VISTA UML DE TIEMPO REAL DE SISTEMAS DISEÑADOS BAJO UNA METODOLOGIA ORIENTADA A OBJETOS.

Por José M. Drake <drakej@ctr.unican.es>
Michael González Harbour <mgh@ctr.unican.es>
Julio L. Medina <medinajl@ctr.unican.es>
Grupo de Computadores y Tiempo Real
Dpto. de Electrónica y Computadores
Universidad de Cantabria

I. Introducción.

En este trabajo se presenta una metodología de modelado de sistemas de tiempo real formulada en UML y basada en la herramienta abierta de análisis de sistemas de tiempo real MAST desarrollada por nuestro Grupo [1]. Tomando como referencia el método de descripción de sistemas denominado "4+1 view" que se muestra en la figura 1, el modelo que se propone constituye una componente adicional de la Process View. Con ella el diseñador puede construir gradualmente el modelo de tiempo real del sistema, de igual manera a como construye el modelo lógico del sistema. Desde las fases iniciales del desararrollo el diseñador puede verificar de forma cualitativa y cuantitativa las prestaciones temporales y las performances de tiempo real del sistema.

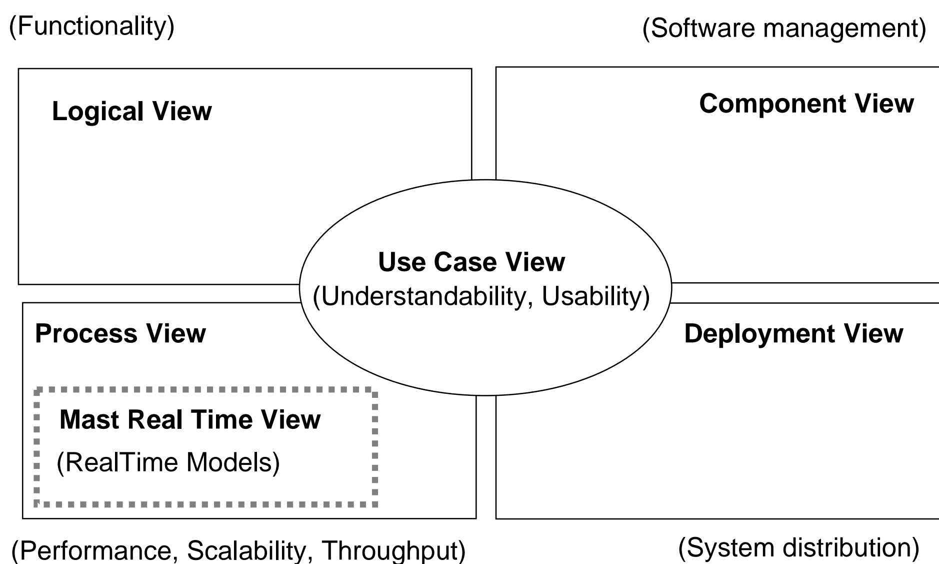

Figure 1: Mast Real Time View in 4+1 view paradigm.

La Mast-RT view está basada en los conceptos y componentes definidos en el entorno MAST (Modeling and Análisis Suite for Real Time Applications). Este es un entorno abierto basado en una descripción textual del modelo del sistema de tiempo real (Mast-File) que se ha propuesto como base a múltiples herramientas de análisis de planificabilidad, análisis de performance,

diseño, animación, etc. de sistemas de tiempo real. El entorno Mast ofrece componentes conceptuales abstractos para modelar:

- Recursos hardware (computers, networks, equipos, timers, etc.) y software (threads, procesos, servers, drivers, etc.) que constituyen la plataforma física del sistema.
- Componentes lógicos (clases, métodos, procedimientos, etc.) y mecanismos de sincronización (mutex, semáforos, monitores, etc.).
- Escenarios de tiempo real que definen las situaciones de análisis y que se formulan como conjuntos de transacciones concurrentes, de fuentes externas de eventos que las invocan y de las restricciones temporales que se requieren en ellas.

Actualmente en el entorno MAST se han desarrollado los modelos relativos a sistemas monoprocesadores y distribuidos basados en prioridades fijas y con diferentes estrategias de planificación (expulsoras y no expulsoras, servidores esporádicos, polling, etc.) Las herramientas actualmente incorporadas permiten el análisis de sistemas de tiempo real basados en sistemas operativos y lenguajes comerciales de uso frecuente tales como POSIX y Ada.

Las herramientas que actualmente ya han sido desarrolladas (✓) o que están en fase de desarrollo (•), son:

✓ Clasic Rate Monotonic Analysis
✓ Varyng Priorities Analysis
✓ Multiple Event Analysis
✓ Monoprocessor Priority Assignment
✓ Linear HOPA
✓ Multiple Event HOPA
✓ Linear Simulated Annealing Priority Assignement
• Multiple Event Simulated Annealing Priority Assignmet
• Monoprocessor Simulation
• Utilization Test
• Check Shared Resources Total Ordering
• Multiple Event Example Generation

La Mast Real Time View puede utilizarse para representar el comportamiento de tiempo real en cada fase del ciclo de desarrollo del sistema. Puede operara en las fases iniciales con modelos poco detallados basados en estimaciones y de igual modo puede describir el comportamiento detallado del sistema en las fases finales en las que el código ya ha sido generado para validar sus prestaciones.

En la Mast RT view se ha establecido una modularidad paralela a la modularidad de la vista lógica. Se dispone tanto de un modelo de tiempo real del sistema completo, como el modelo de cada clase lógica, o incluso el modelo de cada método de su interfaz. Con ello se consigue que los modelos de tiempo real sean reusables y que puedan constituir parte de la especificación de componentes de tiempo real.

La Mast RT View pone a disposición del diseñador de sistemas de tiempo real muchas herramientas sofisticadas para el análisis cualitativo y cuantitativo de su diseños. Para hacer uso de estas herramientas, el diseñador solo tiene que saber modelar su sistema mediante los componentes que se definen en el metamodelo, y no necesita conocer los detalles de los algoritmos en que se basan las herramientas.

La Mast RT view está formulada en UML y definida formalmente mediante el metamodelo que se describe en este documento. La Mast RT view se puede incorpora a una herramienta CASE basada en UML, añadiendo un framework específico y adaptando a ella el compilador que genera el modelo Mast-File a partir de la bases de datos de la herramienta CASE. Actualmente se ha implementado el framework y el compilador para la herramienta ROSE-2000 de Rational.

En este documento se describe:

1) La estructura y los componentes de la Mast UML Real Time View.
2) El metamodelo UML que define formalmente la vista de tiempo real.
3) Los criterios adoptados para formular la vista de tiempo real sobre la herramienta ROSE'2000 de Rational.

II. Vista UML de tiempo real.

La vista UML de tiempo real es una vista adicional que complementa la descripción estándar UML de un sistema durante su fase de diseño, su finalidad es definir un modelo de tiempo real que describa la temporización y los recursos que se asignan a las actividades del sistema y que sirve de base para analizar su planificabilidad mediante herramientas automáticas. La vista de tiempo real se incorpora al modelo UML en la fase de diseño del sistema, después de que se hayan definido los componentes lógicos, se hayan organizado las actividades en threads y se haya asignado su ejecución a los procesadores.

Los componentes que constituyen la vista UML de tiempo real, son objetos, clases, enlaces y diagramas de actividad que en su conjunto representan el comportamiento dinámico de tiempo real de los componentes hardware y software del sistema que se modela. Los tipos de componentes que constituyen la vista de tiempo real y las relaciones que se pueden establecer entre ellos, se definen a través del metamodelo "MAST_UML", que se describe formalmente en UML en un documento adjunto a este.

La vista UML de tiempo real se compone de tres subvistas complementarias, cada una de ellas describe un aspecto específico del modelo de tiempo real.

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: procesadores, threads, coprocesadores, equipos hardware específicos, redes de comunicación, etc. que tienen en común ser los agentes que ejecutan las actividades del sistema. El modelo se compone de un conjunto de objetos enlazados entre sí, que en función de las clases especializadas del metamodelo Mast_UML de las que son instancias, de los valores que

se asignan a sus atributos y de los enlaces que se establecen entre ellos, describen la capacidad de procesado de la plataforma.

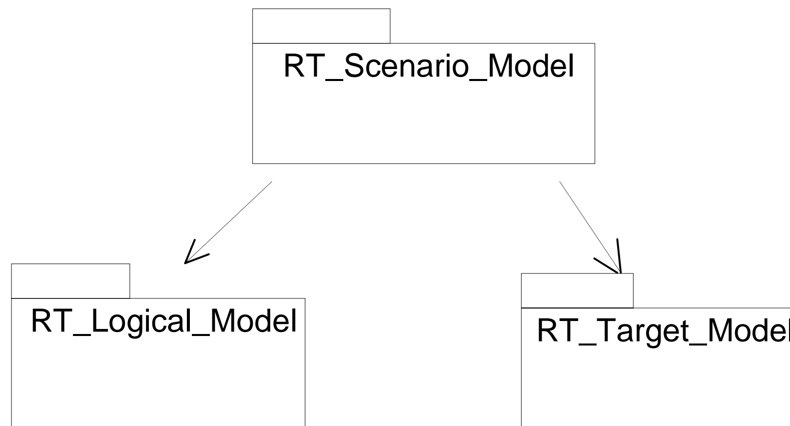

Figure 2: Mast_RT_View modules

Modelo de los componentes lógicos: Modelan los requerimientos de procesado que exige la ejecución de las operaciones funcionales definidas en los componentes lógicos que se utilizan en el diseño, tales como, métodos, procedimientos y funciones definidos en las clases, primitivas de sincronización de threads, procesos de comunicación por las redes, operaciones que realizan los dispositivos hardware, etc. En este modelo se declaran los recursos que cada operación necesita para poder llevarse a cabo, en especial aquellos que por ser requeridos por varias operaciones con régimen de exclusión mútua, pueden ser origen de retrasos en la ejecución de las operaciones. El modelo de tiempo real de los componentes lógicos ha sido definido con una modularidad paralela a la que existe en el diseño lógico del sistema. Los componentes de la vista de tiempo real que caracterizan el comportamiento de los métodos de una clase constituyen un módulo asociado al modelo lógico de la clase.

Escenarios de tiempo real: Modelan las diferentes configuraciones hardware/software que puede alcanzar el sistema y en las que estén establecidos requerimientos de tiempo real. Cada escenario se modela como un conjunto de transacciones que describen las secuencias de eventos y actividades que deben ser analizados en cuanto a que se satisfagan los requerimientos de tiempo real establecidos 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 autónomos (procedentes del entorno exterior al sistema, de timers, de relojes, de dispositivos hardware integrados, etc.) y de los requerimientos temporales que se definen en ellas para especificar la evolución temporal que se requiere. El análisis de tiempo real se realiza en base a las transacciones, pero para que una transacción de tiempo real pueda ser analizada, se necesita tener definidas todas las transacciones del escenario al que pertenece, o lo que es lo mismo, todas las transacciones de tiempo real que puedan ejecutarse en concurrencia con ella. El conjunto de transacciones de un escenario constituye la carga del sistema en esa configuración y afecta al análisis de cada una de ellas.

II.1 Modelo de tiempo real de la plataforma.

El modelo de la plataforma describe la capacidad de procesamiento de los recursos hardware y software que se declaran en el sistema y que ejecutan las actividades que constituyen el sistema que se modela.

El modelo de la plataforma describe los recursos hardware/software de procesamiento en los que se ejecuta la aplicación y así mismo describe la capacidad de procesamiento disponible para ello. La capacidad de procesamiento disponible para ejecutar las actividades del sistema resulta de la diferencia entre la capacidad de procesamiento que proporcionan los procesadores, redes de comunicación y equipos hardware en las que se ejecutan las actividades y la capacidad de procesamiento consumida por las tareas de background que requieren los recursos software (thread, planificadores, drivers, etc.) que gestionan y planifican la ejecución de las actividades de la aplicación.

Los componentes básicos que constituyen el modelo de la plataforma son :

- Scheduling_Server que modelan los procesos y las políticas de planificación de las actividades que se le asignan.
- Processing_Resource que modela los componentes hardware y el software de background (sistema operativo, drivers, etc.) en los que se ejecuta las actividades del sistema.

Scheduling Servers

La capacidad de operar en tiempo real de un sistema resulta de asignar las actividades que deben llevarse a cabo a threads concurrentes que se priorizan de acuerdo con los requerimientos de tiempo real que deben cumplirse. Los dos aspectos específicos que se abordan en el diseño de tiempo real de un sistema son la definición del nivel de concurrencia que se necesita para planificar las actividades del sistema y el establecimiento de los modos y parámetros de planificación en la ejecución de las actividades dentro de los thread. Por ello, el componente básico del modelo de la plataforma es el "**Scheduling Server**" que modela los threads y las políticas de planificación que se le asigna.

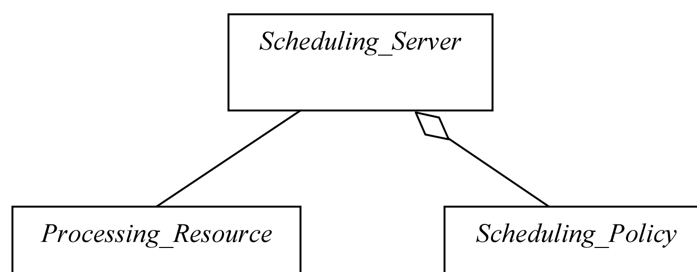

Figura 3: Schedulig_Server Class

Con un objeto Scheduling_Server se describen la capacidad de procesamiento disponible para ejecutar las actividades asignadas a un thread y la política de planificación que se utiliza para ejecutas estas actividades. Las dos informaciones que proporciona un objeto Scheduling_Server, son:

- El **Processing_Resource** en el que se ejecuta, que proporciona información sobre la capacidad de procesamiento de que se dispone para la ejecución de la actividades que se ejecutan en el Scheduling_Server. La capacidad de procesamiento resulta de la potencia del propio Processing_Resource y de que otros Scheduling_Servers están también utilizando la capacidad de procesamiento del mismo recurso.

- La política de planificación (**Scheduling Policy**) que define el criterio de ejecución de las diferentes actividades que estando asignadas a él se encuentran en estado de poder ser ejecutadas.

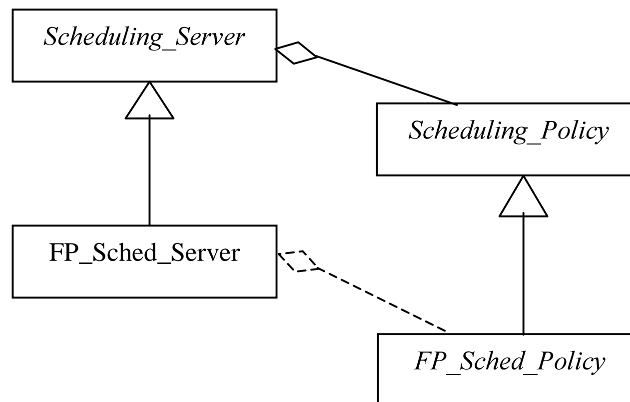

Figure 4: FP_Sched_Server Class

La clase Scheduling_Server es definida como clase abstracta a fin de dejar abiertas futuras extensiones del modelo de tiempo real hacia nuevos tipos de proceso. En la versión actual se define una única clase concreta denominada **FP_Sched_Server**, que modela un proceso simple (con un solo thread) con capacidad de ejecutar secuencialmente las actividades que tiene asignadas, y que compite con otros Scheduling_Server que hacen uso del mismo Processing_Resource siguiendo la política de planificación basada en prioridades fijas que tiene asignada.

Cada Scheduling_Server tiene agregado un objeto del tipo **Scheduling_Policy**, que define la política de planificación que se utiliza para ejecutar las actividades asignadas al thread que modela. La clase Scheduling_Policy es abstracta y en la versión actual solo se considera la extensión de ella FP_Sched_Policy que describe políticas de planificación basada en prioridades fijas y que también se define abstracta. El atributo principal de la clase FP_Sched_Policy es la prioridad (The_Priority) que establece la prioridad con la que se planifica el thread dentro del Processing_Resource en que se ejecuta.

En el modelo de tiempo real, las prioridades son del tipo **Any_Priority** y representan niveles absolutos de prioridad definidos con ámbito global dentro del sistema. El atributo The_Priority en la clase FP_Sched_Policy es del tipo **Priority** que es un subtipo de Any_Priority con el rango de niveles reducido al definido en el Processing Resource en que se ejecuta el thread.

Se han definido las siguientes extensiones especializadas y concreta de la clase FP_Sched_Policy:

- **Non_Preemptible_FP_Policy:** El thread se planifica con una política de prioridad fija y no expulsora.

- **Fixed_Priority_Policy:** El thread se planifica con una política de prioridad fija y expulsora.

- **Polling_Policy:** El thread se planifica con una política de prioridades fijas y basándose en un escrutinio periódico del evento que lo activa. El periodo de escrutinio se define mediante el atributo Polling_Period y el overhead que supone para el Processing Resource la actividad de escrutinio de la condición de planificación (que se ejecuta periódicamente con independencia de que se realice o no la planificación) se caracteriza por los parámetros Polling_Worst_Overhead, Polling_Avg_Overhead y Polling_Best_ Overhead.

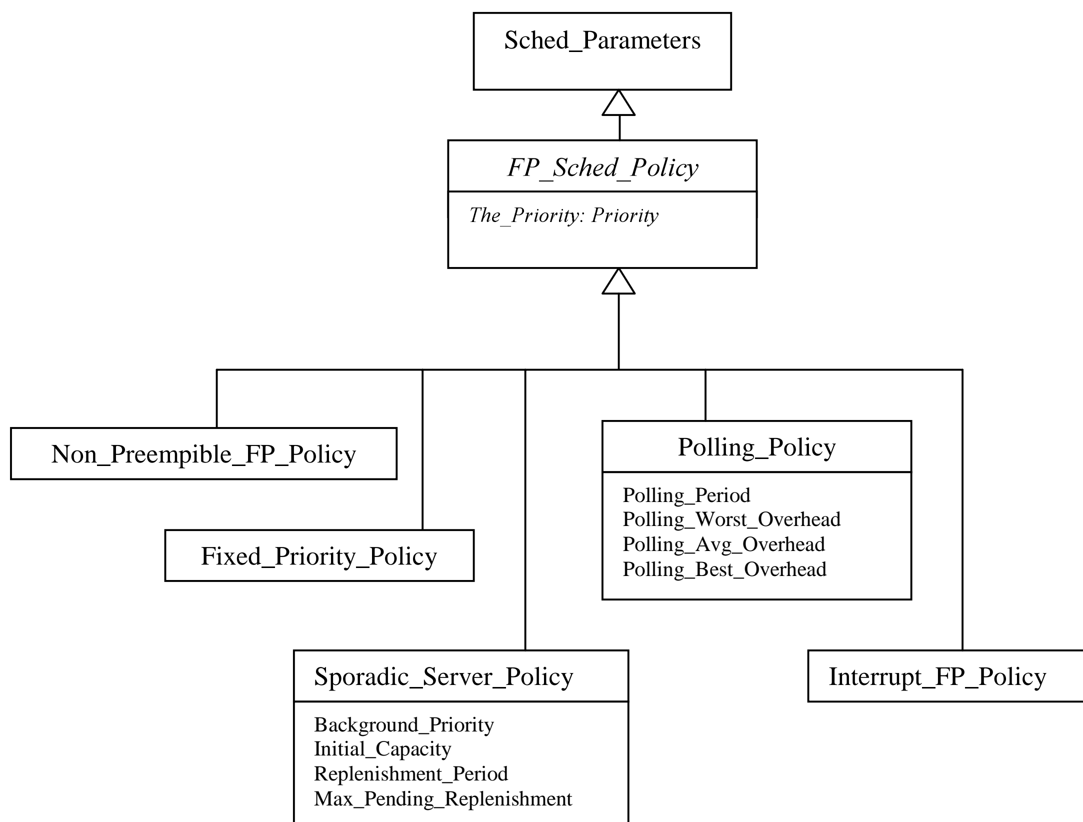

Figure 5: FP_Sched_Policy Class

- **Sporadic_Server_Policy:** El thread se ejecuta bajo el control de un planificador esporádico, lo cual es útil cuando en un sistema de tiempo real se necesita planificar actividades que son activadas con un patrón de tiempos no acotado. Los atributos que caracterizan este planificador son:

    - **The_priority**: representa en este caso la prioridad normal (alta) con la que se planifican las actividades cuando el servidor aún dispone de parte de la capacidad de ejecución que tiene asignada.
    - **Background_Priority** es la prioridad de background (baja) con la que se ejecutan las actividades cuando el servidor tiene consumida la capacidad de ejecución inicialmente asignada.
    - **Initial_Capacity:** representa el tiempo de ejecución que tiene asignado para ejecutar a alta prioridad las actividades que planifica.
    - **Replenishment_Period**: representa el intervalo de tiempo tras el que se restituye de nuevo la Initial_Capacity al servidor.
    - **Max_Pending_Replenishment**: Número máximo de actividades pendientes de ser planificadas que admite el servidor esporádico.

- **Interrupt_FP_Policy:** El thread se planifica con una política expulsora y basada en prioridades fijas que se establecen a los niveles de prioridades de las rutinas de interrupción (subtype **Interrupt_Priority**).

Un objeto de la clase ***Processing_Resource*** modela una plataforma (procesador y sistema operativo, red de comunicaciones, equipo con hardware embarcado, etc.) que ejecuta alguna operación o actividad del sistema cuya duración es relevante en su respuesta de tiempo real.

El atributo mas relevante de cualquier Processing Resource es el **Speed_Factor**, que caracteriza la velocidad de procesamiento del componente. Los tiempos de ejecución de las operaciones se describen como tiempos normalizados (Normalized_Execution_Time), y el tiempo real que tarda una operación cuando se ejecuta por un Processing_Reource concreto, se obtiene dividiendo el tiempo normalizado de la operacion entre el Speed_Factor del procesador, esto es,

$$Real\ Execution Time = \frac{Normalized\ Execution Time}{Speed\ Factor}$$

La clase Processing Resource se define como abstracta y de ella se derivan dos clases especializadas la clase Processor que modela los componentes del sistema capaces de ejecutar código, y la clase Network que modela los sistemas de comunicación entre procesadores que transmiten mensajes entre ellos.

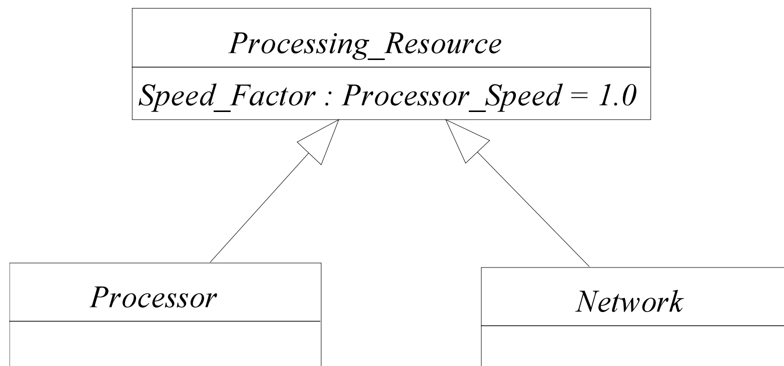

Figure 6: Porcessing_Resource Class

Un ***Processor*** es una clase especializada de Processing_Resource que modela un procesador que ejecuta segmentos de código de la aplicación. Se modelan con objetos de la clase Processor los coprocesadores, periféricos o equipos hardware específicos que llevan a cabo operaciones específicas del sistema que le han sido asignadas y que ejecutan en concurrencia física con las operaciones que llevan a cabo otros Processor. Processor es una clase abstracta introducida como concepto y que se utiliza como base a posibles clases especializadas futuras.

Un objeto Processor distribuye su capacidad de cómputo entre la ejecución de código de las operaciones explícitas del sistema que se le asignan y la ejecución de las tareas de gestión y cambios de contexto que son realizadas en background y que se modelan como diversos tipos de tiempo de overhead.

El **Fixed_Priority_Processor** es la clase especializada de Processor que se tiene modelada en las herramientas de análisis actuales. Representa un procesador junto con su sistema operativo con capacidad de ejecutar en multiproceso las operaciones pendientes en los diferentes Scheduling_Server que lo utilizan como plataforma. El Fixed_Priority_Processor ejecuta las operaciones pendientes utilizando diferentes políticas de planificación, pero todas ellas basadas en prioridades estáticas fijas compatibles con el FP_Sched_Server.

Los parámetros específicos que caracterizan el comportamiento de un objeto Fixed_Priority_Processor son:

- **Min_Priority**, **Max_Priority**, **Min_Interrupt_Priority** y **Max_Interrupt_ Priority**: En el entorno Mast todas la prioridades corresponden a un único tipo **Any_Priority** derivado del tipo Natural, y que tiene carácter absoluto en todo el sistema. Sin embargo, estos cuatro parámetros limitan los rangos de prioridades que pueden asignarse a los thread y operaciones que se ejecutan dentro del procesador. El rango de prioridades (subtype **Priority** range **Min_Priority** .. **Max_Priority)** delimita las prioridades asignables en ese procesador a los threads y operaciones de aplicación, y el rango de prioridades (subtype **Interrupt_Priority** range **Min_Interrupt_Priority** .. **Max_Interrupt_Priority**) delimita las prioridades asignables a las rutinas de atención a interrupciones hardware. Ambos rangos pueden estar solapados o no.

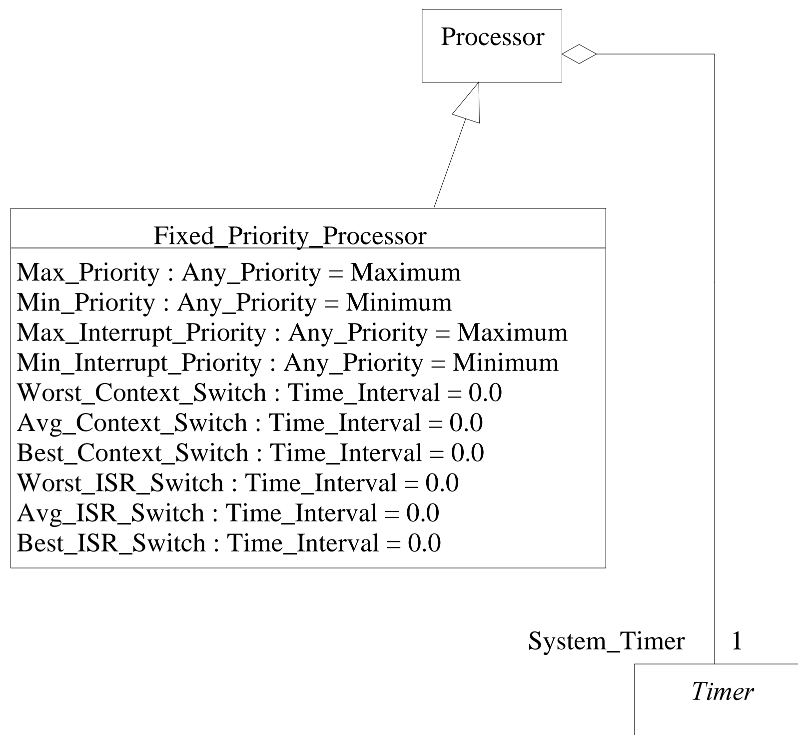

Figure 7: Processor Class

- **Worst_Context_Switch, Avg_Context_Switch** y **Best_Context_Switch**: Caracterizan el tiempo que el procesador necesita para realizar un cambio de contexto entre dos threads. El valor de peor caso es el parámetro básico para los análisis de planificabilidad de tiempo real, el valor de mejor caso se usa en estrategias mas refinadas y para el cálculo de jitters, y el valor promedio se utiliza en los simuladores. Los tres parámetros son del tipo **Time_Interval** que representa intervalos temporales expresados en segundos.

- **Worst_ISR_Switch, Avg_ISR_Switch** y **Best_ISR_Switch**: caracteriza el tiempo que el procesador necesita para realizar un cambio de contexto a una rutina de atención a una interrupción hardware. El caso mejor, promedio y peor juegan el papel ya descrito.

Cada procesador tiene agregado un mecanismo de temporización (**System_Timer**), que se modela a efecto de tiempo real mediante un objeto del tipo Timer.

Los objetos de la clase Timer describen la influencia que introduce el dispositivo hardware que tienen los procesadores para temporizar los retrasos o esperar a los plazos temporales. Estos dispositivos requieren del procesador al que sirven la ejecución de ciertas tareas de background que se ejecutan con prioridad de interrupción y por tanto, afectan a la ejecución de las operaciones del sistema. Los objetos de la clase Timer modelan dos aspectos relevantes del comportamiento de tiempo real: los tiempos de overhead que se producen en el procesador por

la atención del dispositivo hardware en background y la granularidad que producen en la medida del tiempo y que se traduce en desplazamientos o jitters de la ejecución de las operaciones.

La clase Timer es abstracta y en la versión actual se definen dos especializaciones concretas:

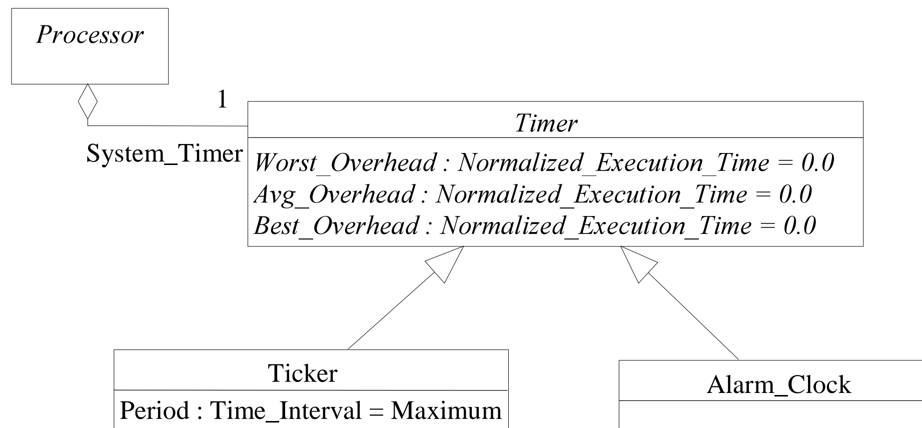

Figure 8: Timer Class

- La clase **Ticker** representa un temporizador basado en interrupciones hardware periódicas. En cada interrupción del temporizador, el procesador evalúa si se ha alcanzado alguno de los plazos de temporización pendientes, y en caso de que se haya alcanzado, gestiona su atención. Los parámetros que lo caracterizan son: los overhead (**Worst_Overhead**, **Avg_Overhead** y **Best_Overhead**) que requiere del procesador la gestión del timer en cada interrupción períodica, y el período (**Period**) de interrupción que define la granularidad de medida del tiempo.

- La clase **Alarm_Clock** representa un temporizador basado en un hardware programable que interrumpe cuando se alcanza el plazo programado. En este caso la granularidad de temporización que se alcanza es despreciable y los overhead que introduce en el procesador para su gestión (**Worst_Overhead**, **Avg_Overhead** y **Best_Overhead**) se introducen sólo al alcanzar cada plazo.

Un ***Network*** es un Processing_Resource especializado que modela la ejecución de las operaciones de transferencia de mensajes entre procesadores a través de un mecanismo de comunicación existente en la plataforma. Los objetos de la clase Network modelan el tiempo finito (función de su anchura de banda) que requiere transferir un mensaje entre dos procesadores y la contención que produce en los procesos que realizan la transferencia el hecho de que la transferencia de los mensajes se realiza en régimen mutuamente exclusivo.

Un Driver modela el costo de procesamiento que requiere a un Processor la transferencia de un mensaje a través de un Network. Así pues el driver modela el tiempo de overhead que supone para el procesador que emite o recibe mensajes a traves de un network, la ejecución de los procesos de background que supervisan y gestionan el acceso al network.

Cada Network dispone de una lista de Drivers (List_of_Drivers) agregados. Cada procesador que utilice un network deberá tener asociado un objeto que corresponda a una instancia de un Scheduling_Server que esté a su vez asociado a un driver de la lista de drivers del network.

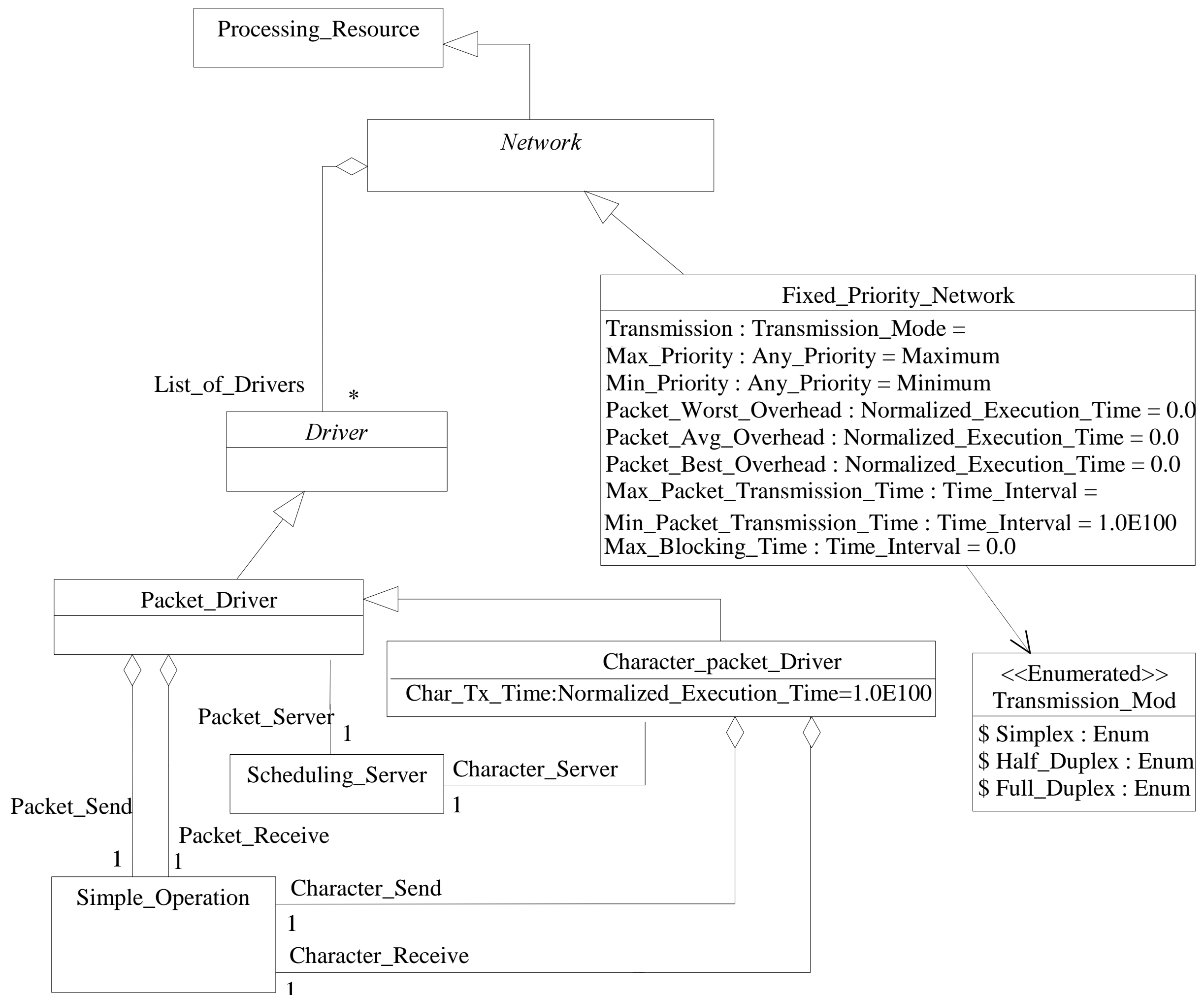

Figure 9: Network class

La clase Network es declarada abstracta. En la versión actual, se define la clase concreta **Fixed_Priority_Network**, que modela un canal de comunicación que transfiere los mensajes descomponiéndolos en unidades (Packet). La transmisión de un packet es una operación que es planificada de acuerdo con una política no expulsable y basada en prioridades estáticas asociadas a los mensajes. Con esta clase especializada de Network se pueden modelar muchos tipos de comunicación utilizables en sistemas de tiempo real, tales como bus CAN, bus VME, comunicación punto a punto por canal serie o comunicación punto a punto por Ethernet.

Los atributos que caracterizan el comportamientos de un Fixed_Priority_Network son los siguientes:

- **Transmission**: es un atributo del tipo enumerado Transmission_Mode y que caracteriza los modos de operación (Simplex, Half_Duplex, Full_Duplex) que admite el canal de comunicación.

- **Max_Priority** y **Min_Priority**: Definen el rango de prioridades dentro del tipo Any_Priority que pueden asignarse a los mensajes que se transfieren por el Network.

- **Packet_Worst_Overhead**, **Packet_Avg_Overhead** y **Packet_Best_Overhead**: cuantifican el tiempo de ocupación del canal de comunicación que requiere el protocolo de gestión de la transferencia del próximo paquete, teniendo en cuenta que debe transferirse el de mayor prioridad de entre los que están pendiente de transmisión en cualquiera de los puerto de la red. Estos tiempos se expresan en unidades normalizadas y requieren ser divididos por el Speed_Factor del network para convertirlos a unidades de tiempo físico.

- **Max_Packet_Transmission_Time** y **Min_Packet_Transmission_Time**: Representan el rango de valores para el intervalo de tiempo durante el que el network queda ocupado (sin posibilidad de liberación) cuando se transmite un paquete entre dos puertos cualquiera del network. Estos tiempos se formulan en unidades de tiempo físico.

Como se apuntaba anteriormente, un ***driver*** modela el overhead que supone para un procesador usualmente mediante procesos en background la gestión de la transferencia de unidades de información por el canal. El driver modela la carga que suponen para el procesador las rutinas conducidas por eventos que realizan operaciones tales como spooling de los mensajes pendientes, planificación de los mensajes, etc. Como ocurre en otros casos la clase driver es abstracta a fin de facilitar su posible extensión mediante clases especialzadas.

El **Packet_Driver** es una clase concreta que modela un driver basado en un único Scheduling_Server (**Packet_Server**) que corresponde al thread de background que es planificado dentro del Processor donde está instalado el driver. Por cada paquete que recibe o que envía, ejecuta las correspondiente operación (**Packet_Send** o **Packet_Receive**) que tiene asociadas. Este driver es válido cuando la comunicación por el puerto está soportada por algún tipo de hardware que controla de forma autónoma la transmisión o recepción del paquete por el network. El procesador interviene sólo una vez por paquete, bien al inicio del envío del paquete o al concluir la recepción del mismo.

El **Character_Packet_Driver** es una segunda clase concreta que constituye una especialización del Packet_Driver, con ella se modela el caso en que se requiere del procesador que además de intervenir en el inicio del envío o al concluir la recepción de cada paquete, intervenga también para el envío o la recepción de cada carácter del paquete. Este driver modela casos como el de la comunicación punto a punto por un canal serie a través de una UART, en el que cada transmisión o recepción de un carácter provoca una interrupción que requiere del procesador que gestione los registros de la UART. El Character_Packet_Driver tiene asignado un segundo Scheduling_Server (**Char_Server**) (que puede ser el mismo o diferente del que gestiona la

transferencia de los packets) dentro del que se ejecutan las operaciones (**Char_Send** o **Char_Receive**) de gestión de cada carácter.

II.1.2 Modelo de la plataforma en la herramienta ROSE'2000.

Conceptualmente el modelo de la plataforma está constituido por un conjunto de clases u objetos enlazados entre sí, que son instancias de las clases definidas en el metamodelo Mast_UML. Aunque los diagramas de objetos están especificados en UML, no son soportados de forma completa por la herramienta ROSE'2000. En particular, esta herramienta no admite introducir los valores específicos de los atributos de cada objeto en los diagramas de objetos.

Por ello se establece que cuando se utilice la herramienta ROSE'2000 los diagramas de objetos se formulan en diagramas de clases y los objetos se representan a través de símbolos de clase. Asignando a cada objeto del modelo como estereotipo el nombre de la clase del metamodelo UML_Mast de la que es una instancia, se describe su pertenencia al modelo y se señala justamente la clase que este instancia.

A efecto de su tratamiento posterior por las herramientas automáticas, el modelo de la plataforma se ubica en el package "Logical_View.Mast_RT_View.RT_Target_Model".

A continuación se enumeran algunas normas para la formulación de los diagramas de objetos :

1) Es ignorado cualquier componente es decir cualquier clase estereotipada que aparezca en estos diagramas y no corresponda a instanciaciones del metamodelo UML_MAST.

2) Los roles en cada asociación sólo necesitan ser incluidos cuando la clase (es decir el estereotipo) del componente enlazado no es suficiente para identificar el tipo de enlace. Cuando esto no ocurra pueden visualizarse a efectos de documentar el diagrama pero en general no son requeridos ni analizados.

3) Es optativo introducir el tipo de los atributos pero si se introducen estos deben ser correctos.

4) A efecto de simplificar la representación gráfica del modelo, se consideran ciertos valores o componentes por defecto. Estos son:

   - Todos los atributos de las clases del metamodelo tienen establecidos valores por defecto. Si a un atributo de un objeto del modelo no se le asigna explicitamente un valor, se le asigna su valor por defecto.
   - Si un Scheduling_Server no tiene enlazado un Processor se supone por defecto que tiene asociado un nuevo "Fixed_Priority_Processor" con los valores por defecto en los atributos.
   - Si un Scheduling_Server no tiene enlazado un Scheduling_Policy, se supone por defecto que tiene asociado un "Non_Preemptible_FP_Policy" con los valores por defecto de los atributos.
   - Si un Fixed_Priority_Processor no tiene enlazado un Timer, se supone por defecto que tiene asociado un "Alarm_Clock" con los valores por defecto de los atributos.

- Si un Packet_Driver no tiene enlazada una Simple_Operation con role Packet_Send, o con role Packet_Receive, se supone por defecto que tiene enlazada con tal role una operación Null_Operation (operación con tiempo de ejecución nulo).
- Si un Character_Packet_Driver no tiene enlazada una Simple_Operation con role Char_Send, o con role Char_Receive, se supone por defecto que tiene enlazada con tal role una operación Null_Operation.

5) Son situaciones de error :

- Un Scheduler_Server enlazado con más de un processor.
- Un Scheduler_server enlazado con más de un Scheduling_policy.
- Un Fixed_Priority_Processor enlazado con más de un Timer.
- Un Packet_driver sin un Scheduling _Server enlazado o enlazado con más de uno.
- Un Character_packet_driver sin un Scheduling _Server enlazado con el role Char_Server o con más de uno.
- Un Packet_Driver con más de una Simple_Operation con un mismo role Packet_Send, o con Packet_Receive.
- Un Character_Packet_Driver con más de una Simple_Operation con el mismo role Char_Send o Char_Receive.

6) Son Situaciones de warning:
- Un Fixed_Priority_Network sin ningún driver enlazado.
- Un Processor sin ningún Scheduling _Server enlazado.
- Una prioridad de Non_Preemptible_FP_Policy, Fixed_Priority_Policy, Polling_Policy o Sporadic_Server_Policy con valor fuera del rango Priority del Processor al que está enlazado a través del Scheduling_Server.
- Una prioridad de Interrupt_FP_Policy con valor fuera del rango Interrupt_Priority del Processor al que está enlazado a través del Scheduling_Server.

II.2.3 Ejemplo de formulación UML de Modelo de la Plataforma.

En los siguientes dos diagramas de objetos UML se muestra un ejemplo de la declaración de un modelo de plataforma. Sobre este modelo se pueden hacer los siguientes comentarios:

- *Se declaran cuatro Processor: Processor_1, Processor_2, Image_Processor y Anonymous_Processor_1. Los tres primeros se declaran explicitamente, el cuarto resulta de asociación por defecto de un Scheduling_Server sin Processor enlazado.*

- *Processor_1 tiene todos los valores de sus atributos declarados explícitamente. Processor_2 tiene algunos valores de los atributos declarados explícitamente y otros por defectos. Image_Processor y Anonymous_Processor_1 tienen todos los valores de sus atributos definidos por defecto, esto es razonable en procesadores monoproceso de sistemas embarcados, en los que los valores de los atributos son irrelevantes y el procesador solo introduce concurrencia física.*

- *Processor_1 y Processor_2 tienen declarados explícitamente un mismo tipo de System_Timer. Image_Processor y Anonymous_Processor_1, no tienen declarado System_Timer, por lo que le queda asignado el Timer definido por defecto. De nuevo a estos tipos de procesadores que operan en monoproceso, no les afecta el overhead introducido por el Timer.*

- *Processor_1 sirve de soporte a dos FP_Sched_Server: Task_1 y Task_2 . Ambos operan bajo una politica de planificación de tipo expulsora. Las actividades de Task_1 se ejecutan por defecto con prioridad mas alta 16 y las actividades de Task_2 se ejecutan por defecto con prioridad mas baja 10. Los restantes tres procesadores tienen un único FP_Sched_Server. Su prioridad, es irrelevante a efecto de la planificación de actividades en el procesador, pero la prioridad es relevante si esas actividades ejecutaran operaciones que hacen uso de Shared_Resources globales.*

- *Aunque no aparece en el diagrama, en el modelo del canal de comunicaciones, se declaran dos nuevos FP_Sched_Server que soportan las actividades de los driver de comunicación: Rutine_Com_Proc_1 soportado por Processor_1 y Rutine_Com_Proc_2 soportado por Processor_2. Ambos tienen declarada un Sched_Policy de la clase Interrupt_FP_Scheduler, lo que le posibilita operar a unas prioridades dentro del rango Interrupt_Priority mucho mas altas que las de aplicación.*

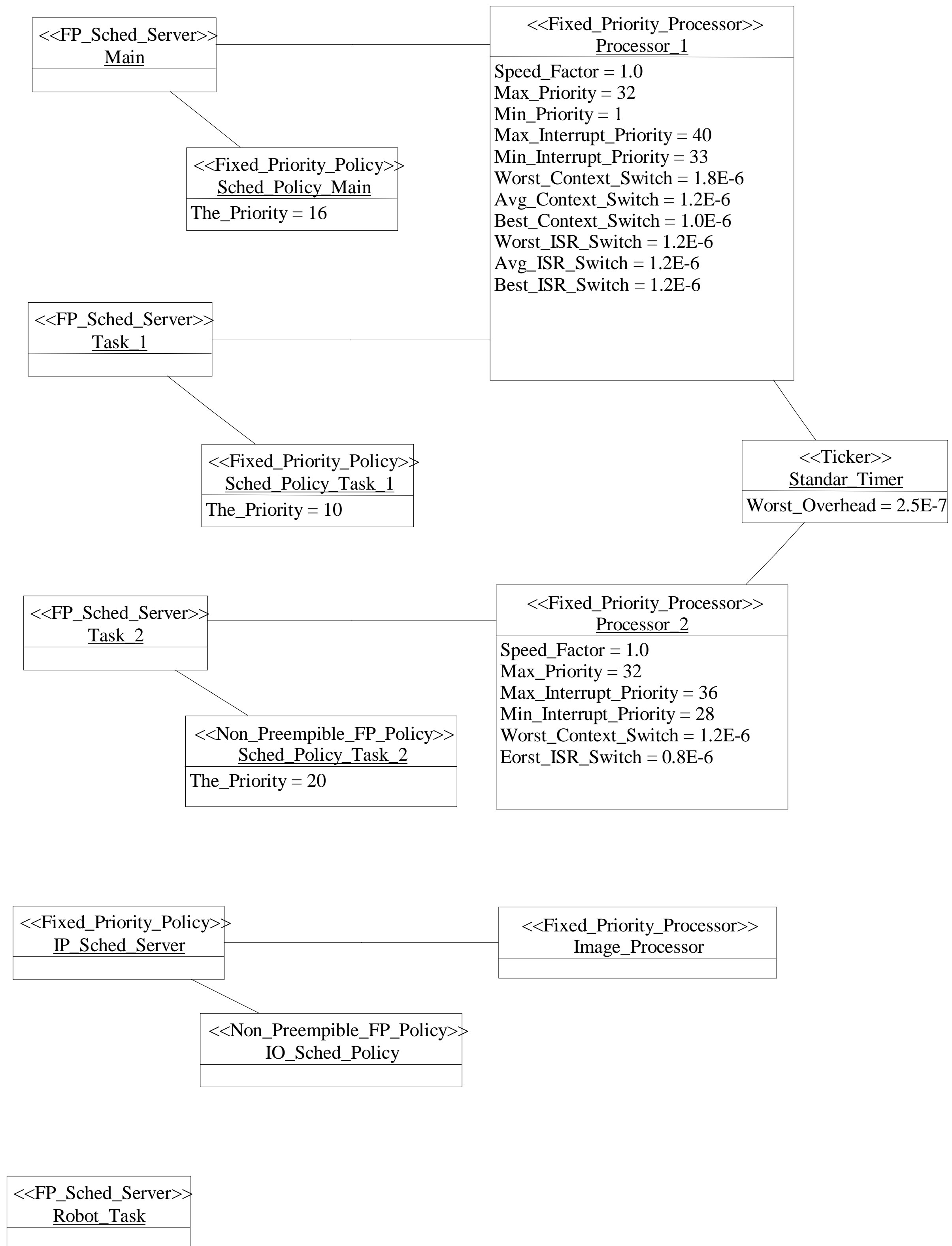

Figure 10: Example of Processing_Resources declaration

- *En el segundo diagrama se muestra el ejemplo de la declaración del network Bus de la clase Fixed_Priority_Network, con la capacidad de operar en Half_Duplex. Los procesadores pueden intercambiar bidireccionalmente mensajes, pero solo hay una transferencia en cada instante. En este caso los valores de los atributos se declaran explícitamente.*

- *Bus_Sched_Server es el Scheduling_Server asociado al network. En él se ejecutan todas las actividades de transferencia de paquetes que se realizan por él, con independencia del procesador desde el que se hacen la transferencia de mensaje del que proceden los paquetes.*

- *El network Bus tiene dos Drivers declarados para los procesadores Processor_1 y Processor_2. Esto sólo significa que la comunicación por el network Bus repercute un overhead de tiempo de ejecución sobre estos procesadores. Esta declaración no implica que los procesadores Embedded_Instrument y Anonymous_Processor_1, no comuniquen por el network Bus, sino que la sobrecarga debida a los procesos de gestión de la transferencia por el bus no es repercutida directamente por la herramienta, y debe ser modelads explícitamente mediante actividades.*

- *Obsérvese que aunque ambos drivers utilizan las mismas declaraciones de las operaciones para transferir y recibir paquetes, la ejecución de las mismas en los dos procesadores supone diferentes tiempo de ejecución, ya que los procesadores tienen diferente Speed_Factor.*

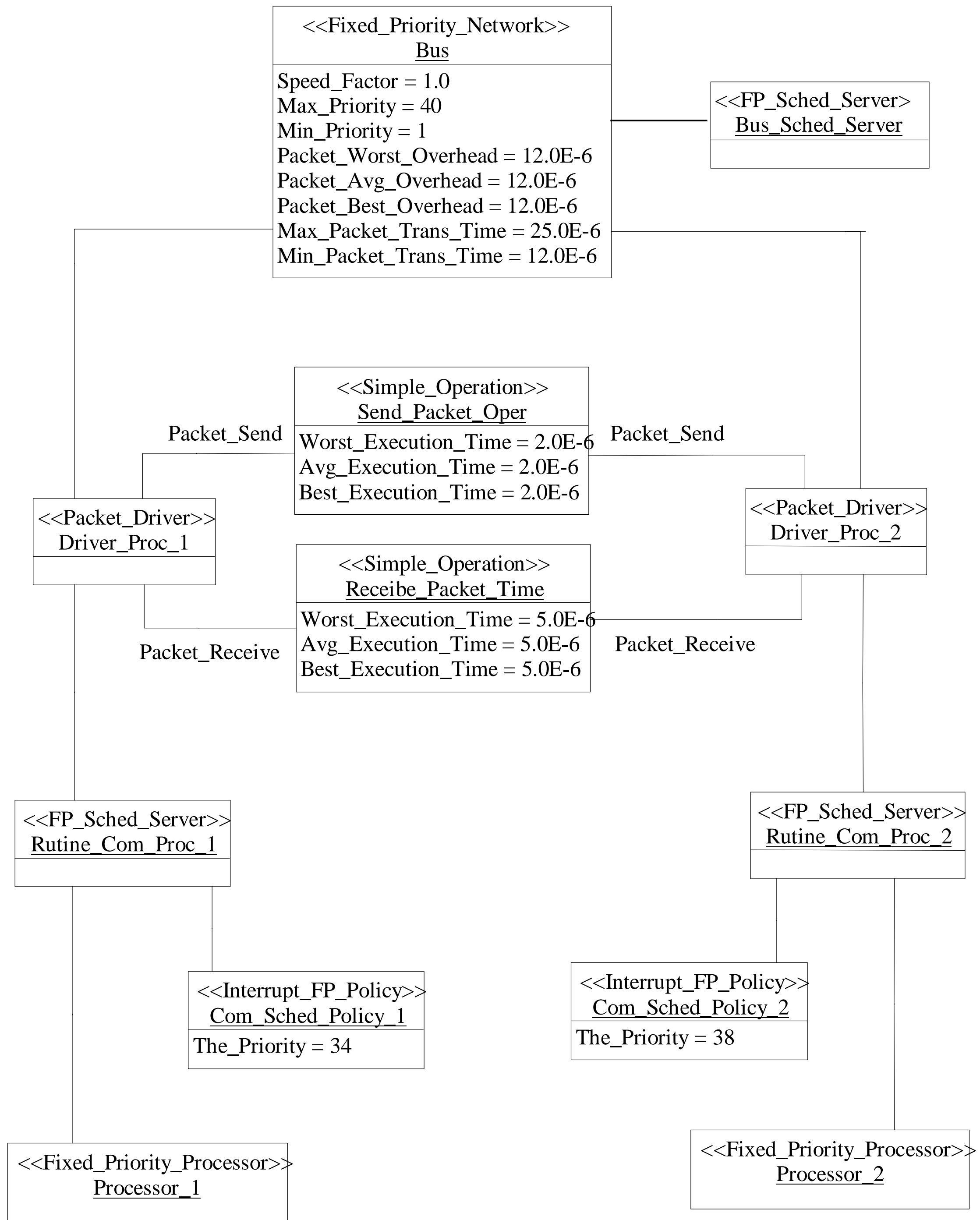

Figure 11:Example of Network declaration

II.2 Modelo de los componentes lógicos.

El modelo de los componentes lógicos describe el comportamiento de tiempo real 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 escenarios de tiempo real. El comportamiento temporal que se modela refleja por un lado, los tiempos de ejecución de las operaciones que son consecuencia de la complejidad de los algoritmos con que se construyen y por otro lado, los tiempos de bloqueo que pueden retrasar su ejecución y que se producen al tener que hacer uso en régimen exclusivo de recursos que también son compartidos por otros componentes lógicos.

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 que 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 bloqueos(Shared_Resource), y dejando al modelo de los escenarios la especificación 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.

Cuando un sistema se diseña siguiendo una metodología orientada a objetos, el esfuerzo de diseño se encauza hacia modularizar la arquitectura del sistema en función de componentes (clases y objetos) definidos mas por su propia consistencia conceptual, que por el papel que juegan en la funcionalidad global del sistema. Características fundamentales de las arquitecturas orientadas a objetos son:

- Los módulos estructurales básicos son las clases y objetos que se conciben y se describen estáticamente y dinámicamente por sí y con independencia del sistema del que forman parte.

- Los módulos se agrupan en estructuras jerarquizadas por relaciones de tipo cliente-servidor. La funcionalidad que ofrece una clase u objeto es consecuencia de la funcionalidad que ha recibido de otras clases u objetos de las que es cliente.

- La funcionalidad que ofrece un objeto es habitualmente utilizada dentro del sistema a través de diferentes vías y en diferentes instantes de tiempo. Si el objeto ha sido concebido para ser reusado, su funcionalidad también será utilizada en otros sistemas.

En la metodología de modelado de tiempo real que se presenta, se ha establecido como objetivo básico que el modelo sea modular y tenga capacidad de modelar cada componente lógico (procedimiento, clase, objeto) de forma individual y con independencia del entorno en el que se hace uso de él. Como esto no es directamente posible, ya que el comportamiento de tiempo real de un componente depende de que otros componentes se ejecutan en concurrencia con él, la

solución se ha encontrado a través de la parametrización. Se parametrizan los posibles elementos del entorno que influyen en el comportamiento de tiempo real del componente, y se formula el modelo en función de estos parámetros.

Con la metodología de modelado de los componentes lógicos que se propone, se consigue:

- Que la estructura del modelo de tiempo real se base en los modelos de tiempo real de las clases y objetos lógicos del sistema.

- El modelo de tiempo real de un objeto se pueda formular en función de los modelos de tiempo real de los objetos de los que es cliente.

- El modelo de tiempo real de un componente tiene una única descripción, aunque el mismo está parametrizado. En cada invocación del modelo del componente, se establecen valores concretos a los parámetros, y con ello se caracteriza el entorno en que se ejecuta (en que Scheduling_Server y en concurrencia con que otros componentes).

II.2.1 Componentes del modelo.

El modelo de los componentes lógicos está constituido por un conjunto de diagramas de clases y de objetos en los que se declaran:

- Los componentes que modelan el comportamiento de 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 soportan la posible interacción con otros componentes que se ejecutan en concurrencia.

Las clases y objetos que forman parte del modelo de tiempo real de los componentes lógicos del sistema son instancias de las clases del metamodelo UML_Mast. Las clases abstractas del metamodelo UML_Mast, de las que resulta el modelo de los componentes lógicos del sistema son:

Primitive_Operation: Modela conceptualmente la operación secuencial que puede realizar un thread simple. Desde que se inicia, hasta que concluye no requiere eventos y su ejecución solo es afectada externamente por las suspensiones que se producen mientras que no es planificada en su Processing_Resource o cuando requiere en régimen exclusivo un Shared_Resource que está ocupado.

Job: Modela la actividad concurrente que se desencadena cuando se ejecuta un componente lógico (procedimiento, función, método, etc.). La actividad requiere que intervengan múltiples threads y que incluso puede perdurar después de haber

concluido el componente que lo desencadenó.

Shared_Resource: Modela un recurso que es requerido en régimen de exclusión mutua por diferentes operaciones concurrentes, y es causa potencial de bloqueos al acceder a él .

Una ***Primitive_Operation*** representa las actividades básicas que se pueden ejecutar en un programa secuencial. La clase Primitive_Operation se define como una clase abstracta, y en el siguiente diagrama se muestran las tres clases concretas que se derivan de ella.

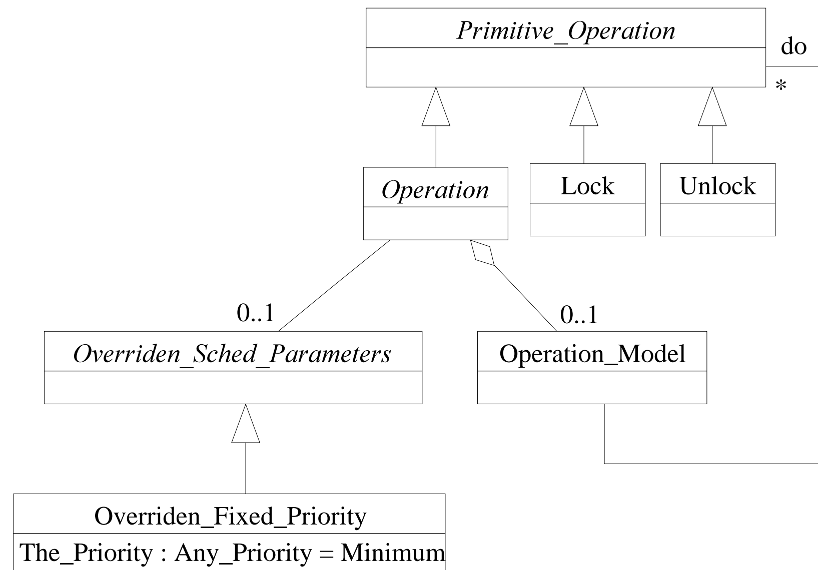

Figure 12: Primitive_Operation class.

La clase abstracta ***Operation*** modela la ejecución de un segmento de código que debe ser todo él ejecutado dentro del Scheduling_Server que se establece en la Activity que la invoca. La característica específica del componente operation es que una vez que ha sido activado, su ejecución solo depende de que sea planificado por el Scheduling_Server al que se ha asignado y de que pueda acceder a los Shared_Resources que se requieren para su ejecución.

Una Operation puede tener agregada una estructura del tipo **Overriden_Sched_Parameter**. Cuando esta existe, durante la ejecución de la operación deben establecerse los correspondientes parámetros de la Scheduling_Policy del Scheduling_Server a los valores fijado por la estructura agregada. La agregación de esta estructura es optativa, lo que significa que si no se agrega, la Scheduling_Policy que se utilizará durante la ejecución de la operación es la establecida en el Scheduling_Server.

En la versión actual se ha definido la clase concreta **Overriden_Fixed_Priority**, que permite modificar transitoriamente la prioridad durante la ejecución de la operación. Esta clase es compatible con las políticas de planificación derivadas de FP_Sched_Policy.

Una Operation se describe mediante un Operation_Model, el cual contiene la secuencia ordenada de Primitive_Operations que implica su ejecución. El Operation_Model de una Operation se describe mediante un diagrama de actividad UML, agregado a la operación y consistente en una única actividad UML con una sentencia do/ por cada Primitive_Operation. Alternativamente se puede emplear un diagrama de actividad con varias actividades enlazadas por transiciones, siempre y cuando conformen un único hilo de control de flujo y se indique el inicio y fin del mismo mediante los correspondientes Initail_State y Final_State. En esta notación se permite modelar operaciones empleando diagramas de actividad como descripción de actividades recursivamente.

En la siguiente figura se muestra un ejemplo de declaración de una Operation. Los shared_resource Resource_1 y Rresource_2 y las operations Analize y Store_Result, deben estar declaradas en otras partes del modelo.

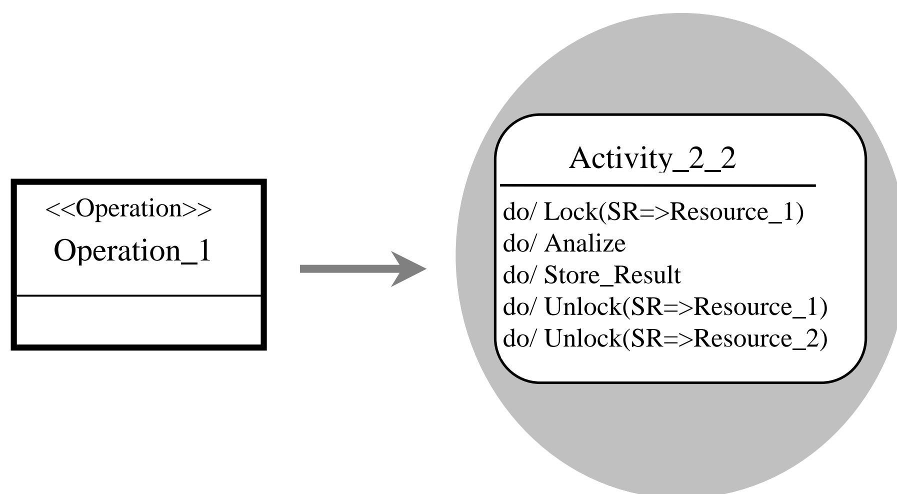

Figure 13: Declaration and definition of an operation.

De la clase Operation se han definido en la versión actual tres clases concretas: Simple_Operation, Composite_Operation y Enclosing_Operation.

La clase concreta **Simple_Operation** representa una operación secuencial uniforme que para ser ejecutada una vez que haya sido activada requiere inicialmente acceder en régimen de exclusión mútua a un conjunto de Shared_Resource y posteriormente ser planificada por el Scheduling_Server sobre el que se ejecute. Al concluir puede liberar un conjunto de Shared_Resources que disponga en régimen exclusivo.

El tiempo que requiere su ejecución se caracteriza por los parámetros WCET (Worst_Case_ Execution_Time), ACET (Avg_Case_Execution_Time) y BCET (Best_Case_Execution_Time) que son del tipo Normalized_Execution_Time. Una vez conocido el Processing_Resource en que se ejecuta, se traducen a tiempo físicos dividiendo por el correspondiente Speed_Factor.

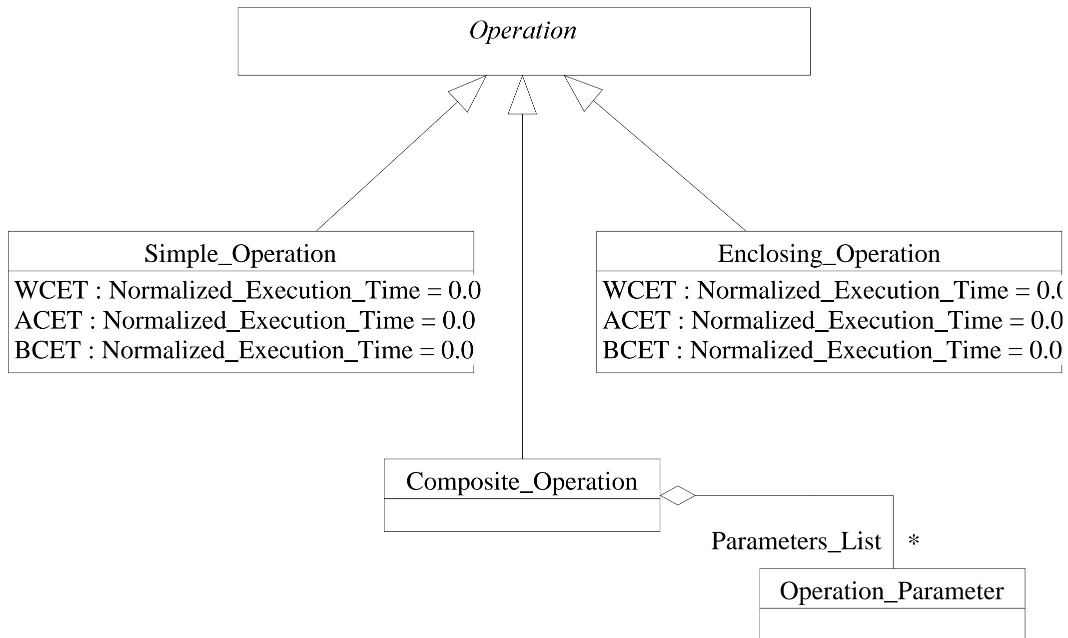

Figure 14: Operation class.

El modelo de una Simple_Operation es optativo, y si existe consiste en un diagrama de actividad agregado con una sola actividad UML que puede tener una sucesión de sentencias do/Lock que describe los recursos que requiere en régimen exclusivo en su inicio y una lista de sentencias do/Unlock que describe al conjunto de recurso que se liberan al acabar. Si se emplea la notación alternativa de múltiples actividades enlazadas las sentencias que pueden ir dentro de las actividades podrán ser sólo de los tipos do/Lock o do/Unlock.

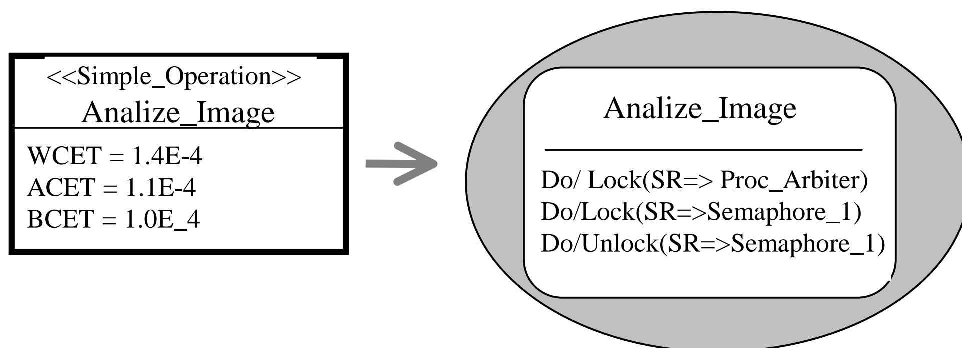

Figure 14: Declaration and definition of Simple Operation object.

La clase concreta **Composite_Operation** representa una operación compuesta por una secuencia de operaciones. Su función es servir de base para la estructuración de operaciones complejas.

Cada Composite_Operation tiene agregado un modelo que describe su estructura. El modelo de

una Composite_Operation está constituido por un diagrama de actividad agregado bien con una única actividad UML que contine una secuencia ordenada de sentencias do/ en las que se declaran las Primitive_Operation (Lock, Unlock y Operation) que la componen o bien mediante la notación alternativa descrita anteriormente.

Las operaciones primitivas se pueden combinar en cualquier orden. El orden de las operaciones en la lista de una Composite_Operation es relevantes a efecto de las herramientas de análisis de bloqueos inducidos por el acceso a los Shared_Resource. Las operaciones primitivas que se referencien en las sentencias do/ deben estar declaradas en alguna sección del modelo.

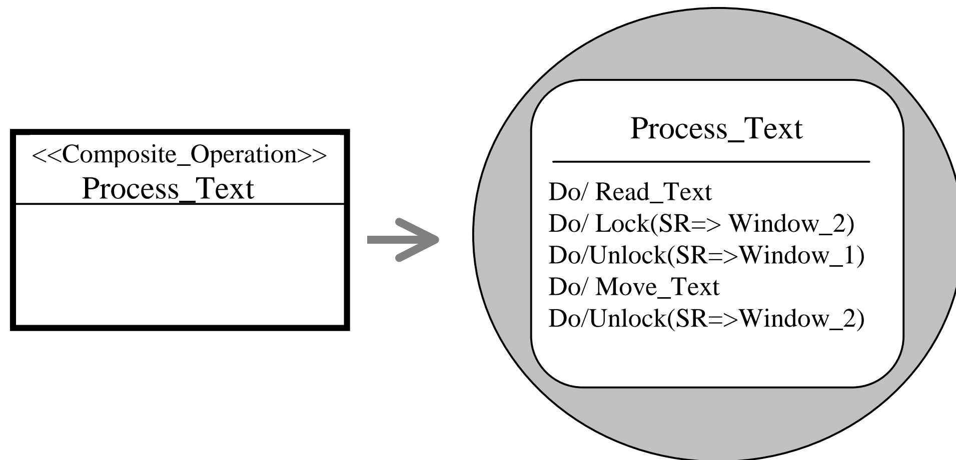

Figure 15: Declaration and definition of Composite Operation object.

Una Composite_Operation puede estar parametrizada, esto es, ciertos componentes que son parte de su descripción pueden ser representados simbólicamente mediante parámetros. Cuando la operación compuesta sea invocada en una actividad (de otras Operation, de un Job o de una Transaction) a los parámetros se le asignan valores concretos, que terminan por definir unívocamente la operación compuesta. En la figura 16 se muestra el metamodelo que describe estos parámetros.

El número de parámetros de una Composite_Operation es ilimitado. Cada parámetro tiene un identificador y un valor que está caracterizado por un tipo. Los valores de los parámetros de una Composite_Operation pueden ser de los tipos:

Type <u>Operation:</u> Representa simbólicamente el identificador de una operación. Dentro de la descripción de la Composite_Operation puede referenciar o bien la operación que se designa dentro de una sentencia do/ o a un parámetro de una operación compuesta declarada en una sentencia do/ que sea a su vez del tipo Operation. El parámetro solo representa el identificador de la operación y es responsabilidad del modelador que cuando existan parámetros asociados a la operación representada simbólicamente, estas sean consistentes con los parámetros que se le asocian.

Type <u>Shared_Resource</u>: Representa simbólicamente el identificador de Shared_Resource que va a ser utilizado como argumento de una operación Lock o Unlock. También puede aparecer como parámetro de una operación compuesta declarada en una sentencia do/ que a su vez sea del tipo Shared_Resource.

Type Overriden_Sched_Parameter: Todas las Composite_Operations disponen de un parámetro implicito (no requiere declaración) de este tipo y que responde al identificador “OSP”. Representa a un Overriden_Sched_Parameter que se asocia a la operación.

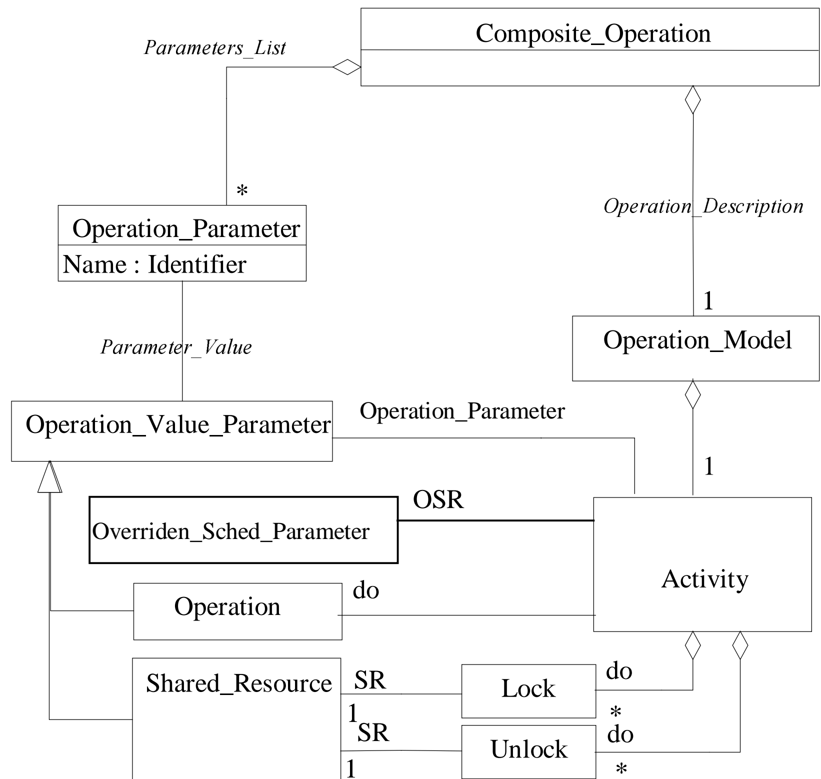

Figure 16: Composite Operation

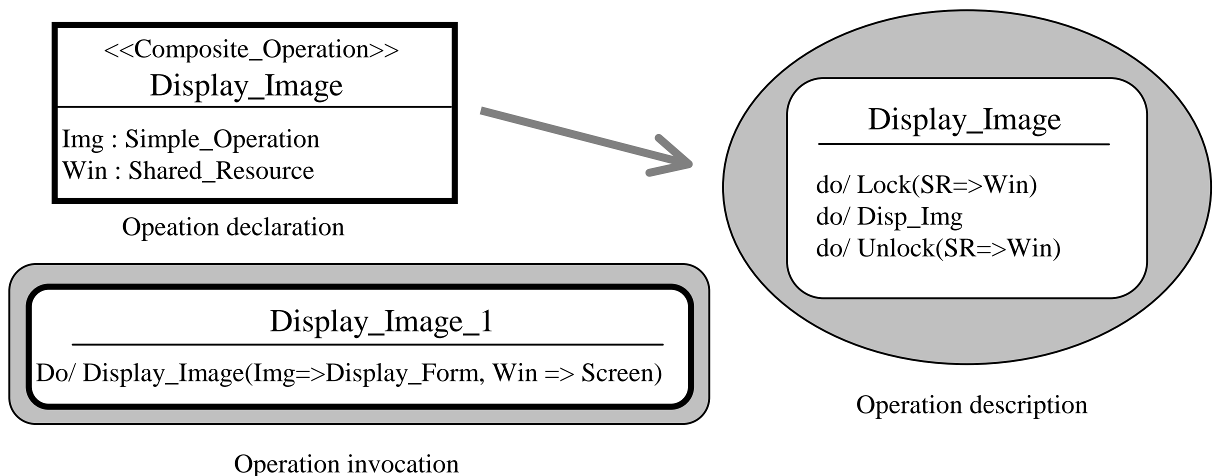

Figure 17: Declaration, definition and invocation of Composite_Operation class.

La clase concreta **Enclosing_Operation** representa una operación compuesta en la que solo se enumeran algunas de las operaciones que lleva a cabo. Se considera implícitamente que además de las especificadas en la lista (Composite_Operations_List), existen otras operaciones simples con las siguientes restricciones:

- Las operaciones no incluidas explícitamente no requieren ni liberan recursos.

- La suma de los tiempos de ejecución de todas las tareas que constituyen la Enclosing_Operation, supuesto que no existen bloqueos por no disponibilidad de recursos, son los que se definen como atributos de ella (WCET, ACET y BCET).

Su descripción es similar a la de una Composite_Operation, salvo que no admite parámetros.

La clase concreta **Lock** representa una operación cuya función es acceder en régimen exclusivo (Lock) al Shared_Resource que se pasa como argumento. Representa una operación de duración nula, pero que puede dar lugar a un retraso de su ejecución si el recurso al que se accede no está disponible (ha sido accedido por otro componente del sistema).

La clase concreta **Unlock** representa una operación cuya función es liberar (Unlock) el Shared_Resource que se pasa como argumento. Corresponde a una operación de duración nula que siempre se ejecuta de forma inmediata. Dentro de una transacción, la operación Unlock solo puede ejecutarse sobre un recurso sobre el que previamente se ha realizado la operación Lock.

Las operaciones primitivas Lock y Unlock solo son invocadas dentro del modelo de actividad que describe una operación, y para ello se utilizan actividades con sentencias del tipo:

do/ Lock(SR => Shared_Resource_Identifier).
do/ Unlock(SR => Shared_Resource_Identifier).

La clase abstracta ***Shared_Resource*** representa recursos que pueden ser accedidos por una operación en régimen de acceso de exclusión mutua. Un Shared_Resource que ha sido reservado por una actividad no puede ser liberado por ningún mecanismo de planificación, solo puede ser liberado por una operación explícita Unlock relativa a él. Como varias operaciones concurrentes pueden tratar de acceder a un mismo recurso, estos representan también una cola de actividades suspendidas a la espera de acceder a él. Así mismo, para evitar situaciones de inversión de prioridad en las actividades que se encuentran a la espera de acceder al recurso, cuando una actividad accede a un Shared_Resource, este modifica algunos parámetros de la Scheduling_Policy del Scheduling_Server en que se ejecuta.

En la figura 18 se muestra el metamodelo que describe los componentes derivados de la clase Shared_Resource.

Para el caso en que el criterio de planificación de las tareas pendientes de acceso sea por prioridades estáticas, se ha definido la clase abstracta **FP_Shared_Resource**.

Así mismo, y dentro de ella, en función de la prioridad con que se lleva a cabo la ejecución de una actividad que haya accedido al recurso, se han definido las dos clases concretas:

- **Immediate_Ceiling_Resource**: representa un Shared_Resource al que se accede siguiendo un criterio de prioridad fija, y de modo que la actividad que ha accedido se ejecuta dentro de su Scheduling_Server con el criterio de "Techo de Prioridad", esto es, se ejecuta a la prioridad nominal **Ceiling** del recurso (que para evitar inversiones de prioridad, debe ser igual a la mayor de las prioridades asignada a actividades que puedan hacer uso del recurso).

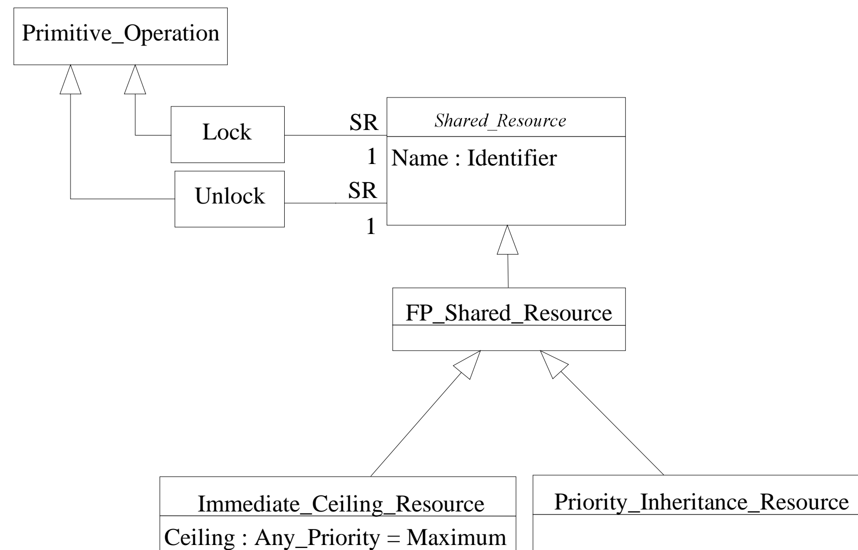

Figure 18: Shared_Resource Class

- **Priority_Inheritance_Resource**: Recurso al que se accede siguiendo un criterio de prioridades fijas. Cuando una actividad lo tiene reservado es planificada con el criterio de "Herencia de Prioridad", esto es, durante el tiempo que la actividad tiene reservado al recurso, se planifica dentro de su Scheduling_Server con la prioridad de la actividad que tiene mayor prioridad de entre las que esperan el acceso al recurso.

La clase concreta **Job** representa la secuencia de actividades concurrentes que se desencadenan como consecuencia de la invocación de un procedimiento, al cual por incluir sincronizaciones con objetos activos del sistema implica en su ejecución a múltiples Scheduling_Server y así mismo requiere mecanismos de sincronización para describir el modelo. Existen dos diferencias básicas entre un Job y una Composite_Operation:

- Una Composite_Operation representa un procedimiento secuencial que se planifica en un único Scheduling_Server. Este se establece a través de la actividad en que se invoca y en su descripción no se hace referencia al Scheduling_Server que lo soporta. Un Job puede representar una actividad que se ejecuta concurrentemente en varios Scheduling_Server, y en consecuencia, su descripción debe incluir el despliegue de las actividades que lo componen en los Scheduling_Server que los planifican.

- Una Composite_Operation es una estructura del modelo cuya única función es encapsular Operations y no admite referencias a estados internos (Named_State). Un Job es básicamente una sección de una transaction (ver II.3.1) y puede contener

interacciones con eventos (External_Event procedentes del entorno o Named_State declarados en otro punto de la transacción o en otros Jobs) y asignar Timing_Requirement a los Timed_States que se declaran en su descripción.

Cada Job tiene agregado un modelo de actividad que lo describe (Job_Model). En la siguiente figura se muestra el metamodelo de un Job_Model. El Job_Model que describe un Job se formula a través de uno o varios diagramas de actividad. A través de los swimlane de estos diagramas se establecen los Scheduling Server sobre los que se ejecuta el Job. Una actividad colocada sobre un swimlane, indica que debe ser planificada por el Scheduling_Server asociado a él.

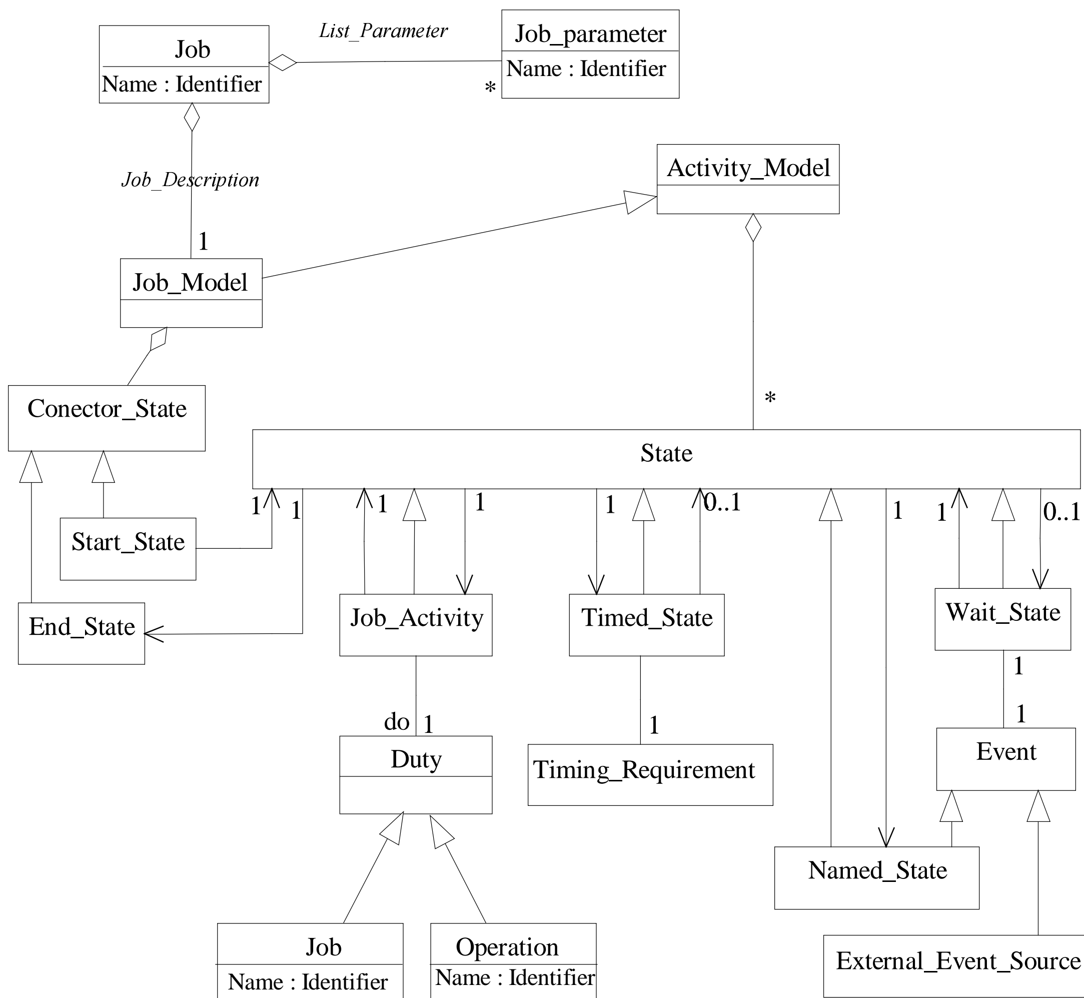

Figure 19: Job_Model and Activity_Model classes

Un Job_Model se describe mediante una máquina de estado (state machine) compuesta por un grafo abierto (sin bucles) y orientado e implementada en diagramas de actividad,. Los nudos del grafo del modelo de actividad son estados y los arcos son transiciones. En las figuras 19 y 23 se muestra el metamodelo del Job_Model que se utiliza para describir un Job.

Los estados del Job_Model, se pueden clasificar en dos tipos: los Conector_States y los States.

Los Conector_States son falsos estados que se introducen como elementos de interconexión entre módulos y son sólo utilizados en los Job_Model. Los States son Activity_States o Action_States, constituyen los nudos de la máquina de estados, y se utilizan tanto en los Job_Model como en los Activity_Model que describirán las Transaction.

Los dos Conector_States que se han definidos son:

Start_State: Representa el estado de partida en cada invocación del Job. En cada Job_Model solo puede haber un único Start_State. En los diagramas se representa mediante un pequeño círculo negro.

End_State: Representa el estado que implica la conclusión de la invocación del Job. En cada Job_Model sólo hay un End_State y se representa mediante un ojo de buey. Un Job es en general un conjunto de actividades concurrentes y formalmente el Job concluye cuando finalizan todas las actividades concurrentes que ha generado. Sin embargo, no es esto lo que se indica cuando se alcanza el End_State, sino que el Job ha alcanzado el estado en que se retorna el flujo de control a la actividad que sigue a la invocación del Job (lo que no necesariamente implica que hayan finalizado todas las restantes actividades concurrentes del Job).

Los States de un Activity_Model son:

Activity: Es un Activity_State en el que ser alcanzado representa que se activa una nueva ejecución de una Duty (Job u Operation) que se declara en su sentencia do/. El estado se abandona, realizandose la transición de salida, cuando se termina la ejecución de la Duty.

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 ejecutadas en el mismo Scheduling_Server.

La Duty que se ejecuta en una actividad se expresa mediante una sentencia do/ Duty_Identifier, con la que se establece la operación o Job cuya ejecución se modela. En este caso la Operation o el Job al que se hace referencia mediante su identificador en la sentencia do/ debe haber sido declarado y descrito en otra sección del modelo.

Las Activity de un Job_Model tienen una única transición de entrada y una única transición de salida.

En el modelo de actividad que describe un Job, cada Scheduling_Server que participa en su ejecución se representa mediante una banda vertical que se denomina **Swimlane**. Cada Activity del Job debe situarse en el Swimlane que corresponde al Scheduling_Server al que se asigna su ejecución. Así si en una actividad lo que se referencia es un Job, este es ejecutado por el Scheduling_Server que indica el Swimlane en que se sitúa la Activity. El Scheduling_Server en que se ejecutan las Operations o

Jobs declarados en un Job_Model, va especificado en la propia descripción del Job. Las Activity que en la descripción de un Job se asignan a un Swinlane sin nombre, se ejecutan en el Scheduling_Server en que se encuentra la Activity que invoca al Job.

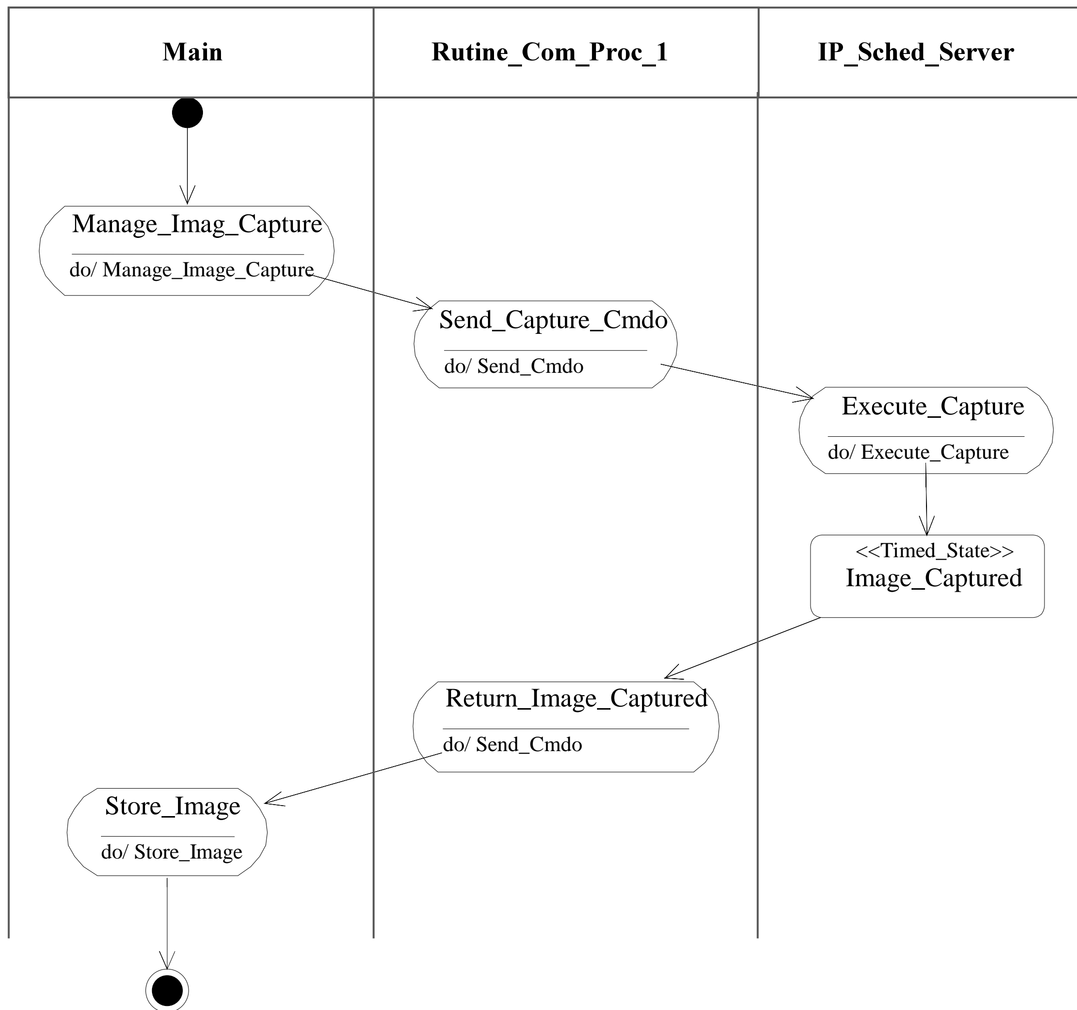

Figure 20 : Swimlanes assign Scheduling_Servers to activities.

Timed_Activity: Representa un Activity_ State que es activado por el timer del procesador. Si el timer es del tipo Alarm_Clock, existe una sección del código que se ejecuta a la prioridad de la interrupción del timer, y que puede interrumpir actividades de mayor prioridad que ella. Si el timer es del tipo Ticker, sólo se diferencia por la presencia de una granularidad extra en el tiempo de disparo y se analiza como un termino adicional de jitter.

Named_State: Representa un Action_State (y por tanto de duración nula) del Job_Model al que se le asigna un nombre para poder referenciarlo dentro de la misma transacción. Un Named_State se utiliza en el modelo de actividad en los siguientes dos casos:

- Cuando se necesita hacer referencia al estado que representa fuera del modelo del Job. Este es el caso, cuando se necesita que una de las líneas concurrentes de flujo de control del Job deba continuar en otro modelo de actividad de la misma transacción.

- Para finalizar una línea de flujo concurrente, que haya permanecido activa después de que haya finalizado la línea que corresponde a la invocación del Job. Si esta línea continua en otra Duty, representa el caso anterior.

El ámbito del identificador de un Named_State es la instancia del Job en que esta declarado. Los Named_States que están declarados en un Job, se instancian como Named_State diferentes en cada instanciación del Job. Si diferentes Job necesitan hacer referencia a un mismo Name_State, este debe ser transferido a través de un parámetro. Un Named_State se declara tanto de manera directa como un action state debida mente estereoyipado, como en un Wait_State que hace referencia a él.

Un Named_State tiene siempre una transición de entrada y puede tener o no una transición de salida.

Timed_State: Representa un Action_State que se declara a fin de asignar un requerimiento temporal (Timing_Requirement) al estado que representa dentro del Activity_Model. El nombre del Timed_State debe tener un identificador idéntico al del Timing_Requirement que se le asocia, y este debe estar declarado y asociado a la Transacción dentro de la que se define el Timed_State. Un Timed_State tiene siempre una transición de entrada, y puede tener una o no tener ninguna transición de salida.

Wait_State: Es un estado en el que al ser alcanzado se suspende la correspondiente línea de flujo de flujo de control a la espera de que se genere un evento externo o de que se alcance un Named_State con el mismo identificador en alguna sección del modelo de la transacción.

Un Wait_ State se representa mediante un pentágono cóncavo como el que se muestra en la figura 21. Una vez alcanzado el Wait_State, la línea de flujo de control se suspende hasta que el evento esperado se produce, y luego, se dispara la transición de salida.

Si el Wait_State representa una espera a un evento externo, el nombre del Wait_State debe coincidir con el nombre de la External_Event_Source, que caracteriza la temporización de los eventos externos y que debe haber sido declarada y descrita en otra sección del modelo como asociada la Transaction dentro de la que se define el Wait_State.

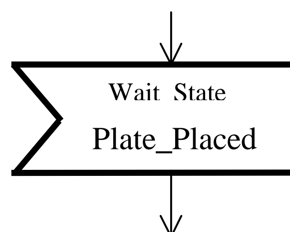

Figure 21.- Wait_State.

Si el Wait_State representa una espera a que el modelo alcance un determinado estado, el nombre del Wait_State debe coincidir con el del Named_State a cuya espera se suspende. La combinación Named_State y Wait_State es el mecanismo que se ofrece en el modelo para transferir una línea de flujo de control concurrente con la de activación de los módulos, de un módulo a otro.

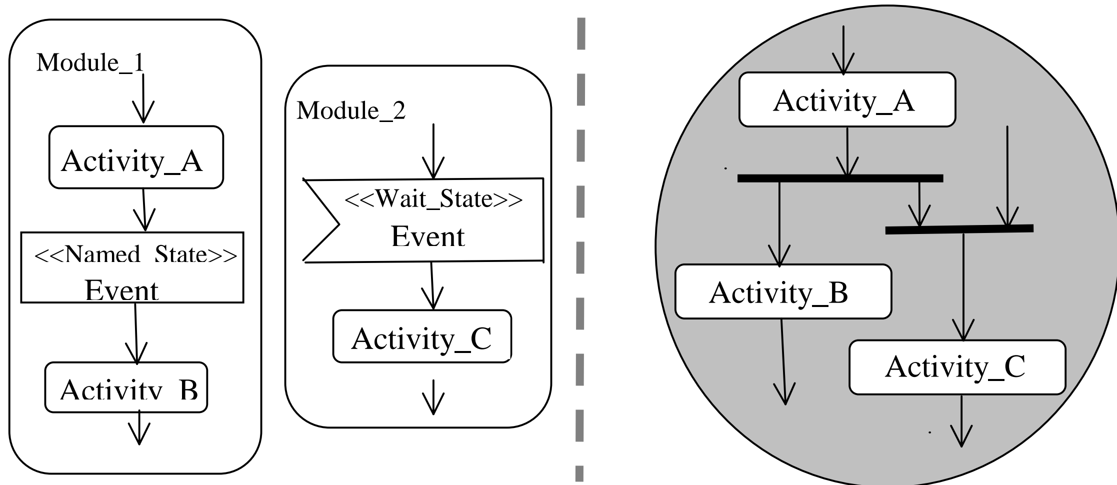

Figure 22.- Transfer of thread control between modules.

Si el Named_State tiene transición de salida, su combinación con un Wait_State implica un fork implicito (Figure 22). Si el Named_State no tiene transición de salida, pero el Wait_State tiene transición de entrada, la combinación representa una sincronización.

Por último, si el Named_State no tiene transición de salida y el Wait_State no tiene transición de entrada la combinación de ambos constituye una transferencia simple de flujo.

Existe la limitación de que solo puede existir un Wait_State por cada Named_State definido en el modelo del sistema. Sin embargo pueden existir Named_State no referenciados por ningún Wait_State.

Las actividades de control representan decisiones sobre transiciones entre estados optativos (branching), generación o finalización de líneas de flujo de control entre estados concurrentes (fork, synchronization), actividades de gestión de transiciones, etc.

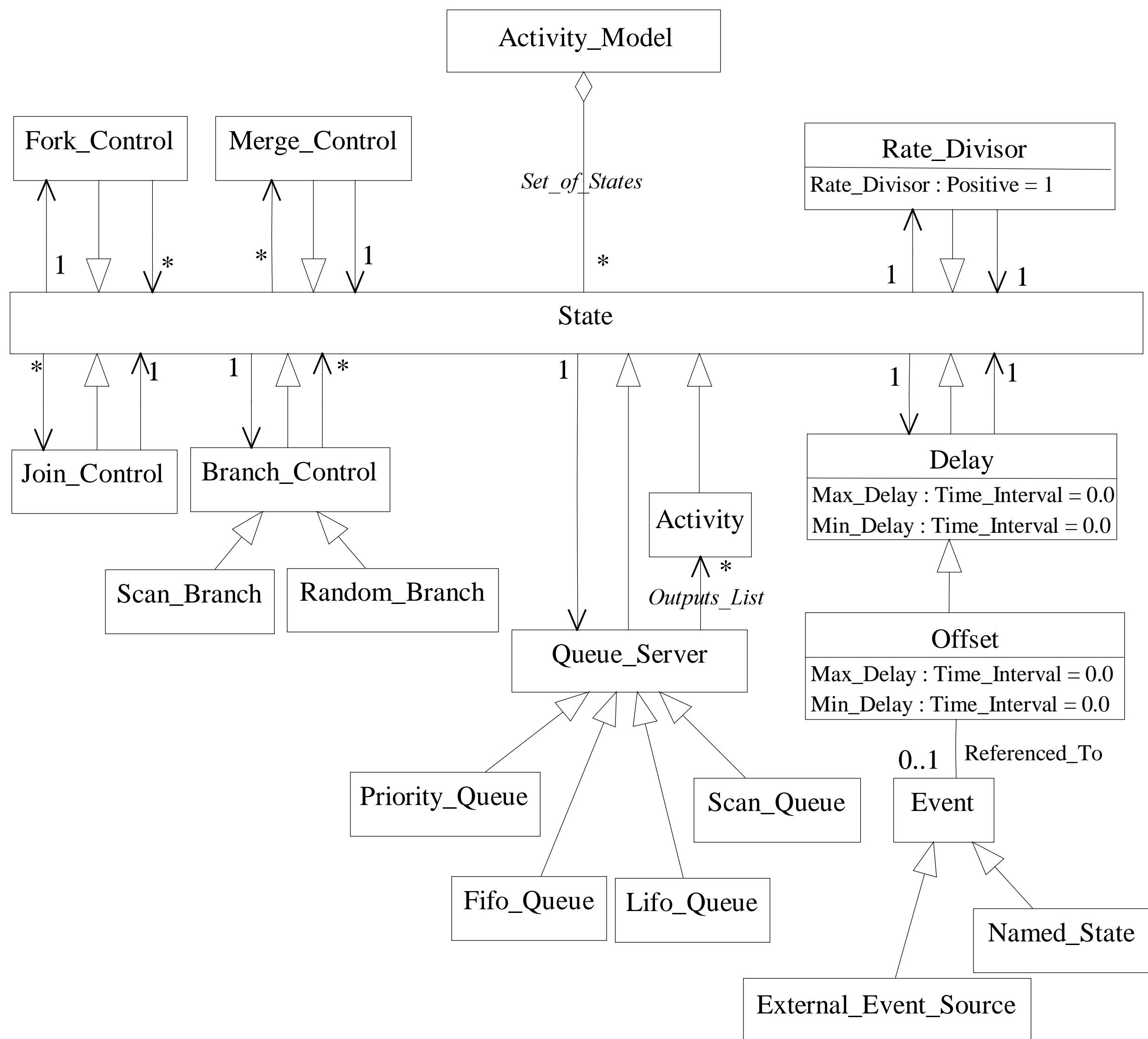

Figure 23.- Metamodel of Job_Model: Control Activities.

Los tipos de Control_Activity que se definen en un Job_Model son:

Branch_Control: Representa la transición desde un estado de partida a uno de varios estados optativos de salida. Se representa mediante un rombo con una única transición de entrada y múltiples transiciones de salida.

En la versión actual, se han definido dos políticas de selección de la transición de salida, y en función de ellas se han definido dos tipos de componente Branch: **Random_Branch** en el que la transición de salida se elige aleatoriamente (todas con igual probabilidad) y **Scan_Branch** en la que se eligen sucesivamente y de forma ordenada las transiciones de salida. El tipo de Branch_Control se establece mediante su estereotipo. Caso de que este no se haya definido para una Branch_control, se considera por defecto, que es del tipo Random_Branch. El tipo de Branch_Control no es relevante a efecto de análisis de tiempo real (análisis de peor caso), solo se tiene en consideración en las herramientas de animación.

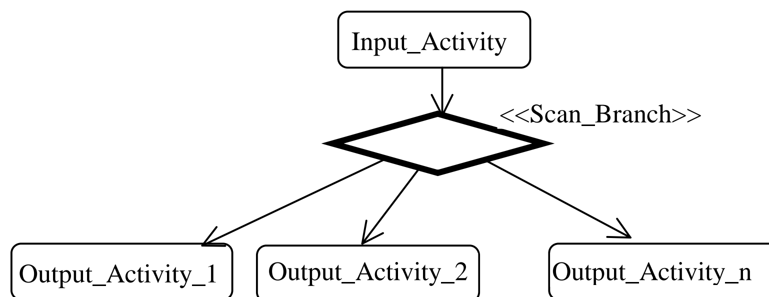

Figure 24: Branching Control.

En el modelo de tiempo real Mast_UML no son relevantes las condiciones que definen las opciones de transición de salida. Se pueden introducir a efecto de documentar el modelo, pero no se analizan.

Join_Control: Representa la reunificación de varias líneas de flujo de control que constituyen entradas alternativas hacia una nueva secuencia de actividades comunes. Dada la naturaleza concurrente y conducida por eventos de los sistemas que se modelan, no necesariamente las líneas de flujo de control proceden de un previo branching, sino pueden proceder de líneas de flujo de control procedentes de eventos externos alternativos.

En el diagrama de actividad, un componente Join_Control se representa por un rombo con múltiples transiciones de entrada y una única transición de salida.

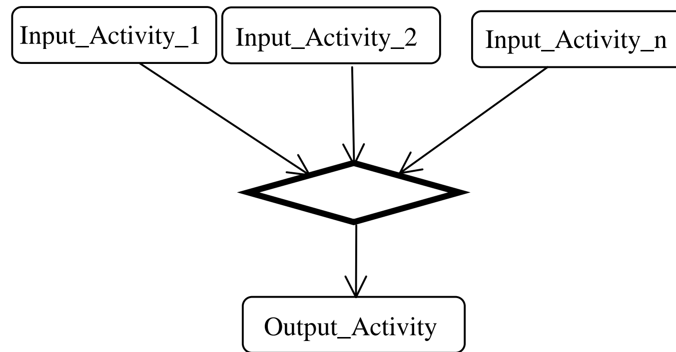

Figure 25: Join Control.

Fork_Control: Representa la activación de múltiples líneas de flujo de control concurrentes a partir de una línea de flujo de control de entrada. Se representa mediante una barra con una única transición de entrada y múltiples transiciones de salida. Cuando finaliza la actividad de entrada, se realizan concurrentemente todas las transiciones de salida.

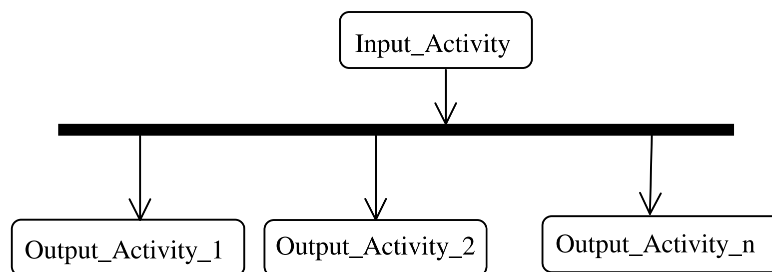

Figure 26: Fork Control

Merge_Control: representa un mecanismo de sincronización de múltiples líneas de flujo de control concurrentes hacia una única línea de flujo de control de salida. Cuando todas las actividades concurrentes de entrada han terminado, se activa la línea de flujo de control de salida. En un diagrama de actividad se representa por una barra de sincronismo con múltiples transiciones de entrada y una única transición de salida.

Un Merge_Control es un componente dinámico que recuerda las activaciones múltiples por cada transición de entrada. Cuando el Merge_Control se activa, consume solo una de las activaciones de cada una de sus entradas, y sigue recordando las que restan, que continúan pendientes. Por ello, en cualquier modelo se requiere que sean idénticas las frecuencia de activación de cada una de las transiciones de entrada en un Merge_Control.

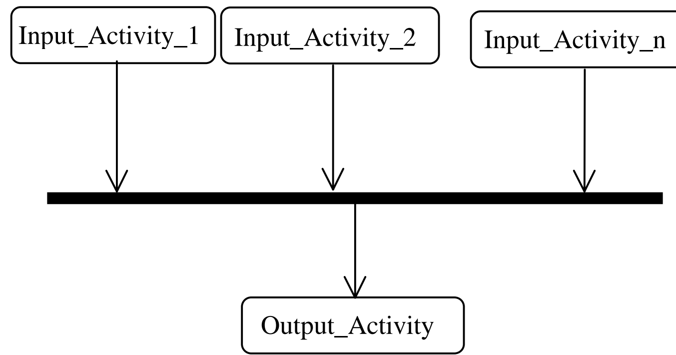

Figure 27 : Merge Control.

- **Queue_Server**: es una Control_Activity con una única transición de entrada y múltiples transiciones de salida. Las transiciones de salida de una Queue_Server deben ser Activities que ejecutan una Operation (tal como el Client_Activity_1 del ejemplo), o que ejecutan un Job que inicialmente ejecuta una Activity que ejecuta una Operation (tal como el Client_Job_2 del ejemplo).

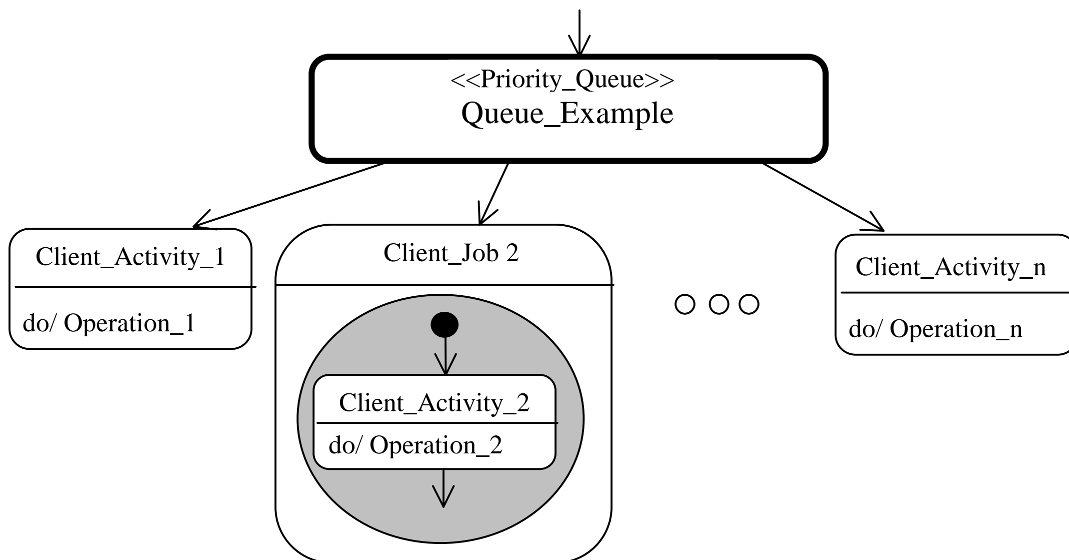

Figure 28 : Queue_Server Control usage.

Un Queue_Server es un Control_Activity de naturaleza dinámica que recuerda las activaciones de entrada, y por cada una de ellas produce una activación de salida por sólo una de las múltiples transiciones de salida que tiene. Una transición de salida sólo ocurre si la actividad cliente está dispuesta para la activación (esto es, ha terminado su activación anterior). Si varias actividades clientes de una Queue_Server están dispuestas para activación, cuando esta tiene pendiente una transición de salida, solo se activa una de ellas, y esta se selecciona de acuerdo con una política de atención (Priority, Fifo, Lifo o Scan) establecida.

De acuerdo con esta política se han definido los siguientes estereotipos: **Priority_Queue** en la que se activa la actividad cliente de mayor prioridad, **Fifo_Queue** en la que se activa la actividad cliente que lleva mas tiempo dispuesta para activación, **Lifo_Queue** en la que se activa la actividad cliente que mas recientemente ha pasado a dispuesta para activación, y **Scan_Queue** en la que se activa la actividad cliente que sigue a la última que fue servida, siguiendo el orden de declaración de las transiciones de salida.

El tipo de Queue_Server que se declara no tiene influencia sobre los análisis de tiempo real (peor caso), solo afectan a las herramientas de animación.

Un Queue_Server se representa en el modelo de actividad mediante una Activity con estereotipo (Priority_Queue, Fifo_Queue, Lifo_Queue o Scan_Queue)

Delay: es una actividad que realiza una transición de salida un tiempo especificado después de que se haya producido su transición de entrada. Se representa como una actividad con estereotipo Delay, y se establecen los tiempos que caracterizan el retraso mediante sentencias:

do/Max_Delay(D : Time_Interval = 0.0)
do/Min_Delay(D : Time_Interval = 0.0)

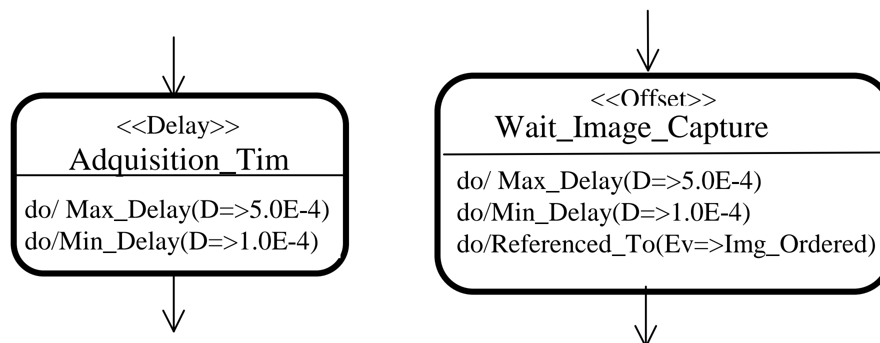

Figure 29 : Delay and offset activity.

Offset: es una actividad que realiza una transición de salida al menos un tiempo especificado después de que se haya producido un evento al que se hace referencia. Para que la transición de salida se produzca es necesario que se haya realizado previamente la transición de entrada. Se representa por una actividad con estereotipo Offset. Los tiempos que caracterizan el retraso se establece a través de las sentencias:

do/ Max_Delay(D : Time_Interval = 0.0)
do/ Min_Delay(D : Time_Interval = 0.0)

y el evento a que hace referencia se establece a través de una sentencia:

do/ Referenced_To(Ev : Identifier)

Si el evento referenciado es externo, Event_Identifier debe ser el identificador de la

External_Event_Source que caracteriza su generación y si el evento referenciado es relativo a alcanzar un estado interno, el identificador debe ser el del correspondiente Named_State. Si esta sentencia no se especifica se supone como evento referenciado elm de activación del Offset.

Rate_Divisor: Es una actividad dinámica que realiza una transición de salida por cada Rate_Divisor transiciones de entrada que ocurren. En el diagrama se representa mediante una Activity con estereotipo Rate_Divisor. El valor del atributo Rate_Divisor se establece mediante una sentencia:

do/ Rate_Divisor(Rd : Positive = 1)

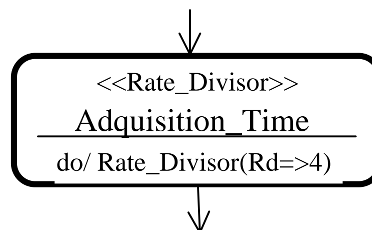

Figure 30 : Rate divisor.

Los parámetros que se definen en un Job se declaran como atributos de la clase que define el Job. Los parámetros se declaran con su identificador, su tipo y, en su caso, el valor por defecto.

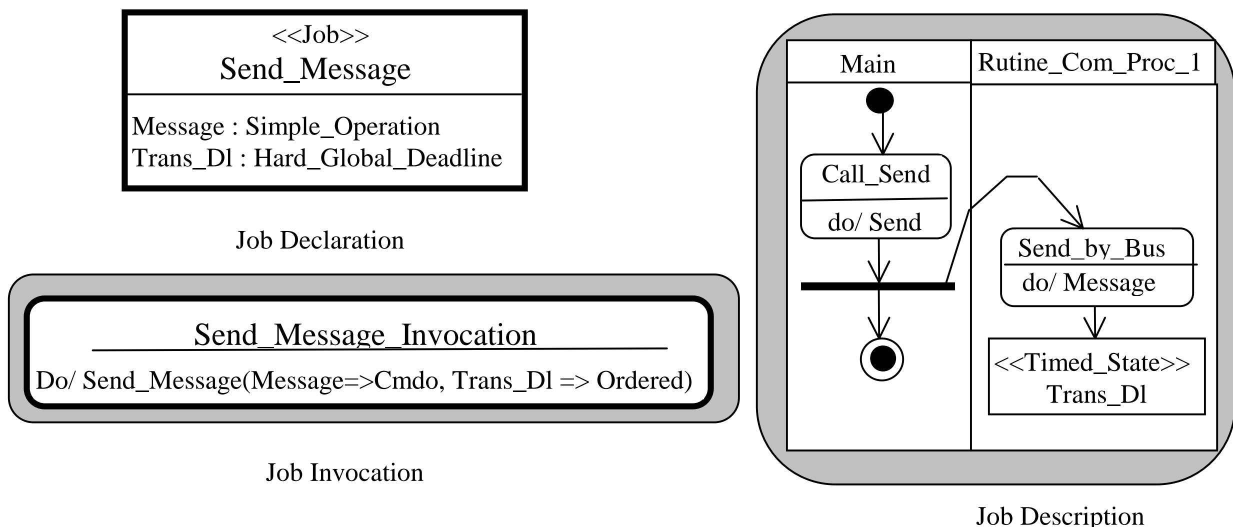

Figure 31 : Declaration, definition and invocation of a Job Component.

En la figura 32 se muestra el metamodelo que describe los parámetros de un Job.

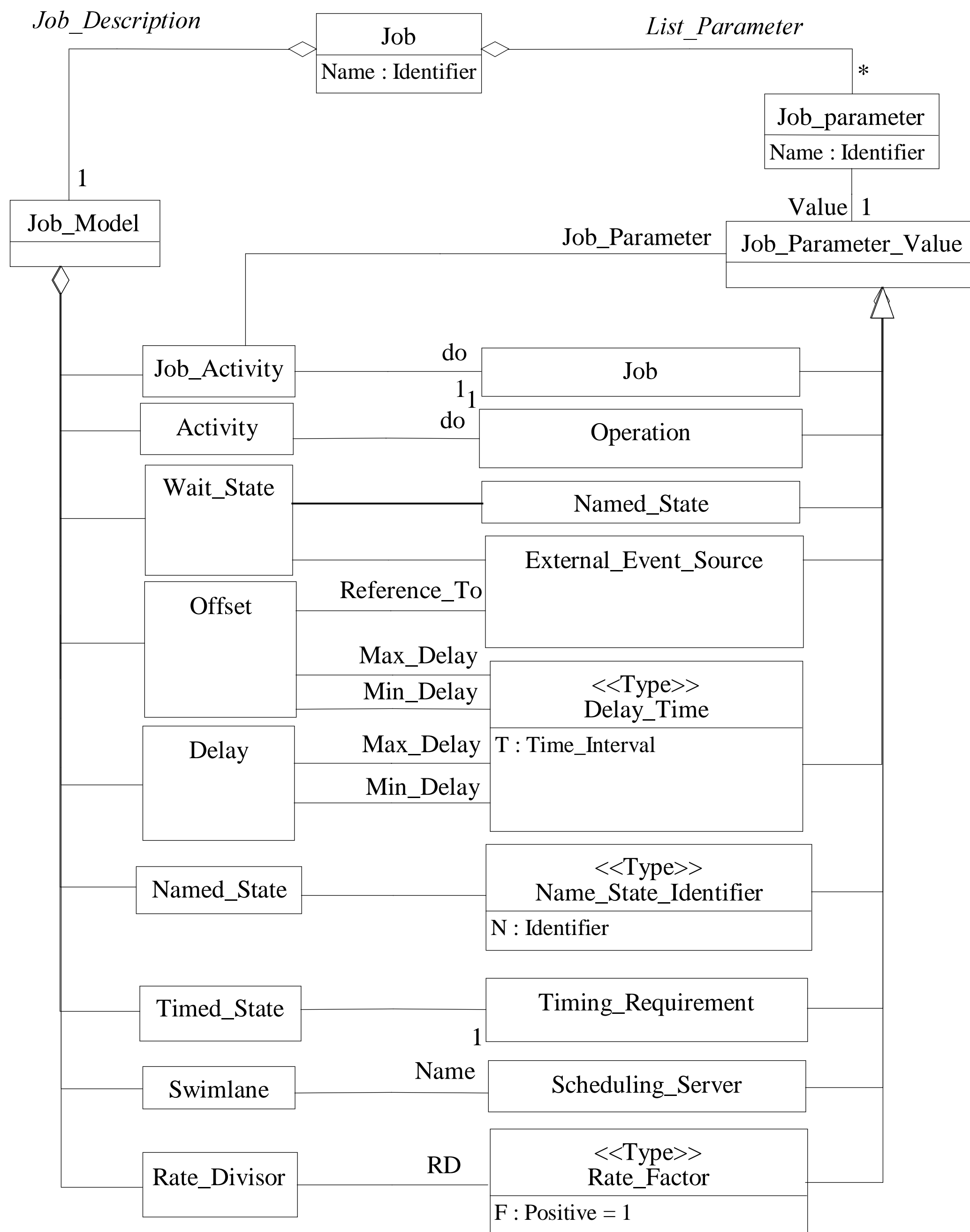

Figure 32.- Metamodel of Job parameters.

Los parámetros que pueden declararse en un Job pueden ser de los siguientes tipos:

Type Duty: Representa simbólicamente el identificador de una Duty, que es una clase abstracta que reune las clases Job y Operation. Este tipo de parámetros se utiliza para asignar simbólicamente una Duty a una activity del Job, o como valor de un parámetro de una Duty que sea a su vez de tipo Duty.

Type Shared Resource: Representa simbólicamente el identificador de un Shared_Resource que va a ser utilizado como argumento de una operación Lock o Unlock. También puede aparecer como parámetro de una duty interna a la descripción en una sentencia do/.

Type External_Event_Source: Representa simbólicamente el identificador de una fuente de eventos externos al que hace referencia un Wait_State o se puede utilizar como argumento en una sentencia do/Referenced_To de un Offset. También puede aparecer como parámetro de una duty interna a la descripción en una sentencia do/.

Type Named_State: Representa simbólicamente el identificador de un Named_State. Puede ser utilizado como identificador de un Wait_State o de un Named_State. También puede aparecer como parámetro de una duty interna a la descripción en una sentencia do/ o como argumento en una sentencia do/Referenced_To de un Offset.

Type Timing_Requirement: Representa simbólicamente el identificador de un Timed_State. También puede aparecer como parámetro de una duty interna a la descripción en una sentencia do/.

Type Scheduling_Server: Representa simbólicamente el identificador de un Scheduling_Server. Se puede utilizar dentro de la declaración de un Job como identificador de un Swimlane. También puede aparecer como parámetro de un Job interno a la descripción en una sentencia do/.

Type Time_Interval: Representa un valor numerico float, que puede utilizarse como argumento de una sentencia de los tipos do/Max_Delay o do/Min_Delay. También puede aparecer como parámetro de una duty interna a la descripción en una sentencia do/.

Type Rate_Divisor: Representa un valor numérico Natural, que puede utilizarse como argumento de una sentencia do/Rate_Divisor de un componente Rate_Divisor. También puede aparecer como parámetro de una duty interna a la descripción en una sentencia do/.

Type Overridden_Sched_Parameter: Representa un tipo que se asocia como parámetro de una operación compuesta o de una duty interna a la descripción en una expresión do/.

II.2.2 Modelo de los componentes lógicos en la herramienta CASE_UML.

El modelo de los componentes lógicos se formula como un conjunto de diagramas de clases y objetos que a efecto de su tratamiento posterior por la herramienta automática se ubican en un package tal como Process View.Mast_RT_View.RT_Logical_Model.

El modelo de los componentes lógicos se compone de:

- Modelo de tiempo real de cada una de las clases software que se describen en la vista lógica que constituye el sistema. Este modelo está constituido por un conjunto de diagramas de clase en los que se declaran y describen los componentes UML_Mast que modelan cada uno de los procedimientos o funciones de la interfaz de las clases de la vista lógica del sistema. Por cada Operation y por cada Job que se declara, se formula uno o varios diagramas de actividad que constituyen el modelo de actividad que lo describe.

- Modelo de las operaciones que corresponden a las operaciones de comunicación y que describen la transferencia de los diferentes tipos de mensajes del sistema por un Network. Es un conjunto de diagramas de clase que describen las operaciones de transferencia de cada tipo de mensaje por el Network. Por cada Composite_Operation se formula el diagrama de actividad que lo describe.

- Modelo de las operaciones que corresponden a tareas o actividades que son ejecutadas por equipos, periféricos, coprocesadores, etc. que contribuyen a la temporización del sistema.y que no corresponden a código desarrollado. Es un conjunto de diagramas de clases que describen las operaciones realizadas por equipos y perifericos. Por cada Composite_Operation se formula el diagrama de actividad que lo describe y el de las operaciones que internamente se invoquen.

- Declaración y caracterización de los Shared_Resources que representan las interaciones entre operaciones del sistema. Estos resultan de identificar los componentes de sincronización presentes en el sistema: Semáforos, Mutex, Objetos protegidos, Secciones críticas, etc. En el modelo son un conjunto de diagramas de objetos en los que se declara y caracteriza cada uno de los componentes.

Normas en la formulación del Modelo de los Componentes Lógicos:

Estas normas no son relevantes desde el punto de vista conceptual, sino que son requeridas por las herramientas de extracción del modelo.

1) Es ignorado cualquier clase u objeto que se introduzca en los diagramas del modelo y que no corresponda a instanciaciones del metamodelo UML_MAST. Por ejemplo, es habitual presentar los modelos como componentes agregados o asociados de las clases lógicas que modelan. Estas sólo tienen la finalidad de documentación y son ignoradas por las herramientas de extracción.

2) En el Modelo de los Componentes lógicos los roles no son relevanes y no se analizan.

Se pueden incluir si se consideran útiles a efecto de documentación, pero no son tenidos en cuenta.

3) Es optativo introducir el tipo de los atributos, pero si se introducen deben ser correctos.

4) Los esterotipos definidos en el modelo Mast_UML son:

Estereotipos de Clases:

- <<Job>> -- Declara un Job
- <<Simple_Operation>> -- Declara una Simple_Operation
- <<Composite_Operation>> -- Declara una Composite_Operation
- <<Enclosing_Operation>> -- Declara una Enclosing_Operation
- <<Immediate_Ceiling_Resource>> -- Declara un Shared_Resource
- <<Priority_Inheritance_Resource>> -- Declara un Shared_Resource

Estereotipos de Activity-State:

- <<Activity>> [*] -- Invoca una Activity
- <<Timed_Activity>> -- Invoca una System-Timer Activated Activity
- <<Delay>> -- Invoca un Delay
- <<Offset>> -- Invoca in Offset
- <<Rate_Divisor>> -- Invoca un Rate_Divisor
- <<Priority_Queue>> -- Declara una Queue
- <<Fifo_Queue>> -- Declara una Queue
- <<Lifo_Queue>> -- Declara una Queue
- <<Scan_Queue>> -- Declara una Queue

Estereotipos de Action-State:

- <<Named_State>> -- Declara un estado nominativo o evento local
- <<Timed_State>> -- Declara un estado con temporización
- <<Wait_State>> -- Declara un Wait_State

Estereotipos de Control_Activity:

- <<Random_Branch>>[*] -- Declara una Branch Control
- <<Scan_Branch>> -- Declara una Branch Control

[*] El estereotipo es optativo equivale a no estar establecido.

6) Valores por defecto:

- Todos los atributos de las clases del metamodelo tienen valores por defecto. Estos valores se muestran en el metamodelo. Si en un objeto del diagrama no se establece valor del atributo en el modelo, se considera que tiene su valor por defecto.
- Si en un diagrama de actividad una Activity se declara fuera de cualquier Swimlane, se considera que se encuentra asignada al Swimlane al que directamente o indirectamente (por aplicar recursivamente esta regla) se encuentra asignada la activity desde la que se ha invocado el Job al que describe el diagrama de actividad.

7) <u>Son situaciones de error:</u>

- La clase que declara un Job, una Simple_Operation, una Composite_Operation o una Enclosing_Operation no tiene agregado un diagrama de actividad que describe su modelo de Actividad.
- En la clase que declara una Simple_Operation o una Enclosing_Operation se definen parámetros no definido en el metamodelo (WCET, ACET y BCET).
- En la clase que declara un Composite_Operation, el tipo de los parámetros es diferente a cualquiera de los tipos:
  - Operation - Shared_Resource -Overridden_Sched_Parameter
- En la clase que declara un Job, el tipo de los parámetros que se definen no corresponde a alguno de los tipos:
  - Duty - External_Event_Source - Named_State
  - Shared Resource - Timing_Requirement - Time_Interval
  - Scheduling_Server - Overriden_Sched_Parameter - Rate_Divisor
- El modelo de actividad de un Job no tiene un Start_State y un End_State.
- Una barra de sincronización tiene a la vez mas de una transicion de entrada y mas de una transición de salida.
- Un componente de control de branching tiene a la vez mas de una transición de entrada y mas de una transición de salida.
- En un modelo de actividad hay asignadas a una Activity_State, Action_State o Activity_Control, un número de transiciones no compatible con el metamodelo:

Component / Stereotype	# Input Transition	# Output Transition
Activity / <<Activity>>	1	1
Activity / <<Timed_Activity>>	1	1
Activity / <<Delay>>	1	1
Activity / <<Offset>>	1	1
Activity / <<Rate Divisor>>	1	1
Action_State / <<Wait_State>>	0..1	1
Activity / <<Priority_Queue>>	1	1..n
Activity / <<Fifo_Queue>>	1	1..n
Activity / <<Lifo_Queue>>	1	1..n
Activity/<<Scan_Queue>>	1	1..n
Action_State/<<Named_State>>	1	0..1
Branch / <<Random_Branch>>	1	2..n
Branch/<<Scan_Branch>>	1	2..n
Join	2..n	1

Fork	1	2..n
Merge	2..n	1

- En la activity que invoca un Job u operation existe más de una expresión do/.
- Se introduce en una actividad una sentencia do/ con una función no contemplada en el metamodelo, o con unos parámetros que no son compatibles con la función:

Activity Stereotype	Function	Parameter	Parameter Type
None or <<Activity>> <<Timed_Activity>>	Do/ Duty	Declarated	Defined
<<Delay>>	Do/ Max_Delay do/ Min_Delay	D D	Time_Interval Time_Interval
<<Offset>>	do/ Max_Delay do/ Min_Delay do/ Referenced_To	D D Ev	Time_interval Time_Interval Identifier
<<Rate_Divisor>>	do/Rate_Divisor	Rd	Positive
<<Wait_State>>	None	-	-
Queue	None	-	-

- El nombre de un Wait_State no corresponde al identificador de un parámetro, al identificador de un External_Event_Source definido en el modelo o al identificador de un Named_State definido en algún Activity_Diagram de la transaction.
- Dos Wait_States tienen el mismo identificador (Hacen referencia a un mismo Event)
- En una invocación se hace referencia a un objeto que no es el identificador de un parámetro del tipo que corresponda o el identificador de un objeto de la clase que corresponda y que esté definido en el modelo.
- Alguna de las transiciones de salida de una Queue no está asociada a una activity cuya primera operation sea una Simple_Operation.
- Existen bucles en un modelo de actividad.

8) <u>Son situaciones que dan origen a un warning:</u>
   - En la declaración de un Job o Composite_Operation se declaran parámetros cuyos identificadores no se utlizan en la descripcón del Job o Composite_Operation.
   - El modelo de actividad de una Operation tiene Start_state y End_state, aún cuando es una notación soportada se emite el warning para evitar confusión en el usuario con el modelado de jobs.

Normas de adaptación a la Herramienta ROSE'2000.

Dado que esta herramienta no permite añadir una nueva vista a un modelo, el modelo de los componentes se ubica en el package Logical View.Mast_RT_View.RT_Logical_Model.

Los componentes se representan mediante símbolos de clase, a través del estereotipo de estas clases se explicita la clase del metamodelo de la que se instancia.

Se recomienda dar al componente del modelo UML_MAST el nombre compuesto (<Nombre de la clase lógica>.<Nombre del procedimiento>) con que se invoca el procedimiento lógico en el código.

Ejemplo:

*En el ejemplo se modela la clase **Image** que corresponde al software relativo a la adquisición de una imagen a través de un tarjeta hardware (**Image_Processor**) de digitalización de vídeo y su posterior procesado para extraer ciertos parámetros de la imagen que son de interés. En la figura 33 se muestra la vista lógica del componente lógico que se modela en este ejemplo. En el diagrama de secuencia de la figura 34 se muestra la funcionalidad básica de la clase.*

La clase Image se ha diseñado activa, ya que crea una tarea auxiliar para procesar la imagen, mientras el thread que invocó la adquisición queda libre para realizar otras actividades de la aplicación.

*La clase Image hace uso de la clase **Processor_Controller** que contiene el software reusable de gestión de la tarjeta Image_Processor. En el ejemplo se supone que la tarjeta va a ser utilizada por diferentes componentes concurrentes, y que deben ser atendidos con exclusión mútua. Por esta razón, la clase **Processor_Controller** se ha diseñado como un objeto protegido.*

*La clase **Processor_Controller** hace uso a su vez de la clase **Bus_Driver** que es una librería estándar de comunicación a través del Bus, que es el mecanismo hardware que conecta físicamente el computador que soporta la aplicación y el Image_Processor. La comunicación a través del Bus se realiza por paquetes, a fin de que ofrezca prestaciones de tiempo real. Por esta razón, las funciones de Bus_Driver manejan como argumentos la estructura Message que representa la información que se transmite segmentada en paquetes.*

*Class **Image**: Ofrece un procedimiento y una función:*

- ***Adquire_and_Process**: Es un procedimiento no bloqueante. Espera que sea concedido el acceso al Image_Processor y una vez conseguido, por un lado retorna el control del thread que lo invocó, y por otro lado, a través de una tarea agente gestiona la captura y el procesado de la imagen. Este procedimiento hace uso de la clase Processor_Controller, para capturar la imagen.*

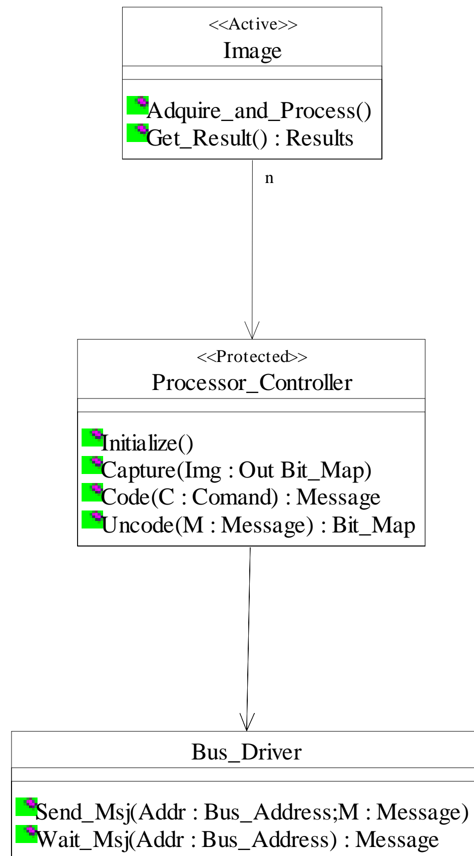

Figure 33: Logical View of Image class.

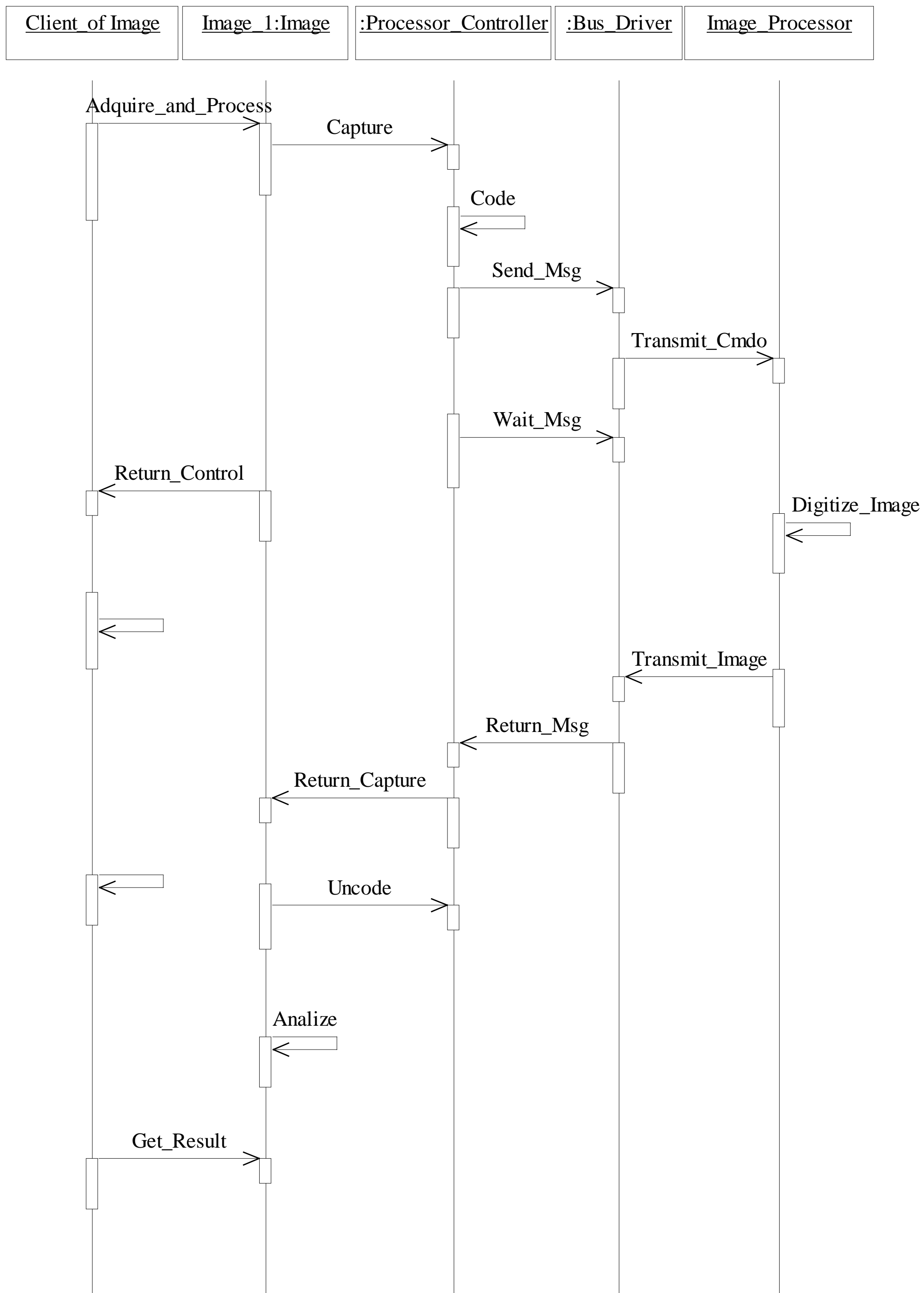

Figure 34: Sequence Diagram of Image class procedures.

- ***Get_Result**: Función que espera (si se necesita) que la tarea agente haya concluido el procesado y luego retorna los resultados del análisis.*

*Class **Processor_Controller**: Ofrece dos funciones y dos procedmienos:*

- ***Initialize**: Es un procedimiento que inicializa la tarjeta Image_Procesor. No se modela ya que corresponde a una operación que se ejecuta fuera de los escenarios de tiempo real.*

- ***Capture**: Es el procedimiento que proporciona la funcionalidad principal de la clase, gestiona el acceso en régimen exclusivo al Image_Processor, envia por el bus el mensaje que ordena la digitalización de la imagen, y recibe la imagen una vez adquirida.*

- ***Code** y **Uncode**: Son dos funciones auxiliares, que realizan respectivamente la segmentación del comando en un message, y la recontruccion del Bit_Map de la imagen a partir de los paquetes del message de retorno. Aunque son funciones pasivas, (no gestionadas dentro del objeto protegido), requieren ser modeladas porque se utlizan dentro de las transacciones de tiempo real.*

*Class **Bus_Driver**: Ofrece dos operaciones que son ejecutadas por las actividades cliente que hacen uso de él.*

- ***Send_Msj**: Procedimiento que inserta el mensaje en la cola de paquetes pendientes de transmisión, para que cuando proceda, sea transferido por el software de transmisión por el Bus.*

- ***Wait_Msj**: función que suspende el thread que lo ejecuta hasta que el software de transmisión por el Bus detecta un mensaje de retorno completo. Retorna el mensaje recibido.*

Modelo de Tiempo Real de la clase Image

Se modelan independientemente la propia clase Image, y las dos clases Processor_Controller y Bus_Driver de las que hace uso .

Modelo Real Time de la Clase Image:

En el diagrama de clases de la figura 35, se declaran los dos objetos que describen el modelo de tiempo real: Image.Adquire_and_Process y Image.Get (observese que se utiliza la designación compuesta aconsejada, en la que se hace referencia a la clase y al procedimiento). Así mismo, en el diagrama se muestran los objetos del modelo de tiempo real como agregados a la clase lógica. Esto es irrelevante, ya que los componentes que no son instanciaciones de elementos del metamodelo Mast_UML (como son las clases lógicas) y por tanto, no son tenidos en cuenta a efectos de la vista de tiempo real.

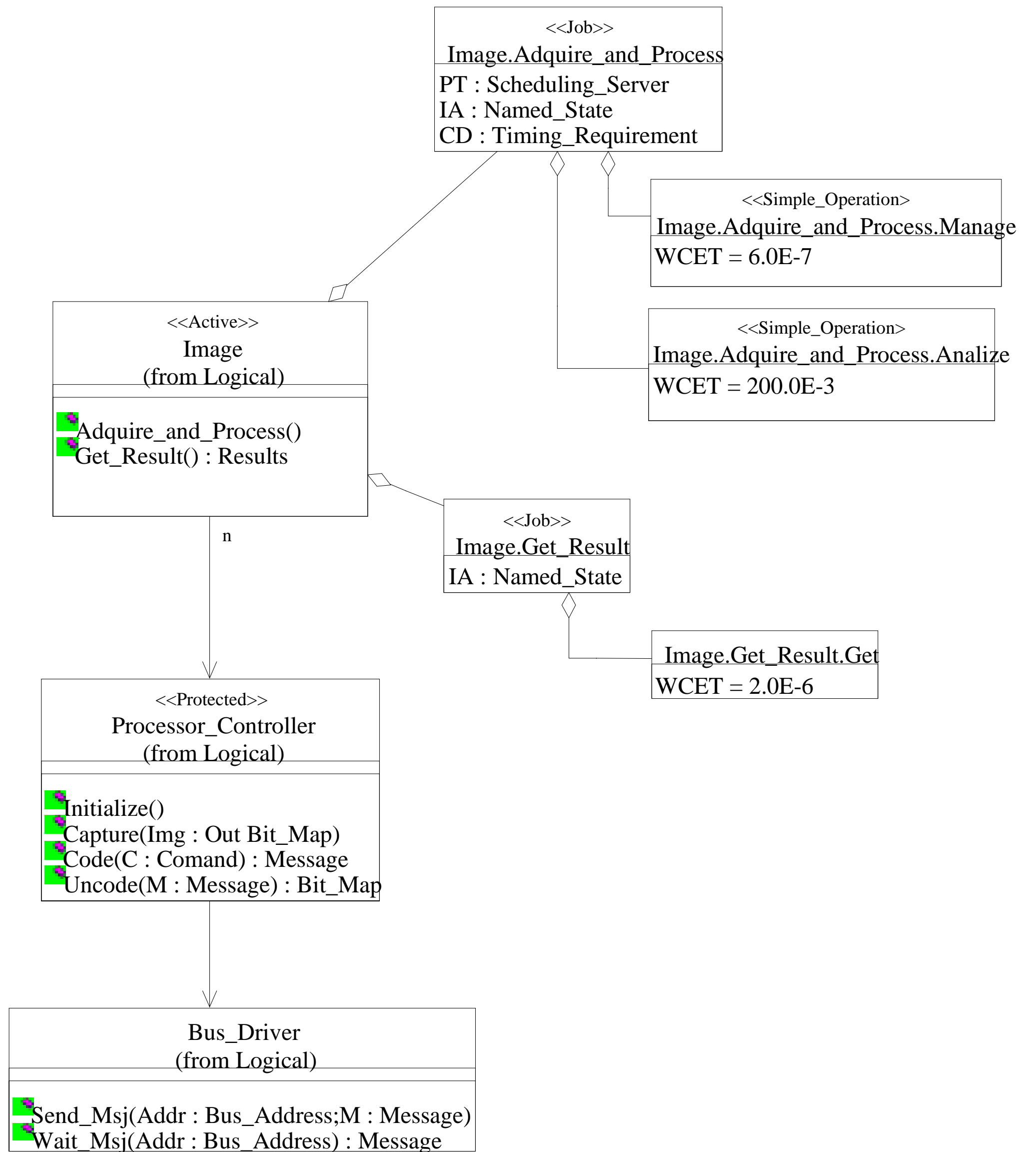

Figure 35.- Components declarations of Real Time model of Image Class.

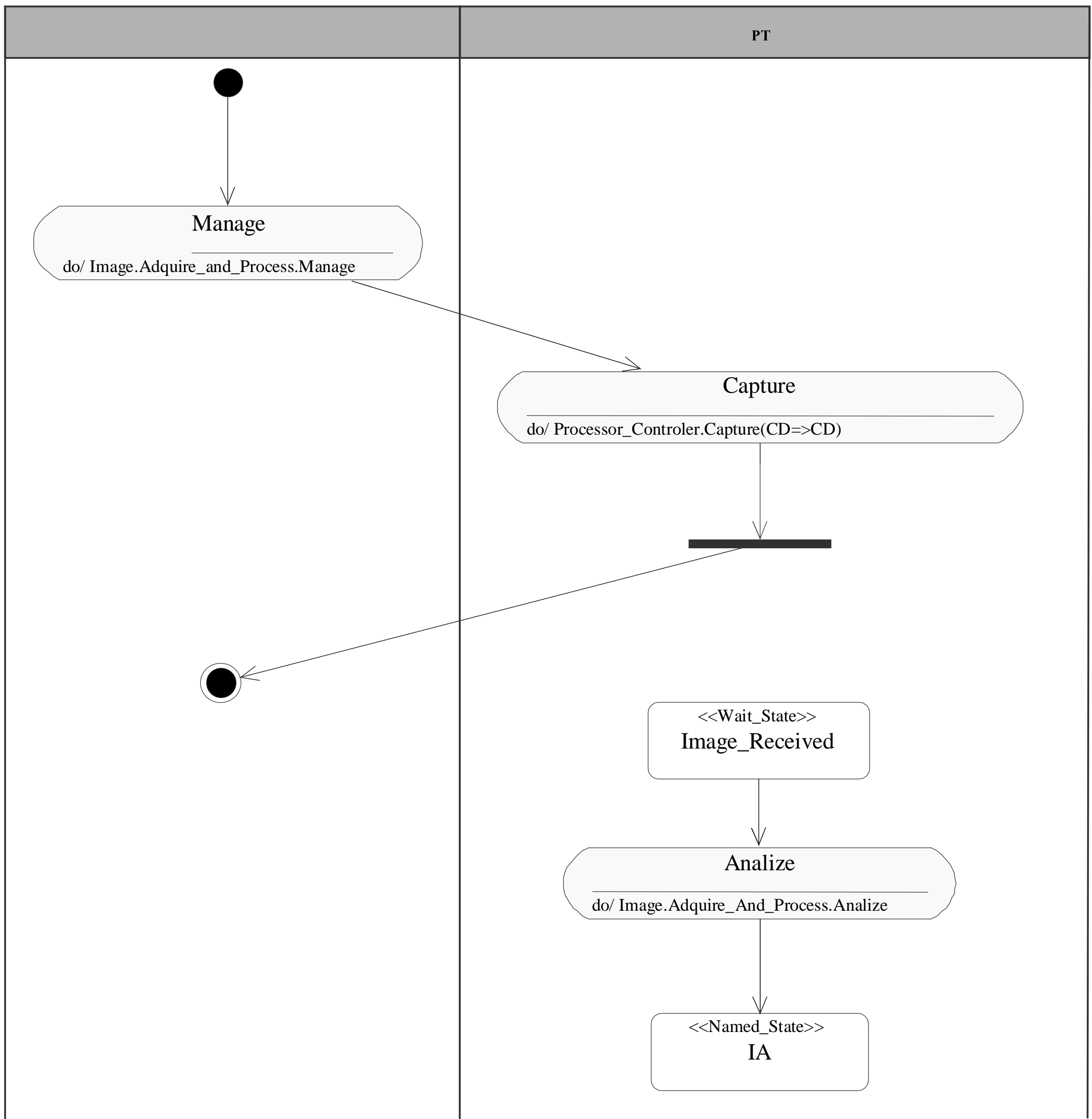

Figure 36.- The activity diagram describes the Image.Adquire_and_Process job.

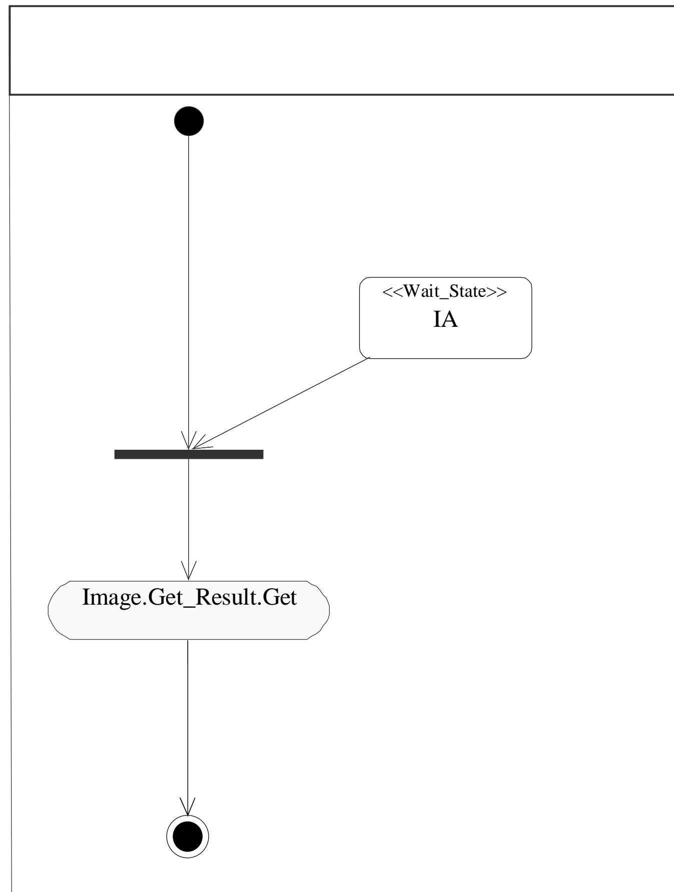

Figure 37.- The diagram activity describes the Image.Get_Result job.

Image.Adquire_and_Process*: Es un Job, ya que implica la ejecución de multiples actividades que se ejecutan en diferentes Scheduling_Server. Tiene definidos dos parámetros:*

> ***Process_Task****: que representa el identificador del Scheduler_Server en que se ejecuta la tarea agente que va a gestionar la adquisición y análisis de la imagen, y que en la invocación de Image (cuando un cliente haga uso de Image) deberá asignarsele el identificador de un componente Scheduler_Server declarado en el modelo de la plataforma. Se define como parámetro porque se supone que va a haber muchas instancias de Image, y en cada una de ellas el Scheduling_Server será diferente.*
>
> ***Capture_Deadline****: que representa el identificador de un deadline. Correspnde al deadline de finalización de la digitalización de la imagen que se requiere, y cuyo cumplimiento va a ser verificada en el análisis. Obviamente es un parámetro porque en cada adquisición de una imagen el requerimiento temporal puede ser diferente.*

*Agregado a él se declaran los modelos de los componentes que se requieren para su descripción. En este caso son dos Simple_Operation: I****mage.Adquire_and_Process. Manage*** *y* ***Image.Adquire_and_Process.Analize****. En ambos casos solo se especifica el valor del atributo de peor caso (WCET), ello indica que los otros dos parámetros (ACET y BCET) toman su valor por defecto.*

La descripción del job Image.Adquire_and_Process se realiza mediante el diagrama de activación que se muestra en la figura 37.

- *Inicialmente se ejecuta en el thread de invocación (observese que el correspondiente Swimlane no tiene identificador) la operación breve de gestión Image.Adquire_and_Process.Manage.*
- *Desde el thread de la tarea agente (Observese que está parametrizada, y que el Shedulling_Server quedará establecido cuando se invoque) se invoca la job Processor_Controller.Capture. Observese que el valor del parámetro Capture_Deadline, se pasa como valor de parametro (de igual nombre) en la invocación de esta job.*
- *Cuando se retorna el control (al ser un Job no significa que haya concluido), por un lado se retorna el control del thread que invocó el job Image.Adquire_and_Process, y por otro lado se permanece a la espera de que se alcance el State_Action Image_Received, que deberá estar declarado en otro módulo (en este caso en Processor_Controller.Capture). (Observese que en este caso no era necesario definir como parámetro este State_Action ya que no es posible que dos Imagenes puedan estar siendo capturadas simultaneamente*).
- *Finalizado el Wait se invoca la operación de análisis de la imagen Image_Adquire_and_Process.Analize.*
- *Finalizado la operación Analize se alcanza el Action_State Image_Analized, que posteriormente será requerido para retornar los resultados del análsis.*

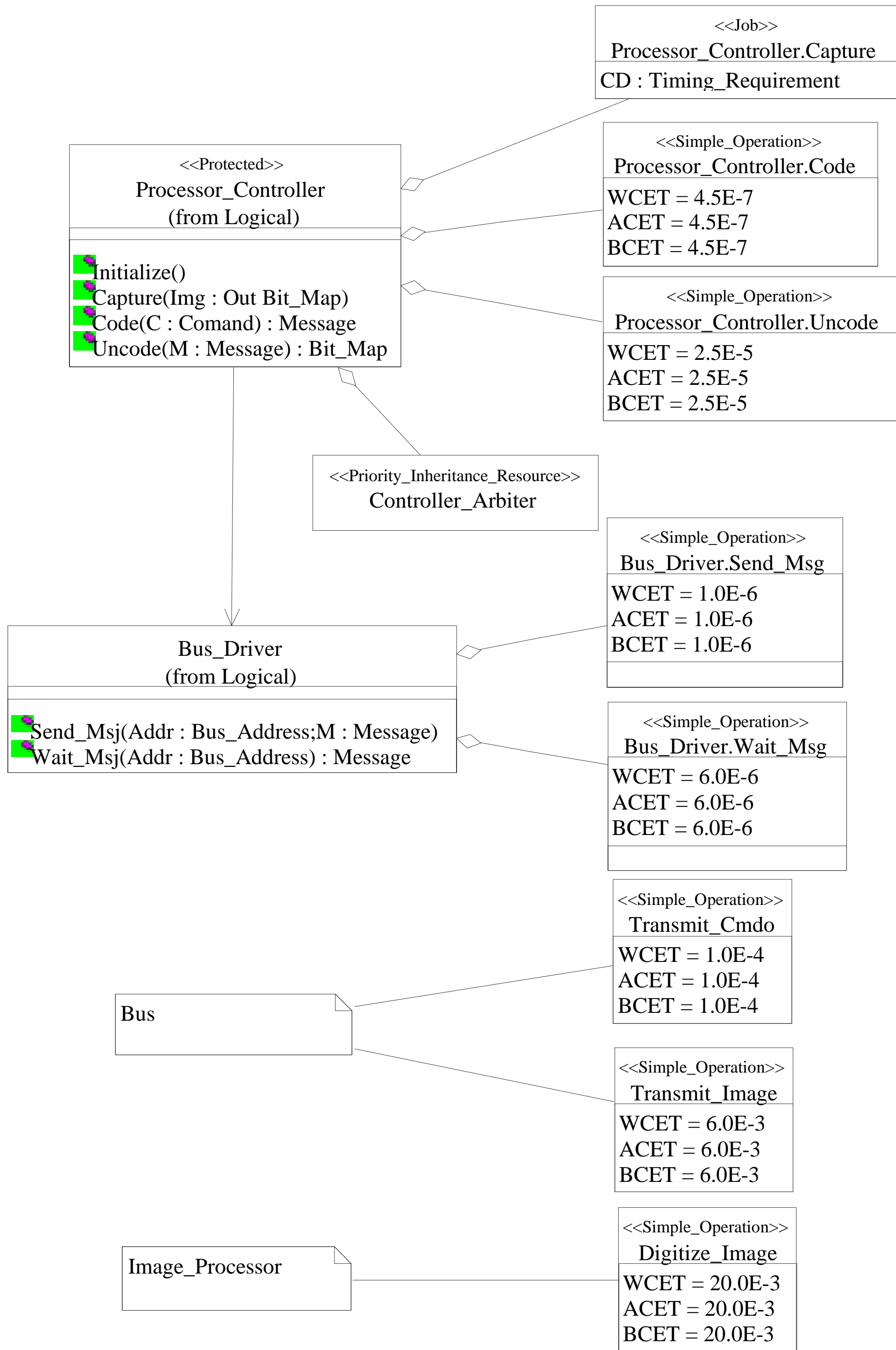

Figure 38.- Components declarations of Real Time model of Processor_ Controller y Bus_Driver Classes.

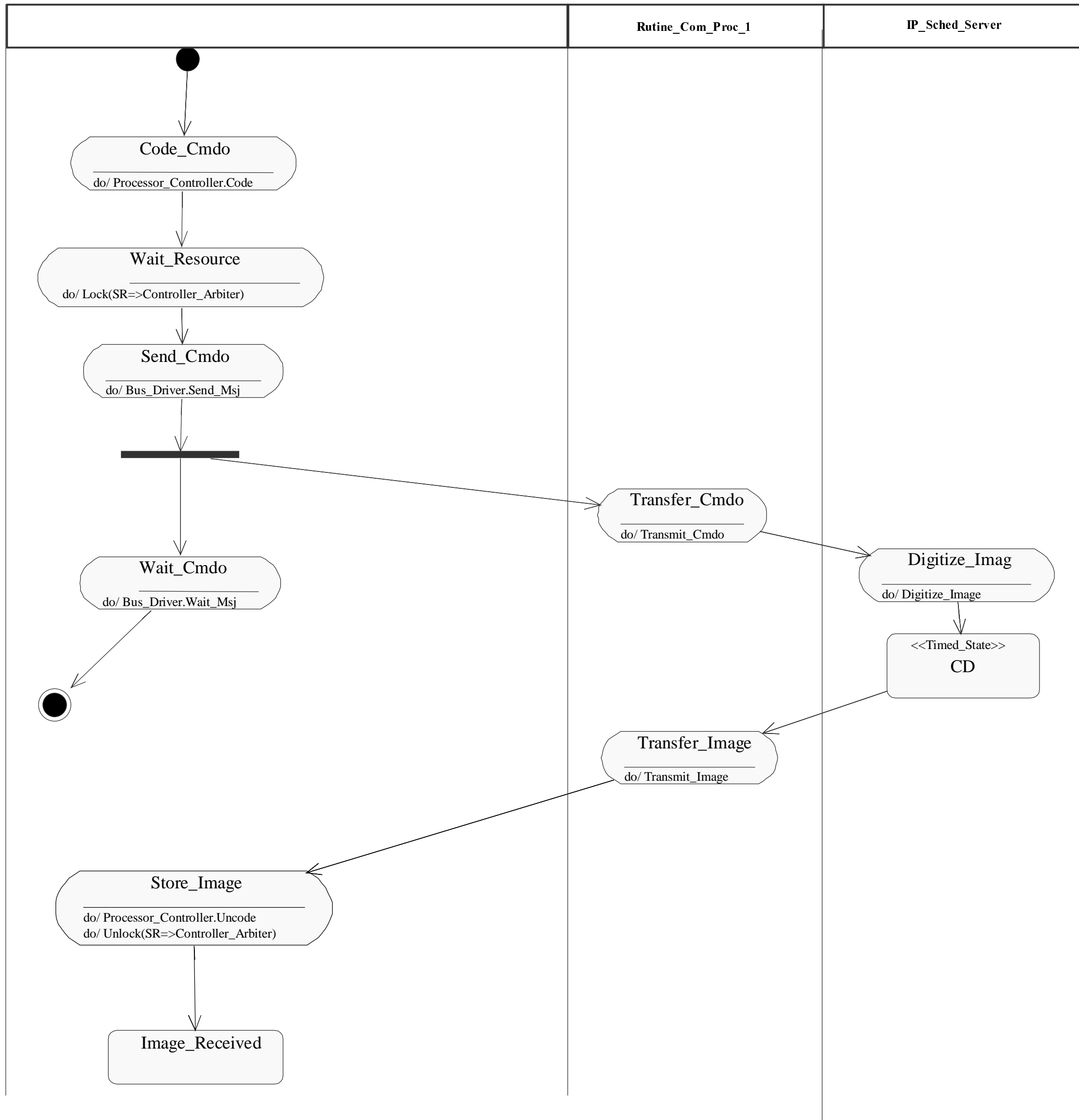

Figure 39.- The activity diagram describes the Processor_Controller.Capture job.

Image.Get_Result: Es un job ya que requiere sincronizar su actividad con la finalización de la actividad que realiza en análisis de la imagen. En la figura 35 se muestra su declaración agregada a la clase lógica Image y en la figura 37 se muestra su descripción.

La operación simple **Image.Get_Result.Get** describe la temporización del código de retorno de los resultados y está agregada al job Image.Get_Result ya que se declara como parte de la descripción del job.

A fin de tener una visión completa de como se despliega el modelo, en las siguientes figuras se muestra el modelo de tiempo real de las clases Processor_Controller y Bus_Driver que sirven de soporte de la clase Image, y del modelo de la operación Digitize_Image que es realizada por la funcionalidad de la tarjeta hardware Image_Processor.

En la figura 38 se muestra la declaración del job **Processor_Controller.Capture**, cuya descripción se muestra en la figura 39. Aspectos de interés de la descripción de este job son:

- Mediante la operación Lock(SR=>Controller_Arbiter) realizada al principio del job, se garantiza que una vez superada sea esta la única actividad que está haciendo uso (de forma exclusiva) del Image_Processor. Mediante la operación Unlock(SR=>Controller_ Arbiter) se libera el recurso para que otras actividades hagan uso de él.
- Observese el mecanismo de modelado de un Network. Las operaciones que proporciona el BusDriver (Bus_Driver.Send_Msj y Bus_Driver.Wait_Msj) son las de acceso y retorno de información que se ejecutan en el thread de la actividad que ejecuta la comunicación.
- Las cargas de overhead (planificación y polling de paquetes, gestión de protocolos, etc.) que el proceso de comunicación carga al procesador que comunica no son explícitas en este diagrama, se han introducido cuando se han declarado el Network Bus y sus drivers. En la declaración de job solo se explicita (Transmit_Cmdo y Transmit_Image) la carga (función del número de paquetes que se requieren para codificar la información que se transfiere) que la transmisión de los dos tipos de mensajes supone para el network Bus.
- El proceso de comunicación no es bloqueante. Ejecutado el envio del mensaje, el job queda suspendido a la espera de que llegue el mensaje de respuesta, y a su vez, retorna el control del thread a la actividad que sigue a la invocación del job.

La clase lógica Bus_Driver se modela con dos operaciones simples (**Bus_Driver.Send_Msg** y **Bus_Driver.Wait_Msj**) que simplemente describen la temporización del acceso al driver.

Las operaciones simple **Transmit_Cmdo** y **Transmit_Image** modelan el tiempo del network Bus que se requiere para transferir los dos mensajes, que por estar compuestos por un número diferente de paquetes, requeriran diferentes tiempos de transmisión.

La operación simple **Digitize_Image** modela la temporización de la operación de digitalización que realiza el hardware del Image_Processor.

II.3 Escenarios de Tiempo REAL.

Los Escenarios de Tiempo real (**Real-Time Scenarios**) son los modos o configuraciones de operación hardware/software del sistema para los que existen definidos requerimientos de tiempo real. Los escenarios representan las diferentes cargas de trabajo del sistema (workload) en las que deben satisfacerse los requerimientos de tiempo real.

Cada escenario de tiempo real se modela como un conjunto de Transacciones (**Transaction**). El conjunto de transacciones definidas dentro de un escenario describen la máxima carga posible del sistema de tiempo real en ese escenario. Cada Transacción describe la secuencia no iterativa de actividades que se desencadenan como respuesta a un patrón (pattern) 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 la actividades que requieren sincronización directa con ellas (intercambio de eventos, sincronización por invocación, etc.). No necesitan modelarse dentro de una transacción, otras actividades concurrentes que influyen en su evolución a través de competir con las actividades de la transacción por hacer uso de recursos comunes, ya sean recursos de procesamiento (processing_resource) o recursos compartidos (shared_resource) a los que se debe acceder con exclusión mútua (objetos protegidos, monitores, mutex, etc.). La presencia de estos recursos limitados que son compartidos por las actividades de todas las transacciones que coexisten concurrentemente dentro de un mismo escenario y que implican bloqueos y retrasos de sus actividades, es la causa de que el análisis de tiempo real deba hacerse contemplando simultáneamente todas las transacciones de un mismo escenario.

Aunque las transacciones son secuencias abiertas (no iterativas) de actividades, las transacciones 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.

Las transacciones de tiempo real son componentes básicos de la concepción y especificación de un sistema de tiempo real. Sin embargo, tienen su origen en diferentes fases del procesos de desarrollo de los sistemas de tiempo real:

- Algunas transacciones resultan de la formulación de los requerimientos temporales de las aplicaciones software descritas como gobernadas por eventos (Event Driven). Las especificaciones de tiempo real se plantean como secuencias de actividades que se desencadenan como respuesta a un patrón de eventos externos de entrada y restricciones temporales relativas a los instantes en que deben haber concluido estas activdades. Este tipo de transacciones suelen identificarse en la fase inicial de especificación del sistema, que en las metodologías basadas en su descripción UML se hace a partir de la formulación de los Casos de Uso.

- Otras transacciones resultan de requerimientos de tiempo real impuestos en la fase de diseño del sistema, como mecanismo interno para satisfacer requerimientos del sistema que en origen pueden no ser de tiempo real. Por ejemplo, si se diseña un sistema de control con ciertas especificaciones de precisión, anchura de banda y estabilidad, es habitual proponer un controlador PID basado en una tarea periódica, que introduce un

requerimiento de tiempo real no existente en la especificación original. Estas transacciones aparecen despues de la fase de diseño, y como consecuencia de decisiones tomadas en ella.

- En algunos sistemas de tiempo real no existen eventos externos y los requerimientos de tiempo real se establecen entre eventos internos. En estos casos no existen transacciones tal como han sido definidas, ya que no hay eventos externos que las generen. Sin embargo es frecuente introducir en estos casos eventos externos y transacciones que son meros artificios de análisis y que se introducen para garantizar los requerimientos temporales bajo condiciones de peor caso. Un ejemplo de este tipo de sistema es un software de monitorización cíclico, en el que se establece como requerimiento temporal que la duración del ciclo sea inferior a un tiempo dado. Este sistema se puede modelar como un sistema periódico con periodo igual a la duración máxima del ciclo, y se analiza si bajo esas condicions el sistema es planificable.

- Por último, hay transacciones que tienen que se introducidas para modelar actividades que concurren dentro del escenario de tiempo real, y que aunque en sí no contiene requerimientos de tiempo real, tienen que ser consideradas porque afectan a las actividades que si tienen requerimientos. Un ejemplo de este tipo de transacción es un módulo software sin requerimientos temporales pero que hace uso de un recurso compartido que también es requerido por transacciones con requerimientos de tiempo real. Otro ejemplo es el caso de un módulo sin requerimientos temporales definidos pero que debe responder a interrupciones hardware, y que en consecuencia tiene actividades que tienen que ser ejecutadas a prioridades superiores a las de actividades con requerimientos de tiempo real.

Aunque los cuatro tipos de transacciones tienen origen diferente, habitualmente convergen en su implementación, ya que el mecanismo mas utilizado para garantizar el comportamiento de tiempo real en estos sistemas es utilizar componentes hardware de tipo timer o clock que regulen la respuesta de tiempo real de forma sencilla y predecible, haciendola mas fácilmente garantizable y robusta frente a futuros cambios del sistema. Los diferentes tipos de transacciones se hacen coincidente si los eventos generados por el hardware de temporización se tratan como eventos externos.

Aunque en la vista de tiempo real de un sistema pueden definirse múltiples escenarios, cada escenario es independiente a efectos de su análisis. Todos ellos tienen en común, que su descripción se realiza compartiendo los componentes del modelo de plataforma y del modelo lógico del sistema, y por ello, resulta conveniente su descripción conjunta dentro de una misma estructura de datos.

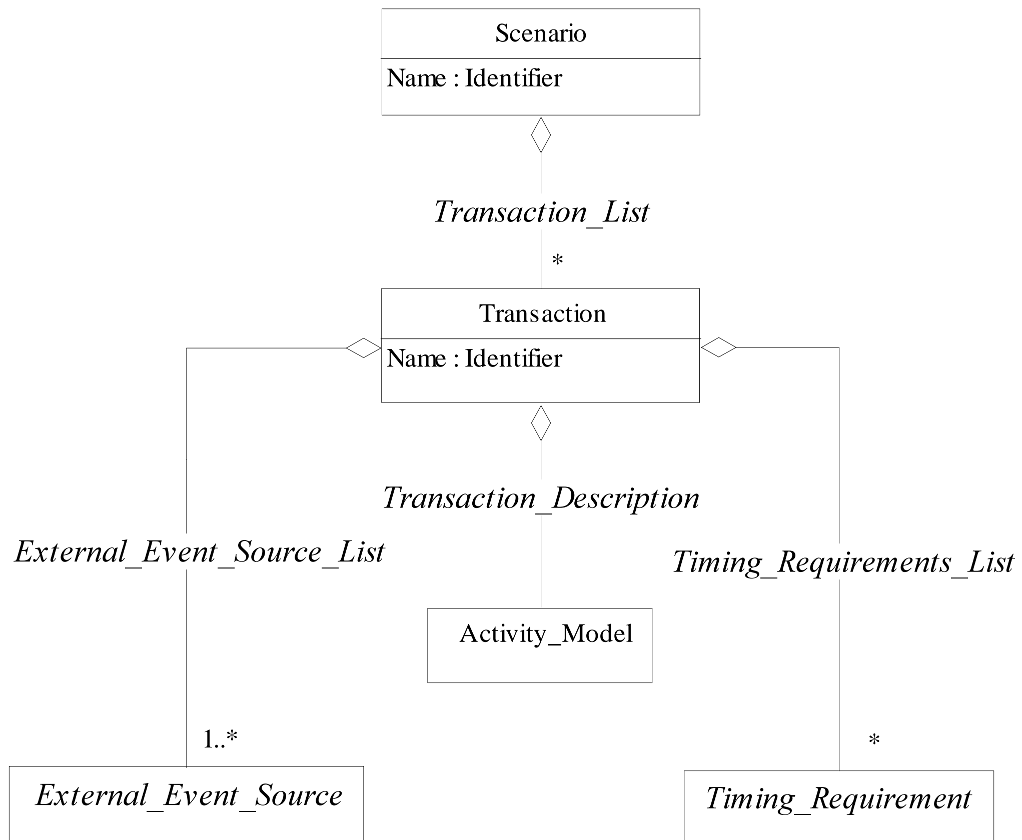

Figure 40.- Metamodel of Scenario class.

II.3.1 Componentes del modelo de tiempo real de un escenario.

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 de su descripción.

La declaración de una transacción (*Transaction*) se realiza mediante un objeto resultante de la instanciación de la clase Transaction del metamodelo, y que contiene agregadas una lista con la descripción de las fuentes de eventos externos (*External_Event_Source*) que constituyen su patrón de eventos externos de disparo y una lista con la descripción de los requerimientos temporales (*Timing_Requirement*) definidos en ella. La descripción de una transacción se realiza mediante un modelo de actividad agregado a su declaración.

Todos los componentes del modelo estático de tiempo real de una transacción son objetos derivados de las clases abstractas:

- *Transaction.*
- *External_Event_Source.*
- *Timing_Requirement.*

En el metamodelo la ***Transaction*** es una clase abstracta que representa una secuencia de actividades que se disparan como respuesta a un patrón de eventos de externos y que sirve de base para formular los requerimientos temporales en esa secuencia. Una transacción incluye todas las actividades interdependientes entre sí por transferencias de flujo o sincronización activa, cuyas líneas de control de flujo tienen su origen en la ocurrencia de los eventos externos que la disparan. Los conjuntos de actividades de las diferentes transacciones de un mismo escenario son disjuntos entre sí y entre las actividades de diferentes transacciones no se intercambian eventos de activación. Sin embargo, las actividades pertenecientes a diferentes transacciones de un mismo escenario compiten entre sí por recursos comunes, tanto del tipo Processing_Resource como del tipo Shared_Resource.

Cada objeto del tipo *Transaction* tiene agregada una lista (**External_Events_List**) con las fuentes de eventos externos que constituyen su patrón de disparo. La transacción se dispara cada vez que se produce un eveto individual de entrada, aunque en función de qué evento se produce o qué combinación de ellos se produce, las secuencias de actividades de la transacción que se activan pueden ser diferentes.

Así mismo, cada objeto de tipo *Transaction* tiene agregada una lista **(Timing_Requirements_ List)** de requerimientos temporales definidos en la transacción.

Una transacción se describe mediante un modelo de actividad agregado a su declaración. El modelo de actividad esta compuesto de un conjunto de diagramas de actividad que describen las secuencias de actividades, estados y transiciones que se desencadenan como consecuencia de un patrón de eventos externos de entrada.

El modelo de actividad (**Activity_Model**) de una transacción, ya fue definido en las figuras 19 y 23 como parte del metamodelo formulado para describir los Job. El modelo de actividad que describe transacciones tienen dos restricciones respecto al que se emplea para definir los Job:

- No contiene componentes Start_State y End_State ya que una transacción no es invocada y consecuentemente tampoco transfiere el flujo de control al thread que la invocó. Las líneas de flujo de actividades (thread) del diagrama de actividad, se inician en Wait_States relativos a eventos externos declarados en la transacción, y concluyen en Timed_States o Named_States, según tengan asociados o no requerimientos temporales.

- No utiliza parámetros, esto es, todos los componentes que se referencian en el modelo son objetos concretos ya declarados en otras secciones de la vista de tiempo real, y todos los valores de los parámetros que se utilizan en la invocación de las Duty, son valores numéricos concretos o identificadores de objetos concretos ya declarados en otras secciones de la vista de tiempo real.

La clase concreta **Regular_Transaction** representa una transacción analizable mediante las herramientas Mast. Las restricciones que debe satisfacer la descripción de la Regular_Transaction son:

a) Cada transacción debe tener asociado, al menos, un evento externo.

b) En la transacción deben estar incluidas todas las actividades que sean activadas en las líneas de flujo de control que tienen su origen en las fuentes de eventos externos de la transacción.

c) En la transacción deben estar incluidas todas la actividades que estén directamente sincronizadas (por invocación, por transferencia de eventos, o por componentes activos de sincronización) con actividades de la transacción. Esta característica hace que los conjuntos de actividades de las diferentes transacciones de un escenario sean disjuntos.

d) No pueden existir dependencias circulares de activación, esto es, no pueden existir bucles en los diagramas de actividad que describen una transacción.

e) Todos los recursos compartidos reservados en la transacción deben ser liberados en ella.

f) No pueden converger en dos ramas de un Merge_Control lineas de flujo de control activadas por diferentes External_Event_Source.

g) No pueden utilizarse componentes de la clase Rate_Divisor en líneas de flujo de control resultantes de la unión (join) de líneas de flujo de control procedentes de diferentes External_Event_Source.

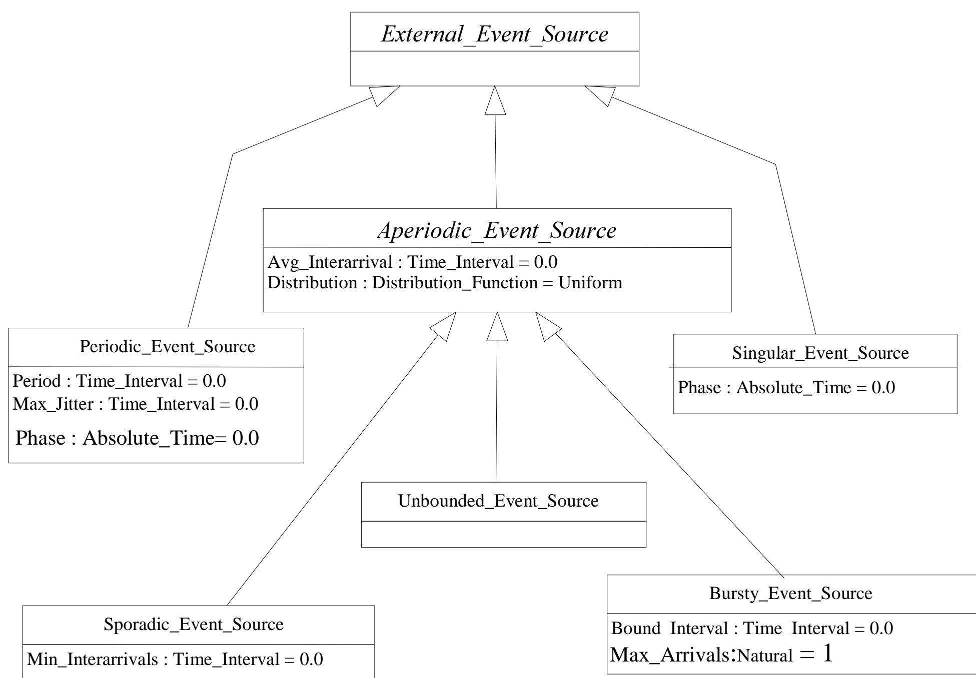

Figure 41.- Metamodel of External_Event_Source class.

La clase abstracta ***External_Event_Source*** representa una secuencia (stream) de eventos que tienen su origen externo al código del sistema. Pueden ser originados por el entorno o por componentes hardware del sistema (timer, etc). Lo específico de su patrón de generación es que es autónomo y no depende del estado o de la evolución del sistema que se está modelando.

La clase concreta **Periodic_Event_Source** representa una secuencia con un patrón de generación aproximadamente periódico. Está caracterizado por su periodo (**Period**), su máximo jitter (**Max_Jitter**) y el tiempo de generación del primer evento (**Phase**).

La clase abstracta ***Aperiodic_Event_Source*** representa la generación de una serie de eventos aperiódicos caracterizados por el tiempo promedio entre eventos consecutivos (**Avg_Interarrival**) y una función de distribución de estos tiempos (**Distribution_Function**), que en la versión actual puede definirse como **Uniform** o **Poisson**.

Dentro de la clase ***Aperiodic_Event_Source*** se definen tres clases especializadas de eventos aperiódicos: **Sporadic_Event_Source** en la que existe establecido un mínimo tiempo entre eventos (**Min_Interarriva**l), **Unbounded_Event_Source** en la que tal mínimo no existe y **Bursty_Event_Source** que es una fuente de eventos a ráfagas en la que los eventos se presenta en grupos caracterizados por un tiempo mínimo entre rafagas (**Bound_Interval**) y el número máximo de eventos en cada grupo (**Max_Arrivals**).

Por último se define la fuente de evento singulares (**Singular_Event_Source**) que representa la generación de un único evento, caracterizado por el instante en que se produce (**Phase**) y que se suele utilizar para describir el instante en que se produce una situación especial, tal como arranque, cambio de modo, etc.

La clase abstracta ***Timing_Requirement*** representa el requerimiento temporal que se asocia a un estado (State_Action) dentro del contexto de una transacción. Puede ser simple (***Simple_Timing_Requirement***) o compuesto (**Composite_Timing_Requirement**) si consiste de un conjunto de requerimientos simples que se aplican a un mismo estado. Cada requerimiento temporal es siempre relativo a un evento externo o a una transición que se encuentra dentro de la línea directa de flujo que ha provocado la transición a la que se asocia el requerimiento temporal. Cuando una transición puede resultar como unión (join) de diferentes líneas de flujos de control con origen en diferentes fuentes de eventos externos, la restricción temporal asociada a ella solo se aplica a los eventos originados por el evento externo a los que hace referencia el requerimiento temporal.

Los diferentes requerimientos temporales de un requerimiento compuesto se aplican con criterio conjuntivo (and) si son relativos a un mismo evento, y con caracter disyuntivo (or) si son relativos a diferentes eventos.

Los requerimientos temporales pueden referirse al plazo en el que debe alcanzarse el estado al que están asociados (***Deadline_Timing_Requirement***) o al jitter de los tiempos en que se alcanza ese estado (**Max_Output_Jitter_Req**).

Los requerimientos temporales pueden referirse a un evento externo (**Referenced_Event**) y en tal caso se denominan globales (***Global_Timing_Requiremet***) o se refieren a la duración de la

actividad a cuyo estado de terminación se asocia el requerimiento caso en el que los denominamos locales (***Local_Timing_Requiremet***). En ambos casos, se definen tres tipos de requerimientos concretos: de plazos estrictos **Hard_Global_Deadline** y **Hard_Local_Deadline** , de plazos laxos **Soft_Global_Deadline** y **Soft_Local_Deadline**, o de plazos laxos con tanto por ciento de fallos limitado **Global_Max_Miss_Ratio** y **Local_Max_Miss_Ratio.**

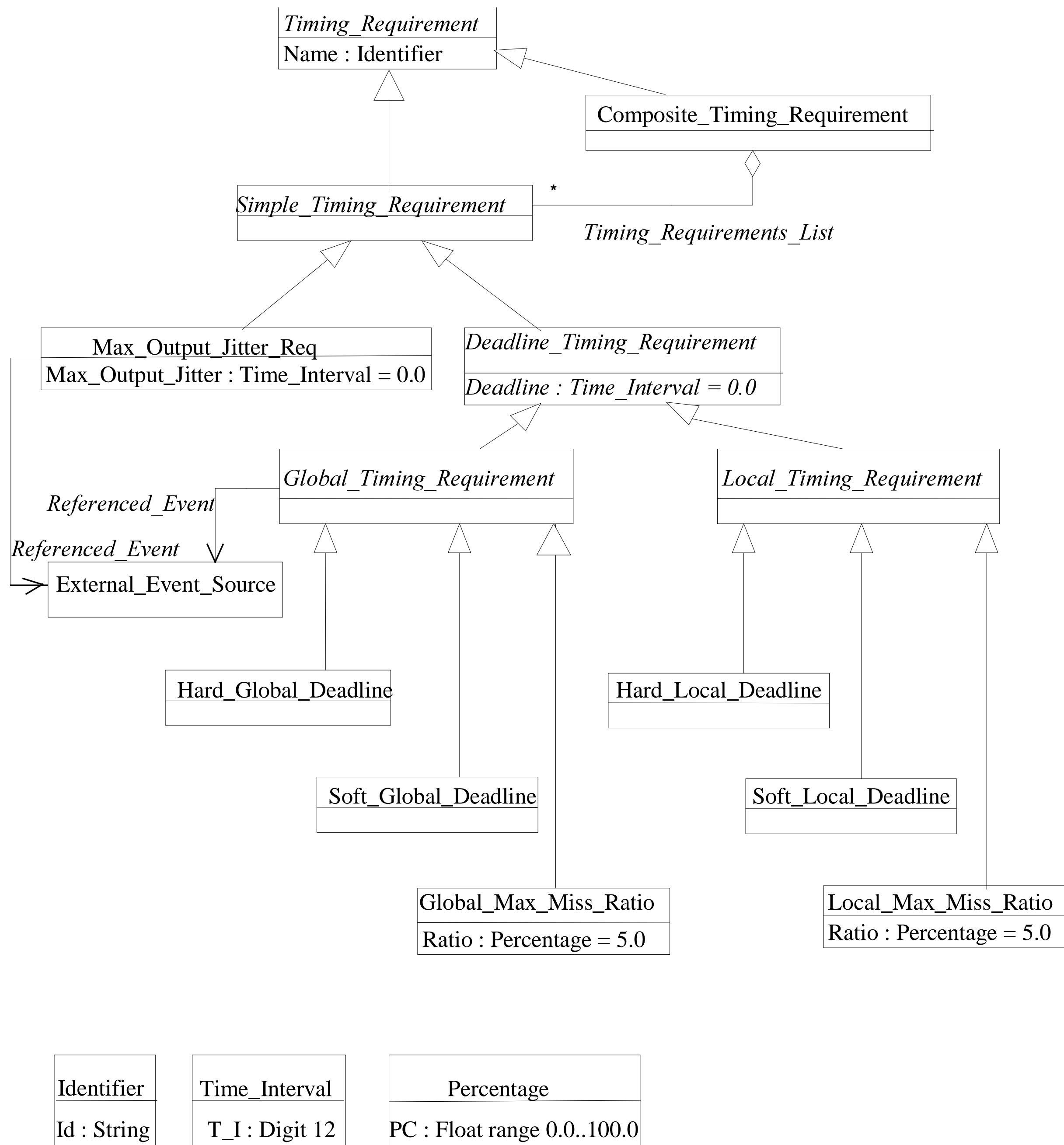

Figure 42.- Metamodel of Timing_Requirement class.

II.3.2 Transacciones en la herramienta CASE UML.

El package Process View.Mast_RT_View.RT_Scenarios_Model contendrá los escenarios de tiempo real. Dentro de él y por cada escenario se define un package específico. Los diferentes escenarios de un sistema son independientes a efecto de su tratamiento por las herramientas. Obviamente todos ellos corresponden a diferentes modos o configuraciones de un mismo sistema, y por tanto comparten muchos componentes del modelo de la plataforma y del modelo lógico.

El modelo de un escenario está compuesto por un conjunto de diagramas de clases en los que se declaran las transacciones que en su conjunto describen el escenario. Cada transacción se declara mediante una clase que corresponde a una instanciación de la clase Transaction del metamodelo y asociados con ella se tedrán un conjunto de objetos que son instanciaciones de alguna de las clases derivadas de External_Event_Source del metamodelo y que representan los eventos externos definidos en la transacción y un conjunto de objetos que son instanciaciones de las clases derivadas de la clase Timing_Requirement del metamodelo y que representan requerimientos temporales establecidos en la transacción. Así mismo cada clase que declara una transacción tiene que tener agregado un modelo de actividad que la describe.

Normas en la formulación del Modelo de un Escenario de Tiempo Real:

Estas normas no son relevantes desde el punto de vista conceptual, sino que son requeridas por las herramientas de extracción del modelo.

1) Son ignorados cualquier clase u objeto que se introduzca en los diagramas del modelo y que no corresponda a instanciaciones del metamodelo UML_Mast. En un diagrama de clases correspondiente a la descripción de un escenario solo son considerados:

    - Clases derivadas del tipo Transaction.
    - Clases derivadas del tipo genérico External_Event_Source que tengan establecido un enlace con una clase derivada de Transaction.
    - Clases derivadas del tipo genérico Timing_Requirement que tengan establecido un enlace con una clase derivada de Transaction.

2) Los roles no son relevantes y no se analizan en la declaración de las transacciones. Se pueden incluir si se consideran útiles a efecto de documentación, pero no son tenidos en cuenta.

3) Es optativo introducir el tipo de los atributos, pero si se introducen deben ser correctos.

4) Los esterotipos definidos en el modelo Mast_UML son:

    Estereotipos de Clases:
    - <<Regular_Transaction>> -- Declara una transacción
    - <<Periodic_Event_Source>> -- Declara un tipo de evento externo.
    - <<Singular_Event_Source>> -- Declara un tipo de evento externo.
    - <<Sporadic_Event_Source>> -- Declara un tipo de evento externo.

- <<Unbounded_Event_Source>> -- Declara un tipo de evento externo.
- <<Bursty_Event_Source>> -- Declara un tipo de evento externo.
- <<Hard_Global_Deadline>> -- Declara un req. temporal simple.
- <<Soft_Global_Deadline>> -- Declara un req. temporal simple.
- <<Global_Max_Miss_Ratio>> -- Declara un req. temporal simple.
- <<Hard_Local_Deadline>> -- Declara un req. temporal simple.
- <<Soft_Local_Deadline>> -- Declara un req. temporal simple.
- <<Local_Max_Miss_Ratio>> -- Declara un req. temporal simple.
- <<Max_Output_Jitter_Req>> -- Declara un req. temporal simple.
- <<Composite_Timing_Req>> -- Declara un req. temporal compuesto

<u>Estereotipos en Activity_Model:</u>

- <<Activity>> [*] -- Invoca una Activity
- <<Timed_Activity>> -- Invoca una Timed_Activate Activity
- <<Delay>> -- Invoca un Delay
- <<Offset>> -- Invoca in Offset
- <<Rate_Divisor>> -- Invoca un Rate_Divisor
- <<Priority_Queue>> -- Declara una Queue
- <<Fifo_Queue>> -- Declara una Queue
- <<Lifo_Queue>> -- Declara una Queue
- <<Scan_Queue>> -- Declara una Queue

<u>Estereotipos de State_Action:</u>

- <<Wait_State>> -- Declara un Wait_State
- <<Timed_State>> -- Declara un Timed_State
- <<Named_State>> -- Declara un Named_State

<u>Estereotipos de Control_Activity:</u>

- <<Random_Branch>>[*] -- Declara una Branch Control
- <<Scan_Branch>> -- Declara una Branch Control

[*] El estereotipo es optativo, equivale a no estar establecido.

6) <u>Valores por defecto:</u>

- Todos los atributos de las clases del metamodelo tienen valores por defecto. Estos valores se muestran en el metamodelo. Si en un objeto del diagrama no se establece valor del atributo en el modelo, se considera que tiene su valor por defecto.

7) <u>Son situaciones de error:</u>

- Una clase de tipo Transaction que no tenga enlazado al menos una clase derivada de External_Event_Source.
- Una clase derivada de la clase External_Event_Source que esté enlazada con mas de una clase tipo Transaction.
- Una clase derivada de la clase Timing_Requirement que esté enlazada con mas de una clase tipo Transaction.
- Una clase de tipo Transaction que no tenga definido un modelo de actividad.
- Un Swimlane del modelo de actividad que no corresponda a un Scheduling_Server declarado en el modelo de la plataforma.
- No existe ningún Wait_State relativo a un External_Event_Source.
- Se asigna un mismo Timing_Requirement a más de un Timed_State.
- Se referencia en un Wait_State a un Timed_State.

- Se referencia en un Wait_State a un Named_State que ya ha sido referenciado por otro Wait_State.
- Una clase derivada de Global_Timing_Requirement o un Max_Output_Jitter_Req que no tenga asociada una y sólo una clase derivada de External_Event_Source.

8) Son situaciones que dan origen a un warning:
   - Una clase derivada de la clase External_Event_Source que no esté enlazada con ninguna clase del tipo Transaction.
   - Una clase derivada de la clase Timing_Requirement que no esté enlazada con una clase del tipo Transaction ni con una del tipo Composite_Timing_Requirement.

Normas de adaptación a la Herramienta ROSE'2000.

Dado que esta herramienta no permite añadir una nueva vista a un modelo, el modelo de un escenario se ubica en el package Logical_View.Mast_RT_View.RT_Scenarios_Model.

Los componentes se representan mediante símbolos de clase, a través del estereotipo de estas clases se explicita la clase del metamodelo de las que son instanciación.

Ejemplo:

En la figura 38 se muestra la estructura de clases lógicas que constituyen el sistema, y en la figura 39 se muestra el diagrama de sequencia que describe la operación básica del sistema.

El sistema adquiere la imagen de cada una de las dos caras de la placa cuando esta se sitúa bajo las cámaras de visión. Las imágenes son procesadas para identificar los defectos superficiales que existan en ellas. Con esta información se elaboran las consignas para que un robot marque los defectos cuando la placa se encuentre dentro de su región de alcance. El sistema opera en pipeline, así que varias placas pueden estar siendo analizadas y en espera de ser marcadas.

Modelo de tiempo real

En la figura 40 se muestra la declaración de la transacción ***Plate_Inspection*** *que modela la operación y los requerimientos de tiempo real del sistema. En la figura 41, se muestra el diagrama de actividad que describe el Activity_Model de la transacción.*

La transacción se dispara con un evento externo:

> ***Plate_Placed****: representa que una placa se ha situado bajo el sistema de visión. Es un evento periódico y tiene un periodo de 2 segundos que es la cadencia con que se presentan las placas por la cadena de producción.*

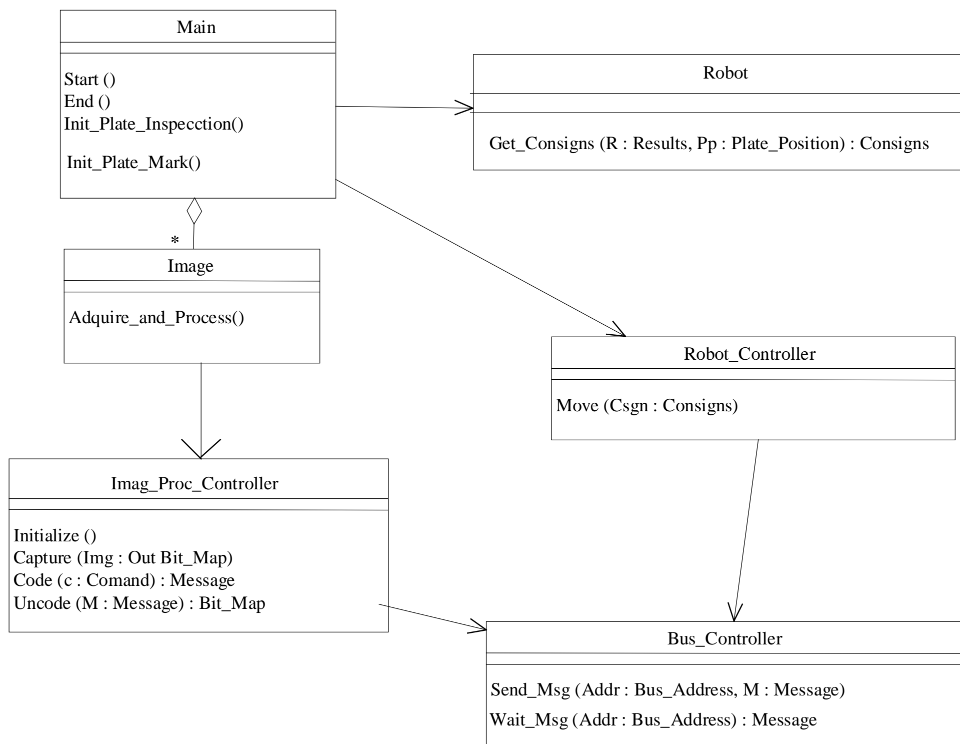

Figure 43.- Logical Components of the Plate Inspection System.

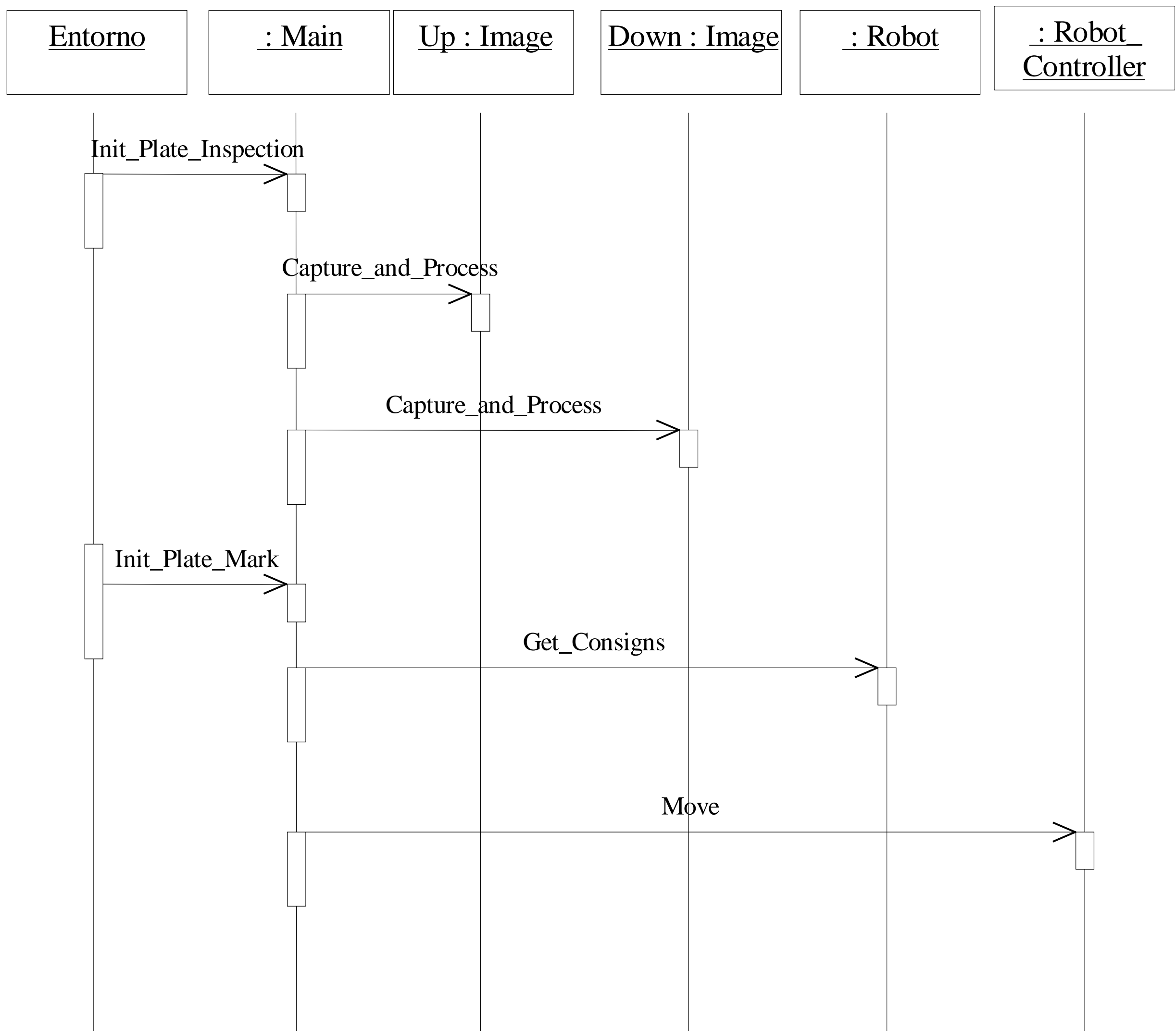

Figure 44.- Squence diagram of plate inspection process.

La transacción tiene tres requerimientos temporales:

***Face_Up_Captured**: Se requiere que la imagen de la cara superior sea adquirida mientras que la placa está bajo el campo de visión de la cámara. Es un Global_Hard_Deadline con un plazo de 100 ms.*

***Face_Down_Captured**: Se requiere que la imagen de la cara inferior sea adquirida mientras que la placa está bajo el campo de visión de la cámara. Es un Global_Hard_Deadline con un plazo de 100 ms.*

***Plate_Marked**: Se requiere que la placa haya sido marcada mientras que se encuentre dentro del campo de alcanzabilidad del robot. Es un Global_Hard_Deadline con un plazo de 8 s.*

Como se muestra en la descripción de la transacción de la figura 41, esta se inicia cuando se recibe un evento Plate_Placed. Se aquieren sucesivamente las imágenes de las dos caras de las placas con el thread activado por el evento externo y adquirida cada imagen se deja a un thread agente que procese las imágenes en concurrencia con la adquisición de las imágenes de las siguientes placas. La captura de las imágenes de las caras debe realizarse antes de los plazos Face_Up_Captured y Face_Down_Captured.

Tras ser procesadas las dos imágenes de una misma placa, se elaboran las consignas de marcado de la placa en un nuevo thread.

Una vez que han trancurrido los 4 segundos que tarda en pasar la placa desde la zona de las cámaras a la zona de marcado, y siempre que se haya concluido la elaboración de las consignas, se transfieren estas al robot, que debe marcar la placa antes del plazo Plate_Marked.

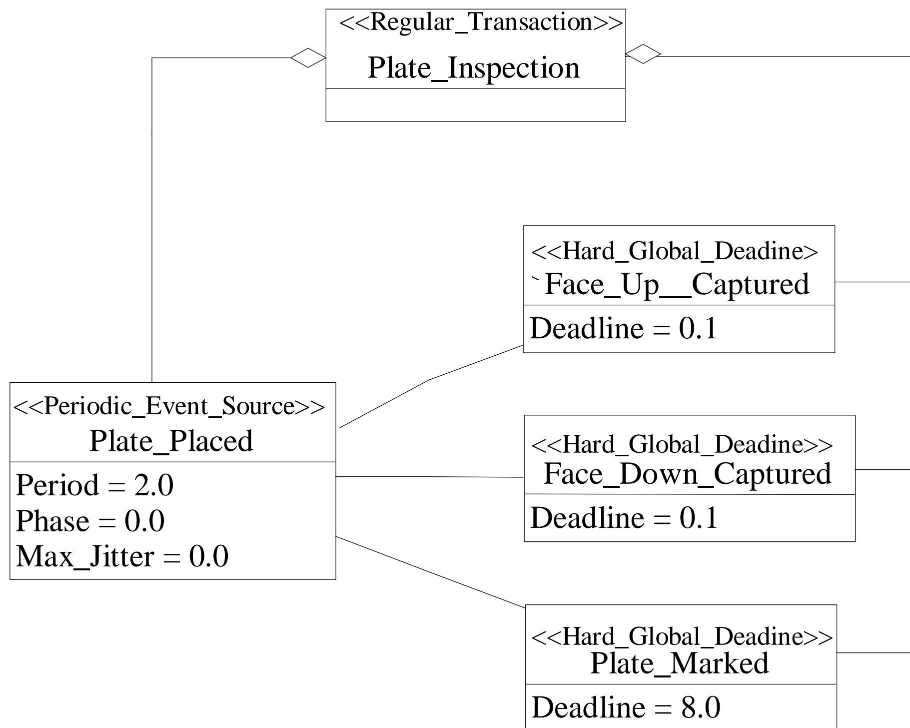

Figure 45.- Declaration of Plate_Inspection transaction.

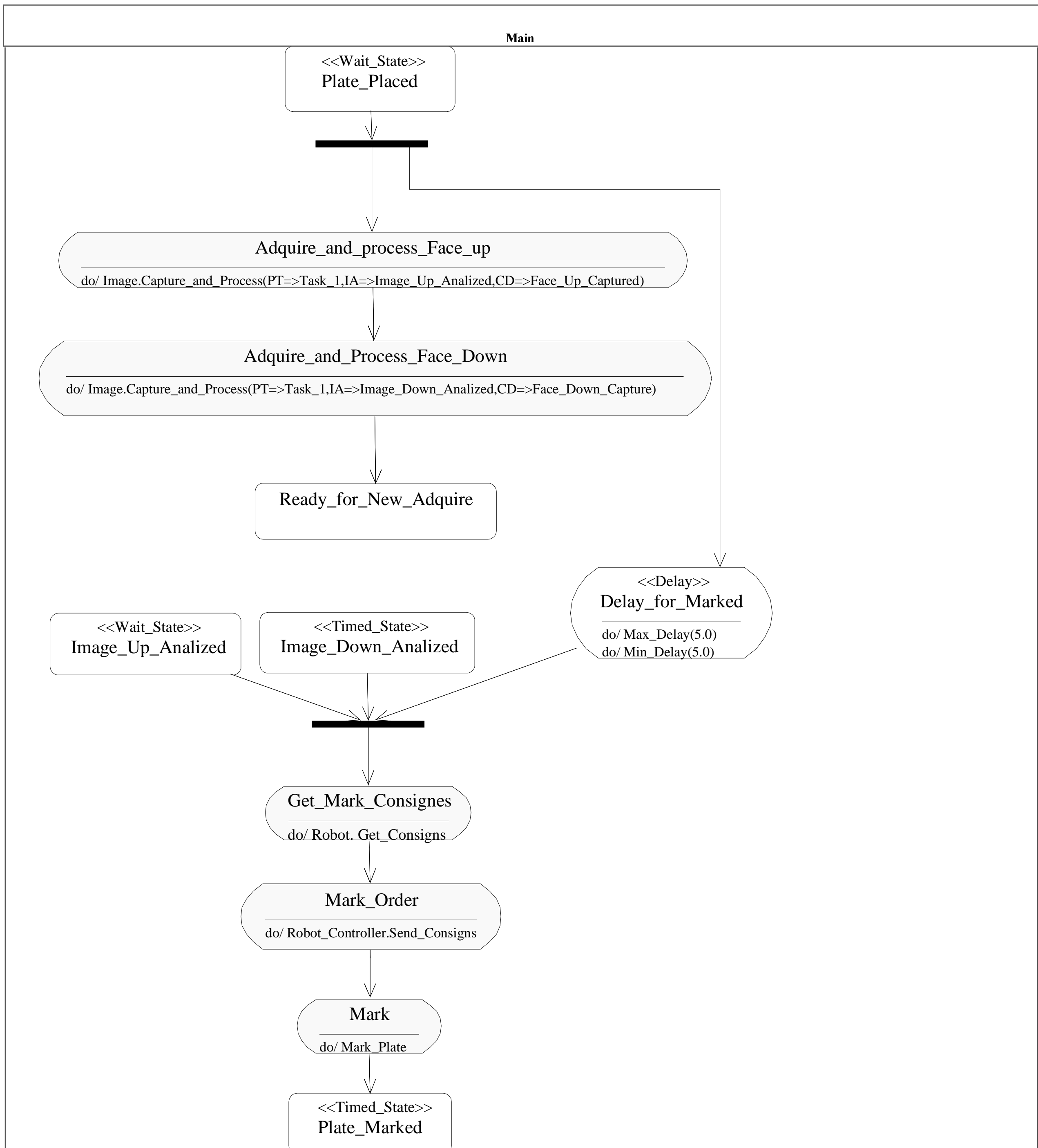

Figure 46.- Description of Plate_Inspection transaction.