelo4, se le van a añadir nuevos componentes y se aumentará la complejidad del escenario.

4.7.1.1 Vista Lógica.

En la figura 60, se observa el diagrama de clases correspondiente al Modelo6, en el cual se observa, en comparación al diagrama del modelo anterior, la inclusión de dos nuevos componentes *NewServer* y *NewDecoupler*, los cuáles formarán un nuevo bloque en el sistema.

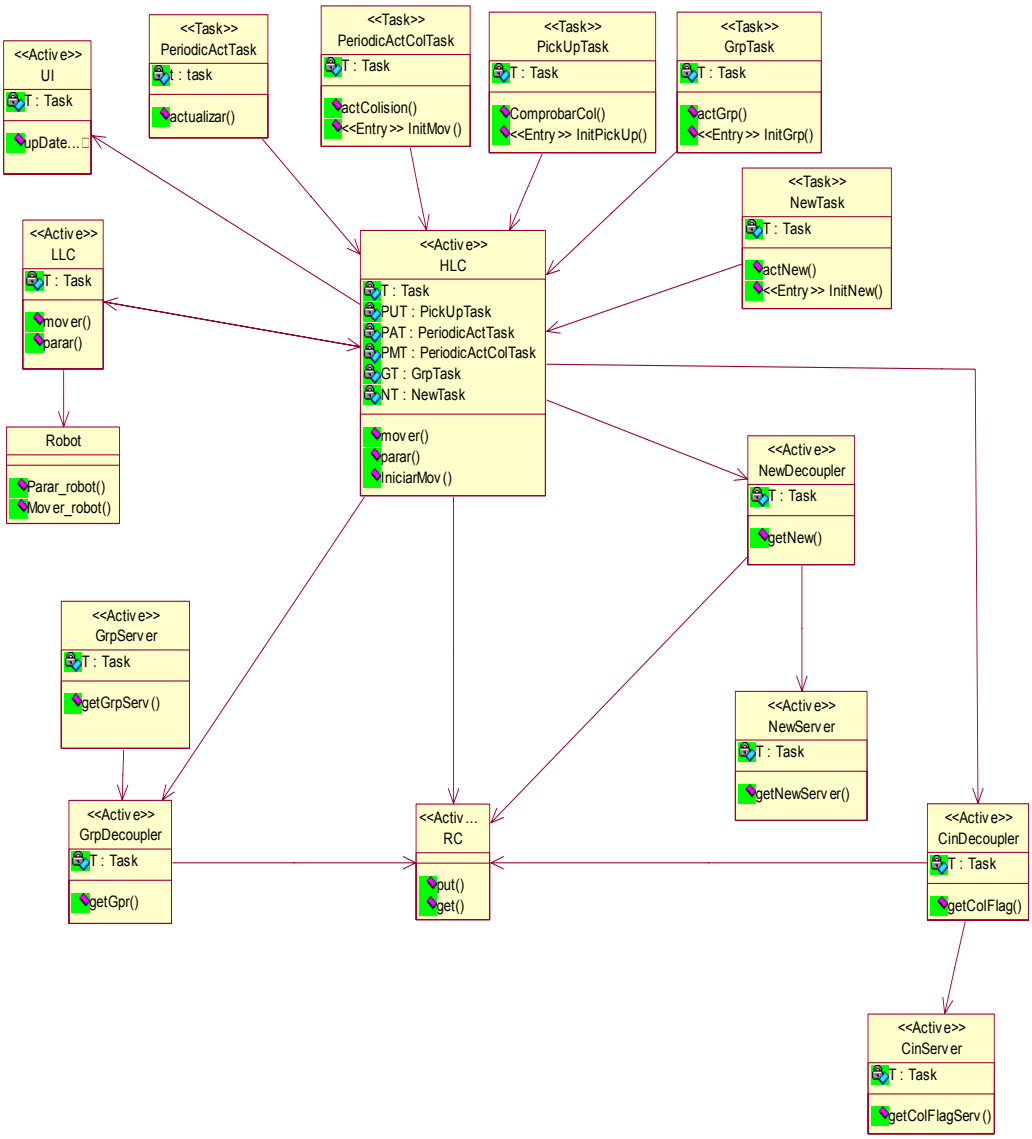


Figura 60

Al observar la figura se destaca la inclusión de un nuevo bloque compuesto por un servidor (*Newserver*) y un módulo desacoplador (*NewDecoupler*) que lo ayudará a comunicarse con el resto del sistema. Estos dos componentes no representarán a un servicio concreto sino más bien se trata de un bloque genérico que se utilizará para que en el estudio de este modelo se tenga en cuenta un caso con una transacción más, poniendo así más exigencias a las capacidades temporales de la arquitectura. También se puede observar la aparición de una nueva clase tipo *Task* denominada *NewTask*. La inclusión de este nuevo bloque afectará al análisis temporal del sistema ya que implicará una nueva transacción, aumentando así la dificultad del escenario planteado para el mismo.

La adición de estos nuevos componentes va a ir acompañado del diseño de una transacción que hará uso de un nuevo diagrama de secuencia para representar el proceso, denominado New_Process, y que se observa en la figura 61.

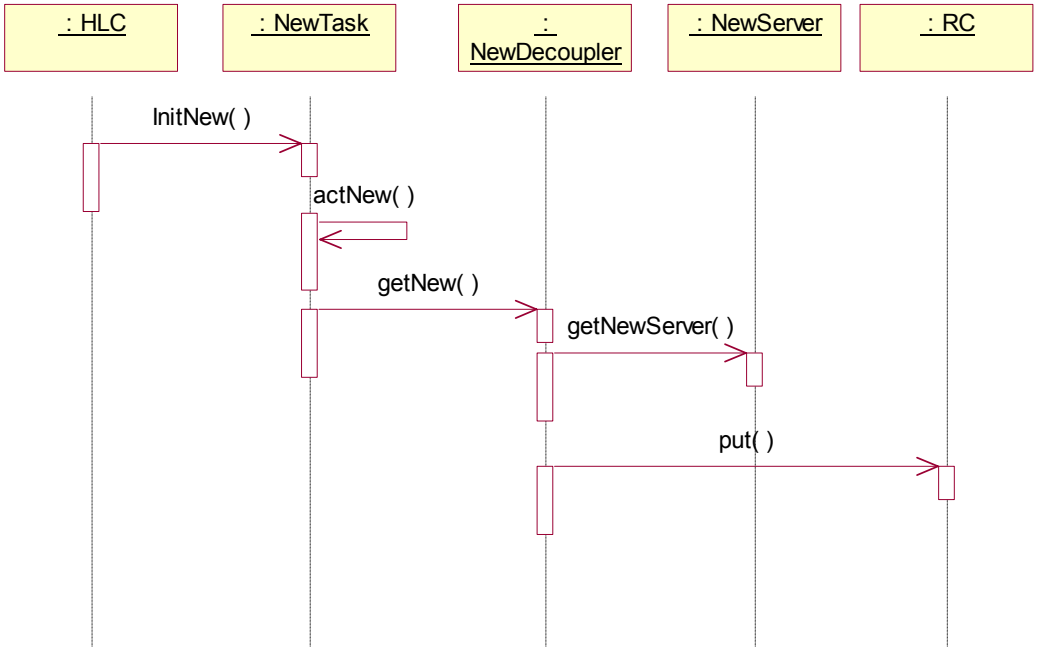


Figura 61 Diagrama de secuencia New_Process

4.7.1.2 Vista lógica de la plataforma.

Esta vista va a sufrir solamente una leve modificación. Se le van a añadir los componentes *Newdecoupler_Task*, *NewDecoupler_SP, NewServer_Task, NewServer_SP, NewTask_Task* y *NewTask_SP,* que se observan en la figura 62.

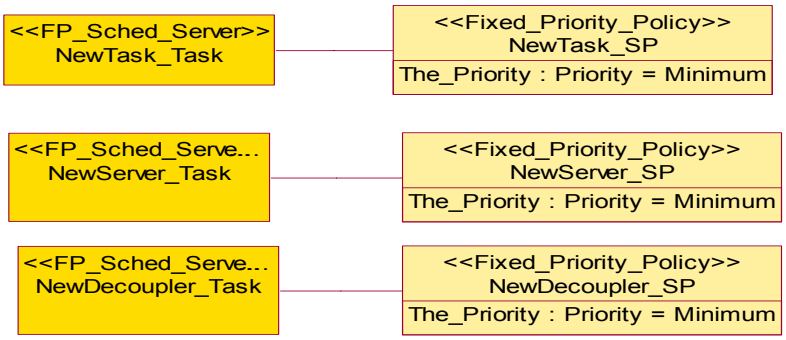


Figura 62

- **NewDecoupler_Task**: define un *thread* que introduce la instancia a la clase *Newdecoupler*. Este *thread* se encarga de controlar los eventos que ocurran sobre el *Newdecoupler*.
- **Newdecoupler_SP**: define al *server* que planifica las actividades de *Newdecoupler_Task* con una prioridad fija que dejaremos que MAST se la asigne en el análisis según RMA.
- **NewServer_Task**: *Thread* que introduce la instancia de la clase *NewServer*. Este *thread* se encarga de controlar los eventos que ocurran sobre el servidor de cinemática.
- **NewServer_SP**: Es el *server* que planifica las actividades del *New_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.
- **NewTask_Task**: *Thread* que introduce la instancia de la clase *NewTask*. Este *thread* se encarga de controlar los eventos que ocurran sobre el *NewTask*.
- **NewTask_SP**: Es el *server* que planifica las actividades del *NewTask_Task* con una prioridad fija, que dejaremos que MAST se la asigne en el análisis según RMA.

4.7.1.3 Modelo de tiempo real de los componentes lógicos.

En esta vista vamos a explicar los diferentes modelos lógicos que hemos añadido a causa de los nuevos componentes que componen este modelo del sistema.

NewTask_Model.

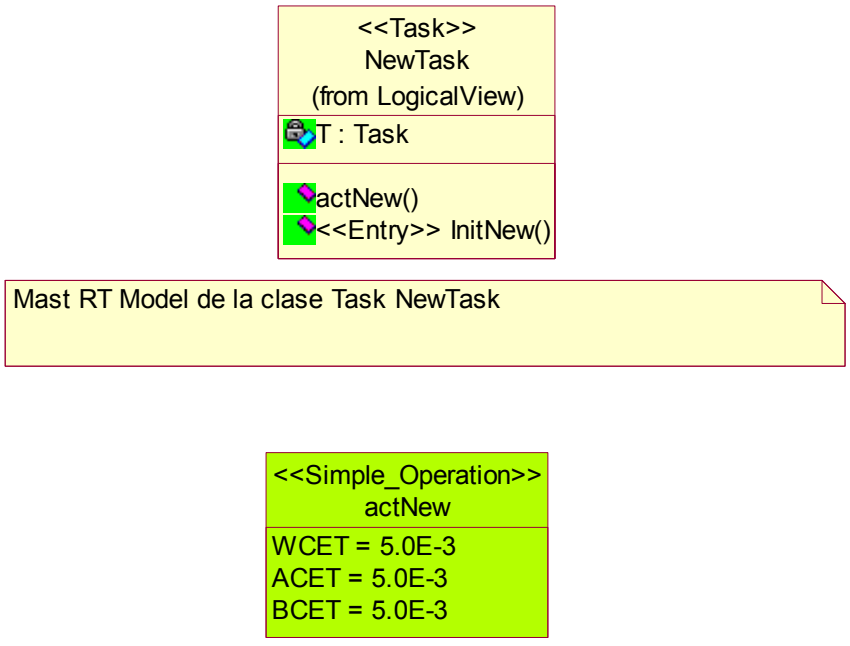


Figura 63

ActNew es un modelo de operación simple con una duración de 10ms. Esta operación está definida dentro de esta tarea, la cual la llama de manera periódica. *ActNew* se encargará de iniciar el proceso por el cual el controlador de alto nivel comprobará en el recurso compartido el valor de la variable *NewVar*, que es la variable con la que va a trabajar el servidor *NewServer.*

NewServer_Model.

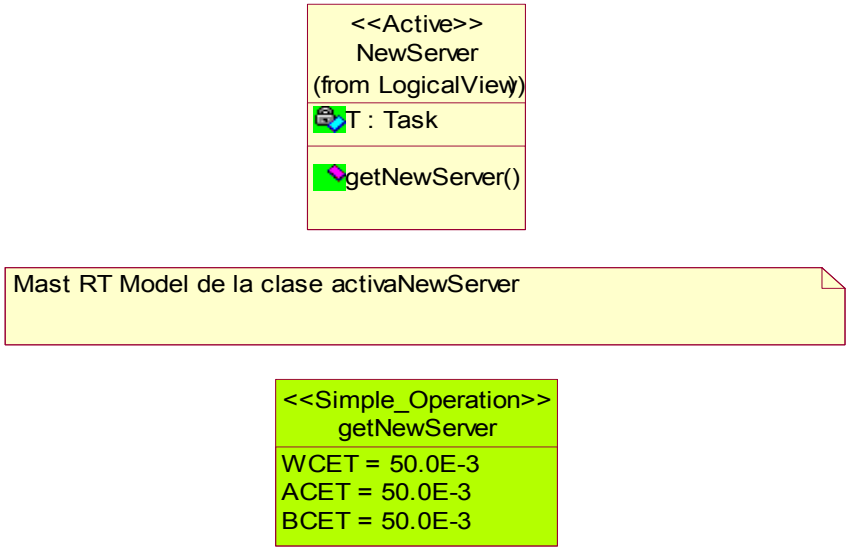


Figura 64

ActNewServ es un modelo de operación simple con un tiempo de ejecución de 50ms. Es un tiempo de ejecución alto debido a que se trata de una operación de acceso a una variable del servidor *Newserver*. Esta operación permitirá conocer el estado de dicha variable, *NewVar*.

GrpDecoupler_Model.

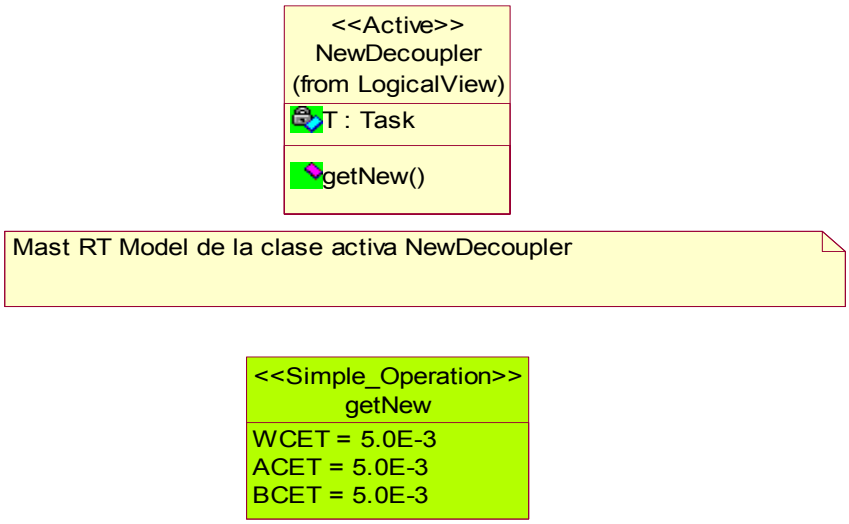


Figura 65

getNew es un modelo de operación simple con una duración de 10ms. Esta operación será utilizada como intermediaría para comunicar el sistema con el servidor *NewServer*, para poder acceder al valor de la variable *NewVar*.

4.7.1.4 Escenarios de tiempo real.

En la figura 66, se puede apreciar el escenario definido para este modelo. Se puede apreciar una diferencia clara con respecto al escenario definido en el Modelo5. La definición de una nueva transacción, *NewTransaction*. En este apartado sólo se va a proceder a la explicación de esta nueva transacción definida, ya que las otras tres transacciones definidas en el escenario se conservan exactamente igual a las definidas en el Modelo5.

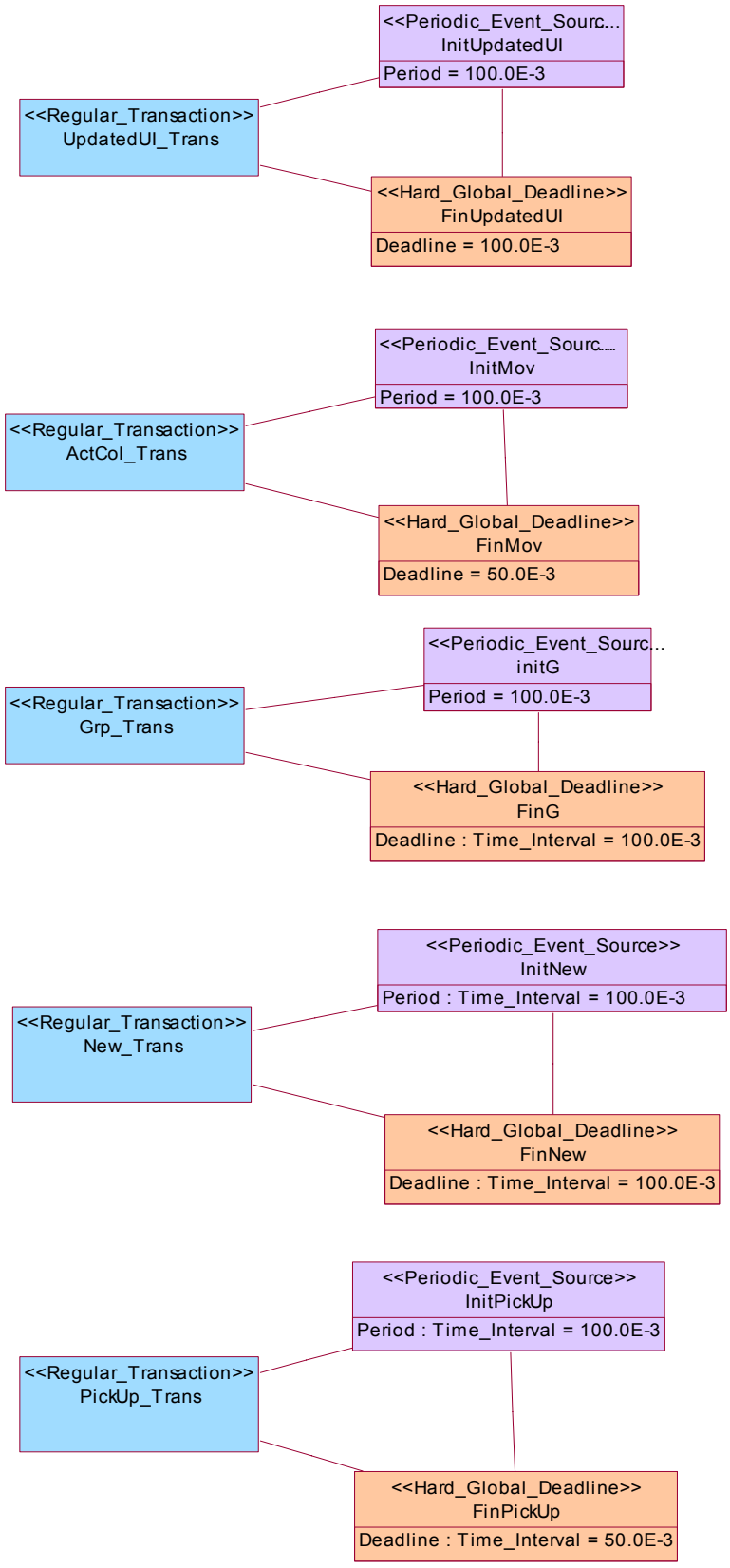

<<Periodic_Event_Sourc...
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
<<Periodic_Event_Sourc...
InitMov
Period = 100.0E-3
<<Regular_Transaction>>
ActCol_Trans
<<Hard_Global_Deadline>>
FinMov
Deadline = 50.0E-3
<<Periodic_Event_Sourc...
initG
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinG
Deadline : Time_Interval = 100.0E-3
<<Periodic_Event_Source>>
InitNew
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
New_Trans
<<Hard_Global_Deadline>>
FinNew
Deadline : Time_Interval = 100.0E-3
<<Periodic_Event_Source>>
InitPickUp
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
PickUp_Trans
<<Hard_Global_Deadline>>
FinPickUp
Deadline : Time_Interval = 50.0E-3

Figura 66

New_Transaction.

Esta transacción se va a encargar de ir recogiendo datos (en este caso una variable *NewVar*) del servidor *NewServer* para posteriormente guardarlos en el recurso compartido, dejándolos así a disposición del controlador de alto nivel para cuando le sean necesarios.

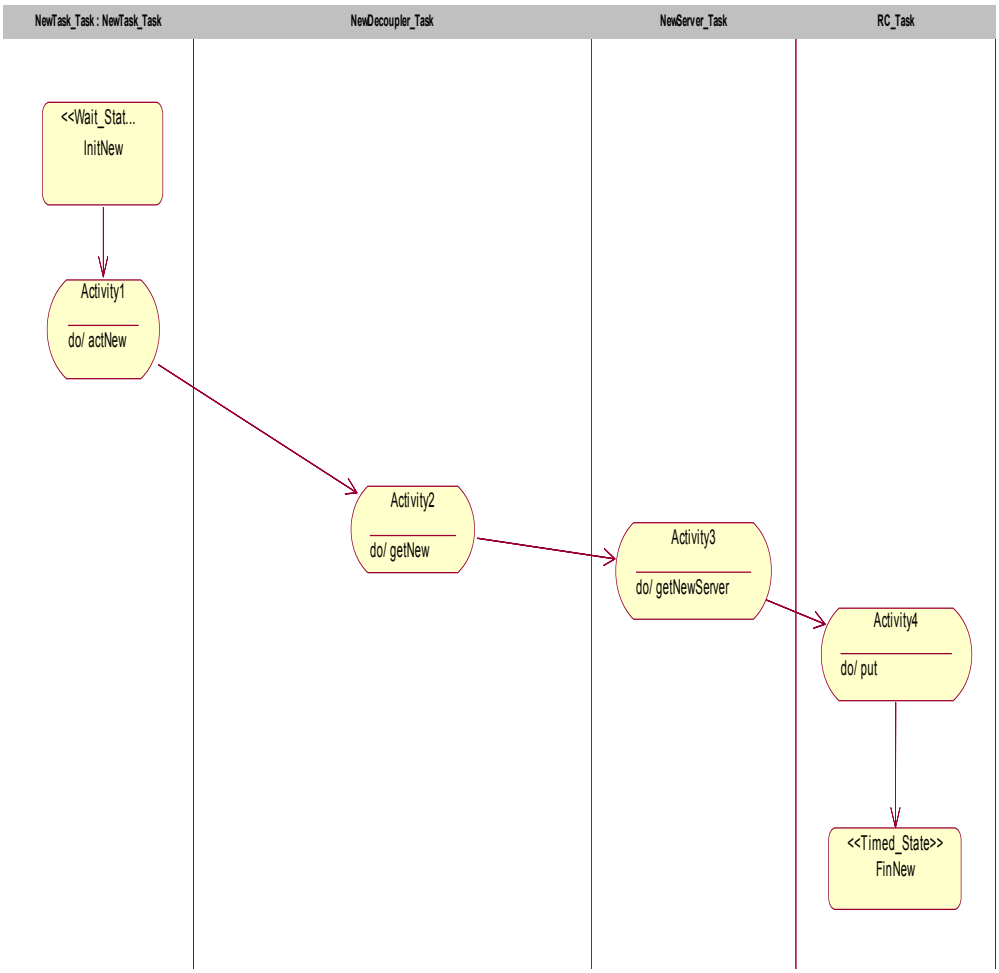


Figura 67

En la figura 67, se puede ver el diagrama de actividad de la transacción. En este diagrama se muestra la secuencia de operaciones que realiza la transacción que será iniciada desde la tarea *NewTask* desde un *Wait_State*. Esto indica que la tarea estará a la espera de una señal de activación (*InitNew*). En cuanto se produzca esta señal, el sistema empezará el proceso de comprobar el estado de los datos desde esta tarea. En primer lugar conseguirá los datos a través del *Newdecoupler* (*getNew*) que se comunicará con el servidor *NewServer*, actuando como intérprete, y una vez obtenga el valor (*getNewServ*), el *Newdecoupler* escribirá este valor en la variable *NewVar* del recurso compartido (*put*). Por último, se llega a un *Timed_State* (*FinNew*). Este estado espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*, en este caso cada 100ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

4.7.2 Modelo6v2.

En este versión2 del Modelo6, se cambia al objeto activo RC, que representa al recurso compartido, por un objeto protegido, tal y como se ha hecho en la versión2 del Modelo5.

4.7.2.1 Vista Lógica.

En la figura 68, se puede observar el diagrama de clases correspondiente a este modelo. En él se aprecia una única diferencia respecto a al mostrado en el Modelo6, que es la definición del recurso compartido (*RC*) como un objeto protegido (Protected Object).

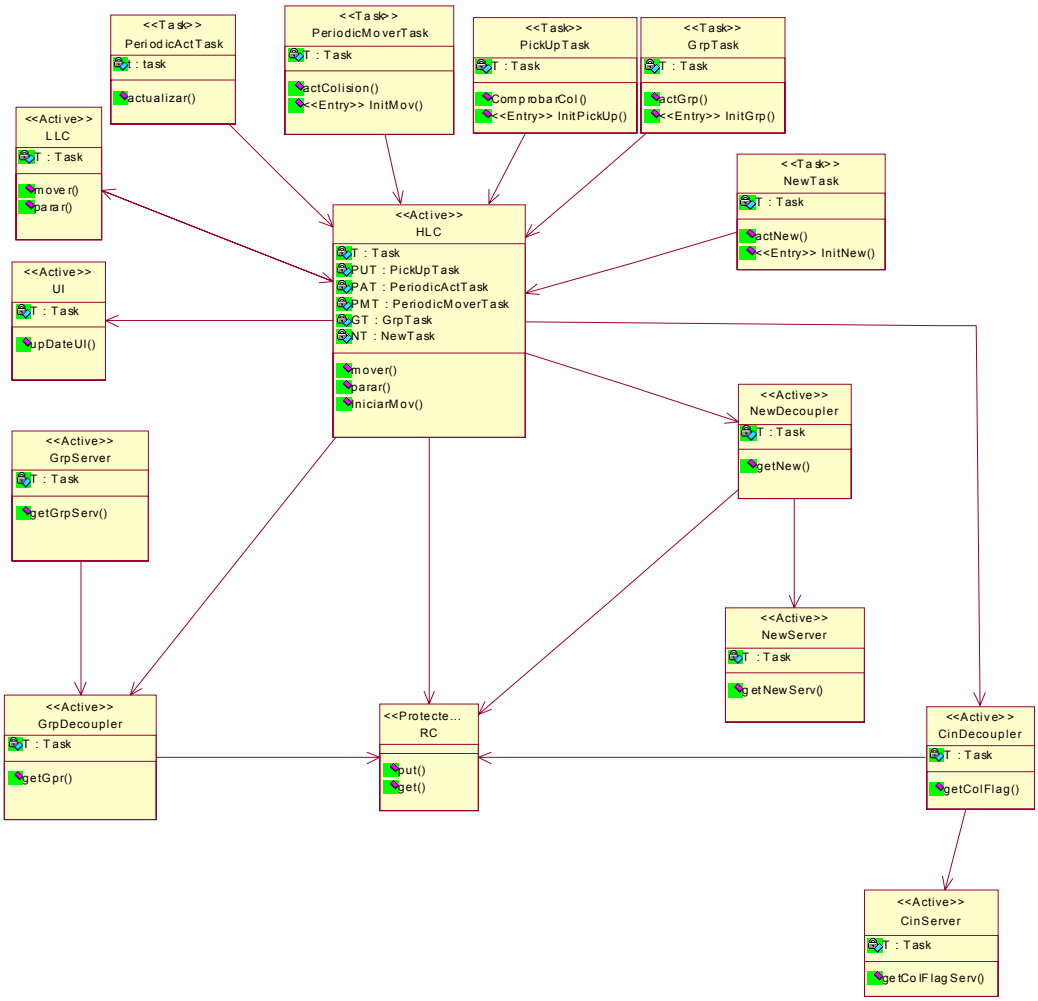


Figura 68

En este modelo, los diagramas de secuencia van a ser los mismos que los definidos en el Modelo6, ya que los procesos van a ser los mismos y solo cambiará la forma de acceso a los datos del recurso compartido, lo cual se explicará en la vista de los componentes lógicos.

4.7.2.2 Modelo de tiempo real de la plataforma.

Es el mismo modelo de plataforma definido en el Modelo6, con la salvedad de que en este caso no se definirá los componentes que definen el *thread* que representa una instancia a la clase RC, ni su servidor de la política de planificación.

4.7.2.3 Modelo de tiempo real de los componentes lógicos.

Para explicar este apartado es mejor tomar como referencia el Modelo5v2 ya que los componentes lógicos de ese modelo se corresponden totalmente con los del modelo que se explica en este apartado. Tomando el Modelo5v2 de partida solo seria necesario añadir a su modelo lógico, el nuevo bloque que representaría el nuevo desacoplador (*NewDecoupler*) y el nuevo servidor (*NewServer*).

RC_Model.

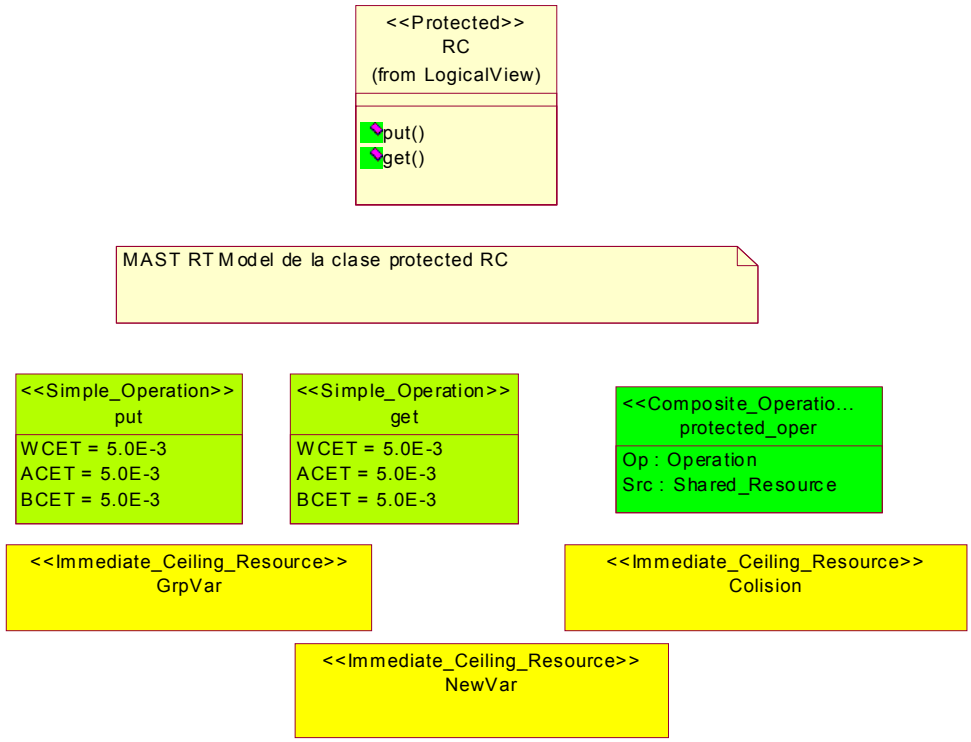


Figura 69

En este modelo solamente se introduce un nuevo cambio respecto al explicado en el Modelo5v2, que es la inclusión de un nuevo *Immediate_Celing_Resource* (*NewVar*), que representa a una nueva variable compartida.

NewDecoupler_Model.

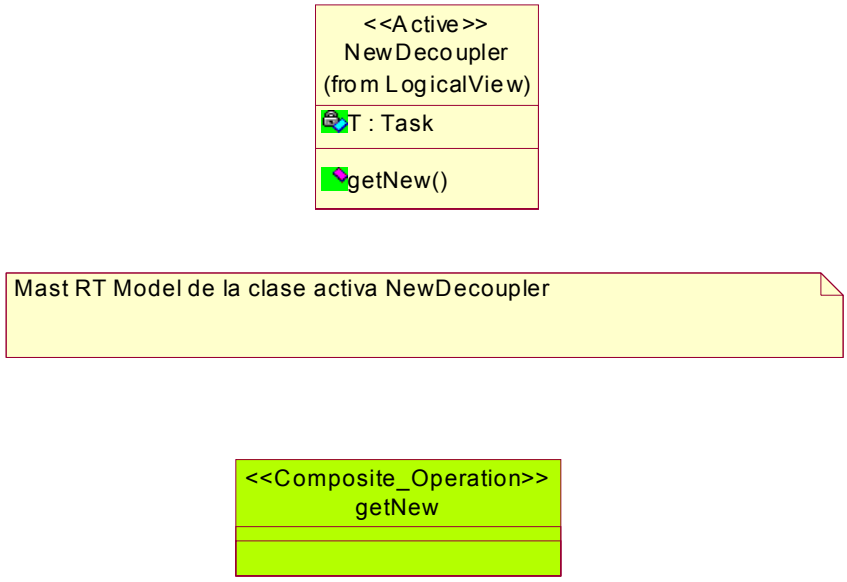


Figura 70

En este caso se va a definir la operación **getNew** como una operación compuesta, ya que deberá acceder al recurso compartido después de haber realizado otra operación. En la figura 49, se ve el diagrama de actividad que explica las operaciones que realiza esta operación compuesta.

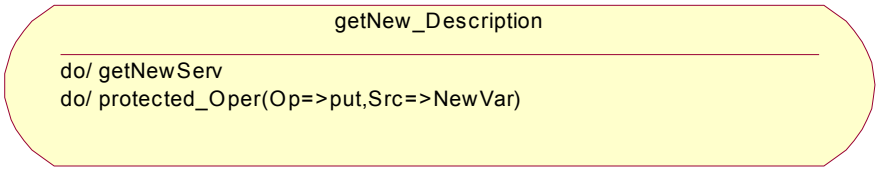


Figura 71

En primer lugar llamará al método ***getNewServ*** para poder acceder al valor del *flag* de colisión, y una vez obtenido accederá en régimen exclusivo al recurso compartido para poder actualizar allí el valor de la variable ***NewVar***.

El resto de modelos lógicos del sistema son los mismos que fueron ya definidos para el Modelo5v2, excepto *NewServer_Model* y *NewTask_Model*, que han sido definidos para el Modelo6.

4.7.2.4 Escenario de tiempo real.

En la figura 71, se ve el escenario definido para este modelo. En el se pueden observar las transacciones que lo componen, que son: *UpDateUi_Transaction*, *Mover_Transaction*, *PickUp_Transaction*, *Grp_Transaction* y *New_Transaction.*

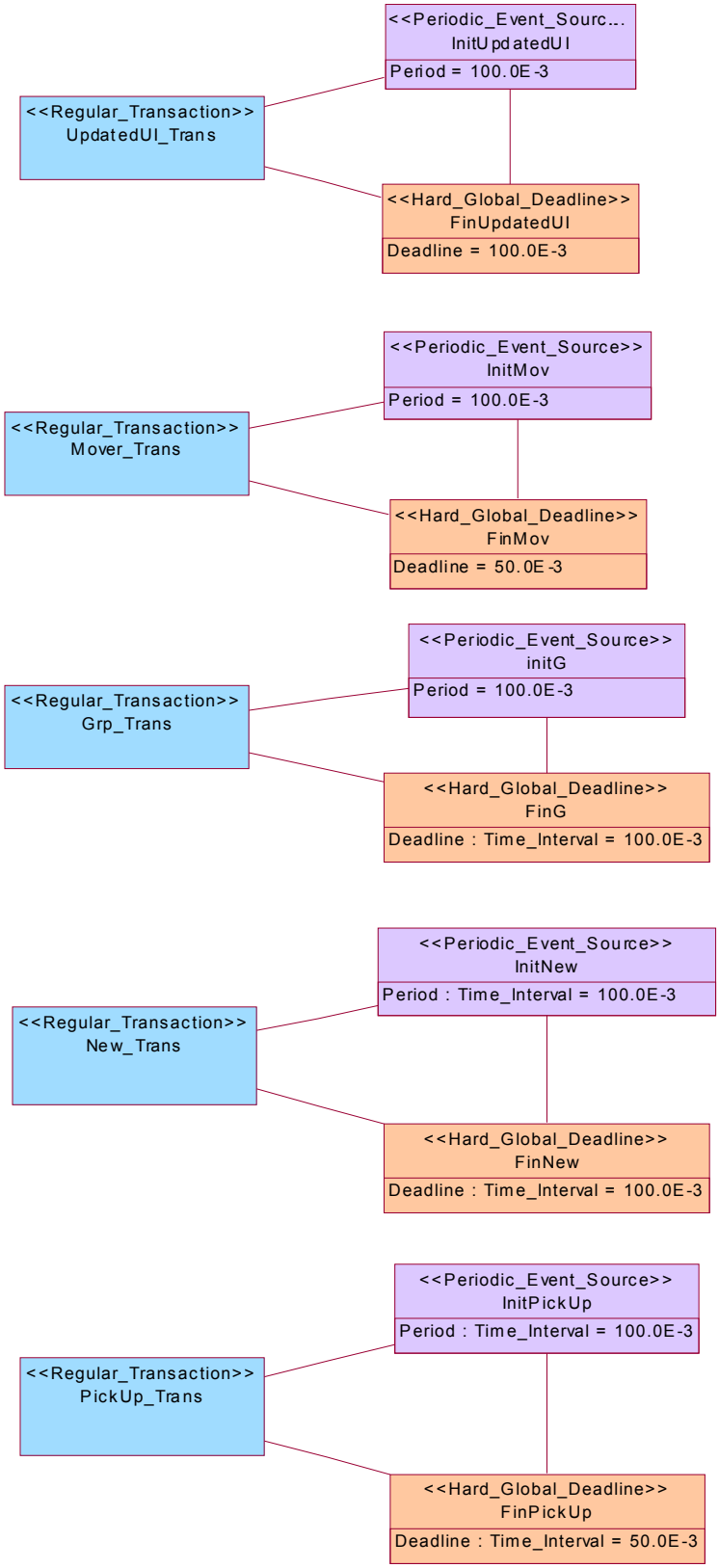

<<Periodic_Event_Sourc...
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
<<Periodic_Event_Source>>
InitMov
Period = 100.0E-3
<<Regular_Transaction>>
Mover_Trans
<<Hard_Global_Deadline>>
FinMov
Deadline = 50.0E-3
<<Periodic_Event_Source>>
initG
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinG
Deadline : Time_Interval = 100.0E-3
<<Periodic_Event_Source>>
InitNew
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
New_Trans
<<Hard_Global_Deadline>>
FinNew
Deadline : Time_Interval = 100.0E-3
<<Periodic_Event_Source>>
InitPickUp
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
PickUp_Trans
<<Hard_Global_Deadline>>
FinPickUp
Deadline : Time_Interval = 50.0E-3

Figura 72

Estas cuatro transacciones, *UpDateUi_Transaction*, *Mover_Transaction*, *PickUp_Transaction* y *Grp_Transaction,* van a ser las mismas explicadas en el Modelo5v2. Por eso, se procede a explicar únicamente la transacción *New_Transaction*.

New_Transaction.

Va a tener la misma funcionalidad que la transacción New_Transaction, definida en el Modelo6, pero el diagrama de actividad va a ser diferente debido a los cambios de los componentes lógicos. El diagrama de actividad se puede ver en la figura 73.

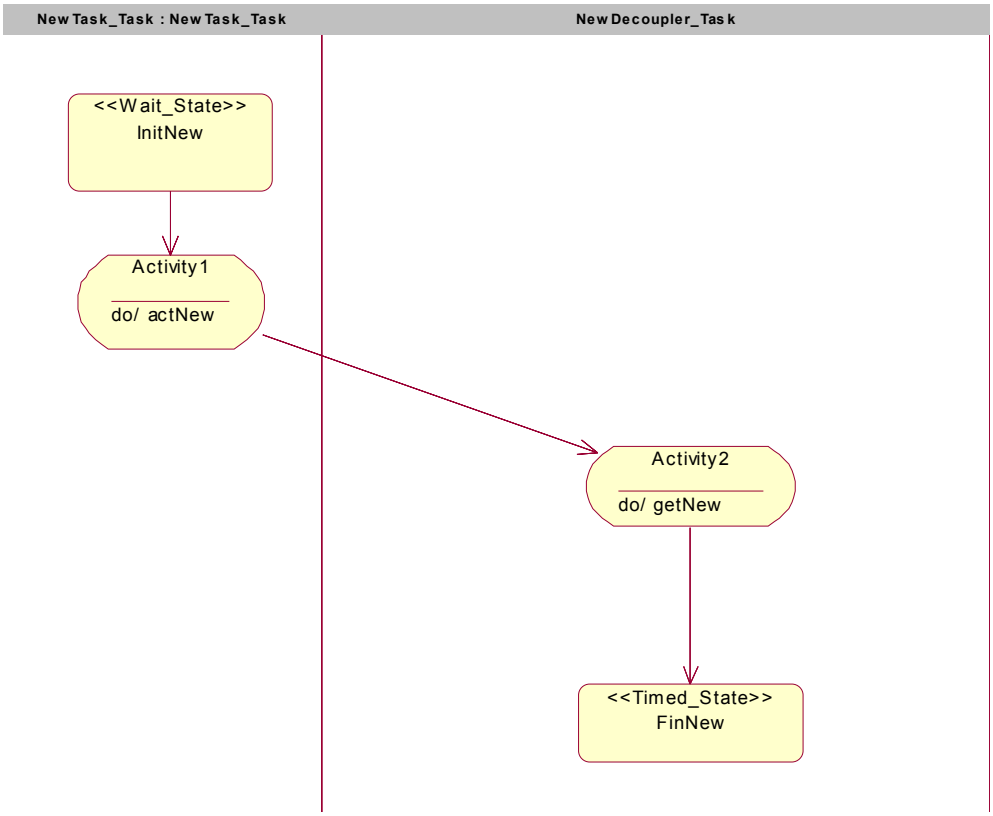


Figura 73

Esta transacción empieza en un estado de espera (*Wait_State*) denominado ***InitNew***. En cuanto se reciba esta señal, dará comienzo del proceso por medio de la operación ***actNew*** que empezará el proceso de actualización. Posteriormente se ejecutará la actividad ***getNew***, la cual como se explicó en el apartado de los componentes lógicos, se encargará de acceder a los datos del nuevo servidor (*NewServer*) para después acceder al recurso compartido y guardarlo en él. Por último, se llega a un *Timed_State* (*FinNew*). Este estado, espera una señal de tipo temporal, que será creada por el *Hard_Global_Deadline*, en este caso cada 100ms, dándole un tiempo límite para se produzca la transacción. En caso que se sobrepase el mencionado tiempo este objeto cortará la ejecución de la transacción con el fin evitar el bloqueo del sistemas.

4.8 Prueba7.

En esta prueba volvemos a representar un modelo que puede considerarse básico, ya que se empezará a representar otro tipo de arquitectura, en este caso distribuida. Se dividirá el control, antes centralizado en un solo módulo, en dos módulos controladores, aunque se va a conservar la funcionalidad del sistema.

4.8.1 Modelo7.

4.8.1.1 Vista Lógica

El diagrama de clases del Modelo7 se puede apreciar en la figura 74. Este modelo está diseñado para poder compararlo con el Modelo1, pudiendo apreciarse en la figura la similitud entre ambos. Además, ambos tienen la misma funcionalidad.

En el diagrama de clases, de la figura 74, se ve como se incluye un nuevo controlador de alto nivel (*HLC2*). Con este otro controlador de alto nivel se distribuye el control del sistema, por lo que ésta se va a considerar una arquitectura distribuida, mientras la utilizada en los módulos anteriores es una arquitectura centralizada.

El resto de componentes comunes a la arquitectura definida para el Modelo1 van a realizar una función similar a la ya explicada en dicho modelo. Por eso solo vamos a proceder a explicar la función realizada por los dos controladores de alto nivel:

- **HLC:** va ser un controlador de alto nivel que se va a encargar de controlar las operaciones realizadas por la interfaz de usuario (*UI*). Además de controlar la tarea *PeriodicActualizarTask,* que seguirá siendo la encargada de realizar una actualización periódica de la interfaz de usuario (*UI*).
- **HLC2:** esta tarea va a ser a partir de ahora de controlar la posibilidad de colisión del Robot, se comunicará con el servidor de cinemática para controlar el estado del *flag* de colisión, por medio de la tarea *PeriodicMoverTask*.

En este nuevo tipo de arquitectura, se puede ver que, va a haber un controlador de alto nivel que se va a encargar de todas las tareas a desarrollar por el sistema (*HLC*) a excepción de la que se encarga de evitar la colisión del Robot, que se podría considerar la más critica.

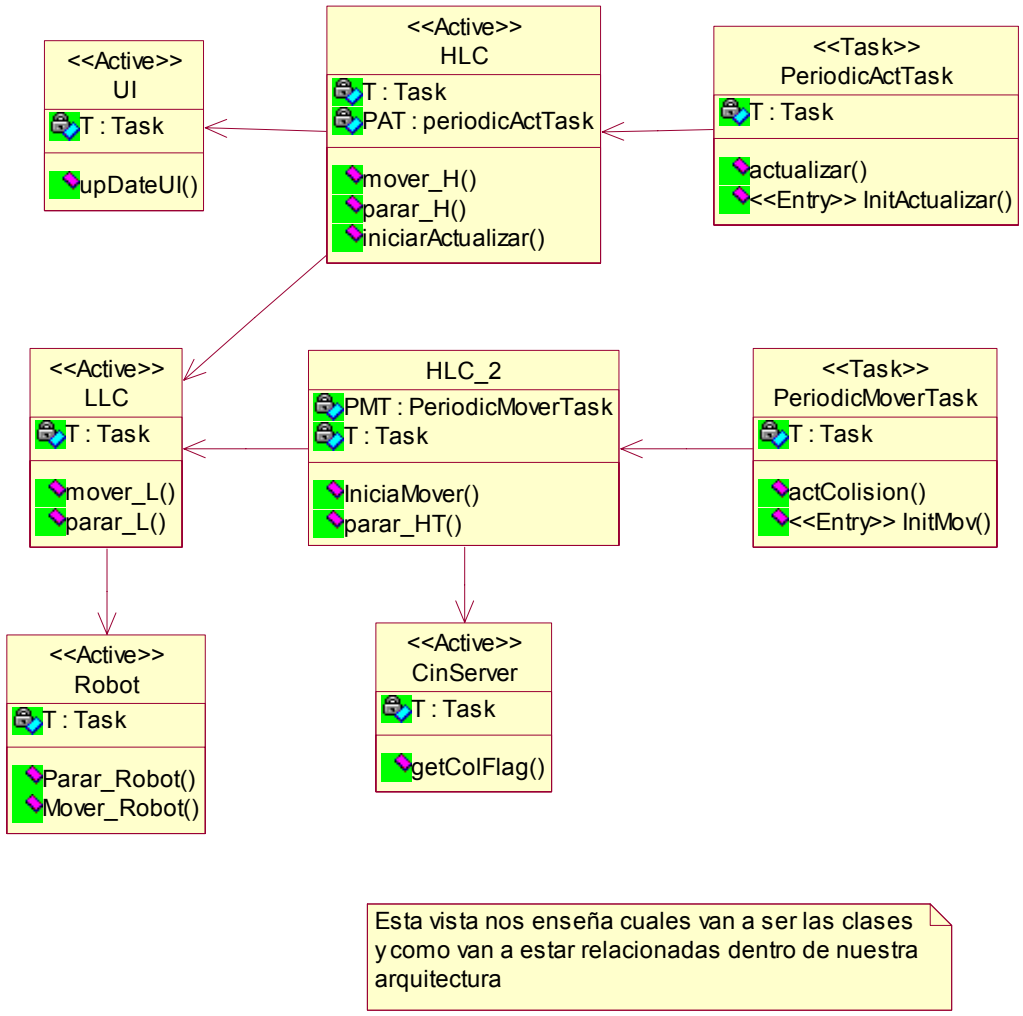


Figura 74

Este modelo también va a tener definidos los procesos que luego van a definir las transacciones que deberán definir el escenario de tiempo real del sistema. Estos procesos, como siempre, los vamos a definir en los diagramas de secuencia que se muestran en las figuras 75 y 76.

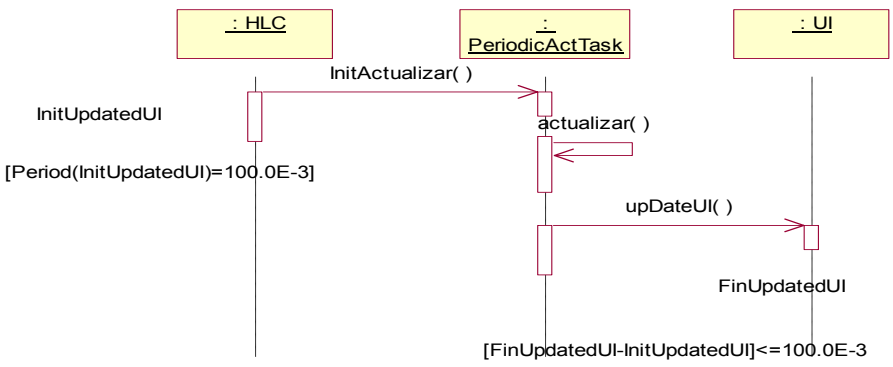


Figura 75 Diagrama de secuencia de UpDateUI_Process

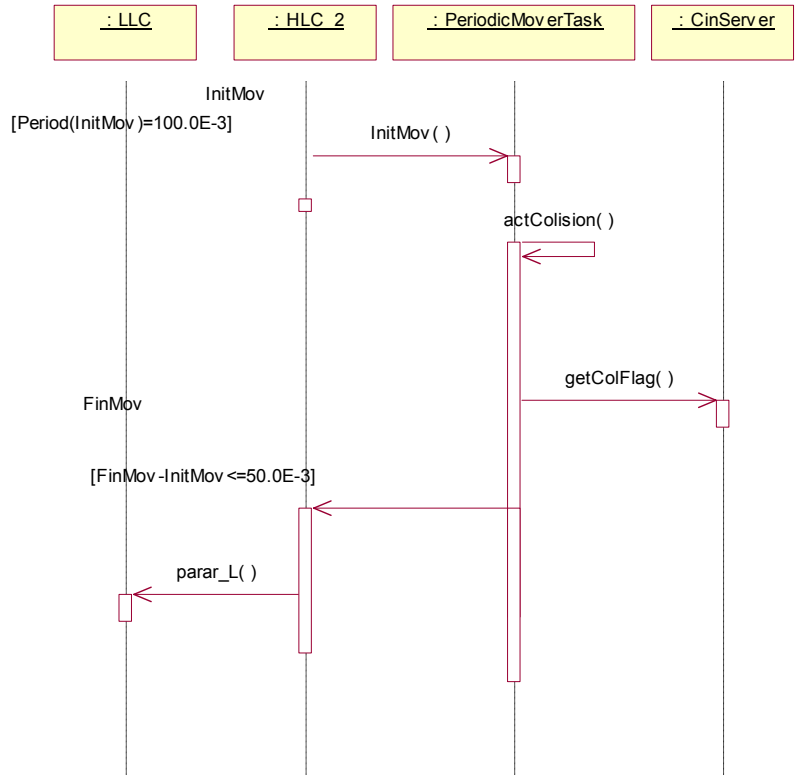


Figura 76 Diagrama de secuencia de Mover_Process

4.8.1.2 Modelo de tiempo real de la plataforma.

El modelo de plataforma va a seguir siendo muy parecido en esta arquitectura. En la figura 77, se puede ver la plataforma definida en este modelo sobre la cual se ejecutará el sistema.

El modelo de procesador definido será el mismo definido en el Modelo1, el cual tiene el nombre de *GeneralProcessor*. También va a ser idéntica la representación de *threads* asociados a este procesador así como los servidores de planificación de las actividades de éstos, pero con la inclusión de un nuevo componente que representará al *thread* que introducirá una instancia a la clase HLC2 (*HLC2_Task*) así como el servidor que planificará sus actividades con una prioridad fija (*HLC2_SP*)

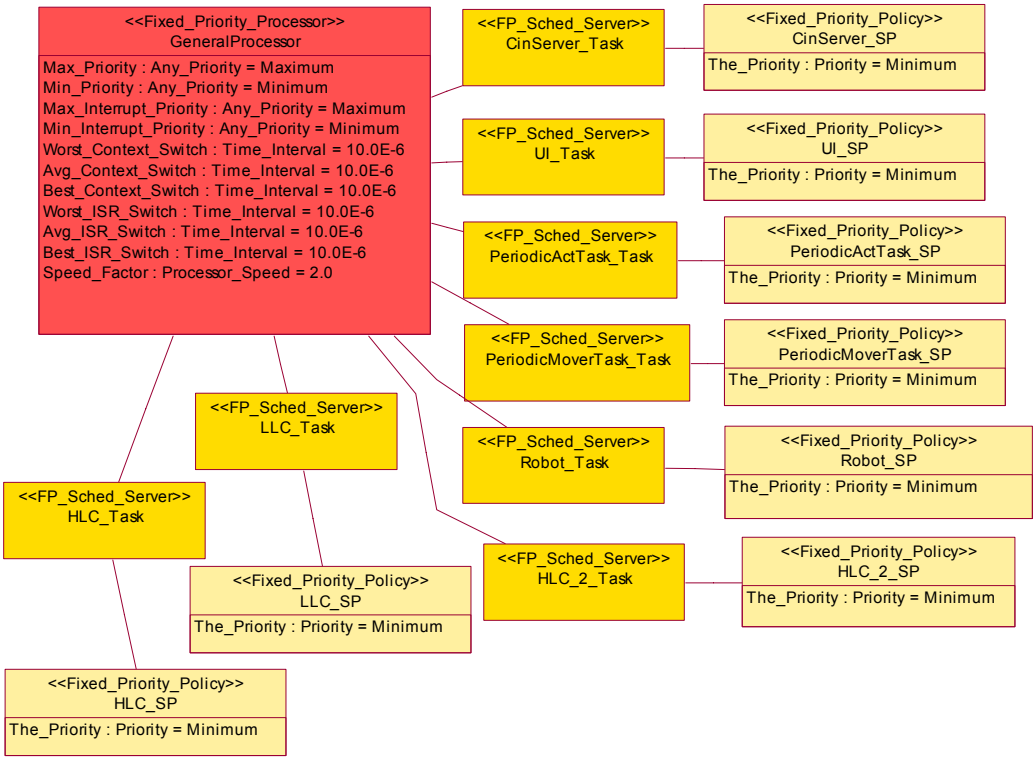


Figura 77

4.8.1.3 Modelo de tiempo real de los componentes lógicos.

Como se ha mencionado en la vista lógica, en este modelo los componentes mantienen la misma funcionalidad que en el Modelo1. A causa de esto, la mayoría de componentes lógicos de este sistema van a mantener la definición realizada en el Modelo1. A continuación, vamos a explicar los modelos lógicos que sufren alguna variación.

HLC_Model.

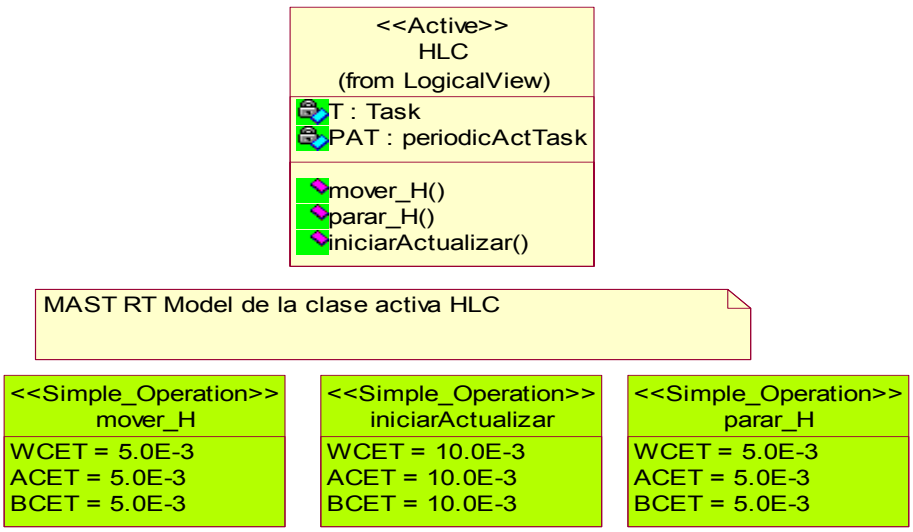


Figura 78

Las operaciones *mover_H* y *parar_H* no sufren variación.

IniciarActualizar es un modelo de operación simple con un tiempo de ejecución de 10ms. Esta operación se va a encargar de iniciar la tarea *PeriodicActTask* así como de controlarla, iniciando así la actualización de la interfaz de usuario.

HLC2_Model.

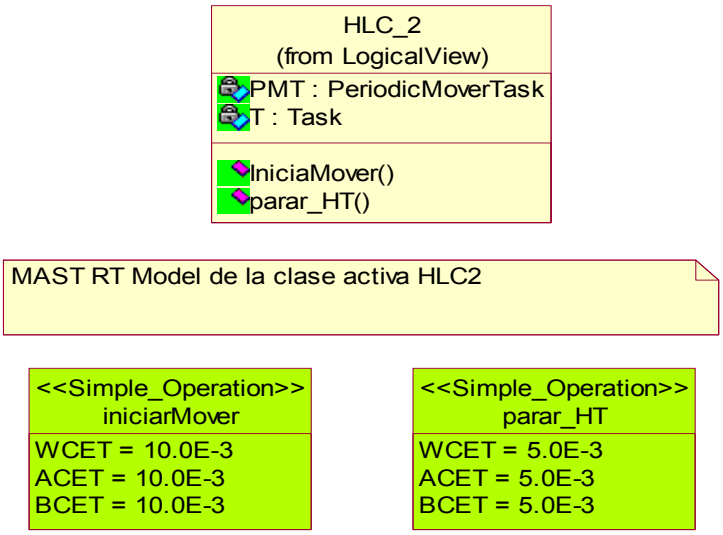


Figura 79

IniciarMover es un modelo de operación simple con un tiempo de ejecución de 10ms. Esta operación se va a encargar de iniciar la tarea *PeriodicMoverTask* así como de controlarla, iniciando así el movimiento.

Parar_HT es un modelo de operación simple con un tiempo de ejecución de 5ms. Esta operación se encargaría de comunicarse con la parte hardware (el robot) y le enviaría el mensaje de detenerlo (esta a la misma altura que parar_H).

4.8.1.4 Escenarios de tiempo real.

En la figura 80 se aprecia el escenario definido en el Modelo7. Este escenario es el mismo definido en el Modelo1. Solamente van a variar los diagramas de secuencia definidos dentro de las transacciones. La secuencia de operaciones definida en las transacciones será también idéntica, pero con la salvedad de que en la transacción *Mover_Transaction* se ejecutará sobre el controlador de alto nivel *HLC2*, como se ha mencionado anteriormente. Así, la carga de operaciones se distribuirá entre los dos controladores de alto nivel definidos en el modelo.

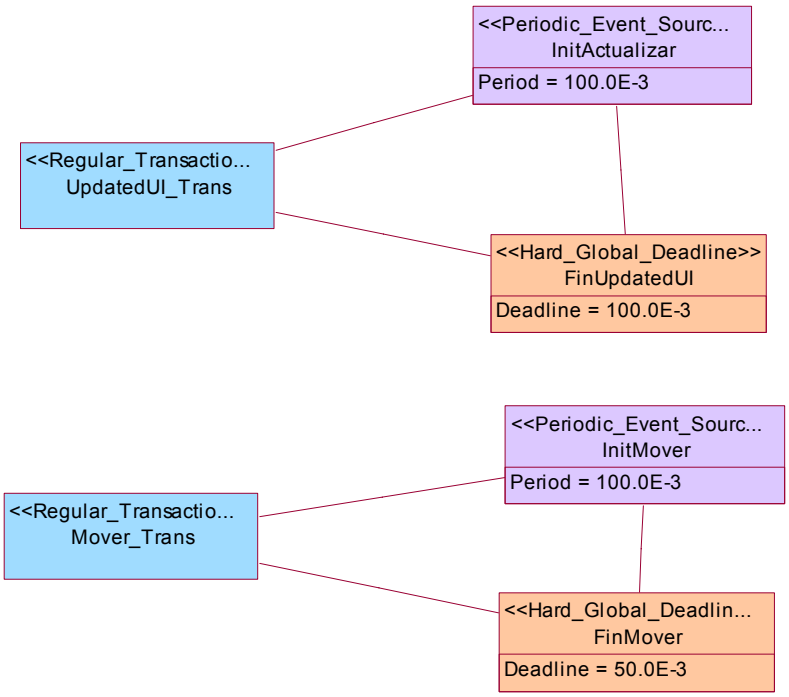


Figura 80

UPDateUI_Transaction.

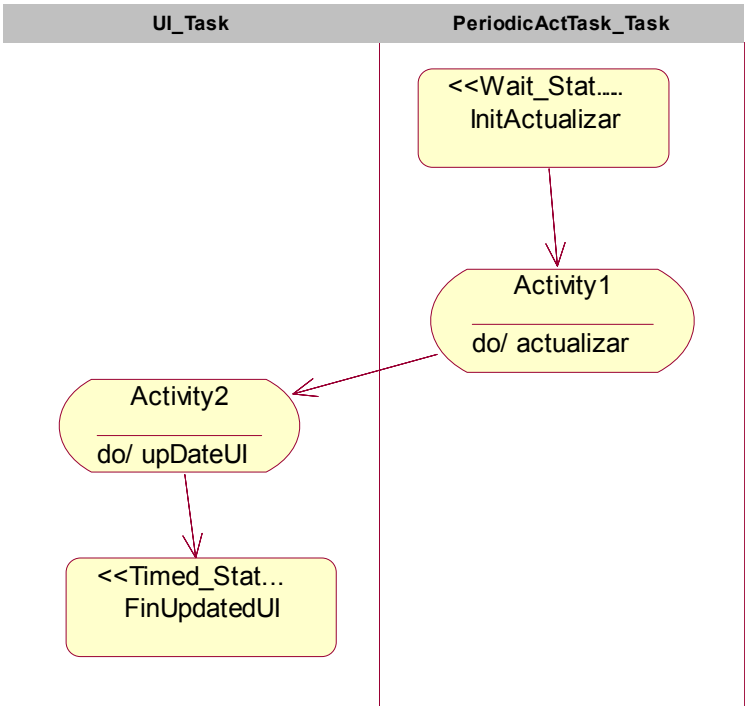


Figura 81

En la figura 81, se observa el diagrama de actividad de esta transacción. Es el mismo definido en todos los modelos, es decir, sigue la misma secuencia de

actividades, pero en este caso es destacable el hecho de que la tarea *PeriodicActualizarTask* está asociada a el controlador de alto nivel *HLC*, lo que significa que este controlador será el encargado de controlarla.

Mover_Transaction.

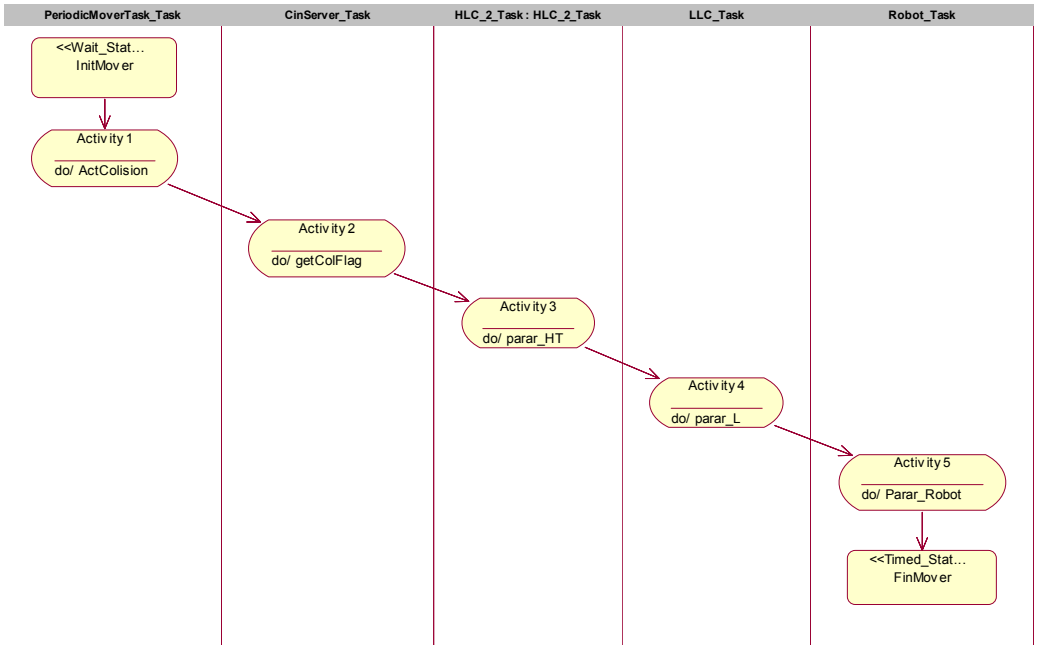


Figura 82

En la figura 82, se observa el diagrama de actividad de esta transacción. Este diagrama va seguir la misma secuencia de actividades que se definía en el Modelo1, pero al igual que pasaba en la transacción anterior, ésta va a ser controlada por el controlador de alto nivel *HLC2* ya que la tarea *PeiodicMoverTask* esta asociada a él, siendo este controlador de alto nivel el encargado de controlarla, así como de controlar la funcionalidad de esta transacción.

De esta manera, vemos que se distribuye el control de las tareas y se descentraliza el control del sistema. Entre los dos controladores de alto nivel definidos en este diseño.

4.9 Prueba8.

4.9.1 Modelo8.

4.9.1.1 Vista lógica.

En la figura 83, se observa el diagrama de clases del Modelo8. Este modelo será equivalente al Modelo5. Para su explicación cabe reseñar que partiremos del Modelo7, ya que en este modelo se incrementa la dificultad del sistema anterior aumentando la complejidad del escenario, así como el número de módulos que componen el sistema.

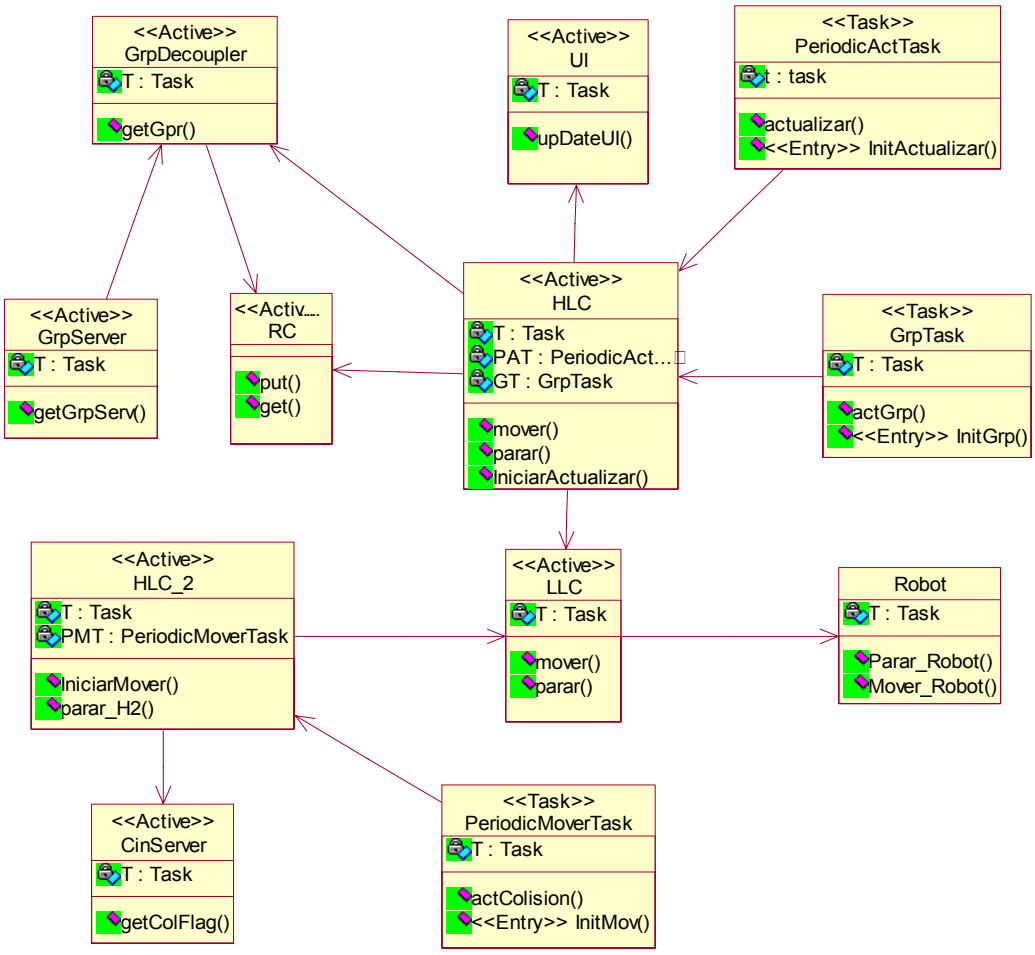


Figura 83

En la figura se ve que han sido añadidos el mismo bloque que se añadía en el Modelo5, donde ya se he explicado estos componentes y la función que realizan.

Los diagramas de secuencia que definen los procesos que llevará a cabo este sistema ya han sido definidos en este trabajo. Se han definido estos tres diagramas de secuencia: *UpDateUI_Process* (figura 75), *Mover_Process* (Figura 76) y *Grp_Process* (figura 40).

4.9.1.2 Modelo de tiempo real de la plataforma.

El modelo de plataforma va a ser el mismo que el descrito en el Modelo7, pero con la inclusión de nuevos componentes. Estos están definidos en el Modelo5, en la figura 41.

4.9.1.3 Modelo de tiempo real de los componentes lógicos.

Los componentes lógicos de este modelo ya han sido explicados a lo largo del capítulo, por lo tanto solo se van a nombrar y decir dónde fueron explicados. Estos son los componentes lógicos:

- **UI_Model:** Explicado en el Modelo1, figura 5.
- **CinServer_Model:** Modelo1, figura 6.
- **Robot_Model:** Modelo1, figura 7.
- **LLC_Model:** Modelo1, figura 8.
- **PeriodicActTask_Model:** Modelo1, figura 10.
- **PeriodicMoverTask_Model:** Modelo1, Figura 11.
- **RC_Model:** Modelo3, figura 25.
- **GrpServer_Model:** Modelo5, figura 43.
- **GrpDecoupler:** Modelo5, figura 44.
- **GrpTask_Model:** Modelo5, figura42.
- **HLC_Model:** Modelo7, figura 78.
- **HLC2_Model:** Modelo7, figura 79.

4.9.1.4 Escenario de tiempo real.

En la figura 84, se aprecia el escenario definido para el Modelo8. En este escenario se distinguen tres transacciones las cuales ya han sido definidas anteriormente en este documento. Estas transacciones son:

- **UpDateUI_Transaction:** definida en el Modelo1, en el apartado de escenario de tiempo real.
- **Mover_Transaction:** definida en el Modelo7, en el apartado de escenario de tiempo real.
- **Grp_Transaction:** definida en Modelo5v2, en el apartado de escenarios de tiempo real.

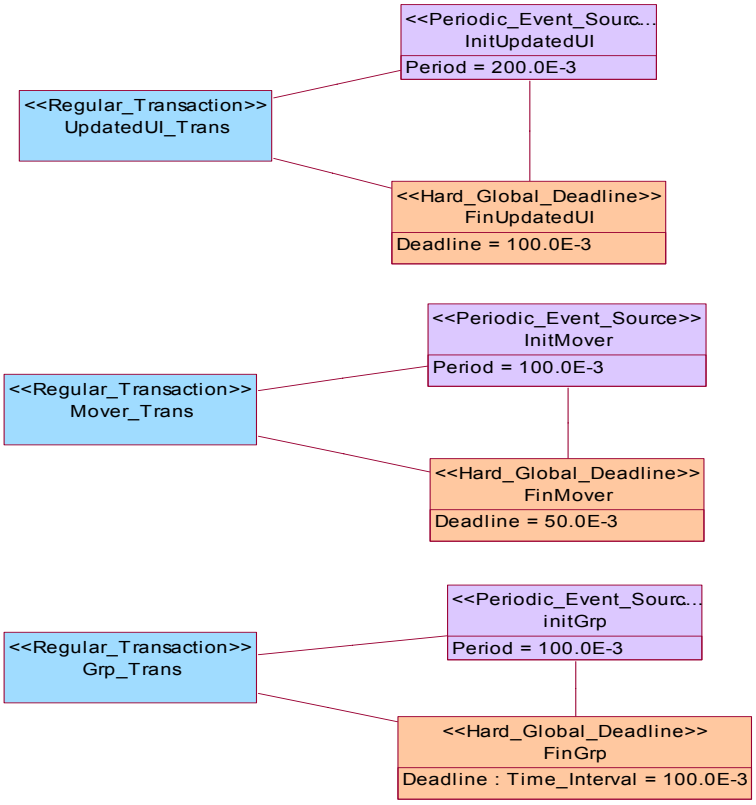


Figura 84

4.9.2 Prueba8v2.

4.9.2.1 Vista Lógica.

En la figura 84, se observa el diagrama de clases del Modelo8v2. Este modelo será equivalente al Modelo5v2. Cabe reseñar que partiremos del Modelo7, ya que en este modelo se incrementa la dificultad del sistema anterior aumentando la complejidad del escenario, así como el número de módulos que componen el sistema.

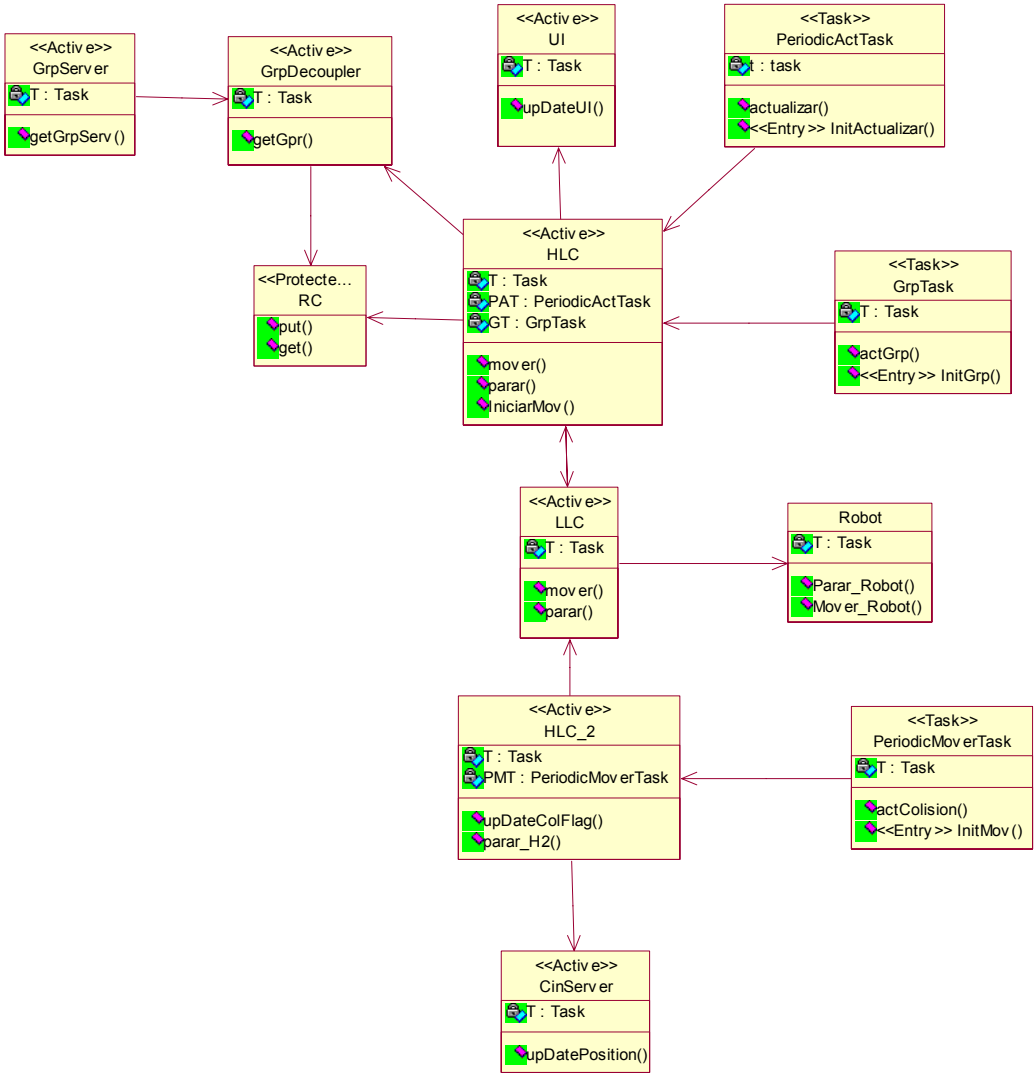


Figura 85

Los diagramas de secuencia serán los mismos que definimos en el modelo8, debido a que en este proceso se realizarán los mismos procesos, y solo cambiará el modo de acceso al recurso compartido.

4.9.2.2 Modelo de tiempo real de la plataforma.

Es el mismo modelo de plataforma definido en el Modelo6, con la salvedad de que en este caso no se definirá los componentes que definen el *thread* que representa una instancia a la clase RC, ni su servidor de la política de planificación.

4.9.2.3 Modelo de tiempo real de los componentes lógicos.

La gran mayoría de los componentes lógicos utilizados en este modelo han sido ya explicados en algún apartado del trabajo, por eso vamos a indicar donde ha sido explicado cada uno, y por último explicar los nuevos componentes introducidos.

- **UI_Model:** Explicado en el Modelo1, figura 5.

- **CinServer_Model:** Modelo1, figura 6.
- **Robot_Model:** Modelo1, figura 7.
- **LLC_Model:** Modelo1, figura 8.
- **PeriodicActTask_Model:** Modelo1, figura 10.
- **PeriodicMoverTask_Model:** Modelo1, Figura 11.
- **RC_Model:** Modelo5v2, figura 54.
- **GrpServer_Model:** Modelo5, figura 43.
- **GrpDecoupler_Model:** Modelo5, figura 44.
- **GrpTask_Model:** Modelo5, figura 42.
- **HLC_Model:** Modelo7, figura 78.
- **HLC2_Model:** Modelo7, figura 79.

4.9.2.4 Escenarios de tiempo real.

En la figura 86, se observa el escenario definido para este sistema. Esta compuesto por tres transacciones, que ya han sido explicadas. Estas transacciones son:

- **UpDateUI_Transaction:** definida en el Modelo1, en el apartado de escenario de tiempo real.
- **Mover_Transaction:** definida en el Modelo7, en el apartado de escenario de tiempo real.
- **Grp_Transaction:** definida en Modelo5v2, en el apartado de escenarios de tiempo real.

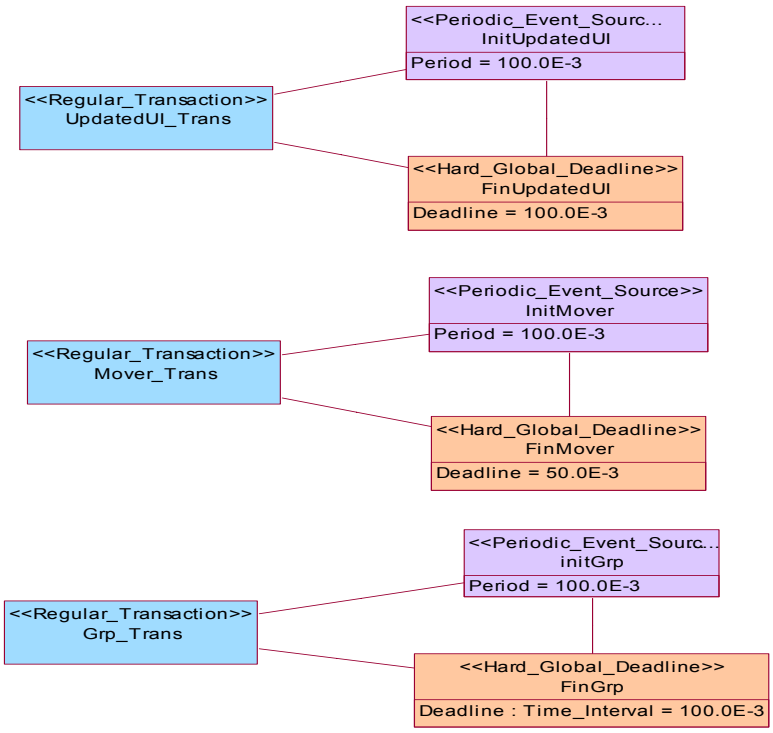


Figura 86

4.10 Prueba9.

4.10.1 Modelo9.

En este modelo se realiza una ampliación en cuanto a componentes y se aumenta de nueva la complejidad del escenario de tiempo real. Se añaden los mismos componentes que en el Modelo6.

4.10.1.1 Vista Lógica.

En la figura 87, se observa el diagrama de clases de este modelo. Se observa que se trata de una versión del Modelo6, pero con distribución de las tareas entre dos controladores de alto nivel.

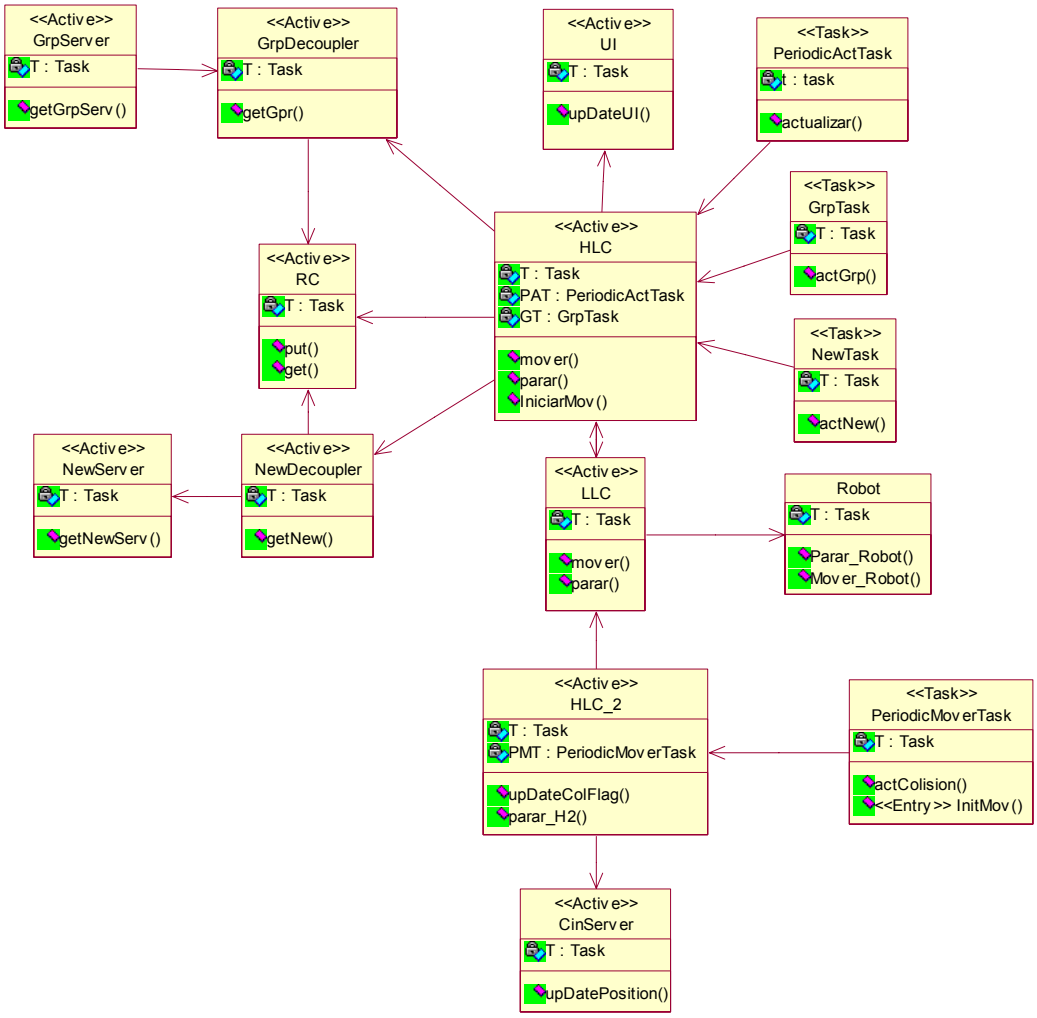


Figura 87

Los procesos que va a realizar este sistema también van a ser definidos por los diagramas de secuencia. Estos procesos ya han sido explicados en el documento y son cuatro: UpDateUI_Process (Figura 75 Modelo7), Mover_Process (Figura 76 Modelo7), Grp_Process (Figura 40, Modelo5) y New_Process (Figura 61, Modelo6).

4.10.1.2 Modelo de tiempo real de la plataforma.

En esta vista se amplia la vista del Modelo8 con los mismos componentes definidos en el Modelo6. Estos componentes se pueden ver en el apartado correspondiente al Modelo6.

4.10.1.3 Modelo de los componentes lógicos.

Todos los componentes lógicos definidos en esta vista del Modelo9 han sido ya explicados en el documento. Estos componentes lógicos están explicados en:

- **UI_Model:** Explicado en el Modelo1.
- **CinServer_Model:** Modelo1.
- **Robot_Model:** Modelo1.
- **LLC_Model:** Modelo1.
- **PeriodicActTask_Model:** Modelo1.
- **PeriodicMoverTask_Model:** Modelo1.
- **RC_Model:** Modelo3.
- **GrpServer_Model:** Modelo5.
- **GrpDecoupler:** Modelo5.
- **GrpTask_Model:** Modelo5.
- **NewServer:** Modelo6.
- **NewTask:** Modelo6.
- **NewDecoupler:** Modelo6.
- **HLC_Model:** Modelo7.
- **HLC2_Model:** Modelo7.

4.10.1.4 Escenario de tiempo real.

En la figura 88, se observa el escenario definido para este sistema. Esta compuesto por cuatro transacciones, que ya han sido explicadas. Estas transacciones son:

- **UpDateUI_Transaction:** definida en el Modelo1, en el apartado de escenario de tiempo real.
- **Mover_Transaction:** definida en el Modelo7, en el apartado de escenario de tiempo real.
- **Grp_Transaction:** definida en Modelo5, en el apartado de escenarios de tiempo real.
- **New_Transaction:** definida en Modelo6, en el apartado de escenarios de tiempo real.

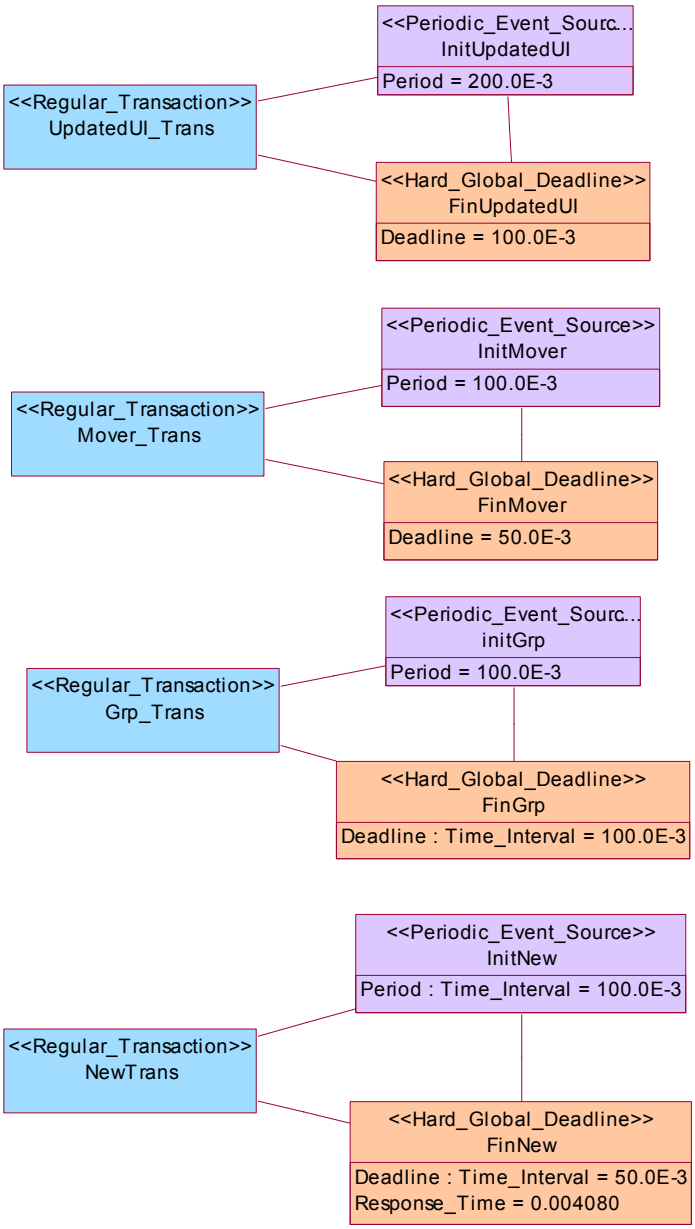


Figura 88

4.10.2 Modelo9v2.

En esta segunda versión del Modelo6, se cambia el objeto activo RC por un objeto protegido, como ha ocurrido en el Modelo5v2.

4.10.2.1 Vista Lógica.

En la figura 89, se ve el diagrama de clases de este modelo y vemos que en el esta definido el recurso (*RC*) compartido como *Protected*, es decir, como un objeto protegido.

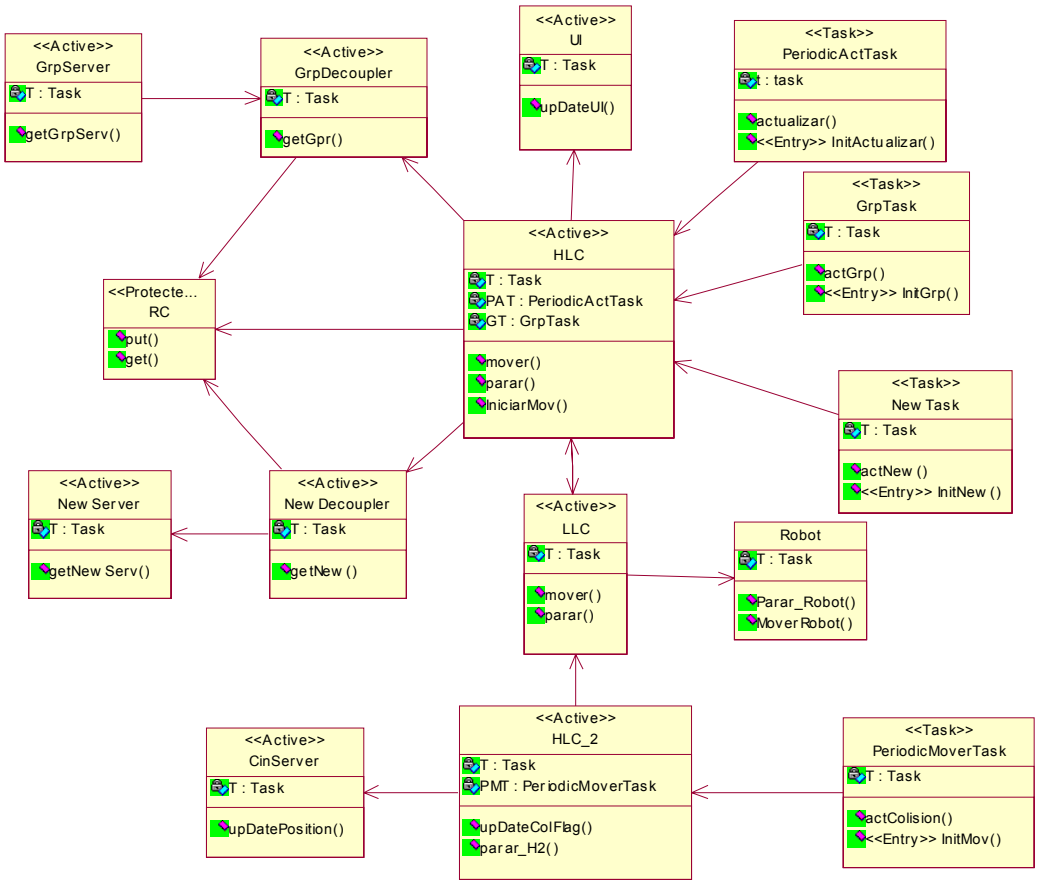


Figura 89

Los diagrama de secuencia serán los mimos del Modelo6, por eso no es necesario volver a mostrarlos aquí.

4.10.2.2 Modelo de tiempo real de la plataforma.

Es el mismo modelo de plataforma definido en el Modelo6, con la salvedad de que en este caso no se definirá los componentes que definen el *thread* que representa una instancia a la clase RC, ni su servidor de la política de planificación.

4.10.2.3 Modelo de tiempo real de los componentes lógicos.

Todos los componentes lógicos de este modelo ya han sido definidos en este trabajo por eso se va a indicar donde fueron explicados.

- **UI_Model:** Explicado en el Modelo1.
- **CinServer_Model:** Modelo1.
- **Robot_Model:** Modelo1.
- **LLC_Model:** Modelo1.
- **PeriodicActTask_Model:** Modelo1.
- **PeriodicMoverTask_Model:** Modelo1.
- **RC_Model:** Modelo6v2.

- **GrpServer_Model:** Modelo5.
- **GrpDecoupler:** Modelo5v2.
- **GrpTask_Model:** Modelo5.
- **NewServer:** Modelo6.
- **NewTask:** Modelo6.
- **NewDecoupler:** Modelo6v2.
- **HLC_Model:** Modelo7.
- **HLC2_Model:** Modelo7.

4.10.2.4 Escenarios de tiempo real.

En la figura 88, se observa el escenario definido para este sistema. Esta compuesto por cuatro transacciones, que ya han sido explicadas. Estas transacciones son:

- **UpDateUI_Transaction:** definida en el Modelo1, en el apartado de escenario de tiempo real.
- **Mover_Transaction:** definida en el Modelo7, en el apartado de escenario de tiempo real.
- **Grp_Transaction:** definida en Modelo5v2, en el apartado de escenarios de tiempo real.
- **New_Transaction:** definida en Modelo6v2, en el apartado de escenarios de tiempo real.

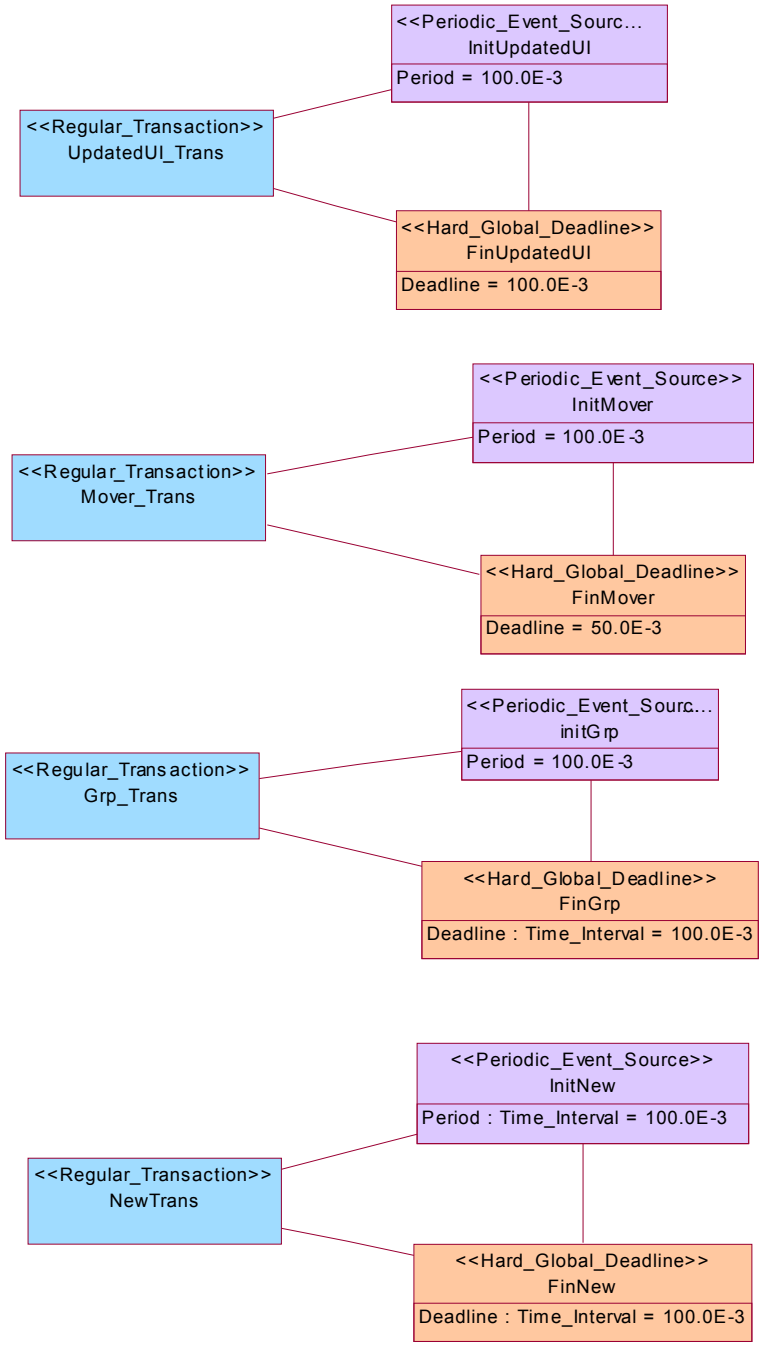

<<Periodic_Event_Sourc...
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
<<Periodic_Event_Source>>
InitMover
Period = 100.0E-3
<<Regular_Transaction>>
Mover_Trans
<<Hard_Global_Deadline>>
FinMover
Deadline = 50.0E-3
<<Periodic_Event_Sourc...
initGrp
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinGrp
Deadline : Time_Interval = 100.0E-3
<<Periodic_Event_Source>>
InitNew
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
NewTrans
<<Hard_Global_Deadline>>
FinNew
Deadline : Time_Interval = 100.0E-3

Figura 90

Capítulo 5
Análisis MAST

5.1 Introducción.

Una vez presentados todos los modelos, en este capitulo se procede a mostrar los resultados obtenidos, así como una interpretación de los mismos. Estos resultados se van a obtener de realizar el análisis de planificación con MAST, y van a estar tabulados para una mejor apreciación y comparación de los mismos.

MAST nos va a ofrecer un archivo donde se representa la simulación del sistema, a partir de la cual se calcularán los tiempos de respuesta de las transacciones. Estos tiempos van a ser los datos que utilizaremos para someter a estudio los modelos. Este archivo incluye otros resultados de interés como la holgura (*slacks*) de las transacciones. Estas holguras ya que indican el tanto por ciento que se deben incrementar uniformemente las operaciones que intervienen en la planificación para que deje de ser planificable.

Este trabajo se va a concentrar en los tiempos de respuesta (Response Times), ya que estos nos permiten comparar los tipos de arquitectura de software que representan los modelos expuestos anteriormente.

La información proporcionada por las holguras, será interesante a la hora de implementar físicamente las arquitecturas, ya que una vez que se conozca de manera más exacta el valor del tiempo de ejecución de las operaciones las holguras indicarán si es posible llevar a cabo o no la implementación de los modelos de manera real.

Como se ha indicado, los resultados explicados en este Capítulo van a ser sacados del archivo creado por MAST al analizar cada modelo. Este archivo llevará la extensión *.out* y el nombre del modelo, ***ModeloX.out***.

5.2 Prueba1.

5.2.1 Análisis de planificabilidad del Modelo1.

El Modelo1 era el modelo centralizado más simple. En este modelo existe un controlador de alto nivel (*HLC*) que se encargara de controlar todos los procesos que ocurren en el sistema.

Una vez diseñado el modelo en MAST, se realiza el análisis sobre él. El análisis se puede descomponer en varias partes. Los datos sobre los que se va a trabajar en este apartado van a ser segundos.

En el caso que resultase ser un sistema no planificable, MAST lo indica durante el análisis y termina el mismo. En este caso esto no ha ocurrido por lo cual se trata de un sistema planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST. Estos requisitos serán cumplidos cuando el tiempo de respuesta dado por MAST sea menor que el plazo de respuesta marcado por los eventos con requisito temporal.

Empezaremos por explicar la transacción *Mover_Transaction*, en la cual se comprueba el estado del flag de colisión, y en caso de estar activado manda parar al sistema.

Transacción => **Mover_Trnas**

Evento con requerimiento temporal =>**EndMover**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitMover**

Deadline =>**0.050**

Resultado del analisis:

Worst_Global_Response_Times =>**0.014300**

Best_Global_Response_Times => **0.014300**

Jitters =>**0.00**

El evento con requisito temporal indica el nombre un estado temporal (Time state), que espera la llegada de una señal para la activación del evento de tipo temporal.

El tipo de requisito temporal, en este caso es un plazo global estricto (Global_Hard_Deadline), el cual requiere que el evento al que esta asociado, se produzca antes de que se cumpla el plazo establecido (Deadline) desde que se produjo el evento de referencia.

El evento de referencia indica el nombre del estado de espera (Wait state), desde el cual se inicia el proceso de la transacción.

El deadline, es el plazo establecido para el requisito temporal, el tiempo de respuesta dado por MAST, debe ser inferior a este plazo, para que se cumplan los requisito temporales establecidos.

Los resultados del análisis indican el mejor tiempo de respuesta (*Best_Global_Response_Times*) calculado por MAST, el peor tiempo de respuesta (Worst_*Global_Response_Times*) y la variación respecto al retardo del sistema (*Jitter*).

Al comparar el plazo establecido por los requisito del sistema, y los tiempos de respuesta resultantes del análisis de MAST, vemos que se cumplen los requisito temporales al ver que los tiempos de respuesta son menores que el plazo especificado en el sistema.

En la siguiente transacción, el controlador de alto nivel se encarga de actualizar periódicamente a la interfaz de usuario.

Transacción => **UpDateUI_Trnas**

Evento con requerimiento temporal =>**EndUpDateUI**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitActualizar**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.008040**

Best_Global_Response_Times => **0.008040**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

El cálculo de holguras (*slack*) se mide como el tanto por ciento que deben incrementar uniformemente las operaciones implicadas en la transacción para que no sea planificable.

- Transacción =>Mover_Trans

slack => 250.78 %

- Transacción =>UpDateUI_Trans

slack => 964.06 %

También calcula la holgura total del sistema => 250.78%

Siguiendo los pasos indicados en el Capitulo3, insertaremos los resultados más relevantes del análisis dentro de la vista de los escenarios de tiempo real. En la figura 1, se ve cómo se han incluido estos resultados.

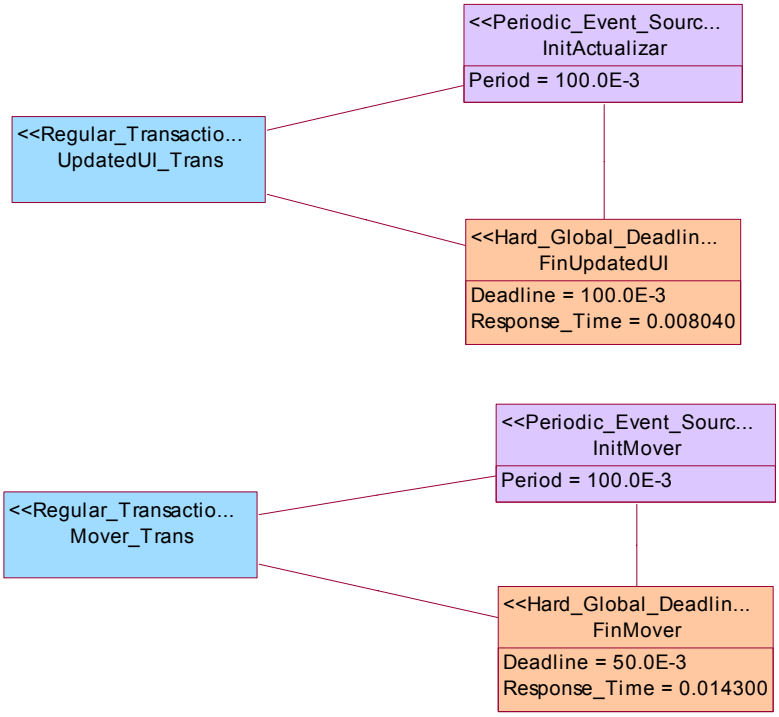


Figura 1

5.3 Prueba2.

5.3.1 Análisis de planificabilidad del Modelo2.

En este modelo será el controlador de bajo nivel (*LLC*) el que actualice los datos de posición en el servidor de cinemática (*CinServer*), en lugar del controlador de alto nivel (*HLC*). El resto de características del Modelo1, se mantienen.

Una vez terminado el diseño del modelo, el análisis del sistema se realiza correctamente, por lo cual se trata de un sistema planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de controlar el estado del Flag de colisión, y en caso de estar activado detendrá el sistemas.

Transacción **=> Mover_Trnas**

Evento con requerimiento temporal **=>EndMover**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitMover**

Deadline **=>0.050**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.016320**

Best_Global_Response_Times **=> 0.016320**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar la interfaz de usuario de manera periódica.

Transacción **=> UpDateUI_Trnas**

Evento con requerimiento temporal **=>EndUpDateUI**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitActualizar**

Deadline **=>0.100**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.008040**

Best_Global_Response_Times **=> 0.008040**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>Mover_Trans

slack => 207.81 %

- Transacción =>UpDateUI_Trans

slack => 939.06 %

También calcula la holgura total del sistema => 207.81%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 2, se ve como se han incluido estos resultados.

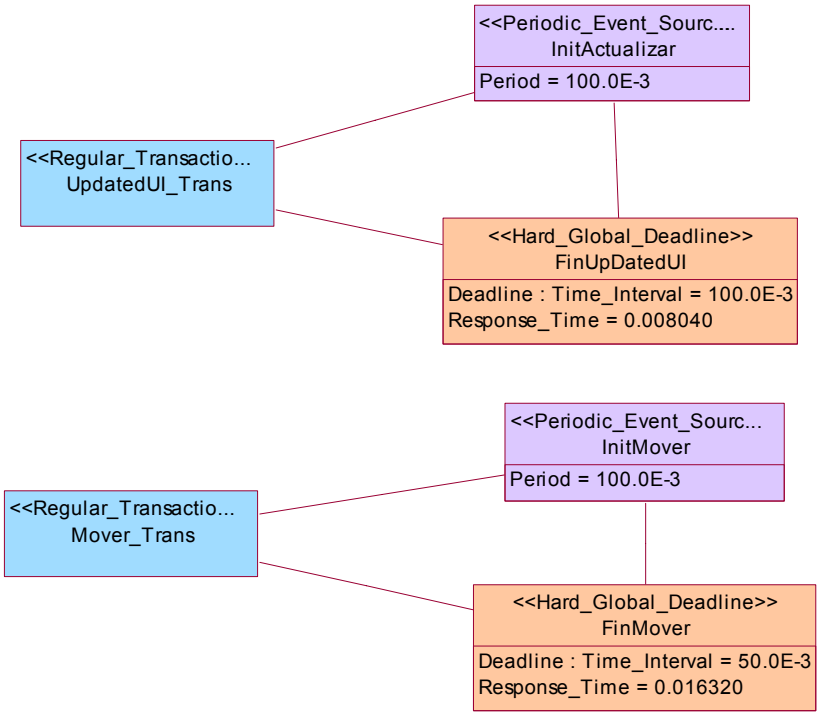


Figura 2

5.4 Prueba3.

5.4.1 Análisis de planificabilidad del Modelo3.

Este modelo añade un nuevo elemento al análisis, un recurso compartido. Éste será un objeto activo, que se encargará de controlar el acceso a una variable, en la cual se guardará el valor del *flag* de colisión.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando un sistema planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción	**=> UpDateUI_Trnas**
Evento con requerimiento temporal	**=>EndUpDateUI**
Tipo de requerimiento temporal	**=>Global_Hard_Deadline**
Evento de referencia	**=>InitActualizar**
Deadline	**=>0.100**

Resultado del analisis:

Worst_Global_Response_Times	**=> 0.008040**

Best_Global_Response_Times => **0.008040**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de controlar que el servidor de cinemática, actualice continuamente el valor del *flag* de colisión, en la variable *colision,* perteneciente al recurso compartido.

Transacción => **ActCol_Trnas**

Evento con requerimiento temporal =>**FinActCol**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitActCol**

Deadline =>**0.050**

Resultado del análisis:

Worst_Global_Response_Times => **0.013060**

Best_Global_Response_Times => **0.013060**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción accederá continuamente al valor de la variable *colision*, del recurso compartido. En caso de que esté activada, detendrá el sistema.

Transacción => **PickUp_Trnas**

Evento con requerimiento temporal =>**Fin PickUp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitPickUp**

Deadline =>**0.050**

Resultado del análisis:

Worst_Global_Response_Times => **0.005300**

Best_Global_Response_Times => **0.005300**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>UpDateUI_Trans

  slack => 913.28 %

- Transacción =>ActCol_Trans

  slack => 283.59 %

- Transacción =>PickUp_Trans

  slack => 859.38 %

También calcula la holgura total del sistema => 278.91%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 3, se ve como se han incluido estos resultados.

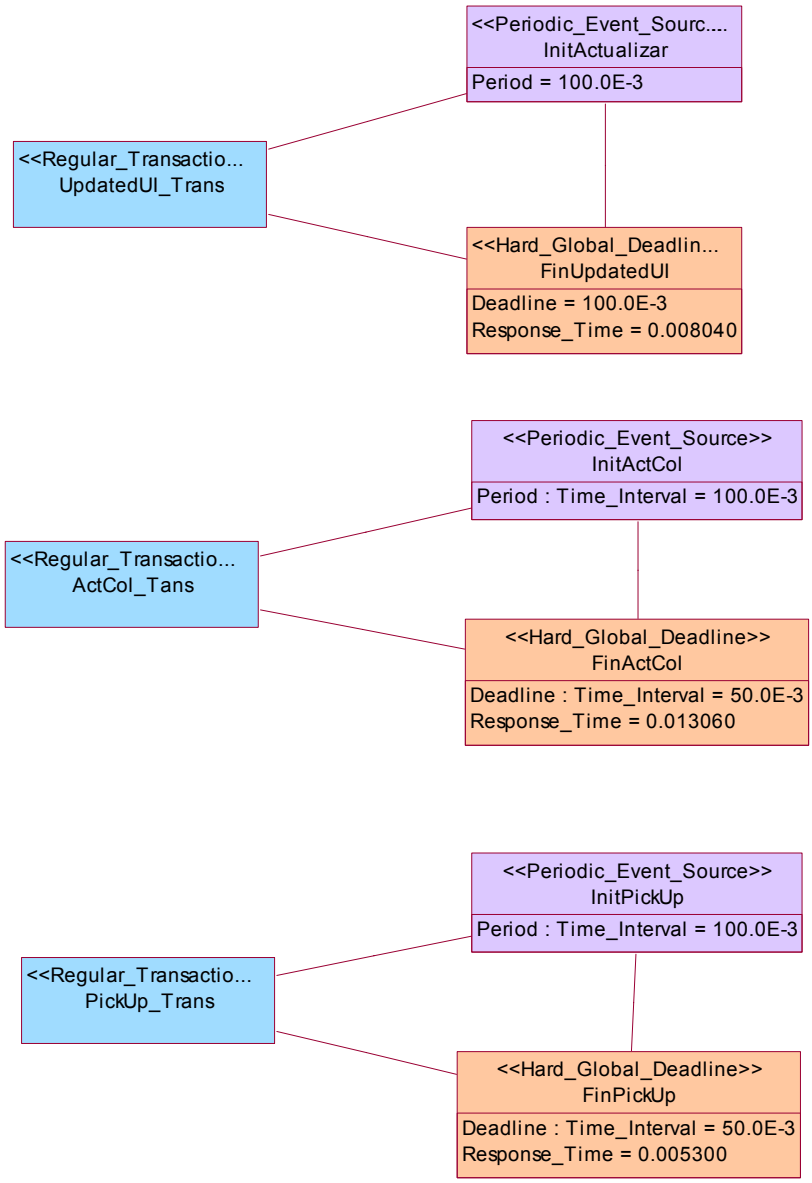


Figura 3

5.5 Prueba4.

5.5.1 Análisis de planificabilidad del Modelo4.

En este modelo, vuelve a haber el incremento de la complejidad del sistema debido a la inclusión de un nuevo componente, el módulo de desacoplo (*CinDecoupler*), el cual se encargará de comunicar al sistema con el servidor de cinemática (*CinServer*). El resto del sistema se mantiene igual.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando un sistema planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.008040

Best_Global_Response_Times => 0.008040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de controlar que el servidor de cinemática, actualice continuamente el valor del *flag* de colisión, en la variable *colision,* perteneciente al recurso compartido.

Transacción => ActCol_Trnas

Evento con requerimiento temporal =>FinActCol

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActCol

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.014080

Best_Global_Response_Times => 0.014080

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción accederá continuamente al valor de la variable *colision*, del recurso compartido. En caso de que esté activada, detendrá el sistema.

Transacción => PickUp_Trnas

Evento con requerimiento temporal =>Fin PickUp

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitPickUp

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => **0.005080**

Best_Global_Response_Times => **0.005080**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este sistema, son:

- Transacción =>UpDateUI_Trans

  slack => 903.13 %

- Transacción =>ActCol_Trans

  slack => 256.25 %

- Transacción =>PickUp_Trans

  slack => 897.66 %

También calcula la holgura total del sistema => 256.25%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 4, se ve como se han incluido estos resultados.

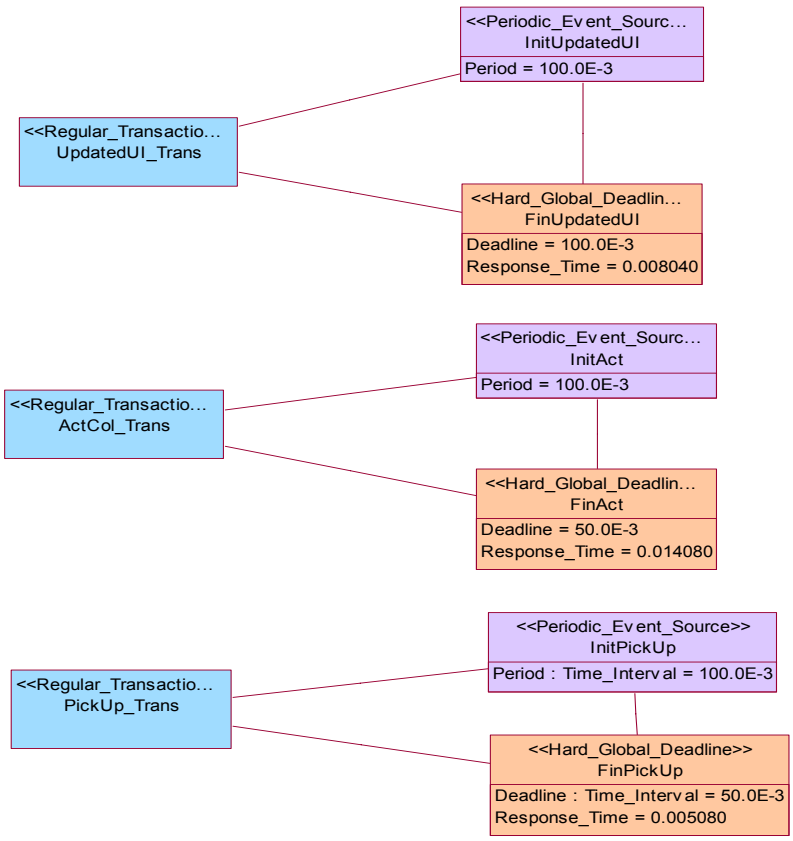


Figura 4

5.6 Prueba5.

5.6.1 Análisis de planificabilidad del Modelo5.

En este modelo, vuelve a haber el incremento de la complejidad del sistema, debido a la inclusión de nuevos componentes. Se añade un nuevo servido, con un desacoplador que media con el sistema. Este nuevo modulo añadido se encargará de controlar una nueva transacción que se incluirá en el escenario del sistema.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, que resulta ser un sistema planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.008040

Best_Global_Response_Times => 0.008040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de controlar que el servidor de cinemática, actualice continuamente el valor del *flag* de colisión, en la variable *colision,* perteneciente al recurso compartido.

Transacción => ActCol_Trnas

Evento con requerimiento temporal =>FinActCol

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActCol

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.014080

Best_Global_Response_Times => 0.014080

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción accederá continuamente al valor de la variable *colision*, del recurso compartido. En caso de que esté activada, detendrá el sistema.

Transacción => **PickUp_Trnas**

Evento con requerimiento temporal =>**FinPickUp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitPickUp**

Deadline =>**0.050**

Resultado del analisis:

Worst_Global_Response_Times => **0.005080**

Best_Global_Response_Times => **0.005080**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción => **Grp_Trnas**

Evento con requerimiento temporal =>**FinGrp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitGrp**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.014080**

Best_Global_Response_Times => **0.014080**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo son:

- Transacción =>UpDateUI_Trans

slack => 727.34 %

- Transacción =>ActCol_Trans

slack => 256.25 %

- Transacción =>PickUp_Trans

slack => 897.66 %

Transacción =>Grp_Trans

slack => 387.50 %

También calcula la holgura total del sistema => 141.41%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 5, se ve como se han incluido estos resultados.

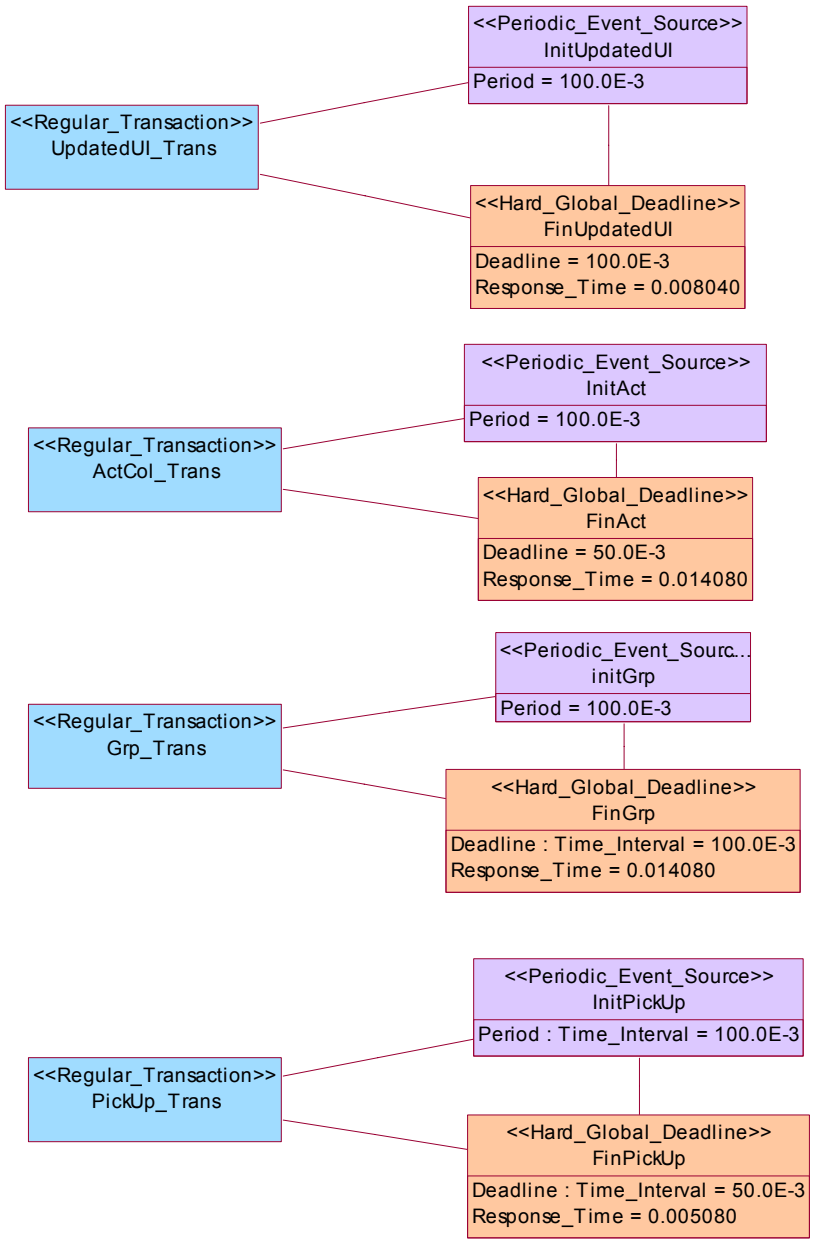


Figura 5

5.6.2 Análisis de planificabilidad del Modelo5v2.

En este modelo, se cambia el comportamiento, definido en la primera versión, del recurso compartido. Ahora se trata de un objeto protegido, que actua de manera pasiva.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando un sistema planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisito han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.008790

Best_Global_Response_Times => 0.008790

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de controlar que el servidor de cinemática, actualice continuamente el valor del *flag* de colisión, en la variable *colision,* perteneciente al recurso compartido.

Transacción => Mover_Trnas

Evento con requerimiento temporal =>FinMover

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitMover

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.015310

Best_Global_Response_Times => 0.015310

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción => Grp_Trnas

Evento con requerimiento temporal =>FinGrp

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitGrp

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.003790

Best_Global_Response_Times => 0.015040

Jitters => -0.011250

Esta transacción accederá continuamente al valor de la variable *colision*, del recurso compartido. En caso de que esté activada, detendrá el sistema.

Transacción => **PickUp_Trnas**

Evento con requerimiento temporal =>**FinPickUp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitPickUp**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.010470**

Best_Global_Response_Times => **0.003790**

Jitters => **0.006680**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>UpDateUI_Trans

slack => 144.53 %

- Transacción =>Mover_Trans

slack => 87.50

-Transacción =>Grp_Trans

slack => 386.72 %

-Transacción =>PickUp_Trans

slack => 192.97 %

También calcula la holgura total del sistema => 40.63%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 6, se ve como se han incluido estos resultados.

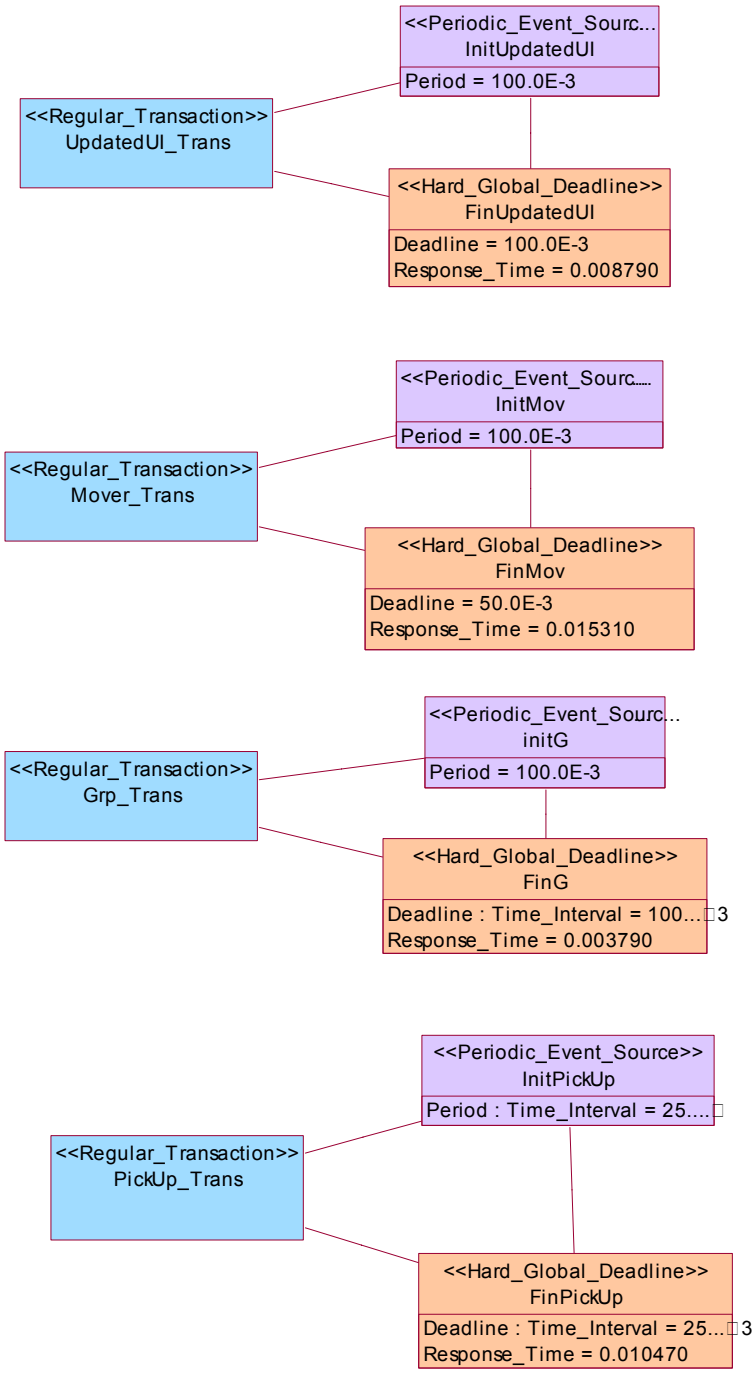

<<Periodic_Event_Sourc...
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
Response_Time = 0.008790
<<Periodic_Event_Sourc.....
InitMov
Period = 100.0E-3
<<Regular_Transaction>>
Mover_Trans
<<Hard_Global_Deadline>>
FinMov
Deadline = 50.0E-3
Response_Time = 0.015310
<<Periodic_Event_Sourc...
initG
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinG
Deadline : Time_Interval = 100...□3
Response_Time = 0.003790
<<Periodic_Event_Source>>
InitPickUp
Period : Time_Interval = 25....□
<<Regular_Transaction>>
PickUp_Trans
<<Hard_Global_Deadline>>
FinPickUp
Deadline : Time_Interval = 25...□3
Response_Time = 0.010470

Figura 6

5.7 Prueba6.

5.7.1 Análisis de planificabilidad del Modelo6.

En este modelo, vuelve a haber el incremento de la complejidad del sistema, debido a la inclusión de unos nuevos componentes. Se añade un nuevo servidor, con un desacoplador que media con el sistema. Este nuevo modulo añadido se encargará de controlar una nueva transacción que se incluirá en el escenario del sistema.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema resultando ser planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción **=> UpDateUI_Trnas**

Evento con requerimiento temporal **=>EndUpDateUI**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitActualizar**

Deadline **=>0.100**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.008790**

Best_Global_Response_Times **=> 0.008790**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de controlar que el servidor de cinemática, actualice continuamente el valor del *flag* de colisión, en la variable *colision,* perteneciente al recurso compartido.

Transacción **=> ActCol_Trnas**

Evento con requerimiento temporal **=>FinActCol**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitActCol**

Deadline **=>0.050**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.016330**

Best_Global_Response_Times **=> 0.016330**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción accederá continuamente al valor de la variable *colision*, del recurso compartido. En caso de que esté activada, detendrá el sistema.

Transacción => **PickUp_Trnas**

Evento con requerimiento temporal =>**FinPickUp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitPickUp**

Deadline =>**0.050**

Resultado del analisis:

Worst_Global_Response_Times => **0.005350**

Best_Global_Response_Times => **0.005350**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción => **Grp_Trnas**

Evento con requerimiento temporal =>**FinGrp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitGrp**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.005080**

Best_Global_Response_Times => **0.016330**

Jitters => **-0.011250**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el nuevo servidor (*NewServer*), en el recurso compartido.

Transacción => **New_Trnas**

Evento con requerimiento temporal =>**FinNew**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitNew**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.016330**

Best_Global_Response_Times => **0.016330**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holaguras calculadas en este modelo, son:

- Transacción	=>UpDateUI_Trans
slack	=> 475.78 %
- Transacción	=>ActCol_Trans
slack	=> 207.03 %
- Transacción	=>PickUp_Trans
slack	=> 850.00 %
-Transacción	=>Grp_Trans
slack	=> 634.38 %
-Transacción	=>New_Trans
slack	=> 253.91 %

También calcula la holgura total del sistema => 92.19%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 7, se ve como se han incluido estos resultados.

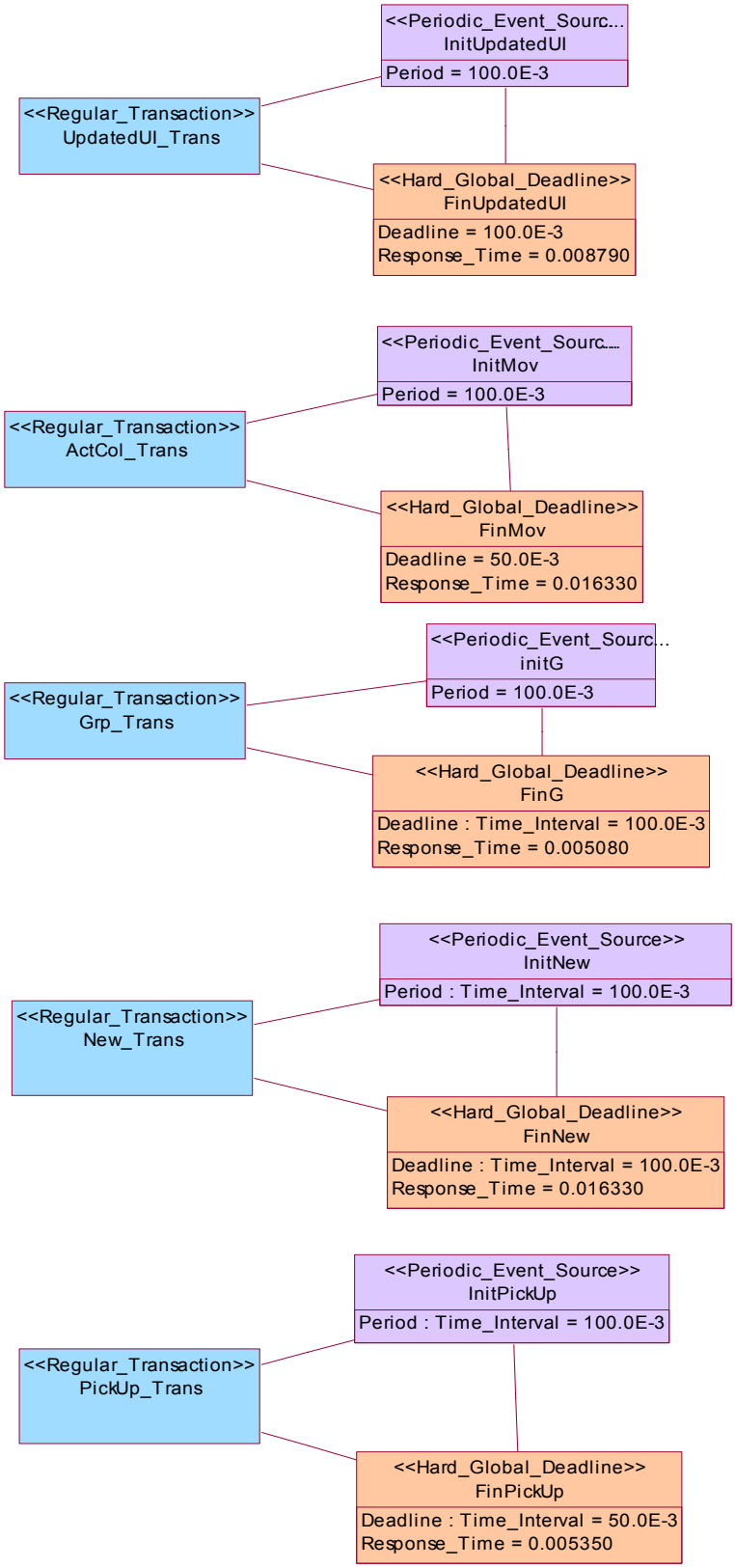

<<Periodic_Event_Sourc...
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
Response_Time = 0.008790
<<Periodic_Event_Sourc....
InitMov
Period = 100.0E-3
<<Regular_Transaction>>
ActCol_Trans
<<Hard_Global_Deadline>>
FinMov
Deadline = 50.0E-3
Response_Time = 0.016330
<<Periodic_Event_Sourc...
initG
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinG
Deadline : Time_Interval = 100.0E-3
Response_Time = 0.005080
<<Periodic_Event_Source>>
InitNew
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
New_Trans
<<Hard_Global_Deadline>>
FinNew
Deadline : Time_Interval = 100.0E-3
Response_Time = 0.016330
<<Periodic_Event_Source>>
InitPickUp
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
PickUp_Trans
<<Hard_Global_Deadline>>
FinPickUp
Deadline : Time_Interval = 50.0E-3
Response_Time = 0.005350

Figura 7

5.7.2 Análisis de planificabilidad del Modelo6v2.

En este modelo, se cambia el comportamiento, definido en la primera versión, del recurso compartido. Ahora se trata de un objeto protegido, que actúa de manera pasiva.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema resultando ser planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.008790

Best_Global_Response_Times => 0.008790

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de controlar que el servidor de cinemática, actualice continuamente el valor del *flag* de colisión, en la variable *colision,* perteneciente al recurso compartido.

Transacción => ActCol_Trnas

Evento con requerimiento temporal =>FinActCol

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActCol

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.015040

Best_Global_Response_Times => 0.015040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción accederá continuamente al valor de la variable *colision*, del recurso compartido. En caso de que esté activada, detendrá el sistema.

Transacción => PickUp_Trnas

Evento con requerimiento temporal =>FinPickUp

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitPickUp**

Deadline **=>0.050**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.004060**

Best_Global_Response_Times **=> 0.004060**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción **=> Grp_Trnas**

Evento con requerimiento temporal **=>FinGrp**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitGrp**

Deadline **=>0.100**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.003790**

Best_Global_Response_Times **=> 0.015040**

Jitters **=> -0.011250**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el nuevo servidor (NewServer), en el recurso compartido.

Transacción **=> New_Trnas**

Evento con requerimiento temporal **=>FinNew**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitNew**

Deadline **=>0.100**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.015040**

Best_Global_Response_Times **=> 0.015040**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>UpDateUI_Trans

slack => 527.34 %

- Transacción =>ActCol_Trans
  slack => 232.81 %
- Transacción =>PickUp_Trans
  slack => 1148.4 %
-Transacción =>Grp_Trans
  slack => 1407.0 %
-Transacción =>New_Trans
  slack => 351.56 %

También calcula la holgura total del sistema => 113.28%

Finalmente, se insertaran los resultados obtenidos del análisis dentro de la vista de los escenarios de tiempo real. En la figura 8, se ve como se han incluido estos resultados.

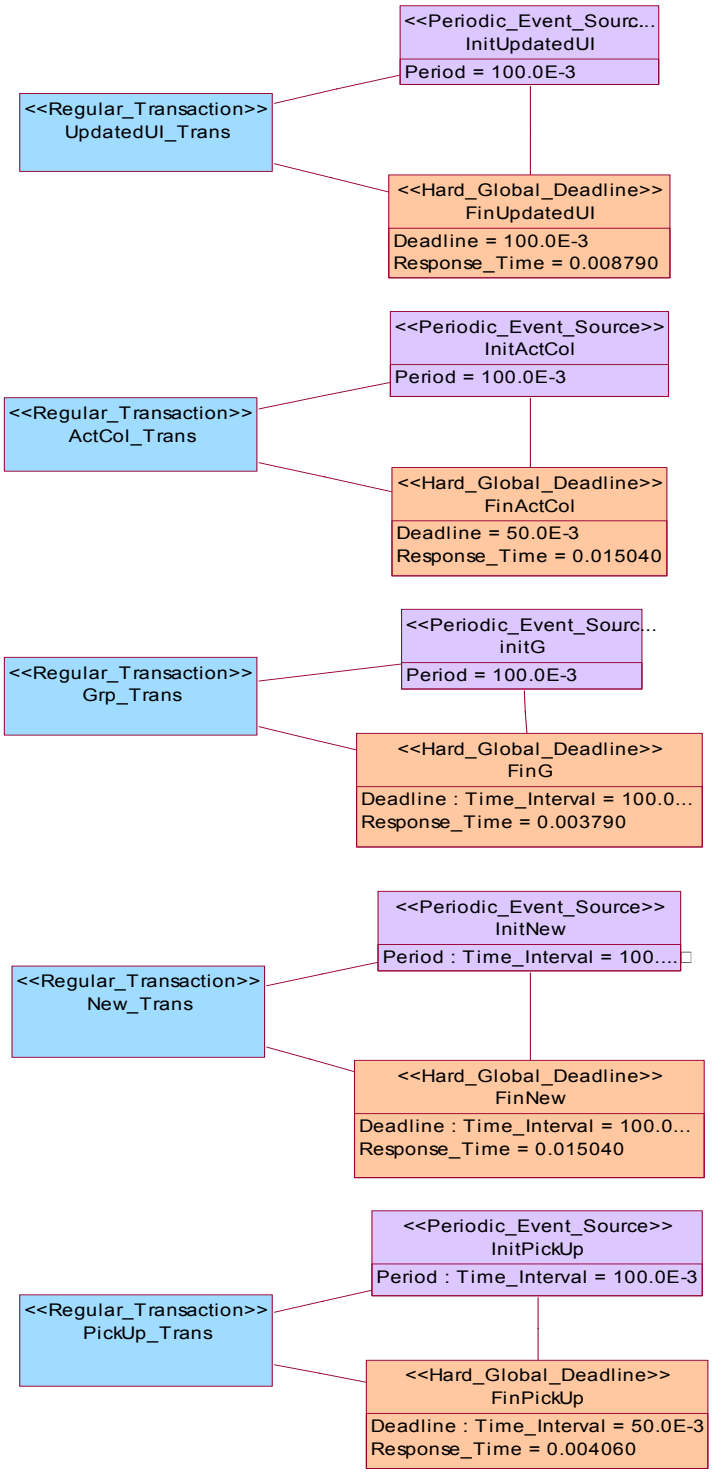

<<Periodic_Event_Sourc...
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
Response_Time = 0.008790
<<Periodic_Event_Source>>
InitActCol
Period = 100.0E-3
<<Regular_Transaction>>
ActCol_Trans
<<Hard_Global_Deadline>>
FinActCol
Deadline = 50.0E-3
Response_Time = 0.015040
<<Periodic_Event_Sourc...
initG
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinG
Deadline : Time_Interval = 100.0...
Response_Time = 0.003790
<<Periodic_Event_Source>>
InitNew
Period : Time_Interval = 100....
<<Regular_Transaction>>
New_Trans
<<Hard_Global_Deadline>>
FinNew
Deadline : Time_Interval = 100.0...
Response_Time = 0.015040
<<Periodic_Event_Source>>
InitPickUp
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
PickUp_Trans
<<Hard_Global_Deadline>>
FinPickUp
Deadline : Time_Interval = 50.0E-3
Response_Time = 0.004060

Figura 8

5.8 Prueba7.

5.8.1 Análisis de planificabilidad del Modelo7.

Se plantea un modelo distribuido simple, con una funcionalidad similar a la explicada en el Modelo1, pero con dos controladores de alto nivel.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando ser planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Empezaremos por explicar la transacción *Mover_Transaction*, en la cual se comprueba el estado del flag de colisión, y en caso de estar activado manda parar al sistema.

Transacción **=> Mover_Trnas**

Evento con requerimiento temporal **=>EndMover**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitMover**

Deadline **=>0.050**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.035600**

Best_Global_Response_Times **=> 0.035600**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción **=> UpDateUI_Trnas**

Evento con requerimiento temporal **=>EndUpDateUI**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitActualizar**

Deadline **=>0.100**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.030040**

Best_Global_Response_Times **=> 0.030040**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son

- Transacción =>Mover_Trans

slack => 39.84 %

- Transacción =>UpDateUI_Trans

slack => 112.50 %

También calcula la holgura total del sistema => 39.84%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 9, se ve como se han incluido estos resultados.

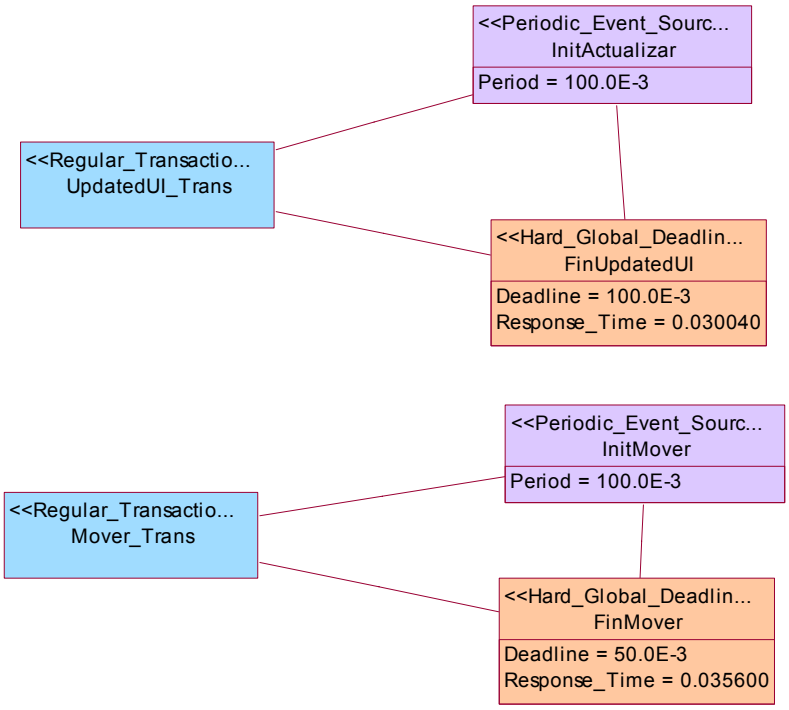


Figura 9

5.9 Prueba8.

5.9.1 Análisis de planificación del Modelo8.

En este modelo, vuelve a haber el incremento de la complejidad del sistema, debido a la inclusión de nuevos componentes. Se añade un nuevo servido, con un desacoplador que medie con el sistema. Este nuevo modulo añadido se encargará de controlar una nueva transacción que se incluirá en el escenario del sistema.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando ser planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal **=>EndUpDateUI**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitActualizar**

Deadline **=>0.100**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.008040**

Best_Global_Response_Times **=> 0.008040**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

En esta transacción se comprueba el estado del flag de colisión, y en caso de estar activado manda parar al sistema.

Transacción **=> Mover_Trnas**

Evento con requerimiento temporal **=>FinActCol**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitActCol**

Deadline **=>0.050**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.005300**

Best_Global_Response_Times **=> 0.005300**

Jitters **=> 0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción **=> Grp_Trnas**

Evento con requerimiento temporal **=>FinGrp**

Tipo de requerimiento temporal **=>Global_Hard_Deadline**

Evento de referencia **=>InitGrp**

Deadline **=>0.050**

Resultado del analisis:

Worst_Global_Response_Times **=> 0.014080**

Best_Global_Response_Times **=> 0.014080**

Jitters **=> 0.00**

Requerimiento cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requerimiento.

Las holguras calculadas en este sistema, son:

- Transacción =>UpDateUI_Trans

slack => 900.78 %

- Transacción =>Mover_Trans
  slack => 859.38 %
- Transacción =>Grp_Trans
  slack => 514.84 %

También calcula la holgura total del sistema => 264.84%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 10, se ve como se han incluido estos resultados.

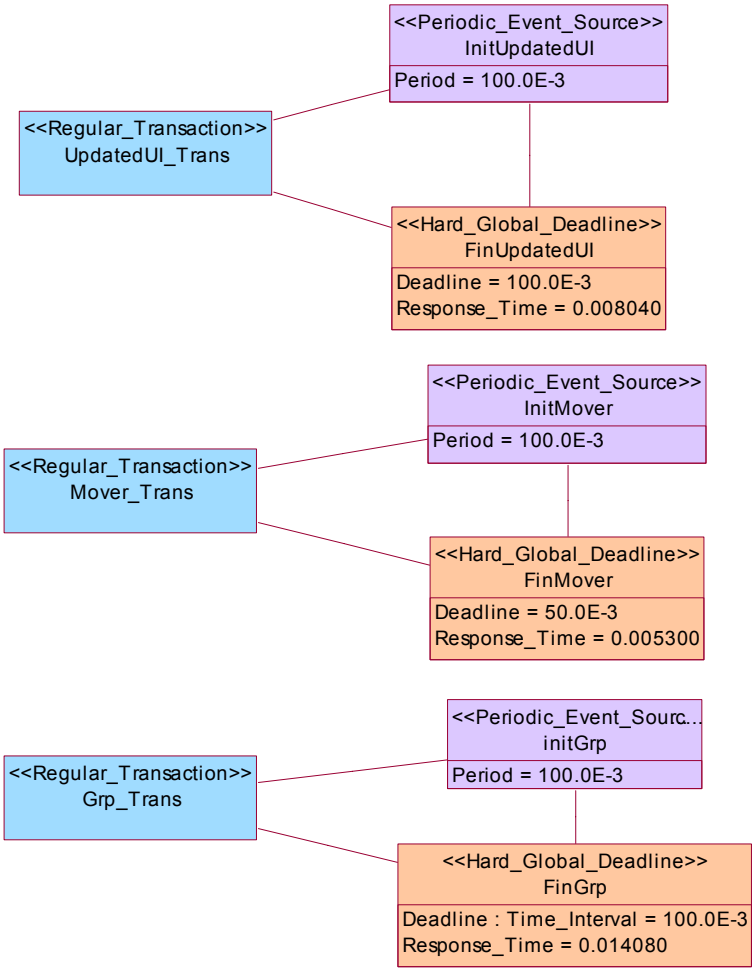


Figura 10

5.9.2 Análisis de planificación del Modelo8v2.

En este modelo, se cambia el comportamiento, definido en la primera versión, del recurso compartido. Ahora se trata de un objeto protegido, que actua de manera pasiva.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando este planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.012040

Best_Global_Response_Times => 0.012040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

En esta transacción se comprueba el estado del flag de colisión, y en caso de estar activado manda parar al sistema.

Transacción => Mover_Trnas

Evento con requerimiento temporal =>FinActCol

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActCol

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.005080

Best_Global_Response_Times => 0.005080

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción => Grp_Trnas

Evento con requerimiento temporal =>FinGrp

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitGrp

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.013040

Best_Global_Response_Times => 0.013040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>UpDateUI_Trans
  slack => 732.81 %
- Transacción =>Mover_Trans
  slack => 897.66 %
- Transacción =>Grp_Trans
  slack => 536.72 %

También calcula la holgura total del sistema => 232.03%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 11, se ve como se han incluido estos resultados.

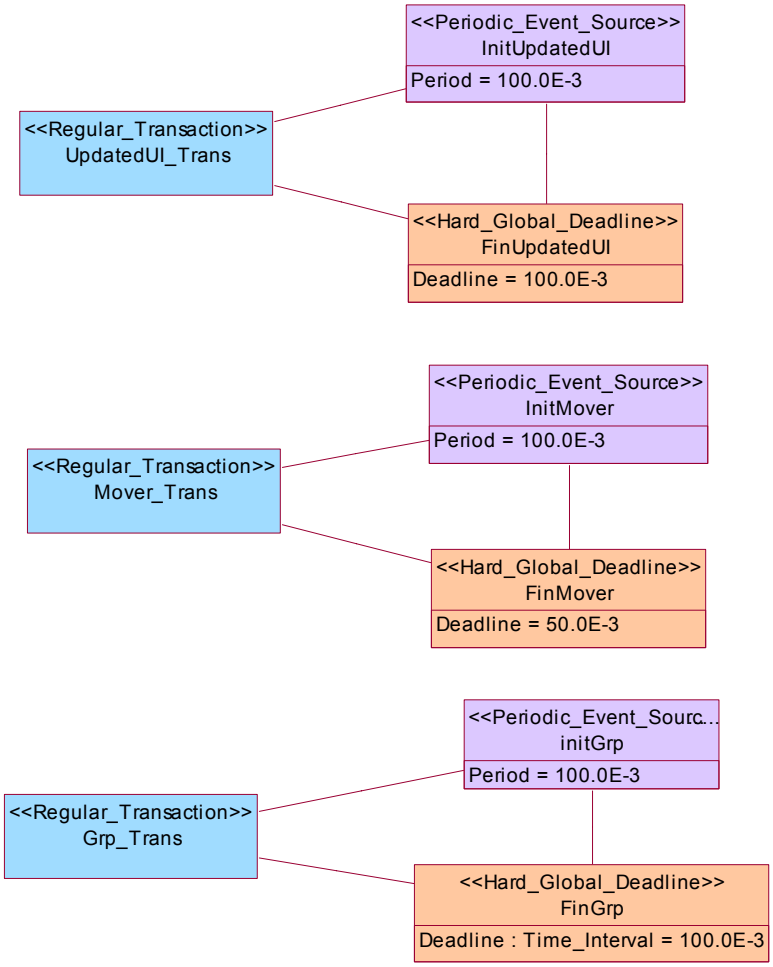


Figura 101

5.10 Prueba9

5.10.1 Análisis de planificabilidad del Modelo9.

En este modelo, vuelve a haber el incremento de la complejidad del sistema, debido a la inclusión de nuevos componentes. Se añade un nuevo servidor, con un desacoplador que medie con el sistema. Este nuevo modulo añadido se encargará de controlar una nueva transacción que se incluirá en el escenario del sistema.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando éste planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.012040

Best_Global_Response_Times => 0.012040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

En esta transacción se comprueba el estado del flag de colisión, y en caso de estar activado manda parar al sistema.

Transacción => Mover_Trnas

Evento con requerimiento temporal =>FinMover

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitMover

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.005080

Best_Global_Response_Times => 0.005080

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el nuevo servidor (NewServer), en el recurso compartido.

Transacción => **New_Trnas**

Evento con requerimiento temporal =>**FinNew**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitNew**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.004080**

Best_Global_Response_Times => **0.004080**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción => **Grp_Trnas**

Evento con requerimiento temporal =>**FinGrp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitGrp**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.014060**

Best_Global_Response_Times => **0.014060**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>UpDateUI_Trans

  slack => 535.16 %

- Transacción =>Mover_Trans

  slack => 897.66 %

- Transacción =>New_Trans

  slack => 1284.4 %

Transacción =>Grp_Trans

  slack => 428.13 %

También calcula la holgura total del sistema => 182.81%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 12, se ve como se han incluido estos resultados.

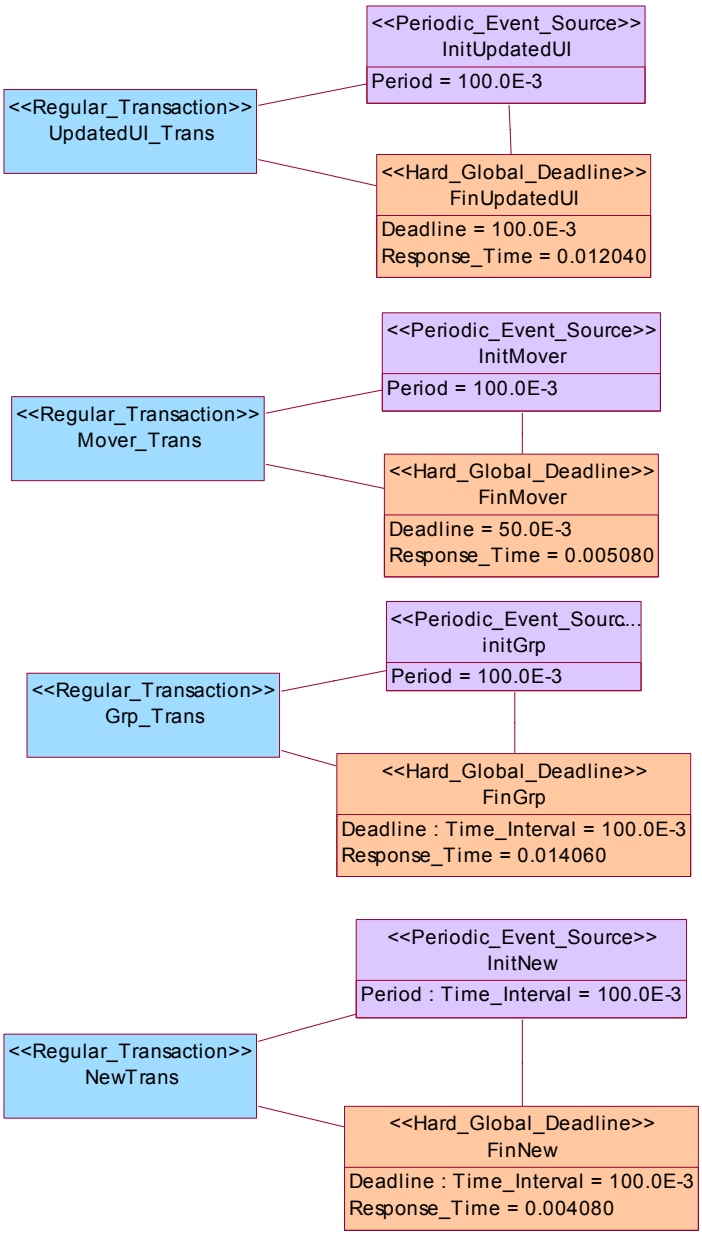


Figura 112

5.10.2 Análisis de planificabilidad del Modelo9v2.

En este modelo, vuelve a haber el incremento de la complejidad del sistema, debido a la inclusión de nuevos componentes. Se añade un nuevo servidor, con un desacoplador que medie con el sistema. Este nuevo modulo añadido se encargará de controlar una nueva transacción que se incluirá en el escenario del sistema.

Una vez terminado el diseño del modelo, se realiza el análisis del sistema, resultando éste planificable.

A continuación se exponen los requisitos temporales que se deben cumplir en cada transacción y la respuesta temporal dada por MAST, para comprobar que dichos requisitos han sido cumplidos.

Esta transacción se va a encargar de actualizar de manera periódica la interfaz de usuario.

Transacción => UpDateUI_Trnas

Evento con requerimiento temporal =>EndUpDateUI

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitActualizar

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.012040

Best_Global_Response_Times => 0.012040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

En esta transacción se comprueba el estado del flag de colisión, y en caso de estar activada manda parar al sistema.

Transacción => Mover_Trnas

Evento con requerimiento temporal =>FinMover

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitMover

Deadline =>0.050

Resultado del analisis:

Worst_Global_Response_Times => 0.005080

Best_Global_Response_Times => 0.005080

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el nuevo servidor (NewServer), en el recurso compartido.

Transacción => New_Trans

Evento con requerimiento temporal =>FinNew

Tipo de requerimiento temporal =>Global_Hard_Deadline

Evento de referencia =>InitNew

Deadline =>0.100

Resultado del analisis:

Worst_Global_Response_Times => 0.003040

Best_Global_Response_Times => 0.003040

Jitters => 0.00

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Esta transacción se encarga de actualizar los datos proporcionados por el servidor gráfico (GrpServer), en el recurso compartido.

Transacción => **Grp_Trnas**

Evento con requerimiento temporal =>**FinGrp**

Tipo de requerimiento temporal =>**Global_Hard_Deadline**

Evento de referencia =>**InitGrp**

Deadline =>**0.100**

Resultado del analisis:

Worst_Global_Response_Times => **0.013040**

Best_Global_Response_Times => **0.013040**

Jitters => **0.00**

Requisito cumplido, ya que los tiempos de respuesta calculados por MAST son menores que el plazo establecido como requisito.

Las holguras calculadas en este modelo, son:

- Transacción =>UpDateUI_Trans
  slack => 552.34 %
- Transacción =>Mover_Trans
  slack => 897.66 %
- Transacción =>New_Trans
  slack => 2209.4 %

Transacción =>Grp_Trans
slack => 509.38 %

También calcula la holgura total del sistema => 200.78%

Finalmente, se insertaran los resultados obtenidos del análisis dentro del a vista de los escenarios de tiempo real. En la figura 13, se ve como se han incluido estos resultados.

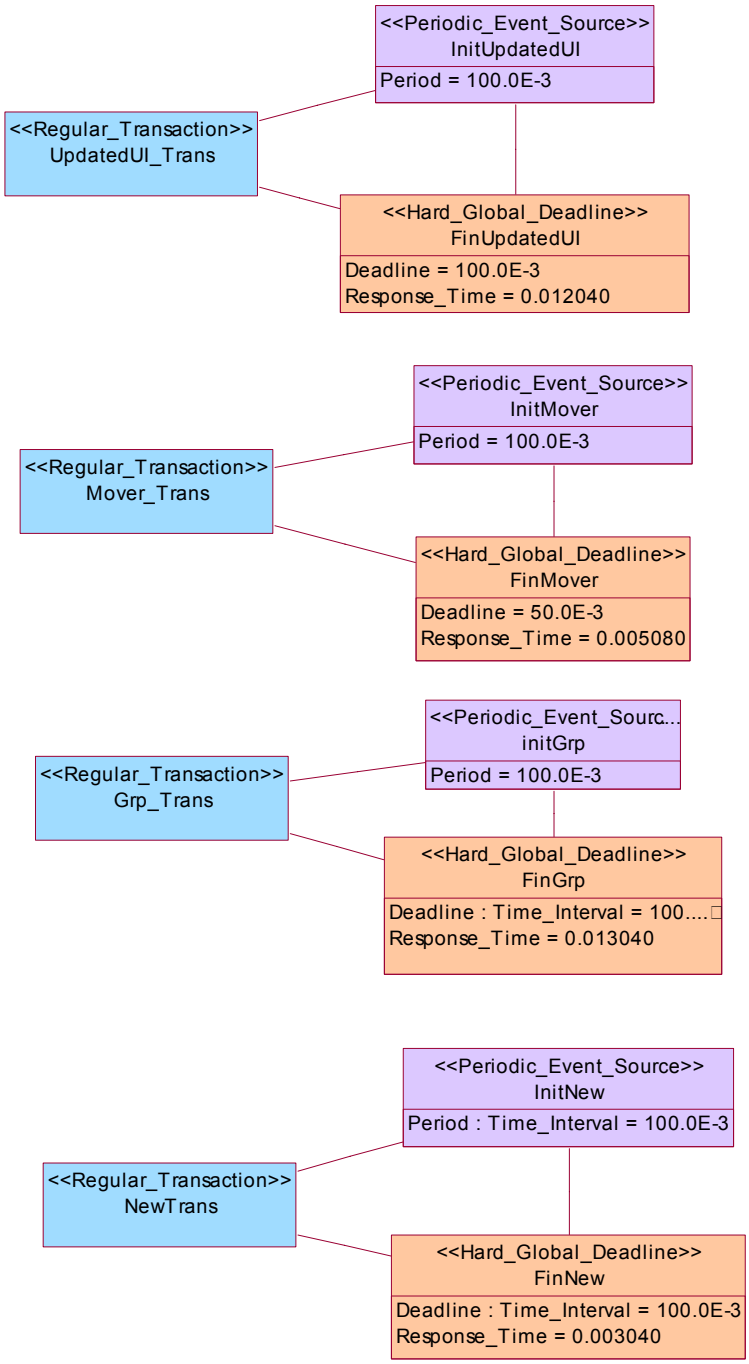

<<Periodic_Event_Source>>
InitUpdatedUI
Period = 100.0E-3
<<Regular_Transaction>>
UpdatedUI_Trans
<<Hard_Global_Deadline>>
FinUpdatedUI
Deadline = 100.0E-3
Response_Time = 0.012040
<<Periodic_Event_Source>>
InitMover
Period = 100.0E-3
<<Regular_Transaction>>
Mover_Trans
<<Hard_Global_Deadline>>
FinMover
Deadline = 50.0E-3
Response_Time = 0.005080
<<Periodic_Event_Sourc...
initGrp
Period = 100.0E-3
<<Regular_Transaction>>
Grp_Trans
<<Hard_Global_Deadline>>
FinGrp
Deadline : Time_Interval = 100....
Response_Time = 0.013040
<<Periodic_Event_Source>>
InitNew
Period : Time_Interval = 100.0E-3
<<Regular_Transaction>>
NewTrans
<<Hard_Global_Deadline>>
FinNew
Deadline : Time_Interval = 100.0E-3
Response_Time = 0.003040

Figura 12

5.11 Interpretación de los resultados.

Lo más destacable a la hora de interpretar los resultados expuestos anteriormente es el hecho de que todos los modelos diseñados han resultado planificables. Además en todos los modelos se ha calculado una amplia holgura del sistema, indicándonos la posibilidad de un aumento bastante sustancial en el tiempo estimado para las operaciones que son ejecutadas por el mismo.

Una vez visto que todos los sistemas planteados en este trabajo son planificables, se va intentar comparar los sistemas de funcionalidad similar, pero con arquitecturas diferentes, desde el punto de vista del comportamiento temporal.

Se van a agrupar las distintas pruebas en tablas para que así sea la comparación más sencilla. En primer término expondremos los modelos más sencillos de cada arquitectura. Éstas serían las pruebas 1,2 y 7 que se muestran en la Tabla 1.

Tabla 1

	Tiempos de Respuesta		
Transacción	**Modelo1**	**Modelo2**	**Modelo7**
UpDateUI	0.008040	0.008040	0.030040
Mover	0.014300	0.016320	0.035600
Slack system	250.78%	207.81%	113.28%

En esta tabla 1, se ve claramente que de los tres modelos comparados, el más óptimo es sería el Modelo1, ya que tiene una mayor holgura que los modelos 2 y 7, y mejor o igual tiempo de respuesta en las dos transacciones calculadas. Como ya se ha mencionado, al tener mayor holgura de sistema, indica que el tiempo de las operaciones puede aumentar en mayor grado para el Modelo1. Es decir, si no hubiera que aumentar el número de componentes se optaría por arquitectura centralizada, como se le ha denominado a lo largo del trabajo.

Las Pruebas 3 y 4, son unas pruebas intermedias, en las cuales se le añaden nuevos componentes, que son un recurso compartido y un modulo de desacoplo al servicio de cinemática, respectivamente. A partir de la Prueba3 aparecerá una transacción, PickUp_Transaction, la cual aumenta la complejidad del escenario. Los datos recogidos de MAST en estas pruebas se mantienen con valores muy similares en los siguientes modelos, como se observa en el punto anterior.

A continuación compararemos los resultados obtenidos del análisis realizado para los modelos diseñados en las pruebas 5 y 8, esta comparación se puede apreciar en la Tabla 2.

Tabla 2

	Tiempos de Respuesta			
Transacción	**Modelo5**	**Modelo5v2**	**Modelo8**	**Modelo8v2**
UpDateUI	0.008040	0.008740	0.008040	0.012040
ActCol	0.014080	0.015310	0.005300	0.005080
PickUp	0.005080	0.010470		
Grp	0.014080	0.003790	0.01480	0.013040
Slack system	141.41%	40.63%	264.84%	232.03%

En esta tabla podemos comparar en primer lugar los modelos correspondientes a cada prueba entre si. Observaremos como, en este caso, al contrario del anterior, los sistemas más óptimos son los pertenecientes a la Prueba8. Se trataría de un modelo distribuido, implicando que esta arquitectura resulta ser mas adecuada cuando mas complejo sea el sistema, teniendo en cuenta que en el modelo distribuido las transacciones ActCol y PickUp se condensan en una sola transacción, como se explicó en el apartado correspondiente al desarrollo de los modelos.

También se puede observar una relación entre las dos versiones de modelo definidas dentro de la misma prueba; en ambas se confirma una misma tendencia. Esta tendencia implica que resulta más efectivo en cuanto a la respuesta temporal del sistema los modelos que no tienen definido el recurso compartido como un objeto protegido. Sin embargo, el hecho de diseñar el recurso compartido como un objeto protegido aumenta la fiabilidad del sistema, aunque le haga perder un mayor tiempo de respuesta.

Por último, se pasa a comparar los sistemas más complejos diseñados, correspondientes a las Pruebas 6 y 9. Así se podrá observar si se mantiene la tendencia de que cuando mas complicado sea el sistema, mas óptimo es utilizar una arquitectura distribuida.

Tabla 3

	Tiempos de Respuesta			
Transacción	**Modelo6**	**Modelo6v2**	**Modelo9**	**Modelo9v2**
UpDateUI	0.008790	0.008790	0.012040	0.012040
ActCol	0.016330	0.015040	0.005080	0.005080
PickUp	0.005350	0.004060		
Grp	0.005080	0.003790	0.014060	0.013040
New	0.016330	0.015040	0.004080	0.003040
Slack system	92.19%	113.28%	182.81%	200.78%

Se compara en primer lugar los modelos correspondientes a cada prueba entre si, y observaremos como siguiendo el patrón del caso anterior, se observa que los sistemas mas óptimos son los pertenecientes a la Prueba9, que se trataría de un

modelo distribuido Así, se puede ya decir que cuando más complejo es el sistema más eficaz es la arquitectura distribuida.

Se puede observar una relación entre las dos versiones de modelo definidas dentro de la misma prueba, en ambas se confirma una misma tendencia. Pero en este caso esta tendencia implica que el sistema que utiliza un recurso compartido como un objeto protegido, disminuye los tiempos de respuesta, aunque se mejora la fiabilidad del sistema.

Capítulo 6
Implementación de los Modelos en Ada95

6.1 Introducción.

Una vez terminado el estudio sobre el diseño de los modelos arquitectónicos sobre la herramienta MAST, dichos modelos, se han implementado en el lenguaje Ada95, ya que se trata de un lenguaje especializado para sistemas de tiempo real.

Como se dijo en la introducción se eligió Ada95, para llevar a cabo este trabajo debido a sus características:

- Legibilidad: programas fáciles de leer.
- Tipado fuerte: todo objeto tiene componentes de valor definido (es mucho más fuerte que Pascal).
- Capaz de construir grandes programas: compilación separada de los distintos paquetes (módulos).
- Manejo de excepciones: para incorporar mecanismos de tolerancia de fallos.
- Abstracción de datos.
- Unidades genéricas: que se pueden agrandar todo lo que queramos con nuevas funciones.
- Procesamiento paralelo: aspecto fundamental para este trabajo.
- Alta transportabilidad de los programas entre distintas plataformas: UNIX, OS/2, Win98 ...

Dentro de este trabajo se han utilizado diferentes mecanismos del lenguaje, entre los que hay que destacar los siguientes: Concurrencia (*Tasking)*, asignación de prioridades (*pragma priority)*, sincronización entre tareas (citas extendidas) y objetos protegidos.

Se denomina *tasking* a la utilización de tareas para la programación concurrente. Ada95 es un lenguaje concurrente, ya que permite la utilización de hilos (*Task*) de ejecución “potencialmente” paralela. Los sistemas de tiempo real son inherentemente concurrentes, por lo cual, Ada95 es una buena elección a la hora de tener que elegir lenguaje para implementarlos.

En Ada95 las tareas pueden ser declaradas en cualquier nivel del programa. Las tareas se crean, cuando lo hacen el resto de variables del bloque en el que están de claradas. Las tareas comienzan su ejecución antes de que se ejecute la primera sentencia del bloque en el que están declaradas, cuyo cuerpo se ejecuta

concurrentemente con éstas. Al igual que los paquetes, las tareas tienen una parte de especificación y una parte de cuerpo. Las tareas se definen como tipos, aunque existe un tipo de tarea anónimo que permite declarar la tarea y su tipo al mismo tiempo. Sin son necesarias varias instancias de un mismo proceso, se puede definir un tipo proceso. Las tareas en Ada solo permiten pasar como parámetros tipos discretos y tipos acceso.

Dentro de las tareas se pueden definir citas, para comunicarse síncronamente con otras tareas. Cuando durante la ejecución de la aplicación se llama a una cita, la tarea llamante se esperará suspendida, a que la tarea llamada termine de ejecutar el código asociado a la cita.

Otro recurso muy importante a la hora de definir sistemas de tiempo real en Ada95, son los objetos protegidos. Los objetos protegidos. El tipo definido en Ada como ***protected***, el cual sirve para identificar objetos protegidos, tiene definidos mecanismos propios de sincronización para poder controlar el acceso compartido a los recursos definidos en él.

Ada95 también permite la definición de prioridades, por medio del ***pragma priority***. La asignación de prioridades es muy útil para los sistemas de tiempo real, ya que permite la planificación del sistema, como ya se ha explicado a lo largo de este trabajo. La asignación de estas prioridades se ha realizado mediante la utilización del método de planificación RMA.

El objetivo impuesto en este trabajo, para este apartado, era la creación de un esqueleto de los modelos diseñados, con el cual posteriormente se puedan realizar pruebas con código real bajo las condiciones adecuadas, y obtener unos datos empíricos sobre los tiempos de respuesta de los modelos, y comparar resultados con los obtenidos con UML-MAST.

Una vez implementados los esqueletos de los modelos y aunque no se disponía de un sistema operativo de tiempo real, se ha ejecutado el código y se han medido y tabulado los tiempos medios de ejecución. En este capitulo se realizará una interpretación de los resultados obtenidos

6.2 Tabla de resultados.

Una vez implementados los modelos en el lenguaje Ada95, se han realizado diez ejecuciones sobre cada modelo, para poder obtener una tabla de resultados de la que calcular unos valores medios, intentando obtener un valor lo mas representativo posible.

En la Tabla 1, se ve un ejemplo de cómo se han tabulado los datos. Esta tabla es parte de la hoja de cálculo donde se han tabulado todos los tiempos obtenidos en cada modelo. La hoja de cálculo se adjunta con este proyecto en un archivo con nombre ResponseTimes.xcl.

En la Tabla 1 se ve el modelo de la hoja de cálculo. En primer lugar se observa una tabla con los valores de tiempo de respuesta calculado. Se utilizarán para calcular los valores que se ven en la tabla inferior, donde se describen los valores mas importantes, como son el valor medio, el mejor tiempo y el peor tiempo.

Tabla 1

Ada Modelo1

Transacción / Nº de paso	UpDatedIU_Trans	UpDatedColFlag
1	4,00000E-05	1,83000E-04
2	3,10000E-05	1,76000E-04
3	3,10000E-05	1,31000E-04
4	3,90000E-05	1,79000E-04
5	3,90000E-05	1,27000E-04
6	3,90000E-05	1,34000E-04
7	3,10000E-05	1,78000E-04
8	3,10000E-05	1,27000E-04
9	3,10000E-05	1,28000E-04
10	3,90000E-05	1,69000E-04

MEDIA	3,51000E-05	1,53200E-04

Datos temporales más relevantes

	UpDatedIU_Trans	UpDatedColFlag
TIEMPO MEDIO	3,51000E-05	1,53200E-04
MEJOR TIEMPO	3,10000E-05	1,27000E-04
PEOR TIEMPO	4,00000E-05	1,83000E-04

Estas tablas nos van a facilitar la observación y comparación de los tiempos de respuesta, y pueden reutilizarse para futuras comprobaciones.

6.3 Interpretación de los datos obtenidos.

En este apartado se va a realizar una interpretación de los resultados obtenidos de las pruebas realizadas con el código Ada95. Se va a seguir un orden de comparación similar al expuesto en el apartado 5.3, comparando los resultados obtenidos entre los modelos de funcionalidad similar. Posteriormente se hará una comparación respecto a los resultados obtenidos por MAST, en el Capítulo5.

En primer lugar se van a comparar los tiempos de respuesta calculados para las pruebas 1, 2 y 7. Estas pruebas tienen una misma funcionalidad pero una arquitectura diferente. Estas pruebas definen los sistemas básicos, la Prueba1, define la arquitectura centralizada. La Prueba2, realiza una variación sobre la Prueba1, en la que el controlador de bajo nivel (*LLC*) es el encargado de actualizar los datos en el servidor de cinemática (*CinServer*) y la Prueba7, representa el modelo más sencillo de la arquitectura distribuida.

Tabla 2

	Tiempos de Respuesta		
Transacción	**Modelo1**	**Modelo2**	**Modelo7**
UpDateUI	3,51000E-05	3,92000E-05	5,58000E-05
Mover	1,53200E-04	3,88800E-04	1,42300E-04

En este caso es difícil establecer una relación clara, debido a que se trabaja con valores muy pequeños, y hay muy poca diferencia entre ellos, pero se puede observar como el Modelo1, parece tener unos valores medios de respuesta ligeramente mejores que a los de los Modelos 2 y 7. Aunque hay que decir que no se puede considerar una conclusión definitiva.

En el Modelo3 se hace la inclusión del recurso compartido. En este modelo ya aparece una variación significativa en los tiempos de respuesta. Concretamente esta variación se produce en *UpDateUI_Transaction*. Si se observa el valor medio que aparece en la hoja de cálculo, se ve que no tiene ninguna correspondencia con los valores obtenidos en la tabla anterior para esta transacción. Es un tiempo que ronda el valor de 1 segundo, muy superior a los obtenidos hasta ahora.

Este valor, si volviéramos a ver las especificaciones temporales del Modelo3, indicaría, que este sistema no es planificable, ya que se trata de una transacción que se activa con un evento periódico cada 100ms, y en cada una de las pruebas efectuadas nunca puede cumplir este requisito. Esto crea un conflicto respecto a los datos obtenidos por MAST, ya que este si se consideraba un sistema planificable.

En el Modelo4 se vuelve a dar esta circunstancia descrita anteriormente para la transacción *UpDateUI_Transaction*, el resto de datos mantienen unos valores normales.

En la siguiente tabla se van a comparar los modelos correspondientes a las Pruebas 5 y 8. Los resultados de las pruebas entre si, y los resultados de los dos

modelos definidos dentro de cada prueba, así como la relación entre las dos versiones diseñadas en cada prueba.

Los modelos correspondientes a la Prueba5 y la Prueba8, representan un sistema en el cual aparece un nuevo servidor, un servidor grafico (*GrpServer*). La Prueba5 representa a la arquitectura centralizada, mientras que la Prueba8 representa la arquitectura distribuida.

Tabla 3

	Tiempos de Respuesta			
Transacción	**Modelo5**	**Modelo5v2**	**Modelo8**	**Modelo8v2**
UpDateUI	1,00573E+00	1,00493E+00	1,81200E-04	3,50000E-05
ActCol	1,30700E-04	9,67000E-05	7,16500E-04	1,61700E-04
PickUp	1,24700E-04	1,13900E-04		
Grp	1,68220E-03	4,23680E-03	3,35000E-04	3,47500E-04

En la Tabla2, se pueden comparar los tiempos correspondientes a la Prueba5 y la Prueba8 En la tabla se confirma la tendencia que se ha producido en las Prueba3 y Prueba4, respecto a la transacción *UpDateUI_Transaction,* en la que el tiempo de respuesta de ésta sobrepasa los límites impuestos por el diseño del sistema en los requisitos temporales, cosa que tampoco pasaba con los modelos correspondientes en MAST. Por el contrario en la Prueba 8, los datos temporales se ajustan a los requisitos impuestos para el sistema. Esto se puede interpretar suponiendo que la arquitectura distribuida es más óptima a la hora de implementar este modelo.

Ya para finalizar, se van a comparar los resultados obtenidos en la Prueba6 y la Prueba9.

Los modelos correspondientes a la Prueba6 y la Prueba9, representan unos sistemas en el cual aparece un último servidor, un servidor genérico (*NewServer*). La Prueba6 representa a la arquitectura centralizada, mientras que la Prueba9 representa la arquitectura distribuida.

Tabla 4

	Tiempos de Respuesta			
Transacción	**Modelo6**	**Modelo6v2**	**Modelo9**	**Modelo9v2**
UpDateUI	1,00472E+00	1,00597E+00	3,31000E-05	3,43000E-05
ActCol	1,25900E-04	1,05300E-04	1,90200E-04	1,68800E-04
PickUp	1,51100E-04	1,08800E-04		
Grp	3,92700E-04	1,22040E-03	3,38900E-04	2,36150E-03
New	5,46000E-04	2,38500E-04	1,01160E-03	2,44940E-03

En la Tabla3, se vuelve a confirmar la tendencia de la arquitectura centralizada, correspondiente a la Prueba6, en la que se sobrepasa los límites temporales impuestos para la transacción *UpDateUI_Transaction*. Se puede concluir, que al aparecer más de una transacción en los sistemas con una arquitectura centralizada, éstos no cumplen las especificaciones temporales.

En los sistemas diseñados con una arquitectura distribuida, se cumplen perfectamente todos los requisitos temporales.

Realmente no se puede llevar a cabo una relación entre los diferentes valores de los tiempos de respuesta obtenidos en MAST y con el código en Ada, ya que no se ha dispuesto de un RTOS (Sistema Operativo de Tiempo Real).

Sin embargo, se ha cumplido el objetivo de crear una plantilla para que posteriormente puedan realizarse estas simulaciones bajo unas condiciones más óptimas, para la ejecución de sistemas teleoperados.

Capítulo 7
Conclusiones

7.1 Conclusiones.

Repasando los objetivos planteados durante la introducción de este trabajo, se observa que éstos han sido cumplidos.

Se ha conseguido realizar una comparación de las dos arquitecturas software planteadas, gracias a su modelado mediante la herramienta UML-MAST. Se han realizado la totalidad de los modelos planteados consiguiendo que resultarán planificables para los análisis de MAST. Se ha conseguido poder determinar cual de ellas resultaría mas optima según las características del sistema que se desee realizar.

Posteriormente se han implementado un esqueleto estos diseños, utilizando el lenguaje de programación de alto nivel Ada95. Se han podido recoger y tabular una serie de tiempos de respuesta para cada una de las transacciones planteadas en los Modelos MAST, para intentar hacer una comparación.

La comparación realizada, ha dado como resultado, que los modelos diseñados en MAST bajo la arquitectura que hemos denominado centralizada, no resultan planificables en la realidad, por el contrario a lo que dicen los resultados de los análisis de MAST. Aunque esta última conclusión debe de tomarse bajo mucha cautela, ya que en este trabajo se pretendía crear un esqueleto de las aplicaciones, para que posteriormente pudiera retomarse el estudio con todos los medios adecuados para su realización, como seria el caso de disponer de un sistema operativo que trabajase en tiempo real, además se debe tener en cuenta que los sistemas diseñados en MAST, han sido ejecutados sobre un sistema operativo Windows98, mientras que los modelos implementados en Ada95, han sido ejecutados sobre un sistema operativo Linux Mandrake 9.0. Este hecho también podría causar que la relación entre los datos temporales tomados en ambos casos no pudieran ser comparables.

Por último, cabe decir se considera que se ha dejado este estudio en las condiciones necesarias para su continuación, de tal manera que se puedan ejecutar los sistemas bajo unas condiciones iguales y los medios convenientes para los sistemas de tiempo real. Éste permitirá más adelante determinar si la herramienta MAST, realmente realiza análisis verídicos sobre los sistemas modelados.

Capítulo 8
Referencias

[Baker91] T.P.Baker."*Stack-based scheduling of realtime processes*" Real-Time Systems 3(1), 1991.

[**Drake** et al 2000]: J. M. Drake, M. González Harbour, J.S. Medina, "Mast Real Time View: Graphic UML Tool for Modeling Object Oriented Real Time Systems", Grupo de Computación y Sistemas de Tiempo Real de la Universidad de Cantabria. Internal Report 2000

[Gamma et al 1995]: E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", Addison Wesley, Reading Mass. 1995

[Goodenough&88] J.B.Goodenough and L.Sha, "*The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks*". Ada letters, Special issue: Proc. 2nd International workshop on Real-Time Ada Issues VIII, Vol.7, pp. 20-31, 1988.

J Barnes. "Programing in Ada95". 2ª Edition, Addison-Wesley, 1998.

[Kruchten 1995]: P. Krutchen. "Architectural blue-prints, the 4+1 view of software architecture". IEEE Software, 21(4), April 1995

[Liu&73] C.L.Liu and J.W.Layland. "*Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*", J. ACM, Vol.20, No. 1, January 1973, pp. 46-61.

[Locke92] C.Locke. "*Software architecture for hard real-time applications: cyclic executives versus fixed priority executives*". Real-Time Systems 4(1), pp. 37-53, 1992.

[Rational 2004]: Rational Software Corporation, www.rational.com.
Burns and A. Wellings. "Real-Time System and Programing Languajes". Ed. Addison-Wesley, 3ª Edition, 2001.

[Sha&90] L.Sha, R.Rajkumar and J.P.Lehoczky, "*Priority inheritance protocols: An approach to real-time synchronization*" IEEE Transactions on Computers, 39(9),pp.1175-85,1990.

Ada Reference Manual www.adapower.com/rm95/arm95_toc.html