Universidad de Cantabria
Departamento de Electrónica y Computadores

Desarrollo de sistemas de tiempo real basados en componentes utilizando modelos de comportamiento reactivos

Tesis Doctoral
Patricia López Martínez
Santander, junio 2010

Universidad de Cantabria
Departamento de Electrónica y Computadores

Desarrollo de sistemas de tiempo real basados en componentes utilizando modelos de comportamiento reactivos

Memoria
presentada para optar al grado de
DOCTOR por la Universidad de Cantabria por
Patricia López Martínez
Licenciada en Ciencias (Físicas)

Universidad de Cantabria
Departamento de Electrónica y Computadores

Desarrollo de sistemas de tiempo real basados en componentes utilizando modelos de comportamiento reactivos

Memoria
presentada para optar al grado de Doctor por la Universidad de Cantabria, dentro del programa de doctorado Electrónica y Computadores, por la Licenciada en Ciencias
Patricia López Martínez

El Director,

Dr. José María Drake Moyano
Catedrático de Universidad

Declaro:
Que el presente trabajo ha sido realizado en el Departamento de Electrónica y Computadores de la Universidad de Cantabria, bajo mi dirección y reúne las condiciones exigidas para la defensa de tesis doctorales.

Santander, junio de 2010

Fdo. Patricia López Martínez

Fdo. José María Drake Moyano

Estas palabras son una forma de dar las gracias a todos los que de una manera u otra me han ayudado a llegar hasta aquí.

En primer lugar, mi más sincero agradecimiento a José María Drake, por su excelente labor como director de esta tesis, sin la que sin duda no hubiese sido capaz de terminar este trabajo. A lo largo de estos años debo agradecerle muchas cosas, desde la oportunidad de entrar a formar parte de este grupo, hasta la confianza depositada en mí a lo largo de los años, pasando por esas largas discusiones de las que siempre he sacado algo de provecho y que espero que continúen de ahora en adelante.

A todos los compañeros del grupo de Computadores y Tiempo Real, a los de los viejos tiempos y a los de ahora, por crear un ambiente tan bueno de trabajo

Y a mis amigas, por estar ahí siempre, y conseguir tan fácilmente que me cambiase el chip en días en los que el trabajo podía conmigo.

Este trabajo está dedicado a mis padres, y a mi tía, por todo su apoyo y por todas las veces que me han preguntado por él y no han recibido respuestas muy claras, muchas veces ni tan siquiera respuesta. Y muy especialmente a mi hermana, que aunque no entendiese muy bien eso de ser doctor sin haber estudiado medicina, siempre ha estado pendiente de cuánto me faltaba para acabar el “test”, como ella lo llama.

Índice de contenidos

Lista de figuras

Lista de abreviaturas

CAN:	Controller Area Network
CBD:	Component-based Development
CBSE:	Component-based Software Engineering
CCM:	CORBA Component Model
CORBA:	Common Object Request Broker Architecture
D&C:	OMG's Deployment and Configuration of Component-based Distributed Applications Specification
EDF:	Earlier Deadline First
IDL:	Interface Definition Language
LwCCM:	Lightweigh CORBA Component Model
MAST:	Modeling and Analysis Suite for Real-Time Applications
MARTE:	UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems
MaRTE OS:	Minimal Real-Time Operating System for Embedded Applications
MDA:	Model Driven Architecture
MDE:	Model Driven Engineering
M2M:	Model To Model Transformation
M2T:	Model To Text Transformation
OMG:	Object Management Group
PIM:	Platform Independent Model
PSM:	Platform Specific Model
RMA:	Rate Monotonic Analysis
RT-EP:	Real-time Ethernet Protocol
RT-CCM:	Real-Time Container Component Model
RT-D&C:	Real-Time Deployment and Configuration of Component-based Distributed Applications
SCADA:	Scada Supervisory and Data Acquisition
T2M:	Text To Model Transformation
UML:	Unified Modeling Language
XML:	Extensible Markup Language
WCET:	Worst Case Execution Time
W3C:	World Wide WEB Consortium

Resumen

El objetivo principal de esta tesis es la definición de una metodología de desarrollo de aplicaciones de tiempo real basadas en componentes, orientada a aplicaciones cuyos requisitos temporales se especifican utilizando un modelo reactivo de comportamiento temporal. La metodología definida permite analizar, y en su caso certificar, el cumplimiento de los requisitos de tiempo real en aplicaciones que se construyen ensamblando componentes software reutilizables, esto es, componentes que han sido previamente diseñados con independencia de las aplicaciones en que van a ser utilizados, y que además son manejados de forma opaca, sin tener acceso a los detalles de su código. La metodología definida incluye tres contribuciones complementarias, que se introducen a continuación.

En primer lugar, se define una metodología de modelado que facilita la construcción del modelo de tiempo real de aplicaciones basadas en componentes, al ofrecer elementos de modelado de alto nivel de abstracción y orientados a la composición. Con ella, se puede formular el modelo de comportamiento temporal de un componente de forma parametrizada y reutilizable, así como obtener el modelo que describe el comportamiento temporal de una aplicación por composición de los modelos de los componentes que la forman, del modelo de la plataforma de ejecución y de la descripción reactiva de la carga de trabajo que soporta la aplicación en el modo de operación que se analiza.

En segundo lugar se define RT-D&C, una extensión de la especificación *Deployment and Configuration of Component-based Distributed Applications* de OMG para la especificación de componentes y aplicaciones de tiempo real. La extensión incorpora metadatos relativos al comportamiento temporal de los componentes, que otorgan al diseñador de una aplicación capacidad para construir su modelo de tiempo real sin necesidad de conocer el código de los componentes que la forman. Asimismo, en base a los resultados obtenidos del análisis de este modelo, la extensión permite al diseñador configurar las instancias que forman la aplicación para que ésta verifique sus requisitos temporales cuando es ejecutada.

En tercer lugar, se propone la tecnología de componentes RT-CCM (*Real-time Container Component Model*), dotada con los recursos necesarios para que los componentes elaborados de acuerdo a ella tengan comportamiento temporal predecible. La tecnología propuesta utiliza el modelo de programación contenedor-componente, ubicando en el contenedor los recursos que controlan la planificación de la ejecución de las aplicaciones, a fin de mantener la opacidad y la reutilización del código de negocio de los componentes.

Todos estos elementos se han integrado en un proceso de diseño de tiempo real de aplicaciones basadas en componentes. Para cada fase del proceso se han identificado las actividades que se llevan a cabo, la información que se gestiona y los agentes responsables.

El proceso de diseño, junto a las metodologías y tecnologías propuestas, se han validado a través de Ada-CCM, que constituye una implementación concreta de la tecnología RT-CCM, formulada enteramente en lenguaje Ada 2005.

1. Componentes y tiempo real

A medida que la complejidad de los sistemas informáticos aumenta, y simultáneamente se demandan tiempos de desarrollo más cortos, se hace necesario introducir nuevas estrategias y procesos de desarrollo de software que hagan compatibles estos dos aspectos. El paradigma de componentes es considerado por la industria actual como una de las soluciones más eficientes para abordar esta complejidad y adaptarla a la velocidad de evolución del mercado. Cuando se utiliza una metodología basada en componentes, las aplicaciones se construyen ensamblando módulos software opacos que han sido desarrollados por terceros con independencia de la aplicación en la que van a ser utilizados. En muchos campos de la informática el paradigma de componentes está plenamente establecido, sin embargo, en sistemas de tiempo real su uso es aún muy poco frecuente, ya que los procesos y las estrategias propias de las metodologías orientadas a componentes no son compatibles con los métodos que se han utilizado tradicionalmente en el diseño de tiempo real. Para hacer factible su implantación en entornos de tiempo real es necesario desarrollar una metodología completa que permita asegurar un comportamiento temporal predecible de las aplicaciones generadas por composición de componentes. Este es el objetivo principal de esta tesis. En este capítulo se plantean los problemas que aparecen cuando se desarrollan sistemas de tiempo real aplicando metodologías basadas en componentes, se analizan los principales trabajos que han sido desarrollados por otros grupos y que constituyen los antecedentes de éste, y por último, se describen los objetivos concretos que se han planteado en la tesis.

1.1. Sistemas de tiempo real

Los sistemas de tiempo real son sistemas informáticos que debido a su naturaleza o funcionalidad interaccionan continuamente con un entorno externo que evoluciona dinámicamente en el tiempo físico. Son por tanto sistemas reactivos, que deben generar respuestas con restricciones temporales a los eventos que reciben del entorno, y que asimismo, pueden tener que generar eventos y acciones hacia el entorno en instantes específicos del tiempo. Debido a ello, su correcto funcionamiento se produce no sólo cuando realizan las acciones correctas o generan los datos especificados por su funcionalidad, sino también cuando las generan en los tiempos adecuados [SR88]. Muy frecuentemente los sistemas de tiempo real son también empotrados, esto es, sistemas computacionales que forman parte de un sistema mayor, y que controlan o supervisan alguna de sus funciones internas. Ejemplos de sistemas de este tipo van desde pequeños controladores de electrodomésticos hasta complejos sistemas de control de vehículos espaciales.

En la especificación de un sistema de tiempo real se puede asociar un plazo temporal máximo para la ejecución de cada una de las acciones que el sistema realiza. Atendiendo al nivel de tolerancia que se acepta en el incumplimiento de dichos plazos temporales, los sistemas de tiempo real se clasifican en sistemas estrictos (*hard real-time*) y laxos (*soft real-time*). En los sistemas de tiempo real estricto, un plazo perdido representa un error fatal e irrecuperable, y por tanto, nunca es admitido. Por el contrario, en el caso de sistemas de tiempo real laxos, la pérdida de un plazo temporal no constituye necesariamente un error fatal. En estos casos las

restricciones temporales se formulan como tasas de fallos que deben satisfacerse estadísticamente. Por supuesto, también existen sistemas en los que se mezclan requisitos de ambos tipos. Es más, en la actualidad y debido al continuo aumento de la capacidad de las plataformas y de los mecanismos de comunicación, las aplicaciones de tiempo real que se desarrollan son cada vez más complejas, y en general, tienden a mezclar actividades que deben realizarse en régimen de tiempo real estricto, con otras de tipo laxo, que requieren niveles de calidad de servicio o incluso que no imponen ningún tipo de requisito temporal [HMN08].

La característica más importante de un sistema de tiempo real es su predictibilidad: el sistema debe ofrecer un comportamiento temporal conocido que permita certificar el cumplimiento de los plazos temporales asociados a las respuestas del sistema [STA88]. El término sistema de tiempo real generalmente se utiliza para referirse a un sistema completo, que incluye tanto la aplicación software, como la plataforma (dispositivos hardware, sistema operativo y servicios de comunicaciones) en la que la aplicación se ejecuta. Para poder ejecutar una aplicación de tiempo real con la garantía del cumplimiento de sus requisitos temporales no sólo es necesario que su código tenga un comportamiento temporal predecible, sino también que la plataforma de ejecución proporcione servicios con tiempos de respuesta acotados y conocidos.

1.1.1. Estrategias de diseño de sistemas de tiempo real

El diseño de un sistema de tiempo real es un proceso complejo, ya que hay que planificar explícita o implícitamente la ejecución de las actividades que se realizan en él, o que en el peor caso se podrían realizar, de forma que todas ellas finalicen dentro de los plazos que tienen especificados. En sistemas sencillos, o en aquellos que requieren un ajuste muy fino del uso de la capacidad de procesamiento, se pueden aplicar mecanismos de planificación estática, como el ejecutivo cíclico [BS88]. Este mecanismo se basa en la elaboración de una tabla, que se recorre de manera cíclica durante la ejecución, en la que explícitamente se definen los instantes de tiempo en que debe ejecutarse cada tarea de la aplicación. Con esta estrategia el nivel de control de la ejecución por parte del diseñador es máximo, pero da lugar a aplicaciones muy poco flexibles, que requieren un costoso rediseño cada vez que se modifica un requisito de su especificación.

Actualmente los sistemas de tiempo real se suelen diseñar como sistemas concurrentes, en los que la ejecución de cada tarea y los accesos a los recursos compartidos se realizan utilizando políticas de planificación y de sincronización dinámicas, aplicadas en tiempo de ejecución. Su utilización da lugar a una ejecución suficientemente predecible como para que mediante un análisis adecuado se pueda garantizar que se satisfacen las restricciones temporales especificadas. En este caso, el proceso de diseño de tiempo real consiste en descomponer el sistema en un conjunto de tareas que van a ser ejecutadas concurrentemente al ser asignadas a líneas de flujo (threads) distintos, y en establecer para cada una de ellas los parámetros de planificación adecuados para que en cualquiera de las posibles formas de ejecución que puedan producirse, siempre se satisfagan las restricciones temporales asignadas.

Las estrategias de diseño han ido evolucionando en las últimas décadas, casi siempre de forma paralela a las metodologías de desarrollo de software en uso. Inicialmente el diseño de tiempo real se basó en estrategias de programación estructurada. En ellas, se utilizan criterios estrictamente funcionales para descomponer el sistema en un conjunto de subsistemas, cada uno encargado de una determinada función. Posteriormente cada subsistema es descompuesto en un conjunto de tareas concurrentes que ejecutan las actividades que requiere su funcionalidad, y finalmente, se elabora el código de cada tarea aplicando de nuevo criterios estructurados. El aspecto clave de esta estrategia es el criterio que se aplica para la identificación de las tareas con

las que se construye cada subsistema, para lo cual se analizan las funciones que pueden ejecutarse de forma concurrente y las que han de ser ejecutadas de forma secuencial por depender entre sí debido a relaciones de flujo. Ejemplos de este tipo de estrategias estructuradas son SDRTS [WM85], y principalmente DARTS [GOM84] y su extensión para sistemas distribuidos, DARTS-DA [GOM89], donde se proponen una serie de reglas o criterios para la identificación y agrupación de tareas. Todas ellas tienen un punto de vista estructural del proceso de diseño, y no ofrecen soporte ni definen estrategias para asegurar la predictibilidad del sistema, o integrar análisis temporal durante el proceso de diseño [KZF91] [BW95].

El éxito de la programación orientada a objetos dio lugar a la aparición de nuevas estrategias de diseño de tiempo real, en los que la descomposición del sistema se basa en la identificación de los objetos que forman parte del dominio del sistema, y no en su funcionalidad. Es posteriormente, cuando con el objetivo de implementar la funcionalidad del sistema, deben identificarse las interfaces de cada objeto y distinguirse la naturaleza concurrente o no concurrente de algunos de ellos (objetos pasivos frente a objetos activos). Algunos de estos métodos surgen como extensiones de métodos estructurados previos, como es el caso de CODARTS [GOM93], que surge de aplicar orientación a objetos a DARTS. COMET [GOM00] es el resultado de utilizar UML como base para un proceso de diseño acorde a CODARTS, partiendo de una especificación del sistema basada en casos de uso. En este caso, el proceso sí propone la integración del análisis temporal del sistema diseñado, bien a través de análisis de planificabilidad basado en técnicas RMA (*Rate Monotonic Analysis*) [KFP93], o a través de análisis de secuencias de eventos [GOM00]. Otras estrategias de diseño orientadas a objetos son ROOM [SGW94] o ROPES [DOU99], que tampoco integran análisis temporal en el proceso, o HRT-HOOD [BW95], que integra análisis basado en técnicas RMA, y que aunque en principio considera una estructura del sistema basada en objetos, también puede ser aplicado a sistemas construidos con estrategias estructuradas.

Otra estrategia de diseño de tiempo real, ortogonal a las anteriores, es la basada en el modelo transaccional [KDK89][KZF91]. En este caso, el concepto clave es el de transacción, que juega un doble papel en el proceso de diseño:

- Por un lado, es el mecanismo utilizado para especificar la aplicación. Una aplicación se concibe como un conjunto de transacciones concurrentes, cada una de las cuales representa la respuesta del sistema a un evento externo, procedente del entorno, o interno, procedente de algún reloj del sistema.
- Por otro, a través de sucesivos refinamientos, sirve como base para la toma de decisiones a lo largo de todas las fases del proceso de diseño, tanto para decidir la arquitectura del sistema, como el modelo de concurrencia, el modelo lógico, etc.

Utilizando esta estrategia se simplifica el paso de la especificación del sistema a su diseño e implementación. Además, diseñar sistemas de tiempo real basados en un modelo reactivo, en el que la funcionalidad y comportamiento temporal se formulan como respuestas a eventos externos o temporizados, es muy conveniente, ya que la reactividad es la principal característica de este tipo de sistemas. En [MED05] se desarrolla una metodología de diseño y análisis de sistemas de tiempo real basada en modelo transaccional, y adaptada a sistemas orientados a objetos.

En la actualidad, con la aparición de nuevas estrategias de desarrollo de software, como el desarrollo basado en componentes, o el desarrollo dirigido por modelos, surge la necesidad de una nueva adaptación de las estrategias de diseño de tiempo real. En concreto, en esta tesis se propone una adaptación de la estrategia de diseño basada en modelo transaccional para el caso de aplicaciones de tiempo real basadas en componentes.

1.1.2. Modelo de tiempo real de un sistema

Independientemente de la estrategia elegida, el diseño de tiempo real debe dar lugar a sistemas cuyo comportamiento temporal pueda ser analizado de forma previa a su ejecución. Para ello, y debido a que los sistemas que se tratan son excesivamente complejos para ser gestionados directamente desde sus modelos lógicos, se hace necesario formular un nuevo modelo, complementario a la descripción funcional del sistema, que describa cualitativa y cuantitativamente el comportamiento temporal del sistema completo. Este modelo, que denominamos modelo de tiempo real, es una abstracción del sistema que contiene toda la información que se necesita para predecir y evaluar su comportamiento temporal, esto es, para hacer predecibles los instantes en que se producen sus respuestas. Como se muestra en la figura 1.1, este modelo es utilizado por los distintos agentes involucrados en el proceso de desarrollo de una aplicación. Se utiliza para formular los requisitos temporales durante la fase de especificación, razonar sobre la arquitectura y el modelo de concurrencia que son más adecuados en las fases de análisis y diseño, y certificar el cumplimiento de los requisitos temporales en la fase de validación.

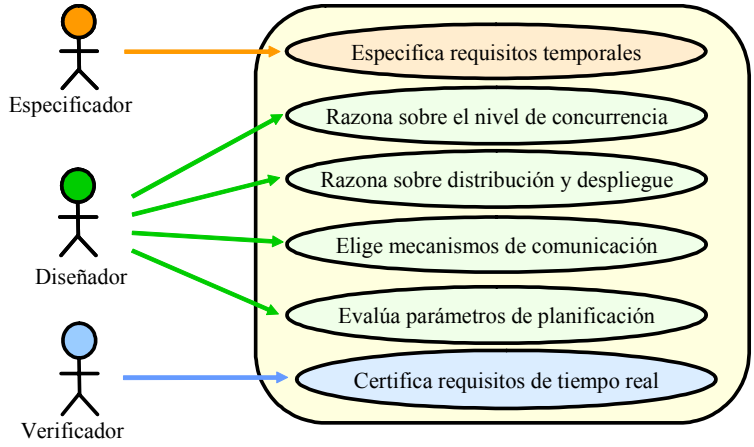


Figura 1.1: Formas de uso del modelo de tiempo real de una aplicación

Existen diferentes metodologías para la elaboración del modelo de tiempo real de un sistema, ligadas de forma directa a las técnicas de análisis a aplicar. Una de las más difundidas, y en la que se centra todo el trabajo expuesto en esta tesis, es la que se basa en el modelo transaccional [KDK89][LIU00], del que ya hemos hablado anteriormente. Su difusión se produjo principalmente a partir de la aparición de la teoría RMA para análisis de planificabilidad [KRP93][LL73].

Siguiendo una metodología transaccional, el modelo de tiempo real de un sistema se formula a través de dos descripciones complementarias:

- Modelo de flujo de control o transaccional: Es un modelo reactivo, que describe el sistema como el conjunto de secuencias de actividades, denominadas transacciones, que concurren en él y que se desencadenan como respuesta a eventos externos (procedentes del entorno) o de temporización (procedentes del reloj del sistema). Cada actividad describe la cantidad de procesamiento (o el tiempo de ejecución) que se requiere para ejecutar un segmento de código, y se relaciona con el resto por el flujo de control o porque compite con otras actividades en el acceso a recursos compartidos. El modelo incluye el patrón con que se generan los eventos que desencadenan cada transacción.

Además, las transacciones sirven como marco de referencia en el que definir los requisitos temporales que se establecen en la especificación de la aplicación. Estos requisitos se especifican como restricciones temporales entre la ejecución de las diferentes actividades que forman la transacción.

- Modelo de uso de recursos o de contención: Es un modelo que describe los recursos de procesamiento y de sincronización que se requieren para la ejecución de cada actividad incluida en una transacción. Estos recursos son comunes a muchas de las actividades, por lo que en su acceso se pueden generar bloqueos y retrasos que son claves para evaluar el comportamiento temporal del sistema. Los recursos que se modelan son:
  - Recursos de procesamiento, que se utilizan para la ejecución de código (procesadores) o transferencia de mensajes (redes de comunicación). Se describe su capacidad y las políticas y parámetros (planificadores) que utilizan para planificar las actividades pendientes de ejecución.
  - Recursos de sincronización (recursos compartidos), que utiliza la aplicación para gestionar el acceso a elementos compartidos entre actividades que se ejecutan concurrentemente. La descripción incluye las políticas y parámetros de planificación que se utilizan para gestionar el acceso de las diferentes actividades a los recursos.
  - Recursos de concurrencia (servidores de planificación), que representan cada una de las tareas concurrentes ejecutadas en el sistema. Su descripción incluye los parámetros y políticas que se utilizan para que sea correctamente planificado su acceso a los recursos de procesamiento y de sincronización.

El conjunto formado por estos dos modelos constituye lo que se denomina una situación de tiempo real [SPT][MAST], o contexto de análisis [MARTE], que representa un modo de operación del sistema caracterizado por una determinada carga de trabajo. Esta carga se describe a través del patrón de generación de los eventos que lanzan las transacciones y las restricciones temporales que se deben cumplir en la ejecución de las respuestas a dichos eventos. A cada situación de tiempo real se le aplican herramientas de análisis que permiten evaluar si se satisfacen los requisitos temporales, así como las holguras con las que se alcanzan, y/o herramientas de diseño con las que se evalúan los parámetros de planificación que conducen al cumplimiento de los requisitos temporales establecidos.

La importancia actual de la metodología de modelado transaccional es consecuencia de tres hechos:

1. Se dispone de un amplio conjunto de métodos y herramientas de análisis de planificabilidad, estimación de tiempo de respuesta y asignación óptima de prioridades basados en esta estrategia [LIU00][CHE02][MAST].
2. Se ha utilizado como referencia para seleccionar los mecanismos de sincronización y las políticas de planificación de los sistemas operativos de tiempo real y propósito general [GOP01].
3. Es el modelo de referencia que se ha adoptado en los perfiles propuestos por la organización OMG para la estandarización de modelos y herramientas utilizados en el diseño de sistemas de tiempo real, tanto el "*UML Profile for Schedulability, Performance and Time*" (SPT) [SPT], como el nuevo "*UML Profile for Modeling and Analysis of Real-Time and Embedded Systems*" (MARTE) [MARTE]. El perfil MARTE sustituye y perfecciona el perfil SPT, que se centraba exclusivamente en el modelado para análisis temporal, añadiéndole nuevos elementos de modelado que permiten modelar otros aspectos de un sistema de tiempo real empotrado.

1.2. Desarrollo de software basado en componentes

A medida que los costos de desarrollo del hardware disminuyen, los procesadores aumentan su capacidad de procesamiento y disponen de una mayor cantidad de memoria, y las aplicaciones que soportan se hacen cada vez más complejas. Por ello, se requieren nuevas estrategias de desarrollo de aplicaciones que sean capaces de abordar esta creciente complejidad y sean más ágiles en la adaptación a la evolución del mercado. La modularización y la reutilización del software constituyen dos puntos claves para el desarrollo de estas nuevas estrategias.

La idea de la reutilización de componentes software se planteó ya en 1968, cuando Douglas McIlroy identificó la necesidad de una industria de componentes software como solución para la denominada "crisis del software" [NR68], aunque su verdadera implantación no comenzó hasta los años 90 [SZY98][LR01]. En la actualidad el desarrollo basado en componentes, CBD (*Component-based Development*), se considera como una de las disciplinas con más éxito para mejorar la productividad del software y afrontar el creciente avance de las tecnologías de la información. Se encuentra plenamente implantado en algunos dominios de aplicación como la ofimática, el comercio electrónico, o las aplicaciones gráficas, en los que tecnologías de componentes de propósito general como COM [COM] y.NET [NET] de Microsoft, o EJB [EJB] de Sun Technologies ocupan la práctica totalidad del mercado.

El objetivo básico del desarrollo de software basado en componentes consiste en construir sistemas mediante ensamblado de módulos software reutilizables (componentes), que hayan sido desarrollados previamente por terceros con independencia de la aplicación en la que vayan a ser utilizados. Se trata, por tanto, de implantar el concepto de mercado industrial en el desarrollo de software. Al igual que un coche o un computador se construyen por ensamblado de piezas procedentes de diferentes fabricantes que son conectadas de forma estándar; el desarrollador de una aplicación, como muestra la figura 1.2, dispone de un repositorio de componentes, donde puede elegir aquellos que son adecuados para su aplicación, la cual es construida mediante ensamblado de los componentes elegidos. Con esta estrategia el papel de los desarrolladores de aplicaciones cambia, pasando de ser programadores a ser ensambladores. En el peor de los casos se les requerirá únicamente la codificación del código necesario para realizar la conexión entre los componentes (*glue code*).

La adopción de estrategias basadas en componentes conlleva un cambio bastante profundo en los procesos de desarrollo de software, pues aparecen nuevas etapas y nuevos agentes implicados [BRO00]. Todos estos cambios respecto de los procesos tradicionales de la ingeniería software han dado lugar a una nueva disciplina dentro de ella, denominada ingeniería software basada en componentes, CBSE (*Component-based Software Engineering*) [CRN03].

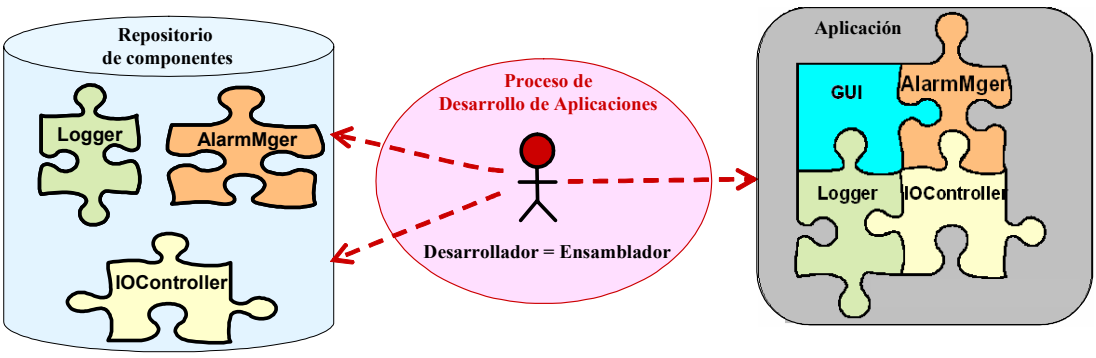


Figura 1.2: Desarrollo basado en componentes

La ingeniería software basada en componentes define los métodos y herramientas que dan soporte al desarrollo de sistemas software por ensamblado de componentes, lo que se traduce en tres grandes aspectos:

- Soporte para la construcción de componentes software reutilizables.
- Soporte para la construcción de sistemas por composición de componentes.
- Soporte para el mantenimiento de este tipo de sistemas, bien por sustitución o por introducción de nuevos componentes.

Una de las grandes diferencias entre los procesos de desarrollo tradicionales y los procesos de desarrollo basados en componentes, es que en este último caso, realmente se distinguen dos procesos diferentes pero complementarios [CLC06], que se muestran en la figura 1.3. Por un lado, el proceso de desarrollo de componentes software reutilizables y distribuibles de manera independiente, y por otro, el proceso de desarrollo de las aplicaciones que se construyen por composición de dichos componentes. Estos dos procesos son independientes, los llevan a cabo agentes diferentes y en tiempos diferentes, pero se complementan y necesitan estar coordinados, ya que los productos que se generan en el primero son las piezas constructivas en las que se basa el segundo. Para garantizar la coherencia entre ellos, deben establecerse reglas que definan los productos y la información que se genera en cada una de sus etapas, así como los formatos en que deben suministrarse.

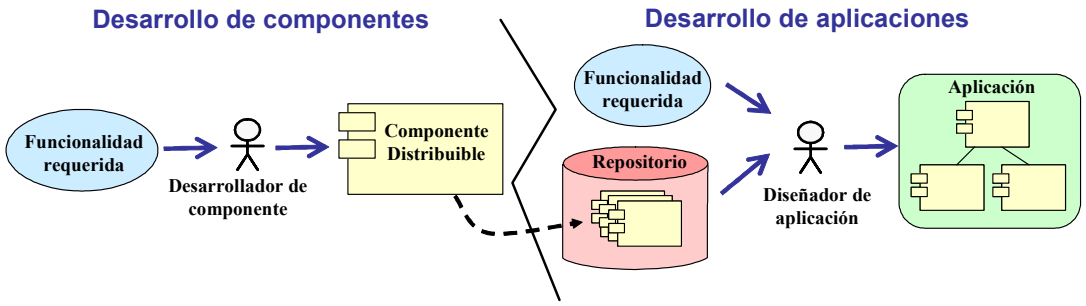


Figura 1.3: Procesos de desarrollo involucrados en CBSE

1.2.1. Concepto de componente software

Incluso dentro del ámbito de CBSE, no existe una definición común o consensuada de lo que es un componente [LW05]. En una de las primeras conferencias sobre componentes software que se celebraron, los participantes discutieron sobre las características que debía cumplir un componente [SP97], y elaboraron una de las definiciones más aceptadas en la actualidad, formalizada posteriormente por Szyperski en [SZY98]:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"*

A partir de esta definición se definen las principales características de un componente software:

- Es una unidad de composición: Las aplicaciones se construyen por composición de componentes, que pueden ser ensamblados unos con otros de un modo conocido y predecible.
- Define mediante interfaces los puntos de acceso a través de los que otros componentes pueden interaccionar con él. Cada interfaz describe un servicio que puede ser requerido

de forma independiente al componente. Estas interfaces representan el contrato entre los componentes cliente y servidor que interaccionan, y es la única información que se conoce cuando ambos son desarrollados sin conocimiento el uno del otro.

- Define los requisitos que impone para su correcto funcionamiento. Dentro de este campo se incluyen tanto los requisitos sobre la plataforma de ejecución, como los requisitos de conexión con otros componentes (generalmente especificados también a través de interfaces, esta vez requeridas).
- Puede ser desplegado (instalado) de forma independiente. Una vez que se comprueba que en la plataforma se satisfacen los requisitos de ejecución especificados en el componente, puede ser instalado en ella como una unidad de despliegue independiente. Asimismo, un componente puede ser reemplazado por otro que ofrezca la misma funcionalidad, sin que ello tenga que suponer ningún cambio para el resto del sistema.
- Los usuarios que gestionen un componente con el objetivo de incluirlo en una aplicación lo hacen considerándolo un elemento opaco, esto es, no necesitan conocer los detalles de su implementación interna, ni tienen que modificar su código.

En resumen, las tres principales características de un componente software son [SZY02]:

- Aislamiento (*Isolation*): Un componente puede ser instalado de forma independiente en una plataforma.
- Componibilidad (*Composability*): Un componente puede ser compuesto con otros para formar aplicaciones.
- Opacidad (*Opaqueness*): Un componente se maneja siempre de forma opaca, sin que los diseñadores de aplicaciones que lo manejan ni el entorno tengan que acceder a sus detalles internos para hacer uso de él.

La separación entre interfaz e implementación es una de las principales características del desarrollo basado en componentes. Es el modo en que se consigue el desacoplo completo entre componentes clientes y servidores. Como se observa en la figura 1.4, el componente cliente ve siempre al componente servidor con vista de caja negra. El desarrollador del componente cliente lo programa en función únicamente de la interfaz requerida, independientemente de la implementación concreta de componente servidor con que sea conectado en una aplicación concreta. Además, de este modo se verifica también la propiedad de aislamiento, al no presentar los componentes dependencias directas con ninguna implementación concreta.

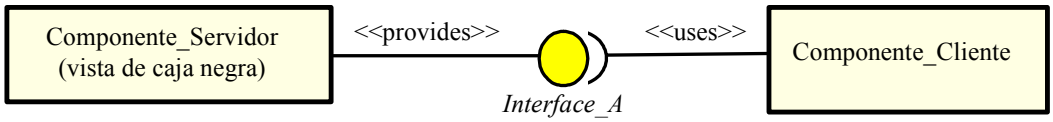


Figura 1.4: La interfaz como medio de desacoplo entre servidores y clientes

La definición de Szyperski es también compatible con la definición de componente que se establece en la última especificación de UML, UML 2.2 [UML2S]. Un componente UML se define como una unidad autónoma dentro de un sistema, con una o más interfaces proporcionadas o requeridas, y cuyos detalles internos son ocultos, pudiendo ser accedidos únicamente a través de sus interfaces. La componibilidad entre componentes se define también en base a la compatibilidad entre interfaces requeridas y proporcionadas.

1.2.2. Uso de metadatos en el desarrollo basado en componentes

Para que pueda ser manejado de forma opaca durante el proceso completo de desarrollo de una aplicación, un componente debe proporcionar como metadatos toda la información que se requiere para hacer uso de él sin necesidad de acceder a su implementación interna. Los metadatos deben proporcionar la información necesaria para incluirlo en una aplicación, conectarlo a otros componentes, configurarlo, etc.

Como se observa en la figura 1.5, un componente se distribuye en un paquete en el que se incluyen tanto todos los ficheros de código que representan cada una de las implementaciones disponibles del componente, como los metadatos a través de la que se puede hacer uso de dichas implementaciones de forma opaca. Estos metadatos son de dos tipos:

- Funcionales: Describen la funcionalidad y las características de componibilidad del componente, o lo que es lo mismo, el conjunto de interfaces ofertadas y requeridas. Incluyen también la declaración de la capacidad de configuración de negocio del componente. Esta información es inherente a la naturaleza del componente, esto es, independiente de la implementación concreta que se desarrolle. Es la única información que maneja el diseñador de una aplicación para decidir la utilización de un componente en ella, o analizar la funcionalidad de un ensamblado de componentes.
- De implementación: Describen características dependientes de cada implementación concreta del mismo componente. Son utilizados por el agente que despliega la aplicación en una plataforma para elegir aquellas implementaciones compatibles con la plataforma de ejecución, y/o para realizar la instanciación de cada componente sin necesidad de acceder a su código.

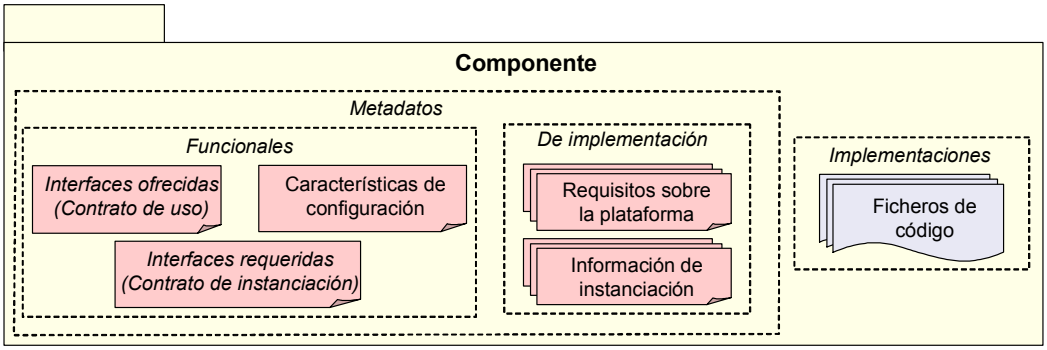


Figura 1.5: Paquete de distribución de un componente software

1.2.3. Modelo de referencia y tecnología de componentes

La definición de componente vista hasta el momento no menciona nada sobre el ciclo de vida de los componentes y/o los protocolos a través de los que éstos interaccionan. Estas características dependen de la tecnología de componentes que se considere. En una tecnología de componentes, un componente se concibe desde dos puntos de vista diferentes [BBB02]:

- Como una abstracción arquitectural, en base a la cual se diseña la arquitectura de las aplicaciones. El modelo de interacción entre componentes se define a través de una serie de reglas que constituyen el modelo de componentes de la tecnología. Todo componente software debe diseñarse conforme a un modelo de componentes, tal y como establece otra de las definiciones de componente software más comúnmente utilizadas [HC01]:

"A component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard"

- Como una implementación, que es instalada en un nodo y ensamblada con otros componentes para dar lugar a una aplicación. La infraestructura que permite realizar estos procesos de instanciación, ensamblado y ejecución de un componente representa el modelo de referencia (*framework*) de componentes de la tecnología.

Por tanto, toda tecnología de componentes debe definir:

- Un modelo de componentes, que es utilizado como guía de diseño por los desarrolladores de los componentes. Define los tipos de componentes que existen, el modo en que éstos se especifican, y los patrones de interacción entre ellos y con el modelo de referencia de ejecución. También será utilizado por los diseñadores de aplicaciones, para que en base a él se definan los ensamblados de componentes que constituyen las aplicaciones. Ejemplos de modelos de componentes son JavaBeans [JBEAN] y Enterprise Java Beans [EJB] de Sun Technologies, COM/DCOM [COM] y COM+ [COM+] de Microsoft, o el modelo de componentes de CORBA, CCM [CCM], de OMG.
- Un framework o modelo de referencia de componentes, que ofrece los servicios necesarios para dar soporte al modelo de componentes en la plataforma de ejecución. Entre estos servicios se encuentran tanto los orientados a la instalación y conexión de los componentes, como los que gestionan su ciclo de vida una vez que ha comenzado la ejecución de la aplicación. Como ejemplos de frameworks de componentes encontramos el que define EJB, basado en una estructura de servidores y contenedores Java, el entorno VisualBasic que soporta el modelo COM, o las diferentes implementaciones que se han desarrollado del modelo de componentes CCM, como OpenCCM [OCCM], MicoCCM [MCCM], CIAO [CIAO], etc.

La figura 1.6, tomada de [BBB02], trata de clarificar la relación entre los diferentes elementos que aparecen en una tecnología de componentes.

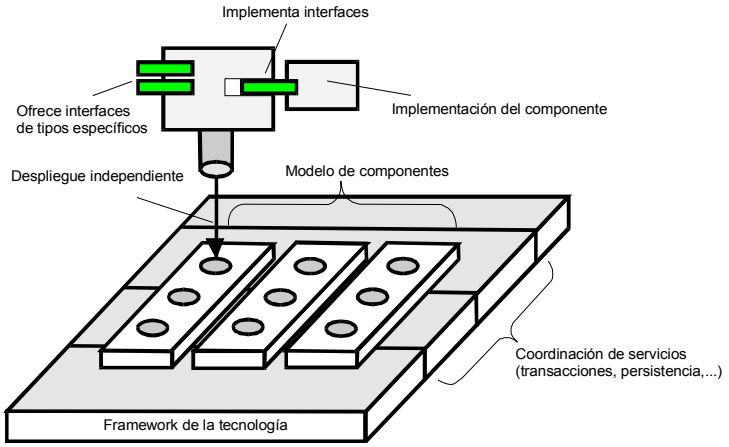


Figura 1.6: Elementos de una tecnología de componentes software

1.2.4. Ventajas y desventajas del desarrollo basado en componentes

Como cualquier otra estrategia utilizada en la industria, el desarrollo basado en componentes ofrece una serie de beneficios que hacen muy atractivo su uso, pero también posee una serie de desventajas que han ralentizado su implantación en algunos dominios.

Entre los beneficios que ofrece una estrategia de desarrollo basado en componentes, se encuentran:

- Mejora de la productividad del programador: El desarrollo basado en componentes se basa en la reutilización de software, y un principio general de cualquier ingeniería es que la reutilización incrementa la productividad. Este incremento no es inmediato, sólo aparece cuando se sobrepasa un cierto umbral de reutilización. En las primeras generaciones de componentes se reduce la productividad, ya que hay que invertir mayor trabajo en desarrollar componentes aptos para su posterior reutilización[SZY98]; sin embargo, cuando la empresa tiene productos desarrollados, o cuando el mercado de componentes ya existe, la productividad se incrementa en mas del 50% [WIL99].
- Reducción de los tiempos de acceso al mercado: Este beneficio está muy ligado a la mejora de la productividad, aunque algunos estudios afirman que en el caso de la tecnología de componentes existen razones adicionales a esta ventaja [CN02]:
  - Las tecnologías de componentes generan líneas verticales de producción dentro de cada dominio de aplicación, lo que contribuye a la definición de arquitecturas y contextos reutilizables muy bien definidos.
  - El uso de las tecnologías de componentes eleva el nivel de abstracción de los elementos que se desarrollan, relegando al modelo de referencia de la propia tecnología la gestión de muchas de las fuentes de complejidad (persistencia, transacciones, gestión de errores, seguridad, etc.). Con ello, el programador puede centrar todo su esfuerzo en la lógica y en la funcionalidad de la aplicación.
- Incremento de la calidad del software: El incremento de calidad ocurre cuando se opera en un mercado estable de componentes, sometido a autoridades que pueden certificar la calidad de los componentes que se utilizan. Por el contrario, el beneficio de incremento de calidad no se obtiene, si a fin de incrementar la productividad y reducir el tiempo de acceso al mercado, se utilizan componente no certificados que no permiten analizar los efectos que su uso va a tener sobre las características no funcionales tales como flexibilidad, escalabilidad, seguridad, etc.

Entre los factores que inhiben la implantación y expansión del mercado de componentes, los más relevantes son:

- Falta de disponibilidad de componentes: La principal dificultad que encuentran las empresas que tratan de implantar metodologías basadas en componentes, es la dificultad de encontrar suministradores de componentes con el estándar industrial requerido y de su dominio de aplicación específico. Es posible que esto sea una consecuencia de la inmadurez del mercado o de que aún no se ha alcanzado una masa crítica suficiente en algunos sectores como control industrial, robótica, etc.
- Falta de estándares estables: Un requisito indispensable para conseguir la confianza de los diseñadores de software en la tecnología de componentes y consolidar así el mercado subyacente, es que el estándar en el que se basa la tecnología permanezca estable a lo largo del tiempo. Actualmente las tecnologías de componentes no son estables como consecuencia de dos tendencias contrapuestas: por un lado el mercado requiere estabilidad, mientras que por otro, es necesario introducir diferencias en los productos para conseguir ventajas competitivas. Solo las tecnologías de componentes soportadas por una única empresa como las de Microsoft o Sun Microsystems han conseguido una estabilidad suficiente para desarrollar su propio mercado.
- Falta de componentes certificados y organizaciones con autoridad de certificación: La adopción de una nueva tecnología se lleva a cabo en dos fases: primero se trabaja en

demostrar su viabilidad, e inmediatamente después, se trabaja para garantizar la calidad de los productos. Dada la opacidad intrínseca que presentan los componentes, la garantía de su calidad no puede ser avalada por el análisis del propio componente, sino que tiene que ser establecida por una autoridad de certificación reconocida.

- Falta de métodos y procesos de ingeniería para desarrollar sistemas con calidad: Actualmente está aceptado que el desarrollo de aplicaciones utilizando tecnologías de componentes requiere un nuevo proceso de ingeniería, con nuevas fases y nuevos agentes. Una organización que desee adoptar una tecnología de componentes software tiene miedo de abordarla por dos razones: el costo que implica el cambio de la organización interna que requiere la nueva ingeniería, y sobre todo en este momento, la falta de experiencia y modelos contrastados.

1.3. Aplicaciones de tiempo real basadas en componentes

Frente a otros dominios de aplicación, donde las tecnologías basadas en componentes han sido implantadas con éxito, su utilización en el caso de sistemas de tiempo real evoluciona de forma mucho más lenta [CRN04]. Aunque la industria automovilística, de aviación, de equipamiento eléctrico, etc. ha demostrado su interés en adoptar este tipo de estrategias [NOR01], ninguna de las tecnologías de componentes existentes cumple todos sus requisitos [MAF04]. Su uso plantea numerosos problemas debidos a la mayor rigurosidad y diversidad de requisitos que imponen los sistemas de tiempo real, y al desconocimiento sobre cómo implementar una tecnología de componentes en un entorno de tiempo real [IN02]. Las tecnologías estándar que se emplean en otros campos, como COM, .NET o CCM son inherentemente pesadas y complejas [CL02][STA01][IN02], introducen sobrecargas difícilmente predecibles [PP99], y no son lo suficientemente escalables como para ser instanciadas en nodos con las restricciones de recursos que son típicas en sistemas de tiempo real empotrados.

La implantación de una metodología de desarrollo basada en componentes en sistemas de tiempo real implica hacer compatibles las principales características de ambos campos:

- La capacidad de construir aplicaciones por ensamblado de módulos software reutilizables, que sean manejados de forma opaca y que puedan ser integrados en diferentes aplicaciones y ejecutados en diferentes plataformas sin necesidad de modificación.
- La capacidad de construir aplicaciones con comportamiento temporal predecible, sobre las que se pueda controlar y certificar la ejecución de todas sus actividades de forma que se satisfagan los requisitos temporales establecidos para ellas.

La orientación a componentes es un patrón estructural que es complementario e independiente al proceso de diseño de tiempo real, pero que al introducir cambios profundos en el proceso de desarrollo de las aplicaciones, interfiere con las estrategias que han venido utilizándose tradicionalmente en el diseño de sistemas de tiempo real. Cuando se aplica una estrategia de diseño de tiempo real basada en modelo reactivo o transaccional, la especificación de la aplicación que constituye el punto de partida del proceso de diseño se formula como un conjunto de transacciones concurrentes. A partir de este punto, como muestra la figura 1.7, el proceso es muy diferente en función del tipo de metodología de desarrollo que se utilice:

- En las metodologías tradicionales, la especificación reactiva de la aplicación sirve como base para decidir el modelo de concurrencia de la misma, esto es, el número de threads que se encargan de la ejecución de cada una de las transacciones, así como la secuencia de operaciones concretas que han de ejecutarse en cada una. En el modelo de tiempo real

de la aplicación se formulan, de acuerdo a la metodología de modelado utilizada, el conjunto de transacciones que forman la especificación, y sobre ellas se mapea el modelo de concurrencia elegido. A continuación, se realiza el diseño lógico del código, agrupando las operaciones identificadas en el paso anterior en módulos o paquetes en función de su naturaleza, y se procede a su codificación. Una vez codificado, el código puede ser medido. Los resultados obtenidos son incorporados al modelo de tiempo real, a través del cual se puede evaluar la planificabilidad de la aplicación u obtener los valores de los parámetros de planificación que deben ser asignados para que la aplicación verifique sus requisitos temporales.

- En metodologías basadas en componentes, lo que se diseña en primer lugar en base a la especificación reactiva de la aplicación es su arquitectura, esto es, el conjunto de componentes que la forman y las conexiones entre ellos. Cuando la estructura de la aplicación queda bien definida, deben identificarse las líneas de flujo de control internas que cada uno de los componentes introduce en el sistema, y asociarlas a los procesos o threads que introducen el modelo de concurrencia, no siendo unívoca ni única la asociación entre componentes y procesos [TH04]. También han de identificarse las cadenas de invocaciones entre componentes a las que se traduce cada una de las transacciones ejecutadas en la aplicación. Dichas secuencias de actividades, junto a las tareas que las ejecutan, son formuladas en el modelo de tiempo real de la aplicación. Al igual que en el caso anterior, este modelo es analizado con el objetivo de extraer la configuración de la aplicación que garantiza su planificabilidad.

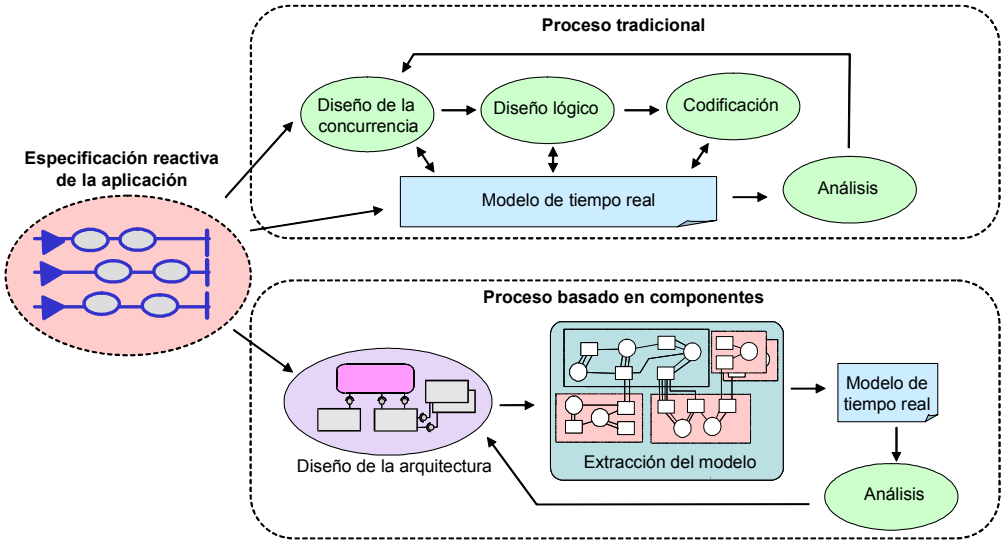


Figura 1.7: Proceso de diseño de tiempo real tradicional vs basado en componentes

Por tanto, mientras que en las estrategias de desarrollo tradicionales, el modelo de tiempo real se utiliza para diseñar la concurrencia y la lógica de la aplicación; en el caso de aplicaciones basadas en componentes, el proceso es opuesto pues dicho modelo debe ser deducido a partir de su descripción arquitectural. Este proceso es complejo, ya que el diseñador de la aplicación maneja los componentes de forma opaca, por lo que no puede conocer de forma directa sus requisitos de concurrencia, las secuencias de actividades que ejecutan, o la temporización de los servicios ofrecidos por cada uno de ellos. Asimismo, mientras que en el caso tradicional los valores de configuración obtenidos del análisis se asignan directamente a los elementos del modelo de concurrencia (threads, mutexes, etc.), en el caso de una aplicación basada en componentes deberán ser asignados como parámetros de configuración de las instancias de

componentes que forman la aplicación. Dichos valores serán gestionados de forma interna, de modo que el componente ofrezca el comportamiento temporal que garantiza que la aplicación en que se utiliza cumple sus requisitos temporales.

La obtención del modelo reactivo de una aplicación a partir de su descripción como ensamblado de componentes sólo es posible si:

- Cada componente que forma parte de la aplicación lleva asociada información que describe las características de temporización, concurrencia y sincronización internas que se necesitan para construir el modelo reactivo de cualquier aplicación de la que el componente forme parte.
- Se utiliza una metodología de modelado con las siguientes características:
  - Esté basada en elementos de modelado modulares que se puedan asociar directamente con la estructura de componentes de la aplicación.
  - Ofrezca mecanismos que permitan generar el modelo de la aplicación completa por composición de los modelos de cada uno de los elementos que la forman.

Por otro lado, para satisfacer el principio de opacidad característico del paradigma de componentes, la información temporal o de tiempo real de los componentes debe ser manejada siempre de forma opaca. Los componentes incluirán metadatos que permitan al diseñador de una aplicación obtener y asignar la configuración adecuada para que la aplicación satisfaga sus requisitos de tiempo real, pero siempre sin necesidad de acceder al código ni a los modelos internos de los componentes.

La verificación de la planificabilidad de una aplicación a partir del análisis de su modelo de tiempo real sólo es posible si la plataforma de ejecución ofrece servicios con comportamiento predecible. Por tanto, en el caso de una aplicación basada en componentes, será necesario contar con una plataforma de ejecución en la que todos los servicios utilizados ofrezcan tiempos de respuesta acotados y con comportamiento predecible.

1.3.1. Concepto de componente de tiempo real

Como muestra la figura 1.8, un componente de tiempo real se define como aquél que incluye en el paquete en el que es distribuido toda la información (modelos) y metadatos que se requieren para predecir y analizar el comportamiento temporal de aquellas aplicaciones en las que sea utilizado. Este concepto se puede encontrar en muchos de los trabajos que abordan este tema [DGL08][WRM05][TN04][BWC04], si bien el tipo o formato de los modelos y los metadatos depende en cada caso del tipo de herramientas o algoritmos de análisis o verificación que se

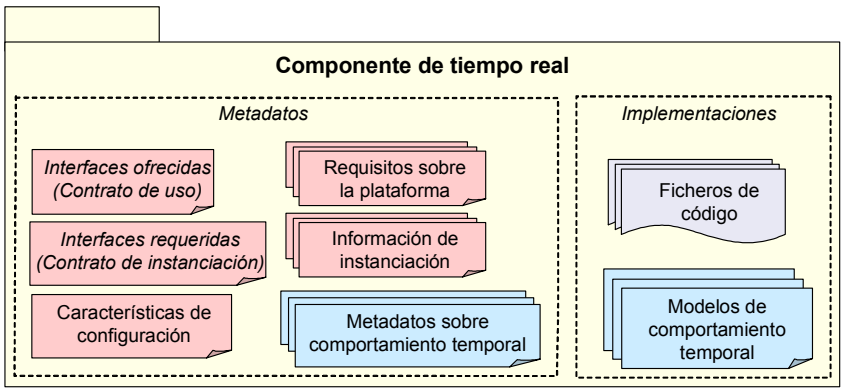


Figura 1.8: Información asociada a un componente software reutilizable de tiempo real

vayan a utilizar. En la siguiente sección se realiza un estudio de las principales propuestas que existen en la actualidad.

En esta tesis se considera que los metadatos de tiempo real que se asocian a un componente son de diferente naturaleza, y se estructuran de tal forma que cada agente involucrado en el desarrollo de una aplicación tenga acceso en cada momento a aquella información que necesita para llevar a cabo las labores que tenga asignadas:

- Un componente debe incluir metadatos de tiempo real que permitan al diseñador de una aplicación evaluar si con el uso de dicho componente se puede dar lugar a una aplicación analizable, esto es, una aplicación de la que se pueda elaborar un modelo de comportamiento temporal que sirva de base para las herramientas de diseño y análisis de tiempo real.
- Deben incluirse metadatos que permitan al diseñador de la aplicación elegir aquellos componentes adecuados para una aplicación, aplicando criterios basados en la reactividad de los componentes y de la aplicación.
- Frente al caso general, en el que únicamente pueden declararse propiedades de configuración relacionadas con la funcionalidad de negocio del componente, en este caso debe darse soporte a la definición de parámetros especiales de configuración que permitan controlar la planificación y la concurrencia. Estos parámetros han de ser manejados de forma oculta al diseñador de la aplicación, siendo calculados y asignados automáticamente por herramientas a partir del análisis del modelo de tiempo real de la aplicación.

Estos metadatos describen los modelos internos que caracterizan el comportamiento temporal de los componentes, y que incluyen toda la información que pueda ser relevante para evaluar el comportamiento temporal de las aplicaciones de las que el componente forme parte.

1.3.2. Modelado de aplicaciones basadas en componentes

Una metodología de modelado del comportamiento temporal que sea adecuada para aplicaciones basada en componentes debe ofrecer los siguientes elementos:

- Un modelo de componente que sea independiente de la aplicación en la que el componente se utiliza o de la plataforma sobre la que se ejecuta. Debe ser también completo, incluyendo todas aquellas características que pueden influir en el comportamiento de cualquier aplicación que haga uso de él. Para ello, debe ofrecer entidades de modelado que permitan formular el modelo de tiempo real de un componente como un conjunto autocontenido de abstracciones que describan las características de temporización y sincronización de su código, con independencia de quién invoca su ejecución o la plataforma que lo ejecuta.
- Un proceso de composición formalizado, que permita obtener el modelo de tiempo real de una aplicación por composición de los modelos de tiempo real de los elementos que la forman.

El tipo de sistema que consideramos aplica la componentización a todos los niveles, no sólo a nivel de software, sino también a nivel de la plataforma de ejecución (sistema operativo, mecanismos de comunicación, etc.). En este tipo de sistemas, es necesario que todos los elementos que forman la aplicación lleven asociada información acerca de su comportamiento temporal, ya sean componentes software o elementos de la plataforma. De este modo, como muestra la figura 1.9, una vez que se ha definido y configurado la estructura de componentes que responde a la especificación reactiva de la aplicación, y se ha definido su despliegue en una

determinada plataforma, se pueden componer los modelos de cada uno de los elementos que formen parte del sistema completo (aplicación y plataforma), obteniendo el modelo de tiempo real final que sirve de entrada a las herramientas de análisis.

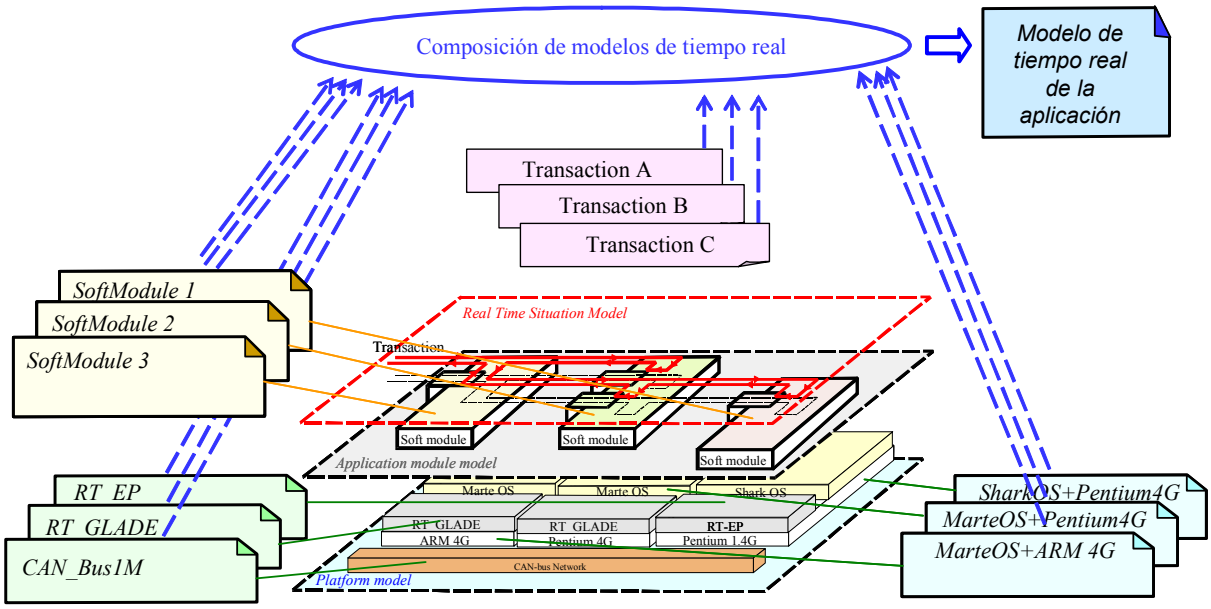


Figura 1.9: Obtención del modelo de tiempo real de una aplicación basada en componentes

Para tener capacidad de describir el comportamiento de tiempo real de un componente a través de un modelo que sea independiente de las aplicaciones de las que forme parte, esto es, un modelo reutilizable, es necesario resolver los siguientes problemas que son inherentes a su naturaleza:

- El comportamiento temporal de los servicios ofrecidos por un componente no sólo depende de su código, sino también del comportamiento temporal de los servicios de otros componentes de los que hace uso para implementar su funcionalidad. Esto significa que el modelado de tiempo real de un servicio debe describir mecanismos de colaboración y sincronización entre componentes a un nivel más bajo del que ofrecen las interfaces que suelen utilizarse para describir los servicios desde el punto de vista funcional.
- La respuesta temporal de un componente depende también de la plataforma que lo soporta. Por tanto, los modelos de tiempo real de los componentes han de ser formulados haciendo referencia a los modelos que describen el comportamiento de los elementos de la plataforma que constituyen su entorno de ejecución.
- La respuesta temporal de un componente depende de la disponibilidad de los recursos que utiliza (procesador, sistema operativo, servicios de comunicación, mecanismos de sincronización, etc.), y ésta es función de la carga de trabajo que se esté ejecutando en la plataforma.

Esto lleva a la conclusión de que el modelo de tiempo real de un componente software nunca puede ser un modelo final, sino que deberá ser formulado en función de aquellos elementos de los que depende el comportamiento temporal del componente. De este modo, la respuesta temporal de los servicios de un componente sólo puede ser evaluada para cada instancia del componente en el contexto de una aplicación completa, una vez conocidos la plataforma en que

se ejecuta, las implementaciones de los componentes que utiliza para implementar su funcionalidad y la carga de trabajo que va a ejecutar la plataforma.

1.3.3. Antecedentes de sistemas de tiempo real basados en componentes

En el ámbito de aplicaciones software de propósito general, existe una gran cantidad de bibliografía acerca de desarrollo de software basado en componentes, donde además de los modelos líderes del mercado, como JavaBeans, EJB o .NET, se encuentran otras propuestas académicas o industriales. En este apartado no se analizan estas propuestas, pues la mayoría de ellas no contemplan ningún aspecto relacionado con tiempo real, sin embargo, en [LW05] se puede encontrar un estudio comparativo de algunos de los modelos de componentes de carácter general más extendidos y utilizados en la actualidad.

La dificultad que conlleva utilizar tecnologías basadas en componentes en aplicaciones con requisitos temporales ha retrasado su implantación en la industria, aunque por otro lado ha generado también una gran actividad investigadora. Todas las propuestas buscan la predictibilidad temporal de las aplicaciones, para lo que consideran necesario ampliar la información que se proporciona con un componente, esto es, incluir metadatos de tiempo real. Sin embargo, el modo de gestionar dichos metadatos y el formato de los mismos varía en función del tipo de estrategias de modelado y análisis utilizadas.

Las tecnologías de componentes con mayor implantación en el mercado, como EJB, y.NET, no ofrecen garantías de predictibilidad temporal, principalmente por el tipo de plataformas en el que ejecutan. Además son demasiado pesadas y complejas para ser ejecutadas en entornos de tiempo real empotrados. No existen tampoco muchas propuestas que traten de adaptar estas tecnologías a aplicaciones de tiempo real; en el trabajo que se presenta en [WRM05] se utiliza EJB sobre una plataforma con soporte para la especificación Java de tiempo real (RTSJ) y con un modelo de RMI (*Java Remote Method Invocation*) modificado con el objetivo de ofrecer garantías de tiempo real. Sin embargo, no se dan detalles acerca de como se han realizado dichas modificaciones ni de las restricciones que se han introducido en el modelo EJB. Por otro lado, aunque no basadas en una extensión de EJB, existen propuestas que tratan de desarrollar modelos de componentes de tiempo real sobre plataformas con soporte para la especificación de tiempo real de Java (RTSJ) [HGC07] [ECB06][PMS08].

“Composability” frente a “Compositionality”

En lo que a la composición de características no funcionales de los componentes se refiere, se distinguen dos propiedades diferentes que pueden ser garantizadas por un modelo de componentes [HDJ08][GS05]:

- *Composability*: Capacidad de garantizar que las propiedades de un componente se conservan cuando éste es integrado en una aplicación.
- *Compositionality*: Capacidad de deducir las propiedades del sistema completo a partir de las propiedades de sus componentes.

Algunas propuestas destacan la importancia de la primera propiedad frente a la segunda. Esta estrategia puede ser adecuada en el tratamiento de algunas características no funcionales de un componente, como la cantidad de memoria requerida; sin embargo, en el caso de la predictibilidad temporal introduce una gran cantidad de restricciones. El comportamiento temporal de un componente depende de la plataforma en la que sea ejecutado, y del resto de componentes que utilice para implementar su funcionalidad, por lo que para garantizar sus tiempos de respuesta de forma independiente de la aplicación, el componente debería

restringirse a una única plataforma, y formular sus tiempos de respuesta en base a estimaciones de las respuestas de los componentes que utilice. En este sentido, la utilización de este tipo de estrategias disminuye la capacidad de reutilización de componentes.

Un ejemplo de trabajo de este tipo es el que se propone en [WRM05], donde se define el concepto de "*Real-Time Service Specification*", que es asociado a cada servicio ofrecido por un componente. En él se definen los plazos que se garantizan para la ejecución de cada invocación del servicio, formulados en función del patrón de invocación correspondiente. En este caso, el modelo de componente es muy sencillo, no existiendo dependencias directas entre componentes, por lo que sus respuestas temporales dependen exclusivamente de cada componente y de la plataforma en que se ejecuta. Además, la formulación en función de los patrones de invocación de los servicios disminuye la capacidad de reutilización de los componentes, pues requiere conocer a priori el tipo de aplicación en que éstos van a ser ejecutados. En la misma línea se encuentra el trabajo presentado en [WT05][CLS06], basado en lo que se denominan "*Real-Time Interfaces*". Cada componente se define a través de su interfaz de tiempo real, en la que se especifica el retraso máximo que se producirá en su ejecución, supuesto un límite superior para la tasa de invocación del componente, y una determinada disponibilidad de recursos de la plataforma. En este caso el modelo de componente es también un modelo de baja granularidad: un componente implementa un único servicio, que procesa los datos de entrada y genera los datos de salida. Con este modelo, no existen dependencias explícitas entre componentes, únicamente se relacionan por flujo de datos.

Frente a esta idea de conservación de las propiedades de los componentes, otras propuestas, entre las que se incluye la de esta tesis, destacan la importancia de las propiedades del sistema completo frente a las de cada componente de forma aislada [HAC01]. Los requisitos no funcionales se especifican sobre la aplicación completa, y para su verificación se hace uso de los metadatos que aporta cada componente. Esta es la visión adecuada en el caso de aplicaciones de tiempo real formuladas a partir de un modelo transaccional, en el que el objetivo es conseguir el cumplimiento de los requisitos *end-to-end* que se imponen en las transacciones, las cuales atraviesan múltiples componentes. Lo importante no es por tanto el comportamiento temporal aislado de cada componente, sino la colaboración entre todos ellos para cumplir el requisito de la transacción.

Esta es la línea de trabajo que plantea el proyecto PACC (*Predictable Assembly for Certifiable Components*) [MH05][PACC]: predecir el comportamiento de un ensamblado de componentes durante la fase de diseño en base a propiedades proporcionadas por los componentes involucrados. Dentro de este proyecto se define la tecnología PECT (*Prediction Enabled Component Technology*) [HMS03] [WAL03], que más que una tecnología de componentes, constituye una guía sobre como integrar tecnologías de componentes con técnicas analíticas que permitan certificar el comportamiento temporal de una aplicación basada en componentes durante su fase de diseño. En base a esta idea se ha desarrollado un entorno de desarrollo MDE [MM08], que en base a transformaciones de modelos, genera tanto el código de las aplicaciones como los modelos de análisis, que son evaluados con las herramientas del entorno MAST. La tecnología de componentes subyacente es la tecnología PIN [HIP05], sobre la que se imponen numerosas restricciones para la generación del modelo de análisis, como que el entorno sea monoprocesador, o que dos componentes con la misma prioridad no puedan ser activados simultáneamente, lo cual es difícil de asegurar en ejecución. En cualquier caso, la idea principal de PECT se puede aplicar a numerosas propuestas, entre ellas la que se expone en esta tesis.

Entre ellas se encuentra también UM-RTCOM [DGL08]. Se trata de un modelo predictible de componentes que incorpora soporte para aplicar análisis de planificabilidad. En él se considera

que un componente de tiempo real no está plenamente especificado si no incluye un modelo de análisis, que en su caso se formula haciendo uso de una extensión para tiempo real del lenguaje SDL [ADL03]. El modelo de una aplicación se obtiene por composición de los modelos de cada uno de los componentes que lo forman. El modelo de cada componente es alimentado con valores de tiempo de peor caso que son obtenidos para cada componente cuando es instalado en una plataforma concreta, con lo que se resuelve la dependencia temporal de la plataforma. Para asegurar el comportamiento esperado en tiempo de ejecución, los componentes son mapeados a objetos CORBA, que son ejecutados sobre una plataforma RT-CORBA.

Modelos de componentes orientados a evaluación de prestaciones (*performance*)

Siguiendo también los fundamentos del proyecto PACC se encuentra el trabajo que se presenta en [BCW04][BCW06], que extiende el modelo de componentes Robocop [RBC03] para soportar predicción de propiedades de tiempo real de un ensamblado de componentes. Un componente Robocop está formado por un conjunto de diferentes modelos: código ejecutable, modelos de uso de recursos, modelos funcionales, etc. En el trabajo referenciado se definen nuevos modelos a incluir en un componente con el objetivo de describir su comportamiento temporal, que junto a una estrategia de composición permiten generar el modelo temporal de una aplicación completa. El entorno de desarrollo CARAT [BCW07] incluye una serie de herramientas que analizan dicho modelo. Los modelos reutilizables y el proceso de composición están formalmente definidos y son muy completos; sin embargo, el tipo de análisis aplicado está principalmente enfocado a evaluación de *performance*, aplicándose técnicas basadas en simulación, que no pueden certificar la planificabilidad de la aplicación. Además, no queda clara la influencia de algunas características de la plataforma, como los cambios de contexto, que no son modelados en ninguno de los elementos, ni tampoco la influencia del middleware o de los sistemas de comunicación utilizados. Por otro lado, no se ofrece capacidad de configuración de los parámetros de planificación de los componentes; si el análisis del sistema resulta no satisfactorio debe considerarse la sustitución de algún componente, o la modificación del despliegue o de la plataforma.

La evaluación de *performance* en aplicaciones basadas en componentes es uno de los campos donde mayor esfuerzo investigador se está realizando en la actualidad [WFP07]. En [BGM06] se puede encontrar un estudio de diferentes propuestas, clasificadas en función del tipo de técnicas de análisis utilizadas. Aunque el tipo de análisis que se aplica en este caso es diferente que en el caso del análisis de planificabilidad, las estrategias de modelado son muy similares. Esto queda de manifiesto en el perfil MARTE, donde los subperfiles de planificabilidad y *performance* son extensiones de un perfil común, denominado GQAM (*Generic Quantitative Analysis Modelling*). Por tanto, muchos resultados de investigación en este área pueden ser de utilidad en el caso de sistemas de tiempo real con requisitos estrictos.

Uno de los trabajos más interesantes y completos en este campo es el modelo de componentes Palladio [BKF07][BKR09]. Aunque está orientado exclusivamente a evaluación de *performance*, comparte con la estrategia que se presenta en esta tesis la importancia de distinguir las diferentes piezas que forman el modelo no funcional de una aplicación, y de asignar la responsabilidad de la elaboración de cada una de ellas a los diferentes agentes que participan en el desarrollo, tanto de componentes como de aplicaciones. El modelo define su propio metamodelo para poder incluir información relativa a *performance* en la especificación de los componentes, y poder hacerlo además de forma paramétrica, dejando sin resolver aquellas características que dependen del contexto en el que el componente sea ejecutado.

Componentes de baja granularidad frente a componentes de alta granularidad

El concepto de componente que se concibe en esta tesis es un componente de alta granularidad, esto es, un componente que puede ofrecer una funcionalidad compleja, a través de la implementación de múltiples interfaces. Este concepto es similar al utilizado en los modelos UM-RTCOM, Robocop, o Palladio. Frente a esta idea, muchos de los modelos de componentes de tiempo real que se encuentran en la actualidad aplican la semántica "*run-do-write*" a sus componentes. Según este modelo, un componente es totalmente pasivo y posee dos tipos de puertos: de datos o de disparo. Cuando todos los puertos de disparo de entrada son activados, el componente lee los valores de sus puertos de datos de entrada, ejecuta una computación en base a ellos, y genera sus respuestas, que son actualizadas en los puertos de datos de salida. Los componentes no pueden interactuar con otros componentes ni con el entorno, salvo a través de sus puertos. A nivel de sistema, la ejecución puede ser iniciada bien por eventos externos o eventos de reloj, que activan alguno de los puertos de disparo de los componentes. Los requisitos temporales se imponen como plazos en la respuesta a dichos eventos. Este modelo es sencillo y da lugar a componentes de baja granularidad. La mayoría de los modelos que lo utilizan están muy orientados a sistemas de control. Ejemplos de modelos de este tipo son la tecnología PIN que nombramos anteriormente, SaveCCM [HAC04], COMDES-II [KSA07] y COMDES-III [ASZ08], PBO [STE01] y PECOS [NAD02].

La tecnología SaveCCT [ACF07], que se apoya en el modelo SaveCCM, destaca el rol de los componentes como unidades de diseño frente a unidades de implementación. Durante el proceso de diseño, el sistema se construye como una interconexión de componentes con la semántica expuesta anteriormente, en base a los cuales se analiza la planificabilidad del sistema. Posteriormente, este diseño de alto nivel es transformado directamente, por generación de código y mapeado de componentes a tareas [AMH05], a entidades ejecutables que son gestionadas directamente por el entorno de ejecución definido en el modelo. El modelo define un entorno integrado de desarrollo, Save-IDE [SHP08], en el que se incluye soporte para asociar a cada componente un modelo de comportamiento formulado en base a autómatas temporizados, así como para analizar el comportamiento del sistema completo en base a dichos modelos a través de la herramienta UPPAAL-PORT [HCM08]. Recientemente, se ha propuesto una extensión de este modelo, denominado ProCom [SVB08], cuyo objetivo es dar solución a la baja granularidad de los componentes. Para ello, se define un modelo de dos capas, donde componentes de más alto nivel de abstracción (capa ProSys) pueden ser generados en base a composición y encapsulación de componentes SaveCCM (capa ProSave).

El criterio de composición de propiedades no funcionales en el caso de SaveCCM, al menos en el caso de las propiedades relacionadas con la planificabilidad, es una mezcla de las dos estrategias de composición de las que se ha hablado anteriormente. Por un lado, la planificabilidad del sistema es analizada para el sistema completo en base a la información aportada por cada componente y los requisitos temporales que se imponen. En este caso, y debido al modelo de computación subyacente, cada componente aporta únicamente el valor de tiempo de ejecución de peor caso (WCET) del algoritmo que aplica sobre sus datos de entrada. Sin embargo, al enfatizarse el papel del componente como elemento de diseño, este WCET puede convertirse en un requisito para la posterior implementación del componente, esto es, se utilizan WCET estimados y se analiza el sistema en base a ellos. Posteriormente, el componente deberá asegurar que su tiempo de respuesta es menor que dicho valor.

Modelos de componentes basados en aspectos

Existe otro grupo de trabajos que utilizan estrategias basadas en aspectos [KLM97] para incorporar la gestión de características de tiempo real en aplicaciones basadas en componentes.

Las características no funcionales de los componentes son gestionadas a través de aspectos, con lo que se reduce la complejidad del proceso de diseño y desarrollo de aplicaciones. En [TNH04] se presenta ACCORD, una metodología de diseño basada en la descomposición de los sistemas en componentes y aspectos, que son posteriormente implementados sobre RTCOM, un modelo de componentes en el que se da soporte a la modificación del comportamiento de los componentes en base a aspectos. El concepto de componente utilizado en este modelo no cumple totalmente con el requisito de opacidad, pues es necesario conocer parte del código del componente para poder definir el código de los aspectos con los que se va a modificar su comportamiento. En cuanto a la predictibilidad de las aplicaciones desarrolladas, el modelo incorpora herramientas de cálculo del WCET de los servicios ofrecidos por un componente en función de metadatos incluidos en la descripción del componente. Sin embargo, no expone el modelo de concurrencia o el entorno de ejecución subyacentes, con lo que no queda claro qué técnicas de análisis de planificabilidad pueden ser aplicadas.

En el entorno VEST [STA01][STA03] se extiende el concepto tradicional de aspecto, entendido como un elemento de código que modifica el código base de un componente, para convertirlo en un concepto de diseño e independiente del lenguaje. Las herramientas del entorno se basan en estos aspectos para aplicar diferentes verificaciones en las aplicaciones, entre las que se encuentra el análisis de planificabilidad, para el que se pueden aplicar diferentes técnicas en función del tipo de sistema desarrollado. Aunque el entorno facilita en gran medida el proceso de análisis de un sistema de tiempo real basado en componentes, no queda clara la metodología de modelado que se aplica. Se basa también en la definición a priori del WCET del componente, esto es, si el resultado de un análisis de planificabilidad es negativo, los aspectos pueden ser utilizados para aplicar reducciones en el WCET de algunos componentes hasta hacer el sistema planificable.

Modelos de componentes basados en métodos formales

Existen varias propuestas que se basan en técnicas formales de modelado, como autómatas temporizados, redes de Petri, etc. para describir los modelos temporales de los componentes y analizar las aplicaciones. El entorno BIP [BBS06] es un framework de componentes que permite modelar y componer componentes de tiempo real heterogéneos, esto es, componentes que interaccionan a través de invocaciones síncronas o asíncronas, con ejecución basada en disparo por tiempo o por eventos, etc. El framework define un lenguaje de descripción y composición de componentes, así como reglas de transformación que permiten traducirlo a modelos que pueden ser analizados con las herramientas del entorno IF [BGO04]. Así mismo, también incluye herramientas que generan el código ejecutable de los componentes en base a su descripción, aunque la plataforma de ejecución es Linux, con lo que no se garantiza comportamiento de tiempo real estricto.

En este grupo se puede incluir también el concepto de "*Real-Time Interfaces*" [WT05][CLS06], visto anteriormente, y trabajos similares a él, como el que se presenta en [HM06]. En él se presenta un álgebra de interfaces basada también en el concepto de supuesto-garantía, esto es, a través de su interfaz el componente garantiza niveles de latencia en la ejecución de sus servicios, en base a una serie de supuestos acerca de los patrones de invocación y de los recursos disponibles. En [JMG07] se propone una metodología de modelado y composición de modelos de componentes basada en autómatas temporizados jerárquicos. En todas estas técnicas el problema es siempre la posible explosión de estados que se puede producir cuando la aplicación que se trata de analizar es compleja, lo que lleva a que en la mayor parte de los casos los modelos de componentes utilizados sean sencillos, basados en componentes de baja granularidad y con muchas restricciones.

Modelos de componentes industriales

Uno de los grandes problemas del desarrollo basado en componentes es la falta de estandarización. En el sector industrial, al no existir un consenso sobre modelos de componentes, formatos de especificación o mecanismos de interacción, cada empresa que ha decidido adoptar esta estrategia ha desarrollado sus propias tecnologías. Todas las propuestas que hemos visto hasta el momento provienen del mundo académico, si bien algunas de ellas han surgido a partir de soluciones industriales. Así, la principal influencia del modelo SaveCCT fue el modelo de componentes Rubus CM [RUBCM], desarrollado por *Arcticus System* y aplicado con éxito en varias empresas, como *Volvo Construction Equipment*, y cuya última versión ha sido recientemente publicada [HMN08]. En esta versión se han incluido todos los avances realizados en SaveCCT, y aunque en un principio el modelo es independiente del framework de ejecución, se ha extendido el sistema operativo Rubus OS [RUBOS], también de Arcticus Systems, para darle soporte. Otro modelo industrial es Koala [OLK00], que sirvió como punto de partida del modelo Robocop. Koala es un modelo de componentes especialmente enfocado a la elaboración de dispositivos electrónicos de consumo inicialmente desarrollado por Philips. Es un modelo sencillo, en el que los componentes se comunican entre sí y con el entorno únicamente en base a interfaces, y son implementados como funciones C, que posteriormente son mapeadas a tareas del entorno. En su especificación inicial, no consideran aspectos sobre predictibilidad ni características de tiempo real, siendo esa la principal extensión que se llevó a cabo en Robocop. Finalmente, ABB desarrolló también su propio modelo de componentes, PECOS [NAD02], especialmente enfocado al desarrollo de dispositivos de campo, como sensores, actuadores, etc. Los componentes PECOS incluyen la posibilidad de definir propiedades de tipo no funcional, que se utilizan para analizar el sistema y obtener, por ejemplo, una planificación adecuada para la aplicación.

En el campo de la automoción, el consorcio AUTOSAR [AUTO], formado por la gran mayoría de fabricantes de dispositivos electrónicos para el automóvil, ha definido un estándar abierto para la arquitectura electrónica del automóvil. Utiliza una arquitectura de software basada en capas, con interfaces estandarizadas para la programación de aplicaciones. Su objetivo es mejorar la escalabilidad, transferencia, y reutilización de módulos software y hardware procedentes de diferentes fabricantes. Sin embargo, en la actualidad el estándar AUTOSAR carece de un modelo de temporización que permita analizar las aplicaciones desarrolladas en base a él [NAT08]. El proyecto europeo TIMMO [TIMMO], actualmente en desarrollo, tiene por objetivo suplir esta carencia, mediante la definición de un modelo de temporización y la estandarización de una infraestructura que de soporte a la gestión de características temporales.

Modelos de componentes acordes a estándares de OMG

Una de las tecnologías de componentes más utilizadas actualmente en sistemas distribuidos es el modelo de componentes de CORBA [CCM]. Su principal interés radica en que ha sido formulado mediante un metamodelo formal, a fin de que cualquier empresa pueda elaborar implementaciones específicas. Este modelo no es compatible con el desarrollo de aplicaciones con restricciones temporales, pero usado en conjunto con la especificación de tiempo real de CORBA, RT-CORBA [RTCRB], y obviamente, con una plataforma con garantías de tiempo real, puede dar lugar al desarrollo de aplicaciones predecibles. CIAO [SDG07][DSG06][CIAO] constituye una implementación con estas características, que además implementa el modelo ligero de componentes de CORBA, LwCCM [CCM], más adecuado a sistemas con recursos limitados. Sobre ella se ha definido un entorno completo de desarrollo basado en estrategias MDD, denominado COSMIC [LTG03], en el que se incluye una implementación de la especificación "*Deployment and Configuration of Component-Based Distributed Applications*"

(D&C) de OMG [D&C] como mecanismo para guiar el despliegue de una aplicación basada en componentes [DBO05]. La tecnología ofrece la posibilidad de configurar características de tiempo real de las aplicaciones, a través de las posibilidades que ofrece RT-CORBA, como las políticas de prioridad, threadpools, etc. de modo que se puedan verificar los requisitos temporales de las aplicaciones [WGS04][KG08]. Sin embargo, no establece ninguna estrategia para obtener dicha configuración a partir de información proporcionada por los componentes, ni aplicar ninguna estrategia de análisis adecuada. La configuración se realiza ad-hoc sobre el diseño completo de la aplicación basándose principalmente en la experiencia del diseñador de la aplicación.

1.4. Objetivos

A lo largo de esta introducción se han definido los conceptos que sirven de punto de partida al trabajo que se expone en esta tesis. Por un lado, se ha introducido el concepto de sistema de tiempo real, junto con la estrategia de modelado transaccional que se concibe en esta tesis para su diseño y análisis. Por otro lado, se han expuesto las principales características de los componentes software reutilizables, tal y como son interpretados en este trabajo. Asimismo, se han identificado los problemas que surgen cuando se trata de desarrollar una aplicación basada en componentes con requisitos temporales estrictos; problemas para los que se proponen soluciones concretas a lo largo de esta tesis.

El objetivo principal de esta tesis es definir una metodología completa de diseño y desarrollo de aplicaciones de tiempo real basadas en componentes, dirigida por un modelo transaccional:

- Por completa entendemos una metodología que abarque todas las fases del proceso de desarrollo de una aplicación, esto es, desde la especificación de requisitos hasta la fase de ejecución. Con este objetivo, y debido a las particularidades del desarrollo basado en componentes, se ha de definir también de forma completa el proceso de desarrollo de componentes como entes independientes y reutilizables.
- Por dirigida por un modelo transaccional entendemos que es un modelo de este tipo el que sirve de base y guía para cada una de las fases del proceso de desarrollo. Se parte de una especificación reactiva de las aplicaciones, sobre la que se especifican los requisitos temporales. En base a ella, se diseña la arquitectura de la aplicación, de la que se obtiene su modelo reactivo. Sobre este modelo, se aplica el análisis del que resulta la configuración de la aplicación (de los componentes que la forman y de la plataforma sobre la que se ejecuta) que garantiza el cumplimiento de los requisitos temporales especificados.

La metodología que se define hace compatibles los principios de reutilización y opacidad característicos del desarrollo basado en componentes, con la predictibilidad temporal y la capacidad para planificar el uso de recursos que se le requiere a las aplicaciones que han de satisfacer requisitos temporales estrictos.

La definición de la metodología se afronta desde diferentes perspectivas, cada una de ellas enfocada hacia uno de los aspectos abordados durante el desarrollo de una aplicación, como son el diseño y elaboración de los componentes individuales, el diseño de la propia aplicación como ensamblado de componentes, el análisis de tiempo real, la ejecución final de la aplicación con garantías de tiempo real, etc. Cada uno de estos aspectos se traduce en un objetivo concreto de esta tesis. Todos ellos se exponen en detalle a continuación.

Objetivo 1: Metodología de formulación de metadatos de tiempo real

El proceso de diseño y análisis de tiempo real debe ser incluido de forma sencilla y natural como una parte del proceso tradicional de desarrollo de una aplicación basada en componentes. Además, debe ser compatible con la opacidad intrínseca de los componentes, esto es, con la utilización de los metadatos que se asocian a los componentes, a la plataforma de ejecución y a las aplicaciones, como única fuente de información utilizada en el proceso. Los metadatos incluyen toda la información necesaria para tomar decisiones acerca de la configuración, despliegue o lanzamiento de una aplicación, así como para que los componentes puedan ser manejados por herramientas automáticas que se encarguen de su configuración, enlazado, instanciación y activación. Por ello, en el caso de aplicaciones de tiempo real, es necesario complementar la descripción de componentes, plataformas y aplicaciones con nuevos tipos de metadatos que permitan gestionar también los aspectos de tiempo real de forma opaca, al igual que ocurre con los aspectos funcionales o de despliegue de los componentes tradicionales.

Como objetivo concreto de esta tesis se definirá el formato y la semántica de los metadatos relativos a tiempo real, así como el modo en que son manejados por las herramientas que soportan el proceso de desarrollo de una aplicación. Con el propósito de facilitar el manejo de los metadatos de tiempo real y de favorecer su integración en los procesos tradicionales de desarrollo, los formatos que se utilicen deberán ajustarse dentro de lo posible a métodos estándar de descripción.

Un aspecto importante en la definición de los metadatos es organizar la información de forma que se presente a cada agente que la utiliza con el nivel de abstracción adecuado al conocimiento que se espera de él. En particular, hay que diferenciar entre la información relativa a la especificación de las características y requisitos de tiempo real, que debe ser conocida por todos los agentes y por tanto será de naturaleza genérica y preferiblemente basada en estándares, y la información relativa a los modelos y técnicas de análisis y diseño de tiempo real, que va a ser utilizada sólo por ciertos agentes especializados y expertos en las metodologías concretas de tiempo real que se estén utilizando.

Objetivo 2: Metodología de modelado de tiempo real orientada a la componibilidad

El comportamiento temporal de las aplicaciones que se desarrollan tiene que ser analizado en base a sus modelos de tiempo real, que para seguir cumpliendo con el requisito de opacidad, deben ser obtenidos a partir de los metadatos incluidos en los elementos que formen la aplicación (tanto componentes como recursos de la plataforma). De aquí surge el segundo objetivo de la tesis, la definición de una metodología de modelado de tiempo real compatible con el diseño basado en componentes.

La metodología tiene que ofrecer una formulación modular, en la que los modelos de tiempo real de los componentes software y de los recursos de la plataforma puedan ser elaborados de forma reutilizable e independiente de las aplicaciones en las que vayan a ser integrados. Asimismo, la metodología estará dotada con las características de componibilidad adecuadas para que a partir de éstos modelos se pueda construir el modelo de tiempo real de la aplicación que se diseña. Los modelos de los componentes se formularán como plantillas que se definen en base a parámetros y referencias a otros modelos, de forma que asignándole los valores adecuados, se puedan adaptar para describir el comportamiento temporal específico de la instancia de componente utilizada en la aplicación que se diseña. Con estas características, se tiene capacidad para generar de forma automática el modelo que describe el comportamiento de tiempo real de una aplicación por composición de los modelos de los componentes que la

forman y los recursos de la plataforma sobre la que son instalados, que con este fin serán configurados de acuerdo a la situación de tiempo real en la que ejecutan.

La metodología desarrollada deberá definir tanto los elementos de modelado que permiten desarrollar modelos de tiempo real reutilizables y componibles, como las herramientas que se encargan del proceso de composición de modelos.

Objetivo 3: Tecnología de componentes con comportamiento temporal predecible

Los aspectos anteriores se centran principalmente en las fases de diseño y análisis de las aplicaciones; sin embargo, ninguno de ellos tiene sentido si no se cuenta con un entorno de ejecución que ofrezca garantías de tiempo real. Por ello, el tercero de los objetivos de la tesis está orientado a la fase de ejecución de las aplicaciones, y consiste en la definición de una tecnología de componentes compatible con aplicaciones de tiempo real. La tecnología debe ofrecer recursos para el diseño y la implementación de componentes de tiempo real, que garanticen un comportamiento temporal predecible de las aplicaciones en las que se utilizan. El modelo de referencia de la tecnología debe definir los mecanismos de interacción y las capacidades de configuración que requiere el tiempo real. De igual manera, el entorno de ejecución debe ofrecer los recursos (threads, mecanismos de sincronización, servicios de comunicaciones, etc.) que requiere el diseño de tiempo real. Esta tecnología tiene que ser compatible con los entornos en los que se ejecutan las aplicaciones de tiempo real, esto es, nodos con sistemas operativos de tiempo real, y en los que frecuentemente hay una severa limitación de recursos disponibles (entornos empotrados).

Para desarrollar componentes de tiempo real cuyos códigos y modelos de comportamiento temporal reutilizables en diferentes aplicaciones y plataformas, es necesario extraer del código de negocio del componente la gestión de aquellos aspectos directamente relacionados con la planificabilidad, como puede ser la gestión de threads o de mecanismos de sincronización. Estos aspectos, que dependen del tipo de plataforma sobre la que se ejecuta la aplicación, deben ser controlados de manera conocida por el entorno de ejecución. Una vez obtenida la configuración de los aspectos de tiempo real de una aplicación en base al análisis de su modelo, la tecnología debe ofrecer los mecanismos necesarios para asignar dicha configuración sin necesidad de acceder al código interno de los componentes. La solución para este problema se basa en la utilización de estrategias basadas en el modelo de programación contenedor-componente. La tecnología que se propone tomará como base para su modelo de referencia este modelo, y lo extenderá para dar soporte al desarrollo de aplicaciones con comportamiento temporal predecible.

La tecnología se va a definir a nivel PIM, esto es, independientemente de la plataforma de ejecución o del lenguaje de programación utilizados para su implementación. Obviamente, para ofrecer garantías de predictibilidad temporal deberá ser implementada sobre plataformas que ofrezcan comportamiento predecible.

Objetivo 4: Propuesta de un proceso de diseño de aplicaciones de tiempo real basadas en componentes

Las soluciones desarrolladas en respuesta a los tres objetivos anteriores deberán ser integradas en un proceso completo de diseño y desarrollo de aplicaciones de tiempo real basadas en componentes, que sirva de referencia a los diseñadores (tanto de componentes como de aplicaciones).

El proceso de desarrollo se ha de definir a partir de un proceso estándar sobre el que se añadan las nuevas actividades que requiere el diseño de tiempo real. Para definir este proceso es

necesario establecer el modo en que se relacionan entre sí los aspectos anteriores, esto es, cómo mapear el comportamiento temporal de un componente en los metadatos que se incluyen en su descripción, o cómo de los resultados del análisis del modelo de una aplicación completa se pueden extraer los parámetros que configuran el comportamiento temporal en ejecución.

Objetivo 5: Aplicación de la metodología a un caso real

A fin de incrementar su aplicabilidad, la metodología que se propone en esta tesis se va a abordar desde un nivel PIM (*Platform Independent Model*). En ella se van a incluir un gran número de abstracciones que requieren ser concretadas cuando se utilice la metodología sobre una tecnología específica, es decir, cuando se pase a un nivel PSM (*Platform Specific Model*). Sin embargo, la aplicación real de la metodología a un caso concreto se considera un objetivo primordial para aclarar y demostrar su utilización. Para ello, se incluye como parte de los objetivos de la tesis el desarrollo de una implementación concreta de la tecnología de componentes propuesta como solución al objetivo 3 (que también se define a nivel PIM) y utilizarla para plantear un ejemplo de referencia del uso de la metodología que se propone.

La aprobación de las especificación Ada 2005 en el periodo que se estaba desarrollando esta tesis ofreció la oportunidad de demostrar como basándose en las nuevas opciones del lenguaje Ada 2005, podía desarrollarse una tecnología completa de componentes de tiempo real. Esta tecnología, denominada Ada-CCM, constituye por tanto, una implementación en lenguaje Ada 2005 de la tecnología especificada previamente, que garantiza la predictibilidad de las aplicaciones desarrolladas en base a ella.

Otros objetivos

Existe otro conjunto de objetivos que son transversales a los expuestos anteriormente:

- En cualquiera de los objetivos expuestos, el concepto de componente al que se quiere dar soporte es un componente de granularidad alta o arbitraria, esto es, un componente con complejidad interna arbitraria que ofrezca diferentes tipos de servicios organizados mediante interfaces.
- Aunque no sea un objetivo directo de esta tesis, se va a tratar de desarrollar soluciones que puedan ser fácilmente extendidas o aplicadas a estrategias MDE (*Model Driven Engineering*) [SCH] o MDA (*Model Driven Architecture*) [MDA]. Para cumplir este objetivo es necesario definir de manera formal los metamodelos correspondientes a cada uno de los modelos que se utilizan durante el desarrollo de un componente o una aplicación, así como las reglas de transformación que permiten pasar de un modelo a otro.

1.5. Organización de la memoria

Los capítulos que conforman esta memoria se corresponden de forma casi directa con los diferentes objetivos que se han planteado para la tesis, y en su conjunto representan la metodología completa de diseño que se plantea como objetivo general.

En el capítulo 2 se expone la metodología modular Mod-MAST, una metodología de modelado y análisis de sistemas de tiempo real basada en modelo transaccional, que permite modelar de forma reutilizable módulos software y recursos de plataformas de modo que su comportamiento temporal pueda ser adaptado a las diferentes aplicaciones en las que sean utilizados.

La metodología se plantea como una extensión de la metodología MAST [GGP01][MGD01] [MAST]. MAST constituye una opción especialmente adecuada para aplicar los conceptos de modularización y reutilización al modelado, ya que formula el modelo de una aplicación en tres secciones independientes: la lógica que modela los elementos software (requisitos de procesamiento y mecanismos de sincronización), la plataforma que modela la capacidad de procesamiento disponible para su ejecución (capacidad de procesamiento y comunicación que aporta el hardware y capacidad consumida por las tareas internas del sistema operativo), y la descripción de las situaciones de tiempo real que modelan desde un punto de vista reactivo el modo concreto del uso de los recursos hardware y software que realiza la aplicación.

El capítulo comienza con la introducción del concepto clave de la metodología modular, la dualidad Descriptor-Instancia. Basado en ese concepto, se explica el resto del metamodelo en que se fundamenta la metodología. En la última parte del capítulo, se explica el proceso de composición que permite generar el modelo de una aplicación por composición de los módulos software y los recursos que forman la plataforma en que es ejecutada. Finalmente, se enumeran en el capítulo las principales semejanzas y diferencias de la metodología con el nuevo perfil MARTE [MARTE] de OMG.

Aunque el objetivo de la tesis está enfocado al desarrollo de una metodología de modelado orientada a componentes, Mod-MAST se ha formulado de una forma más genérica con el objetivo de poder emplearlo en otras estrategias de desarrollo modulares. Con ella se van a poder elaborar modelos de tiempo real de cualquier módulo software reutilizable, sin requerir que éstos ofrezcan las características de un componente software. La particularización de Mod-MAST para la formulación de modelos de tiempo real de componentes software se expone en el capítulo 3.

El objetivo principal del capítulo 3 es el de definir los metadatos que deben asociarse a componentes y aplicaciones a fin de incluir el proceso de diseño de tiempo real en el proceso tradicional de desarrollo de aplicaciones basadas en componentes. Con el propósito de utilizar un mecanismo que favorezca en la mayor medida posible la estandarización del proceso, los nuevos metadatos se definen a través de una extensión de la especificación "*Deployment and Configuration of Component-Based Distributed Applications*" [D&C] de OMG. La extensión, a la que denominamos RT-D&C, se ha realizado a nivel PIM, esto es, de forma independiente de la tecnología de componentes a utilizar, o de la metodología de modelado utilizada para formular los modelos de componentes y aplicaciones.

Al final del capítulo se muestra la extensión realizada a la metodología Mod-MAST con el objetivo de dar soporte al modelado de componentes software reutilizables que formulan su funcionalidad a través de puertos requeridos y ofertados, tal y como se concibe un componente en la especificación D&C.

El capítulo 4 detalla las principales características de la tecnología RT-CCM, una extensión de tiempo real del modelo de componentes LwCCM, que se ha definido como entorno de ejecución para aplicaciones basadas en componentes con comportamiento predecible. A lo largo del capítulo se exponen las características de un componente RT-CCM que lo convierten en un componente de tiempo real. Se detallan también los nuevos mecanismos y servicios de la plataforma con los que se ha enriquecido el modelo de programación básico en que se basa LwCCM, el modelo de contenedor-componente, para que la tecnología ofrezca garantías de tiempo real durante la ejecución de las aplicaciones.

Al final del capítulo se expone la adaptación de la extensión RT-D&C a la tecnología RT-CCM, o lo que es lo mismo, el modelo PSM de RT-D&C para RT-CCM.

En el capítulo 5 se explica el proceso que se debe seguir para diseñar las características de tiempo real de una aplicación basada en componentes. Sobre un proceso de diseño estándar y tradicional, el descrito en la especificación D&C, se introducen las nuevas tareas que deben llevar a cabo los diferentes agentes implicados en el proceso para elaborar el modelo de comportamiento temporal de la aplicación y extraer de su análisis la configuración que la lleva a cumplir los requisitos que se asignan en su especificación.

Finalmente, en el capítulo 6 se aborda la validación de la metodología propuesta, utilizando para ello una aplicación real. La aplicación de ejemplo se desarrolla para la tecnología Ada-CCM, que constituye una implementación de la tecnología RT-CCM sobre Ada 2005 y nodos ejecutando el sistema operativo MaRTE OS. Al principio del capítulo se destacan algunos aspectos característicos de dicha implementación. Posteriormente se explica el proceso de desarrollo de uno de los componentes utilizados en la aplicación, y a continuación el proceso de desarrollo de la aplicación completa por composición de instancias de componentes.

Finalmente, en el capítulo 7 se exponen las conclusiones del trabajo presentado en esta tesis, junto a las líneas de trabajo futuro que surgen a partir de ella.

Los anexos A, B y C se incluyen como información complementaria a los capítulos 2, 3 y 6, respectivamente. El anexo A muestra el metamodelo Mod-MAST completo, a través de una serie de diagramas de clase UML. El anexo B muestra el metamodelo CBS-MAST, que surge al adaptar Mod-MAST al modelado de aplicaciones basadas en componentes. Finalmente, el anexo C muestra algunos de los descriptores (en forma de ficheros .xml) utilizados en la aplicación de ejemplo que se utiliza en el capítulo 6.

2. Modelos de tiempo real orientados a la componibilidad

Cuando se diseña un sistema de tiempo real, se necesita poder predecir su comportamiento temporal bajo las situaciones de ejecución que se ensayan y se proponen y para las cuales hay definidos requisitos temporales. Éstos son siempre consecuencia de la interacción del sistema con un entorno externo que evoluciona en el tiempo físico con dinámica propia. La predicción del comportamiento temporal del sistema requiere analizar la información que describe la capacidad de evolución temporal del sistema y la que describe los escenarios que se pueden presentar en el entorno. Estas informaciones se pueden formular como dos modelos independientes, pero complementarios:

- El modelo de tiempo real que contiene la información que se necesita para estimar la evolución en el tiempo de las actividades que realiza la aplicación cuando es ejecutada. En él, se describe tanto la capacidad de procesamiento que se requiere para implementar la funcionalidad de la aplicación, como la capacidad de procesamiento efectiva disponible en la plataforma. Del balance entre ambas resulta la evaluación del comportamiento temporal del sistema.
- El modelo de carga de trabajo que describe estadísticamente los escenarios dinámicos de interacción que el entorno físico puede presentar. Contiene la descripción de los patrones con que pueden ocurrir las interacciones, así como las restricciones temporales que han de satisfacer las respuestas del sistema para que las interacciones sean correctas.

Los modelos de los sistemas de tiempo real actuales, que se ejecutan en plataformas distribuidas y sobre sistemas operativos y middlewares de tiempo real comerciales, contienen una información muy compleja. Para su gestión práctica debe utilizarse un entorno de diseño con un repositorio estructurado y estandarizado, en el que los modelos puedan ser almacenados y procesados por herramientas informáticas instaladas en él. Desde un punto de vista genérico, los modelos se procesan con dos objetivos:

- Analizar la planificabilidad de una aplicación, esto es, verificar que las actividades de la aplicación se van a ejecutar satisfaciendo los requisitos temporales que tienen especificados en la carga de trabajo.
- Evaluar los valores que deben ser asignados a los parámetros de configuración que influyen en la planificabilidad de la aplicación, a fin de que satisfaga sus restricciones temporales, y en la medida de lo posible, lo haga con la máxima holgura.

La necesidad del entorno y de las herramientas en el diseño de tiempo real se refuerza si se considera que el proceso de diseño es un proceso interactivo e iterativo. Como se muestra en la figura 2.1, en cada fase de iteración, los modelos se modifican con los resultados obtenidos por las herramientas en el paso anterior, o los cambios que introduce el diseñador en base a ellos. El proceso debe repetirse hasta que se consigue ajustar los parámetros de configuración del sistema de forma que se satisfagan sus requisitos temporales.

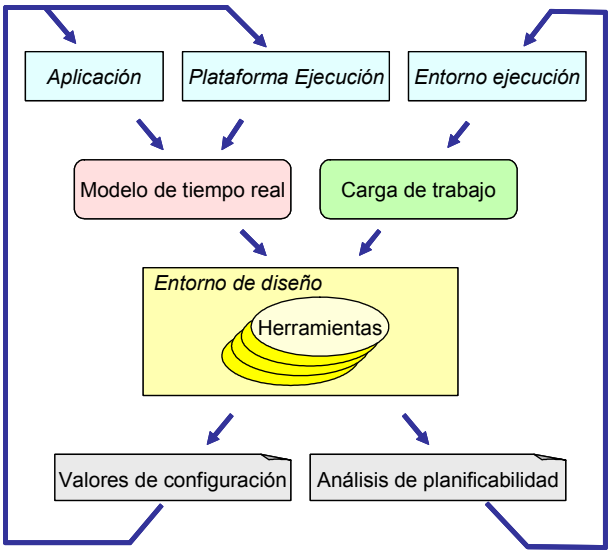


Figura 2.1: Proceso iterativo de diseño de un sistema de tiempo real.

En las estrategias clásicas de diseño de sistemas de tiempo real, el modelo transaccional constituye la base del proceso de diseño. Es la forma en la que el diseñador concibe la aplicación, y asimismo sirve de guía para definir la arquitectura de la aplicación y escribir su código. Sin embargo, esta estrategia no siempre es aplicable, ya que la complejidad de las aplicaciones y las plataformas de ejecución actuales hace imposible que el diseñador pueda conocer todos los detalles de un sistema que se necesitan para construir su modelo de tiempo real. En otros casos, el paradigma de diseño utilizado no deja al diseñador libertad para fijar la arquitectura y/o el código de todos los elementos del sistema, ya que se han de utilizar plataformas diseñadas para otros fines, o módulos legados que pueden ser opacos. En conclusión, para el diseño de los sistemas de tiempo real actuales se necesitan nuevas estrategias de construcción del modelo reactivo que no requieran el conocimiento completo del sistema. Se necesitan estrategias en las que el modelo de tiempo real pueda construirse por composición de los modelos de los módulos que forman el sistema, los cuales son elaborados por los diseñadores de los propios módulos, y posteriormente manejados por los diseñadores de las aplicaciones, que los utilizan sin conocer sus detalles internos.

En este capítulo se expone una metodología para construir el modelo de tiempo real transaccional de sistemas que presentan las siguientes características:

- Arquitectura modular aplicada en cualquiera de sus niveles: en el código, en la plataforma, en el sistema operativo, en el middleware, etc. Consideramos módulos a aquellos subsistemas previamente elaborados que el diseñador utiliza para construir el sistema sin tener que conocer sus detalles internos, y por lo tanto, sin tener capacidad de construir su modelo de tiempo real. La metodología de modelado que se propone proporciona recursos para que el diseñador de cada módulo, que es quien conoce todos sus detalles internos, pueda formular un modelo de tiempo real reutilizable del módulo. Asimismo, proporciona al diseñador del sistema que utiliza estos módulos para construirlo, procedimientos y herramientas para elaborar el modelo de tiempo real del sistema por composición de los modelos de los módulos que son parte de él.
- Aplicaciones muy complejas o que forman parte de sistemas muy complejos, y que por tanto, tengan un número de transacciones y recursos tan elevado que su modelo de tiempo real tenga que ser construido por un equipo de personas que elaboran independientemente

secciones del mismo. Para esta situación, la metodología proporciona recursos que permiten construir independientemente los modelos de cada una de las secciones, así como herramientas para su integración final.

En la siguiente sección se introducen los conceptos clave de la metodología. Se plantean de forma genérica, por lo que constituyen una solución universal que permite incorporar modularidad y reutilización a diferentes metodologías de modelado (siempre que estén basadas en un modelo transaccional). A continuación se expone la metodología Mod-MAST, como prueba de la aplicación de dichos conceptos a la extensión de una metodología concreta basada en una formulación puramente transaccional, en este caso MAST [GGP01] [MGD01][MAST].

2.1. Modelos orientados a la composición

2.1.1. Dualidad Descriptor- Instancia

Cuando se trata de modelar el comportamiento temporal de un módulo, no puede exigirse que el modelo describa el comportamiento temporal del módulo de forma completa y aislada, esto es, con independencia de los otros módulos con los que interacciona en el contexto de una aplicación, de la plataforma en la que ejecuta o del resto de aplicaciones que compiten con él por los recursos disponibles para la ejecución.

En la metodología de modelado de tiempo real que se propone, el modelo de un módulo (hardware o software) es una abstracción que describe todos aquellos aspectos relevantes de los elementos pertenecientes al módulo, que son necesarios para evaluar el comportamiento temporal de cualquier sistema que lo utilice. De esta definición se pueden deducir las características que debe satisfacer el modelo de tiempo real de cualquier módulo:

- Debe ser reutilizable: El modelo puede ser aplicado para describir el comportamiento temporal de cualquier sistema en el que se utilice el módulo. Esto requiere que toda la información que contiene el modelo corresponda a características que son consecuencia de los elementos contenidos en el módulo, y ninguna sea función de elementos de otros módulos con los que éste pueda interaccionar en el contexto del sistema.
- Debe ser completo: El modelo debe contener toda la información relativa a los elementos del módulo que pueda ser relevante para evaluar el comportamiento de cualquier sistema que lo utilice, independientemente de cómo lo haga. La completitud puede conducir a modelos muy complejos. Para evitarlo se utiliza la parametrización, que permite adaptar el modelo a diferentes modos de uso o configuraciones, sin tener que repetir o enumerar todos los modos de operación en que se pueda encontrar el módulo.
- Debe ser componible: El modelo de tiempo real de un módulo debe poder ser ensamblado con los modelos de los otros módulos que interaccionan con él, de forma que la composición resultante constituya el modelo completo del sistema al que la asociación de todos ellos da lugar. Para ello debe ser elaborado siguiendo una serie de reglas que hagan posible dicho proceso de composición.

De la definición del modelo de un módulo no se deriva que deba ser analizable por sí mismo de forma independiente. Esto conduce a que el término modelo que estamos utilizando no sea estrictamente correcto. El modelo de un módulo debe considerarse como una pieza o parte de un modelo, que no puede ser procesado para determinar el comportamiento temporal de los servicios del módulo hasta que no se encuentre ensamblado con los modelos del resto de módulos con los que el módulo interacciona en el contexto de un sistema.

En la figura 2.2, se muestra un ejemplo sobre las dependencias que presenta el modelo de un módulo. En ella se representa el módulo software *SoftMod_A*, que ofrece la operación *Oper_A,* cuya latencia se desea evaluar bajo una determinada carga de trabajo. En el modelo de tiempo real del módulo se describen los elementos incluidos en él: la capacidad de procesamiento que requiere la ejecución de las actividades internas *Act_1* y *Act_3*, así como la referencia a la actividad *Act_2*, cuyo código pertenece a otro módulo externo a él. La latencia de *Oper_A* no puede evaluarse procesando independientemente el modelo del módulo *SoftMod_A* que lo describe, sino que éste debe ser procesado conjuntamente con el modelo del procesador *Proc_1* (en que se ejecuta *SoftMod_A*), del módulo *SoftMod_B* (en el que está definida *Act_2*), del procesador *Proc_2* (en que se ejecuta *SoftMod_B*), de la red *Net* (a través de la que se invoca *Act_2*), de las aplicaciones *Appl_1* y *Appl_2* (que compiten por recursos compartidos en *Proc_1*) y de la carga de trabajo que define el patrón de invocación de la operación.

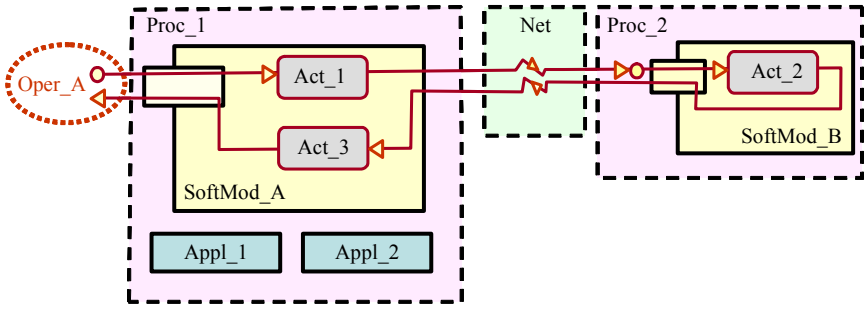


Figura 2.2: Dependencias del modelo de comportamiento temporal de un módulo

A fin de formalizar la relación entre el modelo reutilizable que proporciona la información de tiempo real de un módulo, y el modelo analizable del módulo que resulta en el contexto de un sistema en la que se encuentra ensamblado, se utilizan los conceptos de descriptor e instancia de modelo:

- Un descriptor de modelo es una plantilla parametrizada, que contiene la información completa de todas las características relativas al comportamiento temporal de un módulo que son relevantes para estimar el comportamiento temporal de cualquier instancia de ese módulo que participe en un sistema. En el descriptor del módulo se incluyen todas las características que tienen su origen en los elementos propios del módulo, y que están presentes en todas las instancias del módulo que se puedan declarar. Los parámetros definidos en un descriptor pueden ser de dos tipos:
  - Referencias a los modelos de tiempo real de aquellos otros módulos hardware o software que interaccionan con el módulo que se modela.
  - Características que cambian en cada instancia del módulo, bien por el modo en que se hace uso de él, o por la configuración específica que se le asigna en el contexto de una aplicación.

  En el descriptor de un módulo no se incluye ninguna información que sea propia de la aplicación en la que se utiliza, de la plataforma en que se ejecuta o del comportamiento de otros módulos de los que hace uso para implementar su funcionalidad.
- Una instancia de modelo es una parte del modelo analizable completo de un sistema, que describe el comportamiento temporal de una instancia de un módulo en el contexto de ejecución concreto que se analiza. La instancia de modelo se genera tomando como referencia el descriptor de modelo del módulo, y asignando a todos los parámetros que el

descriptor tiene definidos, valores concretos y/o referencias a otras instancias de modelo concretas. Los valores que se asignan a los parámetros se deducen del contexto del sistema en el que se incluye la instancia del módulo.

En el proceso de desarrollo de un sistema con características de tiempo real y construido por composición de módulos reutilizables, deberán existir mecanismos que permitan incluir como parte de la información que describe el módulo, aquella que da acceso al descriptor de su modelo de tiempo real. El concepto de descriptor es un concepto persistente, representa un recurso que va a ser almacenado en el repositorio de la plataforma de diseño junto con el resto de recursos necesarios para hacer uso de un módulo en un sistema. Por el contrario, el concepto de instancia de modelo es efímero, va a ser empleado únicamente por las herramientas de composición encargadas de generar el modelo final de un sistema en base a la descripción de su arquitectura.

Como ejemplo para clarificar el uso de descriptores e instancias utilizamos un módulo software, *SoundGenerator,* que se representa en la figura 2.3. Desde el punto de vista funcional, el módulo implementa la interfaz *iPlayer*, que representa un servicio para la generación de sonidos. Su modelo de tiempo real deberá describir el comportamiento temporal de los métodos que implementan las operaciones declaradas en la interfaz, esto es, el uso de recursos generado cuando se invoca cada uno de ellos, que puede afectar al comportamiento temporal de otros módulos que usen los mismos recursos:

- El método *fail* se utiliza para generar un sonido predefinido. Se modela mediante una actividad que se ejecuta completamente a través de código incluido en el módulo, y por tanto, toda la información necesaria para predecir su comportamiento temporal está incluida en el descriptor. En este punto aparece una dependencia implícita con el modelo de procesador en que el módulo sea instalado, pues el tiempo real de ejecución del método depende de la velocidad del procesador en que sea ejecutado, por tanto sólo puede ser calculado cuando se tenga acceso al modelo de tiempo real de dicho procesador. Esta dependencia con el modelo del procesador va a existir por defecto en cualquier descriptor de modelo de un módulo software.
- El método *play* genera el sonido que se pasa como argumento, y su ejecución incluye la invocación del método *log* en un módulo externo que implemente la interfaz *iLogger*. El comportamiento temporal de dicho método no puede quedar definido completamente con los datos incluidos en el descriptor del módulo, y no será conocido hasta que en el contexto de una aplicación concreta se defina la instancia específica de módulo servidor que se usa. Por ello, en el modelo del método *play* se incluye como parámetro una

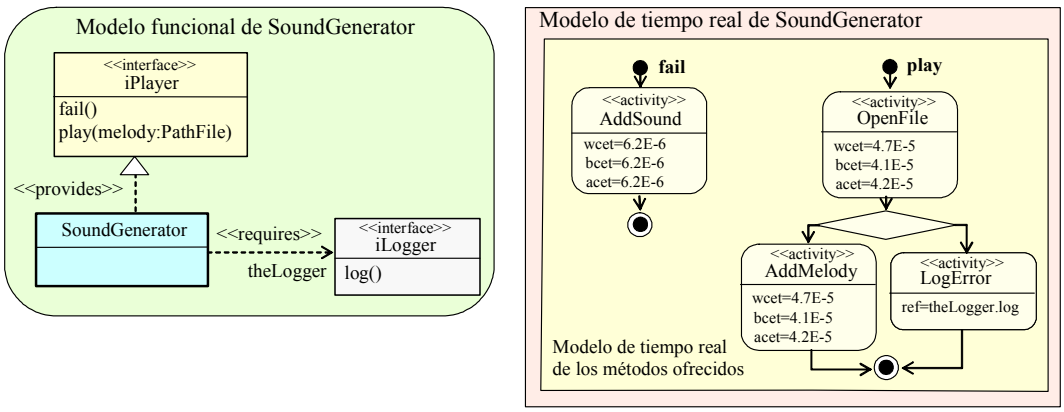


Figura 2.3: Ejemplo de descriptor de modelo de un módulo software

referencia al modelo del método *log*, que únicamente podrá ser resuelta cuando se conozca la instancia de módulo servidor concreta que se va a utilizar. Cualquier módulo al que una instancia de *SoundGenerator* acceda a través de la referencia *theLogger*, debe incluir el modelo de tiempo real del método *log*, de manera que se pueda generar la instancia de modelo final del método *play*.

El descriptor de modelo correspondiente al módulo *SoundGenerator*, *SoundGenerator_Model*, se almacena en el repositorio para poder ser utilizado en cualquier aplicación en la que se haga uso del módulo. Por ejemplo, la aplicación que se muestra en la parte derecha de la figura 2.4, formada por una instancia de tipo SoundGenerator, *mySoundGenerator*, que hace uso de otra instancia de un módulo de tipo *Logger*, *myLogger* (suponemos que un módulo de tipo *Logger* implementa la interfaz *iLogger*). Ambas se instancian en el procesador *myProc*, que es un procesador tipo PC ejecutando el sistema operativo MaRTE OS. Los descriptores de cada uno de los elementos de la aplicación se encuentran almacenados en el repositorio. En base a la descripción de la aplicación se generan y enlazan las correspondientes instancias de modelo, a partir de las que se genera el modelo final de la aplicación. Por ejemplo, para poder generar el modelo final del método *play* ofrecido por la instancia *mySoundGenerator*, su correspondiente instancia de modelo, *mySoundGeneratorModel*, accede a la instancia de modelo *myLoggerModel* para conocer el modelo del método *log* concreto que se está invocando. Además, para evaluar los tiempos físicos de ejecución de ambos métodos, las instancias de modelo necesitan acceder al modelo del procesador, *myProcModel*.

Los siguientes apartados describen el tipo de información que se incluye en los diferentes descriptores de módulos (software o hardware) para poder llevar a cabo este proceso de evaluación de los tiempos de respuesta, y por tanto, de evaluación de la planificabilidad de los sistemas.

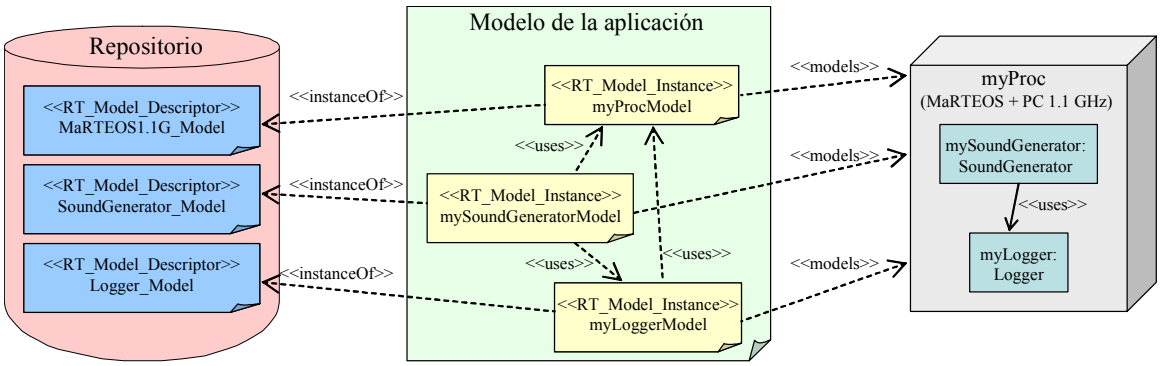


Figura 2.4: Conceptos de descriptor e instancia de modelo

2.1.2. Modelo de tiempo real reutilizable de un módulo software

En esta sección se define la información que ha de contener el descriptor de modelo de un módulo software para que en el contexto de una aplicación se pueda generar la instancia de modelo que permite predecir su comportamiento temporal.

La metodología de modelado que se propone corresponde a una estrategia de formulación basada en modelo transaccional. En base a ello, el descriptor de un módulo software se compone de los siguientes elementos:

- Modelos de comportamiento temporal de los métodos que ofrece el módulo y que pueden ser invocados desde otros módulos durante la ejecución de una transacción. Para cada

método se describe el uso de recursos de la plataforma que el código del módulo realiza cuando es invocado, así como las relaciones de sincronización entre los estados del método, y los de otros que pueden ser invocados concurrentemente con él.

El comportamiento temporal de un método se describe a través del conjunto de actividades (*Step*) que se ejecutan en respuesta a su invocación. Como muestra la figura 2.5, hay dos tipos de actividades:

- Actividades internas: Son las actividades que corresponden a la ejecución de bloques de código pertenecientes por entero al módulo, cuya descripción completa y final se puede incluir en el descriptor al ser conocidos cuando el módulo es implementado. Son ejemplos de este tipo de actividades internas las actividades *A11* o *B11* de la figura 2.5, o el método *fail* que vimos en el ejemplo de la figura 2.3. Una actividad interna se describe mediante una variable probabilística que describe la capacidad de procesamiento que conlleva su ejecución en el peor, mejor y caso promedio, y la lista de recursos compartidos (de acceso protegido) a los que ha de acceder para poder ser ejecutada.
- Actividades externas: Corresponden a métodos ofrecidos por módulos externos que se invocan en el código del método para implementar su funcionalidad. Es el caso de las actividades *A12* o *B12* en la figura, que referencian los métodos *B1* (en *Module_B*) y *C1* (en *Module_C*), respectivamente. En este caso, en el descriptor de *Module_A* sólo se incluye la referencia al modelo del método que se invoca. Esta referencia será resuelta en el contexto de una aplicación cuando se conozca la instancia de módulo concreta que se utiliza como servidor (como ocurría con la referencia al método *log* en el ejemplo de la figura 2.3).

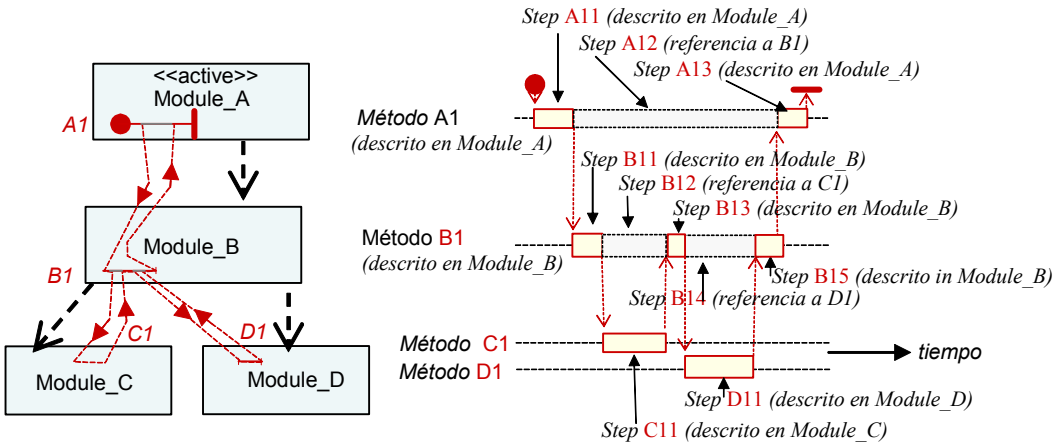


Figura 2.5: Actividades internas y externas en el modelo de un servicio o transacción

- Modelos de las transacciones que se pueden iniciar en el módulo. Existen módulos software, que denominamos activos, que tienen capacidad de iniciar tareas en el sistema en respuesta a eventos externos procedentes del entorno o a eventos temporizados que gestiona el propio módulo. Aplicando una formulación transaccional a los modelos, estas tareas se describen como transacciones, y por lo tanto, este tipo de módulos deberán incluir como parte de su modelo, los descriptores de las transacciones que potencialmente pueden generar. El descriptor de una transacción contiene la declaración de:
  - El patrón de activación, que describe la cadencia de los tiempos en los que se inicia la transacción.

- La declaración del conjunto de actividades que constituye la tarea que se ejecuta en el sistema en respuesta al evento que lanza la transacción. Como ocurre con los métodos, algunas de las actividades ejecutadas dentro de una transacción corresponden a la ejecución de código propio del módulo y se definen completamente en su descriptor, mientras que otras representan invocaciones de métodos externos y sólo se incluye la referencia al modelo del método invocado.
- La identificación de los puntos del flujo de la transacción donde se pueden imponer restricciones temporales. Éstas se declaran como monitores de tiempo, a los que se les puede asignar diferentes requisitos temporales a evaluar.

El descriptor de una transacción puede declarar parámetros que hagan referencia a cualquiera de sus elementos, como el patrón de lanzamiento, las actividades a ejecutar o los requisitos temporales que se quieren verificar. Como ejemplo, el modelo de tiempo real del módulo *SoundGenerator,* además de los modelos de los métodos ofertados que vimos anteriormente, debe incluir el modelo de la transacción *SoundTransaction*. Esta transacción modela la tarea que el módulo ejecuta internamente para poder generar los sonidos sin necesidad de que el módulo cliente (que ordena la ejecución de un sonido) tenga que quedar suspendido hasta que el sonido es generado. Como vemos en la descripción (no formal) que se muestra en la figura 2.6, la transacción se ejecuta de forma periódica, con periodo parametrizable *soundThreadPeriod*, y consiste en la ejecución de dos actividades internas, *GetNextNote* y *UpdateSound*. La transacción tiene definido un requisito temporal de tipo global, *Updated*, cuyo plazo de ejecución es también configurable, tomando el mismo valor que el valor asignado al periodo. Cuando se instancie un módulo de tipo *SoundGenerator* en una aplicación, esta transacción influirá en su comportamiento temporal, por lo que una instancia de modelo de la transacción debe ser incluida en el modelo de la carga de trabajo de la aplicación. Para caracterizar dicha instancia deberá asignarse un valor concreto al parámetro *soundThreadPeriod*.

- Modelos de los recursos de concurrencia y de sincronización utilizados por el módulo. Los métodos ofrecidos por un módulo pueden ser invocados concurrentemente por diferentes clientes, y en consecuencia, pueden requerir la existencia de recursos de

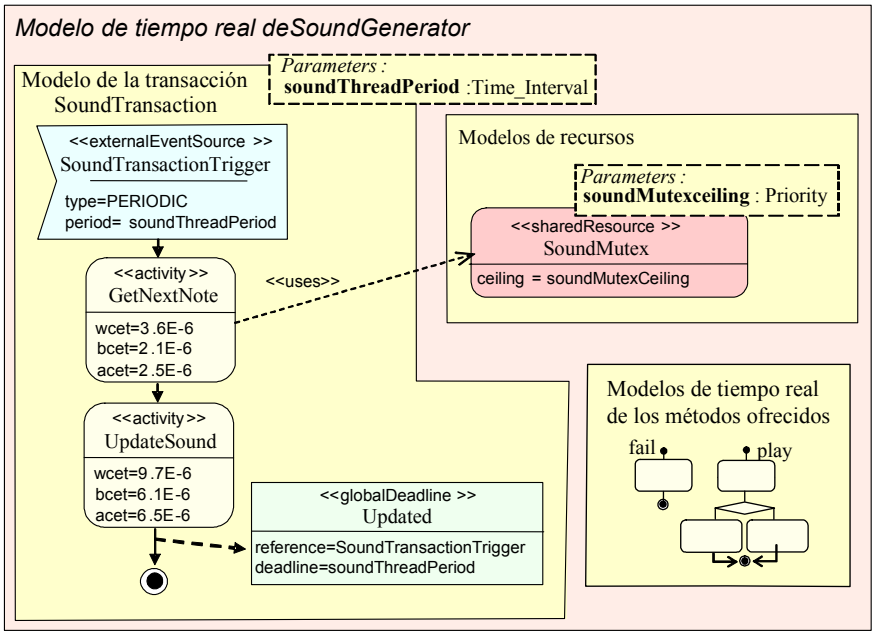


Figura 2.6: Ejemplo de modelo de transacción en el descriptor de un módulo software

concurrencia (threads), y/o recursos de sincronización (mutex, objetos protegidos, etc.) que son utilizados para gestionar internamente la ejecución concurrente del código propio. En la figura 2.6 se muestra como la ejecución de la actividad interna *GetNextNote* debe realizarse en modo protegido, o lo que es lo mismo, implica el acceso a un recurso compartido. El modelo de este recurso, *SoundMutex*, debe ser incluido también en el descriptor de modelo del módulo, de manera que cada vez que una instancia del módulo se añada a una aplicación, una instancia de este elemento se añada al modelo de tiempo real de la aplicación. Los recursos definidos en el descriptor de un módulo pueden ser parametrizados, como ocurre con el recurso *SoundMutex*, cuyo techo de prioridad, *soundMutexCeiling*, se define como configurable. Los valores de dichos parámetros son asignados a la instancia de modelo del módulo cuando ésta es declarada dentro del modelo de una aplicación.

2.1.3. Modelo de tiempo real reutilizable de una plataforma de ejecución

La respuesta temporal de los servicios ofrecidos por cualquier módulo software depende de manera directa de las características de la plataforma en que el módulo es ejecutado. Para conseguir que los modelos de comportamiento temporal de los módulos software sean reutilizables en el contexto de las diferentes aplicaciones que los usan, es necesario formular independientemente las características que son propias de su código, de las que son propias de la plataforma de ejecución. El comportamiento temporal de un módulo software presenta dos dependencias principales del modelo de la plataforma en la que es instalado en el contexto de una aplicación:

- El tiempo de ejecución de las actividades del módulo es función de la capacidad de procesamiento que proporciona el procesador en que se instancia el módulo.
- Las características de los servidores en los que se planifican las actividades del módulo deben ser compatibles con la naturaleza del planificador del procesador que los gestiona.

La separación de los aspectos que determinan el tiempo de ejecución de una actividad entre aquellos que son responsabilidad del módulo software y los que son responsabilidad de la plataforma de ejecución, se puede formular siguiendo dos estrategias diferentes:

- El modelo del módulo software describe la cantidad de procesamiento que se requiere para ejecutar las actividades de los métodos ofrecidos por el módulo en forma de ciclos de reloj, de instrucciones máquina a ejecutar, de longitud del mensaje a transmitir, etc. El modelo de la plataforma describe la capacidad que ofrecen los recursos de procesamiento en forma de instrucciones máquina por segundo, ancho de banda, etc. De la combinación de ambas informaciones se puede deducir el tiempo físico de uso de los recursos que se requiere para ejecutar cada una de las actividades.
- La cantidad de procesamiento que requiere la ejecución de una actividad se formula en el modelo del módulo software como un tiempo normalizado (*normalized execution time*), que representa el tiempo físico que se requiere para ejecutar la actividad en un recurso que se toma como recurso de referencia. La capacidad de un recurso perteneciente a una plataforma concreta se caracteriza a través de un atributo, denominado *speedFactor*, que se define como la razón entre el tiempo que tarda en ejecutar cualquier actividad en el recurso de referencia, que tiene como valor de *speedFactor* 1.0, y el tiempo que tarda en el propio recurso. De esta forma, el tiempo físico en el que la actividad necesita hacer uso de un determinado recurso, resulta del cociente del tiempo de ejecución normalizado de la actividad entre el *speedFactor* del recurso en que sea ejecutada.

$$executionTime = (normalizedExecutionTime)/(speedFactor)$$

Esta segunda formulación es la que se utiliza en la metodología MAST y la que se utiliza también en esta tesis.

La capacidad de procesamiento de una plataforma es función tanto de la capacidad aportada por los procesadores y la anchura de banda aportada por las redes de comunicación, como de la reducción que se produce en dicha capacidad debido al consumo de la misma que generan los procesos internos de la propia plataforma, como atención a interrupciones, gestión del tiempo, procesos del sistema operativo, etc. La diferencia entre ambos valores determina la capacidad de procesamiento efectiva de la plataforma, que es la que influye en la evaluación de los tiempos de respuesta de los servicios de los módulos involucrados en una aplicación. Para modelar estos aspectos, los elementos que se incluyen en el modelo de una plataforma de ejecución, como muestra la figura 2.7, son:

- Modelos de los procesadores: Describen la capacidad de procesamiento ofrecida por un nodo, derivada del procesador y del hardware que integra.
- Modelos de los sistemas operativos: Describen las características de administración de los recursos software, y del costo en capacidad de procesamiento que supone su gestión: estrategias de planificación, capacidad de procesamiento que consume la gestión de procesos, etc.
- Modelos de las redes de comunicación: Describen la anchura de banda que proporciona una red, el nivel de granularización en paquetes de los mensajes que transfiere y la capacidad de planificar la transferencia concurrente de mensajes.
- Modelos de los *drivers* de comunicación: Describen el consumo de capacidad de procesamiento que se produce en los procesadores que participan en una comunicación como consecuencia de la atención y gestión de los protocolos de red que se usan.
- Modelos del middleware o software de intermediación: Describen el comportamiento temporal y la capacidad de procesamiento que consumen los servicios con que están dotados los nodos para la implementación de comunicaciones remotas.

La formulación del modelo que describe las características de comportamiento temporal de la plataforma a fin de cuantificar el comportamiento temporal de las aplicaciones que se ejecutan en ella, es básicamente la misma en aplicaciones basadas en estrategias modulares que en aplicaciones desarrolladas de manera directa. El aspecto que se añade cuando se trata de adaptar

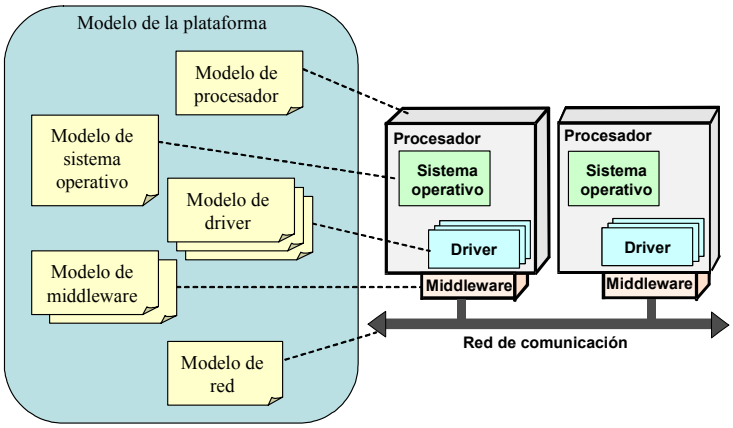


Figura 2.7: Modelo de tiempo real de plataformas de ejecución

el modelado de plataformas a una estrategia modular es nuevamente el de la parametrización y la reutilización. Los elementos de la plataforma se tratan también como módulos, para los que se pueden elaborar descriptores de modelo parametrizados que se almacenan en el repositorio de diseño. Cuando se define una plataforma concreta para ejecutar una aplicación, se construye su modelo de tiempo real componiendo los modelos de los elementos que la forman: tipo de procesador, sistema operativo soportado por el procesador, protocolo de red, software de comunicación a través del que implementan las interacciones remotas, etc. La instancia de modelo de cada elemento se declara referenciando un descriptor previamente definido, y asignando valores concretos a los parámetros que éste tenga definidos.

La parametrización permite, por ejemplo, elaborar un único descriptor de modelo de un nodo PC ejecutando el sistema MaRTE OS, independiente de la velocidad del procesador. Para ello, será necesario definir el *speedFactor* como parámetro del descriptor (considerando *speedFactor*=1.0 para una velocidad concreta). Después, cuando en una plataforma se use un nodo de velocidad distinta al de referencia, bastará con asignar el valor de *speedFactor* que corresponda en la declaración de la instancia de modelo. La parametrización nos permite, en este caso, reutilizar un mismo descriptor para modelar muchos procesadores diferentes que sólo se diferencien en un conjunto reducido de características.

2.2. Modelo de tiempo real de un sistema con estructura modular

2.2.1. Modelo de una situación de tiempo real

Cuando se aplica una metodología de modelado transaccional, el contexto de análisis y diseño de un sistema de tiempo real lo representan las situaciones de tiempo real, designadas en la bibliografía como *Real-Time Situation* [SPT][GGP01] o *Analysis Context* [MARTE]. Una situación de tiempo real representa un modo de operación de un sistema para el que existen requisitos de tiempo real especificados. Para cada situación de tiempo real se elabora un modelo completo e independiente, que puede ser procesado por las herramientas de análisis de planificabilidad o de estimación de respuesta. Certificar la planificabilidad de un sistema implica analizar el sistema en todas y cada una de las situaciones de tiempo real en las que pueda ser ejecutado.

El modelo de una situación de tiempo real se define a través de dos descripciones complementarias: la descripción de la aplicación, que especifica el uso de recursos que requiere su ejecución bajo las condiciones de carga de trabajo que se consideran; y la declaración de la plataforma, que especifica la capacidad de ejecución que ofrecen los recursos que constituyen la plataforma. El análisis de planificabilidad de una situación de tiempo real requiere tener bien definida la carga de trabajo concreta para la que se realiza, descrita a través de un modelo reactivo basado en transacciones, y en la que se definen los patrones de activación y las restricciones temporales que deben ser satisfechas.

Si interpretamos esta definición desde el punto de vista de una aplicación de tiempo real con una estructura interna constituida por módulos software, cada situación de tiempo real queda definida por:

- El conjunto de instancias de módulos software que la forman, cada uno de ellos con una configuración concreta e interaccionando entre sí.

- La plataforma en la que se ejecuta, incluyendo todos los recursos que se utilizan y la configuración que se asigna a cada uno de ellos.
- El despliegue de las instancias de los módulos en la plataforma de ejecución, esto es, la asignación de cada módulo al nodo procesador en el que ejecuta, junto con la definición de los mecanismos de comunicación a través de los que los nodos se comunican.
- La carga de trabajo que se impone a la ejecución de la aplicación en dicha situación.

A partir de esta información, la formulación del modelo de la situación de tiempo real requiere dos tareas:

- La primera concierne al modelado de la aplicación desde el punto de vista estructural, y consiste en declarar y enlazar las instancias de modelo que describen el comportamiento temporal de cada elemento software y hardware que participa en la ejecución de la aplicación.
- La segunda consiste en el modelado de la aplicación desde el punto de vista reactivo, esto es, la declaración de la carga de trabajo que debe atender la aplicación en dicho modo de operación, y los plazos temporales impuestos en su ejecución.

Como muestra la figura 2.8, el modelo de una situación de tiempo real se organiza con una estructura de elementos de modelado que se corresponde fielmente con la estructura modular con la que se construye el sistema. La situación de tiempo real se define por:

- El modelo de la aplicación, que agrupa las instancias de modelo de cada uno de los módulos software que participan en ella. De acuerdo a la configuración de la aplicación, se asignan valores a los parámetros definidos en los correspondientes descriptores y a las referencias entre instancias de modelos, de manera que cada uno puede acceder a la información de otras instancias de las que depende.
- El modelo de la plataforma, del que siempre depende el modelo de la aplicación. Entre otros aspectos, el descriptor de modelo de cualquier módulo software siempre tiene una referencia al procesador en el que el módulo es instanciado. El modelo de la plataforma esta formado por las instancias de modelo de todos los recursos hardware y software que integran dicha plataforma.
- El modelo de la carga de trabajo, que queda definida por el conjunto de transacciones que se ejecutan concurrentemente en la situación que se modela. En el caso de una aplicación modular, dicho conjunto está formado por las instancias de modelo de las transacciones

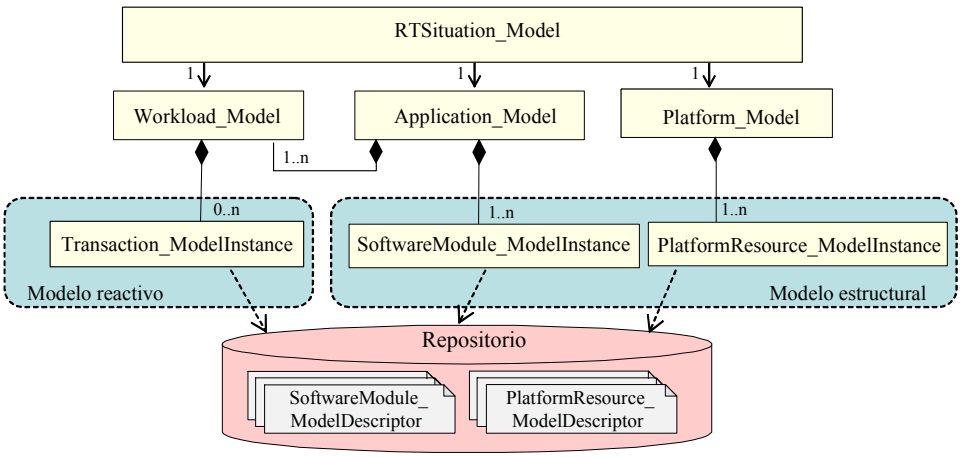


Figura 2.8: Modelo de una situación de tiempo real en una aplicación modular

que son iniciadas por sus módulos activos. En la sección 2.1.2 se explicó como el descriptor de modelo de un módulo activo debe incluir modelos de aquellas transacciones que el módulo es capaz de gestionar. Cada transacción describe su patrón de disparo, posiblemente parametrizado, el cual representa la carga real de trabajo que la aplicación soporta debida a dicha transacción. Asimismo la transacción es el marco en el que se definen los requisitos temporales que se establecen para la ejecución de la aplicación.

Las transacciones que forman parte de la carga de trabajo de una situación de tiempo real se pueden clasificar en dos grupos, atendiendo a su origen:

- Algunas son inherentes a los modelos de los módulos que constituyen la aplicación y la plataforma, por lo que el diseñador no tiene que declararlas explícitamente. Corresponden a aquellas transacciones que serán ejecutadas en la aplicación como consecuencia de la instanciación y activación del módulo. Cada vez que una instancia del módulo es añadida a una aplicación, estas transacciones son también agregadas automáticamente al modelo de la situación de tiempo real, constituyendo parte de la carga de trabajo correspondiente a dicho modo de operación.
- Otras son función de la situación de tiempo real que se modela. Son declaradas explícitamente por el diseñador que construye el modelo de tiempo real, y para ello, hace uso de los descriptores parametrizados de las transacciones que se encuentran declarados en los descriptores de los módulos activos. El diseñador asigna valor a los parámetros definidos para cada transacción de acuerdo con sus características dentro de la situación de tiempo real modelada. Esta agrupación de instancias de transacciones forma el modelo de carga que se muestra en la figura 2.8, *Workload_Model*. La figura muestra como una aplicación puede soportar, y por tanto definir, diferentes cargas de trabajo; cada situación de tiempo real referencia a una sola de dichas cargas. Para certificar de manera total su planificabilidad, la aplicación debe ser analizada en cada situación de carga posible.

2.2.2. Modelo de tiempo real de un sistema

Con la información que se incluye en el modelo de una situación de tiempo real aislada, se puede analizar el comportamiento de una aplicación sólo en el caso de que sea la única que se está ejecutando en la correspondiente plataforma. Sin embargo, el comportamiento temporal de una aplicación depende del resto de aplicaciones que ejecutan concurrentemente con ella, y que por tanto, compiten por el acceso a los recursos de la plataforma. Por ello, para analizar el comportamiento de una aplicación, realmente ha de analizarse el modelo del sistema completo, esto es, de la plataforma de ejecución junto con el conjunto completo de aplicaciones que se ejecutan en ella, cada una en un determinado modo de operación.

El modelo de tiempo real de un sistema completo, como se muestra en la figura 2.9, está formado por:

- El modelo de la plataforma en la que se ejecuta el sistema.
- El modelo de todas las aplicaciones de tiempo real que se estén ejecutando concurrentemente en la plataforma. Cada una de ellas debe incluir la descripción de las diferentes cargas de trabajo que puede soportar.
- El modelo de cada situación de tiempo real del sistema completo a analizar. Cada situación corresponde con una posible combinación de las diferentes cargas de trabajo que pueden soportar las aplicaciones, incluyéndose siempre una sola carga por aplicación.

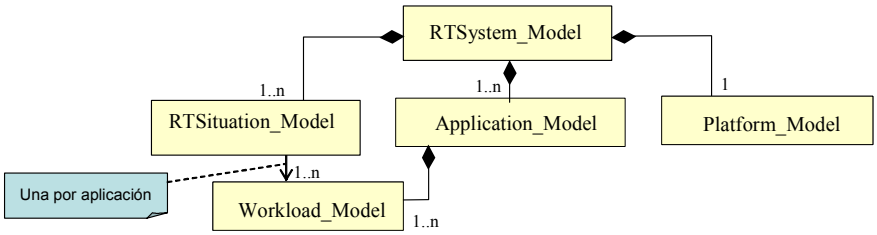


Figura 2.9: Modelo de un sistema de tiempo real

Los modelos de las aplicaciones y de la plataforma se definen del mismo modo que en el caso anterior. Para cada situación de tiempo real se genera un modelo completo y analizable del sistema. Será necesario analizar cada uno de esos modelos para certificar la planificabilidad del sistema en todos sus posibles modos de ejecución.

2.2.3. Generación del modelo de tiempo real de un sistema modular

El modelo analizable de un sistema de tiempo real con estructura modular es el modelo de instancias que se construye cuando ya se ha diseñado el sistema completo y se conocen:

- Las instancias de modelo que constituyen el modelo de la plataforma.
- Las instancias de modelo que constituyen el modelo de la aplicación o aplicaciones.
- El ensamblado entre las instancias de modelo que es consecuencia de la estructura de las aplicaciones y de como éstas se instancian en la plataforma.
- El modelo reactivo constituido por las instancias de modelo de las transacciones que se ejecutan en los elementos activos, que han sido configuradas de acuerdo con el contexto de la aplicación.

El modelo de instancias es construido cada vez que se analiza el comportamiento temporal del sistema. Es un modelo efímero que se genera mediante un proceso sistemático cada vez que se modifica el sistema. La figura 2.10 resume este proceso. Una vez que la aplicación es diseñada, esto es, se cuenta con las descripciones de su arquitectura modular, de su despliegue sobre una plataforma y de la carga de trabajo que soporta, se puede formular el modelo de tiempo real del sistema tal y como se expuso en el apartado anterior (ver figura 2.9). Para poder obtener este modelo:

- En la descripción de la arquitectura de la aplicación, por cada instancia de módulo que forme parte del sistema habrá que incluir la referencia a su correspondiente descriptor de modelo de tiempo real, junto con la asignación de valores concretos a sus parámetros. Esta información será la que tome la herramienta para declarar las correspondientes instancias de modelo de los módulos software. Las conexiones entre módulos, especificadas también en la arquitectura, se utilizarán para resolver automáticamente las referencias a instancias de modelo requeridas.

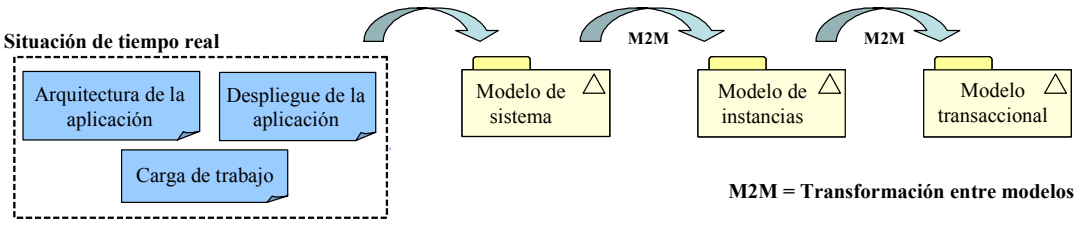


Figura 2.10: Generación del modelo de tiempo real de un sistema con estructura modular

- En la descripción del despliegue, por cada recurso que forme parte de la plataforma de ejecución, habrá que incluir la referencia a su correspondiente descriptor de modelo de tiempo real, junto con la asignación de valores concretos a sus parámetros. Esta información será la que tome la herramienta para declarar las correspondientes instancias de modelo de los recursos de la plataforma. Además, el despliegue debe incluir para cada instancia de módulo software, la referencia al recurso de la plataforma en que se instala. A través de dicha referencia, se resuelven las dependencias que los modelos de los módulos software presentan respecto a los modelos de la plataforma en la que ejecutan.
- En base a la descripción de la carga de trabajo, se declararán las correspondientes instancias de modelo de las transacciones ejecutadas en cada uno de los módulos activos que formen parte de la aplicación.

A partir del modelo del sistema formulado en base a elementos de alto nivel (declaración de instancias de modelo de módulos software y de elementos de la plataforma) una herramienta de transformación genera el modelo de instancias del sistema. Para ello utiliza la información de cada uno de los descriptores referenciados, que se encuentran almacenados de forma persistente en el repositorio del sistema de desarrollo. Al finalizar esta fase, se cuenta con un modelo final, en el que cada elemento (instancia) que forma parte de él se encuentra totalmente resuelto, con valores asignados a todos sus parámetros, y por tanto, se trata de un modelo analizable.

La metodología de modelado de tiempo real que se presenta está orientada a la reutilización y a la componibilidad, por lo que el modelo de instancias que se obtiene tiene una organización interna que es reflejo de la estructura de módulos con la que se diseña una aplicación. Sin embargo, estos modelos no son válidos para aplicar técnicas de análisis de planificabilidad sobre ellos directamente, ya que no existen técnicas que se basen en estrategias de formulación del modelo modulares. A fin de realizar el análisis y diseño de la aplicación, se necesita transformar el modelo a una formulación puramente transaccional, que sea compatible con las técnicas clásicas de análisis. Para ello, como muestra la figura 2.10, se aplica una nueva transformación, que transforma el modelo de instancias en un modelo transaccional puro, acorde a alguna metodología de modelado tradicional.

Las dos herramientas de transformación que se utilizan en la generación del modelo de un sistema modular dependerán de la metodología concreta de modelado que se utilice. En nuestro caso, la metodología Mod-MAST, que se expone a continuación. En la sección 2.3.3 se explican con más detalle los procesos de transformación que se llevan a cabo para este caso.

2.3. Mod-MAST: Extensión de la metodología MAST al modelado modular

En las secciones previas de este capítulo se han definido desde un punto de vista conceptual, los principios de una metodología de modelado cuyo fin es generar el modelo de comportamiento temporal de una aplicación por composición de los modelos de los elementos que la forman. En esta sección, se propone la metodología Mod-MAST, que sigue estos principios y que permite formular los modelos de aplicaciones basadas en composición de módulos.

La metodología Mod-MAST ha sido concebida como una extensión de la metodología de modelado de tiempo real MAST:

- Utiliza los principios de modelado de MAST tanto a alto nivel, al basarse en un modelo transaccional del sistema; como a bajo nivel, pues utiliza directamente las primitivas de

modelado definidas en MAST para describir el comportamiento temporal de los elementos hardware y software de bajo nivel del sistema.

- Hace uso de las herramientas que ofrece el entorno MAST tanto para el análisis de planificabilidad, como para el diseño de los parámetros de configuración que hacen planificable una aplicación.

Por tanto, Mod-MAST extiende a MAST, dotándolo de elementos de modelado que proporcionan un mayor nivel de abstracción, necesario cuando se quiere aplicar la metodología a aplicaciones desarrolladas siguiendo estrategias de composición modular. A lo largo de este apartado se definen la semántica y la estructura de los elementos de modelado que forman su metamodelo, así como las formas en que éstos se asocian para construir un modelo de tiempo real acorde al entorno MAST.

Aunque la metodología concreta que se presenta constituye una extensión de MAST, los conceptos y soluciones que se proponen son igualmente aplicables a otras metodologías de modelado. En particular, a aquellas que se basan en conceptos de modelado similares a los propuestos en el nuevo estándar de OMG para el modelado y análisis de sistemas empotrados y de tiempo real, el perfil MARTE [MARTE]. Con este objetivo, y como se explicará en la sección 2.4, se ha tratado de adaptar lo más fielmente posible la semántica de los elementos de modelado utilizados en Mod-MAST a la de los conceptos propuestos en dicho estándar.

2.3.1. Metamodelo Mod-MAST

La metodología Mod-Mast queda definida por un metamodelo formulado en UML. La figura 2.11 muestra la estructura de paquetes en que se han organizado los elementos del metamodelo:

- El paquete *rtmod_Core* contiene los elementos que constituyen la base conceptual común sobre la que se define el resto de elementos de modelado con los que se construyen los modelos de módulos y aplicaciones. Incluye el concepto dual de descriptor e instancia de

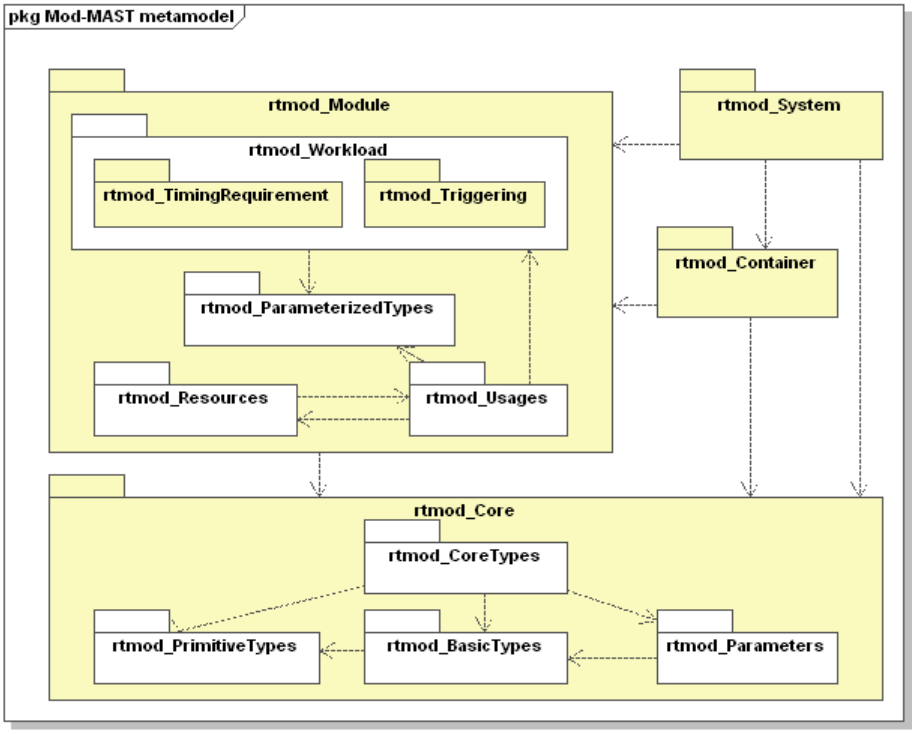


Figura 2.11: Estructura de paquetes del metamodelo Mod-MAST

modelo, los tipos necesarios para definir parámetros y los tipos básicos para la declaración de tiempos, capacidades, prioridades, etc.

- El paquete *rtmod_Module* define las primitivas de modelado de bajo nivel que se pueden utilizar para describir internamente el comportamiento temporal de los módulos software, de los recursos de las plataformas hardware/software, y de las cargas de trabajo de las aplicaciones.
- El paquete *rtmod_Container* define elementos contenedores, que sirven para organizar los modelos con una estructura similar a la que tienen la aplicación desde el punto de vista de la arquitectura software y el despliegue.
- El paquete *rtmod_System* define los elementos de modelado que permiten describir aplicaciones construidas por composición de módulos software de aplicación, desplegados en plataformas de ejecución multiprocesadoras y distribuidas.

A lo largo de esta sección se van a introducir los elementos principales del metamodelo, resaltando únicamente sus características más generales. El metamodelo completo y con todos sus detalles se expone en el anexo A[1].

2.3.1.1. Paquete rtmod_Core: Núcleo del metamodelo

La metodología de modelado Mod-MAST se basa en la dualidad descriptor-instancia de modelo, por lo que las clases raíces del metamodelo, que se muestran en la figura 2.12, son:

- *Descriptor*: Clase raíz de cualquier descriptor de modelo. Un *Descriptor* es un elemento abstracto de modelado que representa una plantilla genérica y parametrizable, que define la información que se necesita para describir el modelo de tiempo real de algún tipo de recurso, módulo o servicio del sistema independientemente de donde sea utilizado. El descriptor proporciona la información semántica y cuantitativa que es común a todos los entes que puedan resultar de su instanciación.
- *Instance*: Clase raíz de cualquier instancia de modelo. Un elemento de tipo *Instance* representa el modelo de tiempo real concreto y final de una instancia individual de un recurso, módulo o servicio del sistema dentro de una situación de tiempo real. Se obtiene haciendo referencia al descriptor al que corresponde y asignando valores a sus parámetros.

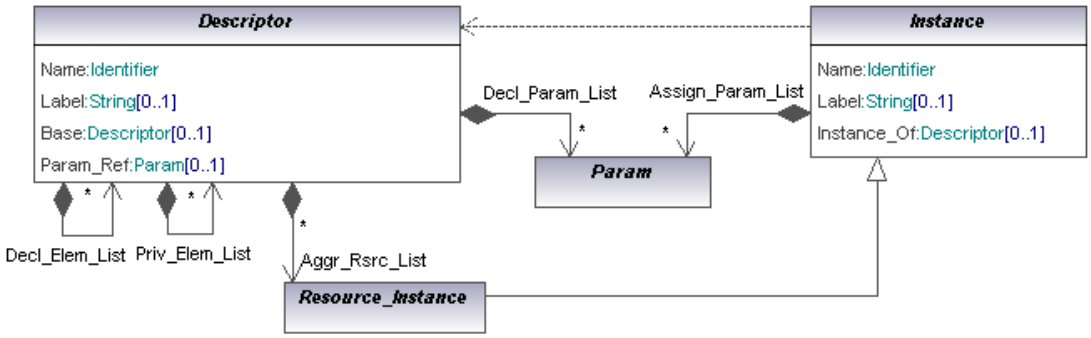


Figura 2.12: Clases raíces del metamodelo Mod-MAST

1. Por razones prácticas, el metamodelo se describe solo en función de los elementos descriptores, teniendo en cuenta en todo momento que cada uno posee su correspondiente elemento instancia. Sólo en aquellos casos en los que se quiera resaltar algún aspecto especial de la declaración de una instancia, se hará explícita en el metamodelo.

Como se observa en la figura, un descriptor de modelo se define a través de las siguientes propiedades:

- Un atributo *Name*, que identifica unívocamente al descriptor, y a través del cual puede ser referenciado por las correspondientes instancias de modelo o por otros descriptores.
- Un atributo *Base*, a través del que se pueden implementar jerarquías de descriptores. Podemos definir un descriptor en base a otro desarrollado previamente, al que se añaden nuevos elementos, o se modifica el valor por defecto de alguno de sus parámetros.
- Un atributo *Param_Ref*, cuya utilidad se explica en la sección 2.3.3.
- La asociación *Decl_Param_List*, en la que se declaran los parámetros del descriptor, a los que habrá que asignar valores concretos cuando se declare una instancia de modelo correspondiente al descriptor.
- La asociación *Aggr_Rsrc_List*, que sirve para declarar la lista de instancias de recursos que se incluirán en el modelo de tiempo real final por cada instancia del descriptor declarada. Por ejemplo, cuando un módulo activo instancie un thread en el sistema, su correspondiente descriptor deberá incluir el modelo de dicho thread como recurso agregado, a través de un elemento de tipo *Scheduling_Server_Instance* añadido a la asociación *Aggr_Rsrc_List*. Cuando posteriormente se genere el modelo de una aplicación en la que se emplee un módulo de ese tipo, al procesar su correspondiente instancia de modelo, se incluirá automáticamente una instancia del modelo del thread. Los elementos agregados pueden estar parametrizados, pero siempre con sus parámetros declarados como parámetros del descriptor raíz al que pertenecen, de modo que queden perfectamente definidos en base a los valores asignados en la instancia de modelo contenedora. Si en el ejemplo anterior, la prioridad del thread es parametrizable, deberá incluirse como parámetro del descriptor del módulo.
- La asociación *Priv_Elem_List,* a través de la que se declara la lista de descriptores de elementos privados que sólo pueden ser referenciados desde otros elementos internos del descriptor. Son utilizados para modularizar y hacer más sencilla su estructura, al permitir la declaración de elementos de modelado complejos (como puede ser una transacción) en función de otros elementos más simples. Estos elementos pueden estar parametrizados, pero sus parámetros deberán estar declarados siempre como parámetros del descriptor raíz al que pertenecen.
- La asociación *Decl_Elem_List*, que representa la lista de descriptores de elementos públicos que pueden ser referenciados desde fuera del descriptor. A diferencia de los elementos agregados, éstos solo son incorporados al modelo final de una aplicación cuando algún elemento externo hace referencia a ellos. Los métodos ofrecidos por un módulo software son un ejemplo claro de elementos declarados, ya que sólo son incorporados al modelo de una aplicación si son invocados desde el modelo de alguna de las transacciones que se ejecutan en ella. Un mismo elemento declarado puede ser incorporado a un modelo varias veces, como ocurre en el caso de un método ofrecido por un módulo.

  Los elementos declarados pueden estar también parametrizados. Sus parámetros pueden estar declarados como parámetros del descriptor al que pertenecen, en cuyo caso se resolverían en la propia instancia del descriptor. Sin embargo, también pueden declarar sus propios parámetros, que deberán ser asignados por aquellos elementos que los incorporen al modelo. Un ejemplo de elemento que puede aparecer en el apartado de elementos declarados de un módulo con sus propios parámetros lo constituyen las transacciones que gestiona un módulo activo. Éstas sólo serán incorporadas al modelo

final cuando formen parte de la carga de trabajo de la aplicación, esto es, cuando la transacción sea realmente ejecutada en el sistema. La transacción puede declarar sus propios parámetros, como por ejemplo el patrón de lanzamiento o el plazo con el que se requiere su ejecución. Estos parámetros no dependen de la configuración del módulo, sino que son exclusivos de la transacción, y por tanto, son asignados cuando se declara una instancia de transacción en el contexto de una carga de trabajo.

Una instancia de modelo se define de forma muy sencilla, únicamente contiene:

- Un atributo *Name*, que identifica unívocamente la instancia.
- Un atributo *Instance_Of*, que referencia el descriptor que se toma como base para la declaración de la instancia.
- La lista de valores asignados a los parámetros definidos en el descriptor correspondiente, asignados a través de la asociación *Assign_Param_List*. Cuando se declare una instancia de modelo, todos los parámetros declarados en el correspondiente descriptor deberán tener un valor asignado, bien a través de este campo, o bien a través de un valor por defecto asignado en la propia definición del parámetro. Los valores asignados a través de *Assign_Param_List* sobrescriben los posibles valores por defecto.

La clase *Param*, utilizada tanto para declarar como para dar valor a los parámetros, se declara como abstracta a este nivel y únicamente posee el atributo *Name*, a través del que se identifica el parámetro. Los diferentes tipos de parámetros que se pueden declarar se definen como clases derivadas de ella. En función del tipo de valor que pueden recibir, los parámetros se dividen en dos grandes grupos:

- Parámetros que reciben valores escalares (numéricos o enumerados). Son derivados de la clase *Param* que añaden un nuevo campo *Value* a través del que se les puede asignar el valor correspondiente. Por ejemplo, si un descriptor quiere declarar un parámetro de tipo *Factor* (*float*), utilizará la clase *Factor_Param*, que como se observa en la figura 2.13, admite valores únicamente de tipo Factor.
- Parámetros a los que se puede asignar una referencia a otras instancias declaradas en el modelo. Por cada tipo de elemento que se quiere referenciar se define una clase concreta cuyo campo *Value* referencia a un elemento del tipo correspondiente. En la figura se muestra como ejemplo el tipo de parámetro *Processing_Resource_Ref*, que permite referenciar elementos de tipo *Processing_Resource*.

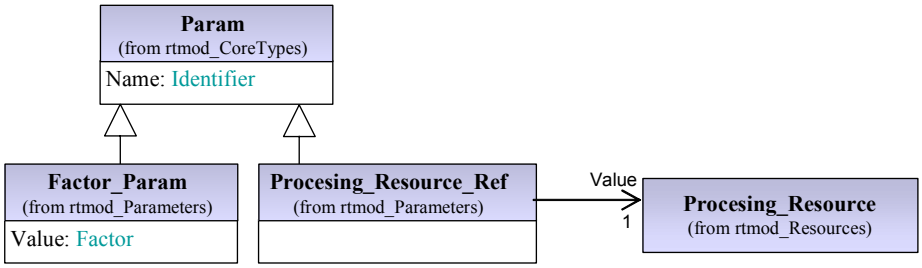


Figura 2.13: Definición de parámetros en Mod-MAST

Las dos clases raíces principales del metamodelo Mod-MAST (*Descriptor* e *Instance*) se especializan, como se muestra en la figura 2.14, en cuatro clases abstractas principales, de las que se van a derivar el conjunto de elementos que se ofrecen para el modelado del comportamiento de una aplicación completa:

- La clase *Container* sirve de raíz para todos aquellos elementos que actúan como contenedores en el modelo. Estos elementos describen el comportamiento de módulos software o hardware completos, identificables externamente en la arquitectura de la aplicación, como pueden ser un nodo de procesamiento, o un módulo software.
- Los elementos de tipo *Usage* sirven para describir formas de uso de los recursos de la plataforma, esto es, la cantidad de procesamiento que corresponde a una actividad que se ejecuta en una aplicación, o la necesidad de sincronización a través de un recurso compartido. Se pueden utilizar para modelar la ejecución de servicios invocados en un módulo, las actividades ejecutadas como consecuencia de procesos internos en las plataformas de ejecución, transacciones, etc.
- Los elementos de tipo *Resource* sirven para describir el comportamiento de los recursos de bajo nivel que contribuyen a la capacidad de procesamiento disponible para ejecutar la aplicación, bien por que la proporcionan, la consumen, o condicionan su uso. Ejemplo de recursos que pueden ser modelados con este tipo de elementos son los procesadores, recursos compartidos, planificadores, etc.
- La clase *Data_Struct* sirve para describir estructuras de datos parametrizadas, que pueden ser útiles en la definición de alguno de los elementos anteriores, como pueden ser parámetros de planificación o sincronización, requisitos temporales, etc.

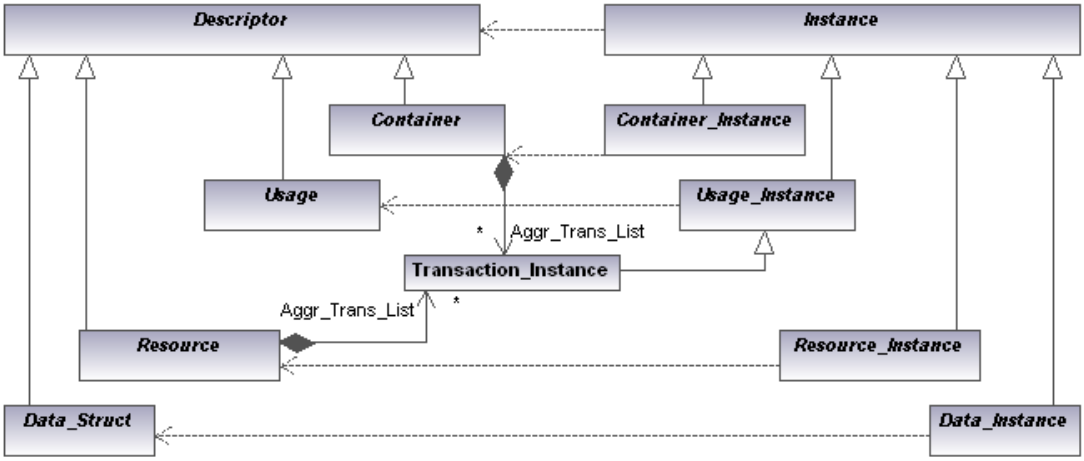


Figura 2.14: Clases principales del metamodelo Mod-MAST

Como se observa en la figura 2.14, tanto los descriptores de contenedores como los de recursos poseen una asociación adicional, *Aggr_Trans_List*, constituida por una lista de instancias de transacción. Esta lista agrupa el conjunto de transacciones cuyos modelos deben ser instanciados por cada instancia del elemento que se añada a un modelo, y representan las transacciones que el recurso añade a la carga de trabajo como consecuencia de su propia actividad interna. Pueden corresponder por ejemplo a transacciones intrínsecas a la naturaleza de un módulo; o a aquellas transacciones que se ejecutan en un procesador como consecuencia de procesos internos del sistema operativo o de los servicios de comunicaciones.

Cada descriptor (e instancia) derivado de estas clases principales tiene una semántica propia, acorde al elemento que representa. Ésta semántica va a restringir tanto el tipo de parámetros que el descriptor puede definir, como el tipo de elementos que puede agregar y/o declarar. Cuando sea de especial interés a lo largo de la explicación del metamodelo se harán explícitas dichas restricciones.

2.3.1.2. Paquete rtmod_Module

El paquete *rtmod_Module* contiene las definiciones de las primitivas de modelado de bajo nivel que se requieren para describir internamente el comportamiento temporal de módulos reutilizables hardware y software.

El paquete está dividido en diferentes subpaquetes, que agrupan los diferentes elementos de modelado de acuerdo a su naturaleza. La mayor parte de los elementos de modelado definidos en estos subpaquetes se corresponden directamente con las primitivas de modelado definidas en el modelo MAST. Mod-MAST añade la capacidad de parametrización a dichas primitivas, por lo que aunque los atributos o asociaciones de un determinado elemento en MAST y en Mod_MAST sean las mismas, en el caso de un descriptor Mod-MAST se declaran de forma que puedan recibir valor tanto de manera directa como a través de parámetros. Para poder declarar atributos y asociaciones parametrizadas se utilizan dos vías:

- En el caso de atributos de tipo escalar, se utilizarán los tipos definidos en el subpaquete *rtmod_ParameterizedTypes*. En él se redefinen todos los tipos escalares para que puedan ser utilizados como parámetros, esto es, para que admitan tanto valores asignados de manera directa, como valores asignados a través de parámetros. Como ejemplo, si un elemento MAST posee un atributo de tipo *Factor* (float), en Mod-MAST dicho atributo pasa a ser de tipo *P_Factor*, donde *P_Factor* se define como se muestra en la figura 2.15. El atributo puede recibir valor tanto de forma directa, a través del campo *Basic_Data*, como a través de un parámetro, por medio del campo *Ref_Value*.

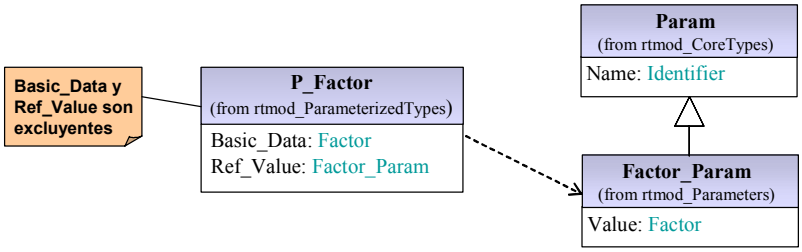


Figura 2.15: Parametrización de atributos en Mod-MAST

- En el caso de asociaciones con otros elementos de modelado, la parametrización se implementa a través del atributo *Param_Ref*. Este atributo se hereda de *Descriptor*, pero en cada descriptor concreto se restringe su tipo de acuerdo a la naturaleza del elemento modelado. Para explicar su funcionamiento utilizamos el ejemplo de la figura 2.16. En un modelo MAST, todo planificador (*Scheduler*) posee una referencia (*host*) al procesador al que planifica, que es un elemento de tipo *Processing_Resource*. En Mod-MAST, la referencia host de un *Scheduler* es parametrizada. Para ello, la asociación sigue siendo con un elemento de tipo *Processing_Resource*, pero en la definición de éste se incluye un atributo *Param_Ref* de tipo *Processing_Resource_Ref*, que corresponde a un parámetro al que se le pueden asignar valores de tipo *Processing_Resource.* Si se asigna valor al atributo *Param_Ref*, no debe asignarse valor al resto de atributos del *Processing_Resource*, pues sus valores serán los del elemento que se asigne al parámetro cuando éste se resuelva en cada instancia del descriptor.

En el resto de esta sección se introducen los elementos definidos en el paquete *rtmod_Module*, sin profundizar en aquellos que sean mapeados directos de primitivas MAST. Su definición detallada se puede consultar en los diagramas mostrados en el anexo A. Aquellos elementos que sean propios del metamodelo Mod-MAST sí se explican en detalle.

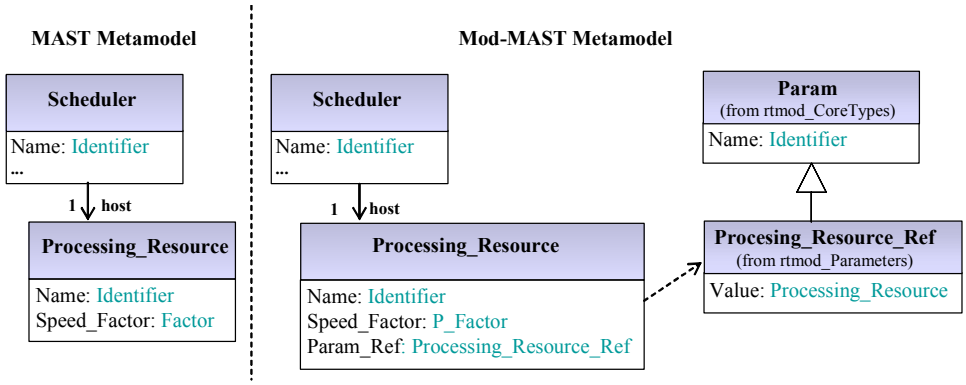


Figura 2.16: Parametrización de asociaciones en Mod-MAST

Paquete rtmod_Resources

El paquete *rtmod_Resources* define los elementos de modelado de tiempo real que tienen como elemento raíz a la clase *Resource*, y que se utilizan para describir la capacidad de procesamiento disponible para la ejecución de una aplicación. Se definen elementos que permiten modelar:

- Los recursos de la plataforma, que proporcionan capacidad bien para ejecutar actividades en el sistema, como un procesador, o bien para enviar mensajes, como una red de comunicaciones.
- El consumo de dicha capacidad que se produce a consecuencia de la ejecución de actividades internas de los recursos, como puede ser la gestión del tiempo, o de los protocolos de comunicación.
- Las limitaciones al uso de la capacidad que introducen los elementos de concurrencia y sincronización que se necesitan para planificar el acceso de las actividades de las aplicaciones a los recursos de acceso restringido.

La figura 2.17 muestra el conjunto de elementos de modelado definidos. Como todos ellos se derivan del metamodelo MAST [MASTd], no se detallan sus propiedades o su estructura, sino que nos limitamos a dar una breve explicación del elemento que cada uno de ellos modela:

- *Processing_Resource*: Es un elemento de modelado abstracto que modela la capacidad de un elemento hardware para llevar a cabo actividades de una aplicación, tales como ejecución de segmentos de código o transferencia de mensajes. De él se derivan dos elementos, también abstractos:
  - *Processor*: Modela la capacidad de un dispositivo para ejecutar líneas de código. Actualmente se ha definido un único elemento de modelado concreto derivado de él, *Regular_Processor*. Un *Regular_Processor* modela un procesador que proporciona capacidad de procesamiento de forma uniforme y continua, incluyendo como parte de su modelo la sobrecarga debida a la gestión del reloj del sistema.
  - *Network*: Modela la capacidad de un dispositivo para transferir mensajes entre procesadores. Derivado de él se define el elemento concreto *PB_Network*, que modela una red de comunicaciones que permite el envío de mensajes fragmentados en paquetes no expulsables.
- *System_Timer*: Modela el consumo de capacidad que conlleva en un procesador la gestión del temporizador, así como la granularidad del tiempo en las actividades temporizadas. Actualmente existen dos modos de modelar esta carga, que se traducen en dos elementos de modelado concretos derivados de la clase abstracta *System_Timer*:

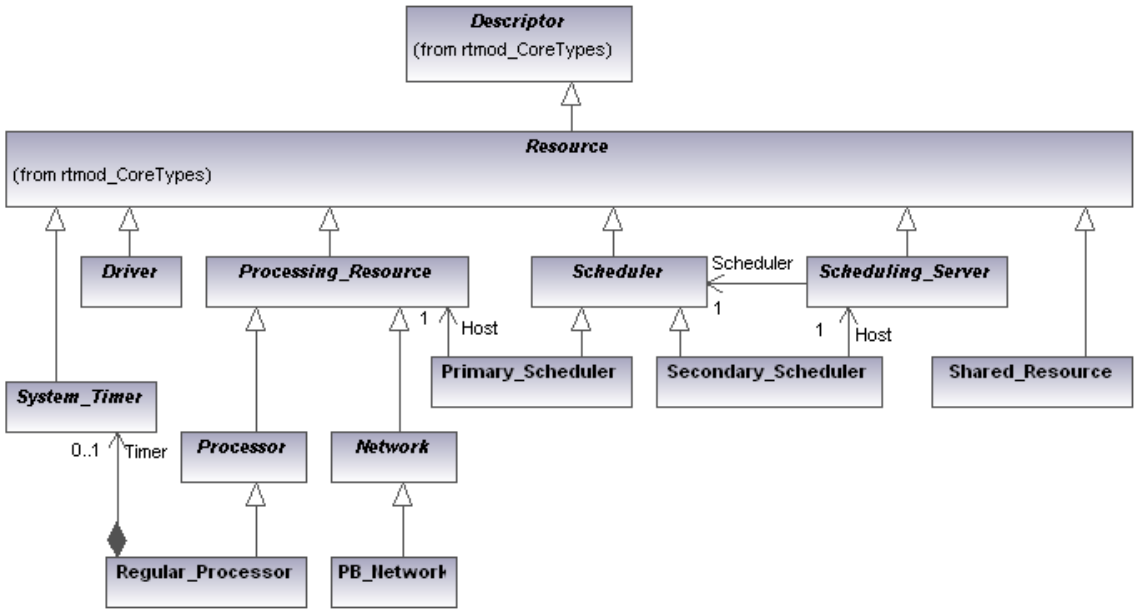


Figura 2.17: Elementos principales del paquete rtmod_Resources

- *Ticker_Timer*: Modela un temporizador que se activa periódicamente mediante una interrupción hardware, introduciendo una sobrecarga al sistema en cada interrupción.
- *Alarm_Clock_Timer*: Modela un temporizador que se activa únicamente cuando se alcanza algún plazo programado en el sistema.

- *Driver*: Modela el consumo de capacidad de un procesador que se emplea en la gestión del envío y recepción de mensajes por una red de comunicación utilizando un protocolo de comunicaciones concreto. Por cada conexión que un procesador realice con una red, y por cada protocolo diferente que se soporte, será necesario incluir un elemento de este tipo, que introduzca en el modelo las sobrecargas correspondientes. Se define como una clase abstracta, de la que se derivarán elementos concretos para modelar los diferentes tipos de protocolos o redes soportadas. Hasta el momento se han definido los siguientes tipos:
  - *Packet_Driver:* Modela aquellos drivers que introducen una carga en el sistema por cada paquete que se envía a través de la red.
  - *Character_Packet_Driver:* Es una especialización del anterior, que añade el modelo del gasto de capacidad que se realiza en el envío de cada carácter de un paquete.
  - *RTEP_Packet_Driver:* Elemento derivado también de *Packet_Driver*, que le añade la sobrecarga producida en el sistema como consecuencia de dar soporte al protocolo de tiempo real RT-EP (*Real-Time Ethernet Protocol*) [MG05]. El envío de un mensaje a través de este protocolo tiene dos fases. La primera fase es la fase de arbitrio, en la que a través del paso de un testigo por todas las estaciones que forman el anillo, se decide cual es la estación con el mensaje de mayor prioridad. En la siguiente fase se produce el envío del mensaje. Las sobrecargas introducidas en el sistema para soportar los pasos del testigo, así como la recuperación de errores durante su transmisión, son incluidas en el modelo de este driver.
- *Scheduler:* Modela la actividad de un planificador. Un planificador es un elemento del sistema operativo que gestiona la distribución de la capacidad de procesamiento

disponible entre las tareas en espera de ser ejecutadas. Un *Scheduler* modela tanto la política de planificación con que se distribuye la capacidad, como las sobrecargas introducidas en el sistema a consecuencia de su gestión, esto es, la capacidad empleada en llevar a cabo los cambios de contexto entre tareas. La metodología permite modelar políticas de planificación basadas en prioridades fijas y EDF. También existe la posibilidad de definir planificadores que modelen protocolos de transferencia de mensajes basados en prioridades, con los que se pueden modelar redes que implementan envíos de mensajes priorizados, como CAN, o Ethernet con RT-EP, de especial interés en sistemas de tiempo real.

El metamodelo prevé una estructura jerárquica de planificadores:

- Los planificadores primarios, *Primary_Scheduler*, gestionan directamente la capacidad proporcionada por un recurso de procesamiento hardware, por tanto, van asociados siempre a un elemento de tipo *Processing_Resource*.
- Los planificadores secundarios, *Secondary_Scheduler*, gestionan en segundo nivel la capacidad de procesamiento que ha sido asignada a un sólo proceso o thread del sistema. Su asociación se realiza en este caso con un elemento de tipo *Scheduling_Server.*

- *Scheduling_Server*: Modela entidades de planificación tales como procesos y threads de un procesador o sesiones de comunicación de una red, en los que se van a ejecutar secuencialmente conjuntos de actividades del sistema relacionadas entre sí por flujo de control. Cada uno tiene agregada una estructura de datos que es diferente en función de la política de planificación con la que está siendo planificado, y que es procesada por el correspondiente planificador, para en función de ella otorgarle o no la capacidad de ejecución.
- *Shared_Resource:* Modela la sincronización entre las diferentes líneas de flujo de control que se produce cuando se ha de acceder en régimen de exclusión mutua a recursos protegidos. Sólo se admiten protocolos que impiden inversión de prioridad, tales como el techo inmediato de prioridad, herencia de prioridad o protocolo basado en pila (SRP) [BAK91].

Paquete rtmod_Usages

El paquete *rtmod_Usages* contiene los elementos de modelado que describen la capacidad de procesamiento que requiere la ejecución de las actividades de las aplicaciones, o lo que es lo mismo, el gasto de la capacidad ofrecida por los recursos de la plataforma que produce la ejecución de la aplicación. Todos los elementos definidos en este paquete tienen como elemento raíz la clase abstracta *Usage,* que representa una forma de uso abstracta de un recurso. Modelan tanto la capacidad de procesamiento que se consume en la ejecución de una actividad, como el acceso a recursos que se requiere en ella y que puede generar retrasos en la ejecución. Permiten modelar, por ejemplo, la ejecución del código de un procedimiento lógico de un módulo, las actividades que los recursos realizan internamente (como la gestión del reloj del sistema), o una invocación remota ejecutada a través de un determinado software de intermediación.

En la figura 2.18 se muestran las clases definidas en el metamodelo para modelar el uso de recursos, que se pueden dividir en dos grandes bloques:

- Los elementos de modelado derivados de *Requested_Usage* se emplean para modelar aquellas formas de uso ofrecidas por los diferentes elementos que forman la aplicación. En este grupo se incluyen tanto formas de uso que pueden ser explícitamente invocadas durante la ejecución de la aplicación, como son los métodos ofrecidos por un módulo

software, como aquellas que se añaden al modelo de forma implícita, como la operación que modela un cambio de contexto entre tareas. A su vez, se dividen en dos grandes grupos, según modelen la invocación remota, *Remote_Usage*, o local, *Local_Usage*, de una forma de uso.

- Los elementos de tipo *Transaction* modelan el uso de recursos que se genera en un sistema como consecuencia de la respuesta a eventos externos o temporizados. Como veremos más adelante, este concepto se emplea para definir la carga de trabajo que soporta una aplicación en un determinado modo de ejecución.

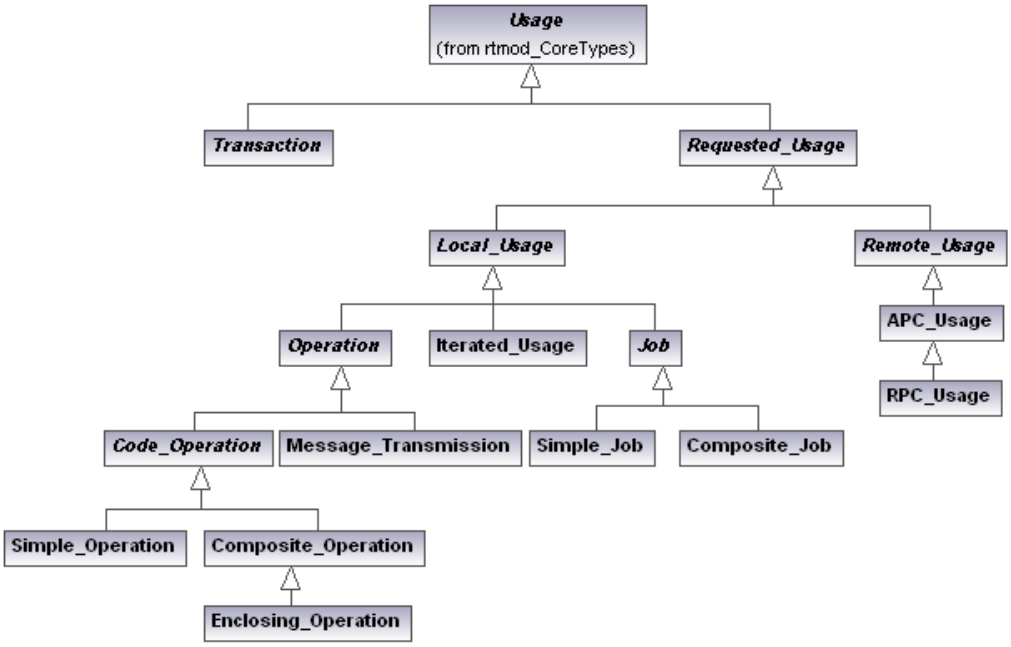


Figura 2.18: Elementos de modelado del paquete rtmod_Usages

Los elementos de modelado que extienden a *Local_Usage*, y que por tanto, incluyen la información necesaria para modelar la ejecución local de una forma de uso ofrecida por algún elemento de la aplicación son:

- *Operation*: Modela la ejecución de una sección de código o la transferencia de un mensaje. Específicamente describe la cantidad de procesamiento que requiere la ejecución de una actividad que se planifica en un sólo proceso (en el mismo servidor de planificación), pudiendo requerir sincronización con otros procesos sólo en los instantes inicial y final. A tal fin, declara los recursos compartidos a los que debe acceder en su inicio para poder realizarse y los recursos que libera tras su conclusión. En el metamodelo se definen diferentes operaciones especializadas:
  - *Code_Operation*: Modela la ejecución de una sección de código en una única unidad de concurrencia de un procesador, y con posible acceso a recursos en régimen de exclusión mutua. Se especializa en:
    - *Simple_Operation*: Modela una sección simple de código.
    - *Composite_Operation*: Modela una operación compleja compuesta por una secuencia ordenada de operaciones. El gasto de capacidad, o lo que es lo mismo, el tiempo de ejecución, corresponde a la suma de los tiempos de las operaciones implicadas.

  - *Enclosing_Operation*: Modela también una operación compuesta por una secuencia de operaciones, pero en este caso la secuencia no es ordenada. Su tiempo total de ejecución, por tanto, no corresponde a la suma de los tiempos de ejecución de cada operación, ya que se pueden ejecutar operaciones adicionales intermedias.
- *Message_Transmission*: Modela la transferencia de un mensaje simple a través de una red.

- *Iterated_Usage*: Este elemento se utiliza para modelar la ejecución repetida de otra forma de uso de cualquier tipo heredado de *Requested_Usage*, un determinado número de veces. Este elemento de modelado es introducido en Mod-MAST para favorecer la modularidad (no existe en MAST). Tanto la forma de uso a ejecutar como el número de veces que se repite su ejecución son parametrizables.
- *Job*: Modela el uso de recursos que se realiza cuando se invoca un procedimiento que ejecuta un conjunto de actividades relacionadas entre sí por flujo de control, pero que pueden ejecutarse en múltiples servidores de planificación, y por tanto, no pueden ser modeladas con un elemento de tipo *Operation*.

  Este elemento de modelado tampoco existe en el metamodelo MAST, aunque sí fue introducido en el perfil MAST para sistemas orientados a objetos [MED05][MGD01]. Este elemento resulta esencial en el modelado de aplicaciones modulares o basadas en componentes. Un caso típico en que se requiere su empleo es el modelado de un método ofrecido por un módulo, cuando su implementación realiza invocaciones en métodos de módulos externos. En este caso además, es de vital importancia que el *Job* pueda ser parametrizado, pues los métodos invocados en módulos externos han de incluirse siempre como referencias, y por tanto, definirse como parámetros en el descriptor. Sólo podrán ser resueltos una vez conocida la estructura de una aplicación, y por tanto, los modelos de los módulos concretos conectados sobre los que se realiza la invocación final. Se definen dos tipos de *Job*:
  - *Simple_Job*: Modela un *Job* simple, cuya estructura se puede definir de forma completa en el descriptor de un módulo. La definición de un *Job* requiere una estructura compleja. Hay que definir las diferentes actividades que conlleva, las relaciones de flujo entre ellas, las formas de uso concretas que se ejecutan en cada actividad, así como las unidades de concurrencia en las que son ejecutadas. El metamodelo que describe a un *Job* se muestra en la figura 2.19. Debido a que los

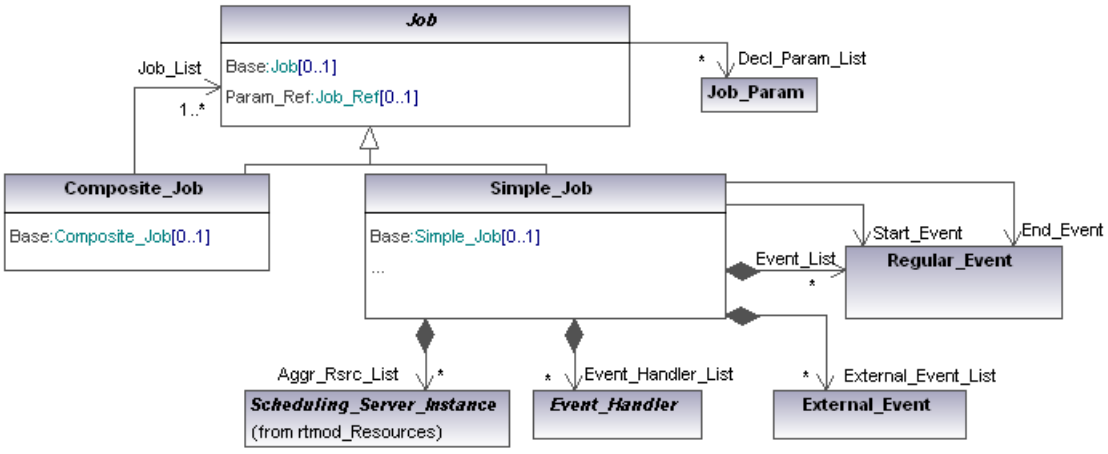


Figura 2.19: Estructura de un elemento de modelado de tipo *Job*

elementos que se emplean en su definición son los mismos que en el caso de una transacción, se traslada su descripción detallada al punto en el que se introduce dicho elemento.

- *Composite_Job*: Modela un *Job* complejo compuesto por una secuencia ordenada de otros *Job*. Este elemento se introduce como mecanismo para modelar formas de uso cuya estructura puede variar en función de la configuración de la aplicación.

Los elementos de modelado que extienden a la clase abstracta *Remote_Usage* modelan formas de uso de un recurso que se pueden invocar de forma remota. Para ello, y como se observa en la figura 2.20, complementan el modelo local de la forma de uso, referenciada a través de la propiedad *usage*, con todas aquellas características necesarias para modelar la invocación remota que son propias del uso del recurso, y no del mecanismo con el que se realice la invocación. Entre los nuevos datos que incorpora se encuentran las operaciones de serialización (*marshalling*) y deserialización (*unmarshalling*) de los parámetros de la invocación, así como las características de los mensajes de invocación y de retorno de resultado.

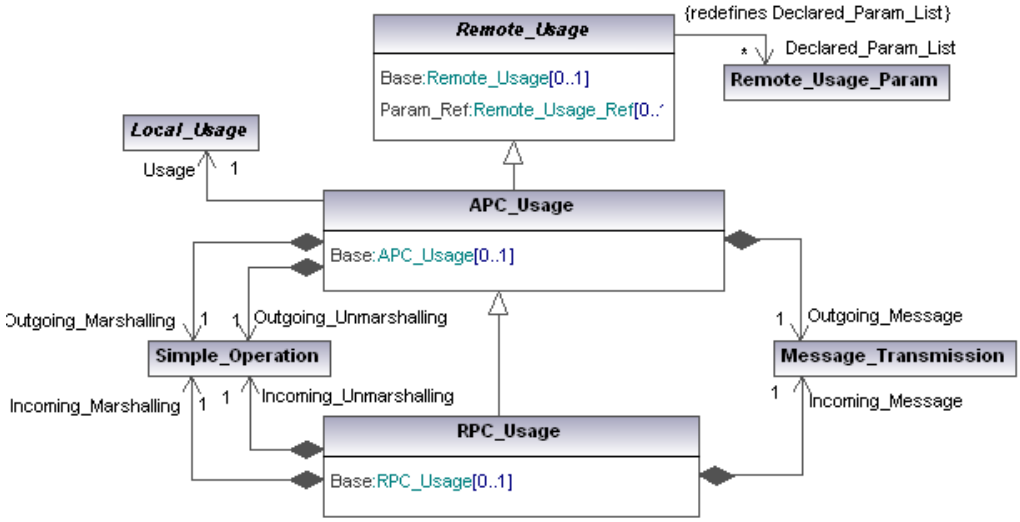


Figura 2.20: Elementos de modelado derivados de la clase *Remote_Usage*

Estas operaciones serán referenciadas desde los modelos de los mecanismos de comunicación, ya que aunque la secuencia de actividades que se ejecutan durante una invocación remota depende del mecanismo escogido, y por lo tanto se encuentra descrita en su correspondiente descriptor, algunas de dichas actividades dependen del módulo concreto sobre el que se realiza la invocación, como es el caso de las operaciones de serialización y deserialización. Es por esto que este tipo de elemento de modelado es característico del metamodelo Mod-MAST, y no aparece en el modelo MAST, pues su objetivo es dar soporte a la modularización de los modelos, permitiendo separar aquellos aspectos que dependen del mecanismo de comunicación de aquellos que dependen del código de los módulos.

Modelo de transacciones

Los elementos del paquete *rtmod_Usages* vistos hasta el momento permiten modelar las formas de uso ofrecidas por los recursos que intervienen en la aplicación, bien sean elementos de la plataforma o los propios módulos que forman la lógica de la aplicación. Junto con los elementos definidos en el paquete *rtmod_Resources*, definen el modelo estructural y lógico de la aplicación.

Como se explicó en la sección 2.2, en el caso de una aplicación de tiempo real, este modelado estructural debe ir acompañado del modelo reactivo de la aplicación. Utilizando metodologías basadas en modelado transaccional, dicho modelo reactivo se describe en base a las transacciones de tiempo real que se ejecutan concurrentemente en el sistema. Cada transacción describe la secuencia de actividades que se ejecutan como respuesta a un evento, y constituye el elemento a través del que se relacionan todos los recursos y formas de uso modelados a través de los elementos vistos hasta el momento. Una transacción representa también una forma de uso, y como tal, se define en este paquete el elemento que permite modelarla, que denominamos *Transaction* (este elemento sí se encuentra en el modelo MAST).

Cada transacción se modela mediante un grafo en el que los nudos son actividades o manejadores de eventos, y los arcos son eventos entre ellos. En conjunto representan las relaciones de flujo que existen entre las actividades que constituyen la transacción. Como muestra la figura 2.21, una transacción se define por medio de tres tipos de elementos:

- *External_Event*: Modela los eventos que activan la transacción, ya sean procedentes del entorno o eventos temporizados generados por el reloj. Incluyen, a través de un elemento de tipo *Arrival_Pattern*, la descripción del tipo de patrón de generación escogido, esto es, la distribución de los tiempos de activación de la transacción, la cual define la carga de trabajo que soporta la aplicación. Por ello, este elemento se describe en el paquete *rtmod_Workload*, que se explica a continuación.
- *Regular_Event:* Modela una dependencia de flujo entre actividades dentro de la transacción. La transacción sirve como punto de referencia para la introducción de las restricciones temporales que se deben satisfacer en la ejecución de la aplicación. Con ese objetivo, un evento puede tener asociados varios elementos *Timing_Req_Observer*, que representan monitores a través de los que se puede evaluar el cumplimiento de los requisitos temporales que se deben satisfacer en la generación del evento. Estos monitores llevan asociados los requisitos temporales que se han de evaluar, modelados a través de elementos de tipo *Timing_Requirement*. Estos requisitos, junto a los patrones de generación de eventos externos, definen la carga de trabajo de la aplicación, por lo que también son introducidos en el paquete *rtmod_Workload*.

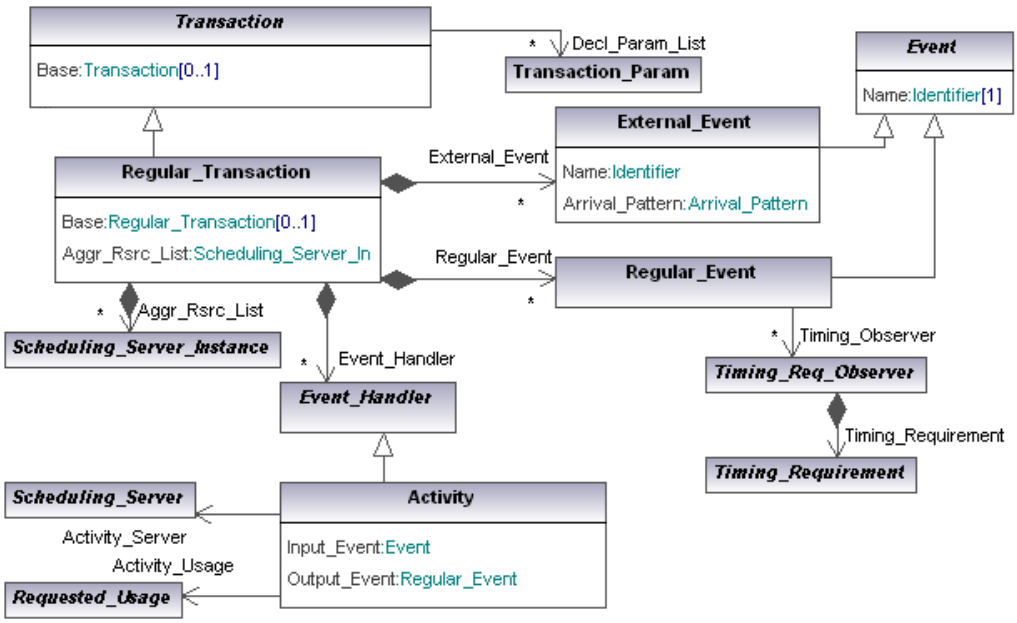


Figura 2.21: Elementos de modelado de una transacción

Tanto los elementos *External_Event* como los *Regular_Event* representan eventos que forman parte del flujo de control de la transacción, por lo que ambos extienden a la clase abstracta *Event*.

- *Event_Handler*: Modelan las acciones o actividades que constituyen la transacción y que se relacionan entre sí a través de los eventos. Cuando se produce un evento, la acción o actividad que modela el *Event_Handler* que lo tiene por entrada se activa. Cuando la acción o actividad finaliza, se generan nuevos eventos que están declarados como eventos de salida del *Event_Handler*. Hay dos tipos de manejadores de eventos: las actividades, *Activity*, representan la ejecución de una operación por un servidor de planificación en un recurso de la plataforma y únicamente poseen un evento de entrada y otro de salida; los otros manejadores representan diferentes opciones de transferencia de flujo de control entre actividades, tales como retrasos (*delay*), invocación concurrente (*fork*), invocación alternativa (*branch*), convergencia sincronizada (*join*) o convergencia alternativa (*merge*).

El proceso de instanciación del modelo de una transacción difiere del resto de elementos. En otros casos, el proceso de instanciación consiste en agregar al modelo final del sistema, los modelos de los recursos que se definen como agregados en el correspondiente descriptor, todos ellos con los correspondientes valores asignados a sus parámetros. En el caso de una transacción, la instanciación de su modelo implica la de todos los elementos de tipo *External_Event*, *Regular_Event* y *Event_Handler* definidos en su descriptor. Estos elementos no modelan ningún recurso o forma de uso de la aplicación. Simplemente se introducen como un medio para definir el flujo de control de la transacción, y de ese modo, relacionar todos los elementos de modelado incluidos en el modelo de la aplicación. En concreto, es el concepto de *Activity* el que relaciona todos los elementos de modelado implicados en la ejecución de una operación. Como se observa en la figura, cada actividad consiste en la ejecución de una forma de uso, referenciada a través de la propiedad *activityUsage*. Dicha forma de uso deberá corresponder a alguna de las ofrecidas por los elementos que forman parte del modelo de la aplicación. A través de dicha referencia se conoce la capacidad de procesamiento requerida, así como el posible acceso a recursos compartidos. Las características de planificación de la ejecución (principalmente, su prioridad), así como la identificación del recurso del que se va a tomar la capacidad, se conocen a través del ente que lo planifica, que se referencia a través de la propiedad *activityServer*.

La estructura requerida para describir un *Job*, como se mostraba en la figura 2.19, es muy similar a la de una transacción. Se modela también a través del conjunto de eventos y manejadores de eventos que definen el flujo de control interno del *Job*. Sin embargo, la principal diferencia deriva del hecho de que un *Job* modela una forma de uso ofrecida por un recurso del sistema, que será referenciada desde una actividad dentro de una transacción. Por ello, todo *Job* posee dos eventos especiales: el evento de inicio de ejecución (*start_Event*), a través del que se referencia al evento de entrada de la actividad desde la que se invoca el *Job*, y el de fin de ejecución (*end_Event*), que coincide con el evento de salida de la actividad invocante. Por el contrario, un elemento de tipo *Transaction* modela la respuesta del sistema a un evento externo o temporizado, por lo que su evento de entrada se modela a través de uno o varios elementos de tipo *External_Event*, a través de los que se define el patrón de generación de dichos eventos. En el modelo de una transacción no se identifica de manera especial el evento de finalización, puede ser deducido del propio flujo de la transacción.

Paquete rtmod_Workload

El paquete *rtmod_Workload* agrupa los elementos de modelado que permiten definir la carga de trabajo de una aplicación. Tomando como base el modelo reactivo de la aplicación, esto es, una vez identificadas las transacciones que se van a ejecutar en un determinado modo de operación de la aplicación, la definición de la carga de trabajo consiste en:

- Fijar el patrón de disparo de todas aquellas transacciones que lo ofrezcan como parametrizable.
- Fijar los requisitos temporales que se han de satisfacer durante la ejecución del sistema, también en aquellas transacciones para las que éstos sean parametrizables.

Como vimos en el apartado anterior, el modelo de una transacción incluye su modelo de disparo y la posibilidad de asignar requisitos temporales a los eventos internos de la transacción. Tanto el patrón de lanzamiento, como las características de dichos requisitos pueden dejarse como parámetros en el descriptor de la transacción, para ser asignados una vez incluida la transacción en un situación de tiempo real concreta.

El paquete *rtmod_Triggering* define los elementos que permiten modelar los patrones de generación de los eventos que lanzan una transacción, los cuales se muestran en la figura 2.22. Como ya vimos en la sección anterior, el evento de lanzamiento de una transacción se modela a través de un elemento de tipo *External_Event.* En él se incluye el nombre del evento, que servirá para hacer referencia a él desde otros elementos de definición de la transacción, y su patrón de generación, modelado a través de un elemento de tipo *Arrival_Pattern*. Estos elementos definen probabilísticamente los tiempos en que se generan los eventos, utilizando los patrones de lanzamiento más comunes contemplados en las técnicas de análisis de planificabilidad, como son patrones periódicos, esporádicos, de ráfaga, etc.

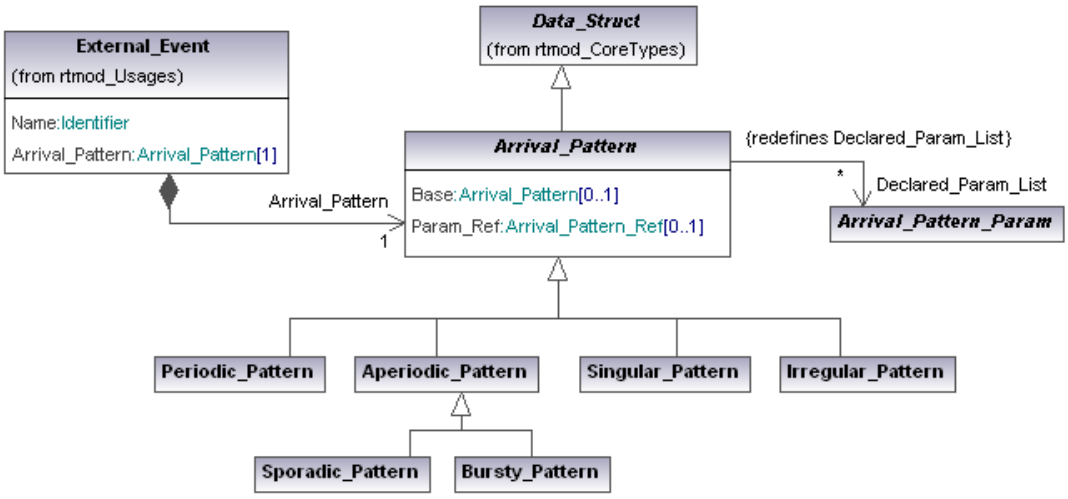


Figura 2.22: Patrones de activación modelados a través del elemento *Arrival_Pattern*

Por otro lado, en el paquete *rtmod_TimingRequirement* se definen los elementos que permiten definir requisitos temporales en el modelo de una transacción. Cuando se asocia un elemento de este tipo a un evento de una transacción, las herramientas de análisis deberán evaluar si el correspondiente requisito se cumple en el momento de la generación del evento. Como se observa en la figura 2.23, se definen dos tipos de elementos abstractos, relacionados entre sí, para el modelado de requisitos temporales. Ya vimos como el elemento *Timing_Observer*

representa un monitor que se asocia a un evento interno de una transacción. Sirve para definir los puntos de la transacción sobre los que se van a poder evaluar requisitos temporales. Se definen dos clases especializadas, *Global_Timing_Req_Observer* y *Local_Timing_Req_Observer*, que distinguen entre requisitos de tipo global (con respecto a un evento de referencia), o requisitos de tipo local (con respecto a la ejecución de la actividad inmediatamente anterior al evento sobre el que se define el requisito). Cada observador, ya sea global o local, lleva asociado un elemento de tipo *Timing_Requirement*, a través del cual se fijan las características del requisito que se quiere verificar: si es de tipo estricto o laxo, si el requisito se aplica sobre el tiempo de respuesta de la transacción (*deadline*), sobre el jitter (*jitter*), etc. Además, se define una última clase que permite agrupar diferentes requisitos en un solo observador, *Composite_Timing_Req.*

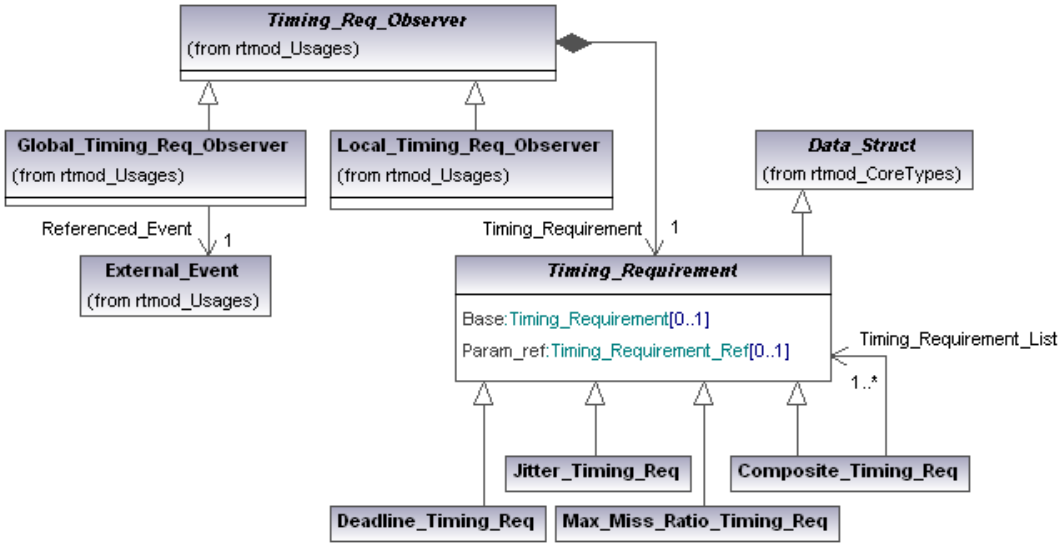


Figura 2.23: Elementos para el modelado de requisitos temporales

La declaración de las restricciones temporales a través dos clases independientes, *Timing_Req_Observer* y *Timing_Requirement,* se ha realizado a fin de distinguir dos aspectos que aun siendo complementarios, se definen por diferentes actores y en diferentes fases del proceso de desarrollo de un sistema de tiempo real. La clase *Timing_Req_Observer* define el estado al que se le asigna la restricción temporal dentro del marco de referencia de la transacción, el cual es establecido por el diseñador del módulo y con independencia del sistema de tiempo real en que se utilice el módulo. La clase *Timing_Requirement* define la naturaleza de la restricción, la cual es definida por el diseñador de la aplicación como parte de la declaración de la carga de trabajo que constituye el contexto de análisis. La definición de las dos clases facilita la reutilización de los modelos de tiempo real.

2.3.1.3. Paquete rtmod_Container

El paquete *rtmod_Container* define los elementos de modelado que sirven de contenedores. Los contenedores organizan y agrupan los elementos de modelado con el objetivo de establecer una correspondencia bien definida entre modelo y sistema, y asimismo, constituyen los módulos de modelado que pueden ser reutilizados en diferentes sistemas.

Un elemento contenedor agrupa los modelos de todos aquellos recursos y formas de uso de los recursos, que constituyen el modelo autocontenido y reutilizable de un módulo software

(librería, componente, etc.) o hardware (nodo de procesamiento, red de comunicación, etc.) independiente. Los elementos contenedores cumplen una doble función:

- La propia función de contenedor, agrupando los diferentes elementos de modelado que constituyen el modelo del módulo, así como sus parámetros.
- Definen un ámbito de visibilidad para los identificadores, parámetros y elementos de modelado.

Los contenedores sirven para organizar los modelos con una estructura similar a la que tiene el sistema desde el punto de vista de su arquitectura software y de su despliegue. Su definición es muy importante para la metodología, ya que constituyen los únicos tipos de descriptores de modelo que van a poder ser elaborados y almacenados de forma independiente, para ser posteriormente utilizados en la construcción de los modelos de sistemas completos.

En la figura 2.24 se muestran los tipos de contenedores que se han definido en la metodología, y que se pueden dividir en dos grandes grupos:

- Aquellos que contienen los modelos de los módulos software que se integran en el sistema, ya sea como parte de la aplicación o de la plataforma. Los contenedores del tipo *Software_Module* describen los modelos de comportamiento temporal de módulos software como elementos completos e independientes.
- Aquellos que contienen los modelos de los recursos que constituyen las plataformas de ejecución. Para describirlas se han definido tres tipos de contenedores:
  - *Processing_Node*: Agrupa todos los elementos de modelado que describen los recursos que forman un nodo procesador de la plataforma.
  - *Communication_Network*: Agrupa todos los elementos de modelado que describen los recursos que forman una red de comunicaciones a través de la que se comunican los nodos procesadores de la plataforma.
  - *Communication_Service*: Agrupa los elementos de modelado que describen los elementos software instalados en la plataforma que se utilizan para realizar interacciones entre módulos instanciados en diferentes nodos procesadores, haciendo uso para ello de la red de comunicaciones.

Es necesario formalizar el modo en que se define y estructura la información incluida en cada uno de estos contenedores, para que las herramientas de composición de modelos puedan extraerla de manera automática y utilizarla para elaborar el modelo transaccional final de la aplicación. En lo que resta de sección se explica la información que debe incluirse en cada tipo de contenedor, las reglas que se siguen para su formalización, así como el modo en que los diferentes tipos de contenedores se relacionan entre sí.

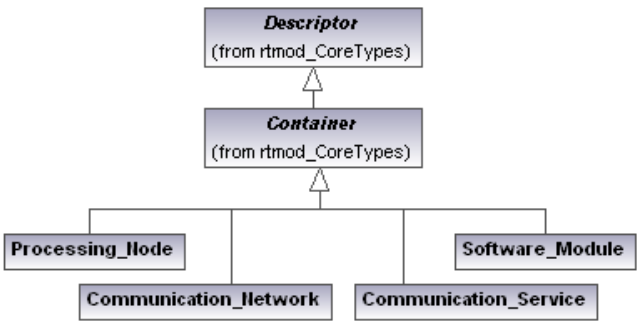


Figura 2.24: Contenedores definidos en el paquete rtmod_Container

Descriptor de un nodo de procesamiento: Processing_Node

Un elemento contenedor de tipo *Processing_Node* agrupa todos los elementos que modelan el comportamiento de un nodo de la plataforma, en el que se instancian y ejecutan módulos o componentes software. Es uno de los elementos que se identifican de forma independiente en la descripción del despliegue de una aplicación.

Como se observa en la figura 2.25, para modelar el comportamiento de un nodo, un elemento *Processing_Node* se formula de acuerdo a las siguientes reglas:

- Contiene como recursos agregados obligatorios:
  - *Processor*: Recurso de tipo *Processor_Instance*, que modela el procesador hardware que permite la ejecución del código. Por el momento se consideran nodos de procesamiento de tipo monoprocesador, por lo que éste elemento es único.
  - *Scheduler*: Recurso de tipo *Primary_Scheduler_Instance*, que modela el planificador que gestiona la capacidad del procesador anterior y la política de planificación que implementa su correspondiente sistema operativo.
  - *Interrupt_Scheduler*: Recurso de tipo *Primary_Scheduler_Instance*, que modela la política de planificación y el consumo de capacidad de procesamiento que se emplea en la gestión de las interrupciones.
- Puede agregar otros recursos, en concreto aquellos que extienden a la clase *Processing_Node_Aggr_Rsrc* (ver anexo A): planificadores secundarios, recursos de planificación, recursos compartidos y drivers.
- Como elementos declarados, tanto públicos como privados, se permiten aquellos que extiendan a la clase *Processing_Node_Elem* (ver anexo A). Un aspecto característico de este tipo de descriptor, es que en caso de que el nodo de procesamiento ofrezca soporte para comunicaciones a través de un determinado tipo de red, o con un determinado protocolo, deberá incluir como elemento declarado, el descriptor del correspondiente driver de comunicación que modela la sobrecarga inducida en el procesador como consecuencia de la gestión del protocolo.
- El tipo de parámetros que puede declarar un elemento de este tipo está restringido a aquellos que extienden al tipo *Processing_Node_Param* (ver anexo A).

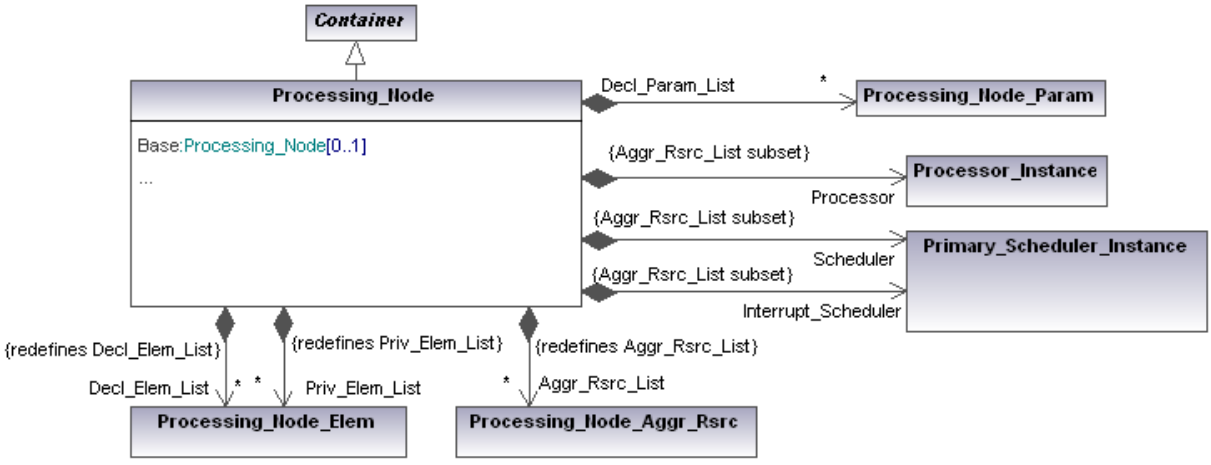


Figura 2.25: Definición de un *Processing_Node*

La declaración de cualquier instancia de un nodo de procesamiento, *Processing_Node_Instance*, requiere:

- Asignar valores concretos a cada uno de los parámetros definidos en su correspondiente descriptor.
- Agregar las instancias de modelo de los correspondientes drivers que se van a emplear en la aplicación, como consecuencia de que el nodo de procesamiento se va a comunicar a través de una o varias redes o protocolos. Por cada conexión que el nodo realice con una red, es necesario incluir la correspondiente instancia de driver que modela la sobrecarga inducida en el sistema. Obviamente, estas instancias se declararán en base a los descriptores de drivers incluidos como elementos declarados en el descriptor del nodo.

Descriptor de una red de comunicaciones: Communication_Network

Un elemento contenedor del tipo *Communication_Network* agrupa los modelos de los recursos que constituyen una red de comunicaciones utilizada como medio de comunicación entre diferentes nodos de la plataforma. Corresponde a otro de los elementos que se identifican de forma independiente en el proceso de despliegue de una aplicación.

Un elemento *Communication_Network* se formula de acuerdo a las siguientes reglas:

- Contiene dos recursos agregados obligatorios:
  - *Network*: Recurso de tipo *Network_Instance*, que modela el dispositivo hardware que permite el envío de mensajes.
  - *Scheduler*: Recurso de tipo *Primary_Scheduler_Instance*, que modela la gestión de la capacidad y la política de planificación del dispositivo anterior para el envío de mensajes.
- Puede agregar más recursos, sólo de tipos que extiendan al tipo *Comm_Network_Aggr_Rsrc* (ver anexo A).
- Puede declarar elementos, tanto públicos como privados, de tipos que extiendan a la clase *Comm_Network_Elem* (ver anexo A).
- El tipo de parámetros que puede declarar un elemento de este tipo está restringido a aquellos declarados en el tipo *Communication_Network_Param* (ver anexo A).

Descriptor de un módulo software: Software_Module

Un elemento contenedor del tipo *Software_Module* agrupa el conjunto de elementos que se requieren para modelar el comportamiento de un módulo software reutilizable. Como se introdujo en la sección 2.1.2, el modelo de tiempo real de un módulo incluye la información que describe la capacidad de procesamiento que requieren los métodos que ofrece, los posibles mecanismos que utiliza para sincronizar la ejecución concurrente de sus actividades internas, y en el caso de módulos activos, los modelos de las transacciones que gestiona. Los módulos son los únicos elementos que se van a emplear en la definición de la arquitectura software de una aplicación, cuya descripción se utilizará como base para la generación del modelo final de la aplicación.

Todo descriptor de módulo posee un parámetro predefinido, *Host,* que referencia al nodo de procesamiento en el que el módulo es instanciado. A través de este parámetro las herramientas acceden a la información sobre la capacidad de procesamiento disponible, y así calculan los tiempos físicos de ejecución de los métodos ofrecidos por el módulo, que en el descriptor se describen a través de tiempos normalizados. Asimismo, a través de esta referencia podrán ser

asociados todos los elementos que agregue el modelo del módulo con el correspondiente recurso de procesamiento al que pertenecen. Si por ejemplo, el elemento agregado es el modelo de un thread, haciendo uso de este parámetro se le podrá asignar la referencia al planificador que le corresponde, como *Host.Scheduler* (gracias a las reglas de formulación de un *Processing_Node* se asegura la existencia del elemento *Scheduler*).

El descriptor de un módulo puede agregar, a través de la asociación *Aggr_Rsrc_List*, cualquier tipo de recurso que requiera para modelar su comportamiento interno y que corresponda a tipos que extiendan a la clase *Software_Module_Aggr_Rsrc* (ver anexo A). Si posee alguna transacción interna, que se ejecuta en el sistema como consecuencia de la propia instanciación del módulo, deberá incluir su modelo en el apartado *Aggr_Trans_List*. Las restricciones o normas de elaboración más importantes aparecen en el apartado de elementos declarados públicos:

- Por cada método que el módulo ofrezca como parte de su funcionalidad, deberá incluirse su modelo de tiempo real. Para ello puede utilizarse cualquiera de los elementos de modelado derivados de *Requested_Usage*, en función de las características de su código interno. El elemento de modelado deberá identificarse con el mismo nombre que el método funcional que modela.
- En el caso de módulos activos, que soporten transacciones de negocio, se incluirán los modelos parametrizados de cada una de ellas. Estas transacciones serán utilizadas en la fase de descripción de la carga de trabajo del sistema.

Además, si el módulo hace uso de otros módulos para implementar su funcionalidad, su descriptor deberá definir un parámetro de tipo *Software_Module_Ref* por cada uno de ellos. De este modo, cuando se incluye una instancia del módulo en una aplicación concreta, se podrán asignar las correspondientes referencias a los modelos de los módulos concretos con los que interacciona.

Para explicar de forma más concreta estos conceptos, la figura 2.26 muestra esquemáticamente alguno de los elementos que formarían parte de los descriptores de modelo de los módulos

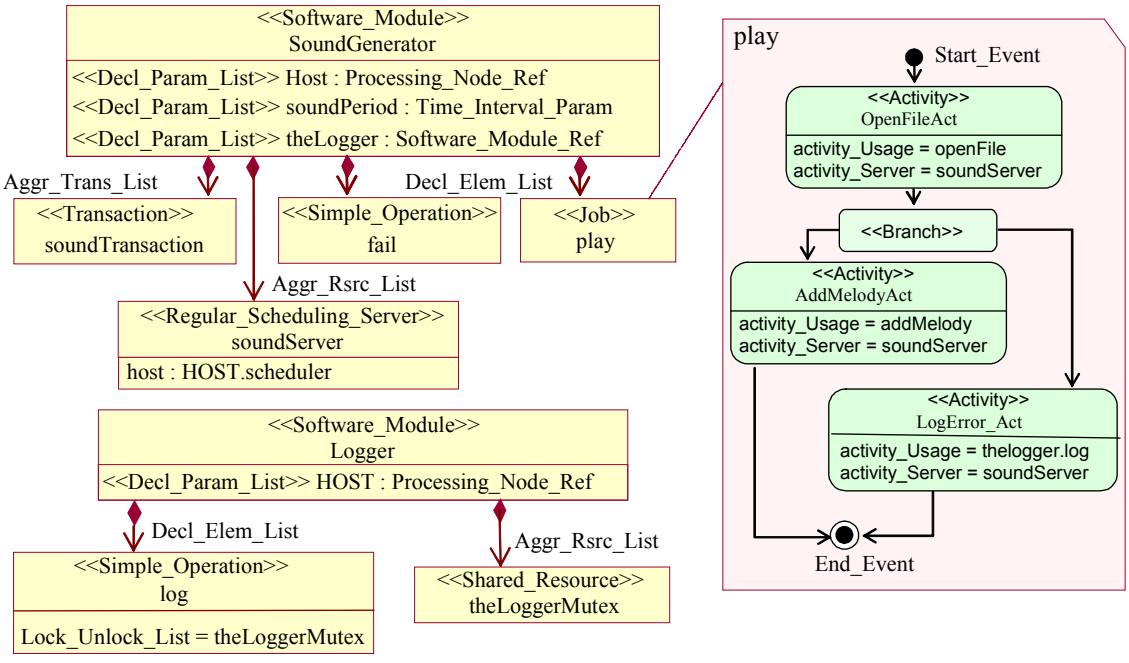


Figura 2.26: Ejemplo de descriptores de módulos software

SoundGenerator y *Logger*, que ya se usaron como ejemplo en la figura 2.3. Como vemos, ambos descriptores incluyen el parámetro *Host*. El descriptor del módulo *SoundGenerator* incluye un parámetro de tipo *Software_Module_Ref*, de nombre *theLogger*, a través del que se referencia el modelo del módulo requerido. Además, define el parámetro *soundPeriod*, que corresponde al periodo con que se va a activar la transacción agregada *SoundTransaction*. El método *play* ofrecido por el módulo *SoundGenerator* se modela mediante un *Job*, ya que involucra una relación compleja de flujo entre actividades. Una de dichas actividades consiste en la ejecución del método *log* en el módulo servidor, por lo que el elemento introducido en el modelo del *Job* es una referencia a dicha invocación, formulada con respecto al parámetro *theLogger*. Por su parte, el modelo del módulo *Logger*, incluye como elemento declarado el modelo del método *log*, que en este caso se modela mediante una *Simple_Operation*, ya que corresponde a la ejecución de un bloque simple de código dentro del módulo. La ejecución de *log* debe realizarse en modo protegido por lo que se incluye como recurso agregado un elemento de tipo *Shared_Resource*, *theLoggerMutex*, que modela un recurso compartido que utiliza el protocolo de herencia de prioridad.

En la figura 2.27 se muestra un posible despliegue de una aplicación formada por dos instancias de cada tipo de módulo, *mySoundGenerator* y *myLogger*. Ambas instancias se ejecutan en un único procesador, *myProc*, de tipo *MaRTEPCNode*. La especificación del despliegue de la aplicación es dependiente de la tecnología empleada, pero siempre ha de incluir información acerca de las instancias que forman la aplicación, las conexiones entre ellas y los nodos de la plataforma en las que son ejecutadas. A partir de dicha descripción, las herramientas declararán las correspondientes instancias de modelo de tiempo real, resolviendo automáticamente sus parámetros. Entre ellos, la herramienta asigna automáticamente a cada instancia de módulo el modelo del nodo en el que se ejecuta (a través del parámetro *Host*). En función de los enlaces entre módulos, se puede asignar al parámetro *theLogger* en el módulo *mySoundGenerator*, el modelo del módulo *myLogger*. Por último, es necesario asignar valor al resto de los parámetros definidos en los descriptores, en este caso el parámetro *soundPeriod*. Para ello, y como se explicará en detalle en el capítulo 3, es necesario ampliar la descripción de las aplicaciones y de los despliegues de las aplicaciones, con el objetivo de poder asignar valores a estos parámetros de tiempo real a través de estos descriptores sin necesidad de acceder a los modelos internos.

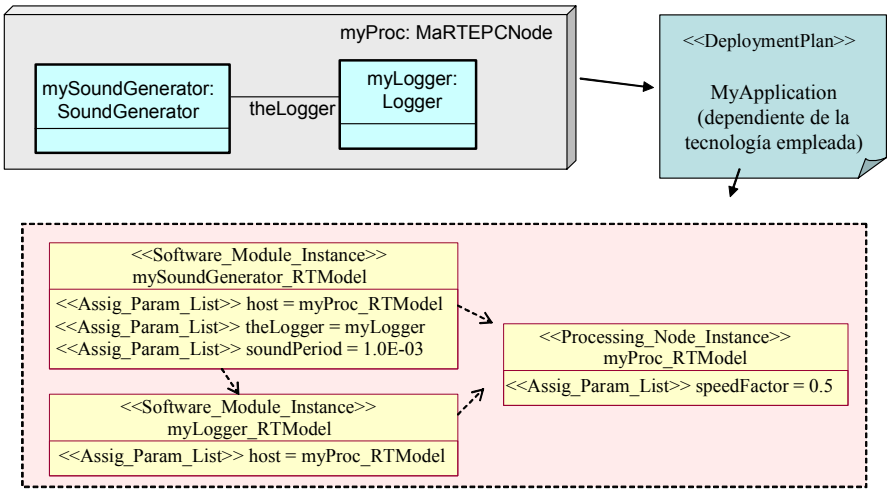


Figura 2.27: Obtención de instancias de modelo en base al despliegue de la aplicación

Descriptor de un servicio de comunicaciones: Communication_Service

En todo proceso de modelado de aplicaciones de tiempo real se necesita caracterizar la secuencia de actividades que se ejecutan cuando un módulo realiza una invocación en otro módulo. Cuando la aplicación es distribuida, modelar este tipo de invocaciones requiere conocer el modelo de los procesadores en que está instalado cada módulo, así como el modelo de la red de comunicaciones que transfiere los mensajes con los que se implementa la invocación. La secuencia concreta de actividades ejecutadas depende del tipo de servicio de comunicaciones empleado en la conexión entre los módulos pues el comportamiento es distinto en el caso de una invocación CORBA, Java RMI, etc. Por ello, el paquete *rtmod_Containers* incluye un nuevo tipo de descriptor, denominado *Communication_Service*, que permite describir el modelo de las diferentes opciones con las que un servicio de comunicaciones permite realizar una invocación. Estas opciones resultan de la combinación de varios aspectos, como la localización (invocaciones locales o remotas) o la sincronización (invocaciones síncronas, asíncronas con y sin resultados, etc.).

Cada modo de invocación soportado se modela como un escenario parametrizado (a través de un elemento de tipo *Job*), que describe la secuencia de actividades a incluir en el modelo final del sistema cada vez que una invocación entre módulos sea realizada a través del servicio. La parametrización de estos elementos resulta muy compleja, pues su comportamiento temporal final depende de multitud de elementos: la localización de los módulos, las características del servicio concreto invocado, la red de comunicaciones a través de la que se realiza, etc. Como ejemplo, en la figura 2.28 se muestra el modelo parametrizado que corresponde a una invocación remota síncrona realizada a través de un determinado servicio de comunicaciones. En la parte izquierda de la figura aparecen el conjunto de parámetros definidos para el escenario y en la parte derecha el conjunto de actividades que se incluirían en el modelo por cada invocación realizada a través de este mecanismo. Estas actividades se definen en función de los correspondientes parámetros.

Por ejemplo, supongamos el caso en que dentro de una transacción *t1* y en su actividad *step4* (dentro de una misma transacción, se puede invocar un mismo método en distintas fases), el módulo *mySoundGenerator* que se ejecuta en el nodo *Proc1*, invoca el método *log* que es

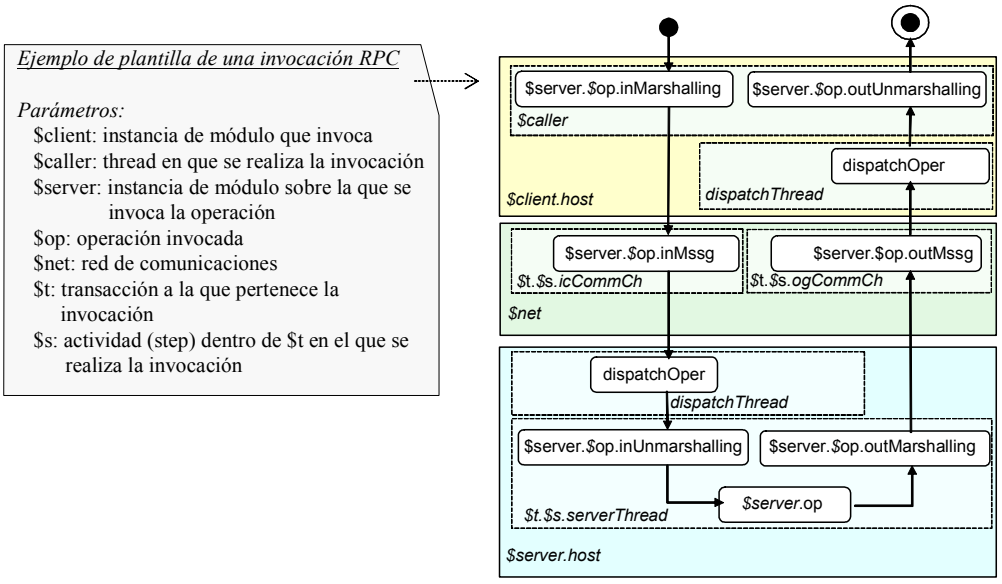


Figura 2.28: Ejemplo de plantilla de invocación remota síncrona en el modelo de un servicio de comunicaciones

ofertado por el módulo *myLogger,* que está instalado en el nodo *Proc2,* a través de este servicio, usando la red *Net1*. La asignación de parámetros sería la siguiente: *$client = mySoundGenerator*, *$server = myLogger*, *$t = t1*, *$s = step4*, *$op = log*, *$net = Net1*. La secuencia de actividades que incluiría la herramienta de composición de modelos en el modelo final de la aplicación, que se muestra en la figura 2.29, sería la siguiente:

1. En el thread del nodo *Proc1* que invoca (supongamos que corresponde al thread *t1Thread*, con lo que *$caller= t1Thread*) se ejecuta la operación de serialización (*marshalling*) de los parámetros de entrada del método *log*. Estos datos se obtienen del modelo temporal del método, que está incluido en el modelo del módulo *myLogger.*
2. En la red *Net1* se transmite un mensaje en una sesión planificable denominada *t1.step4.icCommCh.* El modelo temporal de la actividad de transferencia del mensaje se obtiene también del modelo del módulo *myLogger*, mientras que el modelo de la sesión de comunicación se obtiene del modelo del servicio de comunicaciones.
3. En el procesador *Proc2* en que se ejecuta el módulo servidor, se ejecuta una operación de recepción y despacho (*dispatching*) del mensaje de invocación. Esta actividad se ejecuta en el thread de gestión de comunicaciones instalado en *Proc2*, cuyo comportamiento temporal se encuentra descrito en el propio modelo del servicio de comunicaciones (el modelo de dicho thread constituye un elemento agregado en el descriptor del servicio de comunicaciones). El modelo temporal de esta actividad (*dispatchOper*) también forma parte del modelo del servicio de comunicaciones.
4. En el procesador *Proc2* se ejecuta la operación de deserialización (*unmarshalling*), cuyo modelo temporal se encuentra descrito en el modelo del módulo *myLogger.* En este caso, dicha operación es ejecutada por un thread denominado *t1.step4.serverThread*, cuyo modelo forma parte del modelo del servicio de comunicaciones. El mismo thread se encarga, a continuación, de la ejecución del método *Oper*, cuyo modelo se encuentra en el modelo del módulo *myLogger.* Una vez finalizada la ejecución, se realiza también en el mismo thread la operación de serialización de los parámetros de retorno.
5. Se transmite el mensaje de retorno a través de *Net1* en una sesión planificable denominada *t1.step4.ogCommCh*. El modelo temporal de la actividad de transferencia del mensaje se obtiene también del modelo del módulo *myLogger.*

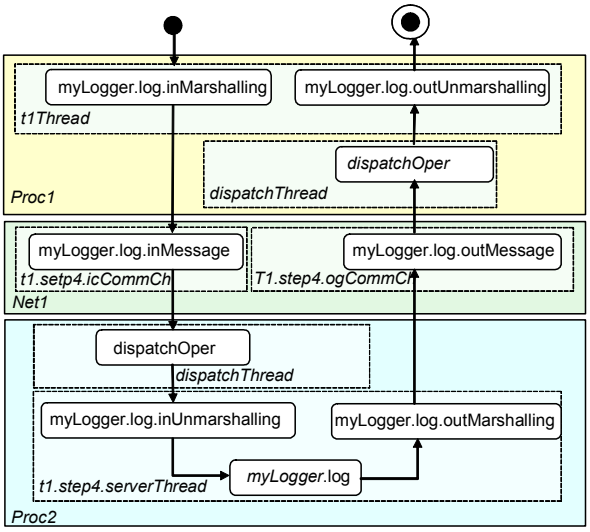


Figura 2.29: Modelo final de una invocación remota síncrona

6. En el procesador *Proc1* se ejecuta la operación de despacho el mensaje de retorno. Esta actividad se ejecuta en el thread de gestión de comunicaciones instalado en *Proc1*, y su comportamiento temporal se encuentra descrito en el propio modelo del servicio de comunicaciones (será el mismo en ambos procesadores).
7. El thread del módulo *mySoundGenerator* ejecuta la operación de deserialización de los parámetros de salida de la operación invocada, cuyo modelo se extrae del modelo del módulo *myLogger*.

En el descriptor del módulo *mySoundGenerator*, la invocación aparece como *theLogger.log,* siendo *theLogger* el parámetro que referencia al módulo requerido (ver figura 2.26). Debe ser la herramienta de composición la que manejando información de diferentes fuentes (la conexión entre las instancias de módulos, el modelo del servicio invocado, el servicio de comunicaciones empleado, el modelo de los procesadores implicados, la red de comunicaciones, etc.) genere la secuencia de actividades correcta correspondiente a la invocación final.

2.3.1.4. Paquete rtmod_System

Los paquetes anteriores agrupan primitivas de modelado que permiten describir de manera independiente el comportamiento temporal de los diferentes módulos que forman un sistema de tiempo real con estructura modular. El paquete *rtmod_System* define los elementos de modelado que permiten formular el modelo de un sistema completo formado por un conjunto de instancias de módulos desplegadas en una plataforma de ejecución distribuida. Los elementos definidos en este paquete tienen todos naturaleza de instancia. Representan elementos directamente instanciables (sistema, aplicación, plataforma, etc.) que no se describen en base a ningún descriptor, sino por agregación de otros elementos (todos ellos instancias).

En la sección 2.2 se explicó el proceso a seguir para la formulación del modelo de un sistema de tiempo real de forma genérica. En este apartado se muestra la implementación de dicho proceso dentro de la metodología Mod-MAST. El conjunto de elementos de modelado que se define en este paquete con dicho propósito se muestra en la figura 2.30. El elemento *System* sirve para describir el modelo de un sistema completo, que como ya se explicó anteriormente, queda definido por el modelo de la plataforma de ejecución, descrita a través de un elemento de

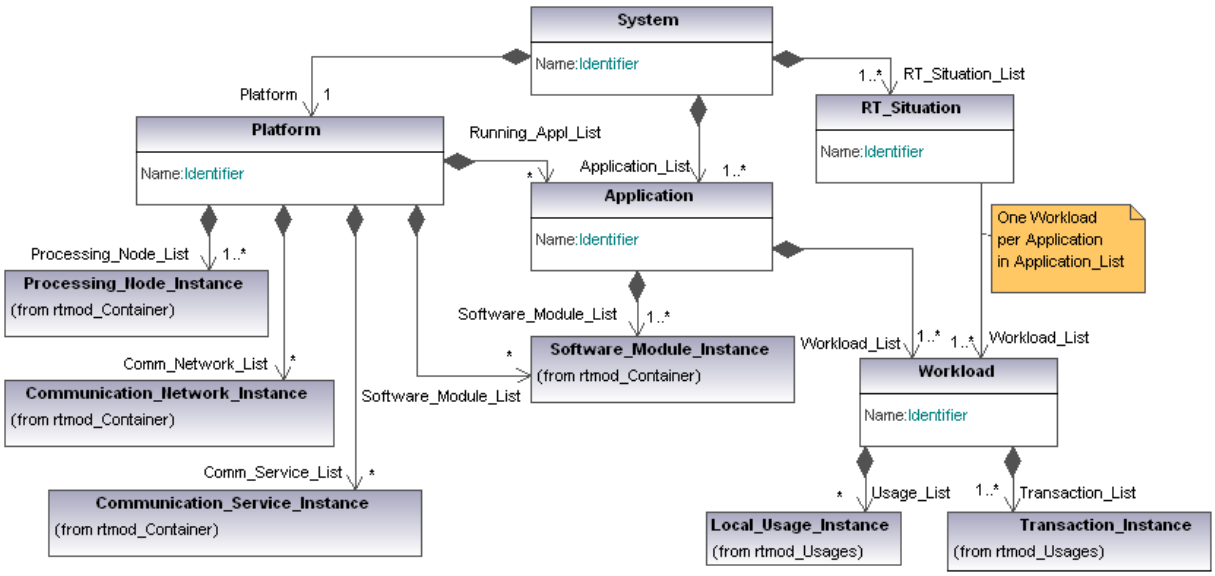


Figura 2.30: Elementos del paquete rtmod_System

tipo *Platform*, y los modelos de cada una de las aplicaciones que ejecutan concurrentemente en la plataforma, cada una modelada a través de un elemento de tipo *Application*. La descripción del sistema se completa con la definición de las situaciones de tiempo real sobre las que se quiere aplicar el análisis. Cada situación, *RT_Situation*, se describe a través de una lista de cargas de trabajo, una por aplicación, que describe la carga con la que cada aplicación contribuye a la carga total del sistema.

Cada elemento *Application* utilizado para describir una aplicación incluye:

- El conjunto de instancias de modelo de los módulos software (*Software_ Module_Instance*) que forman la aplicación.
- La lista de cargas de trabajo bajo las que puede ser ejecutada la aplicación, *Workload_List*, cada una correspondiente a un modo de operación posible. Una carga de trabajo se describe a través de un elemento de tipo *Workload* y consiste en la declaración del conjunto de instancias de transacciones que se ejecutan en dicho modo de operación.

El modelo más genérico de una plataforma, *Platform*, se describe como:

- El conjunto de instancias de modelo de los nodos (*Processing_Node_Instance*) que forman la plataforma.
- Las instancias de modelo correspondientes a las redes a través de las que se conectan los nodos (*Communication_Network_ Instance*).
- Las instancias de modelo de los servicios de comunicación (*Communication_Service_ Instance*) soportados por los nodos y las redes.

Para hacer el modelo más flexible, se pueden incluir también en el modelo de la plataforma, instancias correspondientes a módulos software e incluso a aplicaciones. El objetivo que se persigue con esta posibilidad es el de focalizar la atención en aquellas aplicaciones a las que se está aplicando el proceso de diseño y análisis de tiempo real. Cuando queremos añadir una nueva aplicación (o conjunto de aplicaciones) en una plataforma, el resto de aplicaciones con las que ha de compartir la capacidad pueden estar previamente diseñadas, incluso sin ofrecer capacidad de modificar sus características con el objetivo de hacer planificable la nueva aplicación. Por tanto, este conjunto de aplicaciones no representan el objetivo del proceso de diseño y se pueden formular como parte del modelo de la plataforma.

2.3.2. Formalización de modelos

La construcción del modelo de tiempo real de un sistema requiere gestionar una gran cantidad de información, que se encuentra distribuida en diferentes fuentes. Aunque el metamodelo Mod-MAST se ha definido usando UML, se optó por no utilizar esta estrategia para el desarrollo y gestión de los modelos. Para simplificar el desarrollo de las herramientas de composición de modelos se ha implementado la metodología utilizando tecnología XML. Se han definido un conjunto de plantillas *W3C-Schema* a través de las que se formalizan los elementos del metamodelo, de modo que cada descriptor o instancia de modelo puede ser descrito a través de un fichero XML. También el modelo MAST final generado por la herramienta de composición se formula de acuerdo a su correspondiente plantilla *W3C-Schema*.

En la figura 2.31 se muestra un esquema del tipo de ficheros que se manejan. En la parte superior izquierda de la figura se muestran las siete plantillas *W3C-Schema* a los que se ha mapeado el metamodelo Mod-MAST. Cada plantilla corresponde a uno de los paquetes definidos dentro del metamodelo, y contiene por tanto los descriptores XML correspondientes a los elementos de modelado definidos en cada uno de ellos.

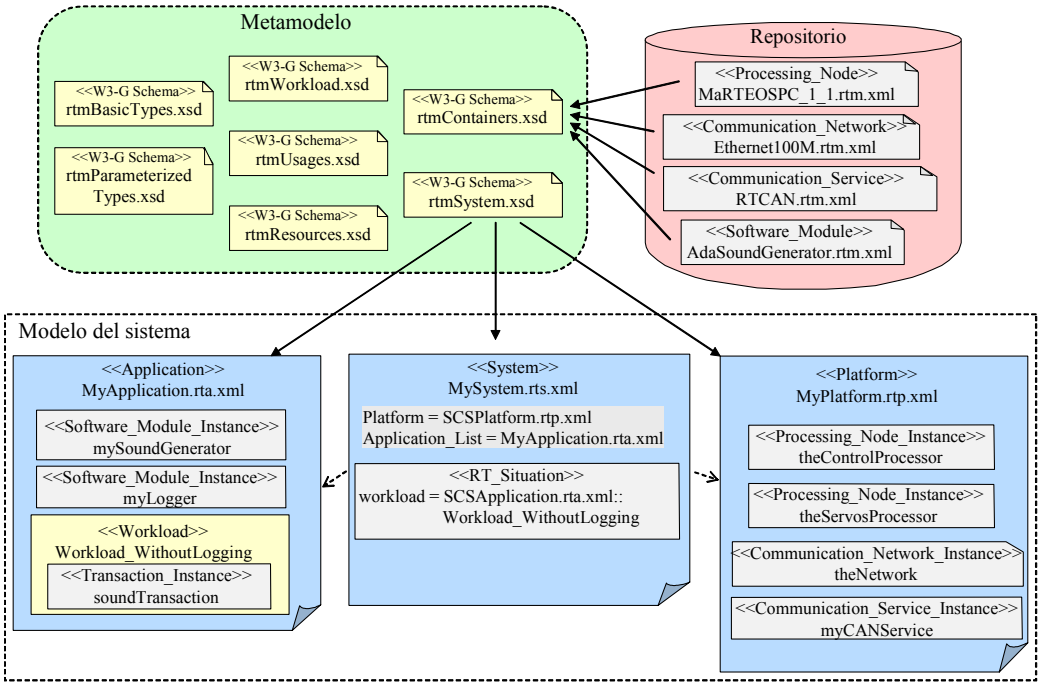


Figura 2.31: Schemas y ficheros XML utilizados en Mod-MAST

En la parte derecha se muestra el tipo de ficheros que se van a poder elaborar y almacenar en el repositorio de la plataforma de diseño. Como se puede observar, corresponden con los elementos que se definen en el paquete *rtmod_Containers*, que representan los únicos descriptores de modelo que pueden ser elaborados y almacenados de manera independiente: *Processing_Node*, *Communication_Network*, *Communication_Service* y *Software_Module*. Todos ellos poseen la extensión .rtm.xml y sus correspondientes plantillas se encuentran en el schema *rtmContainers.xsd*.

Cuando se desarrolla el modelo de un sistema completo, se declara un elemento de tipo *System* a través de un fichero con extensión *.rts.xml* (como *MySystem.rts.xml* en la figura). Para elaborarlo es necesario contar con el modelo de la plataforma. Éste se describe siempre de forma independiente en un fichero con extensión *.rtp.xml*, cuyo elemento raíz es de tipo *Platform* (como *MyPlatform.rtp.xml*). Por otro lado, para describir las aplicaciones que forman el sistema, cada una de ellas en una determinada situación de tiempo real, se requiere:

- El modelo de la aplicación, que en este caso puede ser declarado directamente en el modelo del sistema, o como se muestra en la figura, hacer referencia al fichero externo que lo contiene, con extensión *.rta.xml* (*MyApplication.rta.xml*).
- El modelo de la carga de trabajo correspondiente a la situación de tiempo real modelada, que deberá referenciar a alguna de las posibles cargas de trabajo definidas para la aplicación anterior (en la figura, *Workload_WithoutLogging*).

Estos tres últimos tipos de ficheros se declaran de acuerdo a los elementos definidos en la plantilla *rtmSystem.xsd*.

La estrategia que se ha empleado para abordar el problema de la parametrización de elementos en XML ha sido la utilización de símbolos especiales. Un descriptor incluye la lista de los parámetros que define, para los cuales se especifica su nombre y su tipo. Cuando desde el interior del descriptor se hace referencia a alguno de los parámetros, se identifica mediante la utilización del símbolo @ antepuesto y pospuesto al nombre del parámetro. De este modo se

simplifica el desarrollo de herramientas, ya que cuando éstas procesan un descriptor y encuentran un valor asignado que comience por dicho símbolo, la herramienta automáticamente lo sustituye por el valor asignado al correspondiente parámetro en la instancia de modelo. Con el mismo fin, se declara el carácter @ no válido para ser utilizado dentro de los identificadores.

2.3.3. Herramientas de procesado y análisis de modelos en Mod-MAST

Las herramientas de análisis de planificabilidad disponibles en el entorno MAST operan sobre un modelo transaccional puro. Por lo tanto, si se quieren utilizar estas herramientas para procesar los modelos modulares formulados con Mod-MAST, éstos deben ser previamente transformados a una formulación compatible con las herramientas (puramente transaccional).

Para llevar a cabo el proceso de transformación se ha desarrollado la herramienta *MM2M Compiler (ModMAST-To-MAST Compiler)*. Con el objetivo de que en el futuro se implemente utilizando estrategias MDA/MDE, la herramienta se ha diseñado como una secuencia de transformaciones, que se muestra en la figura 2.32. Como se describirá posteriormente en el capítulo IV, esta transformación constituye sólo el paso final de una cadena de transformaciones más amplia que permite obtener los modelos de tiempo real de las aplicaciones desde niveles de abstracción superiores.

En la herramienta *MM2MCompiler* las transformaciones terminales T2M (texto a modelo) y M2T (modelo a texto) son triviales, ya que las correspondientes codificaciones XML de partida y de destino ofrecen una correspondencia directa entre las estructuras de los tipos complejos XML y las clases de los correspondientes modelos Mod-MAST y MAST. La complejidad se encuentra en la transformación central M2M (modelo a modelo) que tiene como entrada los objetos del modelo Mod-MAST del sistema (conforme al metamodelo Mod_MAST) y como salida los objetos del modelo MAST (conforme al metamodelo MAST).

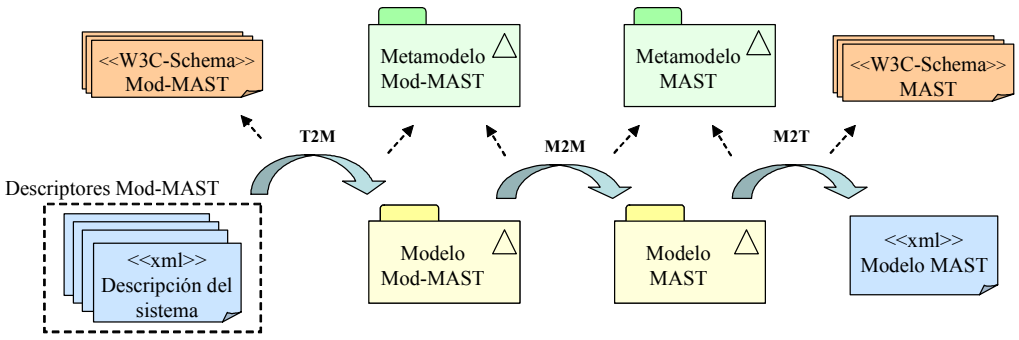


Figura 2.32: Obtención del modelo MAST de una aplicación modular

La transformación del modelo Mod-MAST al modelo MAST se realiza en dos etapas, que se reflejan en la figura 2.33. El proceso comienza con la obtención del modelo de instancias del sistema. Este modelo es un modelo efímero, compuesto por listas de instancias de elementos de tipo contenedor, esto es, aquellas que representan los elementos de primer nivel del sistema: módulos software, nodos, redes y servicios de comunicación.

El proceso de generación de este modelo consta de dos etapas:

- En la primera etapa se procesan todas las instancias de modelo declaradas en el modelo del sistema y correspondientes a elementos estructurales de la aplicación, esto es, a contenedores (*Container_Instance*). Por cada una de ellas se genera una instancia Mod-

MAST como una copia de su correspondiente descriptor de modelo, pero sustituyendo cada parámetro por el valor asignado en la declaración de la instancia. Al final de esta fase, cada instancia representa un modelo no parametrizado, en el que los únicos elementos que pueden quedar todavía sin resolver son aquellos elementos declarados (campo *Decl_Elem_ List*) que definan sus propios parámetros.

- En la segunda etapa se procesa el modelo reactivo del sistema. Las transacciones que contribuyen a la carga de trabajo del sistema tienen dos orígenes:
  - Por un lado las transacciones que son propias de los elementos que forman parte de la aplicación y de la plataforma, y que se encuentran declaradas en el campo *Aggr_Trans_List* de los descriptores de esos elementos. A los parámetros de estas transacciones se le asigna valor desde los parámetros de las instancias de los elementos en que están definidas, por lo que los valores ya habrán sido asignados en el paso anterior.
  - El segundo grupo de transacciones son las definidas en la carga de trabajo de la aplicación, que están declaradas en el campo *workload* de cada *RTSituation*. Cada una de ellas debe ser procesada con el objetivo de añadir su modelo resuelto, no parametrizado, a la correspondiente instancia de módulo que la genera, en el campo *Aggr_Trans_List*.

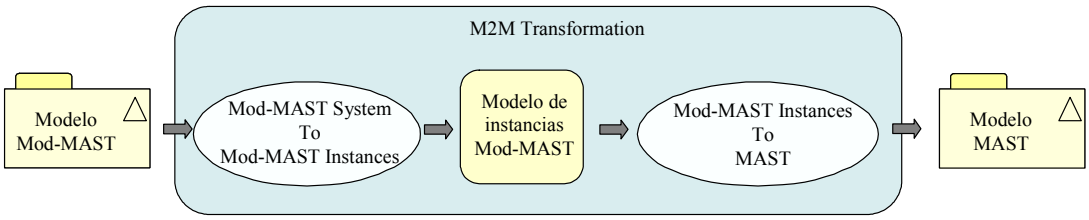


Figura 2.33: Etapas de la transformación de Mod-MAST a MAST

El procesado del modelo de una transacción es un proceso complicado, ya que puede implicar varios niveles de resolución de parámetros y referencias. Por ejemplo, una forma de uso referenciada en una actividad dentro de una transacción puede a su vez referenciar otra forma de uso externa, y así recursivamente. A través de un ejemplo sencillo se resaltan algunos aspectos a tener en cuenta durante el procesado de una instancia de transacción. En la figura 2.34 se muestra un ejemplo de descriptor de módulo software, *ModuleA*, cuyas características son las siguientes:

- Además del parámetro predefinido *Host*, declara un parámetro *usedModule* a través del que se asignará la referencia al modelo del módulo que se usa para implementar parte de la funcionalidad del job *JobA*.
- Agrega un elemento de tipo *Regular_Scheduling_Server*, *ServerA,* que modela la entidad de planificación que se emplea para ejecutar algunas de las actividades internas del módulo.
- Declara varias formas de uso (operaciones simples y un *Job*), así como una transacción *TransA* de la que se podrán añadir instancias en la carga de trabajo de la aplicación. Esta transacción representa un ejemplo del anidamiento de formas de uso, ya que en su primera actividad invoca *JobA* (definido en el propio módulo), que a su vez invoca una forma de uso, *operB*, que debe estar definida en el modelo del módulo que se referencia a través del parámetro *usedModule*.

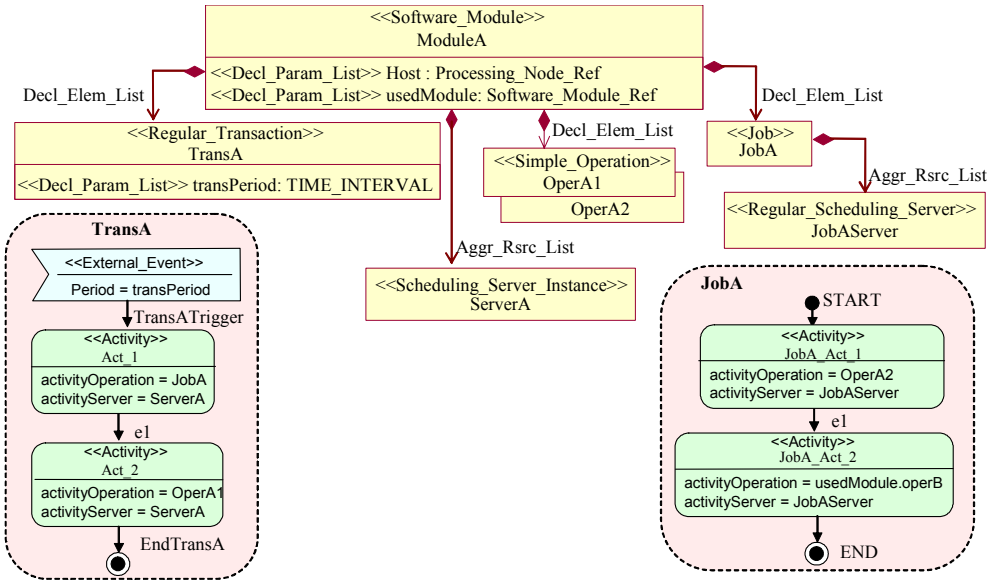


Figura 2.34: Descriptor de un módulo y su transacción declarada

Supongamos una aplicación formada por un módulo de tipo ModuleA, *myModuleA*, y otro módulo que ofrezca la interfaz requerida, *myModuleB*. La carga de trabajo de la aplicación incluye una instancia de la transacción TransA, *myTransA*. En la figura 2.35 se muestra como se declararían las instancias *myModuleA* y *myTransA*, asignando valores a sus correspondientes parámetros. En la misma figura se muestra la cadena de actividades finales que resultan de la instanciación de la transacción, en base a la cual se puede explicar el proceso de instanciación:

- La primera actividad, *Act_1*, consiste en la ejecución de *JobA*. Será necesario acceder al descriptor de dicho *Job* (que en este caso no es parametrizado, luego deberá estar ya plenamente resuelto en la instancia de módulo *myModuleA*), añadiendo cada actividad definida en dicho descriptor a la instancia de la transacción. En la figura 2.34 se muestra como el descriptor de *JobA* define un *Scheduling_Server_Instance* agregado,

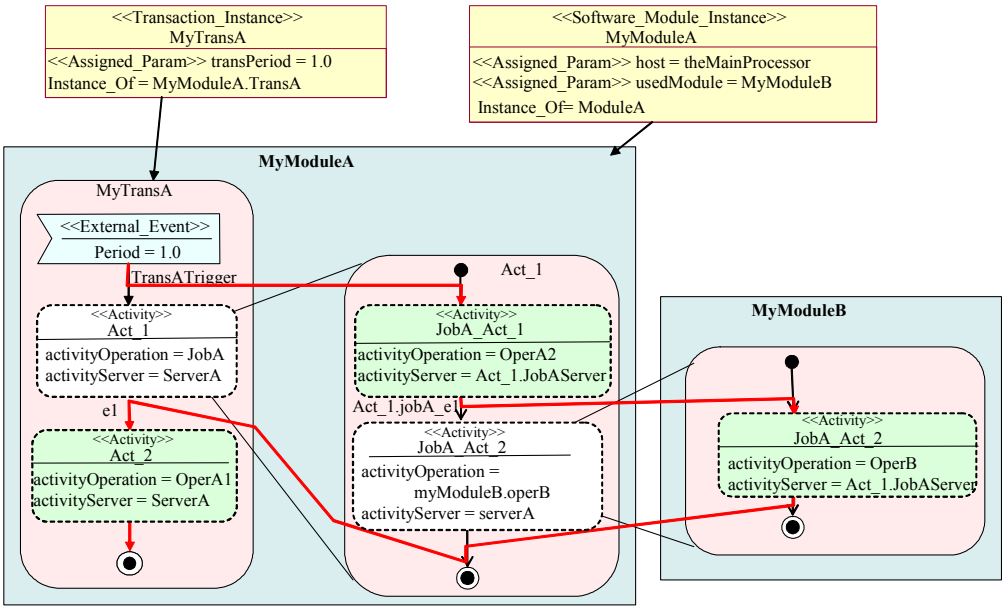


Figura 2.35: Resolución parcial de la transacción de ejemplo

JobAServer, que modela el thread que se encarga de su ejecución. Por cada actividad desde la que *JobA* sea invocado, debe añadirse a la instancia *myModuleA* un *Scheduling_Server_Instance* cuyo nombre se construye en base a la actividad en la que *JobA* es invocado (en este caso *Act_1.JobAServer*). Utilizando esta regla para la nomenclatura podemos distinguir en el modelo final las diferentes invocaciones de un mismo modelo de *Job*. Este *Scheduling_Server_Instance* está asociado a todas las actividades que se ejecutan en el *Job*, dos en este caso:

  - la primera corresponde a la ejecución de una operación simple, *OperA2*, perteneciente al propio *myModuleA*, por lo que su modelo se encontrará definido en el campo *Decl_Elem_List* de la instancia, y se puede hacer referencia directa a él como *myModuleA.OperA2*.
  - la segunda actividad corresponde a la invocación del servicio *OperB* en el módulo *myModuleB*. Será necesario acceder a la instancia de módulo *myModuleB* para poder añadir el modelo de dicha operación. En este caso se trata de una operación simple, pero en un caso más complejo podría a su vez tratarse de un nuevo *Job*, que habría de ser procesado de la misma manera. En este caso se hace referencia directa a la operación, como *myModuleB.OperB*.

- La segunda actividad ejecutada en *myTransA* corresponde a la ejecución de la operación simple *OperA1* del propio módulo *myModuleA*. Su *Scheduling_Server* asociado es *ServerA*, que corresponde a un elemento agregado por el propio módulo y por tanto, habrá sido instanciado y agregado a la instancia de módulo en la fase anterior, pudiéndose hacer referencia a él como *myModuleA.ServerA*.

El modelo de esta transacción tal y como aparecería en el modelo MAST final es el que se muestra en la figura 2.36. En ella aparece la secuencia final de actividades con los nombres absolutos que recibe cada elemento involucrado (eventos, operaciones, servidores de

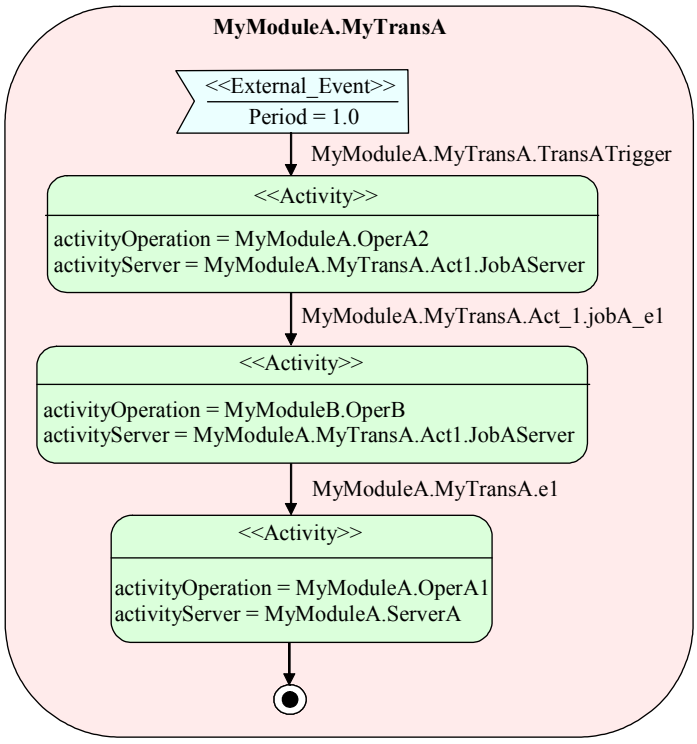


Figura 2.36: Modelo MAST final de la transacción de ejemplo

planificación, etc.). Como vemos, la regla general es que todos los elementos del modelo MAST reciben un nombre que identifica su origen:

- En caso de elementos agregados por un módulo: [nombreMódulo].[nombreElemento]. Por ejemplo, el servidor de planificación *myModuleA.serverA*.
- En caso de elementos agregados por una transacción: [nombreMódulo].[nombreTransacción].[nombreElemento]. Por ejemplo, el evento de lanzamiento de la transacción *myModuleA.myTransA.TransATrigger*.
- En caso de un elemento añadido en una actividad (a consecuencia del procesado de un *Job*): [nombreMódulo].[nombreTransacción].[nombreActivity].[nombreElemento]. Por ejemplo el servidor de planificación *myModuleA.myTransA.Act_1.JobAServer.*

Una vez obtenido el modelo de instancias del sistema, la segunda parte del proceso de transformación consiste en la generación del modelo MAST del sistema a partir de él. Este proceso a su vez se puede dividir en dos subfases:

- En primer lugar se introduce el modelo estructural del sistema. La herramienta recorre de nuevo todas las instancias de contenedores, y añade al modelo MAST final los elementos agregados por cada uno. Realmente, lo que se añade es el elemento o elementos MAST a los que se mapea cada elemento de bajo nivel del metamodelo Mod-MAST. Como ejemplo, por cada *Processing_Node_Instance* declarado en el modelo de la plataforma, se añadirá al modelo final un *Regular_Processor* y un *Primary_Scheduler*. La correspondencia es directa en casi todos los casos, al haber sido desarrollada Mod-MAST como una extensión de MAST. Un ejemplo de elemento agregado en este paso sería el *Scheduling_Server myModuleA.ServerA* que hemos visto en el ejemplo anterior, que se agrega al procesar la instancia de módulo *MyModuleA*.
- La segunda subfase introduce la parte reactiva del modelo, y consiste en añadir los modelos de cada una de las transacciones que forman la carga de trabajo total del sistema, las cuales corresponden a todas aquellas incluidas en el campo *Aggr_Trans_List* de cada instancia de contenedor. La inclusión del modelo de las transacciones puede requerir la inclusión de otros elementos de modelado, que se encuentren en el campo *Decl_Elem_List* de la correspondiente instancia. Deberá elaborarse una estrategia para evitar que se añadan elementos duplicados al modelo, como consecuencia de que sean invocados desde varios puntos de una transacción o de diferentes transacciones. El modo más sencillo consiste en marcar como añadido cada elemento declarado que se incorpore al modelo. Ejemplos de este tipo de elementos serían las operaciones *MyModuleA.Oper1 y MyModuleA.Oper2* del ejemplo anterior.

Una vez formulado el modelo del sistema en el formato adecuado, podemos usarlo como entrada para el conjunto de herramientas que MAST ofrece para el análisis de sistemas de tiempo real. En el capítulo 5 se detallan las diferentes herramientas que se ofrecen y el modo en que se pueden ser utilizadas.

2.4. Compatibilidad con el perfil MARTE de OMG

El consorcio OMG ha elaborado diferentes propuestas para la estandarización del modelado de sistemas de tiempo real a fin de que sobre los modelos puedan interoperar herramientas de análisis y diseño de diferentes fabricantes. La iniciativa más reciente la constituye el perfil MARTE (*UML Profile for Modeling and Analysis of Real-time and Embedded Systems*) [MARTE]. MARTE incluye un subperfil, denominado SAM (*Schedulability Analysis Model*), que trata específicamente el desarrollo de modelos orientados al análisis de planificabilidad, con

el que se sustituye el perfil SPT (*UML Profile for Schedulability, Performance and Time*) [SPT], que tenía un ámbito más reducido. El perfil MARTE incluye más subperfiles enfocados a otros aspectos del modelado, como son modelos de diseño de software, de hardware, modelos orientados a análisis de prestaciones (*performance*), etc. Aunque durante el desarrollo de nuestra metodología, MARTE todavía no constituía una especificación formal, se ha tratado de buscar una alineación máxima con él.

MARTE busca cubrir estrategias de modelado muy diferentes, por lo que en el propio perfil se definen diferentes casos de conformidad, en función del objetivo con el que se aplique. Cada caso de conformidad define el conjunto de subperfiles que deben ser soportados por las herramientas que lo implementan. Uno de los casos de conformidad identificados es el de análisis de planificabilidad. Incluye todos aquellos subperfiles, o submetamodelos, que se requieren para modelar sistemas sobre los que se vayan a aplicar técnicas de análisis de planificabilidad. El caso de conformidad en este caso consiste básicamente en ser compatibles con el subperfil SAM, junto con los subperfiles de los que éste a su vez constituye una especialización (GRM, *Generic Resource Modeling*, y GQAM, *Generic Quantitative Analysis Modeling*) y aquellos de los que depende (los que definen los conceptos de modelado del tiempo, *Time*, y de propiedades no funcionales, *NFPs*).

Aunque la metodología Mod-MAST no representa una implementación formal del subperfil SAM, los conceptos de modelado utilizados son muy similares, ya que ambos se fundamentan en un modelo transaccional del sistema. Con el objetivo de mostrar la equivalencia entre los elementos de modelado utilizados en Mod-Mast y en MARTE, se van a utilizar algunos diagramas de clase, en los que se muestra para cada elemento de modelado incluido en Mod-MAST, el correspondiente concepto del perfil MARTE al que equivale semánticamente. Para mostrar estas correspondencias se han utilizados relaciones de dependencia y no de herencia o generalización, ya que los conceptos incluidos en Mod-MAST no constituyen una especialización formal de los conceptos definidos en MARTE. Se quiere dejar de manifiesto únicamente su similitud desde el punto de vista semántico, esto es, del concepto de modelado que representan, aunque el modo exacto de representación pueda variar de un caso a otro.

El modelo de análisis de un sistema en un determinado modo de operación, *AnalysisContext* en MARTE y *RT_Situation* en Mod-MAST, se elabora identificando la carga de trabajo del sistema, *WorkloadBehaviour* en MARTE y *Workload* en Mod-MAST. Esta carga de trabajo se formula como el conjunto de transacciones, *Transaction* en Mod-Mast y *EndToEndFlow* en MARTE, que se ejecutan concurrentemente en el sistema, y para las que es necesario fijar su patrón de lanzamiento, así como los requisitos temporales que deben satisfacer durante su ejecución. En la figura 2.37 se muestran los mapeados entre estos conceptos. En Mod-MAST, el conjunto de recursos utilizados para la ejecución de una determinada carga de trabajo en un contexto de análisis, que en MARTE se engloban en el elemento *ResourcesPlatform*, resulta de la unión de los recursos de la plataforma de ejecución (*Platform*) y aquellos recursos agregados por los componentes que forman las diferentes aplicaciones (*Application*) que se ejecutan en el contexto de análisis.

La forma de describir una transacción también es muy similar entre ambas metodologías. La figura 2.38 muestra las similitudes semánticas entre los principales elementos utilizados en el caso de Mod-MAST y aquellos elementos que se utilizan en MARTE. Por ejemplo, vemos como los conceptos de *Timing_Req_Observer* y *Timing_Requirement* (separados en Mod-MAST) se modelan en MARTE a través de un único elemento, denominado *TimedObserver*. El concepto de *EndToEndFlow* definido en el capítulo SAM de MARTE es sólo un elemento conceptual, que se describe en base a su patrón de disparo (*WorkloadEvent*) y al conjunto de

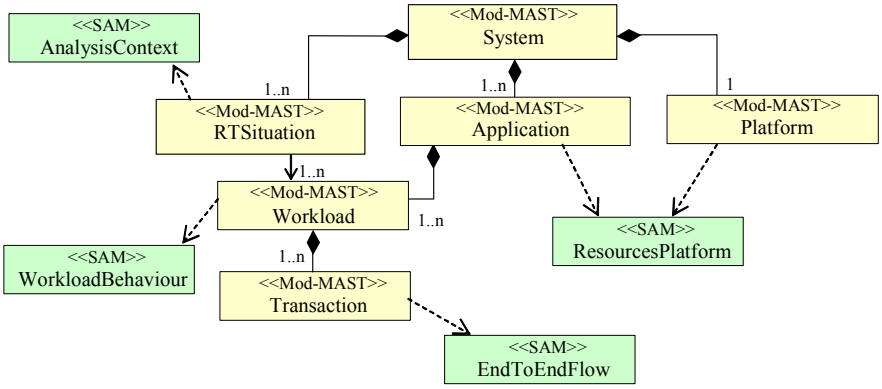


Figura 2.37: Equivalencias entre Mod-MAST y MARTE a primer nivel

actividades que desencadena (*WorkloadBehaviour*). Es por eso que el concepto de Mod-MAST equivale semánticamente a ambos conceptos, pues en Mod-MAST la secuencia de actividades ejecutadas en una transacción forma parte de su propia descripción.

La figura 2.38 muestra también como se puede realizar un mapeado entre las clases *Usage* de Mod-MAST y *ResourceUsage* de MARTE, pues ambas sirven para describir formas de uso de recursos. En ambos casos, además, la transacción se define como una especialización de la forma de uso genérica.

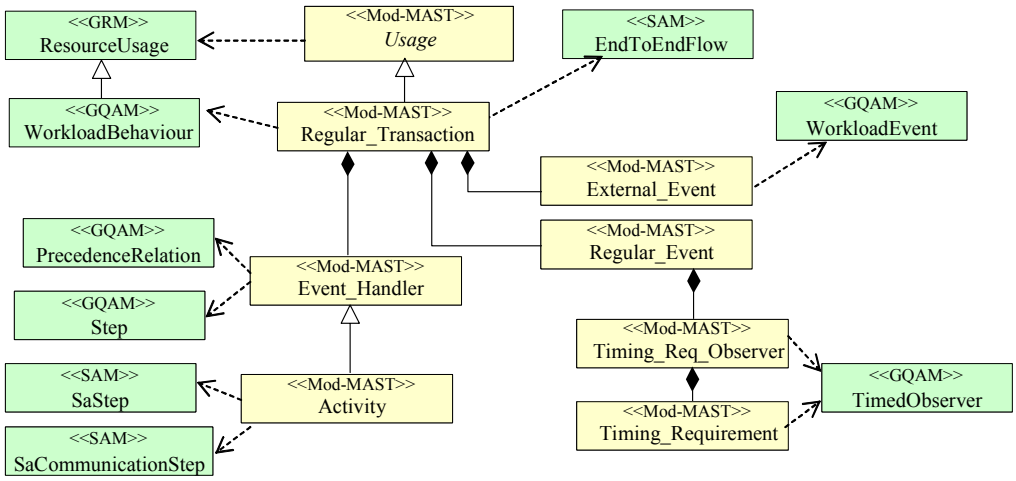


Figura 2.38: Equivalencias entre Mod-MAST y MARTE en la descripción de transacciones

Desde el punto de vista del modelado de los recursos que contribuyen, de manera positiva o negativa, a la capacidad de procesamiento disponible para la ejecución, los conceptos de modelado se mapean de forma casi directa entre ambos metamodelos. Todos los recursos extienden a la clase abstracta *Resource* (denominada así tanto en MARTE como en Mod-MAST). Entre el resto de elementos la equivalencia es la que se muestra en la tabla 2.1.

En este apartado, una de las principales diferencias identificadas concierne al modelado de la sobrecarga que se induce en los procesadores como consecuencia de la gestión del envío y recepción de mensajes a través de una red. En el modelo utilizado en Mod-MAST, estas sobrecargas se modelan mediante los elementos de tipo *Driver*. Por cada protocolo de red a través del que un procesador pueda comunicarse con otro, deberá incluirse una instancia de tipo *Driver_Instance* en la correspondiente instancia del nodo de procesamiento de tipo

Tabla 2.1: Equivalencias entre Mod-MAST y MARTE en la descripción de recursos

Mod-MAST	MARTE
Resource	GRM::Resource
Processing_Resource	GRM::Processing_Resource
Processor	GRM:ComputingResource
Regular_Processor	SAM::SaExecutionHost
Network	GRM:CommunicationMedia
PB_Network	SAM::SaCommunicationHost
Shared_Resource	SAM.:SharedResource
Synchronization_Params	GRM:MutualExclusionProtocol GRM::ProtectionParameters
Scheduling_Server	GRM::SchedulableResource
SchedulingParams	GRM::SchedulingParameters
Network_Scheduling_Server	GAQM::CommunicationChannel
System_Timer	GRM:TimingResource
Alarm_Clock	GRM:ClockResource
Ticker	GRM::ClockResource
Scheduler	GRM::Scheduler
Primary_Scheduler	GRM::Scheduler
Secondary_Scheduler	GRM::SecondaryScheduler
Scheduling_Policy	GRM::SchedulingPolicy

Processing_Node_Instance. El descriptor de un tipo de driver incluye los modelos de las operaciones que se introducen en el sistema por cada envío y recepción de un paquete, o incluso, dependiendo del tipo de red, posibles transacciones que se lleven a cabo internamente para su gestión. En el caso del perfil MARTE no se modelan estas sobrecargas por separado. En el propio modelo del procesador, descrito a través de un elemento de tipo *ExecutionHost,* se incluyen dos propiedades denominadas *commTxOverhead* y *commRcvOverhead,* que modelan la demanda de capacidad de procesamiento requerida para el envío y recepción de mensajes respectivamente. De este modo, el modelo de un procesador se encuentra restringido al modelo de una única red, para la cual se hayan calculado dichas sobrecargas. No se podría modelar un procesador con capacidad de comunicarse a través de más de un protocolo de red, si éstas tuviesen un modelo distinto de sobrecarga. Por otro lado, uno de los objetivos que persigue nuestra metodología es la reutilización de modelos, que se alcanzará en mayor grado si se modelan las características dependientes del hardware y del sistema operativo del procesador, con independencia del resto de capacidades que se le puedan añadir posteriormente, como en este caso la comunicación a través de cierto protocolo de red.

Otra diferencia, en este caso una limitación de Mod-MAST, aparece en la definición de los planificadores (*Scheduler)*. En el caso de MARTE, como vemos en la figura 2.39, la asociación entre el *Scheduler* y los *ProcessingResource* cuya capacidad gestiona es múltiple. El *Scheduler*

reside en un único *ComputingResource*, *host*, pero puede gestionar la capacidad de otros elementos. Uno de los ejemplos de aplicación de este tipo de asociación sería el de un nodo multiprocesador. En el caso de Mod-Mast, ésta relación es unívoca, un *Scheduler* gestiona la capacidad de un único *Processing_Resource*. Esta limitación deviene de las técnicas de análisis desarrolladas en la actualidad, que todavía no ofrecen la capacidad de analizar sistemas con nodos multiprocesadores. Sin embargo, la adaptación de Mod-MAST a sistemas de este tipo no resultaría muy costosa.

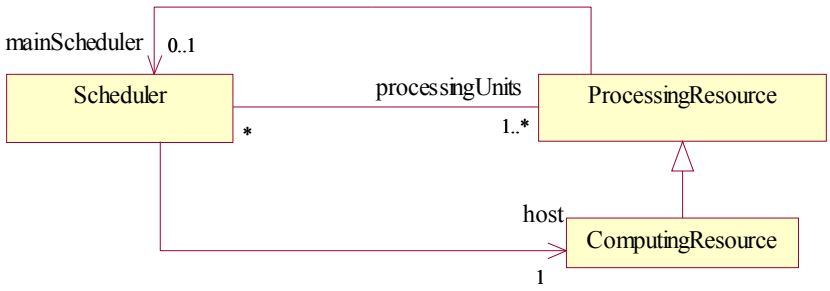


Figura 2.39: Asociación entre Scheduler y ProcessingResource en MARTE

Hemos analizado la compatibilidad entre Mod-Mast y MARTE desde el punto de vista del modelado orientado a análisis de planificabilidad, mostrando las correspondencias de los elementos de modelado de bajo nivel. Sin embargo, el aspecto esencial de nuestra metodología es la capacidad de agrupar todas estas primitivas de modelado en contenedores, que correspondan a los módulos software y hardware a través de los que se define la arquitectura y el despliegue de una aplicación basada en composición de módulos. Por ello, resulta también interesante estudiar el perfil MARTE con el objetivo de identificar correspondencias a un nivel más alto de abstracción.

En este sentido MARTE define un subperfil, denominado *Generic Component Model*, que introduce una serie de conceptos útiles para el modelado de aplicaciones basadas en componentes o módulos software reutilizables. Sin embargo, el concepto de componente se ha desarrollado en MARTE únicamente desde el nivel de abstracción más alto, esto es, como módulo software a través del que se puede definir la arquitectura de una aplicación. La relación con su correspondiente modelo de tiempo real (u orientado a análisis de planificabilidad) queda sin definir en el perfil. Cuando se quieran desarrollar modelos con dicho objetivo, será necesario definir la semántica de ejecución concreta asociada a un componente y a sus puertos, así como reglas de diseño y de mapeado, de manera que se puedan desarrollar herramientas automáticas de transformación entre modelos que permitan llegar desde el modelo de más alto nivel hasta el más concreto, requerido para aplicar análisis de planificabilidad. La estrategia propuesta en Mod-MAST solventa de manera directa ese salto entre el modelo arquitectural y el modelo de análisis, al incluir como parte de sus primitivas de modelado el concepto de modelo de tiempo real de un módulo software.

2.5. Conclusiones

La metodología Mod-MAST que se expone en este capítulo constituye una metodología modular de modelado y análisis de tiempo real, esto es, una metodología que ofrece elementos de modelado con las propiedades de componibilidad y parametrización necesarias para poder obtener el modelo de tiempo real de una aplicación construida modularmente, por composición de los modelos de los elementos que la forman.

Sus características principales son:

- Está basada en la dualidad descriptor e instancia de modelo de tiempo real. A cada módulo software o hardware reutilizable se le asocia un descriptor de modelo, esto es, una plantilla parametrizada que incluye toda la información relevante para describir el comportamiento temporal del módulo en cualquier aplicación en la que sea utilizado. Cuando una instancia del módulo se incluye en una aplicación concreta, se genera una instancia del modelo de tiempo real del módulo para ella, la cual, ensamblada con las instancias de modelo de los otros módulos de la aplicación, dan lugar al modelo de tiempo real de la aplicación. Una instancia de modelo de un módulo se genera a partir de su correspondiente descriptor, asignando a los parámetros y referencias definidos en él los valores que corresponden al contexto de aplicación modelado.
- Define el nivel de conocimiento que han de tener cada uno de los agentes que participan en el desarrollo de la aplicación respecto del proceso de modelado. El diseñador de un módulo, que conoce su estructura y los detalles de su código, es el responsable de formular el descriptor de modelo del módulo como una abstracción independiente del uso que se le de al módulo en las futuras aplicaciones en que se integre. El diseñador de un sistema de tiempo real, construye su modelo de tiempo real generando y componiendo las instancias de los modelos de los módulos que lo forman, a partir de los descriptores y sin tener que analizar la estructura interna o el código de los módulos ni de los propios descriptores. Toda la información que se necesita para la obtención del modelo de tiempo real de la aplicación se obtiene de la descripción de la arquitectura y del despliegue de la aplicación, sobre la que el diseñador de la aplicación sí tiene toda la responsabilidad.
- En la metodología Mod-MAST se define el procedimiento sistemático para generar el modelo de tiempo real de una aplicación por composición de los modelos de tiempo real de los módulos que constituyen el sistema (aplicación y plataforma). Este procedimiento genera las trazas adecuadas para que los resultados obtenidos del análisis del modelo final se puedan asignar a los parámetros que corresponden en los módulos de partida.
- La metodología Mod-MAST se define como una extensión de la metodología MAST, y se basa en una estrategia de modelado transaccional. Describe el sistema de tiempo real tanto desde el punto de vista estructural, como desde el punto de vista reactivo. Desde el punto de vista estructural define los requisitos de procesamiento de las actividades que se definen en la aplicación y la capacidad de procesamiento que proporcionan los recursos de la plataforma de ejecución. Desde el punto de vista dinámico cada situación de tiempo real que va a ser objeto de análisis se caracteriza mediante un modelo de carga, en el que se describen los eventos externos o temporizados a los que responde la aplicación, las actividades que se ejecutan en respuesta a estos eventos y las restricciones temporales en que deben ser ejecutadas.

Una de las contribuciones principales de esta propuesta es que aunque la metodología se presenta como una extensión de MAST, los conceptos en los que se basa son directamente aplicables a otras metodologías existentes compatibles con un modelo transaccional de sistema, es decir, aquellas compatibles con el subperfil referido a análisis de planificabilidad de la especificación MARTE [MARTE].

Aunque en este caso se aplique al modelado transaccional, alguna de las soluciones adoptadas, como la utilización de la estrategia descriptor/instancia, representan soluciones genéricas que podrían ser aplicadas a metodologías basadas en otro tipo de modelos. Para ello sería necesario:

- añadir capacidad de parametrización a los conceptos de modelado primitivos de la metodología,

- añadir elementos de modelado contenedores, con las correspondientes reglas para la creación de cada uno (qué tipo de elementos pueden contener, qué tipo de parámetros admiten, etc.),
- y finalmente, definir el proceso de composición, esto es, cómo se relacionan los elementos de los distintos contenedores entre sí para generar un modelo plano, sin elementos de alto nivel.

Algunas de las propuestas que tratan de aplicar modularidad y componibilidad al modelado de tiempo real [DGL08][MM08][HCM08] la consiguen suponiendo determinados modelos de computación subyacente, con una serie de restricciones que facilitan el proceso de composición. Frente a ellas, nuestra estrategia ofrece una mayor libertad, ya que proporciona elementos de modelado y reglas de composición genéricas, que pueden ser utilizadas o mapeadas a diferentes modelos de computación. En la misma línea, la propuesta que aparece en [BCW06] ofrece también una rica capacidad de modelado, con un proceso de composición perfectamente definido; sin embargo, no permite la reutilización de los modelos en cualquier contexto, pues éstos no son parametrizados, con lo que valores como las prioridades de los threads internos de los componentes, por ejemplo, no pueden ser reajustados en función del tipo de aplicación en el que el componente es utilizado.

3. Especificación de la configuración y el despliegue de aplicaciones de tiempo real basadas en componentes

Una de las principales características que diferencia el desarrollo basado en componentes de otras estrategias de diseño software es la opacidad. Durante el proceso de desarrollo de una aplicación, los componentes son manejados sin conocer los detalles de su implementación interna, por lo que en el paquete en que se distribuye un componente deben incluirse junto a los ficheros con el código, los metadatos y modelos que se necesitan para gestionar y utilizar el componente de forma opaca.

En una aplicación tradicional, con requisitos exclusivamente funcionales, los metadatos describen la funcionalidad y propiedades de componibilidad y configuración del componente, así como las características que el componente requiere de la plataforma para poder ser ejecutado en ella. En el caso de aplicaciones con requisitos de tiempo real es necesario incluir información adicional, que permita evaluar el comportamiento temporal del componente cuando es incluido en una aplicación y configurar los atributos del componente que influyen en la planificabilidad de la aplicación, de manera que se pueda asegurar el cumplimiento de sus requisitos temporales. Al igual que ocurre con los metadatos funcionales, estos nuevos metadatos y modelos son definidos por los agentes que desarrollan los componentes y utilizados posteriormente por los agentes que desarrollan las aplicaciones.

Por tanto, para definir una metodología basada en componentes de tiempo real es necesario extender los metadatos que se asocian a los componentes y aplicaciones y añadir nuevas tareas y herramientas al proceso de desarrollo de las aplicaciones. El objetivo principal de estas extensiones es permitir que el análisis y configuración de las aplicaciones se realice utilizando tan sólo la información que llevan asociada los componentes, sin modificar ni tener que conocer su código.

A fin de que la extensión de tiempo real que se propone pueda servir de base a futuros estándares sobre configuración y despliegue de componentes, se ha buscado hacerla independiente de las metodologías de modelado, de los métodos de análisis y de las estrategias de diseño de tiempo real que se puedan proponer ahora y en el futuro. Esto requiere delimitar para cada agente del proceso de desarrollo el nivel de conocimiento que necesita sobre las características de modelado, análisis y diseño de tiempo real, y definir la funcionalidad de las herramientas que dichos agentes utilizan para llevar a cabo las responsabilidades que tienen asignadas.

El éxito del desarrollo de software basado en componentes depende en gran medida del grado de estandarización de los procesos de desarrollo. Por ello, como marco de referencia para la definición de las extensiones se ha tomado la especificación "*Deployment and Configuration of Component-based Distributed Applications Specification*" (D&C) de OMG [D&C]. Su objetivo principal es el de definir los formatos de los modelos y metadatos que se utilizan para describir componentes y aplicaciones, facilitando un proceso estándar de despliegue y configuración de aplicaciones basadas en componentes en plataformas distribuidas heterogéneas. De esta

especificación se toman los conceptos básicos de componente y de aplicación desarrollada como ensamblado de componentes, así como la definición de los procesos complementarios de desarrollo de componentes y aplicaciones.

Aunque la especificación D&C sólo contempla aspectos funcionales, se la consideró adecuada como base para la definición de las extensiones de tiempo real porque el esqueleto del proceso de desarrollo básico que propone se adecua al caso de aplicaciones de tiempo real. Además, ofrece las siguientes características:

- Su formulación se basa en una especificación independiente de la plataforma, PIM (*Platform Independent Model*), que puede ser fácilmente aplicada a diferentes tecnologías de componentes.
- Está formulada mediante un metamodelo UML, lo que facilita su extensión. Todas las extensiones introducidas se han formulado como asociaciones o atributos opcionales, de manera que no se hagan incompatibles los modelos que contemplan aspectos no funcionales con aquellos que siguen el metamodelo estándar.

A lo largo del capítulo se describen los principales elementos que constituyen la extensión de tiempo real realizada, tanto a nivel del metamodelo que describe los metadatos, como de las tareas que se añaden al proceso de desarrollo con el objetivo de diseñar el sistema desde el punto de vista de tiempo real. Como punto de partida para poder explicar las extensiones definidas, en la siguiente sección se realiza una breve introducción a la especificación D&C.

3.1. Especificación de despliegue y configuración de aplicaciones distribuidas basadas en componentes de OMG

3.1.1. Objetivo y estructura

El consorcio OMG propone la especificación "*Deployment and Configuration of Component-based Distributed Applications Specification*", adoptada como especificación formal en Abril de 2006, con el objetivo principal de que la composición, configuración y despliegue de componentes procedentes de diferentes suministradores sean compatibles, y a su vez, las diferentes herramientas y entornos de desarrollo disponibles puedan interoperar entre sí.

La especificación se formula a través de dos tipos de modelos:

- Modelos de datos: Definen el conjunto de elementos de modelado que se pueden utilizar para describir componentes, plataformas y aplicaciones.
- Modelos de gestión: Definen la funcionalidad de las herramientas que se encargan de gestionar la información descrita a través de los modelos de datos durante los procesos de desarrollo de componentes y aplicaciones.

Como muestra la figura 3.1, la especificación D&C incluye los siguientes elementos:

- Los metadatos o modelos que permiten describir componentes y aplicaciones basadas en componentes. Esta información se define en el Modelo de Datos de Componentes (*Component Data Model*). La funcionalidad de las herramientas que almacenan, localizan y presentan esta información se define en el Modelo de Gestión de Componentes (*Component Management Model*).
- Los metadatos que permiten describir las plataformas distribuidas en que se van a ejecutar las aplicaciones. Su definición se incluye en el Modelo de Datos de la Plataforma

(*Target Data Model*). La funcionalidad de las interfaces que obtienen y presentan esta información al usuario se define en el Modelo de Gestión de la Plataforma (*Target Management Model*).

- Los metadatos que permiten describir el despliegue de una aplicación en una plataforma, definida en el Modelo de Datos de Ejecución (*Execution Data Model*). Las interfaces de las herramientas que ejecutan dicho proceso de despliegue se define en el Modelo de Gestión de la Ejecución (*Execution Management Model)*.
- La definición del proceso de despliegue de una aplicación basada en componentes, desde que los componentes son adquiridos hasta que la aplicación es ejecutada en una determinada plataforma.
- Una adaptación, o especialización, de todos estos modelos para el caso de aplicaciones basada en el modelo de componentes de CORBA [CCM].

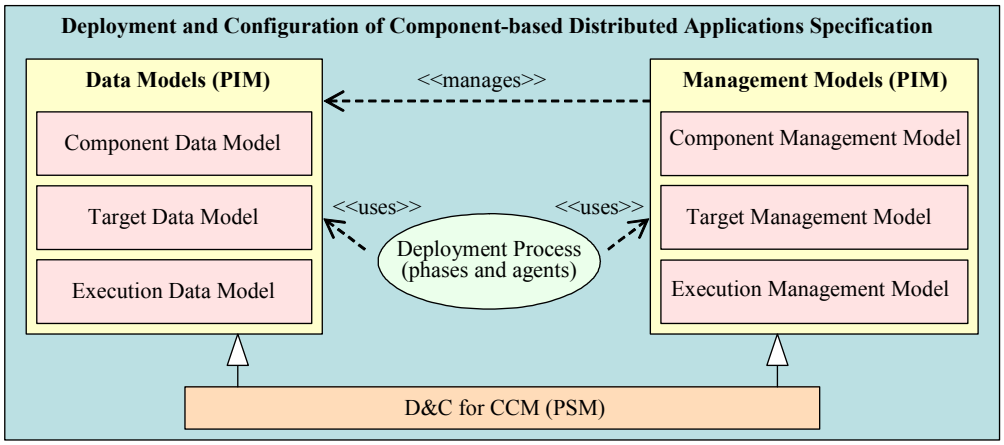


Figura 3.1: Estructura de la especificación D&C

La especificación D&C se ha formulado conforme a la especificación MDA. El núcleo de la especificación se define como un modelo PIM, dividido en los tres submodelos que se han citado: componentes, plataforma y ejecución. Todos los conceptos incluidos en estos submodelos son independientes de la tecnología de componentes, lenguaje de programación, formato de almacenamiento de información, etc., que sean empleados para el desarrollo de los componentes o de las aplicaciones. Si se quiere adaptar la especificación D&C a una tecnología de componentes específica, es necesario realizar la transformación del modelo PIM a un modelo PSM, incluyendo en él todos aquellos aspectos que sean característicos de dicha tecnología. La especificación incluye un ejemplo de dicho proceso de transformación para el caso de la tecnología CCM.

3.1.2. Concepto de componente y aplicación

La especificación D&C se basa en el concepto de componente que se define en la especificación del lenguaje UML 2.0 [UML2S]: *"Un componente representa un módulo de un sistema que encapsula su información en un elemento que puede ser reemplazado dentro del entorno para el que ha sido diseñado. Un componente tiene una estructura interna opaca y define su funcionalidad y comportamiento en función de las interfaces que provee y requiere. Su objetivo es hacer posible que grandes elementos de la funcionalidad de las aplicaciones puedan diseñarse ensamblando componentes reutilizables, disponibles en catálogos o en otras fuentes de distribución, a través de la conexión de los puertos que requieren y proporcionan".*

A efectos del proceso de configuración y despliegue, un componente es un paquete que contiene tanto metadatos como los artefactos de código que constituyen las diferentes implementaciones del componente. Los metadatos proporcionan al diseñador de una aplicación toda la información que necesita para decidir la inclusión de un componente en la aplicación, instalarlo en un nodo de la plataforma, y enlazarlo con otros componentes de los que haga uso o que hagan uso de él, siempre sin requerir acceso a su código interno.

Los componentes implementan la funcionalidad comprometida a través de las interfaces que se definen en su especificación. Un componente puede ofrecer múltiples implementaciones, en función de que pueda ser instanciado en diferentes tipos de nodo: un mismo componente puede contener implementaciones para entornos Windows, Linux, Java VM, etc. Cada una de las implementaciones incluidas en un paquete de un componente puede ser monolítica o generada por ensamblado de componentes. Este concepto de recursividad es básico en las tecnologías de componentes, y lleva a la conclusión de que una aplicación es un componente más, cuya ejecución de forma individual tiene utilidad y que generalmente está formada por ensamblado de otros componentes. Para ejecutar una aplicación basada en componentes, cada instancia de componente que la constituye debe ser instanciada, conectada y configurada en aquel nodo de la plataforma al que haya sido asignada. La especificación define todo el conjunto de datos y herramientas que son necesarios para llevar a cabo este proceso de manera automatizada.

3.1.3. Modelo de datos de componentes

El modelo de datos de componentes (*Component Data Model*) incluye toda la información que permite describir componentes, y aplicaciones basadas en componentes, junto a los requisitos que establecen a fin de ser utilizados. Una vez que un componente ha sido desarrollado, se empaqueta de acuerdo a este modelo y se hace público, de forma que el desarrollador de una aplicación en la que se requiera utilizar un componente de su tipo, lo almacene en su repositorio y pueda hacer uso de él de forma opaca.

El esquema del modelo básico que D&C define para la descripción de un componente se muestra en la figura 3.2. Un paquete de componente (*PackageConfiguration*) describe una configuración de un componente reutilizable para el diseño de una aplicación. Cada paquete puede contener varias implementaciones del componente (*ComponentImplementation Description*), bien monolíticas (*MonolithicImplementationDescription*) o descritas como ensamblado de componentes (*ComponentAssemblyDescription*). Todas estas posibles implementaciones verifican la misma interfaz externa de componente (*Component InterfaceDescription*), esto es, implementan la misma funcionalidad.

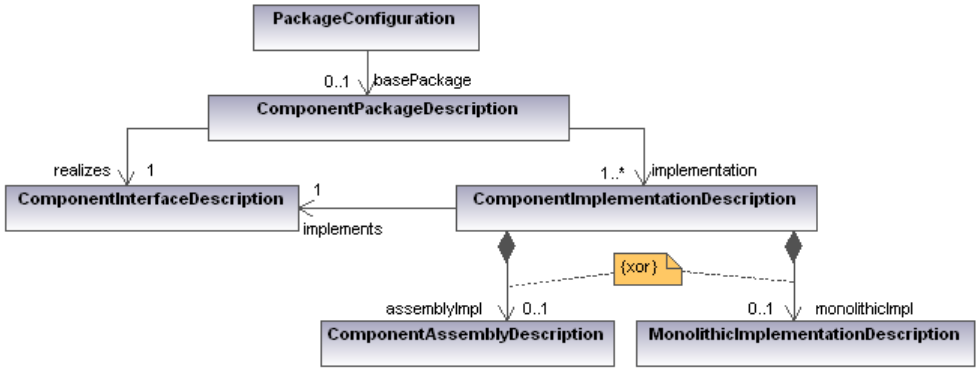


Figura 3.2: Descripción D&C de un componente software reutilizable

La descripción de la interfaz de un componente, cuya estructura se muestra en la figura 3.3, describe el componente desde el punto de vista externo. En ella se describe la funcionalidad del componente, a través de la declaración de sus puertos de conexión tanto ofertados como requeridos, cada uno descrito a través de un elemento del tipo *ComponentPortDescription*. Se incluye también la declaración de las propiedades de configuración del componente (a través de elementos de tipo *ComponentPropertyDescription*), a las que se les puede asignar valores concretos (mediante elementos de tipo *Property*) tanto a este nivel como a niveles superiores. Las propiedades que se definen en la interfaz del componente son las únicas propiedades configurables que éste ofrece, y son usadas para adaptar la funcionalidad de negocio del componente a la aplicación concreta en que es utilizado.

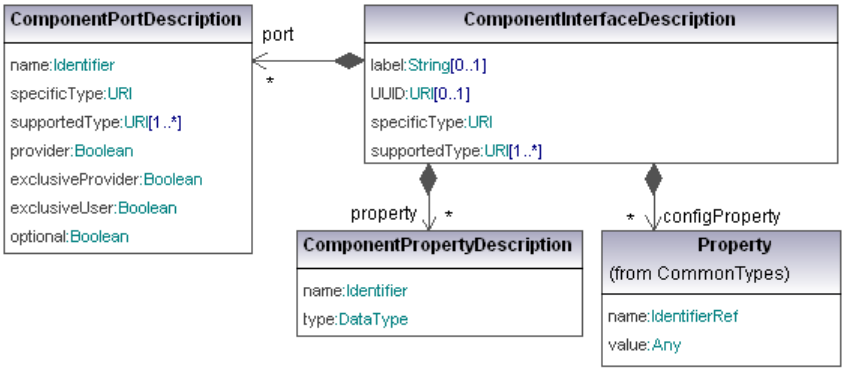


Figura 3.3: Descripción D&C de la interfaz externa de un componente

La descripción de la interfaz externa es el punto de partida en el desarrollo de un componente. Juega un doble papel:

- Constituye el patrón de comportamiento con el que se decide si un componente es útil para ser utilizado en una aplicación o si es reemplazable por otro componente.
- Constituye la plantilla respecto de la que se validan las diferentes implementaciones del componente, esto es, toda implementación debe verificar la funcionalidad expuesta a través de su interfaz.

La estructura del descriptor de una implementación de un componente se muestra en la figura 3.4. Toda implementación satisface una interfaz concreta, referenciada a través del atributo *implements*, pudiendo dar valor o sobrescribir alguna de las propiedades de configuración declaradas en dicha interfaz, a través de los atributos *configProperty.* Como ya se ha explicado, una implementación puede ser monolítica o formada por ensamblado de componentes:

- Una implementación monolítica (*MonolithicImplementationDescription*) describe los elementos de código que la forman *(ImplementationArtifactDescription)* y los requisitos que la implementación establece para poder ser ejecutada en un nodo *(Implementation Requirement)*. Dichos requisitos serán comparados con los recursos declarados por los nodos procesadores para verificar que el componente puede ser instanciado en ellos.
- La descripción de un ensamblado de componentes (*ComponentAssemblyDescription*) incluye el conjunto de instancias que forman el ensamblado (*SubcomponentInstantiation Description*), las conexiones entre sus puertos (*AssemblyConnectionDescription*) y el mapeado de propiedades desde la propiedades externas del ensamblado a las internas de las instancias que lo forman (*AssemblyPropertyMapping*).

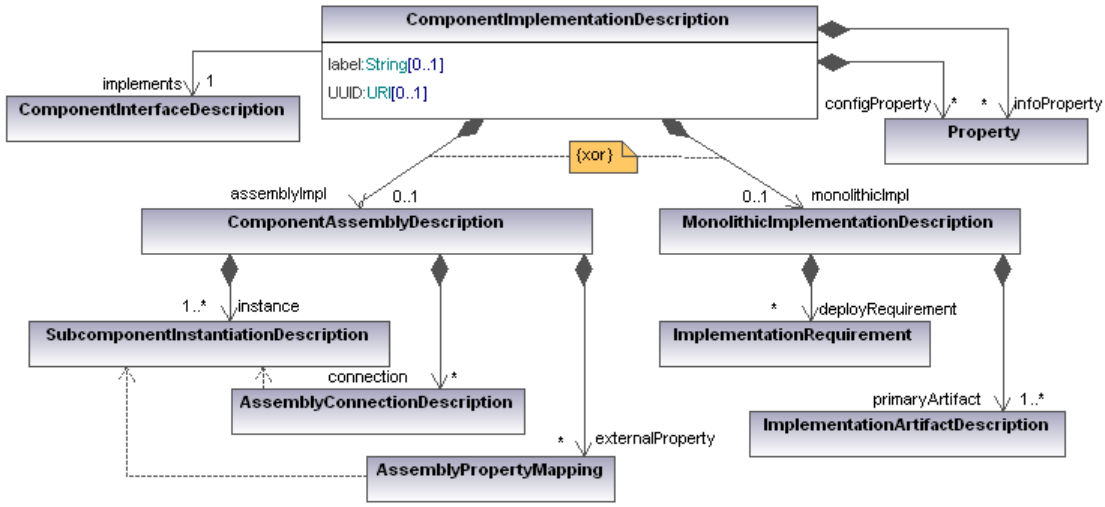


Figura 3.4: Descripción D&C de una implementación de componente

3.1.4. Modelo de datos de la plataforma

El modelo de datos de la plataforma (*Target Data Model*) define las clases y conceptos necesarios para describir las plataformas en las que van a ejecutar las aplicaciones.

El modelo de plataforma (*Domain*) que soporta la especificación D&C, como se muestra en la figura 3.5, está compuesto de nodos (*Node*), canales de comunicación (*Interconnect*), recursos compartidos (*SharedResource*) y puentes (*Bridge*). Los nodos tienen la capacidad computacional y constituyen el recurso que ejecuta el código de los componentes y con ello la aplicación. Esta definición abarca cualquier tipo de procesador ya sea computador personal o procesador embarcado. Los canales de comunicación representan las vías de comunicación directa entre nodos que permiten el intercambio de información, como puede ser una línea Ethernet o un bus CAN. Por último, un puente permite la conectividad entre diferentes canales de comunicación y representa tanto enrutadores (*routers*) como conmutadores (*switches*).

Tanto los nodos como los canales de comunicación poseen recursos (*Resources*), que constituyen el mecanismo empleado para describir sus características. Dichas características son contrastadas con los requisitos impuestos por los componentes para ser instalados en un determinado nodo. Ejemplos de recursos para un nodo son, por ejemplo, el sistema operativo, la memoria, etc. Pueden existir también recursos que se compartan entre diferentes nodos (*SharedResource*).

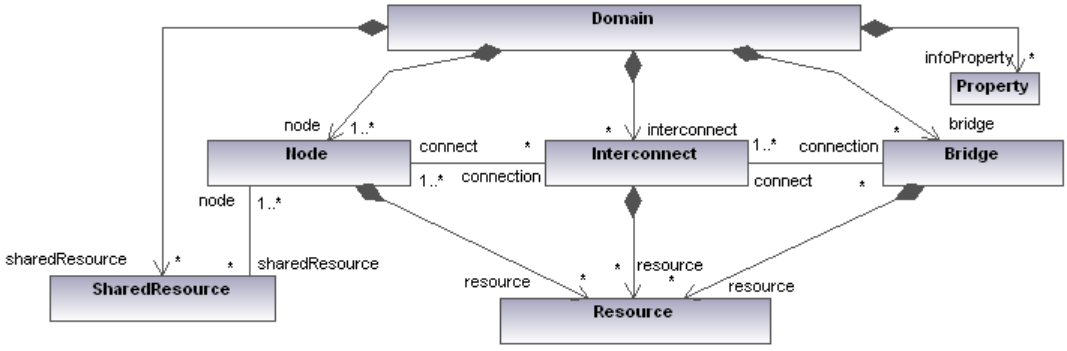


Figura 3.5: Descripción D&C de una plataforma de ejecución

3.1.5. Modelo de datos de ejecución

Cuando se define una aplicación, se decide cuáles son las implementaciones concretas de componentes que la forman, la configuración de cada uno, sus conexiones y en qué nodos de la plataforma se van a ejecutar. Toda esta información se recoge en el plan de despliegue, que sirve de entrada a las herramientas que lanzan la ejecución de la aplicación. La información del plan de despliegue se describe de acuerdo al modelo especificado en el modelo de datos de ejecución (*Execution Data Model*), cuya estructura básica se muestra en la figura 3.6.

La estructura de un plan de despliegue es muy similar a la de un ensamblado de componentes, ya que no olvidemos que una aplicación se considera como un componente más con utilidad por si mismo. Por tanto, un plan de despliegue describe:

- El conjunto de instancias de componentes que forman la aplicación *(InstanceDeploymentDescription)*, indicando para cada una de ellas el nodo en el que ha de ser instanciada (el nodo correspondiente debe estar declarado en el modelo del dominio). Cada instancia puede ser configurada mediante la asignación de valores concretos a sus propiedades, a través del campo *configProperty*.
- El conjunto de recursos, generalmente ficheros de código, que han de ser instalados en los nodos para que la herramienta de lanzamiento sea capaz de instanciar los componentes. Se describen a través de elementos del tipo *ArtifactDeployment Description*, generados a partir de la información incluida en los correspondientes descriptores de implementación de los componentes.
- Las conexiones entre puertos de componentes, descritas por medio de objetos del tipo *PlanConnectionDescription*. Cada uno identifica el par de instancias que participan en la conexión, indicando cuáles son los puertos que se conectan. Para que una conexión pueda realizarse las interfaces de los puertos conectados deben ser compatibles.
- El mapeado de las propiedades, desde las externas, esto es, las propias de la aplicación, a las propiedades internas de las instancias que la forman. Este mapeado se define a través de un objeto de tipo *PlanPropertyMapping*.

La información definida de acuerdo a esta estructura es toda la información que las herramientas de lanzamiento necesitan para ejecutar la aplicación: las instancias de componentes que se han de instanciar, los ficheros de código que se utilizan para la instanciación, los valores que se deben asignar a las variables de configuración de cada instancia y las conexiones entre instancias que deben establecerse.

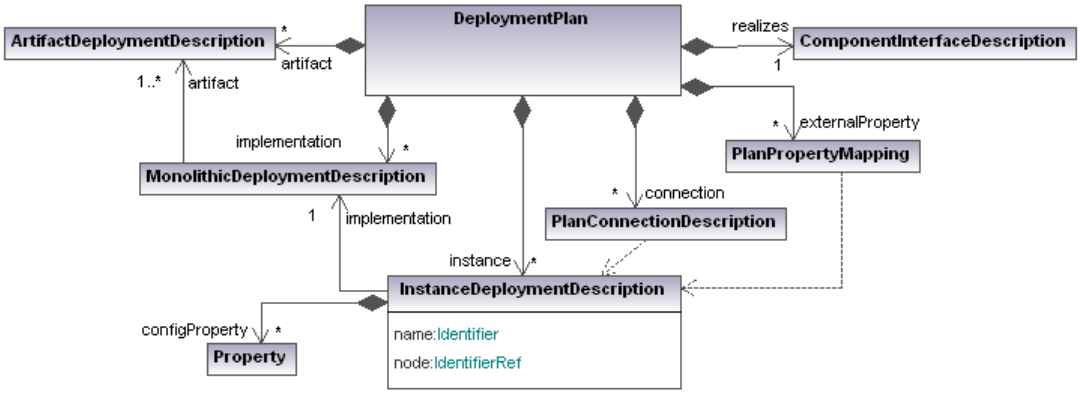


Figura 3.6: Descripción D&C de un plan de despliegue de una aplicación basada en componentes

3.1.6. Proceso de desarrollo de aplicaciones y componentes según D&C

La especificación D&C define también los agentes que intervienen en el proceso de desarrollo de aplicaciones y componentes, que son quienes elaboran y/o hacen uso de la información que se describe a través de los formatos definidos en los modelos de datos.

3.1.6.1. Proceso de desarrollo de componentes

Aunque la especificación D&C no define explícitamente el proceso de desarrollo de un componente, sí identifica a los agentes implicados y fija sus responsabilidades. Como resultado, surge el proceso que se muestra en la figura 3.7, que consta de las siguientes fases:

1. Especificación (*Specification*): El agente *Specifier* es un experto en un determinado dominio de aplicación, que descubre en él la necesidad de un nuevo componente con una determinada funcionalidad y lo describe a través de su interfaz externa. La información que genera para formular la interfaz es un elemento de tipo *Component InterfaceDescription*.
2. Implementación (*Implementation*): En esta fase se diseñan y elaboran las múltiples implementaciones de un componente (de la interfaz externa de un componente). Para cada implementación se desarrolla su código y se formula un elemento de tipo *ComponentImplementationDescription*, en el que se incluye la información necesaria para manejar el código del componente y poder instanciarlo y configurarlo de forma opaca.

   Existen dos opciones para el desarrollo de una implementación:

   - El agente *Developer* desarrolla directamente el código de negocio del componente. En este caso lo describe utilizando un elemento de tipo *MonolithicImplementation Description*.
   - El agente *Assembler* construye una implementación ensamblando componentes previamente implementados. En este caso se describe a través de un elemento de tipo *ComponentAssemblyDescription.*
3. Empaquetamiento (*Packaging*): El agente *Packager* es el experto en la tecnología de componentes que empaqueta toda la información (código, metadatos, etc.) con la que el componente debe ser distribuido de forma que pueda ser utilizado por terceros como un módulo independiente. Elabora un elemento de tipo *ComponentPackageDescription*, a través del que se referencian, y por tanto se hacen accesibles, todos los elementos que se incluyen en el paquete que constituye el componente distribuible.

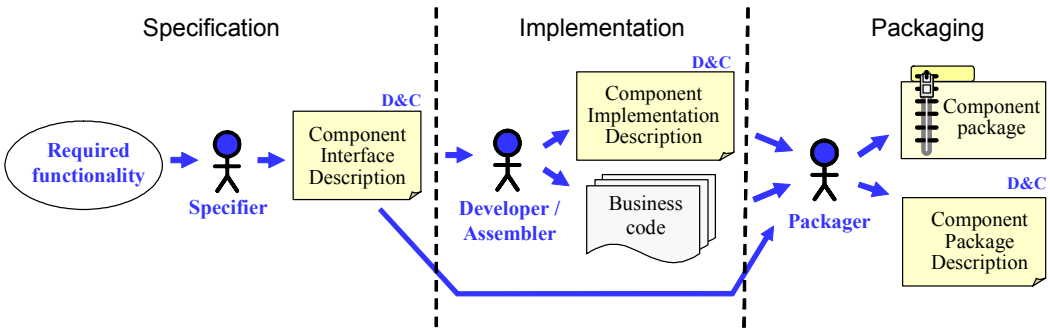


Figura 3.7: Proceso de desarrollo de componentes en D&C: agentes y productos involucrados

3.1.6.2. Proceso de desarrollo de aplicaciones

En la especificación D&C, el proceso de desarrollo de aplicaciones toma como punto de partida un conjunto de componentes que ya están implementados y que han sido empaquetados y descritos de acuerdo a la propia especificación. La especificación describe como debe formularse la aplicación, su configuración y su despliegue en una plataforma, así como los agentes que intervienen en su desarrollo. La figura 3.8 muestra la relación que existe entre los agentes, los productos generados y las etapas del proceso.

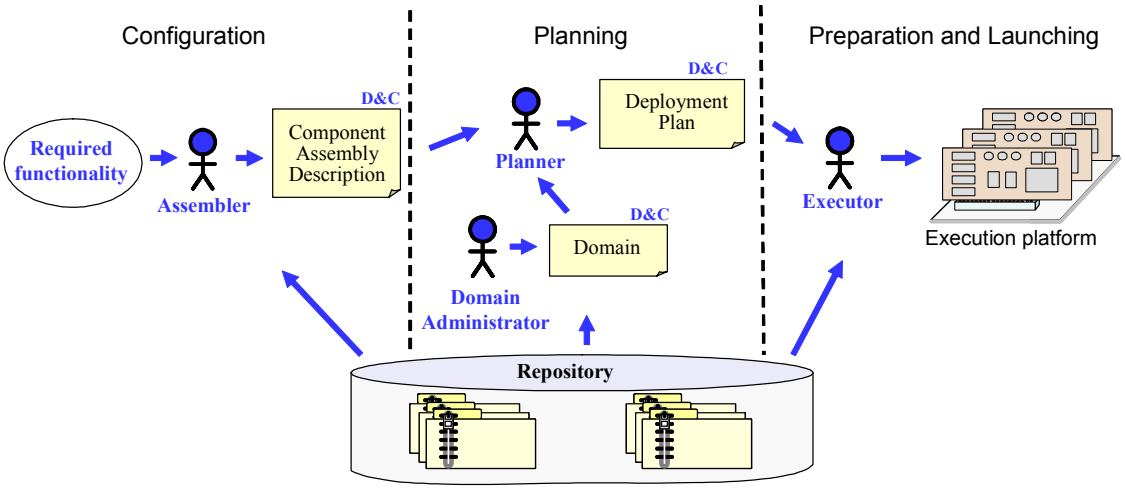


Figura 3.8: Proceso de desarrollo de aplicaciones en D&C: agentes y productos involucrados

La especificación D&C identifica una primera etapa del proceso de despliegue que no se muestra en la figura. Corresponde a la fase de instalación (*Installation*), en la que se adquieren los componentes y se almacenan en el registro o repositorio de la plataforma de diseño, donde son accesibles para los diseñadores de la aplicación de forma normalizada. A partir de ese punto, el proceso de despliegue de una aplicación conlleva cuatro fases:

1. Configuración (*Configuration*): En base a la funcionalidad que quiere obtenerse con la aplicación, el *assembler* la diseña y la describe como un ensamblado de instancias de componente configuradas, elegidas de entre las disponibles en el entorno de desarrollo. La especificación D&C establece que la aplicación se formula a través de un *ComponentAssemblyDescription*.
2. Planificación (*Planning*): En base al conocimiento que se tiene de la plataforma en la que se va a ejecutar la aplicación, el agente *planner* decide cómo y dónde se van a ejecutar los elementos de la aplicación en el entorno de ejecución. El *planner* decide las implementaciones concretas que se utilizan para cada instancia de componente y los nodos de procesamiento en los que se van a instalar y ejecutar. Para ello debe ponderar los requerimientos de las implementaciones de los componentes y los recursos que tienen declarados los elementos de la plataforma. El resultado de esta fase es el plan de despliegue, formulado mediante un elemento de tipo *DeploymentPlan*. El modelo de la plataforma necesario para llevar a cabo este proceso, formulado a través de un elemento de tipo *Domain*, es elaborado por el agente *Domain Administrator*.
3. Preparación (*Preparation*): Se compone de las tareas de distribución o de verificación que aseguran que en la plataforma de ejecución están disponibles los elementos que se necesitan para ejecutar la aplicación.

4. Lanzamiento (*Launching*): Engloba las tareas que deben realizarse para ejecutar la aplicación en la plataforma. Esto supone instanciar el código de los componentes en los nodos asignados, establecer los enlaces entre ellos, reservar los recursos que se requieren en los nodos y en las redes de comunicación e iniciar la ejecución. Esta fase junto a la anterior las lleva a cabo el agente *Executor*.

3.2. RT-D&C: Extensión de tiempo real de la especificación D&C de OMG

RT-D&C es la extensión de tiempo real de la especificación D&C que se presenta en este capítulo, que tiene por objetivo definir los nuevos metadatos y/o modelos que deben ser incluidos en el paquete de un componente para que los diseñadores de las aplicaciones que hacen uso de él puedan:

- Construir un modelo de tiempo real de la aplicación, que permita predecir su comportamiento temporal y analizar la planificabilidad de sus actividades dentro de los plazos de tiempo requeridos en la especificación de la aplicación.
- Obtener de dicho análisis los valores a asignar a los parámetros de configuración de los componentes y de la plataforma que controlan la planificabilidad de las aplicaciones, de modo que aseguren el cumplimiento de los requisitos temporales impuestos en la especificación de la aplicación.

La extensión se formula considerando que las aplicaciones de tiempo real se conciben, especifican, analizan y diseñan utilizando el paradigma reactivo o transaccional, que fue descrito en la sección 1.1.2. Siguiendo este paradigma, las aplicaciones se especifican a través de la descripción de las actividades que han de ejecutar en respuesta a los eventos del entorno o temporizados que gestionan, las cuales sirven también de base para definir los requisitos temporales que se deben satisfacer en su ejecución.

El metamodelo RT-D&C representa estrictamente una extensión del metamodelo que describe D&C para aplicaciones con especificación únicamente funcional. La información adicional que se introduce para describir los aspectos relativos a tiempo real se organiza de forma totalmente compatible con la definida en D&C. Asimismo, el proceso de desarrollo de aplicaciones de tiempo real toma como referencia el proceso especificado en D&C, al que se han añadido nuevas fases en las que, de forma complementaria, se abordan los aspectos de diseño específicos de tiempo real. Para ello, en el metamodelo RT-D&C, la información adicional se ha introducido en los elementos adecuados, de forma que cada agente que interviene en el proceso de desarrollo, encuentre la información que necesita para llevar a cabo las tareas que tiene encomendadas. En la figura 3.9 se resumen los nuevos elementos de información y las nuevas tareas que aparecen cuando la aplicación desarrollada tiene requisitos de tiempo real:

- El agente *assembler* diseña la aplicación como un ensamblado de componentes, para lo cual ha de considerar su especificación completa, esto es, tener en cuenta tanto la funcionalidad como los requisitos temporales. El *assembler* dispone sólo de la información que se incluye en la descripción de la interfaz externa de los componentes. Al estar la aplicación formulada de acuerdo a un modelo reactivo, el *assembler* elegirá los componentes principales que van a formar parte de ella por su capacidad de respuesta a eventos, buscando aquellos componentes que implementen las respuestas que constituyen la especificación de la aplicación.

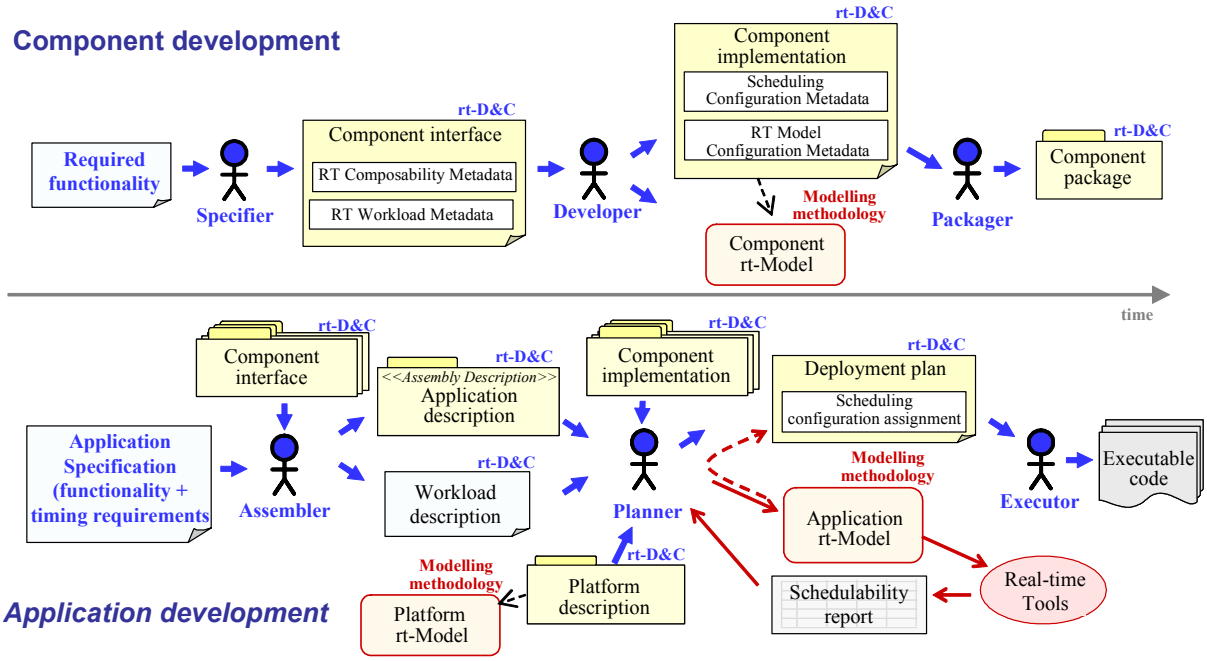


Figura 3.9: Extensión de los procesos de desarrollo D&C con aspectos de tiempo real

Con este objetivo, se incluyen en la interfaz de cada componente los metadatos a través de los que el *assembler* conoce las transacciones que se pueden iniciar en el componente (*RT Workload Metadata)*. Basándose en esa información, deberá formular el modelo de carga de trabajo de la aplicación, declarando las transacciones que se inician en cada instancia de componente y caracterizándolas de acuerdo al contexto de ejecución en que van a ser ejecutadas. Este modelo describe la situación de tiempo real para la que se va a verificar el cumplimiento de los requisitos temporales.

Además, el *assembler* debe garantizar que la aplicación diseñada ofrece comportamiento temporal predecible, por lo que la interfaz de cada componente incluye también metadatos de tiempo real que describen las características de componibilidad de los modelos de tiempo real de los componentes (*RT Composability Metadata)*. En base a ellas se conoce si a partir de un conjunto de componentes se puede generar el modelo de tiempo real de su ensamblado.

La figura 3.9 muestra como la responsabilidad de incluir estos dos nuevos tipos de metadatos en la descripción de la interfaz del componente es del *specifier*.

- El agente *planner* es el responsable de establecer la configuración que hace planificable a la aplicación. Además de asignar el procesador en el que se va a ejecutar cada instancia y elegir las implementaciones apropiadas, debe evaluar e incorporar al plan de despliegue los valores de los parámetros de planificación que se asignan a las instancias y a la plataforma para que durante la ejecución se satisfagan los requisitos temporales. Para tomar sus decisiones, el *planner* se basa en los metadatos incluidos en las implementaciones de los componentes. Es responsabilidad del *developer* incluir en los descriptores de cada implementación aquellas características del código que influyen en la planificabilidad de las aplicaciones (*Scheduling Configuration Metadata*), de manera que el *planner* pueda conocerlas y deducir el valor que se le debe asignar para que la aplicación sea planificable.

  El proceso de configuración de la planificación es complejo, por lo que el *planner* debe basarlo en la construcción de un modelo de comportamiento temporal de la aplicación,

que sea procesado con herramientas de análisis y diseño de tiempo real. El conjunto de valores de parámetros de planificación que se deben asignar es obtenido por estas herramientas. El modelo se construye por composición de:

- Los modelos de tiempo real de cada instancia de componente, que habrán sido proporcionados como parte de la información incluida en la distribución del componente.
- La descripción de la carga de trabajo de la aplicación, que ha elaborado el *assembler* en el proceso de diseño de la aplicación.
- Los modelos de tiempo real de los elementos de la plataforma, que deben encontrarse disponibles en el repositorio del entorno de diseño. El agente *Domain Administrator* que se encarga de elaborar el modelo de la plataforma, deberá encargarse también de desarrollar modelos de cada uno de los elementos que formen dicha plataforma.

Para poder obtener este modelo de tiempo real de la aplicación, los modelos reutilizables de componentes y elementos de la plataforma deberán ser formulados utilizando metodologías con las características de componibilidad adecuadas, como la expuesta en el capítulo 2. Los modelos pueden ser configurables, para poder ser adaptados a las diferentes situaciones en que sean utilizados. Es responsabilidad del *developer* incluir los metadatos que permiten al *planner* adaptar los modelos de los componentes a la aplicación concreta (*RT Model Configuration Metadata*). El *planner* maneja estos metadatos y las herramientas de composición y análisis de los modelos, e interpreta los resultados que se generan. Sin embargo, no necesita ser experto en la metodología utilizada para formular los modelos de tiempo real ni en los algoritmos de composición y análisis.

- Finalmente, el *executor*, antes de lanzar la ejecución de la aplicación debe configurar los elementos de la plataforma (nodos, redes de comunicación y recursos compartidos) a fin de que los recursos que se necesitan para ejecutar la aplicación estén disponibles. Para ello hace uso de la información que se ha incluido en el plan de despliegue.

El resto de la sección explica en detalle las diferentes extensiones que se han introducido en los elementos de modelado D&C para dar soporte a estos nuevos metadatos, y como éstos son manejados a lo largo de las diferentes etapas de los procesos de desarrollo.

3.2.1. Descripción de la especificación de componentes de tiempo real

La respuesta temporal de un componente no es una característica propia del componente por sí, depende de la propia implementación, de la plataforma en que se ejecuta, del comportamiento temporal de otros componentes de los que requiere servicios, y de otras aplicaciones que se ejecutan concurrentemente en la misma plataforma y compiten por sus servicios. Por ello, el modelo de tiempo real de un componente no describe el comportamiento temporal final de sus servicios, sino que proporciona toda la información que se puede deducir de su código y que puede ser utilizada para deducir el comportamiento temporal de las aplicaciones que hacen uso de él. A nivel de la especificación (interfaz externa) del componente no se conoce ni siquiera su código, por lo que la información sobre comportamiento temporal que se incluye en ella es la que necesita el *assembler* para conocer si con el uso del componente resulta una aplicación de la que se puede obtener un modelo de tiempo real analizable.

En la figura 3.10 se muestran los metadatos que incluye la interfaz externa de un componente e interpreta el *assembler* para diseñar una aplicación. En primer lugar, incluye los metadatos que describen la componibilidad de tiempo real del componente (*RT Composability Metadata*).

Proporcionan información sobre la compatibilidad de la conexión de dos componentes a fin de construir el modelo de tiempo real de la aplicación que resulta de su composición, o lo que es lo mismo, sobre la componibilidad de sus modelos de tiempo real. Esta información consiste en:

- Por cada puerto ofertado por el componente se declaran los servicios para los que las implementaciones disponen de modelos de comportamiento temporal, y que por tanto, pueden ser referenciados desde los modelos de tiempo real de los componentes clientes que los usen.
- Por cada puerto requerido por el componente se declaran los modelos de tiempo real que tienen que estar definidos en los componentes que se conecten al puerto, como consecuencia de que son referenciados desde el propio modelo del componente.

Con esta estrategia, dos componentes se pueden conectar desde el punto de vista de la predictibilidad de su comportamiento si por cada puerto requerido por el componente cliente, el componente servidor dispone de los modelos de comportamiento temporal de los servicios que el cliente requiere en su especificación.

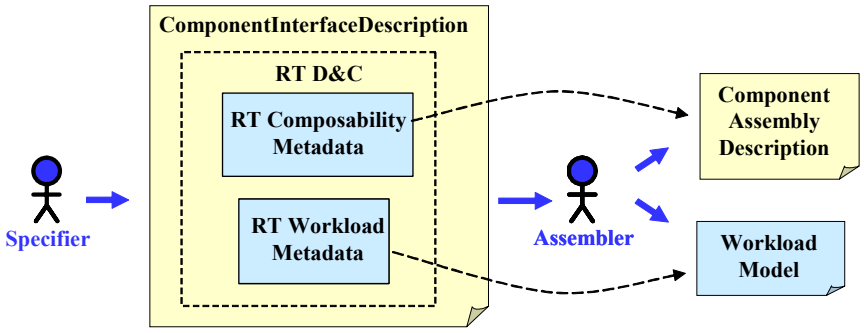


Figura 3.10: Metadatos añadidos a la interfaz externa de un componente en RT-D&C

La descripción de la interfaz de los componentes ha de incluir también la información necesaria para que el *assembler* formule la carga de trabajo de la aplicación en cada modo en que ésta puede operar. Esta carga de trabajo se modela como el conjunto de transacciones que son ejecutadas concurrentemente en dicho modo de operación. En el caso de aplicaciones basadas en componentes la carga de trabajo está formada por el conjunto de transacciones que se inician en las instancias de aquellos componentes que denominamos activos, esto es, componentes que tienen la capacidad de responder a eventos externos o temporizados desencadenando secuencias de actividades que se ejecutan en el sistema. Como ya se explicó en el capítulo anterior, las transacciones manejadas o iniciadas por un módulo activo, en este caso un componente, se clasifican en dos grupos:

- Transacciones que son inherentes a la naturaleza de ciertos componentes, esto es, que se ejecutan automáticamente una vez que el componente es instanciado en una aplicación. El modelo de estas transacciones será interno al modelo del tiempo real del componente, y por tanto, totalmente oculto al diseñador de la aplicación.
- Transacciones que dependen del modo de operación de la aplicación. Algunas de las transacciones que pueden ser lanzadas en una aplicación por un componente dependen de la configuración de la aplicación o de los datos de entrada. El *assembler*, en función de la configuración de la aplicación, deberá definir el conjunto concreto de transacciones de este tipo que se ejecutan en cada contexto, basándose para ello únicamente en la información que se aporta en la descripción externa de los componentes.

Por tanto, el segundo aspecto de tiempo real que se añade a la descripción de la interfaz externa de un componente es la declaración de las transacciones que pueden tener su origen en las instancias del componente (*RT Workload Metadata*). Como muestra la figura 3.10, estos metadatos serán utilizados por el *assembler* para definir las transacciones con que contribuye cada instancia de componente a la carga de trabajo final. La declaración de una transacción al nivel de la interfaz externa del componente incluye sólo aquellas características que permiten identificarla y caracterizarla de forma opaca, esto es, no se describe su comportamiento interno, que depende de cómo se implemente el componente. Esta declaración contiene:

- El nombre de la transacción, que sirve de referencia para la descripción del modelo completo de la transacción que se incluirá en el modelo de tiempo real del componente.
- La declaración de los parámetros que tiene definida la transacción, a través de los que el *assembler* asigna las características de cada instancia concreta de la transacción que se declare en una carga de trabajo.
- Información destinada al *assembler* para que conozca la naturaleza de la transacción y de cada uno de los parámetros que tiene definidos.

Cualquier implementación que verifique una determinada interfaz de componente deberá incluir un modelo de tiempo real que contenga todos los elementos declarados en la interfaz, de forma que todas ellas sean intercambiables desde el punto de vista de sus modelos. Si todos los componentes que forman la aplicación ofrecen modelos de tiempo real elaborados de acuerdo a esta estrategia, se podrá generar un modelo analizable de la aplicación por composición de los modelos de los componentes que la forman.

3.2.1.1. Implementación formal de la extensión en RT-D&C

En la figura 3.11[1] se muestran los elementos que se añaden al elemento *ComponentInterfaceDescription* a fin de incorporar los aspectos anteriormente expuestos:

- A través del campo *rtWorkloadEntity* se describen las transacciones de negocio que se pueden iniciar en el componente. Cada transacción se declara a través de un elemento de tipo *RTEndToEndFlowDescription*.
- A través del campo *rtUsageEntity* se describen posibles formas de uso que se puedan declarar en el componente. Corresponden a subtransacciones o formas de uso

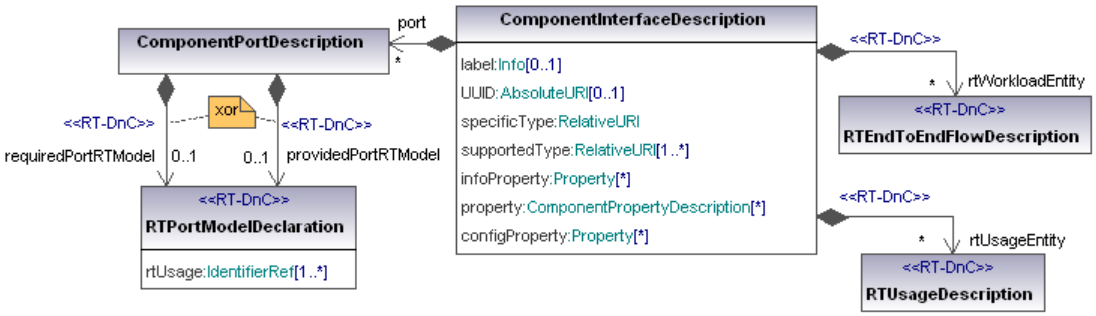


Figura 3.11: Extensión RT-D&C del descriptor de la interfaz de un componente

1. Las figuras que se usan a lo largo del capítulo se centran en mostrar las extensiones añadidas, por lo que en general no se muestra la estructura estándar completa de los elementos de modelado. El estereotipo <<RT-DnC>> se utiliza para identificar las nuevas clases o propiedades (tanto asociaciones como atributos) que se añaden en RT-D&C.

parametrizadas que se van a incluir en el modelo de una transacción completa, cuyo flujo de control interno depende del modo en que el componente sea usado dentro de la aplicación. Se definen a este nivel sus plantillas, que serán usadas como base para definir las formas de uso concretas a introducir en el modelo de cada transacción. Un ejemplo de utilización de este tipo de elemento se explica más adelante.

La descripción de los puertos que un componente ofrece o requiere se ha extendido también con dos nuevos campos, excluyentes entre sí, que se emplean para definir los requisitos de componibilidad de tiempo real del componente:

- Si se trata de un puerto ofrecido, se utiliza el campo *providedPortRTModel*, que declara los elementos de modelado que se ofrecen a través del puerto para ser compuestos con los modelos de los componentes cliente que hagan uso de él. Consiste en la enumeración de las operaciones para las que el modelo del componente incluirá modelo de tiempo real.
- Si se trata de un puerto requerido, se utiliza el campo *requiredPortRTModel*, que declara el conjunto de servicios para los que el componente servidor debe ofrecer modelos de tiempo real.

Para dar soporte a la extensión ha sido necesario definir nuevos tipos de datos y clases. La clase *RTEndToEndFlowDescription* se introduce para declarar aquellas transacciones que puede iniciar el componente y que dependen del modo de operación de la aplicación. Como muestra la figura 3.12, la declaración de una transacción incluye su nombre, un comentario informativo que la describe y el conjunto de propiedades configurables que ofrece. El modelo de la transacción que se incluya como parte del modelo de tiempo real de cualquier implementación del componente deberá incluir los parámetros que aquí se definen como configurables, de manera que sean asignados cuando se incluya una instancia de la transacción como parte de una carga de trabajo. Para declarar estas propiedades se utiliza la clase *RTEndToEndFlow PropertyDescription*. Toda propiedad de tiempo real se define a través su nombre, una etiqueta informativa y su tipo. El tipo de datos *RTEndToEndFlowDataType* se utiliza como método para restringir el tipo de parámetros que se pueden definir en una transacción. En la figura se muestran sólo algunos de ellos. La naturaleza de estos tipos se ha de declarar de forma independiente a la metodología de modelado y análisis que se emplee para elaborar los modelos de tiempo real de los componentes, por ello es de interés definirla con una semántica lo más universal posible. En nuestro caso se ha intentando utilizar en todo lo posible los términos y conceptos definidos en el estándar MARTE, en concreto en el subperfil SAM.

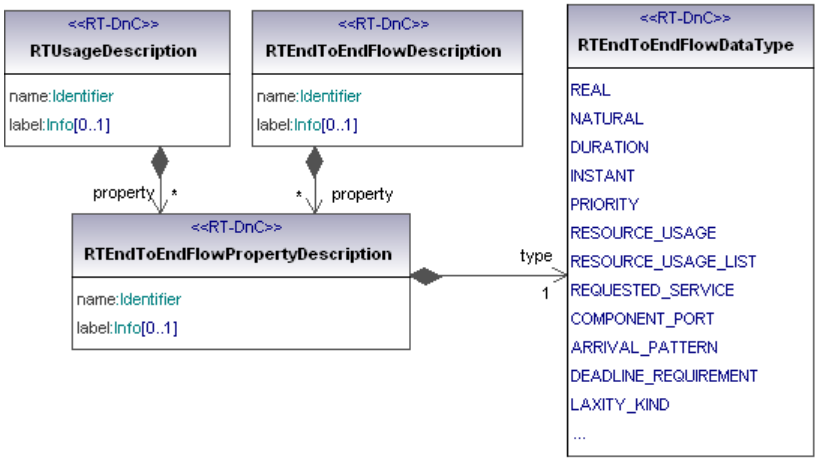


Figura 3.12: Declaración de una transacción en la interfaz externa de un componente

3.2.1.2. Ejemplo de aplicación

Como ejemplo de las extensiones introducidas en la descripción de la interfaz externa de un componente, se muestran en la figura 3.13 algunos aspectos de la interfaz externa correspondiente al componente *ScadaEngine*, que forma parte de la aplicación de ejemplo utilizada en el capítulo 6. Para lograr una mayor expresividad y legibilidad, los descriptores RT-D&C que se utilizan como ejemplo en este capítulo se representan en forma de diagramas de objetos UML, aunque como se explicará más adelante y se verá en el capítulo 6, los descriptores RT-D&C se van a formular utilizando tecnología XML.

El componente *ScadaEngine* (ver página 191) implementa a través de su único puerto ofertado, *controlPort*, una funcionalidad SCADA (*Scada Supervisory and Data Acquisition*), supervisando una serie de magnitudes físicas y almacenando valores estadísticos sobre ellas. Para implementar su funcionalidad requiere servicios de otros componentes, a los que accede a través de los puertos requeridos *adqPort* y *logPort*. La interfaz del componente incluye la descripción de estos puertos, que debe incluir sus requisitos de componibilidad de tiempo real:

- En la descripción del puerto *controlPort* se enumeran los servicios para los que se ofrece modelo de tiempo real, en este caso *getBufferedData* y *getLastLoggedMssg*.
- En la descripción de cada puerto requerido se enumeran los servicios de los que se requiere que el componente conectado ofrezca modelo temporal. En el fragmento de modelo que se muestra solo aparecen detallados los requisitos asociados al puerto *adqPort*. En este caso sólo se requiere el modelo del servicio *aiReadCode*, que es el único que el componente *ScadaEngine* utiliza como consecuencia de la funcionalidad que implementa.

El descriptor de interfaz del componente *ScadaEngine* incluye también las propiedades de configuración de negocio del componente, *samplingPeriod* y *loggingPeriod*.

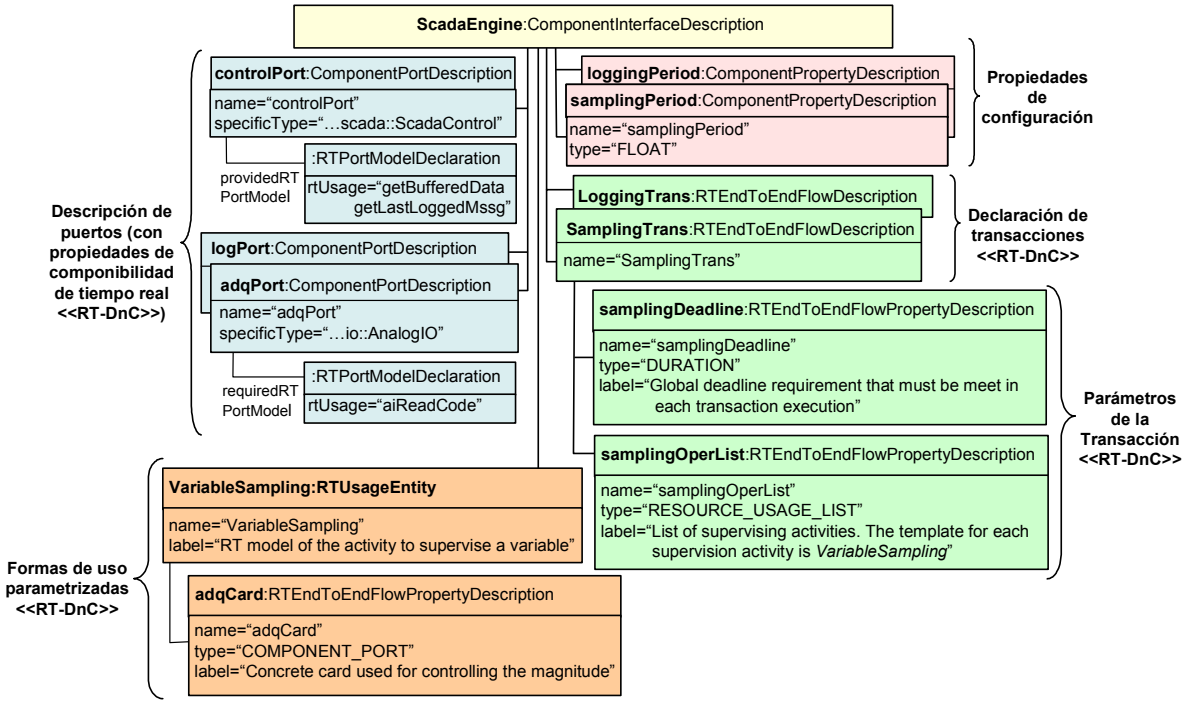


Figura 3.13: Ejemplo de descriptor de interfaz con extensiones de tiempo real

El número de magnitudes que el componente supervisa varía en función del contexto concreto de aplicación. A mayor número de magnitudes a supervisar, mayor carga de trabajo se introduce en el sistema. La actividad periódica (de periodo *samplingPeriod*) que se encarga de la supervisión se modela como una transacción parametrizada, que debe ser declarada en la interfaz del componente para que posteriormente el *assembler*, una vez conocido el número de magnitudes a supervisar, incluya la correspondiente instancia configurada de la transacción en la carga de trabajo. La transacción se denomina *SamplingTrans* y posee dos parámetros, como se muestra en la figura:

- *samplingDeadline*: Representa el plazo con el que se requiere que se ejecute cada proceso completo de supervisión. El periodo de ejecución de la transacción es fijo, *samplingPeriod*, pero de este modo podemos hacer el plazo configurable y así, ajustarlo en función de la situación de carga de la plataforma.
- *samplingOperList:* Este parámetro es el que nos permite adaptar la transacción al número de magnitudes que se supervisan. Por cada una, deberá añadirse a la transacción una actividad de supervisión, cuya plantilla *VariableSampling* se declara también a nivel de la interfaz. Esta plantilla es a su vez parametrizada, siendo necesario identificar para cada instancia que se declare, el componente concreto a través del que se accede a la magnitud de entre aquellos que se encuentran conectados al puerto múltiple *adqPort*. Este parámetro es necesario porque diferentes implementaciones del componente servidor conectado pueden tener diferentes modelos temporales para la misma función. Cuando en una determinada aplicación se conozca el número concreto de magnitudes a controlar, el *assembler* se encargará de instanciar una forma de uso del tipo *VariableSampling* por cada uno, y asignar la lista generada al parámetro *samplingOperList* de la transacción (más adelante se explica cómo se realiza esta asignación). Las herramientas de composición de modelos serán las encargadas de generar el modelo final de la transacción encadenando cada una de las actividades, siempre de forma oculta al *assembler*, que únicamente debe encargarse de la declaración de la instancia mediante la asignación de valores a sus correspondientes parámetros.

3.2.2. Descripción de la implementación de componentes de tiempo real

El principal elemento de extensión que se añade a nivel de la descripción D&C de una implementación de un componente es la inclusión de su modelo de tiempo real. Cuando se desarrolla una implementación concreta de una interfaz de componente, el *developer* tiene pleno conocimiento de su código interno, de forma que junto con ella elabora el modelo reutilizable que describe todas aquellas características acerca de su comportamiento temporal que se necesitan conocer para evaluar el comportamiento de una aplicación en la que se use el componente. Como se muestra en la figura 3.14, dicho modelo es referenciado desde el descriptor de la implementación (elemento *ComponentImplementationDescription*).

Los modelos de las implementaciones de un componente deben ser consistentes con la interfaz externa del componente:

- Debe incluir modelos concretos para todos aquellos elementos que se hayan declarado como referenciables desde el exterior, esto es, para todos los servicios enumerados a través del campo *providedRTPortModel* de los puertos ofertados.
- Solo podrá hacer referencia a aquellos elementos de modelado externos (servicios) que se definen como requeridos a través del campo *requiredRTPortModel* de los puertos requeridos.

- Debe incluir modelos concretos para las transacciones de negocio declaradas a nivel de la interfaz. En los modelos de tiempo real de dichas transacciones solo se podrán incluir como parámetros configurables aquellos que se hayan declarado en la descripción externa del componente.

El modelo de tiempo real del componente no describe su comportamiento temporal final, ya que éste depende de la aplicación concreta en que sea empleado. El modelo que se proporciona con un componente, elaborado de acuerdo a alguna metodología de modelado concreta, es un modelo parametrizado, que deja sin resolver todos aquellas características que dependen del contexto en que sea usado el componente. El descriptor de una implementación de componente incluye metadatos que permiten adaptar el modelo del componente a cada contexto concreto de aplicación sin necesidad de acceder a él. Como se ve en la figura 3.14, estos metadatos declaran los parámetros configurables del modelo de tiempo real de la implementación (*RT Model Configuration Metadata*). Los valores de dichos parámetros serán asignados cuando se defina una instancia del componente dentro de una aplicación en función del modo en que éste sea utilizado. La diferencia entre las propiedades de configuración de negocio definidas en la especificación D&C y las propiedades de configuración de tiempo real que se proponen en la extensión RT-D&C, es que las primeras son relativas a la funcionalidad del componente y afectan únicamente al código de negocio, mientras que las segundas afectan al comportamiento temporal del componente. Como veremos más adelante, en muchos casos ambos tipos de propiedades están relacionadas, pues las propiedades de configuración del negocio del componente pueden afectar también a su comportamiento temporal.

Existen parámetros del modelo de tiempo real del componente que no necesitan ser declarados de forma explícita, pues pueden ser deducidos del propio despliegue de la aplicación. Entre ellos se encuentran el parámetro que referencia al procesador en el que se instancia cada componente, o las referencias a los componentes servidores sobre los que se realizan invocaciones. La resolución de dichos parámetros la realiza automáticamente la herramienta de composición de modelos en base al plan de despliegue de la aplicación.

Una modificación relevante que se ha introducido en la extensión RT-D&C a nivel de la descripción de implementaciones es la capacidad de definir propiedades específicas de implementación. Éstas son propiedades relacionadas con el modo en que el código de un componente hace uso de los recursos de la plataforma, pero que no afectan al negocio o funcionalidad externa del componente. En la mayoría de los casos estas nuevas propiedades tienen que ver con aspectos relacionados con la planificación (*Scheduling Configuration Metadata*), como puede ser la concurrencia o la sincronización. Ejemplos de parámetros de este

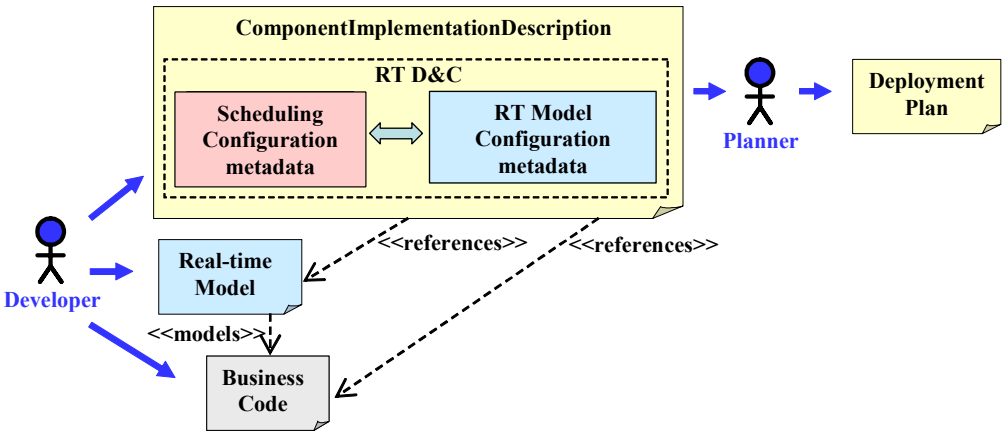


Figura 3.14: Metadatos añadidos a la implementación de un componente

tipo pueden ser prioridades de tareas, techos de prioridad de recursos compartidos, etc. Se definen a nivel de implementación ya que el número y el tipo de estas propiedades son función de las propias decisiones de diseño interno de cada implementación.

Por esa misma razón, estas nuevas propiedades son asignadas por el *planner*, que es el agente encargado de elegir las implementaciones concretas de componentes que se van a emplear en la aplicación y por lo tanto tiene acceso a dicha información. Sin embargo, en el caso de propiedades relacionadas con la planificación, aunque el *planner* pueda asignarles un valor por defecto en el plan de despliegue, los valores óptimos a asignar pueden ser en muchos casos calculados automáticamente por herramientas de análisis de tiempo real en base al modelo de la aplicación. En ese caso, deberán existir mecanismos que permitan reasignar dichos resultados obtenidos del análisis a las propiedades de configuración correspondientes, para que en efecto la ejecución de la aplicación satisfaga los requisitos que tiene especificados.

3.2.2.1. Implementación formal de la extensión en RT-D&C

Como se muestra en la figura 3.15, en la descripción RT-D&C de una implementación de componente se incluyen nuevos elementos que describen los aspectos relacionados con el comportamiento de tiempo real de una implementación:

- El atributo *rtModel* referencia el modelo de tiempo real que describe el comportamiento temporal de la implementación del componente. Es utilizado por las herramientas que construyen el modelo de tiempo real de las aplicaciones en las que se hace uso del componente. Este atributo se introduce a nivel del elemento *MonolithicImplementation Description*, de modo que cuando se describa una implementación por ensamblado de instancias de componentes, el modelo final se podrá generar por composición de los modelos de las correspondientes instancias monolíticas que la forman.
- La asociación *rtProperty* representa la lista de las propiedades configurables del modelo de tiempo real de la implementación del componente. La definición de estas propiedades se realiza a través de objetos de la nueva clase *RTComponentPropertyDescription*.
- La asociación *rtConfigProperty* es una lista de valores por defecto que se asignan a las propiedades que han sido declaradas a través del campo *rtProperty*. Para realizar esta asignación se utilizan objetos de la nueva clase *RTProperty*.
- La nueva asociación *specificProperty* representa la lista de las nuevas propiedades de configuración, aquellas que son propias de la implementación y que generalmente están relacionadas con la planificación.

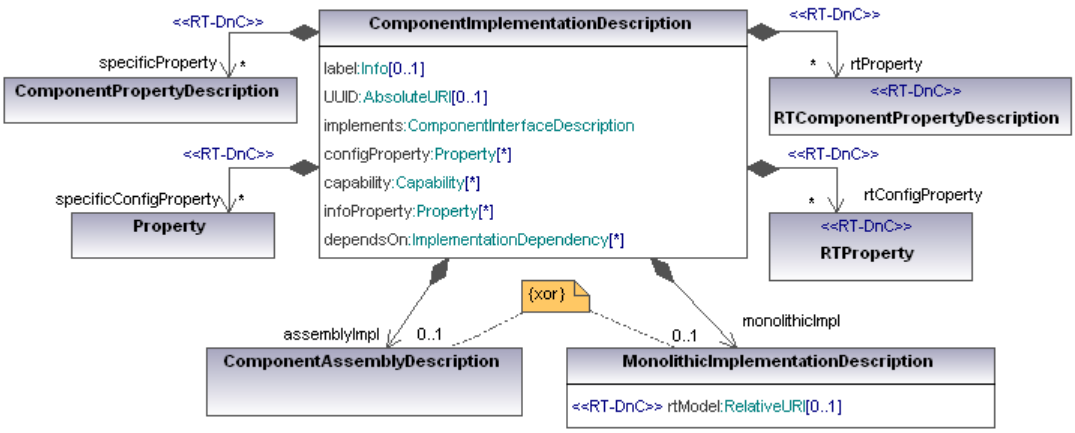


Figura 3.15: Extensión RT-D&C del descriptor de una implementación de componente

- La asociación *specificConfigProperty* permite asignar valor por defecto a las propiedades declaradas previamente en el campo *specificProperty.*

Para definir propiedades de tiempo real configurables se utiliza la clase *RTComponentPropertyDescription.* Al igual que ocurre con las propiedades de configuración funcionales que se definen en la especificación, toda propiedad de tiempo real se define a través un nombre, una etiqueta explicativa y un tipo. Los tipos de datos que pueden ser utilizados como parámetros de tiempo real de un componente se restringen a través de la clase *RTComponentDataType*. Las propiedades de configuración del modelo de tiempo real poseen dos campos adicionales:

- *takeValueOf*: En muchos casos, los parámetros de tiempo real de un componente coincidirán en valor con alguna de sus propiedades de configuración funcionales y no podrán ser asignados de forma independiente. Para evitar errores en la asignación de valores en estos casos, se utiliza este campo a través del que se referencia la propiedad funcional de la que se toma el valor. En este caso, no es necesario asignar valor al parámetro de tiempo real, en su lugar una herramienta asignará de manera automática el mismo valor que se le de a la propiedad funcional.
- *sameValueOf*: Este campo se introduce con el objetivo de contemplar el caso de propiedades de tiempo real que puedan ser recalculadas por herramientas de análisis, y que a su vez sean mapeadas a propiedades de configuración del componente. A través de este campo se realiza el mapeado entre ambas. En la siguiente sección se explica más en detalle su utilidad.

Los campos *sameValueOf* y *takeValueOf* en la definición de una propiedad de tiempo real son mutuamente exclusivos. Cuando se usa *takeValueOf*, la correspondiente propiedad de configuración debe tener un valor asignado, y la propiedad de tiempo real tomará siempre el mismo valor que ella, no pudiendo ser modificada. En el caso de *sameValueOf*, en el plan de despliegue inicial se le puede dar valor a la propiedad de configuración, la correspondiente propiedad de tiempo real tomará ese mismo valor de partida, pero si las herramientas de análisis lo modifican, el valor será modificado también en la propiedad de configuración.

Por otro lado, para asignar valores concretos a las propiedades configurables de tiempo real definidas tanto para un componente como para una transacción, se ha definido una nueva clase

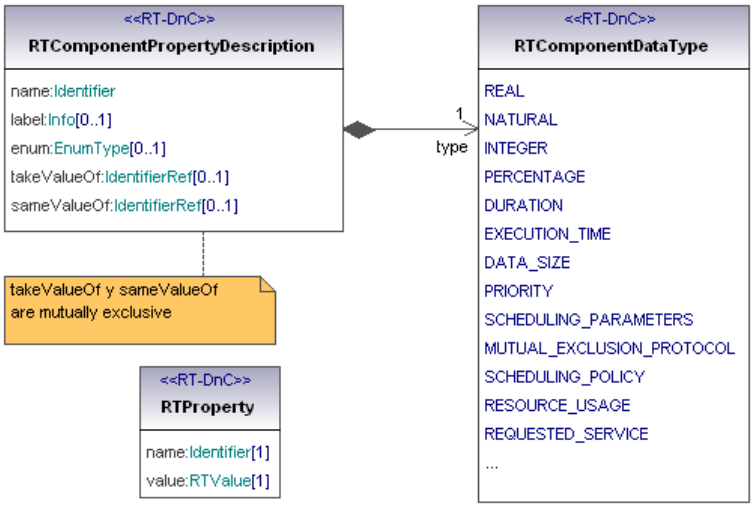


Figura 3.16: Declaración de una propiedad configurable del modelo de tiempo real de un componente

denominada *RTProperty*. Como cualquier otro valor a asignar se define a través del nombre que identifica la propiedad y el valor concreto a asignar. Los tipos de valores que se pueden asignar a una propiedad de tiempo real son los que correspondan al tipo *RTValue*. Este tipo se define como abstracto en el modelo PIM de la especificación. Para cada metodología de modelado que se utilice en conjunto con RT-D&C, será necesario definir los valores admitidos, estableciendo el mapeado entre los tipos de propiedades admitidos (*RTComponentDataType* y *RTEndToEndFlowDataType)* y los valores concretos que se pueden asignar en cada caso.

3.2.2.2. Modelo de tiempo real de un componente software

El modelo de tiempo real que se asocia a una implementación de un componente a través del campo *rtModel* debe formularse siguiendo una estrategia de modelado modular, que permita obtener el modelo de la aplicación completa por composición de los modelos de los elementos que la forman, en este caso componentes y recursos de la plataforma. Para ello, el modelo de un componente software debe ser un modelo parametrizado, elaborado independientemente de la aplicación en la que vaya a ser utilizado, de la plataforma en la que sea instanciado y del resto de componentes que utiliza para implementar su funcionalidad.

La metodología que se utilice para formular los modelos de tiempo real de los componentes tiene que proporcionar primitivas de modelado que se correspondan con los referenciados desde la extensión RT-D&C:

- Modelos de comportamiento temporal de los servicios que ofrece un componente, agrupados por puertos. Cada modelo de puerto se identifica por el nombre definido para el puerto en la interfaz externa del componente. De este modo serán distinguibles los modelos de servicios con el mismo nombre pero ofertados a través de puertos diferentes.
- Referencias a los modelos de los puertos requeridos por el componente. Cada referencia se identifica a través del nombre que se da al puerto requerido en la interfaz externa del componente, de manera que pueda ser resuelta automáticamente en base a las conexiones establecidas para el componente en el plan de despliegue.
- Modelos de las transacciones que gestiona el componente, que deben responder al patrón establecido para ellas en la interfaz externa del componente.

Todas las primitivas de modelado utilizadas para definir el modelo pueden ser parametrizadas, a condición de que queden completamente resueltas una vez asignados valores a los parámetros definidos para el componente a nivel de la implementación, o en el caso de las transacciones, cuando se asigne valores a sus parámetros en la declaración de la carga de trabajo.

El modelo de tiempo real reutilizable que se asocia a una implementación de un componente es manejado únicamente por las herramientas de composición que generan el modelo de análisis de la aplicación. Sería de gran interés formular los modelos de tiempo real utilizando un lenguaje estandarizado, independiente de las herramientas de análisis que procesen el modelo final. Sin embargo, en el momento de desarrollo de esta tesis, los lenguajes estándar disponibles (MARTE) no ofrecían aún la suficiente capacidad de modelado para describir los detalles de interacción que se requiere en el análisis temporal de este tipo de aplicaciones, por lo que los modelos detallados del comportamiento temporal interno del componente deben ser desarrollados utilizando los lenguajes propios de las herramientas. En nuestro caso, y como el objetivo es utilizar el entorno MAST, el lenguaje utilizado es una extensión de Mod-MAST a la que se le han añadido nuevos elementos de modelado que permiten modelar componentes software, con las características expuestas anteriormente. La extensión se denomina CBS-MAST y se explica en la sección 3.3.

3.2.2.3. Ejemplo de aplicación

Como ejemplo de aplicación de las extensiones a la descripción de la implementación de un componente, se muestra en la figura 3.17 algunos aspectos del descriptor de una implementación de la interfaz *ScadaEngine* (el descriptor completo se expone en el capítulo 6). El componente se denomina *AdaScadaEngine* y es una implementación desarrollada en lenguaje Ada 2005, acorde a la tecnología Ada-CCM que se introduce también en el capítulo 6. En esta sección sólo se resaltan algunas de las características que se introducen con la extensión de tiempo real:

- El campo *AdaImpl.rtModel* constituye el localizador del modelo de tiempo real del componente.
- Las propiedades de planificación *samplingTh y dataMtx* describen dos de las nuevas propiedades de configuración de la implementación *AdaScadaEngine*, que son consecuencia de que para el diseño de su código se han utilizado una tarea Ada y un objeto protegido que requiere una primitiva mutex de sincronización. El uso de estos elementos requiere que en el proceso de configuración se asigne valor a sus prioridades y techos de prioridad.
- Las propiedades del modelo de tiempo real *samplingThPeriod*, *samplingThPrty* y *dataMtxCeiling* declaran algunas de las propiedades que están definidas en el modelo de tiempo real (no en el código, como las propiedades de configuración), y a las que se debe asignar valor para que describan correctamente el comportamiento temporal de la implementación del componente.

Los tipos de datos que corresponden a los valores que hay que asignar en el plan de despliegue a las propiedades de configuración se definen en el metamodelo PSM de la tecnología que se utiliza (en este caso Ada-CCM). De forma similar, los tipos de los valores que hay que asignar a las propiedades relativas al modelo de tiempo real, se definen en el metamodelo PSM de la metodología de modelado utilizada (que en este caso es CBS-MAST).

Aunque las propiedades de ambos grupos son diferentes, porque tienen diferente papel en el desarrollo de una aplicación (las propiedades de planificación se utilizan para configurar el código que se ejecuta, mientras que las propiedades de tiempo real sirven para que el modelo de tiempo real describa el comportamiento temporal del código), el valor que hay que asignar a alguna de ellas coincide con el de las otras. Por ejemplo, los valores que hay que asignar a las propiedades de planificación *samplingTh.period* y *dataMtx.ceiling* deben coincidir con los valores de las propiedades del modelo de tiempo real *samplingThPeriod* y *dataMtxCeiling*. Sin

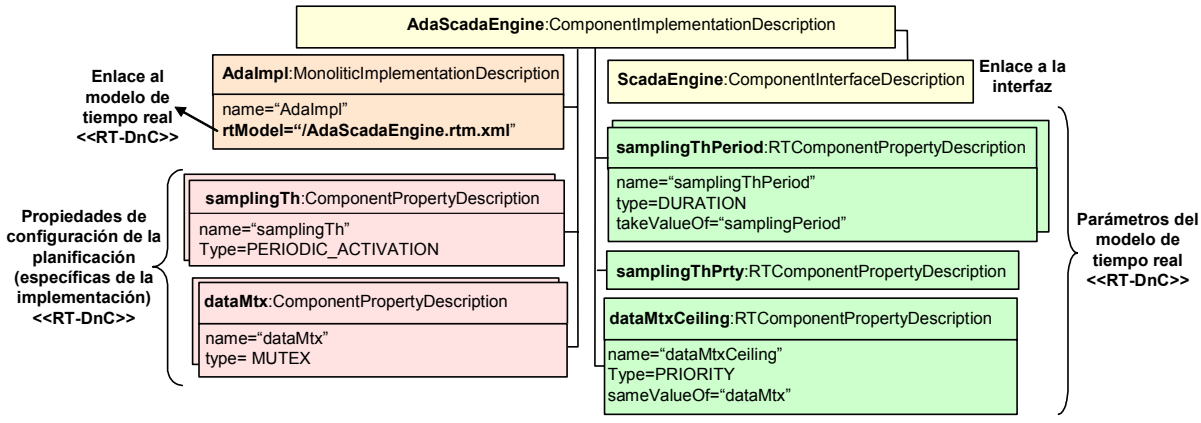


Figura 3.17: Ejemplo de descriptor de implementación con extensiones de tiempo real

embargo, en estos dos casos el origen del valor es diferente. El valor del periodo es consecuencia del diseño, y la asigna el *assembler* de acuerdo con la funcionalidad del modelo, por lo que se utiliza el calificador de asignación *takeValueOf* para relacionarlas. Por el contrario, el valor del techo de prioridad resulta del análisis de planificabilidad realizado sobre el modelo de tiempo real, y por ello se utiliza el calificador *sameValueOf*.

Asimismo, en la figura 3.18 se muestran algunos aspectos del modelo de tiempo real de la implementación del componente referenciado a través del campo *rtModel*. El modelo se formula utilizando CBS-MAST y debe ser consistente con las características definidas tanto en la interfaz como en la implementación del componente, por lo que incluye, entre otros, los siguientes elementos:

- El modelo del recurso compartido *dataMtx*, cuyo techo de prioridad *dataMtxCeiling* es configurable tal y como se ha establecido en la descripción de la implementación.
- El modelo del thread *samplingTh*, cuya prioridad *samplingThPrty* es uno de los parámetros del modelo.
- Los modelos de las operaciones que ofrece el componente a través del puerto *controlPort*, *getLastLoggedMssg* y *getBufferedData*. Esta última se realiza en modo mutuamente exclusivo, lo que se modela indicando que requiere acceso al recurso compartido *dataMtx*.
- La referencia a los modelos de los puertos ofrecidos por los componentes que se conecten a los puertos *adqPort* y *logPort,* que en el modelo de tiempo real representan parámetros del modelo.
- El modelo de las transacciones *SamplingTrans* y *LoggingTrans*. Se muestra en detalle sólo la primera, y se observa como declara como parámetros los establecidos para ella en la interfaz e incluye el resto de elementos que modelan su flujo de control.

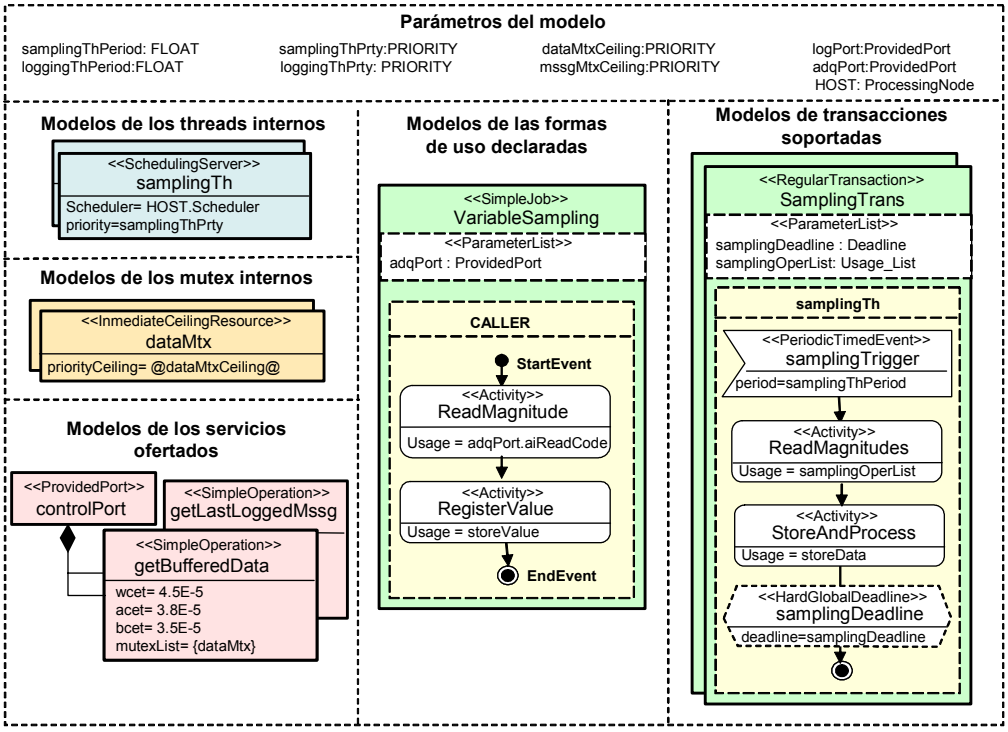


Figura 3.18: Ejemplo de modelo de tiempo real de una implementación de componente

- El resto de parámetros que se han definido para el modelo en la implementación del componente, como *samplingThPrty*, *samplingThPeriod*, etc.
- El modelo de la forma de uso *VariableSampling*, que como se indicó en la descripción de la interfaz del componente sirve como plantilla para declarar las formas de uso con las que se configura la instancia de la transacción *SamplingTrans*.

3.2.3. Descripción de plataformas de tiempo real

La capacidad de procesamiento que proporciona la plataforma y la forma en que es utilizada por los componentes que se instancian en ella son necesarias para evaluar el comportamiento temporal de los componentes software. Por tanto, para evaluar el comportamiento de una aplicación se deberán conocer los modelos de tiempo real de los recursos de la plataforma de los que se haga uso en ella. Dichos modelos pueden ser, al igual que ocurre con los componentes, parametrizados y por lo tanto reutilizables. De esta forma se podrán definir plataformas configurables.

La especificación D&C no define ningún elemento de modelado que permita almacenar información acerca de elementos de la plataforma. La plataforma se concibe y se describe directamente como una instancia, como el conjunto de nodos y redes concretos sobre los que la aplicación va a ser desplegada. En consecuencia, la extensión de tiempo real que RT-D&C introduce en el modelo de datos de la plataforma consiste en definir nuevas entidades de modelado que nos permitan almacenar los modelos de aquellos elementos que pueden formar parte de una plataforma de ejecución.

En la figura 3.19 se muestra como con las extensiones introducidas, el agente que se encarga de definir el modelo de la plataforma, *Domain Administrator*, elaborará previamente los descriptores de los diferentes recursos que se pueden utilizar en las futuras plataformas. Al igual que ocurre con los descriptores de componentes, a través de ellos se podrá acceder a los modelos de tiempo real de los recursos y configurarlos. Posteriormente, cuando se va a planificar el despliegue de una aplicación, se construye el modelo de la plataforma concreta (*Domain*), declarando cada elemento contenido en ella en base a su correspondiente descriptor, y asignándole a los parámetros de configuración los valores concretos que correspondan.

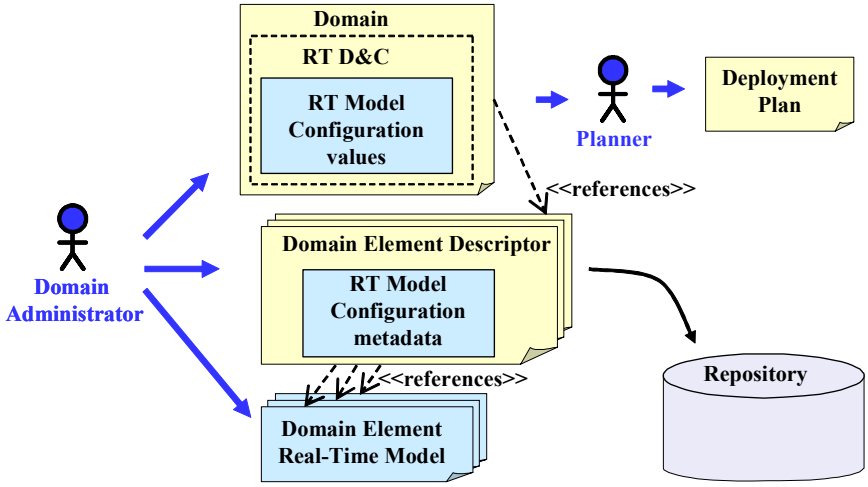


Figura 3.19: Metadatos añadidos al modelo de datos de la plataforma

3.2.3.1. Implementación formal de la extensión en RT-D&C

La estrategia que se adopta para extender el modelo de la plataforma es similar a la que se utiliza en el caso de los componentes. Un dominio se describe como un conjunto de instancias de nodos, redes y recursos. En RT-D&C cada instancia incluida en un dominio se describe referenciando a través del nuevo atributo *source* a su correspondiente descriptor, tal y como una instancia de componente incluida en un plan de despliegue referencia a su descriptor de implementación.

Un descriptor de elemento de plataforma, como se muestra en la figura 3.20 para el caso de un nodo (*NodeDescription*) incluye la referencia al modelo de tiempo real del elemento (atributo *rtModel*) y la definición de las propiedades configurables de dicho modelo, a través del atributo *rtProperty*. Asimismo, a través del atributo *rtConfigProperty* se pueden asignar valores concretos a dichas propiedades cuando una instancia de nodo, red o recurso sea declarada en un dominio.

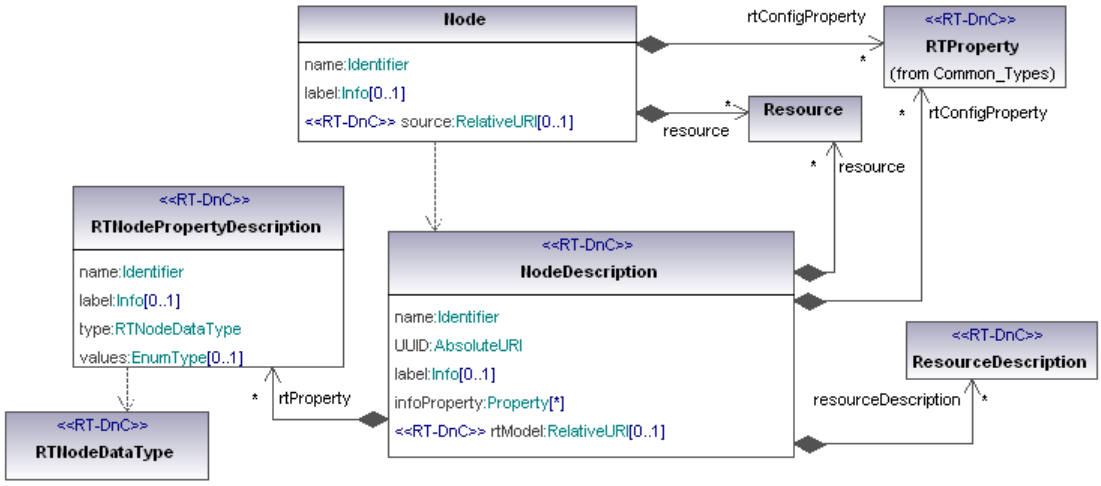


Figura 3.20: Descriptor de un nodo en RT-D&C

Se han definido cuatro tipos de descriptores dentro del modelo de datos de la plataforma:

- Descriptores de nodos (*NodeDescription*): Almacenan la información relativa a los nodos procesadores. Como se ve en la figura 3.20, el descriptor de un nodo incluye aquellos recursos que están definidos en él. Cuando una instancia de nodo se incluya en un dominio, se incluyen los recursos declarados en su descriptor, además de otros que se pueden añadir en la instanciación.
- Descriptores de redes (*InterconnectDescription*): Almacenan información acerca de las redes de interconexión. Al igual que en los descriptores de los nodos, contienen la información relativa a los recursos definidos en ellas.
- Descriptores de recursos (*ResourceDescription*): Permiten almacenar el modelo de tiempo real de recursos que:
  - Formen parte de la descripción de un nodo o red de interconexión, y cuyo comportamiento temporal influya en el comportamiento de la aplicación. Será necesario por tanto contar también con los modelos de tiempo real que describen dicho comportamiento. Un ejemplo de este tipo de recursos lo constituyen los *drivers* que modelan la sobrecarga que se introduce en un nodo como consecuencia de la gestión del envío y recepción de mensajes a través de una determinada red de comunicación.

Los descriptores de este tipo de recursos se incluirán en la descripción de los propios nodos o redes de los que pueden formar parte.

- Se puedan utilizar en una plataforma de forma compartida entre diferentes nodos, e influyan en el comportamiento temporal. Por ejemplo: una impresora en red, una tarjeta de adquisición de datos conectada a una red, etc. En este caso, se almacenarán como elementos independientes en el repositorio.

- Descriptores de dominios completos *(DomainDescription)*: Podemos incluso definir descriptores de dominios completos, que permitan almacenar conjuntos de nodos, redes de interconexión y recursos compartidos, cada uno con su correspondiente modelo de tiempo real. En este caso es necesario definir el mapeado entre las propiedades externas definidas para el modelo completo del dominio y las propiedades de los elementos internos a las que corresponden.

Se han restringido los tipos a los que puede corresponder los parámetros de tiempo real que se pueden definir en el modelo de los elementos de la plataforma. En el caso de un nodo se ha definido el tipo *RTNodeDataType*, en el de una red el tipo *RTInterconnectDataType*, y en el de un recurso *RTResourceDataType*. El tipo *RTDomainDataType* es la unión de todos ellos, ya que en un dominio se podrán definir propiedades para todos esos tipos de elementos.

3.2.3.2. Introducción de modelos de mecanismos de conexión entre componentes

Un aspecto importante a tener en cuenta cuando se trata de analizar una aplicación de tiempo real basada en componentes es el comportamiento de los mecanismos de invocación entre componentes, especialmente si la invocación es remota y requiere el uso de los servicios de comunicación. Los modelos de estos mecanismos dependen del tipo de servicio de comunicaciones, o middleware, empleado.

Aunque la especificación D&C tiene por objetivo soportar el despliegue de aplicaciones basadas en componentes en plataformas heterogéneas, no incluye ningún medio que permita especificar el mecanismo concreto que se desea emplear para realizar cada conexión entre componentes. Éstas se conciben como conexiones directas, eliminando de este modo la posibilidad de que puedan ser implementadas a través de diferentes mecanismos. Por ejemplo, si deseamos hacer que dos componentes de una aplicación utilicen CORBA [CORBA] para comunicarse, pero otros dos componentes empleen una implementación del anexo DSA de Ada [DSA], no tendríamos modo de definirlo a nivel del modelo que propone la especificación. Para ser capaces de elaborar aplicaciones basadas en componentes que empleen mecanismos de comunicación heterogéneos, es necesario extender la especificación D&C de forma que se pueda definir el mecanismo concreto que se quiere emplear para cada conexión entre puertos de componentes.

La idea es similar a la que se presenta en [BB05], donde se propone una extensión de la especificación D&C con el objetivo de desarrollar aplicaciones en las que interaccionen componentes desarrollados de acuerdo a diferentes modelos. Para ello introducen soporte para conectores software en la especificación. Un conector software es un elemento que implementa una interacción entre componentes, que se puede emplear para aislar el código del componente de la gestión de las comunicaciones o permitir la comunicación entre componentes desarrollados según diferentes tecnologías. La estrategia propuesta consiste en especificar el tipo de conector que se desea para cada conexión en términos de un “estilo de comunicación” (invocación de procedimiento, envío de mensajes, memoria compartida, etc.) y de propiedades

no funcionales, que se describen a través de elementos de tipo *Property*. Estos datos servirán para generar el código del conector en base al despliegue de la aplicación. La propuesta es adecuada, sin embargo, no se realiza ninguna restricción sobre cuáles son las propiedades que se pueden asociar a cada tipo de conexión, y además su especificación se realiza a nivel de instancia, sin almacenamiento de ningún tipo de información acerca de los tipos de conexión disponibles. La estrategia que se propone en esta tesis es similar, sin embargo, el aspecto clave por el que se introduce la extensión y que aparece como consecuencia de la naturaleza de tiempo real de las aplicaciones, es la necesidad de contar con un modelo de comportamiento temporal de cada tipo de conexión que se pueda utilizar, esto es, de cada tipo de conector software. Dicho modelo de tiempo real describe la secuencia de actividades que se ejecuta para completar una interacción entre componentes cuando se emplea el conector elegido.

Nuestra propuesta define un nuevo tipo de descriptor dentro del modelo de datos de la plataforma, que referencia el modelo de tiempo real de un mecanismo de conexión y declara sus propiedades configurables tanto de tiempo real como de planificación. La figura 3.21 muestra el descriptor, denominado *ConnectionMechanismDescription*, que permite describir las características de los diferentes tipos de conexión que se soporten en la tecnología. A través de este elemento:

- Se identifican las herramientas que más tarde se utilizan para generar tanto el código como el modelo de tiempo real de cada conector que utiliza el mecanismo.
- Se referencia el modelo de tiempo real del mecanismo de conexión y se declaran sus propiedades configurables, a través del campo *rtProperty*. Este modelo incluye los elementos de modelado comunes a todas las conexiones que se realicen de acuerdo a él. Por ejemplo, si un mecanismo utiliza un thread global para atender todas las invocaciones recibidas a través de las diferentes conexiones, el recurso de tipo *Scheduling_Server* que lo modela deberá ser incluido en el modelo del mecanismo. Una instancia de él se añadirá al modelo de tiempo real del sistema final por cada nodo de la plataforma en que se utilice el mecanismo. Los modelos de cada conector elaborado en base a este mecanismo harán referencia a dichas instancias. Por tanto, las propiedades *rtProperty* son propiedades de tiempo real globales del mecanismo de conexión, y serán resueltas cuando en el modelo de dominio se declare que un nodo da soporte al mecanismo.

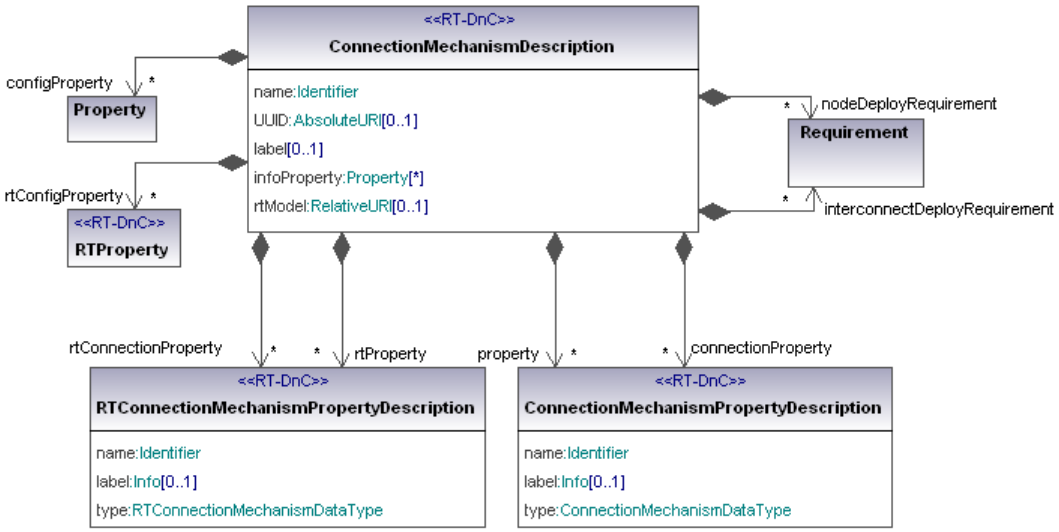


Figura 3.21: Descriptor RT-D&C de un mecanismo de conexión

- Se declaran las propiedades configurables del modelo de tiempo real de cada conexión que se realiza a través del mecanismo, a través del campo *rtConnectionProperty*. Estas propiedades, frente a las anteriores que son de carácter global, sirven para configurar el modelo temporal de cada conexión (conector) de manera independiente. Serán resueltas cuando se declare una conexión concreta entre dos puertos de componentes en un plan de despliegue.
- Se declaran las propiedades de configuración del mecanismo, orientadas principalmente a la planificación. Muchos de sus valores podrán ser obtenidos automáticamente a partir del análisis del modelo de la aplicación. Al igual que ocurre con las propiedades de tiempo real, se pueden definir propiedades globales del mecanismo, a través del campo *property*, o características de cada conexión, a través del campo *connectionProperty*.
- Se declaran los requisitos que el tipo de conexión impone sobre la plataforma para poder ser utilizado. Estos requisitos pueden afectar tanto a los nodos en los que se instancien los componentes implicados, *nodeDeployRequirement*, como a la red a través de la que se comuniquen, *interconnectDeployRequirement*.

En el modelo de una plataforma se incluye información que indica qué mecanismos de conexión son soportados por cada uno de los nodos que la forman, pues sus modelos de tiempo real configurados deben ser incorporados al modelo de tiempo real de la plataforma. Para ello, se ha definido un nuevo campo en la declaración de una instancia de nodo, denominado *supportedConnection*, como muestra la figura 3.22. A través de él se definen todos los mecanismos soportados y se configuran sus propiedades globales, tanto de tiempo real como de planificación. Como se verá más adelante, cuando en el plan de despliegue se defina una conexión concreta entre dos puertos de componente, se identificará el mecanismo que se quiere utilizar, y deberá comprobarse que los nodos en que están instalados los componentes ofrecen soporte para dicho mecanismo de conexión. A ese nivel se asignarán además los valores de las propiedades de configuración y de tiempo real propias de cada conexión.

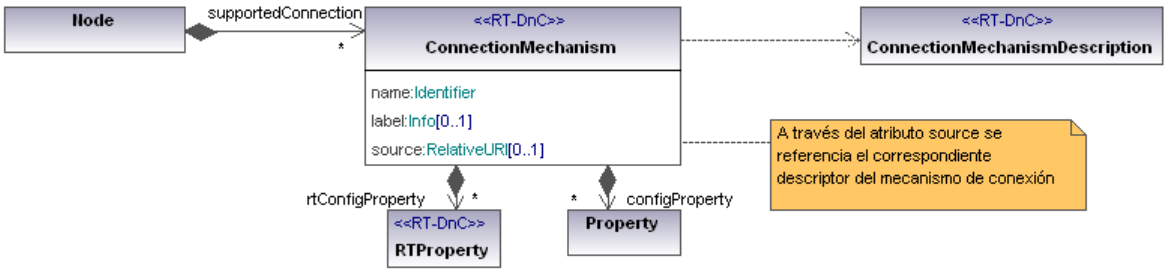


Figura 3.22: Declaración de los mecanismos de conexión soportados en un nodo

3.2.3.3. Ejemplo de aplicación

En la figura 3.23 se representa una parte de la descripción de la plataforma elegida para ejecutar la aplicación que se utiliza como ejemplo en el capítulo 6. En él se encuentran las declaraciones de los nodos, *CentralProc* y *RemoteProc*, y la red a través de la que se comunican, *Net*. En todos los casos, a través del campo *source* se accede al descriptor en el que se encuentra la referencia al modelo de tiempo real del elemento correspondiente.

Como ejemplo, se muestra en la figura 3.24 el descriptor al que referencian los dos nodos. Describe un nodo tipo PC de 750MHz ejecutando el sistema operativo MaRTE OS. En él se hace referencia al correspondiente modelo de tiempo real y se declaran sus parámetros. En este

caso sólo hay uno, el que sirve para adaptar el modelo a la velocidad concreta del procesador, *speedFactor*. Dicho parámetro se configura en cada instancia de nodo a través del campo *rtConfigProperty*. En este caso vemos como los dos nodos instanciados en el dominio son del mismo tipo (referencian al mismo tipo de descriptor), pero uno de ellos (*CentralProc*) es dos veces más rápido que el otro, por lo que se le asigna una valor 2.0 de *speedFactor*.

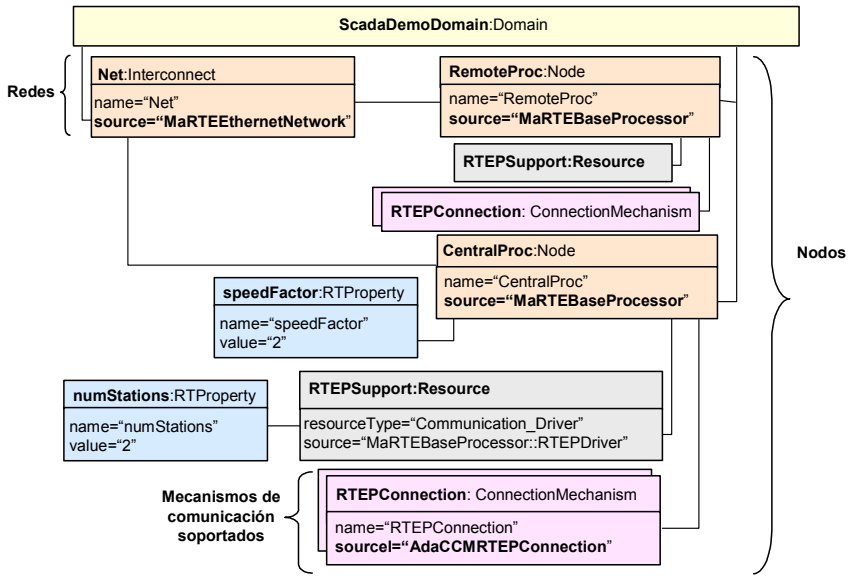


Figura 3.23: Ejemplo de modelo de plataforma con extensiones de tiempo real

En el descriptor de un nodo se incluyen también los descriptores de aquellos recursos que pueden ser utilizados en el nodo y que influyen en su comportamiento temporal. En el caso del descriptor del nodo MaRTE OS, se incluye el descriptor del driver (*RTEPDriver*) que se ha de incluir en el sistema cuando un nodo de este tipo se comunique con otro a través del protocolo RT-EP. Cuando se define una instancia de nodo en el dominio, se le añade un recurso de este tipo, tal y como se muestra para los dos nodos del ejemplo en la figura 3.23. El modelo del driver es a su vez parametrizado, por lo que en la declaración de cada uno es necesario asignarle valor al parámetro *numStations*.

El modelo de dominio muestra también como cada uno de los nodos soporta un tipo de mecanismo de conexión entre componentes basado en el protocolo RT-EP, que ha sido desarrollado como parte de la tecnología Ada-CCM. Las características de las conexiones realizadas a través de este mecanismo y el modo de configurarlas se explican en el capítulo 6.

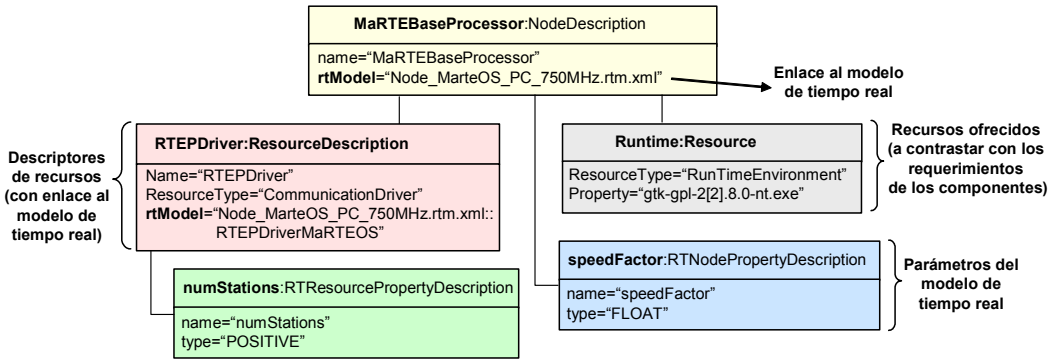


Figura 3.24: Ejemplo de descriptor de nodo

3.2.4. Descripción de aplicaciones de tiempo real basada en componentes

La descripción del plan de despliegue de una aplicación basada en componentes de tiempo real se realiza de forma similar a la de una aplicación general:

1. Se definen las instancias de componentes que forman la aplicación.
2. Se le asigna a cada instancia el nodo en el que va a ejecutar.
3. Se definen las conexiones entre instancias.
4. Se mapean las propiedades de la aplicación a las propiedades de cada instancia.

Desde este punto de vista estructural, no existe ninguna característica especial que se haya de añadir para el caso de una aplicación con requisitos de tiempo real. Las diferencias aparecen en lo referente al diseño de tiempo real de la aplicación. El diseñador de una aplicación basada en componentes (*planner*) sólo puede controlar la ejecución de la aplicación a través de la información que incluye en el plan de despliegue. Por ello, el primer aspecto en que se extiende un plan de despliegue, como muestra la figura 3.25, es proporcionando al *planner* la capacidad de configurar la planificación de la aplicación (*Scheduling Configuration Values*), mediante la asignación de valores a las propiedades de planificación definidas en las instancias de componentes y en los elementos de la plataforma.

En la mayor parte de los casos estos valores serán extraídos del análisis del modelo temporal de la aplicación, por lo que a partir del plan de despliegue el *planner* debe ser capaz de acceder a toda la información necesaria para generar dicho modelo:

- El modelo de tiempo real de cada una de las instancias de componentes que forman la aplicación.
- El modelo de tiempo real de los conectores que se introduzcan para dar soporte a las comunicaciones entre puertos de componentes.
- El modelo de tiempo real de los elementos que forman la plataforma de ejecución.
- El modelo de la carga de trabajo que ejecuta la aplicación.

En el caso de las instancias de componentes, a partir del plan de despliegue se debe poder acceder a los modelos de cada una, y particularizarlos según el uso que se haga de ellos. El acceso al modelo de cada instancia es directo, ya que toda instancia declarada en un plan de despliegue posee una referencia a su correspondiente descriptor de implementación, en el que se encuentra a su vez la referencia al modelo. Para adaptar los modelos al contexto concreto de la aplicación, el *planner* asigna valores concretos a los parámetros configurables de los modelos de tiempo real de cada instancia de componente (*RT Model Configuration Values*).

Como se expuso anteriormente, un aspecto esencial para poder analizar el comportamiento de una aplicación basada en componentes distribuida, es contar con el modelo que describe la secuencia de actividades que se ejecutan cada vez que un componente cliente invoca un método en un componente servidor remoto. A diferencia de la estrategia expuesta en [BB05], donde se cualifica el tipo de conexión entre componentes en la descripción de un ensamblado, en nuestro caso extendemos la descripción de una conexión entre componentes a nivel del plan despliegue, pues la decisión acerca del tipo de comunicación a utilizar se realiza durante la fase de planificación. Cuando se describe un ensamblado, se realiza únicamente a nivel estructural, dejando todas las decisiones que dependen de la plataforma en que se vaya a ejecutar para la fase de planificación. Por tanto, en cada conexión entre puertos de componentes declarada en un plan de despliegue se especifica y configura el mecanismo concreto a través del que se quiere realizar la interacción (*Connections Configuration*).

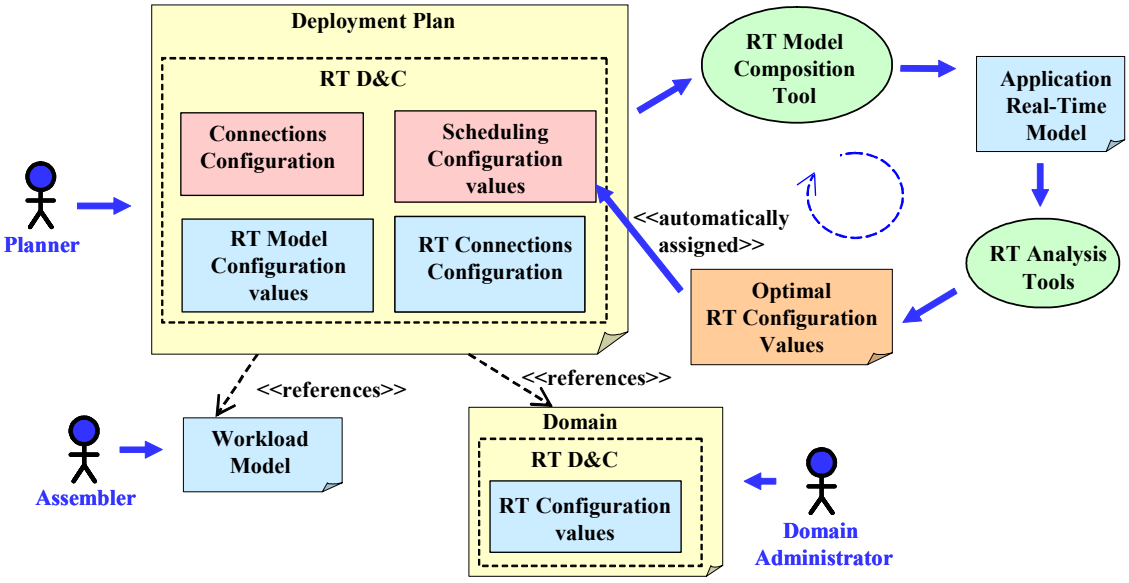


Figura 3.25: Metadatos añadidos al modelo de datos de ejecución

Asimismo, desde el plan de despliegue se puede acceder al modelo de la plataforma y al modelo de la carga de trabajo (la forma en que éste se especifica se explica más adelante). En base a toda esta información, la herramienta de composición de modelos genera el modelo de la aplicación que sirve de entrada a las herramientas de análisis de tiempo real. Esta herramienta es PSM, esto es, depende de la metodología de modelado concreta que se utilice como base para formular los modelos y aplicar análisis temporal. En la sección 3.4 se muestra el proceso de composición de modelos para el caso de utilizar la metodología de modelado MAST con sus correspondientes extensiones.

El modelo obtenido se utiliza como base para el cálculo de los valores óptimos de aquellas propiedades que afectan al cumplimiento de los requisitos temporales de la aplicación. Los valores obtenidos serán reasignados a las correspondientes propiedades de configuración a las que se encontrasen mapeadas en el plan de despliegue, como se muestra en la figura 3.25. La herramienta que se encarga de trasladar los resultados del análisis a las propiedades de configuración correspondientes en el plan de despliegue es también una herramienta PSM, pues depende de la tecnología de componentes que se utilice. En los capítulos 4 y 6 se explica el proceso de configuración en el caso de la tecnología de componentes RT-CCM.

3.2.4.1. Implementación formal de la extensión en RT-D&C

Toda la información que se ha de añadir al plan de despliegue con el objetivo de configurar los modelos de tiempo real y la planificabilidad de los componentes se realiza a nivel de las instancias, por tanto el elemento que se extiende es *InstanceDeploymentDescription*. Las extensiones se muestran en la figura 3.26. Se añaden nuevos campos para sobrescribir tanto las propiedades de tiempo real (*rtConfigProperty*), como las propiedades de configuración propias de la implementación y relacionadas con planificación (*specificConfigProperty*).

Para configurar cada conexión entre puertos de componentes se ha añadido un nuevo campo, *connectionConfiguration*, al elemento *PlanConnectionDescription*. Como muestra la figura 3.27, cada conexión entre puertos de componentes incluida en un plan de despliegue podrá referenciar a través del campo *source* el tipo de conexión deseado (de entre aquellos soportados por los nodos a los que sean asignados). A través del campo *configProperty* se configura la

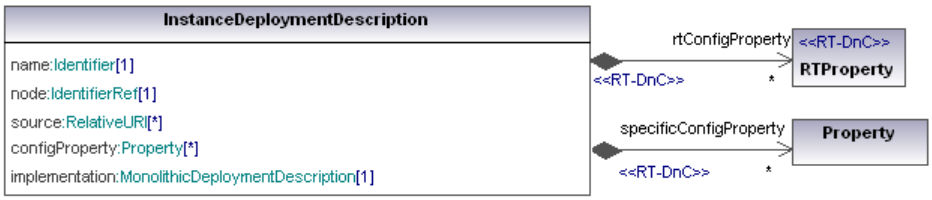


Figura 3.26: Extensión RT-D&C de la declaración de una instancia en un plan de despliegue

conexión, asignando valores a cada propiedad declarada en el descriptor, y del mismo modo se asignan valores a las propiedades que afectan al modelo de tiempo real, a través del campo *rtConfigProperty*. Asimismo, se identifica la red a través de la que se realiza la conexión, por si los nodos estuviesen conectados a través de más de una red de interconexión (este aspecto no estaba resuelto en la especificación D&C). Toda esta información es procesada por la herramienta de generación de conectores, que realiza una doble tarea:

- Genera el código del conector correspondiente, que será intercalado entre los puertos de los componentes que conecta.
- Genera el modelo de tiempo real de dicho conector, que será utilizado como entrada de la herramienta de composición de modelos.

En la figura 3.27 se muestran también los elementos que proporcionan el acceso desde el plan de despliegue al modelo de la carga de trabajo (cuya definición se explica en la siguiente sección). En el plan de despliegue se incluye la referencia al elemento de modelado que describe la carga de trabajo de la aplicación, *rtWorkloadModel*. El acceso al modelo de la plataforma se realiza a través del elemento de tipo *Domain*, que D&C ya considera que se procesa de forma conjunta con el plan de despliegue.

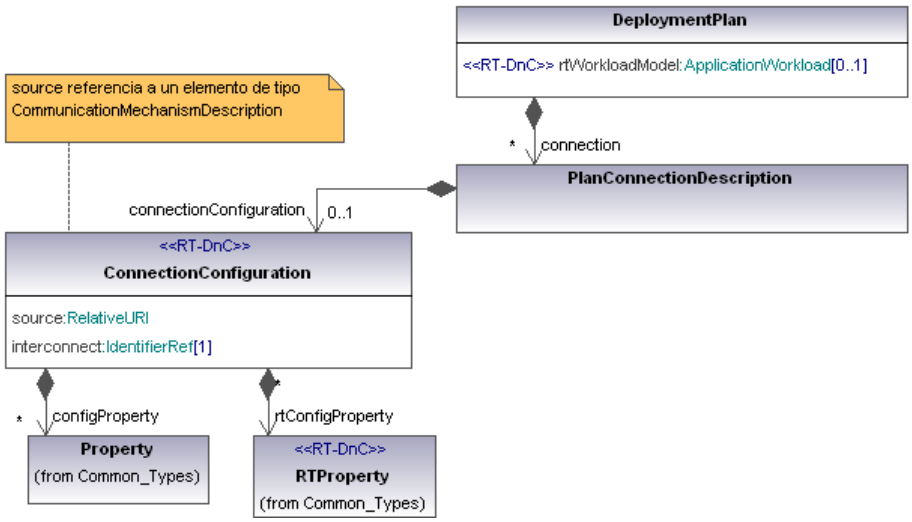


Figura 3.27: Extensión RT-D&C de la declaración de conexiones en un plan de despliegue

3.2.4.2. Ejemplo de aplicación

Como ejemplo de la extensión de tiempo real de la descripción de un plan de despliegue, se muestra en la figura 3.28 un fragmento del plan de despliegue de la aplicación *ScadaDemo* utilizada como ejemplo. En el fragmento mostrado sólo aparece detallada la declaración de la

instancia *engine*, correspondiente a un componente de tipo *ScadaEngine*. Vemos como se realiza la asignación de la propiedad de configuración *samplingPeriod*, cuyo valor habrá sido definido por el *assembler*. No es necesario asignar valores de forma explícita a aquellas propiedades de tiempo real que están mapeadas a propiedades de configuración, como es *samplingThPeriod*. Lo mismo ocurre con la propiedad específica de la implementación *dataMtx* y la propiedad de tiempo real *dataMtxCeiling*. Al estar mapeadas la una a la otra, basta con darle valor a una de ellas. En este caso además, la propiedad de tiempo real puede ser calculada automáticamente por las herramientas de análisis, y en consecuencia, el valor de la propiedad de configuración puede ser modificado en la versión final del plan de despliegue.

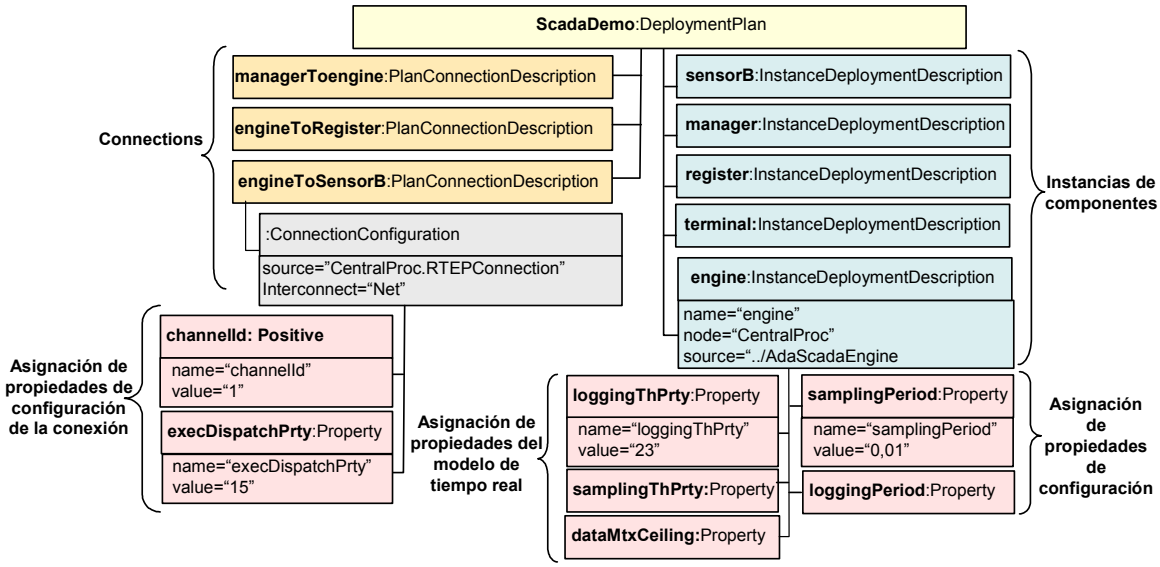


Figura 3.28: Fragmento de declaración del plan de despliegue de una aplicación con extensiones de tiempo real

Se muestra también en la figura un ejemplo de configuración de una conexión. En el despliegue elegido, la instancia *engine* y una de las instancias de sensores que controla a través de su puerto *adqPort*, *sensorB*, se encuentran asignadas a nodos diferentes, y es necesario definir el tipo de mecanismo de comunicación que se va a utilizar entre ellos. En este caso se va a utilizar un mecanismo basado en el protocolo RT-EP. Para configurar la conexión de este modo, se utiliza un elemento de tipo *connectionConfiguration*, y desde él, se referencia el mecanismo de conexión correspondiente, *CentralProc.RTEPConnection* en este caso. A la herramienta que se encarga de generar el modelo de tiempo real de los conectores le basta con conocer dicha referencia, junto con la interfaz de los puertos conectados, y los valores asignados a sus parámetros de tiempo real. Asimismo, los valores asignados a las propiedades de configuración de la conexión, *execDispatchPriority* y *ChannelId*, serán procesados por la herramienta que genera el código del conector.

3.2.5. Descripción de cargas de trabajo

Como se explicó en el capítulo 2, el contexto de análisis de una aplicación es el marco en el que se aplica el análisis de planificabilidad. En el caso de una aplicación basada en componentes, el contexto de análisis queda definido por:

- La descripción de la plataforma que define los recursos disponibles para ejecutar la aplicación.

- El plan de despliegue que define la aplicación.
- La carga de trabajo, que describe estadísticamente las actividades que ejecuta la aplicación de acuerdo a las interacciones que se esperan del entorno y del modo en que se ejecutan.

La especificación D&C no ofrece ningún elemento de modelado que describa las cargas de trabajo de una aplicación, y por ello, en RT-D&C se definen de forma completa los elementos de modelado que permiten describirlas. Estos elementos constituyen el modelo de datos de la carga de trabajo (*Workload Data Model*), complementario a los modelos de componentes, plataforma y ejecución.

Como se muestra en la figura 3.29 (y se mostraba también en la figura 3.25) la carga de trabajo la describe el *assembler*, en base a la especificación de la aplicación y del entorno en el que opera. Para su formulación, hace uso de la especificación de las transacciones que pueden iniciar los componentes que forman parte de la aplicación, que se encuentran declaradas en sus interfaces externas. Se declara una carga de trabajo por cada modo de operación diferente en que puede ejecutar la aplicación y que por tener establecidos requisitos de tiempo real deba definir un contexto de análisis. Para definir el *Workload Data Model* se ha optado por desarrollar una estructura de datos independiente del plan de despliegue, aunque relacionada con él a través de referencias. Con esta estrategia se aísla la parte específica de tiempo real, sin afectar por tanto al plan de despliegue de aplicaciones de propósito general.

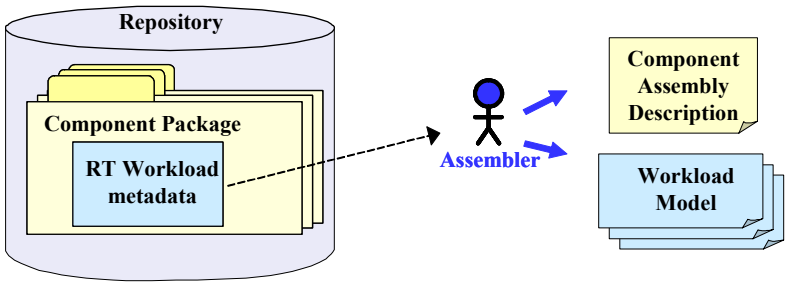


Figura 3.29: Definición del modelo de carga de trabajo

Las cargas de trabajo se describen como listas de instancias de transacciones cualificadas con determinadas características. Por ello, se simplifica su declaración si se utiliza una estrategia de anidamiento, en la que se declara una carga de trabajo partiendo de otra previamente definida, a la que se añaden nuevas transacciones. La estructura resultante, como se muestra en la figura 3.30, se denomina *ApplicationWorkload* e incluye un conjunto de elementos de tipo *RTWorkloadInstance*, cada uno de ellos representando una carga de trabajo de la aplicación susceptible de ser analizada. Cada carga de trabajo se define a través de los siguientes atributos:

- *name*: Nombre que identifica al modo de operación al que corresponde la carga de trabajo.
- *base*: Carga de trabajo raíz, si existe, sobre la que se añaden nuevas transacciones.
- *endToEndFlow*: Conjunto de transacciones ejecutadas en el correspondiente contexto de ejecución de la aplicación. Cada elemento de tipo *RTEndToEndFlow* representa una instancia de una transacción que se incluye en la carga de trabajo. Cada transacción se describe a través de los siguientes atributos:
  - *name*: Nombre que identifica la transacción.

- *instanceName*: Nombre de la instancia de componente que inicia la transacción. El nombre debe corresponder con el que la instancia tenga asignada en el correspondiente plan de despliegue.
- *description*: Modelo de la transacción, que debe corresponder a uno de los tipos de transacción soportados por el componente definidos en su interfaz externa.
- *rtConfigProperty*: Valores concretos asignados a las propiedades configurables definidas para la transacción en su correspondiente declaración.

Cuando alguno de los parámetros de una transacción corresponda a una forma de uso parametrizable, que hay que particularizar para cada transacción, la instancia de la forma de uso se declara de forma independiente, a través del campo *usage*, y a continuación se le pasa por referencia al correspondiente parámetro de la transacción. A continuación se muestra un ejemplo de este caso.

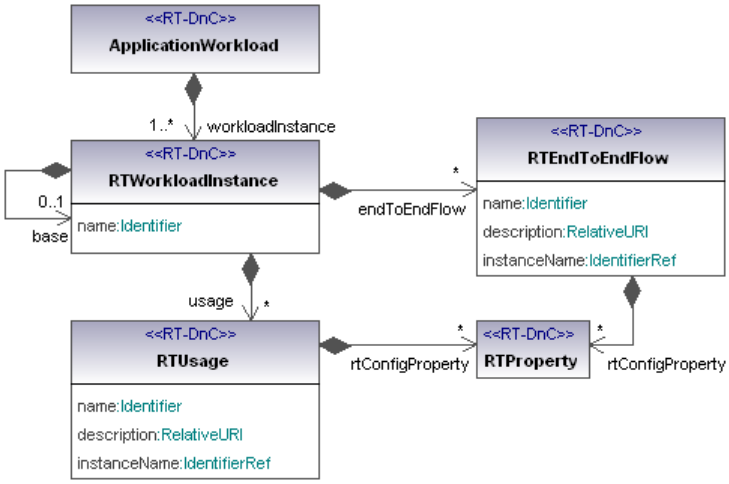


Figura 3.30: Definición RT-D&C del modelo de carga de trabajo

3.2.5.1. Ejemplo de aplicación

Como ejemplo de formulación de la carga de trabajo de un sistema, se muestran en la figura 3.31 las posibles cargas de trabajo de la aplicación *ScadaDemo*. La aplicación posee dos contextos de análisis, y por tanto dos posibles modelos de carga de trabajo, que corresponden a la supervisión de 2 y 3 señales. En este caso se muestra en detalle uno de ellos, *TwoSupervisions*. La carga de trabajo está formada por cinco transacciones: *displayTransInst*, *changeTransInst, checkingTransInst, loggingTransInst* y *samplingTransInst*, mostrándose en el fragmento la declaración completa de esta última (su correspondiente declaración se mostró como ejemplo en la sección 3.2.1.2). Para definir esta transacción deben asignarse valores a su parámetros *samplingDeadline* y *samplingOperList*.

Para asignar valor al parámetro *SamplingOperList* es necesario instanciar previamente dos formas de uso de tipo *VariableSampling*, una por cada magnitud que se va a controlar en este caso. La plantilla para instanciar estas formas de uso se encuentra también definida en la interfaz externa del componente *ScadaEngine* (ver figura 3.13). Una vez instanciadas las dos formas de uso, *Supervision1* y *Supervision2*, cada una recibiendo como parámetro la referencia al componente que actúa como sensor de cada magnitud, se le pasan como valor asignado al parámetro *samplingOperList* de la instancia de la transacción.

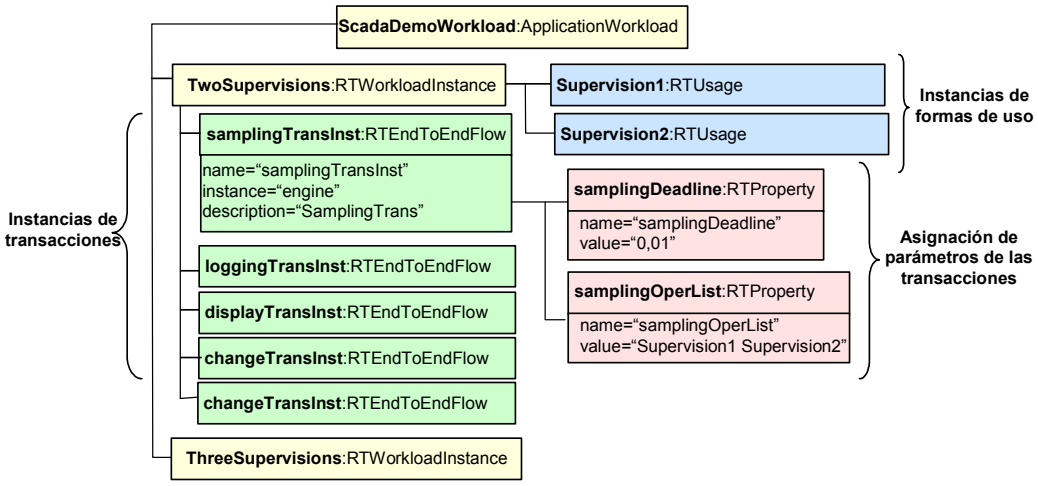


Figura 3.31: Ejemplo de declaración de la carga de trabajo de una aplicación

3.2.6. Formulación de los modelos

La especificación D&C define los modelos de datos que se han de utilizar para almacenar de forma persistente la información que describe componentes, aplicaciones y plataformas. Para poder almacenar dicha información de forma eficiente se van a utilizar ficheros XML, ya que el almacenamiento en modelos UML resultaría más pesado y difícil de manejar. A tal fin, se ha realizado el mapeado de los modelos de datos definidos en la especificación a plantillas *W3C-Schema*, que servirán de base para la validación de los diferentes tipos de ficheros XML utilizados para almacenar la información. Todas las extensiones de tiempo real definidas se han formulado a su vez en los correspondientes schemas.

La figura 3.32 muestra el conjunto de plantillas desarrolladas, junto con las dependencias entre ellas. La relación entre las plantillas es la misma que la relación entre paquetes del modelo UML de la especificación. Sin embargo, aparecen dos plantillas que no se corresponden a ninguno de estos paquetes: *RTMethodologyTypes.xsd* y *TechnologyTypes.xsd*. Su aparición se debe a la estrategia escogida para abordar la transformación del modelo PIM de RT-D&C a un modelo PSM. Existen tipos de datos que se definen como abstractos en el modelo PIM, que tendrán que ser definidos formalmente cuando se pase a un modelo PSM. Esta definición se llevará a cabo a través de estas dos plantillas. Gracias a ellas se distinguen en el proceso de transformación aquellas características que dependen de la tecnología de componentes utilizada (CCM,

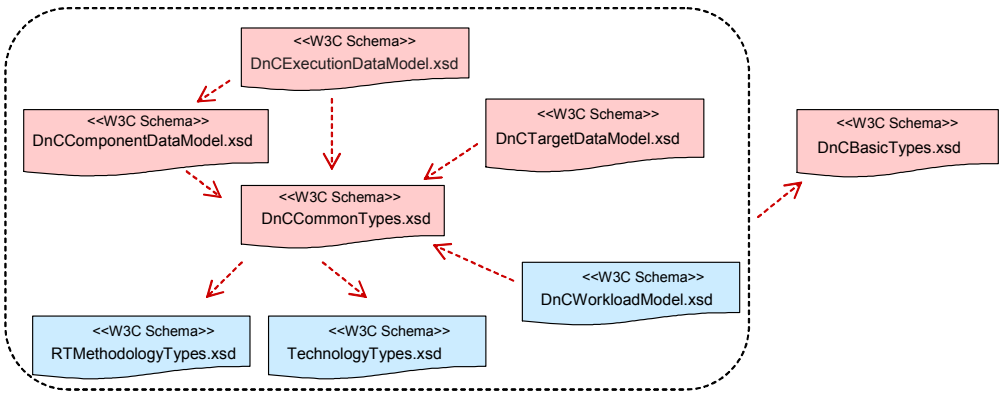


Figura 3.32: Plantillas W3C-schema de la especificación D&C extendida

JavaBeans, etc.), de aquellas que dependen de la metodología de modelado (CBS-MAST en nuestro caso):

- En el primer caso hay que definir, por ejemplo, los tipos de valores que se pueden asignar a propiedades de configuración funcionales (a través de los tipos *DataType* y *Any*). Estas características se definen en el schema *TechnologyTypes.xsd*.
- En el segundo caso hay que hacer lo mismo pero para propiedades de tiempo real (tipo *RTValue*), y se incluyen en el schema *RTMethodologyTypes.xsd*.

Los Schemas anteriores sirven de plantilla para la definición de los diferentes ficheros XML que se pueden almacenar en el repositorio conteniendo información acerca de componentes, aplicaciones o elementos de la plataforma. En la siguiente tabla se muestran los descriptores que se pueden almacenar como elementos independientes, con sus correspondientes extensiones.

Tabla 3.1: Correspondencia entre elementos del modelo (descriptores) y ficheros XML

Elemento de modelado	Extensión (+ .xml)
Component Interface Description	.ccd
Component Implementation Description	.cid
Implementation Artifact Description	.iad
Component Package Description	.cpd
Package Configuration	.pcd
Deployment Plan	.cdp
Domain	.tdm
Domain Description	.dmd
Node Description	.ndd
Interconnect Description	.icd
Connection Mechanism Description	.cmd
Resource Description	.rsd
Application Workload	.wld

3.3. Adaptación de Mod-MAST al desarrollo de software basado en componentes

Para generar el modelo de tiempo real de una aplicación basada en componentes es necesario que la metodología de modelado utilizada ofrezca primitivas de modelado que representen de forma directa los elementos que se utilizan para definir la arquitectura de estas aplicaciones: componentes, conectores, puertos de componentes, etc. En el capítulo anterior se introdujo la metodología Mod-MAST, que proporciona capacidad para formular modelos de tiempo real de módulos software y hardware reutilizables. Con el objetivo de adaptarla a tecnologías basadas en componentes, se ha definido el metamodelo CBS-MAST, que extiende a Mod-MAST con nuevos elementos de modelado que ofrecen una semántica más cercana al desarrollo basado en componentes. Los aspectos más relevantes de la extensión se presentan en esta sección, mientras que en el anexo B se pueden consultar los detalles de las clases añadidas.

Para poder formular modelos de tiempo real reutilizables de los elementos con los que se construye una aplicación basada en componentes se definen dos nuevos tipos de contenedores: *Software_Component* y *Software_Connector*. Como muestra la figura 3.33, *Software_Component* es un tipo derivado de *Software_Module*, que representa un componente software reutilizable cuya funcionalidad se describe en función de los puertos que ofrece y requiere. Como aspecto característico, todo descriptor de tipo *Software_Component* puede definir en su lista de elementos declarados:

- Elementos de tipo *Provided_Port*, que agrupan los modelos de los servicios ofrecidos a través de un puerto de un componente. Cada grupo se identifica por el nombre del puerto. Para modelar los servicios ofertados se pueden utilizar cualquiera de los tipos derivados de *Requested_Usage*.
- Elementos de tipo *Required_Port*, a través de los que se identifican los puertos requeridos del componente y se referencian los componentes conectados a ellos. Las invocaciones realizadas en servicios de componentes conectados a través de dichos puertos se formulan en el descriptor del componente a través de estos elementos, en forma *RequiredPort.serviceName*. Cuando una instancia de *Software_Component* sea declarada en un sistema, cada elemento de tipo *Required_Port* que se haya declarado en su descriptor deberá recibir la referencia concreta al modelo del puerto conectado.

El resto de la estructura de un *Software_Component* es similar a la del resto de contenedores: puede definir otros tipos de elementos declarados y de recursos agregados (los mismos que un módulo software) e incluir los modelos de las transacciones que maneja el componente, tanto agregadas como declaradas. Además, se hereda de *Software_Module* el parámetro *Host*, a través del que se asigna el modelo del nodo en que se instancia el componente.

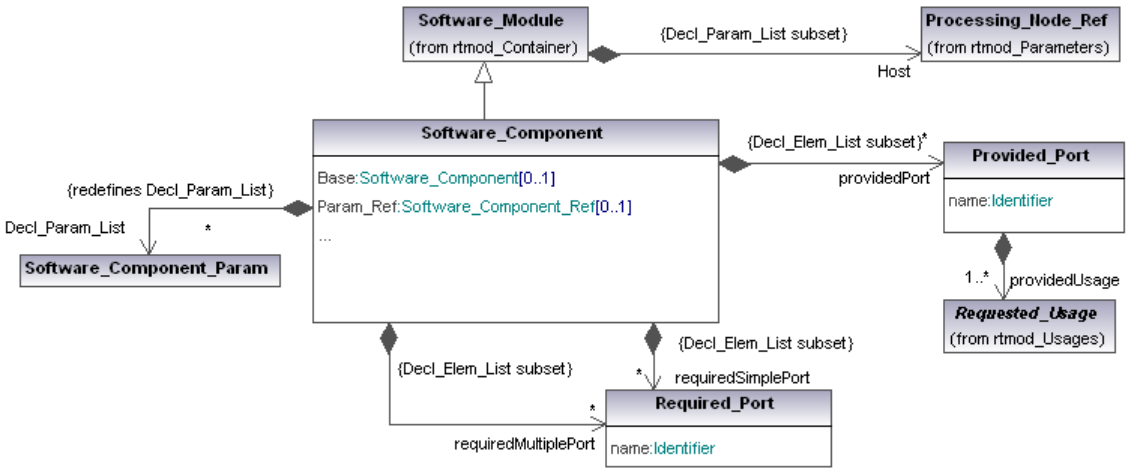


Figura 3.33: Descriptor de modelo CBS-MAST de un componente software

La figura 3.34 muestra la declaración de una instancia de componente. Además de recibir valores para los parámetros que declare el correspondiente descriptor y el parámetro por defecto *Host*, se le han de asignar las referencias a los modelos de los puertos conectados a sus puertos requeridos. Para ello se utilizan elementos de tipo *Required_Port_Assignment*, a través de los que se asigna la referencia a un puerto ofrecido por otro componente (en el caso de conexiones directas entre componentes) o a un conector (en el caso de conexiones realizadas a través de conectores).

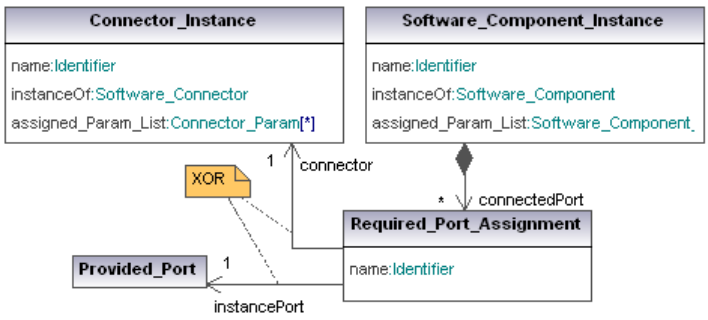


Figura 3.34: Declaración de instancias de componentes y conectores en CBS-MAST

El otro nuevo tipo de contenedor, *Software_Connector*, se muestra en la figura 3.35 y nos permite formular descriptores de modelo de conectores software. Un conector se define como un tipo especial de módulo software, que se intercala entre un componente cliente y un componente servidor para implementar las invocaciones entre ellos. Las características de este descriptor son las siguientes:

- Define un parámetro por defecto, *serverPort*, de tipo *Provided_Port_Ref*, a través del cual recibe la referencia al puerto del componente servidor sobre el que realiza las invocaciones.
- Los únicos elementos que un conector software puede declarar como públicos son los modelos de los servicios correspondientes a la interfaz que implementa. Aún en el caso más trivial, estos servicios implican trasladar la invocación al componente servidor, por lo que no pueden ser modelados como operaciones simples. Por ello, el tipo de elementos que un conector puede declarar se restringe a *Simple_Job*.

En muchos de los casos el conector tiene además carácter distribuido, encapsulando el mecanismo necesario para ejecutar una invocación remota entre puertos de componentes. Para modelar este caso se define un nuevo tipo de contenedor, *Distributed_Software_Connector*, que añade dos nuevos parámetros por defecto:

- *RemoteHost*: Referencia al modelo del nodo en el que se instancia el componente servidor, al que el conector transmite la invocación.
- *CommNet*: Referencia al modelo de la red de comunicaciones utilizada para realizar el envío de la invocación.

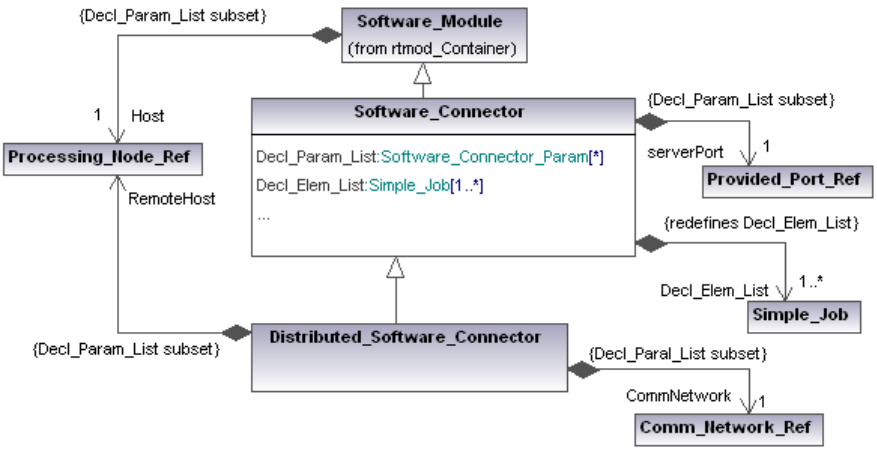


Figura 3.35: Descriptor de modelo CBS-MAST de un conector software

El último aspecto en que es necesario particularizar Mod-MAST para dar soporte al desarrollo de aplicaciones basadas en componentes es en la propia descripción del sistema a analizar. El metamodelo correspondiente al modelo de tiempo real de un sistema basado en componentes se muestra en la figura 3.36. La estructura es similar a la de un sistema basado en módulos, pero en este caso los únicos elementos que se utilizan para formar aplicaciones son instancias de componentes y de conectores, tanto locales como distribuidos.

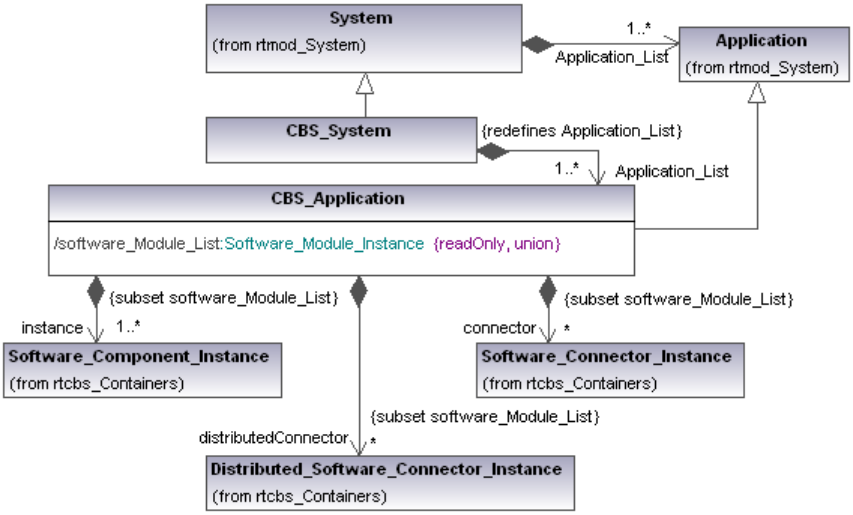


Figura 3.36: Definición de modelo de sistema en CBS-MAST

3.4. Generación del modelo MAST de aplicaciones descritas con RT-D&C

La información contenida en el modelo RT-D&C de una aplicación se utiliza en las tareas que se añaden al proceso de configuración y despliegue de una aplicación debido a su naturaleza de tiempo real. La tarea principal consiste en analizar el comportamiento temporal de la aplicación, para en base a dicho análisis, obtener una configuración que asegure el cumplimiento de los requisitos temporales. Las extensiones que se han definido en RT-D&C no presuponen ninguna metodología de modelado concreta. Cuando el proceso se adapte a una metodología determinada, será necesario desarrollar las herramientas que generan el modelo final de la aplicación a partir de la información disponible en los descriptores RT-D&C de la aplicación y de los componentes que la forman.

En nuestro caso, el análisis del comportamiento temporal se realiza utilizando las herramientas que se proporcionan en el entorno MAST, por lo que es necesario obtener un modelo de la aplicación acorde a dicho entorno. El proceso que se lleva a cabo para obtener el modelo MAST analizable de una aplicación basada en componentes y desarrollada en base a la especificación RT-D&C, es el que se muestra en la figura 3.37. De nuevo, aunque las herramientas actuales no se hayan desarrollado aplicando estrictamente técnicas MDA, sí se ha tratado de seguir este enfoque, de forma que se facilite su posterior adaptación a ellas. En base a ello, el modelo final se obtiene a través de una serie de transformaciones de modelos intermedios, que se describen a continuación.

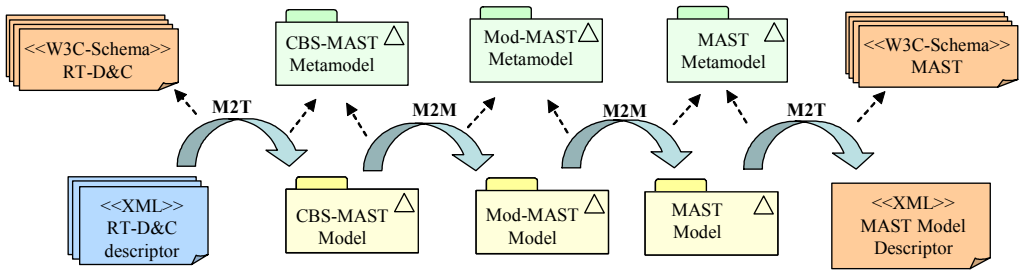


Figura 3.37: Obtención del modelo MAST de una aplicación descrita con RT-D&C

Transformación RT-D&C a CBS-MAST

En primer lugar, a partir de las descripción RT-D&C del sistema, se obtiene su correspondiente modelo CBS-MAST, de acuerdo al metamodelo visto en la sección anterior. La figura 3.38 muestra a través de qué descriptores RT-D&C se obtiene la información necesaria para declarar las instancias de modelo de cada uno de los elementos que forman el sistema. El modelo de un sistema en un determinado contexto de ejecución queda definido por:

- El modelo de la plataforma, que en este caso se obtiene a través del fichero de descripción del dominio de ejecución (fichero.tdm.xml). A través de él se conocen las instancias de modelo de nodos y redes a declarar, la referencia a sus correspondientes descriptores de modelo y los valores a asignar a sus parámetros (asignados en el *Domain* a través del atributo *rtConfigProperty*).
- El modelo de la situación de tiempo real de cada aplicación que forma parte del sistema y que es objeto del análisis. Para ello se requiere:
  - El modelo estructural de cada una de las aplicaciones, que se extrae del fichero que describe su correspondiente plan de despliegue (fichero.cdp.xml). Del plan de despliegue se obtiene la información necesaria para declarar la instancia de modelo de cada instancia de componente y de conector, asignar las referencias a sus correspondientes descriptores y asignar valores concretos a sus parámetros (los asignados a cada instancia a través del atributo *rtConfigProperty*).
  - El modelo de carga de trabajo de cada aplicación en el contexto de ejecución correspondiente. Se corresponde con cada uno de los elementos de tipo *RTWorkloadInstance* que se declaran en los ficheros que definen las posibles cargas de trabajo de cada aplicación (fichero .wld.xml).

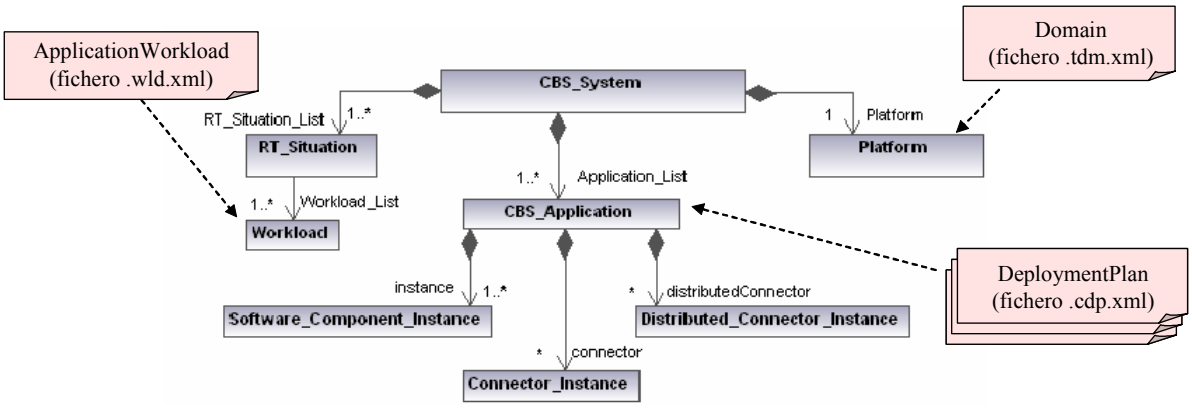


Figura 3.38: Descripción del modelo de un sistema de tiempo real basado en componentes

A partir de la información disponible en los ficheros RT-D&C, la herramienta de transformación añade al modelo CBS-MAST las correspondientes instancias de modelo de componentes, conectores, elementos de plataforma y transacciones. En la figura 3.39 se muestra un fragmento del modelo CBS-MAST que correspondería al sistema utilizado como ejemplo a lo largo del capítulo. Los descriptores que se utilizan como entrada en este caso son el plan de despliegue que se mostraba en la figura 3.28 (página 113), la carga de trabajo descrita en la figura 3.31 (página 116) y el modelo de dominio mostrado en la figura 3.23 (página 109). Cada instancia de modelo incluida en el sistema *ScadaDemoSystem*, como *engine* por ejemplo, referencia a su correspondiente descriptor de modelo, *AdaScadaEngine.rtm.xml* en este caso, que se encuentra almacenado en el repositorio. Los descriptores correspondientes a conectores, como *AnalogIOAdaCCMRTEPConnection.rtm.xml*, son generados por herramientas especializadas en base a la información de configuración de conexiones incluida en el plan de despliegue, y posteriormente almacenados también en el repositorio.

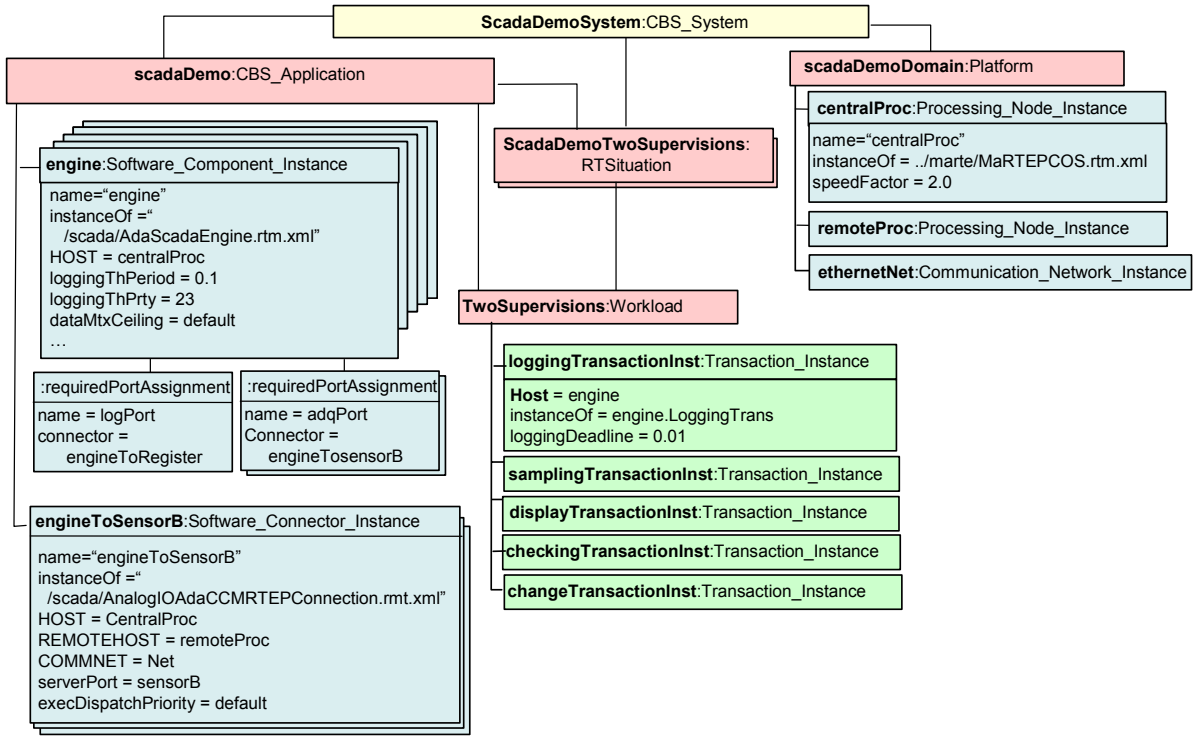


Figura 3.39: Ejemplo de modelo CBS-MAST de una aplicación

Transformación CBS-MAST a MAST

El objetivo del modelo CBS-MAST es obtener a través de él, el modelo transaccional MAST del sistema, por composición de los elementos que lo forman. En lugar de aplicar una transformación directa, que implica una mayor complejidad, se ha optado por realizarla en dos pasos, reutilizando transformaciones previamente definidas.

Primero, el modelo CBS-MAST se transforma en el modelo Mod-MAST del sistema. Ambos modelos son muy similares, salvo que en el caso de Mod-MAST los elementos de modelado utilizados reflejan la estructura modular de la aplicación desde un punto de vista más genérico que CBS-MAST. Las aplicaciones no están basadas en componentes reutilizables conectados a través de sus puertos, sino simplemente en módulos software genéricos que pueden presentar dependencias explícitas entre sí (formuladas a través de parámetros en el correspondiente

descriptor). Por tanto, la transformación de CBS-MAST a Mod-MAST consiste básicamente en transformar cada instancia de componente y conector en una instancia de módulo software genérico. Las reglas más relevantes para llevar a cabo esta transformación son las siguientes:

- Cada instancia de componente y de conector se transforma en una instancia de módulo software con idéntico nombre e idéntica asignación de parámetros. En la figura 3.40 se muestra la transformación de la instancia *engine* y el conector *engineToSensorB* a instancias de módulos software.
- En el caso de las instancias de componentes, las asignaciones de puertos requeridos se transforman en parámetros que se añaden a la correspondiente instancia de módulo, como ocurre en la instancia *engine* con los puertos *logPort* y *adqPort*. El valor que se asigna al parámetro es la referencia a la instancia de módulo correspondiente.
- Los descriptores a los que referencia cada instancia de modelo deben ser transformados también para eliminar su estructura interna basada en puertos. Por ejemplo, los servicios que estaban agrupados en estructuras de tipo *Provided_Port* dentro del descriptor de un componente pasan a ser elementos de primer nivel declarados en el descriptor del módulo, con sus nombres transformados a *nombrePuerto.nombreServicio*. De este modo se pueden distinguir servicios iguales pero ofrecidos a través de diferentes puertos.

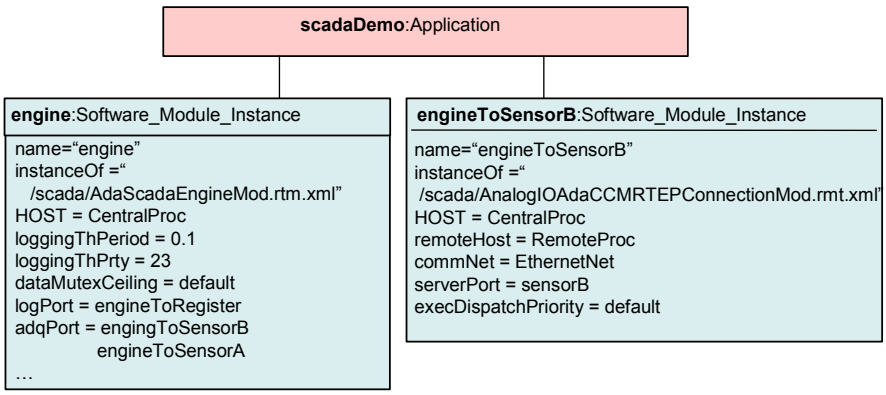


Figura 3.40: Ejemplo de transformación de instancias CBS-MAST a instancias Mod-MAST

En el segundo paso, el modelo Mod-MAST, basado exclusivamente en módulos software, es transformado en su correspondiente modelo MAST. Esta transformación y las reglas en las que se basa ya se expusieron en el capítulo 2.

Del análisis de este modelo final se podrán extraer los diferentes valores de configuración que han de ser asignados a componentes, conexiones y elementos de la plataforma, de manera que se asegure el cumplimiento de los requisitos de tiempo real cuando la aplicación sea ejecutada. Para ello será necesario desarrollar una herramienta de configuración, que como muestra la figura 3.41, genere el plan de despliegue final de la aplicación, totalmente configurado, en base a los resultados obtenidos del análisis del modelo MAST y a los descriptores RT-D&C de la aplicación. Esta herramienta depende de la tecnología de componentes sobre la que se aplique el proceso, pues de ella depende el modo de configurar la planificación.

En caso de que no se encuentre ninguna combinación de valores que verifique la planificabilidad de la aplicación, será responsabilidad del *planner* realizar modificaciones en el plan de despliegue de la aplicación para hacer posible el cumplimiento de los requisitos temporales. En el capítulo 5 se muestran algunas estrategias de diseño que pueden ser aplicadas con este objetivo.

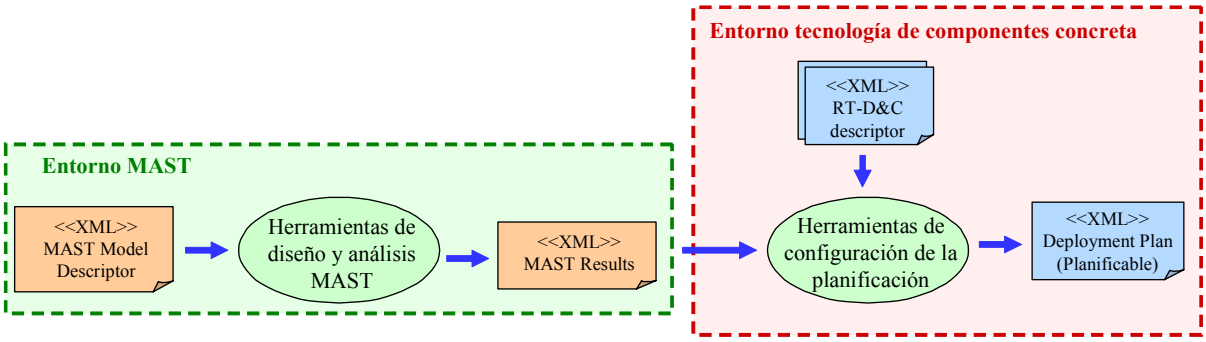


Figura 3.41: Configuración de la planificación de una aplicación RT-D&C

Un aspecto muy importante de toda esta secuencia de transformaciones es la trazabilidad. Los resultados del análisis del modelo transaccional final son revertidos en el plan de despliegue de la aplicación. Es necesario por tanto, establecer unas reglas muy claras de transformación para cada elemento, de modo que en la fase final se pueda acceder a los valores de configuración correctos. Por ejemplo, en el caso del componente *ScadaEngine*, la propiedad *dataMtx.ceiling* que se utiliza para dar valor al techo de prioridad del recurso compartido *dataMtx*, puede ser recalculada por las herramientas que analizan el modelo final del sistema. Como se muestra en la figura 3.42, para conocer el valor final a asignar se aplican las siguientes transformaciones:

- desde la declaración de la instancia *engine* en el plan de despliegue se genera la correspondiente declaración de instancia de modelo en el modelo CBS-MAST, con idéntico nombre y con un parámetro denominado *dataMtxCeiling* (correspondiente a la propiedad de tiempo real declarada en el descriptor de la implementación *AdaScadaEngine* y mapeada al valor de *dataMtx.ceiling*).
- desde la instancia de modelo del componente en CBS-MAST se genera la correspondiente instancia de módulo software en el modelo Mod-MAST, que declara un parámetro del mismo nombre.
- Este elemento da lugar al elemento *engine.dataMtx* de tipo *InmediateCeilingResource* en el modelo MAST, cuyo parámetro *ceiling* recibe de entrada el valor que se asignó en el plan de despliegue, pero puede ser recalculado por las herramientas.

De este modo, el valor que debe ser asignado a la propiedad *dataMtx.ceiling* en el plan de despliegue final es el valor del atributo *ceiling* del elemento *engine.dataMtx* obtenido por las herramientas de análisis MAST.

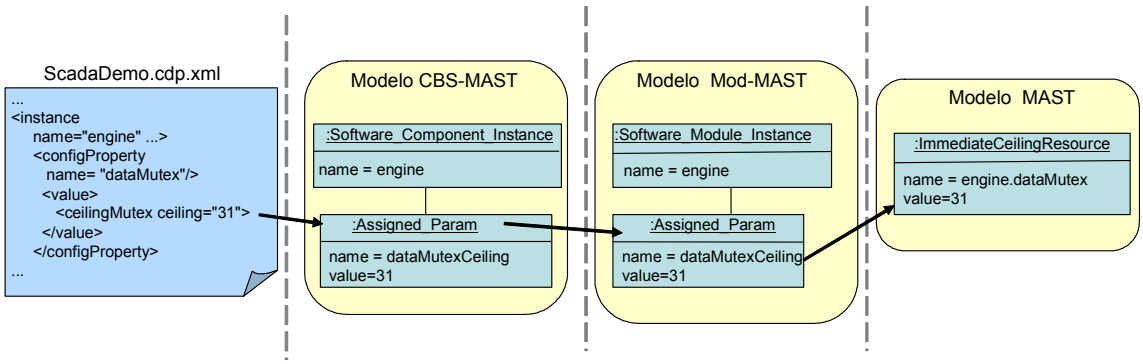


Figura 3.42: Reasignación automática de valores a propiedades de configuración

3.5. Conclusiones

Con la extensión de tiempo real propuesta en este capítulo se dota a la especificación D&C de capacidad para ser utilizada en el desarrollo de aplicaciones basadas en componentes de tiempo real. En este tipo de aplicaciones es necesario añadir a las fases típicas del proceso de desarrollo nuevas tareas que verifiquen el cumplimiento de los requisitos temporales de la aplicación. Para ello es necesario contar con un modelo del comportamiento temporal de la aplicación, que sirva de entrada a las herramientas de análisis de tiempo real. Este modelo, siguiendo el principio de opacidad básico en la estrategia de componentes, debe ser generado de manera oculta a los diseñadores de la aplicación, que únicamente manejan los metadatos que se incluyen en las descripciones de los componentes.

La extensión propuesta define los nuevos metadatos que se añaden a componentes y elementos de la plataforma para poder llevar a cabo ese proceso de análisis del comportamiento temporal de una aplicación de forma opaca. Los metadatos que se definen son de tres tipos:

- Metadatos asociados a la descripción de los componentes software reutilizables. Son establecidos por los diseñadores de los componentes y utilizados por los diseñadores de las aplicaciones con dos objetivos:
  - Decidir si la inclusión de un componente en una aplicación da lugar a una aplicación con comportamiento temporal predecible.
  - Analizar el comportamiento temporal de las aplicaciones que se diseñan, en base a la generación de un modelo de la aplicación creado por composición de los metadatos de cada elemento que forme parte de ella.
- Metadatos asociados a la descripción de los elementos de la plataforma. Su objetivo es aportar la información acerca de la capacidad de las plataformas y otras de sus características temporales, que se necesitan para obtener el comportamiento temporal de las aplicaciones ejecutadas en ellas.
- Metadatos asociados a la descripción de las aplicaciones. Su objetivo consiste en dar capacidad a los diseñadores de la aplicación para configurar la plataforma y las instancias de los componentes a fin de que se cumplan los requisitos temporales especificados. Los valores utilizados para la configuración serán obtenidos del análisis del modelo de tiempo real de la aplicación.

Un aspecto importante de la extensión de tiempo real propuesta es que delimita los conocimientos de tiempo real que requiere de cada uno de los agentes que participan en el desarrollo de una aplicación de tiempo real:

- El *developer* debe ser un experto en tiempo real. Por un lado debe desarrollar código con capacidad de ofrecer respuestas con temporización predecible, y por otro, debe formular los modelos de tiempo real que representan dicho código, utilizando para ello metodologías de modelado de tiempo real especializadas.
- El *planner* que analiza y configura la aplicación de tiempo real debe ser experto en estrategias de planificabilidad. Debe conocer los aspectos del despliegue que afectan a la planificabilidad y como gestionarlos para llegar a ella. Sin embargo, supuesto que dispone de herramientas de diseño de tiempo real, no tiene que conocer las técnicas de análisis y diseño de tiempo real que utilizan las herramientas ni la metodología de modelado utilizada en los modelos.

- El *specifier* y el *assembler* sólo deben comprender los metadatos con los que se especifican los requisitos de tiempo real de las aplicaciones y la caracterización del comportamiento dinámico del entorno en el que opera la aplicación.

Con el fin de que pueda ser aplicada en diferentes entornos, la extensión se ha realizado a nivel PIM. Su especialización a modelos PSM se puede abordar desde dos puntos de vista complementarios pero independientes. Por un lado la tecnología de componentes, y por otro la metodología de modelado utilizada para elaborar los modelos de tiempo real de componentes, elementos de plataforma y aplicaciones. En nuestro caso la metodología de modelado utilizada es CBS-MAST, una extensión de Mod-MAST en la que se han definido nuevos elementos de modelado que permiten desarrollar y almacenar de forma independiente los modelos temporales de componentes software reutilizables.

Frente a otras tecnologías de componentes que usan metamodelos o lenguajes propios para incorporar la información sobre comportamiento temporal [MM08][BKR09], RT-D&C constituye una solución estándar, que puede ser adaptada a diferentes tecnologías y entornos de modelado. En las propuestas referenciadas, la información sobre comportamiento temporal se especifica de acuerdo al modelo propio de las herramientas de análisis que se van a utilizar, lo cual no favorece la extensibilidad, pues el metamodelo debería ser modificado si se modificase la estrategia de análisis. Además, en estas estrategias no se hace uso de los metadatos para adaptar el comportamiento del componente a la aplicación concreta, pues los modelos asociados a los componentes no son parametrizados. En la tecnología de componentes CIAO sí se utiliza la especificación estándar D&C para describir componentes y aplicaciones, e incluso se incorpora en ella información que permite configurar aspectos relacionados con el comportamiento temporal [WGS04][KG08]. Sin embargo, como ya se expuso en el capítulo 1, estos datos se incorporan directamente en el plan de despliegue sin estar basados en ninguna información aportada por los componentes, sino únicamente en el conocimiento que tiene el *planner* de la plataforma subyacente, basada en este caso en RT-CORBA.

4. Extensión de tiempo real de la tecnología de componentes CCM

Los conceptos propuestos en capítulos anteriores para el modelado y la especificación de aplicaciones basadas en componentes se han formulado independientemente de la tecnología de componentes subyacente. Sin embargo, su aplicación sólo es posible si se cuenta con una tecnología de componentes que ofrezca comportamiento predecible y que pueda utilizarse en sistemas de tiempo real. En este capítulo se propone una tecnología de componentes, que denominamos RT-CCM (*Real-Time Container Component Model*), que constituye un entorno para el desarrollo de aplicaciones de tiempo real basadas en componentes, y que se va a utilizar para validar el uso y consistencia del resto de soluciones expuestas en esta tesis.

RT-CCM surge como resultado de la búsqueda de una tecnología de componentes que cumpla los siguientes requisitos:

- Capacidad para construir aplicaciones con requisitos de tiempo real estricto. A tal fin, los componentes con los que se ensamblan las aplicaciones tienen que ofrecer servicios con tiempos de respuesta predecibles y analizables. La tecnología, a su vez, debe ofrecer mecanismos que proporcionen control sobre aquellos aspectos que hacen predecible el comportamiento temporal de una aplicación, tales como la concurrencia, la sincronización, etc.
- Compatibilidad con sistemas empotrados de perfil mínimo. La tecnología debe ser lo suficientemente ligera como para ser compatible con la limitación de recursos característica de los sistemas empotrados en los que tradicionalmente se ejecutan las aplicaciones con requisitos de tiempo real.
- Capacidad para ser ejecutada en plataformas distribuidas. Para asegurar la predictibilidad temporal en este caso, los mecanismos de comunicación utilizados en las interacciones entre componentes tienen que ofrecer comportamiento temporal predecible y analizable.
- Capacidad para utilizar diferentes mecanismos de comunicación entre componentes. Toda tecnología de componentes está implementada sobre una plataforma que proporciona un middleware de base, a través del que se realizan las interacciones entre componentes tanto locales como remotas (como CORBA en el caso de CCM). En este caso, aún tomando un determinado middleware como base, la tecnología debe permitir la interacción entre componentes a través de otros mecanismos, elegidos y configurados de acuerdo a los requisitos que exija cada aplicación.
- Soporte para el desarrollo de componentes de granularidad alta, con capacidad para ofrecer diferentes servicios agrupados en interfaces.

Ninguna de las tecnologías convencionales y más usadas en el mercado, como EJB o .NET satisfacen estos requisitos. No son compatibles con aplicaciones de tiempo real, ya que están destinadas a plataformas que no ofrecen ese tipo de comportamiento. Tampoco con sistemas empotrados, ya que requieren sistemas operativos, sistemas de ficheros, redes de comunicación, etc., incompatibles con la limitación de recursos que es habitual en estos sistemas.

Uno de los modelos de componentes abierto con mayor difusión es CCM, el modelo de componentes de CORBA [CCM]. Este modelo ofrece las ventajas de ser multiplataforma y multilenguaje; sin embargo, en su implementación estricta está basado en CORBA, dando lugar a implementaciones que resultan muy pesadas para sistemas empotrados. Debido a ello, OMG propuso la especificación LwCCM, *Ligthweight Corba Component Model* [CCM], en la que se reduce la complejidad de la especificación CCM a fin de que pueda ser implementada en sistemas de perfil mínimo. La tecnología de componentes CIAO [SDG07][DSG06][CIAO] representa una implementación del modelo LwCCM que además ofrece soporte para tiempo real al estar implementada sobre la versión de tiempo real del estándar CORBA, RT-CORBA [RTCRB]. Con la utilización de esta tecnología, deberíamos restringirnos al uso de CORBA, que en algunos casos sigue resultando demasiado pesado para el tipo de aplicaciones que se consideran, y lo que es más importante, no permitiría hacer configurable el mecanismo de comunicación elegido para cada conexión entre componentes. Por otro lado, el modelo de asignación de prioridades del estándar RT-CORBA no es suficientemente flexible para implementar con él algunas de las políticas de planificación que se utilizan en ciertas metodologías avanzadas para diseño de tiempo real, en concreto, en aquellas basadas en el modelo transaccional o reactivo de la aplicación [GG99].

Todas las empresas que han querido adoptar una tecnología de componentes de tiempo real han optado finalmente por desarrollar soluciones propias, adaptadas a sus propias necesidades. En el capítulo 1 se expusieron algunos ejemplos de este tipo de tecnologías, como Koala [OLK00] o Rubus [RUBCM]. Obviamente, estas tecnologías son tecnologías propietarias y por tanto no es aconsejable tomarlas como referencia para nuevas extensiones. Sin embargo, en esta tesis se decidió seguir una estrategia similar, esto es, desarrollar una tecnología de componentes propia, que tomando como punto de partida una tecnología de componentes abierta y bien conocida como es LwCCM, añade los elementos necesarios para verificar todos los requisitos que se han planteado. Sus características principales se exponen a lo largo de este capítulo.

4.1. Características generales de la tecnología RT-CCM

El modelo de programación que se propone en la especificación LwCCM resulta muy atractivo para el desarrollo de una tecnología de componentes. Se basa en el paradigma contenedor/ componente (*Container Component Model*), cuyo objetivo es hacer que el código de negocio de un componente sea independiente de la plataforma en la que va a ser ejecutado y del mecanismo de comunicación a través del que se produce la interacción con otros componentes. Con su uso se consigue que los desarrolladores del código de negocio puedan trabajar sin necesidad de ser expertos en la plataforma de ejecución. Como muestra la figura 4.1, siguiendo este modelo, una instancia de componente que se ejecuta en una determinada plataforma se compone de dos elementos:

- El código de negocio, que implementa la funcionalidad del componente y que es desarrollado sin conocer las características de la plataforma en que se va a instanciar ni la aplicación en la que el componente va a ser ensamblado.
- El contenedor, que proporciona al código de negocio acceso a los recursos de la plataforma para que pueda ejecutarse en ella e interaccionar con el resto de componentes. El código del contenedor depende tanto de las características del componente como de la naturaleza de la plataforma en que se ejecuta, y es generado de forma automática a partir del plan de despliegue de la aplicación y de la información proporcionada por los componentes en forma de metadatos.

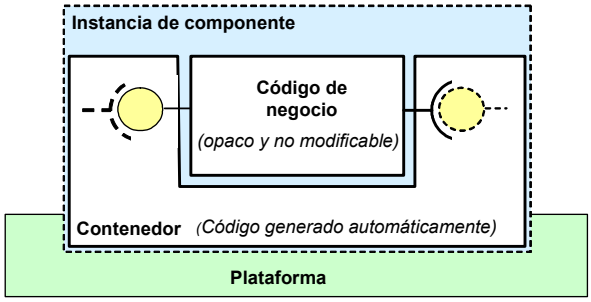


Figura 4.1: Modelo de programación basado en contenedor/componente

La tecnología que se propone, que hemos denominado RT-CCM (*Real-Time Container Component Model*), toma como punto de partida el modelo de componentes y el modelo de referencia propuestos en la especificación LwCCM, éste último basado en el modelo contenedor/componente, y los extiende con nuevos elementos que la capacitan para el desarrollo de aplicaciones distribuidas, configurables y de tiempo real. Dichas características, que se resumen en la figura 4.2, son las siguientes:

- Se dotan los contenedores de mecanismos que permiten controlar la concurrencia, sincronización y planificación de la ejecución de las actividades invocadas en los componentes, de manera que se haga predecible y configurable el comportamiento temporal de las aplicaciones en que se utilicen.
- Se elimina el uso de CORBA como mecanismo exclusivo para las comunicaciones entre componentes, utilizándose otros mecanismos con comportamiento temporal predecible y que requieran recursos más ligeros, compatibles con plataformas empotradas. La comunicación entre componentes se realiza a través de un tipo especial de componente, denominado conector, que engloba en su código interno los mecanismos de comunicación necesarios para que dos componentes interaccionen entre sí. El código del conector es generado de manera automática en base a la configuración de la conexión. Cuando en la plataforma existe un middleware de base por defecto, el conector se implementa en base a él; sin embargo, a través del plan de despliegue se puede establecer el servicio de comunicación en el que se ha de basar el conector utilizado en cada conexión entre instancias de componentes de la aplicación.

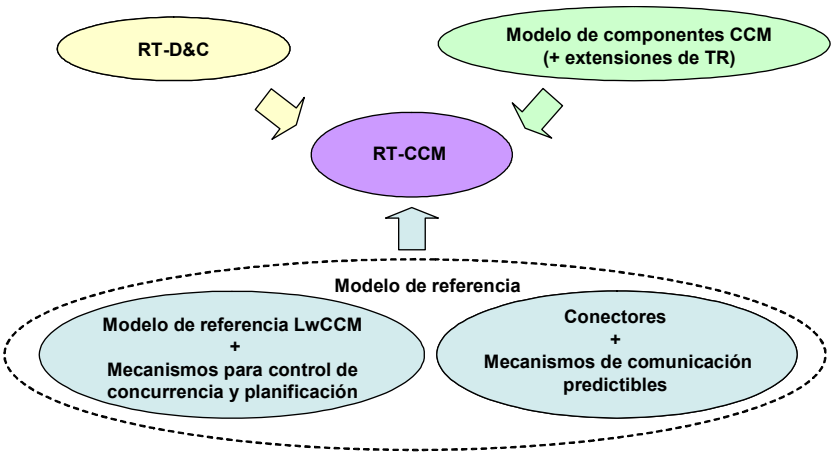


Figura 4.2: Principales características de la tecnología RT-CCM

Una tecnología de componentes debe definir la estrategia de especificación de los componentes y de las aplicaciones en base a la que los operadores humanos, y sobre todo las herramientas, gestionan la información asociada a un componente como elemento distribuible y/o a una aplicación. En la estrategia se han de definir los contenidos y los formatos de los documentos que se utilizan para localizar el código y para acceder a los metadatos que describen los componentes y las aplicaciones. En la tecnología RT-CCM se utiliza con este fin la especificación RT-D&C definida en el capítulo 3. Con las extensiones de tiempo real introducidas se dispone de información acerca del comportamiento temporal de los componentes y de las plataformas, que es utilizada durante el desarrollo de una aplicación para analizar su comportamiento de tiempo real o diseñar la aplicación de modo que se satisfagan sus requisitos de tiempo real. Los descriptores RT-D&C de componentes y aplicaciones constituyen también la entrada para las herramientas que generan el código de los contenedores de los componentes, así como los conectores necesarios para implementar la comunicación entre ellos.

En este capítulo se define un modelo PSM de RT-D&C adaptado a las características de la tecnología RT-CCM. Con el uso de RT-D&C, los procesos de desarrollo tanto de un componente como de una aplicación RT-CCM, se ajustan exactamente a los definidos en el capítulo 3. Constan de las mismas fases y agentes, los cuales a su vez poseen las mismas responsabilidades, por lo que no se repiten de nuevo en este capítulo. Aquellos aspectos característicos de RT-CCM, como el tipo de información que se incluye en la descripción de un componente o aplicación, el modo en que se desarrolla el código de negocio durante la fase de desarrollo, el modo de configurar la planificación de una aplicación, etc., es lo que se describe en el resto del capítulo.

4.2. Concepto de componente RT-CCM

Un componente RT-CCM se concibe como un componente software de granularidad alta, que ofrece una funcionalidad bien definida a través de sus puertos y que es reutilizable, esto es, ha sido desarrollado con independencia de la aplicación en la que pueda ser ensamblado.

Como muestra la figura 4.3, un componente RT-CCM se distribuye como un paquete que contiene elementos heterogéneos de información: ficheros de código, metadatos, modelos de comportamiento temporal, etc. La información que incluye el paquete es suficiente para que en el proceso de ensamblado, configuración y despliegue de cualquier aplicación en la que se utilice el componente:

- Se pueda decidir si el componente es de utilidad como parte de ella, tanto desde el punto de vista de la funcionalidad de negocio, como del comportamiento temporal.
- Se pueda verificar la compatibilidad del componente con la plataforma de ejecución.
- Se pueda asignar la configuración adecuada a la instancia del componente para que ofrezca la funcionalidad y el comportamiento temporal que se requiere en la aplicación.
- Una herramienta pueda generar el modelo de análisis temporal de la aplicación por composición de los modelos de los componentes que la forman, junto con los modelos de los recursos de la plataforma es que es ejecutada. Del análisis de este modelo podrá extraerse información que sirva para configurar los componentes y el entorno de ejecución, de modo que la aplicación verifique sus requisitos temporales.
- Una herramienta pueda generar el código del contenedor que permite ejecutar el código de negocio en el entorno de ejecución que se utilice. El código de negocio del componente será siempre manejado de forma opaca, esto es, incorporado en la aplicación sin ningún tipo de modificación.

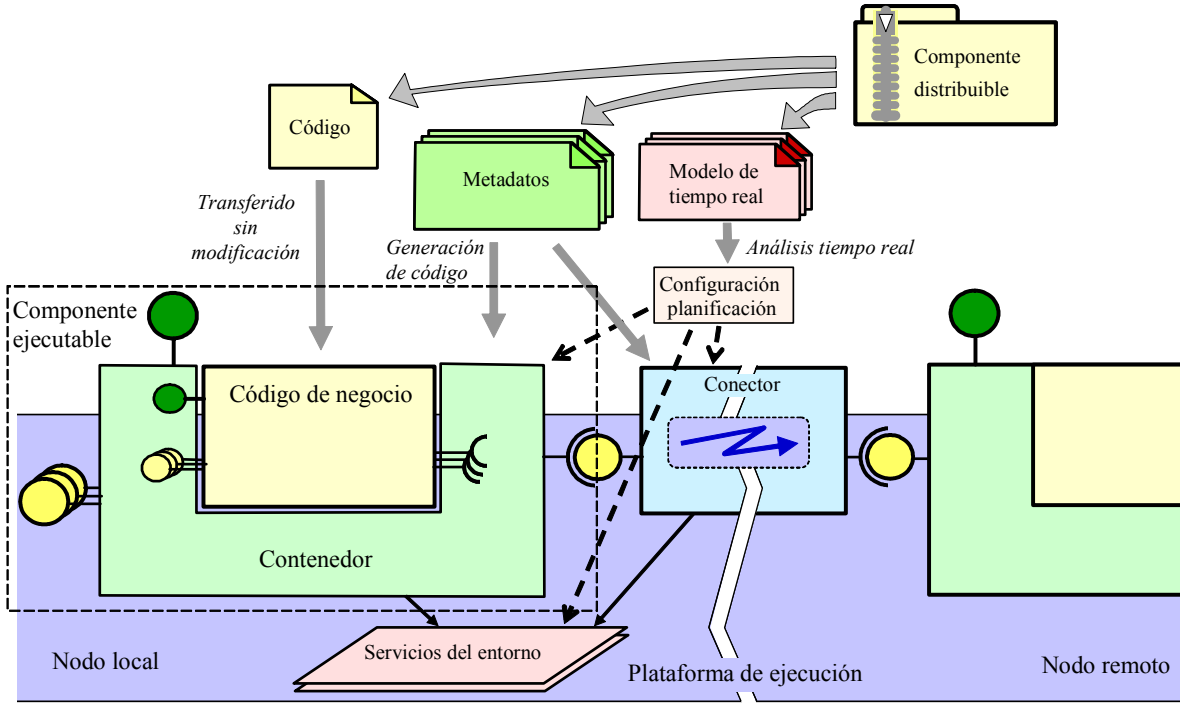


Figura 4.3: Elementos de un componente RT-CCM

- Una herramienta pueda generar el código y los modelos temporales de los conectores que implementan las conexiones del componente con otros componentes, haciendo uso de los servicios de comunicación disponibles en la plataforma.

Toda esta información se organiza siguiendo los formatos y modelos propuestos en la especificación RT-D&C. Los modelos temporales deben formularse con una metodología que permita llevar a cabo el proceso de composición que genera el modelo final de la aplicación. En nuestro caso se utiliza la metodología CBS-MAST, aunque podría utilizarse cualquier otra que ofrezca las características de opacidad y componibilidad adecuadas.

4.2.1. Especificación externa o modelo funcional

La funcionalidad externa de un componente RT-CCM se define a través de los puertos de interacción que el componente posee, tanto ofrecidos como requeridos. La naturaleza y nomenclatura de estos puertos se toma de la especificación CCM, existiendo cuatro tipos de puerto cuya representación gráfica se muestra en la figura 4.4:

- Facetas: Representan servicios, agrupados por interfaces, que el componente ofrece para que puedan ser invocados por componentes cliente.
- Receptáculos: Representan puntos de interacción a través de los que el componente hace uso de servicios ofrecidos por otros componentes que él necesita para implementar su funcionalidad. Un receptáculo puede ser simple, en cuyo caso se puede conectar con un único componente servidor, o múltiple, en cuyo caso puede conectarse con varios componentes servidores, debiendo ofrecer todos ellos la misma funcionalidad.
- Fuentes de eventos: Representan puntos de interacción que generan eventos de un determinado tipo. Los eventos están destinados, bien a otros componentes que se hayan subscrito como interesados en ellos, o bien a un canal de eventos.

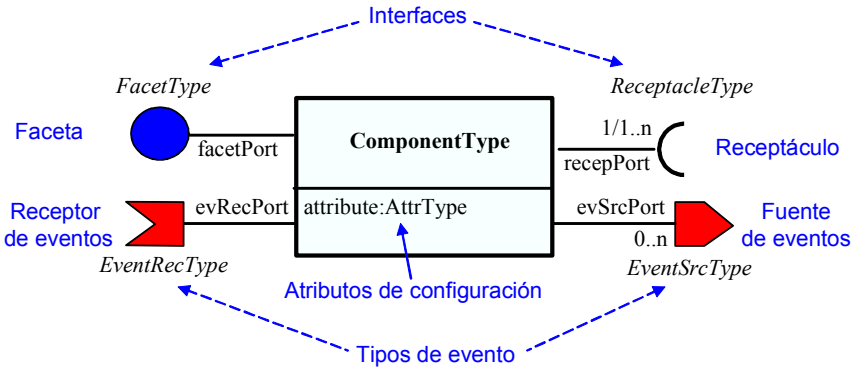


Figura 4.4: Elementos y representación gráfica de la especificación funcional de un componente RT-CCM

- Receptores de eventos: Representan puntos de interacción a través de los que se reciben eventos de un determinado tipo, en los que el componente se declara interesado.

Además de los puertos, cada componente puede definir un conjunto de atributos de configuración, a los que se asigna valor en la fase de configuración de la aplicación para adaptar el código de negocio del componente al contexto concreto en que vaya a ejecutar.

La funcionalidad de facetas y receptáculos queda definida unívocamente por la interfaz que implementan, mientras que en el caso de fuentes y receptores de eventos se define en base al tipo de evento que gestionan. Ambas (interfaces y tipos de eventos) se describen usando algún lenguaje de especificación de interfaces independiente del lenguaje de programación, en nuestro caso IDL [CORBA]. La conexión entre dos puertos de dos componentes podrá establecerse cuando ambos implementen la misma interfaz o gestionen el mismo tipo de evento.

En resumen, desde el punto de vista funcional, el modelo de componentes RT-CCM es el mismo que el de CCM. El aspecto característico de una especificación de componente en RT-CCM, consecuencia de su naturaleza de tiempo real, es que además se incluyen en su especificación aquellas características de tiempo real necesarias para decidir si con el uso del componente se va da lugar a una aplicación analizable, y para declarar la parte de la carga de trabajo de la aplicación que tiene su origen en él. La forma de declarar esta información en la interfaz externa de un componente RT-CCM es la misma que se introdujo en el capítulo anterior (sección 3.2.1), donde incluso se expuso un ejemplo para el caso del componente *ScadaEngine*.

4.3. Modelo de concurrencia en componentes RT-CCM

Uno de los aspectos clave en el desarrollo de aplicaciones de tiempo real es el diseño de la concurrencia. Las aplicaciones de tiempo real tradicionalmente se han concebido, diseñado y construido como un conjunto de tareas concurrentes que se inician en respuesta a eventos del entorno o a eventos temporizados generados por el reloj. Estas tareas ejecutan las actividades que constituyen la respuesta del sistema, debiendo satisfacer los requisitos temporales establecidos en la especificación de la aplicación. Dado que son tareas que se ejecutan concurrentemente y que hacen uso de los recursos que existen en la plataforma, es necesario utilizar los mecanismos de sincronización adecuados para garantizar la seguridad en el acceso a los recursos y para que las suspensiones y bloqueos producidos en dichos accesos sean compatibles con los requisitos temporales de la aplicación. En procesos de diseño tradicionales [GOM84][WM85], así como en los basados en objetos [DOU99][GOM00], el diseño de la

concurrencia comienza con la identificación de las tareas que se ejecutan en el sistema. Posteriormente, en el diseño lógico, las actividades que constituyen las tareas se organizan de acuerdo a alguna arquitectura software dirigida, bien por los dominios de aplicación, o bien por los subsistemas en que se particiona el sistema. De este modo, la tarea se convierte en la unidad básica de diseño y de generación del código de la aplicación.

Esta estrategia no puede ser aplicada en metodologías de diseño de aplicaciones de tiempo real basadas en componentes, en las que las aplicaciones se construyen ensamblando componentes opacos. Las tareas, y por tanto las actividades que ejecutan, son parte del código interno de los componentes. De acuerdo con el principio de opacidad del paradigma de componentes, a menos que se hayan hecho explícitas como parte de la especificación externa del componente, no pueden ser identificadas ni gestionadas por el diseñador de la aplicación.

Para hacer compatibles los principios del diseño de tiempo real, que requieren conocimiento y gestión global de los threads del sistema, con los principios del paradigma de componentes, que imponen la opacidad en el manejo del código de negocio, es necesario introducir restricciones en el diseño del código de los componentes, así como establecer patrones de concurrencia adecuados, que permitan la identificación, control y configuración automática de los threads y de los mecanismos de sincronización utilizados por los componentes, siempre sin acceder a su código de negocio. La solución adoptada consiste en transferir al contenedor la responsabilidad de la creación, control y configuración de los threads y de los mecanismos de sincronización y planificación utilizados en un componente, ya que al ser generado el contenedor mediante herramientas, los patrones de diseño adecuados se pueden introducir en él de forma sistemática. Además, como la plataforma es conocida cuando se elabora el contenedor, se pueden incorporar en su código los threads y mecanismos de sincronización nativos disponibles en ella.

Esta solución introduce una restricción importante en la libertad de diseño del código de negocio del componente, ya que impide la creación directa de threads y mecanismos de sincronización dentro de él. Sin embargo, ofrece la ventaja de que con ella el diseñador del componente no necesita conocer la plataforma de ejecución para elaborar el código de negocio, y así, el mismo componente puede ser instanciado en diferentes plataformas.

En la tecnología RT-CCM, con el objetivo de tener pleno control sobre el número y características de los threads que se ejecutan en una aplicación, es el entorno, a requisito del contenedor, el que crea los threads que necesita el código de negocio para ejecutar las actividades de una aplicación. Estos threads se crean además de forma estática durante la fase de lanzamiento de la aplicación, evitando así la falta de predictibilidad que introduce la creación dinámica de threads, que dificulta además la planificación de las tareas.

4.3.1. Origen y gestión de los threads

La construcción de una aplicación como un ensamblado de componentes es una vista estructural estática de la modularización del código, que no describe por sí misma el flujo de control de la aplicación. Para comprender y poder gestionar la ejecución, hay que identificar los threads que las instancias de los componentes aportan, las actividades que ejecuta cada uno y la evolución de su ciclo de vida.

En función de los threads que aportan los componentes cuando se instancian, éstos se clasifican en dos tipos:

- Componentes pasivos: Son aquellos cuya instanciación no inicia ningún thread. Los threads que ejecutan su código proceden siempre de los componentes que invocan sus servicios.

- Componentes activos: Son aquellos que en su especificación declaran uno o más threads. Cuando el componente se instancia, el entorno de ejecución, a través del contenedor, crea los threads requeridos, que ejecutan el código del componente. Los componentes activos son los puntos de inicio de la ejecución del código de la aplicación. Para que una aplicación se ejecute, debe tener al menos una instancia de componente activo en su estructura.

Existe un tercer tipo de componentes, denominados componentes cliente, que son aquellos con capacidad para iniciar transacciones de negocio. Una transacción se inicia en un thread de un componente, y en el caso general, la transacción se compone de actividades que se ejecutan en múltiples threads iniciados en diferentes componentes. La naturaleza de cliente de un componente está relacionada no con la capacidad de iniciar threads, sino con la capacidad de controlar los parámetros de planificación de los threads desde su propio código de negocio. Todo componente cliente debe ser activo, pues necesita al menos un thread para iniciar una transacción; sin embargo, un componente activo no tiene por qué tener naturaleza de componente cliente. Más adelante se explica más en detalle este concepto.

Hay múltiples causas por las que en un componente necesita que en él se inicie un thread:

- Thread principal: Son threads que se inician al instanciarse el componente. Su función es construir o inicializar elementos dinámicos del componente y su duración puede ser limitada o permanecer durante toda la vida de la aplicación. En el caso de aplicaciones secuenciales corresponde al único thread que ejecuta todo el código.
- Atención a dispositivos de entrada/salida activos: Son threads que gestionan los requisitos asíncronos que realizan dispositivos activos de entrada/salida a través de la generación de interrupciones hardware. Suelen ser threads que se crean en la activación del componente, se suspenden a la espera de que se produzca la interrupción hardware del dispositivo, y cuando ésta ocurre, ejecutan las actividades de atención al dispositivo.
- Atención a dispositivos de entrada/salida pasivos: Son threads periódicos que consultan el estado de dispositivos de entrada/salida pasivos. Cuando detectan que el dispositivo requiere atención, ejecutan las actividades correspondientes.
- Actividades temporizadas: Son threads que se inician en respuesta a eventos temporizados producidos por el reloj del sistema. Ejecutan actividades que la lógica del componente requiere que sean ejecutadas periódica o esporádicamente.
- Actividades de control: Son threads que se utilizan para gestionar la evolución del estado de un componente. Se presentan frecuentemente en componentes que tienen un modelo de actividad interna basado en una máquina de estados. Su actividad consiste en hacer cambiar el estado en función de las interacciones externas que recibe el componente, o ejecutar la actividad temporal o persistente del estado actual en que se encuentra.
- Atención de eventos: Son los threads que se asocian a los puertos de atención de eventos del componente y su función es ejecutar las actividades que corresponden a la recepción de cada evento.
- Flexibilidad de planificación: Son threads que utilizan internamente los componentes para ejecutar con diferentes prioridades las actividades de un mismo servicio. Estos threads suelen ser utilizados cuando los plazos impuestos en la respuesta del servicio se refieren a un estado intermedio del mismo, lo que hace que cambie la urgencia de ejecución de las actividades previas y posteriores a dicho estado.

- Ejecución asíncrona: Son threads que se utilizan para ejecutar asíncronamente ciertas actividades de los servicios del componente que son ejecutadas en concurrencia con la línea principal de la respuesta.
- Atención de comunicaciones: Son threads que se incluyen para gestionar los mensajes que se reciben o los mensajes que se envían a través de los canales de comunicación asíncronos. Los threads de bajo nivel suelen estar asociados a los servicios de comunicaciones de la plataforma de ejecución y no son parte de los componentes software. En la tecnología RT-CCM los threads de comunicación de alto nivel (nivel de aplicación) son tratados exclusivamente por los conectores, que incorporan en su código los mecanismos de comunicación entre componentes de negocio.

Independientemente de que sean activos o pasivos, todos los componentes son intrínsecamente concurrentes, esto es, en cualquier instante pueden existir múltiples threads que estén ejecutando concurrentemente secciones de su código. Estos threads pueden haber sido iniciados en los propios componentes (en el caso de componentes activos), o pueden pertenecer a componentes externos que están ejecutando la invocación de uno de sus servicios. Como se describirá en la sección 4.3.2, cada componente debe aportar los mecanismos de sincronización que garantizan una ejecución segura de su código.

Aunque en algunos lenguajes de programación (como Ada o Java) los threads pueden declararse y controlarse mediante sentencias primitivas del lenguaje, en muchos casos y especialmente en los sistemas de tiempo real, su gestión se realiza por invocación directa de servicios del sistema operativo. Con el objetivo de que el código de negocio de un componente pueda ser desarrollado de forma totalmente independiente a la plataforma de ejecución, en RT-CCM el código de negocio de un componente es pasivo, esto es, en él no se crean ni gestionan directamente los threads que el componente necesita para ejecutar su código.

Para desarrollar componentes activos se propone un mecanismo que permite al desarrollador definir los threads que necesita el componente y establecer las actividades que debe ejecutar cada uno. Por cada thread requerido en el diseño del componente, el desarrollador declara un puerto especial, denominado puerto de activación, como parte de los metadatos del componente. Además, el desarrollador encapsula el código a ejecutar por dicho thread en el método ofrecido por el puerto. Cuando el componente es instanciado, su contenedor (cuyo código es generado por una herramienta que conoce la plataforma en que se instancia y los requisitos de threads declarados por el componente) requiere a la plataforma de ejecución, por cada puerto de activación declarado, la creación de un thread que ejecute el método asociado al puerto. Con el objetivo de poder planificar la aplicación, el contenedor debe conocer y asignar los parámetros de planificación con los que debe comenzar la ejecución de cada thread.

En RT-CCM el ciclo de vida de un thread que se inicia en un componente está supeditado al estado de ejecución del componente:

- Cuando un componente se instancia, el contenedor crea todos los threads que se han declarado en el componente.
- Cuando el componente se activa, los threads declarados en él se activan. En el caso de la primera activación esto implica el comienzo de la ejecución del thread.
- Cuando un componente se suspende, la suspensión efectiva del componente sólo se confirma si todos los threads definidos en el componente han sido suspendidos.
- Cuando un componente se desinstancia, la desinstanciación efectiva del componente sólo se produce cuando todos los threads declarados en el componente han finalizado su actividad.

Esta sincronización se impone a fin de garantizar que los threads internos de un componente sólo se ejecutan cuando el estado del componente y sus conexiones con otros componentes son seguras, y su actividad no puede conducir a situaciones de error.

4.3.2. Estrategia de gestión de la sincronización

El código de negocio de un componente se desarrolla de forma independiente de la aplicación en que sea ejecutado, por lo que el desarrollador del componente no conoce el nivel de concurrencia con el que se pueden invocar los servicios que ofrece. Por tanto, aunque el código de negocio sea pasivo, debe estar diseñado de modo que se garantice la operación segura de los threads que se encuentren ejecutando código del componente en un instante dado. En una aplicación basada en componentes las sincronizaciones entre threads sólo se llevan a cabo entre los threads que se encuentran ejecutando el código de un mismo componente.

En RT-CCM se dota a los componentes con dos tipos de mecanismos de sincronización:

- Mutex: Mecanismo de sincronización que permite garantizar que múltiples threads concurrentes puedan ejecutar secciones críticas de código en régimen de exclusión mutua. Habitualmente se utiliza para garantizar la actualización segura de estructuras de datos del componente que deben ser leídas y escritas por los threads que ejecutan concurrentemente su código.
- Variable de condición: Mecanismo de sincronización que permite que un thread que ejecuta el código del componente se suspenda hasta que otro thread que también opera sobre el componente lo active. Habitualmente se utiliza para suspender durante un tiempo indefinido un thread que opera en el componente hasta que éste alcance un determinado estado o reciba un determinado evento.

Al igual que ocurre con la gestión de los threads, aunque algunos lenguajes de programación concurrente disponen de sentencias primitivas para gestionar la sincronización, en el caso general los mecanismos de sincronización son proporcionados por el sistema operativo. Por ello, y con el objetivo de mantener el principio de que el código de negocio de un componente sea independiente de la plataforma de ejecución, se impide la instanciación directa en el código de negocio de mecanismos de sincronización y se delega en el contenedor para que los gestione.

A tal fin, se define en la tecnología un tipo especial de puerto requerido denominado puerto de sincronización. Por cada mecanismo de sincronización que se necesita utilizar, el desarrollador de una implementación de componente declara un puerto de sincronización del tipo adecuado (mutex o variable de condición) en los metadatos del componente. En ambos casos se trata de puertos requeridos, de modo que el código de negocio puede ejecutar fragmentos de código en modo protegido o suspender un thread en un punto de ejecución hasta que otro thread lo señalice, en base a la invocación de las operaciones definidas en estos puertos.

La herramienta que genera el código del contenedor de un componente en el contexto del plan de despliegue de una aplicación, conoce la declaración de los puertos de sincronización del componente que está generando. En base a esta información incluirá en el código la invocación de los servicios de la plataforma que crean los correspondiente objetos de sincronización y le proporcionará a los componentes acceso a ellos. Con el objetivo de que se pueda configurar la aplicación de modo que satisfaga sus requisitos temporales, el contenedor debe poder conocer y asignar los parámetros de planificación de los objetos de sincronización que gestiona.

4.3.3. Estrategia de gestión de la planificación

La función básica que desempeña cualquier componente es la de ofrecer un conjunto de servicios que pueden ser invocados por otros componentes, esto es, actuar como componente servidor. Siendo el código de negocio de un componente RT-CCM pasivo, las invocaciones recibidas en dichos servicios serán siempre ejecutadas por un thread externo, generado en alguno de los componentes involucrados en la transacción dentro de la que se realiza la invocación. En estos casos, como ocurre en el diseño de aplicaciones de tiempo real basadas en arquitecturas cliente-servidor, se requieren estrategias de asignación de parámetros de planificación que definan los parámetros con los que debe ejecutarse cada invocación recibida en un servicio. Entre las estrategias más comúnmente utilizadas en este tipo de aplicaciones se encuentran las definidas en RT-CORBA. Aunque se definen únicamente para la asignación de prioridades, podrían ser igualmente utilizadas con otros tipos de parámetros de planificación:

- Prioridad declarada por el servidor (*Server_Declared)*: Cada servidor (componente en este caso) tiene declarada la prioridad con la que se ejecutan los servicios que ofrece, con independencia del origen de la invocación.
- Prioridad propagada por el cliente (*Client_Propagated*): La prioridad a la que se ejecuta un servicio es la del cliente que realiza la invocación, que se propaga a lo largo de toda la cadena de invocaciones a la que da lugar. Si el sistema es heterogéneo y en cada nodo hay diferentes rangos de prioridades, esta estrategia requiere la existencia de una escala de prioridad global de sistema, sobre la que se mapean las prioridades de cada nodo.

Ambas estrategias permiten planificar muchas de las situaciones de tiempo real que se pueden presentar en una aplicación. Para mostrar las ventajas e inconvenientes de ambas, planteamos un ejemplo cuyo modelo reactivo se muestra en la figura 4.5. Consiste en dos transacciones concurrentes lanzadas por sendos clientes, que implican la ejecución de diferentes servicios en diferentes componentes. Centramos nuestra atención en el servicio *ServiceB*.

- Si los requisitos de tiempo real que se imponen en este sistema se asocian a la finalización de las transacciones, el sistema puede hacerse planificable asignando una mayor prioridad a las ejecuciones de *ServiceB* correspondientes a la transacción con mayor frecuencia de disparo y menor plazo de ejecución. Esta asignación puede realizarse con cualquiera de las dos estrategias anteriores. Si se utiliza el modo propagado por el cliente, como aparece en la parte superior de la figura 4.6, el cliente iniciaría cada transacción a la prioridad adecuada, con la que se ejecutaría cada invocación del servicio en la instancia de componente *CompB*. En el caso de utilizar una estrategia declarada por servidor, y debido a que es necesario realizar invocaciones de un mismo servicio a distintas prioridades, sería necesario declarar dos instancias del servicio, en este caso, dos instancias del componente que lo implementa, cada uno a la prioridad adecuada. En este caso, que se muestra en la parte intermedia de la figura 4.6, la complejidad de la aplicación aumenta, al ser necesario instanciar tantos componentes como prioridades. No

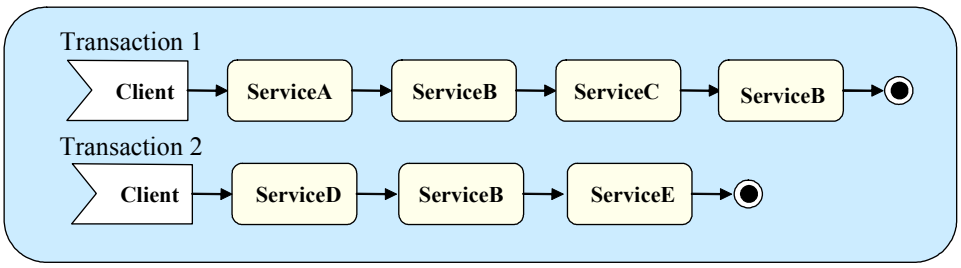


Figura 4.5: Ejemplo de sistema a planificar

es una solución siempre posible, ya que si el servicio depende del estado del componente, dos instancias tienen estados independientes y el comportamiento ofrecido no es el mismo. Por último, con esta estrategia, si se modifican las frecuencias de disparo y en consecuencia el valor de las prioridades fuese invertido, sería necesario modificar el código del cliente para planificar de nuevo el sistema, pues debería realizar las invocaciones en componentes diferentes.

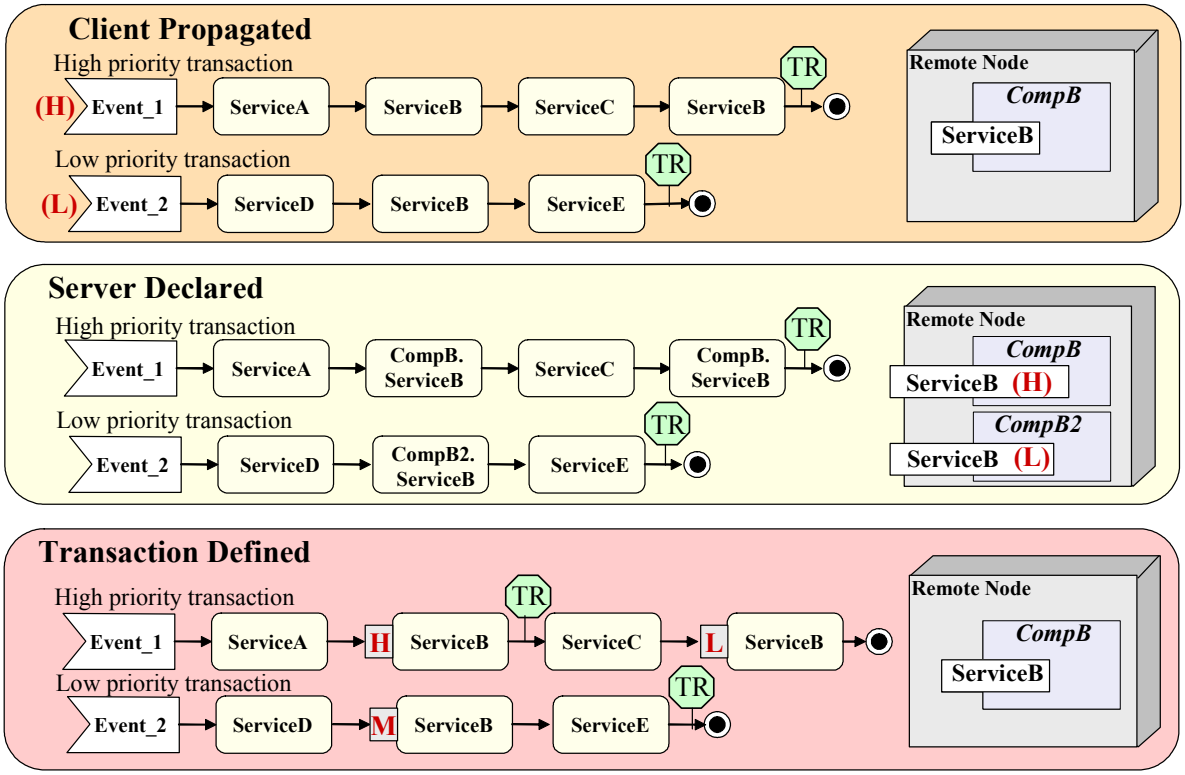


Figura 4.6: Diferentes estrategias de asignación de parámetros de planificación

- Si los requisitos temporales que se imponen a la ejecución de la aplicación se asocian a puntos intermedios de las transacciones, como corresponde al último caso de la figura 4.6, la estrategia de asignación de parámetros se hace más compleja. Una transacción que lleve a cabo el proceso de adquisición y almacenamiento de una imagen puede responder a este modelo. Las actividades involucradas en la adquisición serían prioritarias si se quiere captar la imagen en un determinado instante de tiempo, mientras que las actividades de almacenamiento podrían ejecutarse a menor prioridad, en el caso de existir otras tareas más urgentes en el sistema. La estrategia propagada por cliente no es óptima en este caso, pues obliga a que se ejecuten a la misma prioridad las dos invocaciones de *ServiceB* que se incluyen en la primera transacción. Con la estrategia declarada en el servidor podríamos conseguirlo, pero nos obligaría, como en el caso anterior, a duplicar las instancias de objetos servidores y además, modificar el código del cliente para realizar la invocación sobre el objeto servidor correcto en cada caso. Este último aspecto es muy importante cuando traslademos esta problemática a estrategias basadas en componentes, en las que el código de negocio es opaco y no se puede modificar.

RT-CORBA define un mecanismo de transformación de prioridades (*Priority Transform*), a través del que se puede modificar el valor de la prioridad con la que se ejecuta una invocación en un servidor, transformando el valor de prioridad recibido. Utilizando este mecanismo

tampoco se da solución al problema anterior, ya que la transformación de prioridad se realiza sólo en base al valor de prioridad recibido y al identificador del objeto que realiza la invocación, por lo que no podríamos distinguir dos invocaciones realizadas por un mismo objeto desde diferentes transacciones.

En RT-CCM se incluye un nuevo tipo de estrategia de asignación, denominada *Transaction_Defined*, en la que el valor que se asigna al parámetro de planificación con el que se ejecuta un servicio de un componente es función de la transacción, o más exactamente, del punto de la transacción desde el que se realiza la invocación. Este tipo de asignación es más flexible que las anteriores cuando se trata de planificar sistemas distribuidos de tiempo real [GG99], y de hecho permite aplicar como casos particulares las estrategias *Server_Declared* y *Client_Propagated*. Los mecanismos introducidos en RT-CCM para dar soporte a este tipo de asignación respetan el principio de opacidad de los componentes y permiten controlar la planificación de los servicios como características configurables de cada aplicación.

4.4. Modelo de referencia de la tecnología

La tecnología RT-CCM que se propone tiene como objetivo el desarrollo de componentes de tiempo real que puedan ser reutilizados en diferentes aplicaciones sin necesidad de modificar su código de negocio. A tal fin, da soporte a la generación automática del código de los contenedores que adaptan ese código a la plataforma de ejecución, y que incluyen los recursos necesarios para que los componentes se comuniquen entre sí y ejecuten su código de forma predecible.

Para facilitar el proceso de generación de código, la tecnología se basa en un modelo de referencia bien definido, en el que se identifican los elementos que forman parte de una aplicación y se definen los patrones de interacción entre ellos. En nuestro caso este modelo toma como base el modelo de contenedor/componente propuesto en la especificación LwCCM, extendiéndolo con algunas características necesarias en el desarrollo de aplicaciones con comportamiento temporal analizable y configurable. En la figura 4.7 se muestran los principales elementos del modelo de referencia, que se describen en detalle a lo largo de esta sección.

4.4.1. Componente instanciable: código de negocio y contenedor

Como muestra la figura 4.7, de acuerdo con el modelo de contenedor/componente, la implementación final de un componente instanciable se divide en dos partes:

- Código de negocio: Es la parte del código que implementa la funcionalidad ofrecida a través de las facetas del componente. Es desarrollado por un experto del dominio de aplicación al que pertenece el componente sin necesidad de conocer la plataforma concreta en la que se va a ejecutar ni las aplicaciones en las que se va a integrar.
- Contenedor: Adapta el código de negocio a la plataforma específica en la que se ejecuta y proporciona los mecanismos con los que el componente se conecta a otros componentes. Su código es generado totalmente por medio de herramientas automáticas a partir de los metadatos ofrecidos por el componente.

Los requisitos que se imponen al código de negocio y la arquitectura interna del contenedor de un componente RT-CCM se exponen más adelante, una vez que se hayan introducido el resto de elementos que forman parte del modelo de referencia.

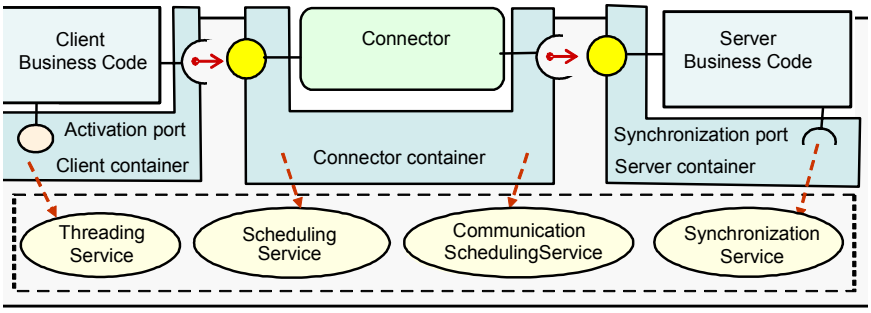


Figura 4.7: Modelo de referencia de RT-CCM

4.4.2. Gestión de la planificabilidad: StimulusId

Se describe en primer lugar la estrategia de gestión de los parámetros de planificación con los que se ejecuta cada invocación recibida en un componente, ya que el resto de elementos dependen y están relacionados con ella. Se basa en la utilización de un identificador, denominado *stimulusId,* que se transmite a lo largo de la ejecución de cada transacción. Su valor identifica el punto de la transacción desde el que se realiza cada invocación. Concretamente identifica cada actividad dentro de una transacción, entendiendo por actividad la ejecución de una operación invocada en un componente. En base al análisis del modelo de tiempo real de la aplicación se obtienen los parámetros de planificación óptimos para cada actividad/invocación realizada en el sistema, los cuales son mapeados al correspondiente valor de *stimulusId*. Durante la ejecución, cada invocación de una operación de un componente será ejecutada con su correspondiente parámetro de planificación, en base al valor de *stimulusId* que se tiene en ese momento. Con esta estrategia se pueden aplicar diferentes tipos de políticas de asignación de parámetros de planificación, en particular, la política *Transaction_Defined*.

La relación entre los conceptos de transacción, actividad y *stimulusId* se explica a través del ejemplo de transacción que se muestra en la figura 4.8. Representa una cadena de invocaciones entre componentes, que se desencadena por la invocación de la operación *OperA* en el componente *InstA* por un cliente. En la figura 4.9 se modela dicha transacción a través de un diagrama de secuencia UML, en el que se muestran los valores que van tomando el *stimulusId* y su parámetro de planificación asociado (prioridad en este caso):

- La transacción se inicia en el cliente como respuesta a un evento que se recibe del entorno. En este caso, en el inicio de cada ejecución de la transacción se asigna valor 10 al parámetro *stimulusId*. Este valor identifica unívocamente la actividad de inicio de esta transacción. En la figura 4.9, se indica que esta actividad se ejecuta a prioridad 20, por tanto se establece la correspondencia entre el valor 10 de *stimulusId* y 20 de prioridad.

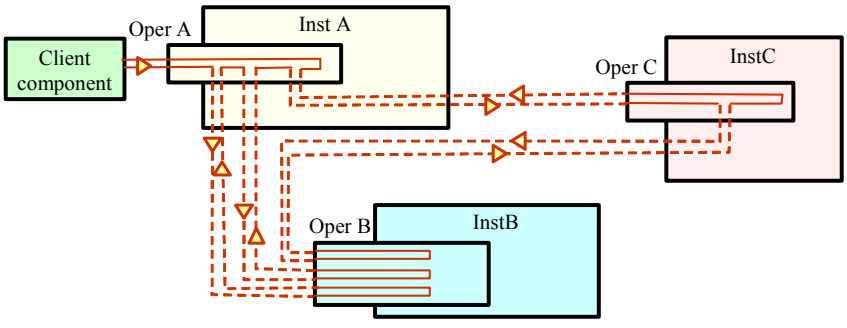


Figura 4.8: Ejemplo de cadena de invocaciones entre componentes

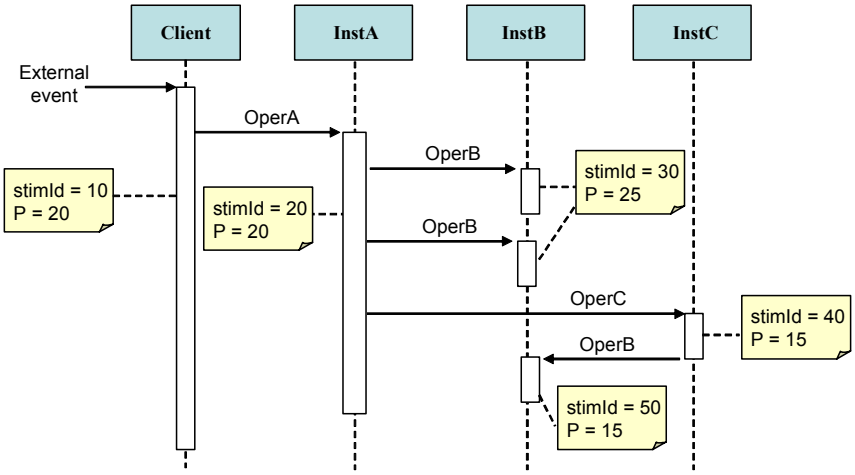


Figura 4.9: Manejo de parámetros de planificación a través del *stimulusId*

- Después de ejecutar un fragmento de código propio (con *stimulusId* 10 y prioridad 20), el cliente invoca la operación *OperA* en la instancia *InstA*, lo que da lugar al inicio de una nueva actividad, y por tanto a una modificación del valor de *stimulusId*, que toma valor 20. Esa operación se ejecuta también a prioridad 20, que es por tanto, la prioridad asociada al *stimulusId 20*. Dentro de esta operación se invoca dos veces consecutivas la operación *OperB* de la instancia *InstB*, y a continuación, la operación *OperC* de la instancia *InstC*. Cada una de esas invocaciones representa una nueva actividad dentro de la transacción.
- Las dos invocaciones de *OperB* realizadas desde *OperA* se realizan con el mismo valor de *stimulusId (20)*, por lo que son mapeadas también al mismo valor, 30 en este caso, y en consecuencia son ejecutadas con la misma prioridad, 25.
- La ejecución de la actividad correspondiente a la invocación de *OperC* se ejecuta con un valor de *stimulusId* 40 y prioridad 15. Desde ella se invoca de nuevo *OperB* en la instancia *InstB*.
- La invocación de *OperB* desde *OperC* se realiza con distinto valor de *stimulusId* que la invocación de *OperB* anterior, por lo que se mapea a un valor distinto, 50. Aquí se observa como, aunque la operación invocada es la misma que anteriormente, su ejecución en este caso lleva asociado distinto valor de *stimulusId* y en consecuencia es ejecutada con diferente prioridad, 15 en este caso.

Para cumplir con el principio de opacidad, la gestión de los *stimulusId* debe realizarse fuera del código de negocio de los componentes. En RT-CCM la interacción entre componentes se realiza a través de los conectores, y por ello, se les ha asignado la responsabilidad de gestionar la planificación de las operaciones que se invocan a través de ellos. Los conectores van a ser los encargados de gestionar la asignación de *stimulusId* y parámetros de planificación, de manera que durante la ejecución del sistema se siga el modelo reactivo o transaccional definido en su especificación y en su modelo de tiempo real.

Los conectores transforman los valores de *stimulusId*, y con ello definen los parámetros de planificación que deben ser asignados en cada invocación de una operación del componente que se realiza a través de ellos. Para ello, cada conector tiene definida una tabla por cada operación ofrecida, en la que para cada valor de *stimulusId* con el que la operación puede ser invocada, se almacena el valor al que el *stimulusId* es transformado durante la ejecución de la operación. En la figura 4.10 se observan los valores de la tabla de configuración de la operación *OperB* que

recibiría el conector que conecta los componentes *InstA* e *InstB* del ejemplo anterior. Además de la transformación de 20 a 30 que se expuso en el ejemplo, aparece otro par de valores que corresponde a una invocación realizada desde otra transacción.

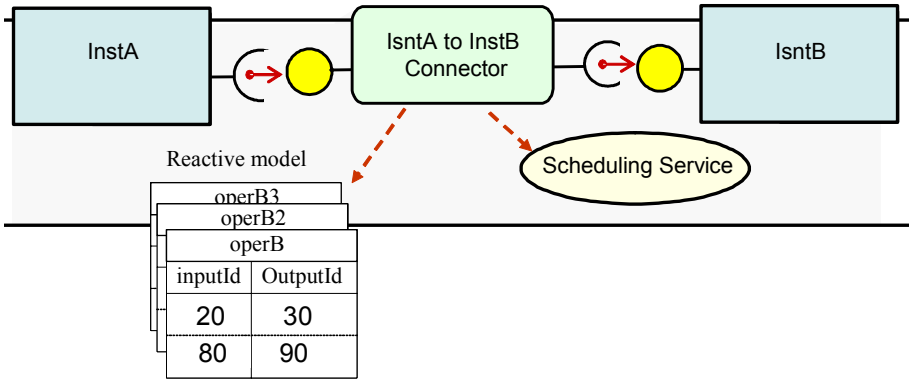


Figura 4.10: Configuración de la gestión de *stimulusId* en los conectores

En función de esta tabla de valores los conectores transforman el valor de *stimulusId* con el que se realiza la invocación, y en base al nuevo valor, modifican los parámetros de planificación del thread que ejecuta la invocación. Para ello, hacen uso de un nuevo servicio del entorno, denominado *SchedulingService* y desarrollado a tal fin. Tanto la operación de los conectores como del nuevo servicio se explican en detalle a continuación.

Volviendo de nuevo al ejemplo, las dos invocaciones de *OperB* realizadas desde *OperA* no pueden diferenciarse, se invocan con el mismo *stimulusId*, luego son mapeadas al mismo valor y por tanto sus parámetros de planificación serán idénticos. Estas invocaciones no pueden planificarse de forma diferenciada como consecuencia de que se realizan desde la misma operación, y desde el código de negocio de una operación, en este caso *OperA*, no se puede acceder o modificar el valor del *stimulusId*, que es gestionado de forma totalmente oculta a través de los conectores. Esto es, las invocaciones de una misma operación de una instancia de componente realizadas desde la misma operación dentro de una transacción son indistinguibles.

Los componentes de tipo cliente representan una excepción a esta norma. Estos componentes lanzan transacciones de negocio en la aplicación, por lo que se les otorga la capacidad de asignar valores de *stimulusId* desde su propio código de negocio, identificando de este modo nuevas transacciones. Los valores a asignar deben ser declarados como propiedades de configuración del componente. En cualquier caso, esta asignación de *stimulusId* se realiza siempre a través de interfaces propias de la tecnología, de manera que se sigue manteniendo la independencia del código de negocio con respecto a la plataforma de ejecución, al permanecer oculto el modo en que se gestiona la transmisión de *stimulusId* a bajo nivel. La interfaz utilizada será la correspondiente a los puertos de sincronización de tipo variable de condición, como explicaremos en la sección 4.4.4.

4.4.2.1. Gestión de la planificación por los conectores

En el caso de componentes que se encuentren en el mismo nodo, el flujo de ejecución cuando se realiza una invocación en una operación de una faceta de un componente es el que se muestra en la figura 4.11:

- Suponemos que en el momento en que el cliente realiza la invocación el valor de stimulusId es *currentStimId*.

- La invocación es recibida en el correspondiente conector, que lee dicho valor y lo transforma de acuerdo a su tabla de configuración. El nuevo valor, *newStimId*, caracteriza la invocación actual (por si desde ella a su vez se efectúan invocaciones a servicios de otros componentes). Con este nuevo valor, el conector invoca el procedimiento *setSchedParam* del *SchedulingService*, a través del que el thread invocante adquiere las características de planificación correspondientes a *newStimId*.
- Una vez fijados los parámetros de planificación que corresponden, se traslada la invocación al componente servidor, donde se ejecuta el código de negocio de la operación.
- Finalizada la ejecución de la operación, el conector recupera la situación existente en el momento de realizarse la invocación. En primer lugar recupera el *stimulusId* con el que se realizó la invocación (*currentStimId*), y en base a él, los parámetros de planificación que tenía el thread cuando se realizó la invocación.
- El componente cliente continúa ejecutando su código en el mismo estado que tenía cuando la invocación fue realizada.

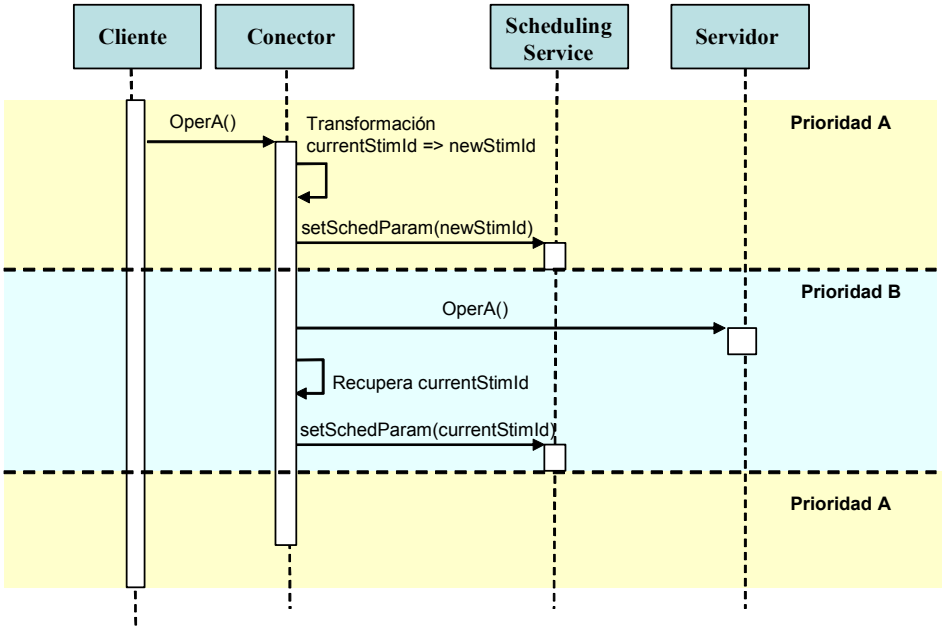


Figura 4.11: Gestión del *stimulusId* en una invocación local entre componentes

En el caso de que la invocación sea remota, el conector (como se describe en la sección 4.4.5) está formado por dos fragmentos: la parte *proxy,* que se conecta al componente cliente en el nodo local, y la parte *servant*, que se conecta al componente servidor en el nodo remoto. En este caso, el conector debe encargarse de transmitir el valor de *stimulusId* dentro del mensaje de invocación, para que el thread que ejecuta la invocación en el nodo remoto pueda adquirir los parámetros de planificación que le corresponden. El flujo de ejecución en este caso es el que se muestra en la figura 4.12:

- Suponemos que cuando el cliente realiza la invocación, el valor de *stimulusId* asociado es *currentStimId*.
- La parte *proxy* del conector recibe la invocación y lee el valor actual, pero en este caso, únicamente consulta el valor al que se mapea, sin modificarlo. Dicho valor, *newStimId*, se utiliza con dos objetivos:

  - Obtener los parámetros de planificación con los que debe realizarse el envío a través de la red de comunicaciones. En sistemas de tiempo real distribuidos no sólo se planifica la ejecución de las actividades en cada nodo, sino también los envíos de mensajes a través de la red. Por ello, se ha definido un nuevo servicio del entorno, denominado *CommunicationSchedulingService*, a través del que un conector puede obtener los parámetros de planificación que corresponden a cada envío, en base al valor actual de *stimulusId*. El conector se encarga de realizar el envío con los parámetros obtenidos en cada caso.
  - Identificar en el nodo remoto (en el que se encuentra el componente servidor) el parámetro de planificación con el que se debe ejecutar la invocación. La parte *proxy* del conector debe enviar el valor *newStimId* como parte del mensaje correspondiente a la invocación.
- Una vez recibida la invocación en el lado *servant*, se decodificará el valor de *stimulusId* y con él se accederá al *SchedulingService*, de manera que el thread encargado de ejecutar la invocación remota adquiera los parámetros de planificación que le corresponden.
- Una vez ejecutada la operación, de nuevo se accede al *CommunicationSchedulingService* (esta vez en el nodo remoto) para obtener los parámetros con los que debe ser enviado el mensaje de retorno, *paramRetorno,* en base al valor *newStimId*.
- Cuando el mensaje de retorno es recibido en la parte *proxy* del conector, se decodifican los valores de retorno y se continua con la ejecución tal cual, pues los valores de *stimulusId* y parámetros de planificación no han sido modificados en el thread invocante.

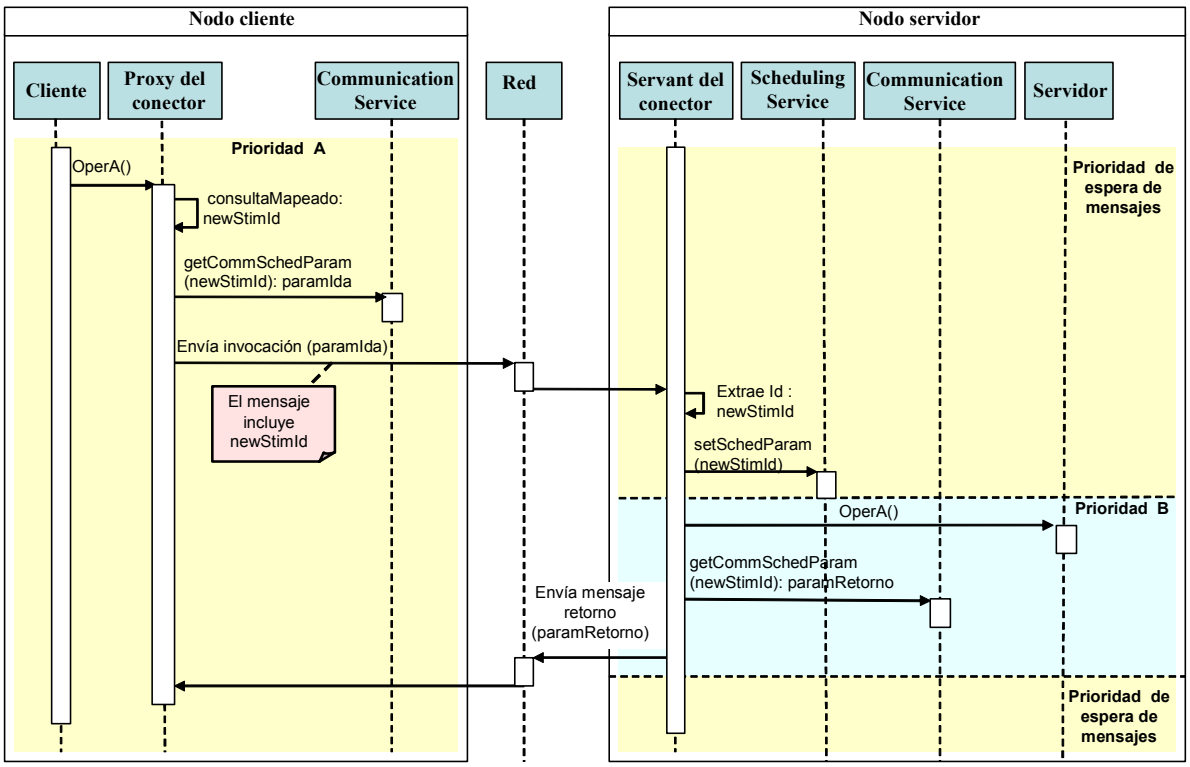


Figura 4.12: Gestión del *stimulusId* en una invocación remota entre componentes

4.4.2.2. SchedulingService y CommunicationSchedulingService

La información necesaria para modificar los parámetros de planificación del thread que ejecuta una invocación en un componente de acuerdo al identificador de transacción asignado se almacena en un servicio del entorno denominado *SchedulingService,* cuyos principales elementos se muestran en la figura 4.13. A través del método *setSchedParam*, que es invocado por los conectores con el valor de *stimulusId* correspondiente, se asigna al thread invocante el parámetro de planificación adecuado. Para ello, el servicio mantiene una estructura de datos, recibida en fase de configuración, en la que se almacenan todos los mapeados posibles entre valores de *StimulusId* y parámetros de planificación. La información para la configuración del *SchedulingService* se especifica en el plan de despliegue de la aplicación, en base a la información extraída del análisis del modelo de tiempo real de la aplicación.

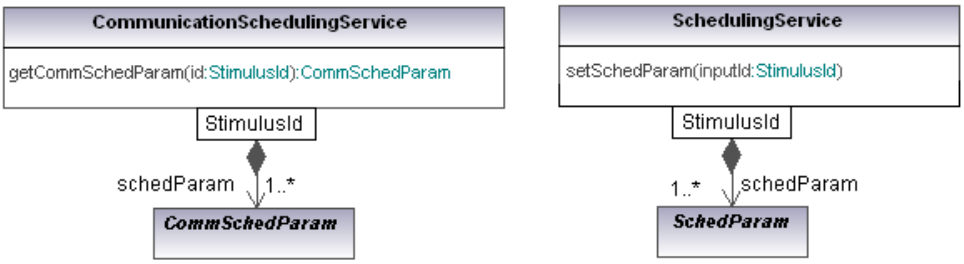


Figura 4.13: Servicios RT-CCM para la gestión de los parámetros de planificación

El tipo *SchedParam* se define como abstracto a este nivel, pues los parámetros de planificación que el servicio gestiona dependen del tipo de política de planificación que se utilice en la correspondiente plataforma, pudiendo tratarse por ejemplo de:

- prioridades, si se utilizan estrategias basadas en prioridades estáticas,
- plazos de ejecución, si la política de planificación es EDF,
- o recursos virtuales, en caso de usar una planificación basada en contratos de reserva de recursos [AGB06].

En cada implementación concreta de la tecnología RT-CCM se concretará este tipo para dar soporte al tipo de parámetro de planificación utilizado.

En el caso de los parámetros que planifican el envío de mensajes a través de una red, el servicio utilizado, *CommunicationSchedulingService*, es muy similar. Como muestra también la figura 4.13, ofrece un único método, *getCommSchedParam*, que devuelve el parámetro de planificación con el que debe realizarse cada envío de una invocación a través de la red, en base al valor de *stimulusId*. En este caso el método no modifica el parámetro de planificación del thread invocante, sólo devuelve el parámetro con el que debe realizarse el envío del mensaje. Será el conector el encargado de realizar el envío con el parámetro adecuado. El tipo *CommSchedParam* se define también como abstracto, y deberá ser concretado para cada mecanismo de comunicación soportado.

4.4.3. Gestión de la concurrencia: Puertos de activación

Los puertos de activación son el medio a través del que el desarrollador de un componente declara los threads que un componente necesita para implementar su funcionalidad. En función del tipo de activación requerido para el thread se han definido dos tipos de puerto de activación,

cuya representación gráfica se muestra en la figura 4.14. Cada uno de ellos se caracteriza por la interfaz que implementa:

- La interfaz *OneShotActivation* declara un único procedimiento, *run*, que será ejecutado una única vez por el correspondiente thread creado por el entorno. Este tipo de puerto representa un thread que el componente va a requerir de forma continua durante todo su ciclo de vida, ya que la actividad interna invocada a través del procedimiento *run* puede durar todo el tiempo que el componente esté operativo. Este tipo de patrón de activación puede ser utilizado por un componente para:
  - Implementar un thread principal que, o bien realice actividades de inicialización temporalmente, o ejecute la actividad de un componente cliente activo de forma permanente.
  - Atender dispositivos de entrada/salida activos implementando los manejadores de las interrupciones hardware que generan.
  - Implementar el thread de control que gestiona la evolución del estado interno del componente.
  - Gestionar los eventos asíncronos que se reciben por un puerto de recepción de eventos.
  - Ejecutar actividades de una transacción que se ejecutan asíncronamente.
- La interfaz *PeriodicActivation* declara un único procedimiento, *update*, que será invocado periódicamente con el thread correspondiente. Un puerto de este tipo representa un requisito de thread periódico. El periodo de invocación es un parámetro de configuración que se asignará para cada instancia de componente en el plan de despliegue. Este tipo de patrón puede servir para ejecutar:
  - Actividades de escrutinio (*polling*) con las que se atienden dispositivos de entrada/salida pasivos.
  - Actividades temporizadas internas del componente.

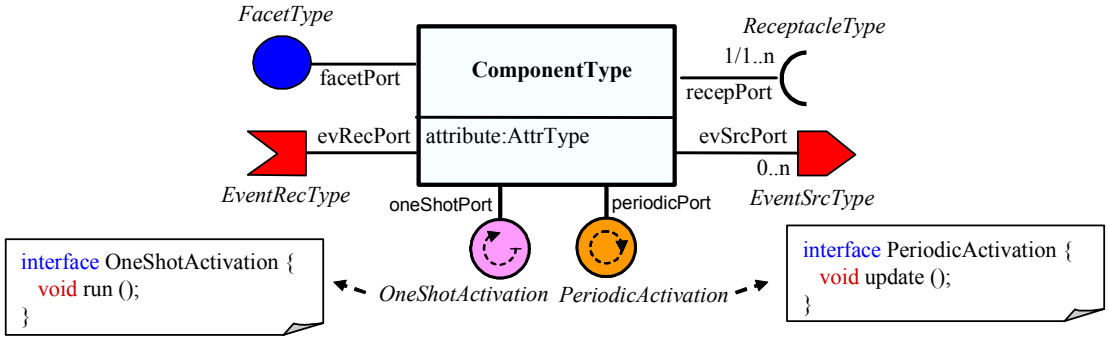


Figura 4.14: Representación gráfica de puertos de activación

Un componente puede implementar múltiples puertos de activación, constituyendo cada uno de ellos una línea independiente de concurrencia que el componente gestiona y que es independiente de los threads que ejecutan su código en base a las invocaciones de negocio recibidas. En base a la información acerca de puertos de activación que se incluye en el paquete del componente como metadatos, la herramienta de generación de código del contenedor introduce las invocaciones sobre los servicios de la plataforma de ejecución a través de las que se crean los threads y se controla su posterior ejecución. En la sección 4.6 se explica el modo de declarar puertos de activación en la descripción RT-D&C de un componente.

La configuración de los puertos de activación se realiza para cada instancia de componente definida en el plan de despliegue de una aplicación. Además del periodo en el caso de un puerto periódico, todo puerto de activación recibe el valor de *stimulusId* a través del que el thread asignado a él adquiere el parámetro de planificación con el que debe comenzar su ejecución.

Desde el punto de vista del modelo reactivo de una aplicación, los dos tipos de puerto presentan comportamientos diferentes:

- Un puerto de activación periódico representa siempre una transacción independiente que se ejecuta en el sistema de forma periódica. Su *stimulusId* de partida es aquél que se asigna al puerto, en base al cual se calculan el resto de transformaciones de *stimulusId* a aplicar.
- En el caso de un puerto de tipo *OneShotActivation* se pueden dar diferentes casos en función del objetivo con el que se requiere el thread:
  - Si se utiliza para atender distintos eventos externos, que dan lugar a diferentes respuestas en el sistema, el mismo puerto puede dar lugar al lanzamiento de transacciones diferentes. Cada una de ellas recibe un *stimulusId* unívoco que la identifica. Este es el caso típico de un componente cliente, que puede asignar *stimulusId* desde su código de negocio a través del uso conjunto de un puerto de este tipo y una variable de condición.
  - Si se utiliza para ejecutar partes del código del componente de manera asíncrona, no representa ninguna transacción independiente, sino realmente un fragmento de otra transacción. Por tanto, recibe el *stimulusId* que le corresponde dentro del flujo de dicha transacción. Esta asignación también se realiza a través de una variable de condición.
  - Si se utiliza para ejecutar alguna actividad interna del componente que se realiza durante todo su ciclo de vida, sí representa una transacción independiente, que se identifica por el *stimulusId* asignado al puerto.

4.4.3.1. ThreadingService: Servicio de ejecución concurrente

Con el fin de simplificar el código que un contenedor debe implementar para dar soporte a los puertos de activación, se define un servicio de ejecución concurrente que es ofrecido por la plataforma de ejecución y que hemos denominado *ThreadingService*. Es el servicio que usan los contenedores para crear los threads requeridos por los componentes.

Los principales elementos que se usan en la definición de este servicio se muestran en el diagrama de clases de la figura 4.15. Para su definición se ha tomado como base conceptual el patrón *Executor* [BW07], que se utiliza como mecanismo para desacoplar una actividad a ejecutar del modo en que se ejecuta desde el punto de vista concurrente: de forma secuencial, con creación dinámica de threads, con un pool de threads, etc. La interfaz *ThreadingService* define dos métodos de ejecución concurrente, *execute*, uno para puertos de activación de tipo *OneShotActivation* y otro para puertos *PeriodicActivation*. En ambos casos su invocación por parte de un contenedor tiene como consecuencia la creación de un thread que ejecutará el método ofrecido a través del puerto, una única vez en el caso de puertos *OneShotActivation* o de forma periódica en caso de puertos *PeriodicActivation*.

Cuando se instancia un componente su contenedor accede al *ThreadingService* e invoca el procedimiento *execute* correspondiente por cada puerto de activación declarado. A partir de ese momento, el contenedor es el encargado de gestionar la ejecución de cada thread, a través de los

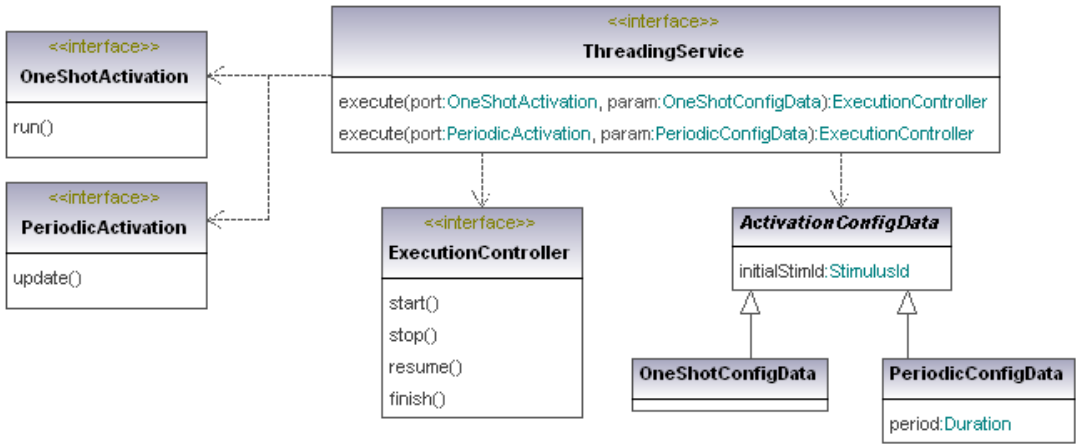


Figura 4.15: Servicio RT-CCM para la creación de threads

objetos controladores (de tipo *ExecutionController*) que le devuelve el procedimiento *execute*. A través de este controlador, el contenedor puede iniciar la ejecución del thread (cuando el componente sea activado por primera vez), suspenderla, reiniciarla o detenerla definitivamente, según corresponda a la lógica de la aplicación.

Ambos procedimientos de ejecución poseen un parámetro de entrada adicional, *param*, cuyo tipo depende también del tipo de puerto, *OneShotConfigData* o *PeriodicConfigData*. A través de este parámetro se asignan las propiedades de configuración que se requieren para planificar la ejecución del puerto. Estas propiedades son el periodo en el caso de puertos periódicos, y en ambos casos, el valor de *stimulusId* con el que se inicia cada ejecución (a través del que se conoce el parámetro de planificación correspondiente). Estos tipos pueden ser extendidos con otras propiedades características de cada implementación concreta de la tecnología RT-CCM.

El servicio de ejecución concurrente que ofrezca la plataforma en una implementación concreta de RT-CCM debe implementar la interfaz *ThreadingService*, y hacerlo de modo que el comportamiento de la aplicación sea predecible. El mecanismo elegido para implementar la concurrencia queda así fuera de los contenedores, con lo cual se simplifica enormemente la generación de su código y el tamaño del código final de cada componente.

4.4.4. Gestión de la sincronización: Puertos de sincronización

De forma análoga a la gestión de la concurrencia, el mecanismo utilizado para la gestión de las necesidades de sincronización de un componente se basa en la utilización de un tipo especial de puerto, denominado puerto de sincronización. Por cada mecanismo de sincronización requerido en el código de negocio, el desarrollador debe declarar un puerto del tipo correspondiente en la descripción del componente.

Existen dos tipos de puerto de sincronización, cuya representación gráfica se muestra en la figura 4.16. Se identifican por la interfaz que requieren:

- La interfaz *Mutex* identifica un puerto a través del que se requiere un mecanismo de tipo *Mutex*. A través de sus métodos *lock* y *unlock* se puede garantizar la ejecución segura de fragmentos de código o el acceso seguro a estructuras de datos internas.
- La interfaz *ConditionVariable* identifica un puerto a través del que se requiere un mecanismo de tipo variable de condición. La invocación del método *wait* produce la suspensión del thread invocante, que es reactivado cuando otro thread invoca los métodos *signal* (se reactiva un sólo thread de la cola de threads suspendidos en el objeto de

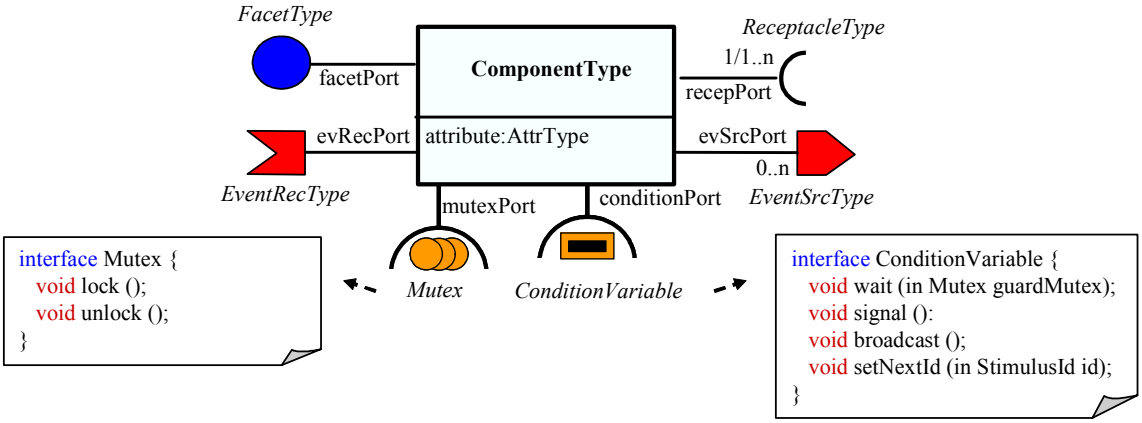


Figura 4.16: Ejemplo de representación gráfica de puertos de sincronización

sincronización) o *broadcast* (se reactivan todos los threads de la cola). Este tipo de mecanismo se puede utilizar para suspender una actividad interna del componente a la espera de que se alcance cierto estado o para implementar una sincronización de espera entre tareas.

En base a los puertos de activación declarados como metadatos del componente, la herramienta de generación de código del contenedor introduce el código que accede a los servicios de la plataforma que crean los mecanismos solicitados, y se los entrega al componente durante la fase de lanzamiento de la aplicación. El modo de declarar puertos de sincronización en la descripción RT-D&C de un componente se define en la sección 4.6.

4.4.4.1. SynchronizationService: Servicio de sincronización

De nuevo con el objetivo de simplificar al máximo la generación del código de los contenedores en lo que a gestión de la sincronización se refiere, se ha definido un nuevo servicio de la plataforma a través del que los contenedores obtienen los mecanismos de sincronización requeridos. El servicio, denominado *SynchronizationService*, cuya interfaz se muestra en la figura 4.17 junto con las interfaces *Mutex* y *ConditionVariable*, utilizará los mecanismos disponibles en la plataforma para implementar los objetos que ofrecen las interfaces requeridas.

En el caso de mecanismos de tipo *Mutex*, se ha de declarar el valor del parámetro de planificación que se utiliza para gestionar los accesos a dicho objeto (parámetro *schedParam* en el método *getMutex*). Dicho valor dependerá del tipo de protocolo de sincronización utilizado en el mutex, y será asignado a cada puerto de sincronización de cada instancia de componente declarada en un plan de despliegue. Su valor es extraído por las herramientas que analizan el modelo de tiempo real de la aplicación.

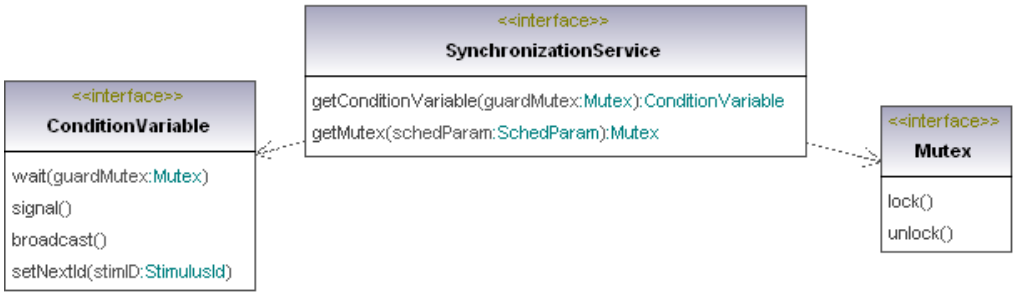


Figura 4.17: Servicio RT-CCM para la gestión de mecanismos de sincronización

En el caso de una variable de condición, no es necesario asignar valores de parámetros de planificación para su gestión directa; sin embargo, al representar un punto de sincronización entre threads, sí es necesario definir el comportamiento de dichos threads respecto de sus propios parámetros de planificación, o lo que es lo mismo, respecto del valor de *stimulusId* que transmiten. Cuando un thread que ha quedado suspendido en un *wait* es activado por otro, existen dos opciones:

- Que el thread suspendido hubiese quedado a la espera de una determinada condición (estado del componente, disponibilidad de datos, etc.) para proseguir con su ejecución dentro de la misma transacción. En ese caso el thread suspendido debe continuar su ejecución con el mismo valor de *stimulusId* que tenía en el momento de la suspensión, pues sigue ejecutando dentro de la misma transacción.
- Que la activación dé lugar al comienzo de una nueva transacción en la aplicación. Es por ejemplo el caso que se expuso en el apartado anterior, en el que un thread obtenido a través de un puerto de tipo *OneShotActivation* queda suspendido a la espera de un conjunto de eventos externos, generando una transacción diferente para cada uno de ellos. En este caso, el thread se suspende en la variable de condición y cuando es activado, continúa su ejecución con un valor de *stimulusId* diferente para cada posible transacción (para cada evento externo diferente recibido). El valor a asignar en cada caso se establece de forma previa a la señalización a través del método *setNextId.* El componente debe estar configurado para asignar de forma correcta el valor que corresponde en cada caso.

4.4.5. Gestión de las comunicaciones: Conectores

En RT-CCM, la conexión entre dos componentes se va a realizar siempre a través de un tipo especial de componente, denominado conector, que engloba en su código los mecanismos que se requieren para llevar a cabo la interacción entre los componentes conectados, siempre en base a mecanismos de comunicación con comportamiento temporal predecible.

El concepto de conector como elemento que encapsula la interacción entre componentes, permitiendo que el desarrollador del código de negocio se centre exclusivamente en la lógica de la aplicación, aparece ya en los inicios del desarrollo de arquitecturas software basadas en componentes [SG96][BCK98]. Actualmente algunos modelos de componentes lo consideran como un elemento de primer orden dentro de la especificación de una aplicación, esto es, la aplicación se define como un ensamblado de componentes conectados a través de conectores que se definen de forma explícita [BHP06][COMPS]. En el caso de CCM, hasta el momento no existía el concepto explícito de conector; sin embargo, la especificación "*DDS for Lightweight CCM*" [DDSCCM], actualmente en fase de finalización, trata de incorporar este concepto a CCM con el objetivo de implementar nuevos modos de interacción entre componentes, especialmente aquellos basados en servicios de distribución de datos (*Data Distribution Service)* [DDS].

Con el uso de conectores se resuelven algunos problemas que son habituales cuando se trata de adaptar tecnologías basadas en componentes a sistemas distribuidos, heterogéneos y de tiempo real, como son los siguientes:

- El middleware o los mecanismos de comunicación disponibles en la plataforma no están concebidos para aplicaciones orientadas a componentes.
- La naturaleza de un componente requiere mecanismos de comunicación especiales, que no son proporcionados por el middleware disponible. Este puede ser el caso de un componente que a través de uno de sus puertos implemente funciones de *streaming* de

video con prestaciones de tiempo real, para lo cual requiere la utilización del protocolo RTP (*Real-Time Transport Protocol*) [RTP], mientras que por el resto de sus puertos pueda utilizar protocolos más genéricos.

- Las aplicaciones a desarrollar tienen requisitos de tiempo real estricto y requieren sistemas de comunicación que ofrezcan prestaciones de tiempo real.

Un conector es, por tanto, un elemento especializado que incorpora en su código los mecanismos de comunicación que se necesitan para establecer la interacción entre un receptáculo de un componente cliente y una faceta de un componente servidor. Cuando ambos componentes se encuentran en el mismo nodo, la estructura del conector es como la de cualquier componente que posee una única faceta y un único receptáculo, implementando ambos la interfaz correspondiente a los puertos que interaccionan (como en la figura 4.7). En el caso de que cliente y servidor se encuentren en nodos diferentes, el conector se divide en dos fragmentos. Como muestra la figura 4.18, el receptáculo del componente cliente se conecta a la faceta ofrecida por la parte *proxy* del conector, mientras que la faceta del componente servidor se conecta al receptáculo del fragmento *servant* del conector. Entre los dos fragmentos la comunicación se realizará de acuerdo al mecanismo elegido.

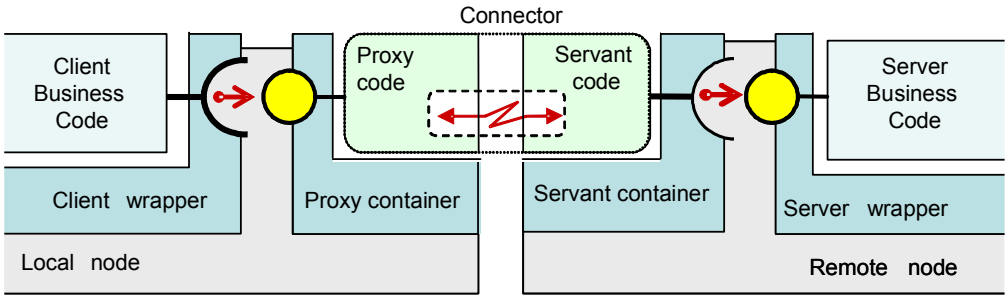


Figura 4.18: Estructura de un conector RT-CCM distribuido

Utilizando esta estrategia se simplifica en gran medida la complejidad del código de los contenedores de los componentes, ya que la conexión entre facetas y receptáculos de componentes se va a realizar siempre de forma local y directa, a través del propio lenguaje de programación, y sin necesidad de incluir mecanismos de gestión de la comunicación en el interior del código del contenedor. El código de negocio se simplifica también, pues es desarrollado como si el componente ejecutase siempre en entorno monoprocesador.

Las responsabilidades, y con ello la complejidad, del código de un conector dependen del tipo de conexión a implementar. En el caso de invocaciones locales, en las que ambas instancias conectadas se encuentran en el mismo nodo, las distintas opciones se muestran en la figura 4.19:

- En el caso más sencillo, cuando los componentes conectados están desarrollados en el mismo lenguaje de programación y las invocaciones entre ellos se realizan de forma síncrona, la única responsabilidad del conector consiste en gestionar los parámetros de planificación de cada invocación realizada a través de él. Un mismo conector puede recibir invocaciones procedentes de distintas transacciones, por lo que como ya vimos en la sección 4.4.2, es configurado con las tablas de valores que indican los mapeados de *stimulusId* a aplicar en cada caso. Con el valor de *stimulusId* correspondiente, el conector utiliza el servicio *SchedulingService* para asignar al thread invocante el parámetro de planificación que le corresponde a la ejecución de la invocación.

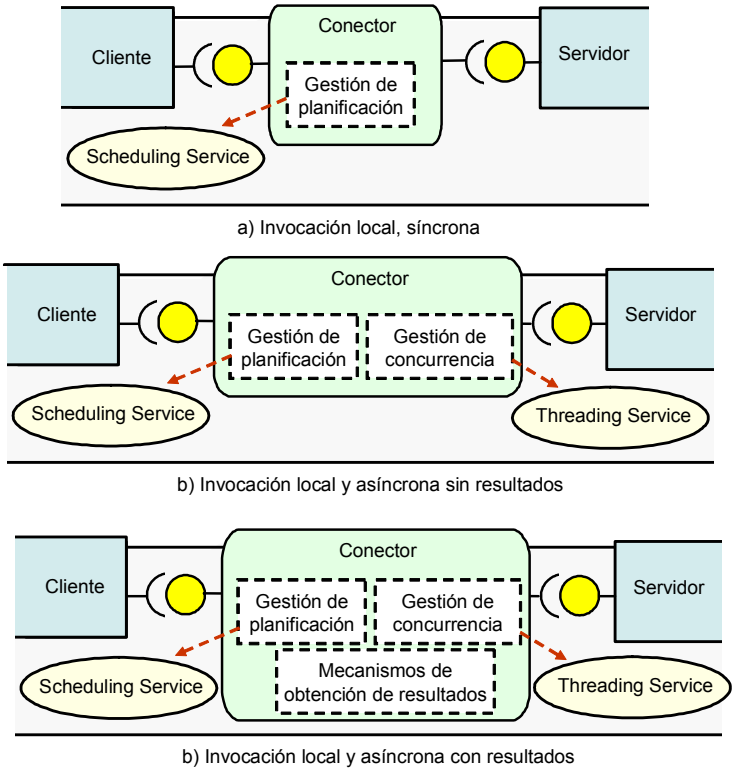


Figura 4.19: Responsabilidades de un conector local

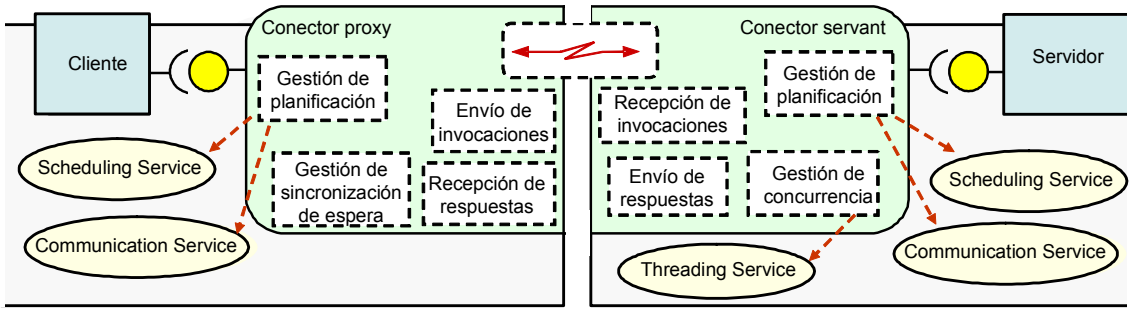
- En el caso de que alguna de las operaciones ofertadas deba ser ejecutada en modo asíncrono, el conector debe ofrecer mecanismos de ejecución concurrente. Haciendo uso del *ThreadingService*, el conector obtendrá un thread del entorno para ejecutar cada operación asíncrona invocada en él, y así, retornar de inmediato al cliente el control de flujo. Si se trata de una invocación asíncrona con resultados, el conector debe incorporar también mecanismos que permitan al componente cliente obtener el resultado de la invocación, a través de estrategias de *callback* o de encuesta.
- Además, con el objetivo de poder conectar componentes desarrollados en diferentes lenguajes de programación, el conector opcionalmente puede implementar los mecanismos de serialización y deserialización necesarios para que los argumentos de invocación y retorno puedan ser procesados por el lenguaje correspondiente.

Los conectores implementan toda su funcionalidad cuando los componentes conectados se encuentran en nodos diferentes, en cuyo caso han de hacer uso internamente de algún mecanismo de comunicación o middleware, que debe ofrecer además prestaciones de tiempo real. Las responsabilidades de un conector distribuido, que se resumen en la figura 4.20, son las siguientes:

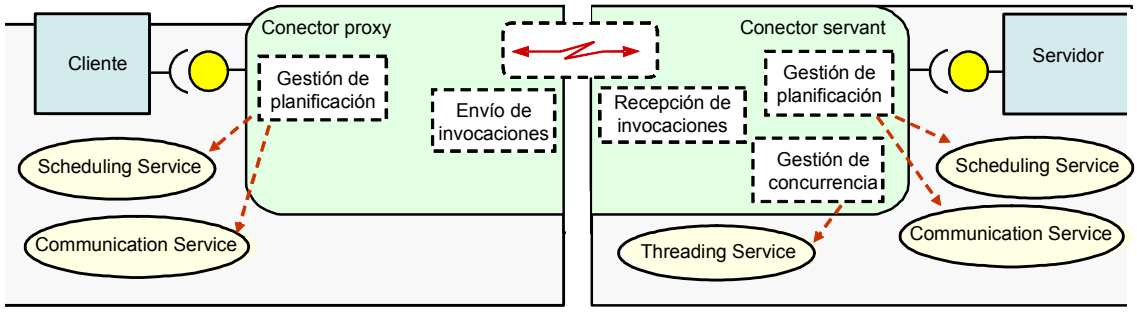
- Debe seguir dando soporte a la gestión de la planificación en base a los valores de *stimulusId*. Como se explicó en la sección 4.4.2, ha de transmitir el valor de *stimulusId* con cada invocación, de modo que ésta sea ejecutada en el nodo remoto con el parámetro de planificación que le corresponde.
- Cuando la invocación realizada sea de tipo síncrono, la parte *proxy* de un conector remoto debe proporcionar mecanismos de sincronización en los que se suspendan los componentes clientes a la espera de que finalice la invocación. Como varios threads

pueden quedarse suspendidos en un mismo conector, éste debe distinguir cuál es el thread al que corresponde cada retorno de invocación.

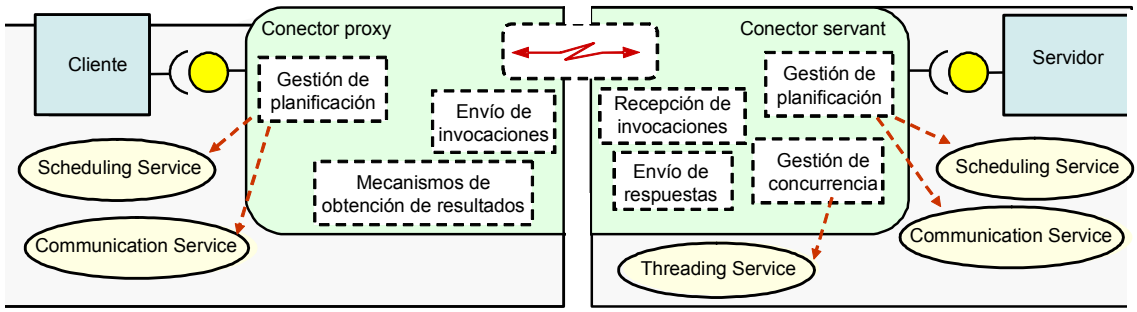
- En el caso de invocaciones asíncronas, no son necesario mecanismos de sincronización en el lado *proxy*. Si la invocación es asíncrona con resultados, al igual que en el caso local, son necesarios mecanismos en el lado *proxy* que permitan al componente cliente recuperar los resultados de la invocación.
- Debe implementar los mecanismos a través de los que las invocaciones y los retornos de resultados son traducidas a mensajes que se envían a través de la red. El tipo de codificación dependerá del mecanismo de comunicación empleado, así como la información que se incluye en el mensaje.
- Una vez generado el mensaje, la parte *proxy* debe encargarse de transmitirlo a través de la red, y la parte *servant* de recibirlo y trasmitirlo al componente servidor. Para ello, debe contar con threads que permitan ejecutar concurrentemente las diferentes invocaciones recibidas mientras sigue atendiendo la posible llegada de nuevos mensajes. Al igual que ocurre con el caso de invocaciones locales y síncronas, la parte *servant* del conector accederá al *ThreadingService* para obtener los threads que necesita.



a) Invocación remota síncrona



b) Invocación remota asíncrona sin resultados



c) Invocación remota asíncrona con resultados

Figura 4.20: Responsabilidades de un conector distribuido

El tipo de conector a incluir en la conexión entre puertos de dos componentes lo decide la herramienta de despliegue, en función de los atributos de la conexión y la ubicación de ambos componentes. En base a esa información y a la descripción de la interfaz de los puertos conectados, una herramienta genera el código completo del conector, utilizando las correspondientes plantillas que se encuentren almacenadas en la plataforma de diseño. En este aspecto la estrategia que se sigue en RT-CCM es diferente a otras en las que los conectores son identificados de forma explícita en el diseño de la arquitectura de la aplicación. En nuestro caso, los conectores son implícitos, por cada conexión entre componentes y en base a las características de la misma, una herramienta genera el código del conector y modifica el plan de despliegue para tener en cuenta su inclusión como parte de la aplicación.

En RT-CCM, el aspecto básico de los conectores es que los mecanismos de comunicación elegidos para su implementación exhiben un comportamiento temporal predecible. De este modo, y tal y como se explicó en capítulos anteriores, al mismo tiempo que se genera el código de los conectores, se genera el modelo de tiempo real que define la secuencia de actividades que se ejecutan en el sistema cada vez que una invocación se realiza a través de ellos. Dichos modelos son utilizados para obtener el modelo de comportamiento temporal de la aplicación completa.

4.5. Arquitectura interna de la implementación de un componente

En tiempo de ejecución un componente es un módulo software ejecutable que resulta de enlazar el código de negocio del componente (sin modificación) con el código del contenedor, que es generado totalmente por las herramientas en base a los metadatos proporcionados en el paquete del componente. La estructura interna de una implementación final y completa de un componente es la que se muestra en la figura 4.21. El código de negocio puede ser elaborado como si el entorno de ejecución fuese siempre monolenguaje, ya que como se aprecia en la figura, con la estructura propuesta todas las interacciones del código de negocio con el contenedor se realizan a través de interfaces nativas del lenguaje de programación utilizado. Además, con la utilización de conectores para el manejo de las conexiones remotas, el código de negocio puede ser desarrollado siempre para entorno monoprocesador.

4.5.1. Arquitectura del contenedor

El contenedor engloba los recursos que adaptan el código de negocio a la plataforma, siguiendo para ello las reglas de interacción propuestas en el CIF (*Component Implementation Framework*) de LwCCM. Como se muestra en la figura 4.21, el contenedor está compuesto por varios elementos, cuya estructura se simplifica enormemente gracias a la utilización de conectores:

- El adaptador (*Wrapper*) es una clase de envoltura, que engloba el resto de elementos y ofrece hacia el exterior la interfaz equivalente del componente. En base a su especificación todo componente posee una interfaz de referencia denominada interfaz equivalente del componente (*Equivalent Interface*), que permite gestionarlo de forma estandarizada, ofreciendo el conjunto de métodos a través de los que los componentes clientes pueden acceder a él. Las reglas para la definición de esta interfaz las establece LwCCM. La interfaz equivalente extiende siempre a la interfaz *CCMObject*, que ofrece métodos para acceder a las facetas del componente, conectar componentes a sus receptáculos, asignar valores a sus parámetros de configuración, etc.

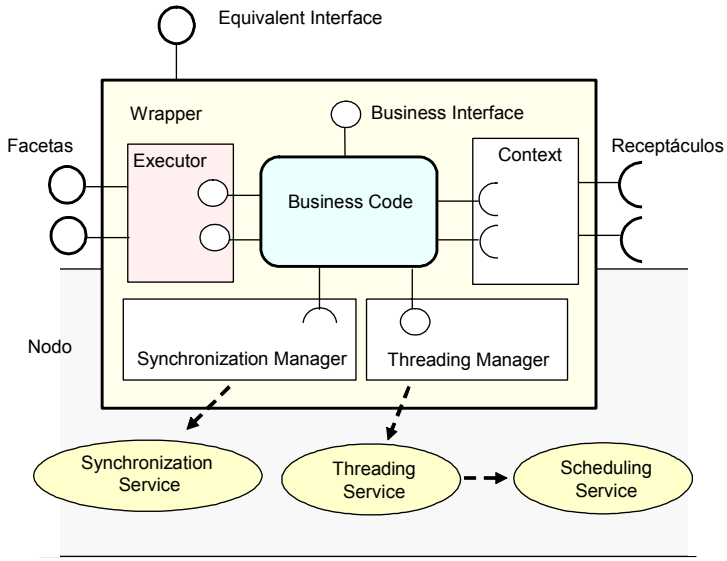


Figura 4.21: Arquitectura interna de la implementación de un componente

- El ejecutor (*Executor*) implementa los adaptadores que trasladan las invocaciones realizadas por los componentes clientes a través de la interfaz equivalente (es decir, en el adaptador) a las invocaciones de las operaciones correspondientes de las facetas del código de negocio. En RT-CCM esta clase sólo realiza una reinvocación del correspondiente método en el código de negocio.
- El contexto (*Context*) es el mecanismo a través del que el código de negocio puede invocar operaciones en los componentes conectados a través de los receptáculos. Engloba los mecanismos que se requieren para traducir la invocación realizada en el código de negocio en una invocación sobre el correspondiente conector.
- El gestor de threads (*Threading Manager*) incluye todos aquellos elementos necesarios para gestionar los puertos de activación que se hayan declarado en el componente. Por cada puerto de activación, el código del gestor crea, a través del *ThreadingService*, el thread para su ejecución y lo gestiona a lo largo de todo su ciclo de vida.
- El gestor de sincronización *(Synchronization Manager*) incluye la infraestructura necesaria para crear los mecanismos de sincronización que el componente requiere a través de los puertos de sincronización que tiene declarados. Para ello hace uso del *SynchronizationService*.

4.5.2. Implementación del código de negocio

Para facilitar la generación de código, la interacción del contenedor con el código de negocio debe realizarse de forma estandarizada. Para ello se han definido como parte de la tecnología RT-CCM una serie de reglas de diseño para la construcción del código de negocio, que se engloban en lo que hemos denominado la interfaz de negocio (*Business Interface*). Esta interfaz agrupa todos los métodos que permiten al contenedor manejar el código de negocio durante todo el ciclo de vida del componente. Se genera de forma automática a partir de los metadatos del componente y se formula haciendo uso únicamente del lenguaje de programación elegido. Por tanto, aunque la estructura interna del código de negocio de un componente es libre, siempre debe implementar esta interfaz.

La figura 4.22 muestra la interfaz de negocio, *AdaScadaEngine_Mngr*, para el caso de la implementación *AdaScadaEngine* que vimos como ejemplo en el capítulo anterior y cuyas principales características se muestran también en la figura. Entre los métodos que se definen en esta interfaz se encuentran:

- Métodos de navegación, con formato *get_<portName>*, que permiten obtener en tiempo de ejecución cada una de las implementaciones de las facetas que ofrece el componente, así como las implementaciones de cada uno de los puertos de activación.
- Métodos de conexión, con formato *<portName>_connect*, que permiten conectar cada uno de los componentes servidores, así como los objetos de sincronización que se obtengan a través de los puertos de sincronización.
- Métodos de asignación y lectura de valor para los atributos de configuración del componente, con formato *set/get_<attributeName>*.
- Métodos para la gestión del ciclo de vida del componente: activación, desactivación, eliminación, etc. Todas ellas se agrupan en la interfaz común RT-CCM, de la que hereda cualquier interfaz de negocio.

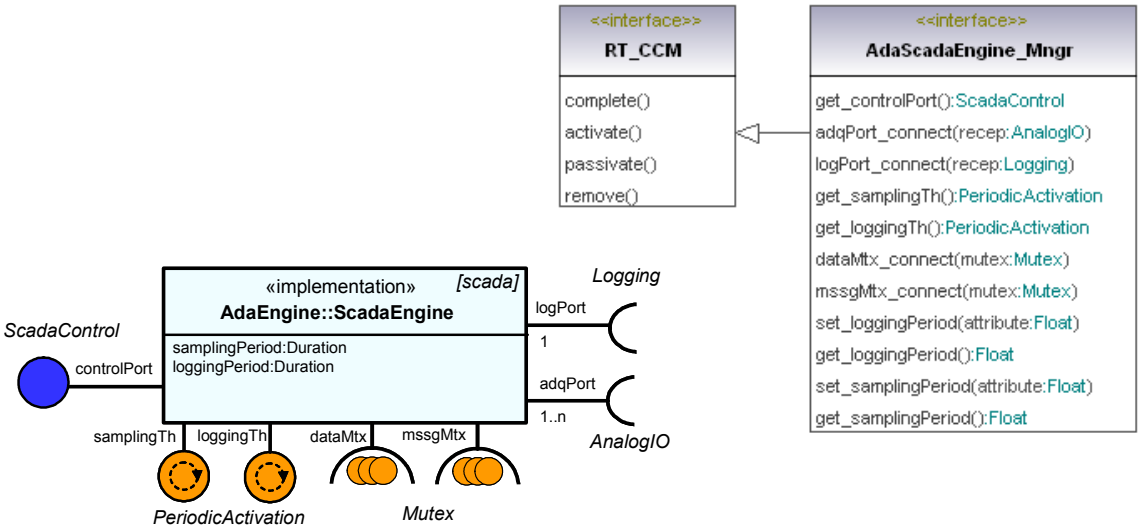


Figura 4.22: Interfaz de negocio del componente *AdaScadaEngine*

El desarrollador del código de negocio tiene total libertad para elaborarlo, salvo las restricciones expuestas anteriormente (implementar la interfaz de negocio y evitar la creación de threads y mecanismos de sincronización). Sin embargo, existen algunos aspectos de su estructura interna que deben ser tenidos en cuenta para desarrollar una implementación correcta:

- El componente deberá ofrecer implementaciones de cada una de las facetas ofertadas. En el caso más simple, cuando cada faceta ofertada por un componente corresponde a una interfaz diferente, el propio componente puede implementarlas directamente. En ese caso, los métodos de navegación utilizados para acceder a cada faceta devuelven siempre una referencia al propio componente. Sin embargo, cuando un componente ofrece varias facetas que implementan la misma interfaz, deberá incluirse un objeto independiente por cada una, pues pueden implementar la funcionalidad de manera distinta. El acceso a cada uno de estos objetos independientes es el que se retorna a través de los correspondientes métodos de navegación.

- Del mismo modo, por cada requisito de concurrencia del componente, además de declarar el correspondiente puerto en los metadatos, hay que incluir como parte del código de negocio un objeto que implemente la interfaz de activación correspondiente. Al igual que ocurre con las facetas, si el componente tiene un solo puerto de activación de cada tipo, el propio componente puede implementar la interfaz correspondiente a los puertos.
- Tanto durante el desarrollo de las facetas como de los puertos de activación, si se requieren mecanismos de sincronización, se declara el correspondiente puerto en los metadatos y el código de negocio se desarrolla en base a invocaciones sobre los métodos ofrecidos por la interfaz correspondiente (*Mutex* o *ConditionVariable*).
- Las implementaciones tanto de facetas como de puertos de activación operarán de acuerdo al estado del componente, que es único por cada instancia. Una estrategia adecuada de implementación consiste en agrupar el estado del componente en un objeto independiente, que pueda ser accedido por todos los elementos, eliminando de este modo posibles dependencias cíclicas. Dentro del estado de un componente se almacenan todos los atributos de configuración definidos para el componente en su especificación, los enlaces a los receptáculos, los enlaces a los mecanismos de sincronización, así como toda la información que el desarrollador del componente considere necesaria para implementar la funcionalidad del componente.

4.6. Especificación de configuración y despliegue de aplicaciones RT-CCM

Como se explicó en el capítulo 3, la especificación D&C está formulada mediante un metamodelo PIM, independiente de la tecnología de componentes empleada, del lenguaje de programación de los componentes, etc. Cuando se adapta a una determinada tecnología, en este caso RT-CCM, hay que definir el modo en que incluyen los aspectos característicos de la tecnología elegida. En definitiva, crear el metamodelo PSM.

El objetivo es que el desarrollador de una aplicación basada en la tecnología RT-CCM maneje únicamente la información que aparece en los descriptores RT-D&C, la cual debe ser suficiente para configurar la aplicación tanto desde el punto de vista funcional como de tiempo real. Hay que definir el lugar y el formato con que se van a reflejar en los correspondientes descriptores aquellas características de RT-CCM que permiten generar aplicaciones predecibles, como son los puertos de activación y sincronización o la gestión de los *stimulusId*.

4.6.1. Descripción RT-D&C de un componente RT-CCM

Los elementos utilizados en RT-D&C para describir un componente deben ser extendidos en el caso de componentes RT-CCM para incorporar los requisitos de concurrencia y sincronización que se identifiquen durante su implementación. A continuación se describen los nuevos elementos de información utilizados en la descripción RT-D&C de un componente RT-CCM, junto con los descriptores en los que son incluidos.

4.6.1.1. Descripción de los tipos de puerto

En la descripción RT-D&C de la interfaz externa de un componente se indica para cada puerto si se trata de un puerto ofertado o requerido, así como el tipo específico del puerto (donde se referencia la interfaz o el tipo de evento correspondiente). Con esta información bastaría para

distinguir entre los 4 tipos de puerto que existen en RT-CCM. Sin embargo, para clarificar esta información y facilitar su procesamiento por parte de herramientas, se ha incluido un nuevo atributo en la descripción de un puerto a través del cual se va a indicar el tipo de puerto: faceta, receptáculo simple, receptáculo múltiple, etc. Esta estrategia es la misma que se adopta en el modelo PSM para CCM que se incluye en la propia especificación D&C.

4.6.1.2. Declaración de los puertos de activación

La decisión del nivel de concurrencia con que un componente implementa su funcionalidad corresponde al desarrollador del código de negocio, quien por cada thread requerido:

- Declara un puerto de activación en los metadatos del componente.
- Encapsula el código que ejecutará dicho thread en un objeto que implemente la interfaz correspondiente.

Por tanto, la declaración de los puertos de activación de un componente debe ser incluida a nivel de la descripción de la implementación del componente (elemento *ComponentImplementation Description*), cuya elaboración es responsabilidad del *developer*. Para ello, hace uso de la extensión incluida en RT-D&C que permite definir nuevas propiedades de configuración del componente a nivel de la implementación (atributo *specificProperty*). Por cada puerto de activación, se declara una nueva propiedad de tipo *Periodic_Activation* o *OneShot_Activation,* cuyo nombre identifica al puerto. Las propiedades de este tipo se incluyen con una doble finalidad:

- A través de ellas se configuran las características de ejecución de cada puerto de activación en las instancias del componente declaradas en un plan de despliegue.
- Informan a las herramientas de generación de código de los requerimientos de threads del componente, de forma que se incluya en los contenedores el código necesario para que en el momento de la instanciación el contenedor los cree a través del *ThreadingService*.

Para dar soporte a este tipo de propiedades en los descriptores es necesario:

- Por un lado, incluir los tipos de propiedad *Periodic_Activation* y *OneShot_Activation* entre los tipos admitidos para las propiedades de configuración de un componente, esto es, en el tipo *DataType*.
- Asimismo, incluir dentro del tipo *Any*, que engloba los valores que se pueden asignar a las propiedades de configuración, soporte para los valores que pueden asignarse en este caso:
  - En el caso de una propiedad de tipo *OneShot_Activation*, el valor a asignar tiene un solo campo, *initialStimId*. A través del que se le asigna el valor de *stimulusId* con el que se inicia la ejecución del thread.
  - En el caso de un puerto de activación de tipo *Periodic_Activation* debe asignarse, además del valor anterior, el periodo de invocación.

Al igual que el resto de propiedades de configuración de un componente, estas propiedades reciben valores concretos en la declaración de cada instancia de componente dentro de un plan de despliegue.

Como ejemplo de declaración de una propiedad de este tipo, vemos en la tabla 4.1 un fragmento del fichero.xml en el que se describe la implementación *AdaScadaEngine*. Esta implementación requiere dos puertos de activación periódicos, *samplingTh* y *loggingTh*. En el fragmento se muestra la declaración del primero de ellos. El valor del periodo a asignar a dicho puerto es el

mismo que se asigne a la propiedad de configuración *samplingPeriod,* definida en la interfaz externa del componente, por lo que se le asigna dicho valor por referencia. Por el contrario, el valor del atributo *initialStimId* será asignado cuando se declare una instancia del componente en el plan de despliegue.

Desde el punto de vista de tiempo real, hay aspectos de los puertos de activación que obviamente influyen en el comportamiento temporal de la aplicación. Así, cada puerto de activación periódico declarado, independientemente del modo en que se modele el sistema, representa una transacción que se va a ejecutar en el sistema y cuyos parámetros de planificación van a influir en su planificabilidad. Es por ello que se deben declarar propiedades de tiempo real que hagan referencia al parámetro de planificación de cada puerto de activación, como *samplingThPrty* en el ejemplo. Las herramientas de análisis del modelo final de la aplicación calculan los valores óptimos de estas propiedades de tiempo real, y una herramienta automática se encarga de mapearlas a los valores de *stimulusId* asociados al puerto de activación, de modo que el puerto comience su ejecución con el parámetro adecuado.

Tabla 4.1: Declaración de puertos de activación y sincronización en la descripción de una implementación de componente

```
...
<!-- ComponentImplementationDescription-->
<implementation name="AdaScadaEngineImpl">
   <referencedImplementation>
      <description label="Ada-CCM Implementation of the ScadaEngine Component">
      ...
   <!--MonolithicImplementationDescription-->
      <monolithicImpl rtModel="components/scada/AdaScadaEngine.rtm.xml">
         ...
      </monolithicImpl>

      <!-- Implementation Specific Properties-->
      <specificProperty name="samplingTh" type="PERIODIC_ACTIVATION"/>
      <specificConfigProperty name="samplingTh.period" type="DURATION">
        <refValue>samplingPeriod</refValue>
      </specificConfigProperty>
      </description>

      <specificProperty name="dataMtx" type="MUTEX"/>
      ...
      <!-- Implementation Real-time Specific Properties-->
      <rtProperty name="samplingThPrty" type="PRIORITY"/>
      <rtProperty name="dataMtxCeiling" type="PRIORITY"
                        sameValueOf="dataMtx.ceiling"/>
      </description>
      ...
   </referencedImplementation>
</implementation>
...
```

4.6.1.3. Introducción de los puertos de sincronización

La estrategia para la declaración de puertos de sincronización es similar a la de los puertos de activación. Por cada requisito de sincronización se define una propiedad específica en el descriptor de la implementación del componente, en este caso de tipo *Mutex* o *Condition_Variable*. Al igual que en el caso anterior, estas propiedades juegan un doble papel:

- Sirven para asignar los valores que configuran los mecanismos de sincronización de modo que la planificabilidad de la aplicación sea adecuada.
- Permiten a las herramientas de generación de código reconocer estos requisitos, de manera que se incorpore al correspondiente contenedor el código necesario para suministrar los mecanismos requeridos cuando el componente es instanciado.

Al igual que en el caso anterior, será necesario extender el tipo *DataType* con estos dos nuevos tipos de propiedades y el tipo *Any* con los tipos de datos que permiten darle valor. En este caso, únicamente requieren la asignación de valores las propiedades de tipo *Mutex*, a las que se le ha de proporcionar el valor del parámetro de planificación, como el techo de prioridad, que se asigna al objeto de sincronización para su gestión.

En la tabla 4.1 se muestra también un ejemplo de declaración de un puerto de sincronización de tipo *Mutex*, *dataMtx*, en el componente *AdaScadaEngine*. El valor de techo de prioridad que se asigne al mutex otorgado influye de manera directa en el comportamiento temporal de la aplicación, luego por cada propiedad de este tipo se declara también una propiedad del modelo de tiempo real del componente, *dataMtxCeiling* en el ejemplo. El valor de dicho techo será calculado automáticamente por las herramientas de análisis de tiempo real y asignado automáticamente al campo *ceiling* de la propiedad *dataMtx* (gracias a la relación indicada a través del atributo *sameValueOf*).

4.6.1.4. Introducción del tipo TransactionId

RT-CCM permite realizar una asignación de parámetros de planificación independiente para cada invocación de un método de un componente, basada en el punto de la transacción desde la que se realiza cada invocación. Para poder realizar este tipo de asignación es necesario poder identificar de forma unívoca cada transacción que se ejecuta en una aplicación. Cada transacción se identifica con el *stimulusId* con el que inicia su ejecución:

- Aquellas transacciones correspondientes a la ejecución de puertos de activación periódicos, o puertos de tipo *OneShotActivation* que realicen alguna actividad interna continua en el componente, reciben como *stimulusId* inicial el que se asigna a la correspondiente propiedad con la que el puerto es declarado.
- En el caso de componentes de tipo cliente, esto es, aquellos que tienen la capacidad de lanzar transacciones de negocio en una aplicación, deben declarar como parámetros de configuración a nivel de su implementación los valores de *stimulusId* con los que dichas transacciones deben comenzar su flujo. Este tipo de componentes tienen la capacidad de modificar el valor del *stimulusId* desde su propio código de negocio (a través de variables de condición, como ya vimos anteriormente) y será en base a esas propiedades de configuración como identificarán cada transacción iniciada en ellos. Con el objetivo de definir y asignar valor a este tipo de propiedades, se incluye el tipo *Transaction_Id* dentro de la clase *DataType*. Asimismo, para poder asignarles valor, se define también un tipo *Transaction_Id*.

En ambos casos el valor de *stimulusId* asignado sirve de punto de partida para configurar las transformaciones que realizan los conectores en las operaciones invocadas a lo largo del flujo de cada transacción.

4.6.2. Descripción RT-D&C de una aplicación RT-CCM

Las extensiones vistas hasta el momento permiten declarar los requisitos de ejecución de los componentes RT-CCM. Una vez que dichos componentes son ensamblados como parte de una

aplicación ejecutada en una plataforma, se puede extraer el modelo de tiempo real de la aplicación y analizarlo. De tal análisis se obtienen los valores que configuran la ejecución de la aplicación, que deben ser asignados en el plan de despliegue.

Como muestra la figura 4.23 el modelo de tiempo real de un aplicación RT-CCM, generado a través de la serie de transformaciones que se vieron en capítulos anteriores, se utiliza como entrada de las herramientas del entorno MAST. En concreto, se utilizan las herramientas de asignación de parámetros de planificación, que buscarán una asignación que verifique los requisitos temporales de la aplicación. Como resultado de esta fase de análisis se genera un modelo planificable del sistema. Este modelo, junto al plan de despliegue inicial, sirven de entrada a una herramienta característica del entorno RT-CCM, que genera la configuración de los servicios del entorno, de las transformaciones a realizar por los conectores y de cada instancia de componente declarada en la aplicación. Esta configuración es asignada en el plan de despliegue final, haciendo uso de los formatos que se exponen a continuación.

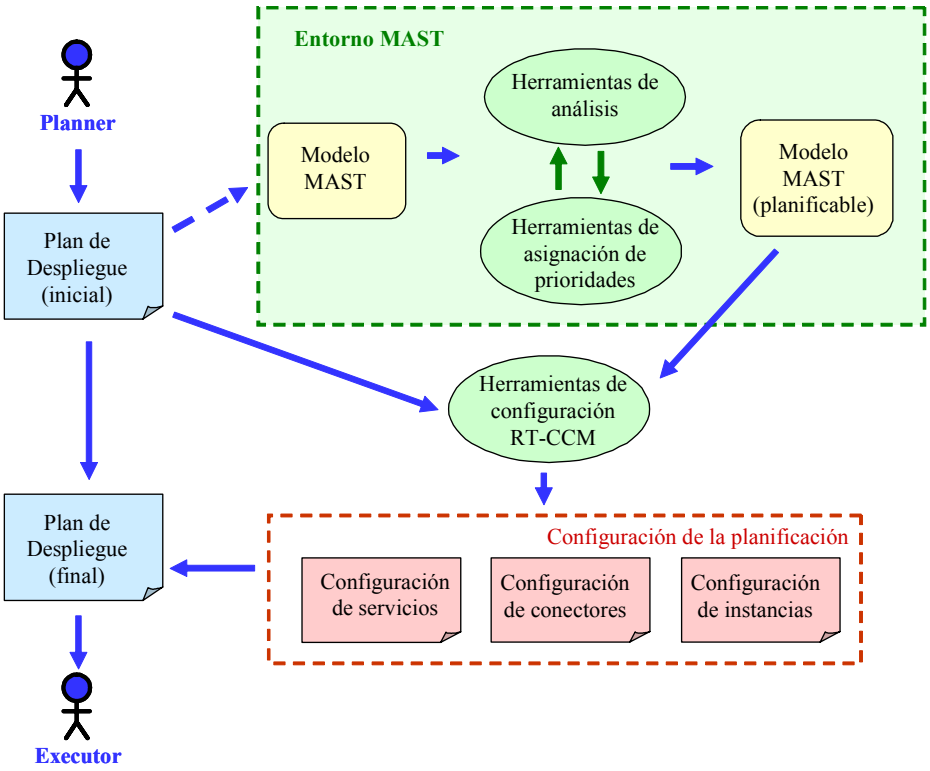


Figura 4.23: Configuración de la planificación de una aplicación RT-CCM

Este plan de despliegue final es manejado por el *executor*, que lo utiliza como entrada de las herramientas de lanzamiento. En base a la información incluida en el plan, las herramientas configuran los servicios e instancias de componentes y conectores de cada nodo, para a continuación, dar comienzo a la ejecución de la aplicación con garantías de que los plazos temporales se van a satisfacer.

4.6.2.1. Configuración de conectores

Además de sus propias propiedades de configuración relacionadas con la planificación (aquellas que se declaran en el descriptor del mecanismo de conexión), los conectores deben recibir la configuración que les permite realizar la asignación correcta de *stimulusId*. Para ello

se añade un nuevo campo, *schedulingConfig,* al elemento *ConnectionConfiguration* introducido en RT-D&C para cualificar las características de tiempo real de cada conexión entre componentes. A través de él se va a asignar a cada conector la tabla de transformaciones de *stimulusId* que debe almacenar. Como se observa en la figura 4.24, a través de elementos de tipo *SchedulingConfiguration* se asignan las transformaciones que se deben realizar por cada operación ofrecida a través del conector.

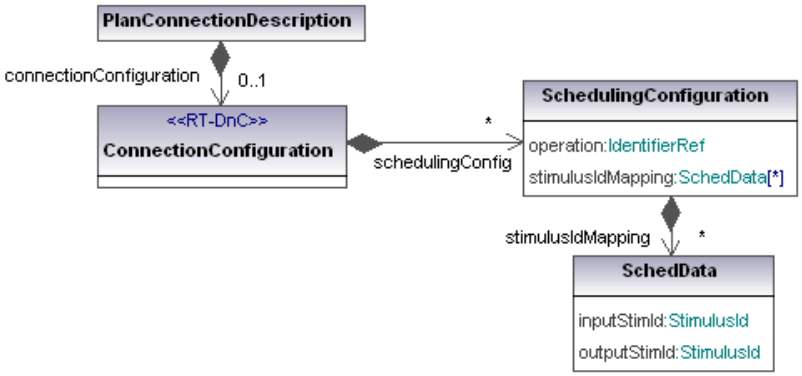


Figura 4.24: Configuración de la planificación a través de los conectores

La asignación de estos valores puede resultar muy compleja, especialmente si es la aplicación es distribuida. El objetivo es extraerlos del análisis del modelo de tiempo real que se genere para la aplicación y asignarlos de modo automático en el plan de despliegue final. Puede darse el caso de que haya operaciones para las que no se defina ninguna correspondencia, pues no sean nunca ejecutadas en la aplicación.

4.6.2.2. Configuración de los servicios de la plataforma

La definición en un plan de despliegue de la configuración de los servicios de la plataforma se realiza utilizando una estrategia extensible, que facilite la inclusión de futuros servicios que se puedan añadir a la tecnología. Por cada nodo que forme parte de la plataforma de ejecución de la aplicación se añade un elemento de tipo *QoSConfiguration* en el plan de despliegue final de la aplicación. Como muestra la figura 4.25, dicho elemento identifica al nodo al que se refiere y consiste en un conjunto de elementos de tipo *QoSServiceConfiguration*, a través de los que se realiza la configuración de cada uno de los servicios del entorno disponibles en el nodo. Esta clase es abstracta, de modo que para cada tipo de servicio se define una clase concreta que la extiende y que define la configuración concreta del servicio de acuerdo a sus características.

Actualmente se han definido cuatro extensiones:

- *SchedulingServiceConfiguration*: Es la clase a través de la que se configura el *SchedulingService*. Se le asigna cada uno de los mapeados entre valores de *stimulusId* y valores de parámetros de planificación (de alguna clase que extienda a *SchedParam*). El tipo de parámetros dependerá de la política de planificación utilizada en la plataforma.
- *CommunicationServiceConfiguration*: Es la clase a través de la que se configura el *CommunicationSchedulingService*. Se le asigna cada uno de los mapeados entre valores de *stimulusId* y valores concretos de parámetros de planificación de comunicación (tipos que extiendan a *CommSchedParam*), cuyo tipo dependerá en cada caso del tipo de mecanismo de comunicación utilizado.

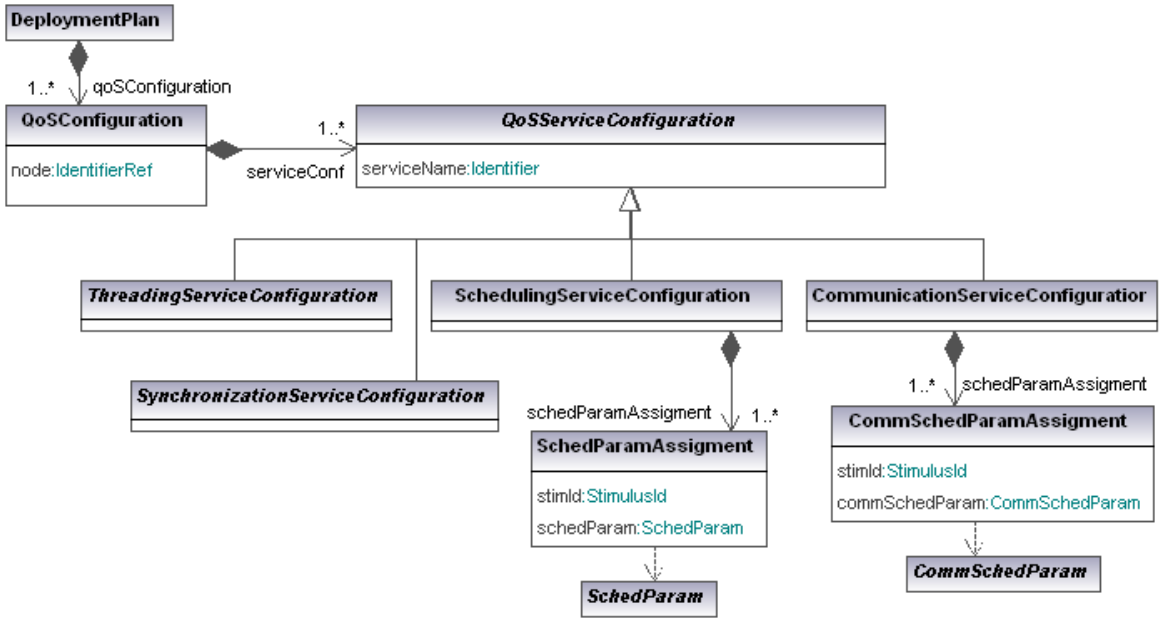


Figura 4.25: Soporte para la configuración de los servicios del entorno

- *ThreadingServiceConfiguration*: Es la clase a través de la que se configura el *ThreadingService*. Se define como abstracta a este nivel, pues sus parámetros de configuración dependerán del tipo de implementación que se elija para él en cada implementación concreta de RT-CCM.
- *SynchronizationServiceConfiguration*: Es la clase a través de la que se configura el *SynchronizationService*. Al igual que en el caso anterior, su configuración dependerá de cómo se implemente el servicio en cada tecnología concreta.

4.7. Conclusiones

En este capítulo se ha presentado la tecnología de componentes RT-CCM, que permite desarrollar aplicaciones basadas en componentes para las que se puede asegurar un comportamiento temporal predecible, y cuya ejecución puede ser planificada y configurada respetando la opacidad de los componentes, esto es, sin necesidad de acceder a su código interno.

La tecnología ofrece estas capacidades gracias a la utilización de un modelo contenedor/componente como base de su modelo de referencia. Con este modelo de programación, el código de negocio puede ser desarrollado de forma independiente de la plataforma de ejecución, siendo el contenedor quien adapta dicho código a la plataforma concreta sobre la que se vaya a ejecutar el componente. El código de este contenedor se genera automáticamente en base a los metadatos ofrecidos por el componente y a la plataforma en que va a ser ejecutado.

El modelo de referencia RT-CCM toma como base el modelo de LwCCM, pero lo extiende con una serie de mecanismos que van a garantizar la ejecución predecible del código de las aplicaciones. Gracias a estos mecanismos (puertos de activación, de sincronización, gestión del *stimulusId*, etc.) se consigue que todos los aspectos relacionados con la gestión de la planificación de una aplicación sean responsabilidad de los contenedores de los componentes, a través de una serie de servicios que se han definido en el entorno con este propósito.

La ventaja de la utilización de la estrategia que propone RT-CCM radica en que la configuración de estos servicios se realiza de manera automática, en base a los resultados obtenidos por las herramientas que analizan el modelo de tiempo real de la aplicación. De esta forma se libera a los diseñadores de la aplicación del proceso de configuración de la planificación de una aplicación, cuya complejidad en este caso es aún mayor que en el caso general, pues se desconoce el flujo de control interno de la aplicación al manejarse el código de los componentes de forma opaca.

Otra ventaja de la tecnología RT-CCM es su independencia de la infraestructura de comunicación utilizada para implementar comunicaciones remotas. Gracias al uso de conectores, el código de negocio de los componentes se desarrolla como si fuese a ser ejecutado siempre en entorno monoprocesador, sin necesidad de incluir código de gestión de comunicaciones. Cuando se declare una conexión remota entre componentes en el plan de despliegue de una aplicación, una herramienta se encargará de generar el código necesario para realizar la comunicación entre ellos, código que se encapsula en un elemento de tipo conector. Junto con el código se generará también el correspondiente modelo de tiempo real, necesario para obtener el modelo de la aplicación completa.

La tecnología se define de forma independiente de la plataforma de ejecución sobre la que va a ser ejecutada o del lenguaje de programación utilizado para desarrollar componentes y contenedores. Para adaptarla a un entorno de ejecución concreto será necesario desarrollar implementaciones de los diferentes servicios especificados, así como herramientas que generen el código de los contenedores y de la interfaz de negocio de los componentes en el lenguaje elegido.

5. Diseño de tiempo real de aplicaciones basadas en componentes

El diseño de tiempo real de una aplicación tiene por objetivo asignar los recursos y planificar la ejecución de sus actividades internas de manera que se garantice el cumplimiento de los requisitos temporales establecidos en su especificación.

En el caso de aplicaciones tradicionales, el diseñador puede gestionar diversos aspectos de la aplicación para conseguir su planificabilidad:

- El nivel de concurrencia, esto es, el número de threads en los que se planifica la ejecución de las actividades.
- La asignación a cada thread de las actividades que debe ejecutar.
- Los mecanismos de sincronización que coordinan la actividad de los diferentes threads cuando compiten por recursos protegidos o cuando se sincronizan para cooperar entre sí.
- La política de planificación con la que se decide el thread que debe ser activado de entre los que se encuentran listos para ser ejecutados en cada procesador o para transmitir un mensaje por las redes de comunicación.
- Los valores de los parámetros de planificación que se asignan a los threads y a los mecanismos de sincronización.

En el caso de aplicaciones basadas en componentes el diseñador gestiona la aplicación a un nivel de abstracción superior. Todos los aspectos previos son internos a los componentes, y debido a la opacidad con que son manejados, le son desconocidos e inaccesibles. En este caso, el diseñador sólo tiene capacidad para gestionar la ejecución de la aplicación a través de la información que se plasma en el plan de despliegue, el cual incluye:

- La selección de la implementación concreta de cada instancia de componente que se usa en la aplicación.
- La asignación del procesador en el que se ejecuta cada instancia de componente.
- La asignación del mecanismo de comunicación que se utiliza entre componentes instanciados en diferentes procesadores o para entre aquellos que estando en el mismo procesador requieren algún tipo especial de mecanismo.
- Los valores de configuración de las instancias de los componentes, de los recursos de la plataforma y de las conexiones.

En ambos casos, las decisiones que debe tomar el diseñador son complejas, por lo que necesita construir un modelo de comportamiento temporal sobre el que aplicar:

- herramientas de diseño, con las que obtener conjuntos válidos, y a ser posible óptimos, de los valores de configuración que controlan la planificación de la aplicación,
- y/o herramientas de análisis, que le permiten verificar la planificabilidad de la aplicación u obtener información sobre los aspectos del sistema que deben ser modificados para

conseguirla, en caso de que no sea posible hacer planificable la aplicación en la situación que se analiza.

En el caso tradicional, este modelo es construido al tiempo que se desarrolla el código; sin embargo, en el caso de aplicaciones basadas en componentes, el modelo se construye en dos fases. Durante el diseño del componente se construye un modelo parametrizable que describe el comportamiento de su código. Posteriormente, cuando el *planner* configura la aplicación, construye su modelo de tiempo real por composición de los modelos de los componentes, a fin de poder evaluar los parámetros de configuración que la hacen planificable sin conocer el código final.

En el capítulo 3, y tomando como referencia el proceso de desarrollo de aplicaciones basadas en componentes que se propone en la especificación D&C de OMG, se ha descrito la información que se incluye en los componentes y en la descripción de la plataforma a fin de que el diseñador configure todos los aspectos de una aplicación relacionados con el diseño de tiempo real. En este capítulo se analiza el proceso de diseño de tiempo real de aplicaciones basadas en componentes, dentro del marco de referencia que establece la especificación RT-D&C. El proceso se describe a nivel PIM, esto es, independientemente de la tecnología de componentes, de la metodología de modelado de tiempo real o de las herramientas de análisis concretas que se utilicen en él. En el siguiente capítulo se mostrará la adaptación de este proceso a una tecnología de componentes y una metodología de modelado concretas, a través de un ejemplo en el que se aplicarán todos los aspectos desarrollados en esta tesis.

5.1. Información gestionada en el proceso de diseño

En este apartado se delimita el proceso de diseño de tiempo real, identificando la información de la que se parte, junto con las fuentes de las que se obtiene, y la información que se genera, junto con los elementos contenedores en los que se introduce.

El diseño de tiempo real de una aplicación basada en componentes se realiza a través de un proceso iterativo, dirigido por modelos y asistido por herramientas. Cada iteración se inicia con la propuesta por parte del *planner* de un posible plan de despliegue. A continuación se elabora el modelo de tiempo real que corresponde a la aplicación tal y como se describe en ese plan de despliegue, y se analiza dicho modelo, verificando si la aplicación es planificable. Si no lo es, el *planner* deberá estudiar en qué aspectos debe ser modificado el plan de despliegue para que lo sea. En la figura 5.1se muestran los elementos de información que intervienen en este proceso de diseño de tiempo real.

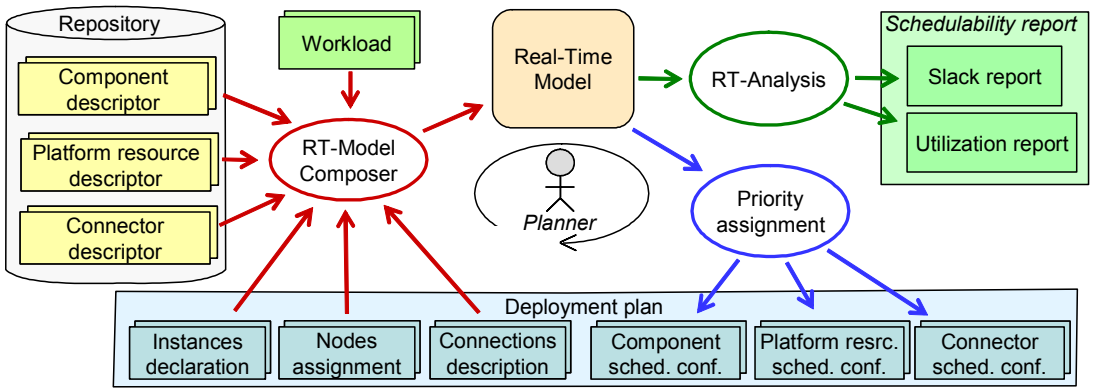


Figura 5.1: Elementos de información en el proceso de diseño de tiempo real

El modelo de tiempo real es construido a partir de la información estructural contenida en el plan de despliegue y la información sobre la actividad y los requisitos temporales de la aplicación contenida en la descripción de la carga de trabajo:

- En el plan de despliegue se declaran las instancias de los componentes que constituyen la aplicación, y en ellas se describen la implementación utilizada y los valores que se han asignado a sus propiedades de tiempo real. Con esta información se accede a los descriptores del comportamiento temporal de las implementaciones, que se encuentran en el repositorio, y se construyen los modelos de tiempo real de los componentes.
- En el plan de despliegue cada instancia de componente tiene asignado el procesador en el que va a ser ejecutada. Con esta información se accede en el repositorio a los descriptores de la capacidad de procesamiento de los nodos procesadores y se construye el modelo de tiempo real de los procesadores que ejecutan la aplicación.
- En el plan de despliegue se declaran y configuran las conexiones entre instancias de componentes. Con esta información se accede en el repositorio a los modelos de comportamiento temporal de cada mecanismo de comunicación que se puede utilizar en la aplicación. El modelo de cada conector se genera automáticamente en base a dicho descriptor y a la configuración asignada a la conexión.
- En el caso de conectores que comunican componentes situados en diferentes procesadores, el plan de despliegue referencia las redes de comunicación que se utilizan. A partir de esta información se accede en el repositorio a los descriptores de modelos de comportamiento temporal de las redes de comunicación, y los correspondientes modelos se incorporan al modelo de tiempo real de la aplicación.
- Cada carga de trabajo (workload) formulada por el *assembler* describe un contexto de análisis de la aplicación que se diseña, correspondiente a un modo de operación del sistema con requisitos de tiempo real, para el que se deben generar los valores de configuración que lo hagan planificable. El modelo de la carga de trabajo contiene la información necesaria para incorporar al modelo de tiempo real de la aplicación las transacciones que concurren en el modo de operación que se describe:
  - Contiene la declaración de las transacciones que se ejecutan concurrentemente en ese modo y los valores asignados a los parámetros declarados en sus correspondientes descriptores. Con esta información se obtienen las secuencias de actividades que se ejecutan en los diferentes componentes cuando las transacciones se activan.
  - Contiene el patrón de generación de los eventos externos que activan las transacciones y que constituyen el modelo de la carga de trabajo con que se ejecuta la aplicación.
  - Contiene la descripción de las restricciones temporales que deben cumplirse en cada transacción para que se satisfagan los requisitos de tiempo real de la aplicación cuando opera en ese modo.

Una vez construido el modelo de tiempo real, se analiza la planificabilidad de la aplicación. Se aplican diferentes herramientas de peor caso o de simulación que permiten determinar si la aplicación es planificable. En cualquier caso, se obtiene un informe sobre la planificabilidad que incluye datos sobre las holguras del sistema (positivas si la aplicación es planificable o negativas en caso contrario) y los niveles de utilización de los recursos (inferiores del 100% si la aplicación es planificable). Estas informaciones son muy útiles para que el *planner* tome decisiones sobre los aspectos estructurales del plan de despliegue que deben ser modificados en caso de que la aplicación no sea planificable.

Una segunda opción de diseño es hacer uso de las herramientas de asignación de parámetros de planificación (prioridades, techos de prioridades, plazos, etc.). Cuando el sistema es planificable, permite obtener configuraciones de planificación óptimas o heurísticamente optimizadas. Los parámetros de planificación que resultan de aplicar el proceso de diseño constituyen una configuración completa de la planificación de la aplicación, que garantiza en ejecución el cumplimiento de los requisitos temporales especificados. En el capítulo 3 se explicó como los diferentes elementos que describen un plan de despliegue han sido extendidos en RT-D&C con el objetivo de almacenar estos valores de configuración relativos a la planificabilidad, a fin de que posteriormente en los procesos de preparación y lanzamiento se hagan efectivos. Esta configuración consiste en:

- Configuración de planificación de cada instancia de componente que forma parte de la aplicación. Los valores correspondientes a las instancias de los componentes de la aplicación se establecen a través de los campos *specificConfigProperty*.
- Configuración de planificación de cada conexión entre dos componentes, que se establecen a través de los campos *configProperty* del elemento *ConnectionConfiguration* que se asocia a la declaración de la conexión.
- Configuración de planificación de los elementos de la plataforma. Estos valores se declaran a través de elementos de tipo *QoSConfiguration* asociados al plan de despliegue, uno por cada recurso de la plataforma (nodos, redes o recursos compartidos).

5.2. Proceso de diseño de tiempo real en RT-D&C

El proceso de diseño de tiempo real afecta a las diferentes fases del proceso de despliegue y configuración definido en D&C. En concreto, añade nuevas tareas que los agentes involucrados en el desarrollo de la aplicación deben realizar. A lo largo de esta sección, se explica cómo se superpone el proceso de diseño de tiempo real en el proceso de desarrollo tradicional.

5.2.1. Fase de configuración

El proceso de diseño de una aplicación basada en componentes comienza en la fase de configuración. En ella el *assembler* diseña la aplicación desde un punto de vista estructural, esto es, como un conjunto de componentes interconectados entre sí que implementan su especificación. En el caso de aplicaciones sin requisitos temporales, el *assembler* realiza su tarea en tres pasos:

- Identifica y agrega los componentes que cubren la funcionalidad especificada.
- Recursivamente, identifica y agrega nuevos componentes que implementan la funcionalidad requerida por los componentes agregados previamente.
- Propone los valores de los parámetros de configuración de negocio que corresponden a los requisitos funcionales de la aplicación.

En aplicaciones sin requisitos de tiempo real, la elección de componentes para una aplicación se basa exclusivamente en la funcionalidad de los componentes, esto es, en las interfaces ofrecidas y requeridas a través de sus puertos.

En el caso de aplicaciones de tiempo real el diseño estructural de la aplicación es similar. La diferencia se centra en que para incluir los requisitos temporales en la especificación de la aplicación, se utiliza una estrategia de naturaleza reactiva, en la que la funcionalidad se formula especificando los eventos a los que la aplicación responde (eventos hardware procedentes del entorno o eventos temporizados generados en el reloj) y las actividades que constituyen la

respuesta a cada uno de estos eventos. Siguiendo esta estrategia, la especificación temporal de la aplicación se realiza a través de la descripción del modelo de carga de trabajo, que describe los tiempos en que se generan los eventos de estímulo y los requisitos temporales que se establecen en las actividades de respuesta. Para formular los requisitos funcionales y temporales dentro de un modelo reactivo se utiliza el concepto de transacción, que describe todos los aspectos relativos a cada respuesta que se requiere en la aplicación: patrón de generación de los eventos, secuencia de actividades de la respuesta, y en su caso, restricciones en los tiempos de finalización de las actividades.

Como muestra la figura 5.2, el *assembler* que define una aplicación de tiempo real debe seleccionar y agrupar los componentes que constituyen la aplicación con el objetivo de cubrir la funcionalidad requerida en las transacciones definidas en su especificación. También debe considerar en la selección y ensamblado de los componentes su compatibilidad para construir un modelo de comportamiento temporal de la aplicación que permita predecir los tiempos de las respuestas a los eventos. Esto es, en el diseño de una aplicación de tiempo real, el *assembler* tiene que seleccionar los componentes y los valores de configuración para cubrir los siguientes tres objetivos:

- Implementar el modelo reactivo: La selección de los componentes debe hacerse de forma que cubran la atención a los eventos identificados en la especificación reactiva de la aplicación. Con este objetivo, el *assembler* decide qué componentes utiliza en la aplicación en base a la información que cada componente incluye acerca de las transacciones que se pueden iniciar en él, o lo que es lo mismo, a qué eventos tiene capacidad de responder. Esta información está disponible como metadatos en la especificación de un componente gracias a las extensiones introducida en RT-D&C.
- Cobertura de la funcionalidad: Con el objetivo de que se puedan ejecutar todas las actividades que tienen lugar en las transacciones, el *assembler* selecciona los componentes de forma que se cubran todos sus requisitos de componibilidad funcional (puertos requeridos). Al igual que en el caso tradicional, este proceso se realiza de forma recursiva, hasta que todos los requisitos de componibilidad son satisfechos.
- Garantía de predictibilidad temporal: En la selección de los componentes, el *assembler* debe verificar que cada componente a agregar tiene asociado un modelo de comportamiento temporal componible con el de los componentes conectados a él. Esta información está disponible también como metadatos en la interfaz externa de un componente gracias a las extensiones introducidas en RT-D&C.

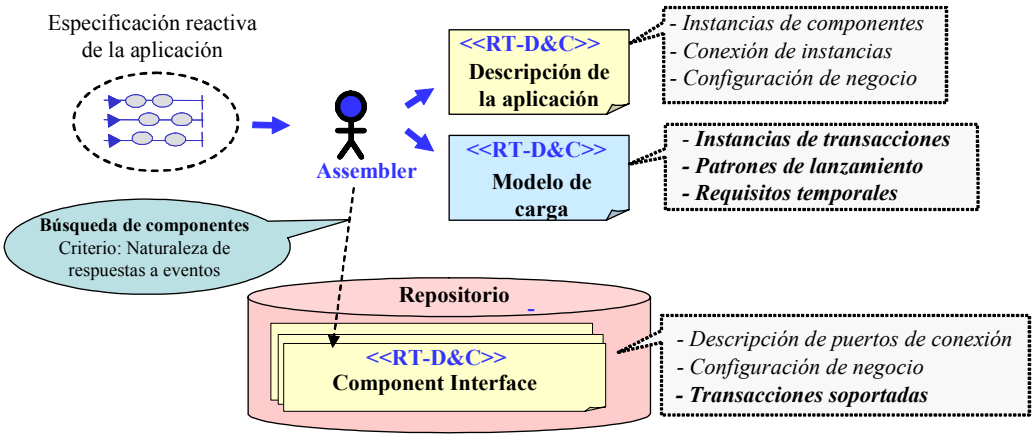


Figura 5.2: Aspectos de diseño de tiempo real en la fase de configuración

Es importante resaltar que la responsabilidad del *assembler* respecto a los requisitos funcionales y a los requisitos temporales es distinta. El *assembler* garantiza que la aplicación satisface los requisitos funcionales conectando componentes y asignándoles los valores de configuración de negocio apropiados. Sin embargo, respecto al comportamiento temporal, sólo garantiza que la aplicación construida dispondrá de un modelo de comportamiento temporal a través del que verificar si se satisfacen o no los requisitos temporales. Esto es consecuencia de que el *assembler* realiza su tarea de diseño utilizando sólo la información de las interfaces de los componentes, y al no conocer la implementación concreta de cada componente que se utiliza ni la plataforma sobre la que se ejecuta, no tiene capacidad de evaluar el comportamiento temporal de los componentes ni de la aplicación, y por tanto no puede garantizar que se satisfagan las restricciones temporales.

Aunque el *assembler* no garantiza el cumplimiento de los requisitos temporales, sí es responsable de formular las restricciones temporales en la especificación de la aplicación, a fin de que puedan ser tenidas en cuenta en futuras etapas del diseño. Por ello, como se muestra en la figura 5.2, una segunda tarea del *assembler* consiste en formular el modelo reactivo de la aplicación de acuerdo a la plantilla *ApplicationWorkload* definida en la especificación RT-D&C. En él se describen tres aspectos de la aplicación:

- El conjunto de instancias de transacciones de negocio que se ejecutan concurrentemente en la aplicación y que, o bien tienen requisitos temporales asignados, o bien, aunque no tengan requisitos, compiten por recursos compartidos con transacciones que sí las tienen. El *assembler* debe identificar a cuales de los descriptores de transacción declarados en los componentes corresponden los requisitos de respuesta de la aplicación y declarar una instancia de transacción por cada uno.
- La frecuencia y los patrones de generación de los eventos que activan cada transacción declarada (en caso de que éstos sean configurables).
- Los requisitos temporales que se imponen a la ejecución de cada una de las instancias de transacción declaradas, también en el caso de que sean configurables.

El *assembler* utilizará la información incluida en la descripción de la interfaz externa del componente para declarar las instancias de transacciones, y adaptarlas al contexto de la aplicación concreta mediante la asignación de valores a cada uno de los parámetros de configuración que tengan definidos. En concreto, a través de estos parámetros se asignarán las frecuencias de activación y los requisitos temporales.

5.2.2. Fase de planificación

La fase de planificación en la especificación D&C tiene como objetivo formular el plan de despliegue que incluye toda la información necesaria para ejecutar la aplicación sobre una plataforma determinada. Entre las tareas de esta fase se encuentran establecer la plataforma de ejecución, asignar a cada instancia de componente de la aplicación el nodo de procesamiento en el que se va a ejecutar, seleccionar la implementación de componente a utilizar y elegir el mecanismo de comunicación entre las instancias.

Desde el punto de vista del diseño de tiempo real la actividad del *planner* es la más importante, ya que ha de calcular los valores que hay que asignar a los parámetros de configuración de la planificación de las instancias de componentes y de los recursos de la plataforma para que se satisfagan todos los requisitos temporales de la aplicación cuando es ejecutada. Cuando no existen requisitos de tiempo real, el *assembler* es quien asigna valores a todos los parámetros de configuración de los componentes de acuerdo con la funcionalidad de negocio que se requiere

a la aplicación. Sin embargo, el *assembler* no tiene capacidad para asignar valor a los parámetros de configuración de la planificación, ya que estos parámetros son específicos de las implementaciones de las instancias y/o de la plataforma de ejecución, y además, se necesita realizar un análisis global de la aplicación para calcular sus valores. Dado que esta información sólo es conocida por el *planner*, tiene que ser suya la responsabilidad de su evaluación y asignación.

Para llevar a cabo esta tarea, el *planner* debe construir un modelo reactivo completo del sistema, haciendo uso de los modelos de los componentes y de la plataforma. Sobre ese modelo, aplicará algoritmos o herramientas de análisis y diseño de tiempo real que obtengan los valores adecuados para cada parámetro. La configuración de la planificación de una aplicación concierne a los componentes software, a los mecanismos de comunicación entre componentes y a los recursos del dominio.

Como se ha señalado anteriormente, esta fase se realiza de forma iterativa y dirigida por modelos. La figura 5.3 resume las tareas que realiza el *planner* en cada iteración:

- Basándose en la descripción de la aplicación realizada por el *assembler*, elabora una propuesta de plan de despliegue compatible con la plataforma de ejecución. Para ello, elige de entre todas las implementaciones disponibles de cada tipo de componente aquellas cuyos requisitos de despliegue son satisfechos por los recursos de la plataforma.
- Incorpora al plan de despliegue el mecanismo de comunicación a utilizar para cada conexión entre instancias. Este mecanismo no tiene efecto desde el punto de vista funcional, pero afecta de forma muy significativa en el comportamiento temporal de la aplicación. Cada mecanismo asignado referencia su correspondiente modelo de comportamiento temporal, el cual permite evaluar su influencia.
- Evalúa los valores de los parámetros de configuración de planificación. Esta es la tarea que constituye en sí el diseño de tiempo real y se trata como una subtarea que se describe en la siguiente sección.
  - Si el resultado de esta tarea consiste en una asignación de parámetros que garantiza el cumplimiento de los requisitos temporales, se elabora el plan de despliegue final incorporando al plan de despliegue inicial los valores de configuración obtenidos del análisis.
  - Si por el contrario, no se obtiene ninguna asignación de parámetros válida, se genera un informe de resultados que el *planner* consulta para decidir las modificaciones que

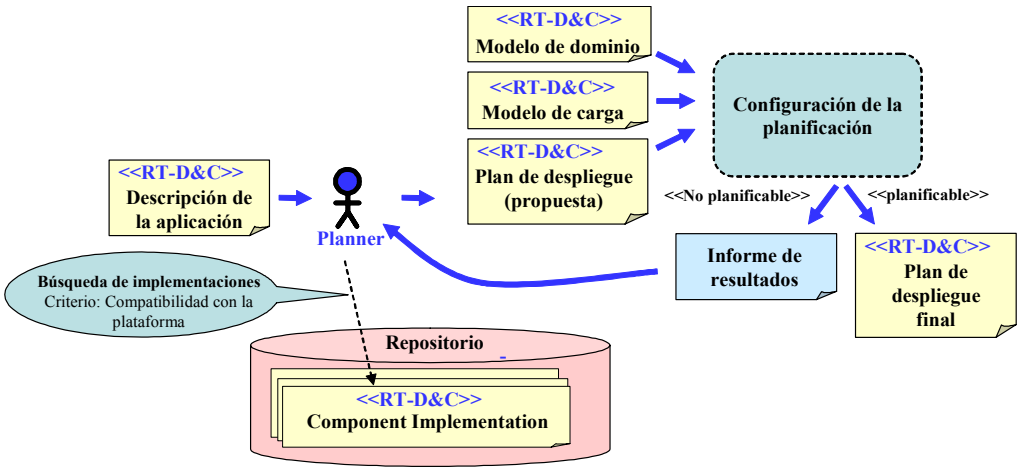


Figura 5.3: Aspectos de diseño de tiempo real en la fase de planificación

se han de hacer en el plan de despliegue en busca de hacer la aplicación planificable. Las estrategias que el *planner* puede seguir en este punto se explican en la sección 5.3.

Si el *planner*, una vez analizadas todas las posibilidades, no encuentra ningún despliegue válido, deberá comunicárselo bien al *assembler*, para que modifique los requisitos temporales de la aplicación o su arquitectura, o bien al administrador de la plataforma, para que modifique la plataforma en busca de una mayor capacidad.

5.2.2.1. Subfase de configuración de la planificación

El modelo temporal reactivo completo del sistema es la base del proceso de configuración de la planificación de la aplicación. Como se muestra en la figura 5.3, este proceso se considera como una subfase dentro de la fase de planificación. La figura 5.4 muestra con mayor detalle la secuencia de tareas que se llevan a cabo en ella. La mayoría de estas tareas son realizadas con el soporte de herramientas adecuadas.

A partir de los descriptores RT-D&C del sistema, esto es, del modelo de dominio, el plan de despliegue y el modelo de carga de trabajo, se genera el modelo reactivo de la aplicación. La herramienta *RTModelComposerTool* es dependiente de la metodología de modelado utilizada. En el capítulo 3 se explicó su funcionamiento para el caso de utilizar metodologías basadas en MAST.

El modelo generado sirve de entrada para el entorno de análisis de tiempo real, en el que se engloban tanto herramientas de análisis como herramientas de diseño:

- Las herramientas de análisis permiten bien certificar la planificabilidad de la aplicación dada una asignación concreta de valores de los parámetros de planificación, o bien, obtener otro tipo de información sobre el comportamiento del sistema, como puede ser porcentaje de utilización de los elementos de la plataforma, holguras, tiempos de respuesta promedio, datos sobre prestaciones, etc.
- Las herramientas de diseño permiten obtener asignaciones de valores de los parámetros de planificación que garantizan el cumplimiento de los requisitos temporales.

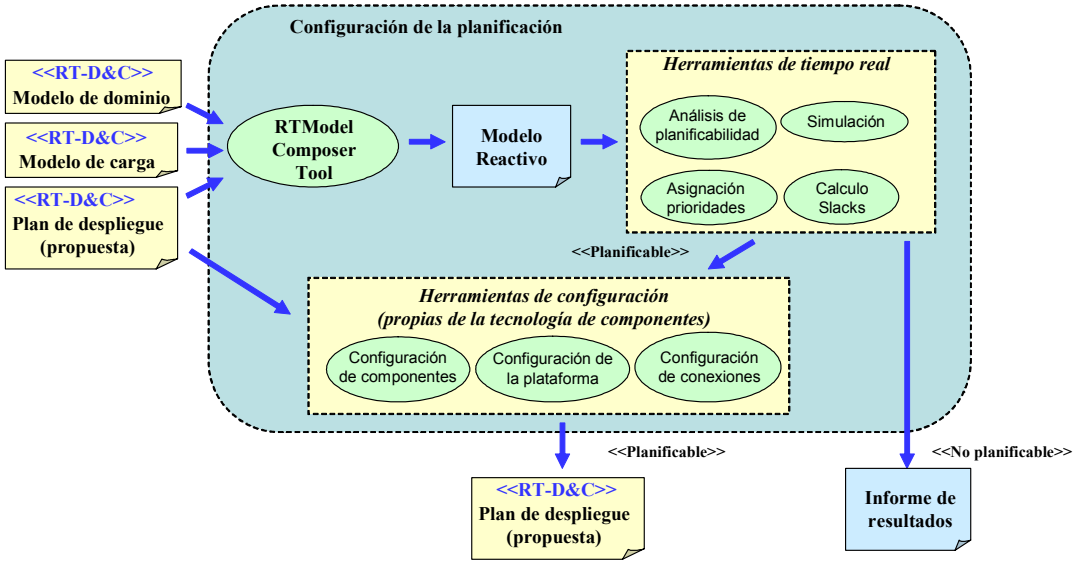


Figura 5.4: Subfase de configuración de la planificabilidad

Ambos tipos de herramientas se utilizan de forma coordinada. Si el *planner* ha hecho una asignación inicial de parámetros de planificación se pueden usar las herramientas de análisis para comprobar si el sistema es ya planificable. Si no lo es (como ocurrirá en la mayoría de los casos), se utilizan las herramientas de diseño para obtener una configuración válida. A continuación, se vuelven a utilizar las herramientas de análisis para obtener otros resultados sobre el sistema con dicha configuración.

Una vez analizada la aplicación, en función de que resulte o no planificable, se llevan a cabo diferentes acciones:

- Cuando la aplicación es planificable, y en base a los resultados obtenidos del análisis, se obtiene la configuración de planificación de la aplicación que hace que durante la ejecución se satisfagan en efecto los requisitos temporales. Para ello, se hace uso de un nuevo conjunto de herramientas que obtienen dicha configuración acorde a las características de la tecnología concreta de componentes que se esté utilizando. Estas herramientas son dependientes de la tecnología utilizada (PSM), y deberán ser desarrolladas para cada tecnología a la que se adapte este proceso de diseño. El modo en que, por ejemplo, se va a configurar una aplicación basada en la tecnología RT-CCM, será muy distinto del modo en que se configure una aplicación CCM pura, basada exclusivamente en CORBA. Mientras que en el primer caso es necesario obtener la configuración de los servicios de concurrencia, de sincronización, de asignación de *stimulusId*, etc., en el segundo se han de configurar las características de los POA o de los ORB involucrados. En cualquier caso, independientemente de la tecnología, esta configuración concierne siempre al mismo tipo de recursos, como son el número de threads o procesos a utilizar, los mecanismos de sincronización entre threads, las políticas de planificación, el número de canales de comunicación necesarios, etc.

  Al plan de despliegue inicial se le añaden los valores de configuración calculados, generándose así un plan de despliegue completo, que configura plenamente la aplicación para su ejecución.
- En el caso de que la aplicación no sea planificable, se genera un informe con los resultados obtenidos de las herramientas de análisis. En base a este informe el *planner* aplica estrategias de diseño (en la siguiente sección se proponen algunas de ellas) dirigidas a modificar algún aspecto del plan de despliegue que le permita conseguir la planificabilidad de la aplicación. Como resultado, se introducen cambios en el plan de despliegue, y se genera un nuevo modelo de la aplicación, sobre el que se vuelve a aplicar de nuevo el análisis. El proceso se repite de forma iterativa, hasta que se encuentre un despliegue válido, en el que se verifique el cumplimiento de los requisitos temporales de la aplicación.

5.2.3. Fase de preparación y lanzamiento

La fase de preparación tiene por objetivo preparar la plataforma para la ejecución de la aplicación, y habitualmente consiste en verificar la disponibilidad del código en los nodos. En el caso de un sistema de tiempo real el problema no es solo de disponibilidad del código, sino también de garantizar la disponibilidad de los recursos requeridos por la aplicación cuando ésta quiera hacer uso de ellos. A tal fin, el *executor*, basándose en los valores de los parámetros de configuración que se han incluido en el plan de despliegue, debe establecer o negociar con cada recurso del dominio (nodos, redes, recursos compartidos) la configuración necesaria para asegurar que los diferentes recursos estén disponibles en el modo que requiera la aplicación.

Las herramientas que utilice el *executor* para llevar a cabo esta tarea también son dependientes de la tecnología subyacente. Por ejemplo, en el caso de la tecnología RT-CCM, en esta fase se lleva a cabo la configuración de cada uno de los servicios del entorno, para asegurar la disponibilidad del número de threads que necesita la aplicación. Una vez configurada la plataforma, la aplicación se puede lanzar, instanciando y enlazando cada una de sus instancias, de igual modo a cómo se hace en el caso tradicional.

5.3. Estrategias de diseño

La estrategia a adoptar en caso de que la aplicación resulte no planificable es el aspecto en que más se diferencian los procesos de diseño de tiempo real de aplicaciones basadas en componentes de los procesos tradicionales. En una aplicación basada en componentes no es posible modificar la arquitectura interna de los componentes, ni tampoco el mapeado del modelo lógico al modelo de concurrencia de los componentes, al ser éste opaco. Por tanto, es necesario identificar nuevas estrategias o acciones que el *planner* puede aplicar para conseguir la planificabilidad de la aplicación.

En el caso de una aplicación monoprocesadora, las posibilidades de actuación del *planner* son muy limitadas. El único medio posible para modificar el comportamiento temporal de la aplicación consiste en sustituir alguna de las instancias de componente por otra instancia del mismo tipo (misma interfaz) pero de diferente implementación (y por tanto, con distinto comportamiento temporal). Para tomar esta decisión, el *planner* deberá contar con información cualitativa que sirva de base para la elección del componente a sustituir. Se proponen dos indicadores:

- Holgura de componente, medido como el porcentaje en que se puede incrementar el tiempo de ejecución de las operaciones de un componente manteniendo el sistema planificable, o en el sentido opuesto, el porcentaje en que se deberían decrementar para hacer el sistema planificable. Este tipo de resultado se podría obtener haciendo uso de herramientas de cálculo de holguras, que agrupasen todas las operaciones pertenecientes a una misma instancia de componente y evaluasen su efecto de forma conjunta. Se podría estudiar su efecto de forma global en el sistema o por transacción ejecutada en el sistema.
- Porcentaje de utilización de los recursos por parte de la aplicación o de cada componente, medido como el porcentaje del tiempo del recurso que está siendo utilizado por la aplicación o componentes. Este tipo de resultado únicamente puede obtenerse a través de herramientas de simulación.

Estas medidas deberían realizarse respecto del código realmente ejecutado en el componente, esto es, excluyendo aquellas partes del código que corresponden a invocaciones en componentes externos. Por otra parte, la elección de la nueva implementación no puede hacerse en base a ningún criterio específico, pues el comportamiento temporal sólo puede evaluarse cuando el nuevo componente se integra en la aplicación.

En el caso de una aplicación distribuida las posibilidades de modificación del comportamiento temporal de la aplicación aumentan. En este caso, además de la sustitución de implementaciones de componentes, también pueden realizarse modificaciones en el despliegue, esto es, trasladar unas instancias de un nodo a otro, o modificar los mecanismos de comunicación elegidos para las conexiones entre componentes. Para ayudar a la toma de decisiones en este caso, el *planner* puede basarse en otro tipo de indicadores:

- Utilización del procesador: Si uno de los procesadores de la plataforma posee una utilización mucho más alta que el resto, el *planner* puede trasladar alguna de las

instancias de componentes asignadas a él a alguno de los nodos menos utilizados, siempre que las características de los componentes así lo permitan.

- Porcentaje de utilización de la red por parte de una conexión, medido como el porcentaje del ancho de banda de la red que se emplea en enviar los mensajes correspondientes a una determinada conexión entre puertos de componentes. Este tipo de resultado se podría obtener haciendo uso de herramientas de cálculo de utilización, que agrupasen todas las operaciones de envío de mensajes implementadas a través de un determinado conector y evaluasen su efecto de forma conjunta. De este modo, se podría identificar la conexión que genera más tráfico en la red, y en consecuencia, pasar a un despliegue en que los componentes involucrados en dicha conexión se comuniquen de forma local.
- Porcentaje de atención al servicio de comunicaciones. Se puede utilizar para medir la influencia de uno u otro tipo de mecanismo de comunicación desde el punto de vista de la ocupación de procesador que generan por cada envío realizado.

Debido a la complejidad del proceso, no es posible ofrecer una estrategia de diseño sistemática, en la que en base a los resultados del análisis del modelo de la aplicación se pueda identificar la opción de despliegue correcta. Debido a que el comportamiento de los componentes que forman una aplicación no está desacoplado, no es posible saber de antemano las consecuencias de las modificaciones realizadas en el despliegue. Por ejemplo, si se identifica que una conexión genera un gran tráfico en la red, y como solución, se instancian los componentes involucrados en un mismo nodo, puede producirse una sobrecarga en dicho nodo, debido a la ejecución de las actividades del componente que primeramente se encontraba en el nodo remoto.

5.4. Entorno y herramientas de diseño

El proceso de diseño de tiempo real que se propone requiere la utilización de tres tipos de herramientas:

- La herramienta de composición de modelos, que permite obtener el modelo reactivo de la aplicación en base a los modelos de los elementos que la forman.
- Las herramientas de análisis y diseño de tiempo real que actúan sobre dicho modelo.
- Las herramientas que en base a los resultados del diseño y análisis obtienen la configuración de planificación de la aplicación.

Mientras que esta última depende de la tecnología de componentes utilizada, las dos primeras herramientas están ligadas a la metodología de modelado de tiempo real que se utilice para describir el comportamiento temporal de componentes y elementos de la plataforma. En nuestro caso, la metodología MAST.

En capítulos anteriores se ha mostrado el funcionamiento de la herramienta de composición de modelos, que a partir de la información incluida en el plan de despliegue es capaz de generar una descripción del sistema acorde a la metodología Mod-MAST, y a partir de ésta, generar el modelo MAST. Dicho modelo puede ser analizado con el conjunto de herramientas que se proporcionan en el entorno MAST. El modelo MAST de una aplicación es un modelo de bajo nivel, en el que no se considera la arquitectura software de alto nivel de la aplicación. Por ello es necesario definir estrategias que permitan adaptar la información que resulta del análisis de un modelo MAST, a la arquitectura de componentes de la aplicación, de modo que el *planner* pueda interpretar los resultados del análisis de forma simple, sin necesidad de ser experto en la metodología de modelado utilizada.

A continuación se exponen las diferentes herramientas disponibles en el entorno, y se explica el modo en que los resultados generados por ellas pueden utilizarse en el desarrollo de aplicaciones basadas en componentes. Las herramientas actualmente disponibles se pueden dividir en dos grandes grupos:

- Aquellas que realizan un análisis del sistema basado en técnicas analíticas de peor caso.
- Aquellas que analizan el sistema utilizando técnicas basadas en simulación.

Dentro del grupo de herramientas analíticas se ofrecen:

- Herramientas de análisis de planificabilidad: Permiten determinar si la plataforma en que se ejecuta la aplicación tiene capacidad para realizar todas las actividades que se planifican en la aplicación, satisfaciendo los requisitos temporales establecidos en ella. Las herramientas disponibles son aplicables tanto a sistemas monoprocesadores como distribuidos, utilizando políticas de planificación basadas en prioridades fijas, EDF o combinaciones de ambas.

  Los resultados de esta herramienta los usa el *planner* directamente para:

  - en caso de ser positivos, pasar a la elaboración del plan de despliegue final,
  - en caso de ser negativos, adoptar alguna estrategia que haga la aplicación planificable.
- Herramientas de cálculo de holguras: Permiten calcular el porcentaje en que se puede incrementar el tiempo de ejecución de las actividades, o disminuir la capacidad de los procesadores o las redes manteniendo la aplicación planificable. A través de los cálculos de holgura se puede determinar, en el caso de que el sistema sea planificable (holgura positiva), la capacidad sobrante de la plataforma, y en el caso de que el sistema no sea planificable (holgura negativa), cuán lejos se encuentra de serlo y qué recurso debe incrementar su capacidad.

  Asimismo, utilizando este tipo de herramientas podrían calcularse las holguras de componente y de conexión que se expusieron anteriormente, y que ayudan al *planner* a identificar aquellos componentes o mecanismos de comunicación que contribuyen más a la no planificabilidad de la aplicación.
- Herramientas de asignación de prioridades: Las herramientas anteriores operan sobre un modelo en que todas las prioridades ya están asignadas, esto es, son herramientas de análisis. MAST ofrece también un conjunto de herramientas de diseño, esto es, herramientas que tratan de encontrar asignaciones de prioridades para las que se verifique el cumplimiento de los requisitos temporales. En sistemas monoprocesadores se requiere asignar las prioridades a los diferentes threads que conducen las transacciones que se ejecutan en ellos. En estos casos, la herramienta establece la asignación utilizando la técnica *Deadline Monotonic* [LL82] o su extensión para el caso en que los plazos son superior a los periodos [AUD91]. En sistemas multiprocesadores y distribuidos el problema de asignar prioridades es más complejo, puesto que hay interrelaciones muy fuertes entre los tiempos de respuesta de diferentes recursos, y el número de prioridades que deben asignarse es muy elevado. En estos casos, no sólo hay que asignar una prioridad por cada transacción de la aplicación, sino que por cada invocación remota entre componentes de diferentes nodos, hay que definir al menos tres prioridades más: la del mensaje que realiza la invocación, la del proceso remoto que lo ejecuta y la del mensaje que retorna los resultados. En este caso el entorno aporta herramientas heurísticas, basadas en la aplicación del análisis de planificabilidad de forma iterativa. De entre ellas, la basada en el algoritmo HOPA [GG05] es la que ofrece mejores resultados.

Estas herramientas pueden ser utilizadas en dos situaciones:

- Cuando la aplicación no es planificable, en búsqueda de una asignación que haga la aplicación cumplir sus requisitos.
- Cuando la aplicación es planificable pero se trata de buscar una asignación más óptima, que de lugar a una utilización menor de la plataforma.

Estas herramientas son básicas en el proceso de diseño de aplicaciones basadas en componentes, pues al no conocerse el flujo de control de las actividades ejecutadas, es aún más difícil realizar una asignación manual que en el caso tradicional.

Los valores obtenidos por estas herramientas nunca van a ser procesadas directamente por el *planner*, pues él no accede al modelo interno de la aplicación. Sin embargo, son básicos para que la herramienta de configuración de la planificación pueda extraer los valores a asignar a las instancias, conexiones, etc., en el plan de despliegue final.

La utilización de técnicas de peor caso en el análisis de sistemas de tiempo real certifica que el sistema va a cumplir siempre sus plazos, sin embargo los resultados de su aplicación son casi siempre pesimistas. En el caso de sistemas con requisitos de tipo laxo, en los que se permite un porcentaje de plazos perdidos, o cuando el objetivo de la herramienta no es garantizar la planificabilidad, sino orientar al *planner* en por qué no es planificable, puede resultar útil la aplicación de otro tipo de técnicas. Este tipo de técnicas están basadas en la simulación del modelo de tiempo real del sistema. La utilización de la simulación ofrece otras ventajas:

- Proporciona capacidad de análisis para sistemas en los que coexistan requisitos temporales estrictos, laxos o especificados como niveles de calidad de servicio. Esto abre nuevas posibilidades al diseñador de sistemas de tiempo real ya que le permite abordar de forma conjunta y equilibrada todos los tipos de requisitos.
- Proporciona capacidad para analizar de forma general cualquier modelo. Con ello se libera al diseñador de tener que limitar la estructura de sus sistemas a aquellas que satisfacen el conjunto de restricciones que imponen las actuales herramientas analíticas. Como ejemplo principal, gracias a las herramientas de simulación se van a poder analizar sistemas basados en transacciones con múltiples eventos de disparo, para los cuales todavía no se han implementado herramientas analíticas en el entorno MAST.
- Ayuda a la depuración de errores estructurales de los sistemas de tiempo real, al proporcionar trazas que permiten documentar los escenarios en los que el fallo se ha producido.
- Constituye un medio de evaluación de los niveles de pesimismo que introducen las herramientas analíticas.

Por el contrario, la utilización de la simulación en el diseño de sistemas de tiempo real tiene dos problemas importantes:

- Es incapaz de analizar el peor caso, y en consecuencia no puede utilizarse para certificar de forma absoluta el cumplimiento de los requisitos en sistemas de tiempo real estrictos.
- Hace uso de una gran capacidad de procesamiento y por tanto no suele ser útil en los casos de validación “en vivo” que se necesitan en el análisis de sistemas con carga dinámica.

Como trabajo previo a esta tesis se amplió la capacidad del entorno MAST, que hasta el momento sólo contaba con herramientas analíticas, dotándolo de una herramienta de análisis basada en simulación. La herramienta, denominada SIM_MAST [LDM04][SMAST], utiliza el

mismo modelo de entrada que el resto de herramientas MAST, pero genera otro tipo de resultados:

- Para cada transacción ejecutada en el sistema, evalúa sus tiempos promedio de ejecución, así como los de peor y mejor caso, que pueden ser contrastados con los obtenidos a través del análisis de peor caso.
- Obtiene los valores de utilización de los procesadores, las redes de comunicación, y los recursos compartidos, permitiendo además distinguir las causas a las que se debe la ocupación. Por ejemplo, en un procesador se puede distinguir el porcentaje empleado en atención a interrupciones, a gestión del temporizador, a actividades de la propia aplicación, etc.

  Estas herramientas se podrían particularizar para que calculasen la utilización debida a la atención de un determinado componente o a una conexión entre componentes. Estos valores, como se indicó anteriormente, sirven al *planner* para decidir los cambios a efectuar en el despliegue de la aplicación en búsqueda de la planificabilidad de la misma.
- Obtiene una traza detallada de la ejecución del modelo, que puede ser empleada para identificar posibles bloqueos o errores en la ejecución del sistema (de forma previa a su ejecución real).

5.5. Conclusiones

En este apartado se presenta el proceso de diseño de tiempo real de una aplicación basada en componentes y especificada de acuerdo a un modelo reactivo. Las diferentes etapas a seguir para obtener una configuración de la aplicación que satisfaga sus requisitos temporales se integran dentro del marco de desarrollo definido por la especificación D&C, y haciendo uso de las extensiones que se introdujeron en el capítulo 3.

El proceso se define a nivel PIM, lo que favorece su adaptación a diferentes tecnologías de componentes e incluso, metodologías de modelado. A lo largo del capítulo se han identificado cuáles de las herramientas utilizadas deben ser adaptadas en cada caso, aunque se ha particularizado el caso de las herramientas de análisis a aquellas disponibles en el entorno MAST.

6. Ejemplo de diseño en Ada-CCM

En este capítulo se presenta un ejemplo completo en el que se aplican todos los conceptos desarrollados a lo largo de la tesis al desarrollo de una aplicación de tiempo real basada en componentes. La aplicación se desarrolla para la tecnología Ada-CCM, que constituye una implementación sobre lenguaje Ada 2005 de la tecnología RT-CCM expuesta en el capítulo 4. Antes de comenzar con el ejemplo propiamente dicho, se exponen algunas características propias de esta tecnología.

6.1. Ada-CCM: Implementación Ada 2005 de la tecnología RT-CCM

El lenguaje Ada está especialmente indicado para el desarrollo de aplicaciones de tiempo real, pues ofrece soporte directo para la formulación del modelo de concurrencia, las políticas de planificación y los mecanismos de sincronización. Utilizándolo como base para una tecnología de componentes podemos gestionar desde el propio lenguaje un comportamiento temporal predecible de las aplicaciones. Además, el lenguaje Ada ofrece un modelo propio de distribución, pudiéndose formular desde él los mecanismos de interacción entre componentes desplegados en diferentes procesadores, sin necesidad de utilizar middlewares más pesados. Sin embargo, su utilización en el campo de las tecnologías de componentes de tiempo real ha sido prácticamente nula, debido en gran parte a la falta de soporte para herencia múltiple e interfaces en Ada 95 [Ada95].

La aparición de la última versión del lenguaje, Ada 2005 [Ada05], y el desarrollo de plataformas que lo soportan, constituyen una nueva opción para implementar tecnologías de componentes de tiempo real. La principal aportación de la nueva versión, que convierte al lenguaje en adecuado para el desarrollo de estas tecnologías, es el soporte para la programación con interfaces. El concepto de interfaz es básico en el desarrollo de una tecnología de componentes:

- Sirve como mecanismo para encapsular los servicios que ofrecen los componentes (facetas en RT-CCM) o para formular las referencias a los servicios que un componente requiere (receptáculos en RT-CCM).
- La utilización de interfaces facilita la implementación de la herencia múltiple, que representa un concepto básico en una tecnología de componentes, ya que permite que un componente herede características tanto de la tecnología con la que se implementa como del dominio de aplicación al que pertenece por funcionalidad. En Ada 95, la implementación de la herencia múltiple era posible en algunos casos, pero siempre a través de estructuras complejas basadas en paquetes genéricos y discriminantes [BAR07].

Haciendo uso de los recursos ofrecidos por Ada 2005 se ha desarrollado una tecnología de componentes que sigue la especificación RT-CCM propuesta en esta tesis. Ventajas relevantes del uso de esta implementación como forma de validar RT-CCM son que no se necesita usar ningún middleware específico de distribución y que puede ser utilizada en plataformas empotradas con perfil mínimo. Un handicap de esta alternativa como forma de validación es que

se renuncia al desarrollo de aplicaciones con componentes desarrollados en diferentes lenguajes, aunque este problema podría solucionarse desarrollando conectores que permitiesen la comunicación entre componentes Ada y componentes elaborados utilizando otros lenguajes.

En esta sección se detallan las características de Ada-CCM, así como las pautas de diseño que sigue para desarrollar el código de negocio de los componentes y el código de los contenedores, conectores y servicios del entorno. El diseño de la tecnología es especialmente sencillo ya que no se basa en ningún middleware de distribución y utiliza únicamente los mecanismos del lenguaje para implementar todos los elementos de la tecnología.

Para asegurar la predictibilidad temporal y la adaptación a sistemas de perfil mínimo, la versión actual de Ada-CCM se desarrolla sobre nodos que operan con el sistema operativo MaRTE OS [AG01][MaROS], que ofrece requisitos de tiempo real estricto. Cuando la plataforma es distribuida se utilizan servicios de comunicaciones basados en Ethernet con protocolo RT-EP [MG05] o comunicación por bus CAN, los cuales satisfacen requisitos de tiempo real estricto y están soportados por los nodos MaRTE OS. Asimismo, como resume la figura 6.1, en Ada-CCM se utiliza RT-D&C para formular los descriptores de componentes y aplicaciones en los que se basan las herramientas de generación. Por último, la formulación de los modelos de tiempo real de los componentes se realiza utilizando la metodología CBS-MAST.

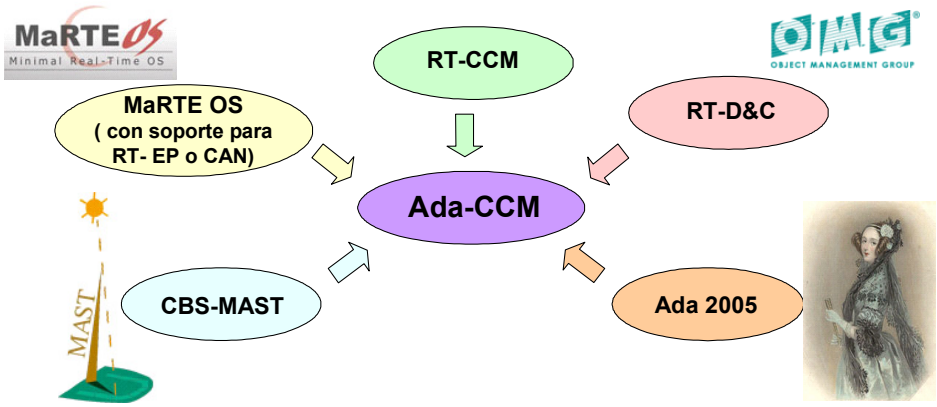


Figura 6.1: Principales características de Ada-CCM

6.1.1. Mapeado de idl a Ada 2005

Una de las bases de la tecnología RT-CCM es la disponibilidad de herramientas que generan automáticamente, en base a la descripción RT-D&C de los componentes, el código completo de los contenedores y de la interfaz que debe implementar el código de negocio. La declaración de las interfaces implementadas por los puertos de un componente se realiza a través del lenguaje IDL, para el que existe un mapeado estándar a Ada 95 [AMS01], elaborado por OMG. Sin embargo, la gran ventaja de la utilización de Ada 2005 es la aparición de las interfaces como elementos constructivos del lenguaje. La interacción entre componentes Ada-CCM se va a basar en el intercambio de punteros a dichas interfaces, por lo que ha sido necesario definir un mapeado de IDL a Ada 2005, con el objetivo de desarrollar las herramientas de generación de código. Aunque no se ha abordado el mapeado de forma completa, sino sólo el de aquellos aspectos requeridos en esta tecnología, sí se han sentado las bases para una futura formalización.

Aunque no es objetivo de este apartado mostrar en detalle las reglas del nuevo mapeado, sí se muestra un ejemplo de la principal modificación que se introduce: el mapeado de una interfaz IDL a una interfaz Ada. En la figura 6.2 se muestra un ejemplo de mapeado de dos interfaces

IDL (*InterfaceName* e *InterfaceName2*) declaradas dentro de un módulo IDL (*ModuleName*). Por cada módulo se genera un paquete Ada de nombre igual al nombre del módulo. Todas las interfaces definidas dentro de dicho módulo se mapean a paquetes hijos de nombre igual al nombre de la interfaz, en los que se definen dos tipos:

- El tipo *Face*, que representa la propia interfaz.
- El tipo *Ref*, que representa un puntero a la interfaz.

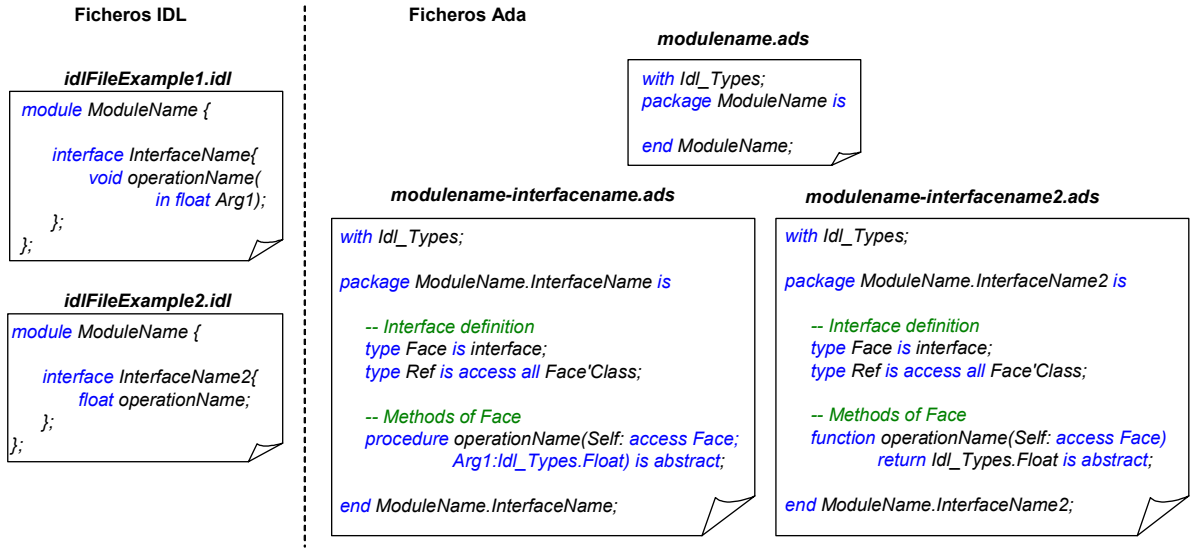


Figura 6.2: Mapeado de interfaces IDL a Ada 2005

Todas las operaciones definidas para la interfaz se mapean a operaciones primitivas del tipo *Face*, para lo que es necesario mapear a su vez todos los tipos que se utilicen como argumentos de dichas operaciones. Por el momento, el mapeado definido da soporte a los tipos primitivos IDL, a secuencias, arrays y estructuras de datos (*record* en Ada).

En resumen, una interfaz identificada en IDL como *ModuleName::InterfaceName*, se mapea a una interfaz Ada identificada como *ModuleName.InterfaceName.Face*. En caso de existir dependencias entre interfaces, deberá añadirse a cada paquete la cláusula *with* correspondiente. Los tipos declarados bien a nivel de módulo, o bien a nivel de interfaz, se declaran en el paquete raíz (el correspondiente al módulo), pues todas las interfaces deben tener visibilidad sobre ellos.

6.1.2. Implementación del código de negocio en Ada-CCM

El código de negocio de un componente en Ada-CCM sigue las reglas expuestas en el capítulo 4, esto es, debe implementar la correspondiente interfaz de negocio y basarse únicamente en código pasivo, sin creación de threads u objetos protegidos. La interfaz de negocio se genera de acuerdo a las reglas que se establecieron también en el capítulo 4.

A modo de ejemplo se muestra el código Ada de la interfaz de negocio correspondiente a la implementación *AdaScadaEngine*, que se ha venido utilizando como ejemplo a lo largo de la tesis, y que se explicará más en detalle como parte del ejemplo posterior. En la figura 6.3 se resumen sus principales características: facetas, receptáculos, puertos de activación y propiedades de configuración, de modo que se puede comprobar cómo cada una de ellas se refleja en la interfaz de negocio.

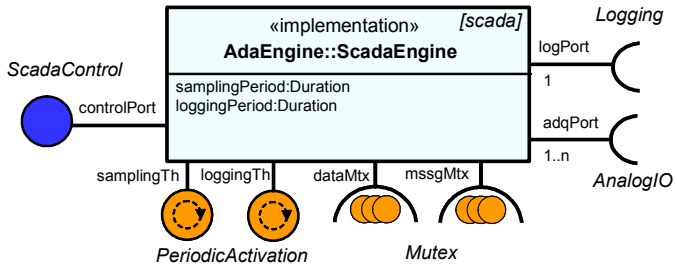


Figura 6.3: *AdaScadaEngine*: Implementación Ada del componente *ScadaEngine*

La interfaz, que se muestra en la tabla 6.1, se declara en un paquete hijo del dominio (o módulo) al que pertenece el componente, *scada* en este caso. De esta forma se tiene acceso desde ella a todas las interfaces y tipos característicos del dominio. El nombre completo de la interfaz correspondiente en este caso es *scada.AdaScadaEngine_Mngr.Face*. Como se observa en su declaración, toda interfaz de negocio en Ada-CCM hereda de la interfaz *RT_CCM.Face*, que agrupa los métodos necesarios para la gestión del ciclo de vida del componente: *activate()*,

Tabla 6.1: Código de la interfaz de negocio de la implementación AdaScadaEngine

```
with RT_CCM; with Idl_Types;
with scada.ScadaControl; with io.AnalogIO; with db.Logging;

package scada.AdaScadaEngine_Mngr is

  type Face is Interface and RT_CCM.Face;
  type Ref is access all Face'Class;

  -- Facets access --
  function Get_controlport(Self: access Face)
                    return scada.ScadaControl.Ref is abstract;
  -- Receptacles connection --
  procedure adqport_Cardinality(Self: access Face; NumObj:
                    idl_types.Unsigned_Short) is abstract;
  procedure adqport_Connect(Self: access Face; Index:idl_types.Unsigned_Short;
                                    Face: io.AnalogIO.Ref) is abstract;
  procedure logport_Connect(Self: access Face;
                                    Face: db.Logging.Ref) is abstract;
  -- Attributes accessors and mutators --
  function Get_samplingPeriod(Self: access Face)
                                    return Idl_Types.Float is abstract;
  procedure Set_samplingperiod(Self: access Face;
                                    Attribute: in Idl_Types.Float) is abstract;
  function Get_loggingPeriod(Self: access Face)
                                    return Idl_Types.Float is abstract;
  procedure Set_loggingperiod(Self: access Face;
                                    Attribute: in Idl_Types.Float) is abstract;
-- Activation Ports access --
  function Get_samplingTh(Self: access Face)
                                    return RT_CCM.PeriodicActivation_Ref is abstract;
  function Get_loggingTh(Self: access Face)
                                    return RT_CCM.PeriodicActivation_Ref is abstract;
-- Synchronization Ports connection --
  procedure set_dataMtx(Self:access Face;
                                    mutex: RT_CCM.Mutex.Ref) is abstract;
  procedure set_mssgMtx(Self:access Face;
                                    mutex: RT_CCM.Mutex.Ref) is abstract;
end scada.AdaScadaEngine_Mngr;
```

passivate(), *remove()*, etc. En el paquete RT_CCM se definen también otros recursos comunes, como las interfaces correspondientes a los puertos de activación (*PeriodicActivation* y *OneShotActivation*) o a los puertos de sincronización (*Mutex* y *ConditionVariable*).

6.1.3. Estructura del contenedor Ada-CCM

La estructura interna de los contenedores de componentes en la tecnología Ada-CCM es muy simple. Al no utilizarse ningún middleware, las conexiones locales entre componentes o las de componentes con conectores se implementan directamente a través de punteros a interfaces Ada. La estructura del *wrapper*, o adaptador, del componente es la que se muestra en la figura 6.4. Todas las interfaces que se muestran corresponden a interfaces puras Ada. El adaptador implementa la interfaz equivalente del componente, denominada *ScadaEngine (scada.ScadaEngine.Face* en Ada*)*, que extiende a su vez a la interfaz *CCM_Object*. Para poder desarrollar su funcionalidad el adaptador incluye:

- El ejecutor del componente (*AdaScadaEngine_Exec*), a través de la cual se accede a la implementación del código de negocio y cuyas normas de generación se establecen también en LwCCM.
- Por cada faceta del componente se crea una clase de envoltura, como *controlPort_Wrapper*. Estas clases serán las que capturen las llamadas que se realicen sobre las facetas del componente y las trasladen a las correspondientes implementaciones a través del ejecutor.
- El contexto del componente, *ScadaEngine_Context*, a través del que las invocaciones realizadas desde el código de negocio en los receptáculos se traducen en invocaciones en los correspondientes componentes conectados.
- Por cada puerto de activación y sincronización se crea una estructura de datos, como *samplingTh* y *dataMtx*, que almacena toda la información que se necesita para realizar las correspondientes peticiones a los servicios del entorno, como prioridades, valores de *stimulusId*, periodos, etc. Como muestra la figura, el adaptador tiene acceso directo a dichos servicios, que como se explicará más adelante, se han implementado como librerías estáticas.

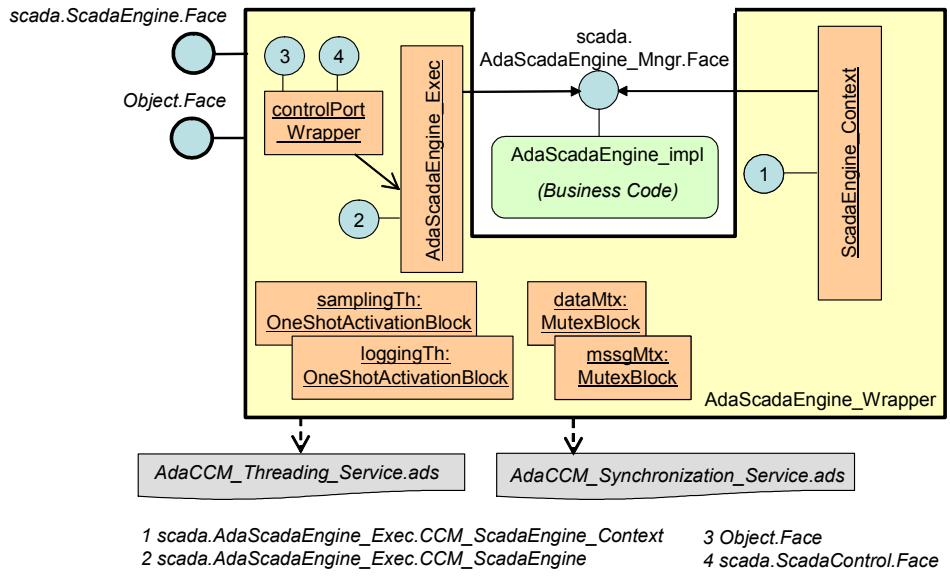


Figura 6.4: Estructura del contenedor de un componente en Ada-CCM

6.1.4. Implementación de los servicios del entorno en Ada-CCM

Los servicios *ThreadingService*, *SynchronizationService*, *SchedulingService* y *Communication SchedulingService* se han implementado en el caso de Ada-CCM como librerías estáticas, ya que sólo se requiere una instancia de cada uno por cada nodo que forma la plataforma de ejecución. En la implementación actual de Ada-CCM, la política de planificación que se considera es basada en prioridades, por lo que la clase *SchedParam*, que en RT-CCM se define como abstracta, se concreta en este caso al tipo Ada *System.Priority*.

6.1.4.1. Implementación del ThreadingService en Ada-CCM

La figura 6.5 representa la estructura del servicio *ThreadingService* en Ada-CCM. Se implementa como una librería estática (paquete *AdaCCMThreadingService*) que ofrece los dos métodos definidos en la especificación del *ThreadingService* más un método utilizado para la configuración del servicio, *configure()*. Está basado en un pool de threads que se crea durante la fase de configuración de la aplicación, evitando así la creación dinámica de threads. Los parámetros que se requieren para configurar el servicio son el número de threads del pool y la prioridad de base con los que se crean los threads.

Internamente el servicio consta de un objeto de tipo *AdaCCM_Thread_Pool_Executor*. Se trata de una versión modificada de la clase *Thread_Pool_Executor* definida por Burns y Wellings en [BW07], que implementa la interfaz *Executor* a través de un pool de threads de tamaño configurable. Las extensiones realizadas sobre la versión de Burns y Wellings permiten:

- recibir como parámetro de la función *execute*, un objeto de tipo *OneShotActivation* o *PeriodicActivation*, en lugar de un objeto de tipo *Callable*,
- definir una prioridad de base para todos los threads del pool,
- retornar un objeto de tipo *ExecutionController* en cada invocación del procedimiento *execute*, a través del que se puede controlar el flujo de ejecución del thread otorgado,
- asignar a cada thread utilizado el valor de *stimulusId* con el que debe comenzar su ejecución, que debe ser transmitido a lo largo de toda la cadena de actividades que se generen a lo largo de las transacciones y que se utiliza como clave para asignar los parámetros de planificación en base a él.

Por último cabe resaltar que la interfaz *ExecutionController* se ha definido como una *Synchronized_Interface* (nuevo elemento introducido en Ada 2005), lo que obliga a que aquellos objetos que la implementen deban asegurar un acceso protegido a sus métodos (se implementen, por tanto, como objetos protegidos Ada). Se han definido dos clases que implementan dicha interfaz, una por cada tipo de puerto definido en Ada-CCM.

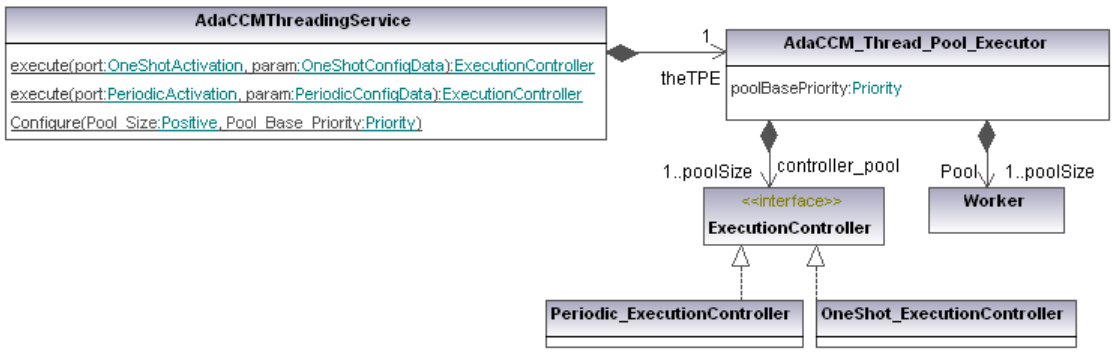


Figura 6.5: Implementación del *ThreadingService* en Ada-CCM

6.1.4.2. Implementación del SynchronizationService en Ada-CCM

Aunque podrían haberse implementado haciendo uso exclusivamente de elementos del lenguaje, la implementación del servicio de sincronización se ha llevado a cabo haciendo uso de los paquetes de MaRTE OS *posix.Mutexes* y *posix.Condition_Variables*, en los que se define una interfaz Ada para la creación y gestión de elementos de tipo *Mutex* y *Condition_Variable* de POSIX, respectivamente.

Como muestra la figura 6.6, los elementos que implementan las interfaces *Mutex* y *ConditionVariable* en Ada-CCM, denominados *Mutex_Impl* y *ConditionVariable_Impl*, son tipos que encapsulan objetos de los tipos *POSIX_Mutexes.Mutex_Descriptor* y *POSIX_Condition_Variables.Condition_Descriptor* respectivamente. Las llamadas a los métodos en cada interfaz (*lock*, *wait*, etc.) son redirigidas a los métodos homónimos implementados por dichos objetos. Los objetos de tipo *ConditionVariable_Impl* necesitan tener acceso a la implementación del *SchedulingService* para poder gestionar los valores de *stimulusId* de acuerdo a las reglas que se explicaron en el capítulo 4.

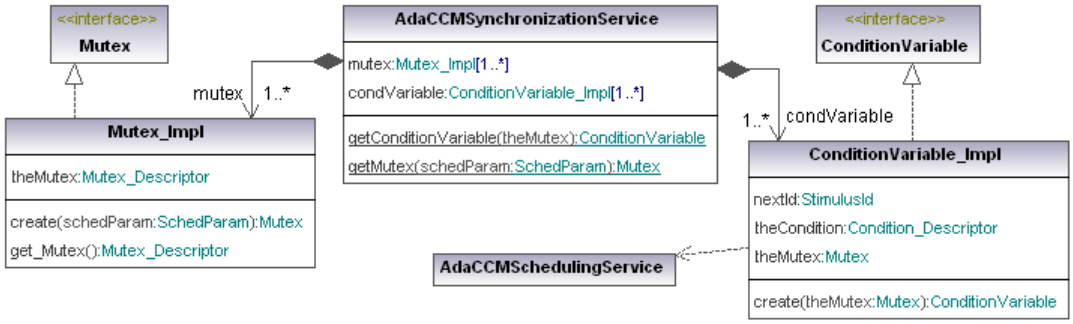


Figura 6.6: Implementación del *SynchronizationService* en Ada-CCM

6.1.4.3. Implementación del SchedulingService en Ada-CCM

El servicio *SchedulingService* se ha implementado de nuevo como una librería estática, *AdaCCMSchedulingService*, que ofrece los métodos definidos en la correspondiente interfaz *SchedulingService* definida en RT-CCM. Aspectos que se pueden destacar acerca de la implementación Ada de este servicio son:

- Se han utilizado las nuevas librerías de contenedores, concretamente la clase *Ordered_Maps*, para almacenar la información de configuración del servicio.
- La modificación de la prioridad de las tareas acorde al valor de *stimulusId* se realiza a través del paquete *Ada.Dynamic_Priorities*.
- La estrategia utilizada para la transmisión del valor de *stimulusId* a lo largo de la ejecución de una transacción se basa en la funcionalidad ofrecida por el paquete genérico *Ada.Task_Attributes*. Se ha definido una instancia de este paquete gracias a la cual se puede asignar un valor de tipo *StimulusId* como atributo de una tarea. La modificación del valor de *stimulusId* se realiza a través de las operaciones *setValue* y *getValue* ofrecidas por dicho paquete.

6.1.5. Conectores en la tecnología Ada-CCM

Cualquier implementación de la metodología RT-CCM, y en concreto Ada-CCM, soporta la introducción de diferentes tipos de conectores. Introducir soporte para un nuevo tipo de conector conlleva las siguientes tareas:

- Añadir el descriptor del nuevo mecanismo de conexión, utilizando un elemento *ConnectionMechanismDescription* de RT-D&C.
- Elaborar las plantillas del modelo de tiempo real del conector, que serán referenciadas desde el correspondiente descriptor.
- Desarrollar la herramienta que genera el modelo de tiempo real final de cada conector utilizado en la aplicación.
- Ampliar la herramienta de generación de código de conectores para que contemple el nuevo tipo de conexión.

El objetivo final de la tecnología Ada-CCM es el de soportar toda la comunicación entre componentes a través del propio lenguaje, sin necesidad de añadir mecanismos o middleware externos. Ada, en su anexo de sistemas distribuidos (DSA), ofrece un modelo propio de distribución. Para la versión Ada 95, GLADE [PT00] constituye la implementación más difundida del DSA, pero no ofrece garantías de tiempo real. RT-GLADE [LGG04][LGG06] representa una extensión de GLADE que optimiza sus características de tiempo real, proporcionando predictibilidad total y una mayor capacidad de configuración de la planificación. Para poder garantizar la predictibilidad, la plataforma de ejecución para RT-GLADE está basada en MaRTE OS y el protocolo de red de tiempo real RT-EP. La opción ideal para garantizar un soporte total por parte del lenguaje de la tecnología Ada-CCM sería la utilización de RT-GLADE; sin embargo, en el momento de realización de este trabajo no existía versión de RT-GLADE para Ada 2005 por lo que no se pudieron desarrollar conectores en base a dicho mecanismo. De hecho, la implementación del DSA con características de tiempo real sobre Ada 2005 se va a realizar sobre PolyORB [VHP04], un middleware que da soporte a la distribución con diferentes personalidades como CORBA, RT-CORBA o DSA en aplicaciones Ada. Actualmente se está trabajando para dotar a PolyORB de las mismas características con que se dotó a RT-GLADE para mejorar la predictibilidad temporal y la flexibilidad en la planificación de aplicaciones de tiempo real estricto [PJ09].

A falta de alguno de estos mecanismos, en la versión actual de Ada-CCM se han desarrollado conectores que usan directamente el protocolo RT-EP para la interacción entre componentes. Sus principales características son las siguientes:

- Por cada invocación recibida, el código del conector genera el correspondiente mensaje de invocación (en la parte *proxy*) o de respuesta (en la parte *servant*), haciendo uso de las librerías de manejo de *streams* que se incluyen en el propio protocolo.
- Para el envío y recepción de mensajes se hace uso directo de las correspondientes operaciones de envío y espera de respuesta definidas en la interfaz del protocolo: *RTEP.Protocol.Send_Info* y *RTEP.Protocol.Receive_Info*.
- La parte *servant* de un conector RT-EP hace uso de *AdaCCMThreadingService* para obtener los threads que necesita para ejecutar las invocaciones recibidas de forma concurrente.
- La sincronización de espera a respuesta se implementa haciendo uso directo del método de recepción bloqueante (*RTEP.Protocol.Receive_Info)* que se ofrece como parte de la interfaz del protocolo RT-EP.

- Las propiedades de configuración de cada conexión son:
  - El identificador del canal a través del que la parte *proxy* del conector realiza sus envíos, y en consecuencia, la parte *servant* se bloquea a la espera de mensajes.
  - La prioridad de base con la que la parte *servant* espera la llegada de mensajes (esa prioridad se transforma después en la que corresponde de acuerdo al *stimulusId* transmitido).

En la versión actual se han desarrollado las herramientas necesarias para generar conectores que implementan invocaciones remotas síncronas y asíncronas sin resultados utilizando RT-EP. Asimismo, para verificar que los conectores podrían ser generados sobre el DSA cuando la versión esté disponible sobre Ada 2005, se han desarrollado conectores que utilizan GLADE sobre plataforma Linux, sin predictibilidad temporal. Esta prueba ha servido además para validar la reutilización del código de negocio del componente en diferentes plataformas de ejecución sin necesidad de modificación.

6.1.6. Implementación del CommunicationSchedulingService en Ada-CCM

El servicio de planificación de las comunicaciones se ha implementado de nuevo como una librería estática, *AdaCCMCommunicationSchedulingService*, que ofrece los métodos definidos en la correspondiente interfaz *CommunicationSchedulingService* definida en RT-CCM.

La principal característica de este servicio es que por el momento admite un sólo tipo de parámetro de planificación, el que corresponde a los conectores anteriormente citados, basados en protocolo RT-EP. Su definición se muestra en la figura 6.7. Los valores que se almacenan asociados a cada *stimulusId* son la prioridad a la que se debe realizar el envío y el número de canal, que tiene un significado diferente según la parte de conector en la que se procese:

- En la parte *proxy*, es el identificador del canal en el que se espera la respuesta.
- En la parte *servant*, es el identificador a través del que se ha de enviar el mensaje con el resultado de la invocación.

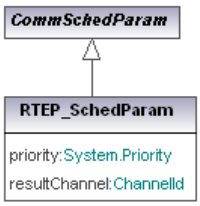


Figura 6.7: Parámetros de planificación correspondientes a conectores RT-EP

6.1.7. Entorno y herramientas de desarrollo de una aplicación Ada-CCM

El desarrollo de un nuevo componente o el diseño y ejecución de una aplicación son procesos que se realizan en el entorno de desarrollo y son asistidos por herramientas que garantizan la corrección "por construcción" de los artefactos que se generan. Para la tecnología Ada-CCM se ha diseñado un entorno de desarrollo para la plataforma Eclipse [ECLIP], el cual proporciona un conjunto de marcos de trabajo y servicios que simplifican la gestión de los recursos y el desarrollo de las herramientas. Existen dos elementos básicos que constituyen el entorno: el repositorio en el que se organizan los productos intermedios y finales que se generan, y las herramientas que realizan las transformaciones entre ellos.

El repositorio, que se muestra en la figura 6.8, se organiza dentro del espacio de trabajo (*workspace*) de Eclipse. Está compuesto por tres proyectos raíz:

- El proyecto *repository:* Almacena de forma organizada la información relativa a los elementos registrados. Dentro de él la información se organiza en cinco grandes secciones: *domains*, *components*, *applications*, *platforms* y *technology*. Cada una de ellas contiene la información correspondiente al tipo de elementos al que hace referencia su nombre. Internamente la información se organiza por dominios de aplicación, que definen diferentes espacios de nombres. Cualquier elemento en el repositorio se identifica utilizando los cuatro elementos <*section*>/<*domain*>/<*name*>.<*extension*>.
- El proyecto *adaccm* es una carpeta en la que se almacenan ficheros de código fuente o código compilado Ada. La estructura interna es la que corresponde a la estructura de paquetes Ada que resulta adecuada para la compilación, enlazado y construcción de aplicaciones, utilizando para ello las herramientas del plug-in Ada de GNAT para Eclipse, con el que se procesa el código Ada.
- El proyecto *tools* es de tipo Java. En él, se dispone de las herramientas disponibles para implementar los procesos de transformación y generación de código. En el estado actual de implementación estas herramientas son independientes y son invocadas directamente por el operador. En versiones futuras constituirán un plug-in integrado en Eclipse.

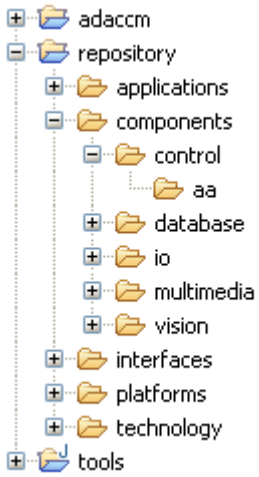


Figura 6.8: Repositorio del entorno de desarrollo para Ada-CCM

Las herramientas que se incluyen son:

- Las herramientas *ComponentImport*, *ComponentExport*, *InterfaceImport* e *InterfaceExport*, que permiten transferir elementos entre la información empaquetada con la que se distribuyen los componentes y el repositorio.
- La herramienta *BusinessInterfaceGenerator*, que genera el código de la interfaz de negocio del componente y opcionalmente un prototipo que sirve de plantilla para su desarrollo. Utiliza como entrada las descripciones RT-D&C de la interfaz externa del componente (*ComponentInterfaceDescription*) y de la implementación (*Component ImplementationDescription*).
- La herramienta *ComponentLibraryGenerator*, que genera la librería .a con que el código del componente es distribuido de forma opaca. Esta librería agrupa todos los ficheros que constituyen el código de negocio del componente y es en base a la cual se realiza posteriormente el enlace con el contenedor.

- La herramienta *ComponentContainerGenerator,* que genera el código Ada del contenedor que completa el código de negocio de un componente para hacerlo disponible a fin de ser integrado en una aplicación. Acepta como entrada el fichero RT-D&C que describe el paquete del componente (*PackageConfiguration*), y que incluye tanto la descripción de su interfaz externa como de las posibles implementaciones Ada-CCM.
- La herramienta *ConnectorGenerator* genera el código de los conectores que se emplean para conectar los componentes. Utiliza como entrada la declaración de la conexión que se incluye en el plan de despliegue, a través de la que se conoce el tipo de conector a utilizar, la interfaz de los puertos que conecta y sus características de configuración.
- La herramienta *MastModelComposer* genera el modelo de tiempo real de la aplicación compatible con el entorno MAST. Esta herramienta aplica las diferentes transformaciones de modelos que se han expuesto en capítulos anteriores para generar, a partir de los descriptores RT-D&C de la aplicación y de la información disponible en el repositorio, su correspondiente modelo MAST.
- La herramienta *AdaCCMShedulingConfiguration*, que genera el plan de despliegue final de la aplicación, plenamente configurado, en base al plan de despliegue inicial y al modelo MAST final que resulta después de aplicar las herramientas de asignación de prioridades sobre el modelo MAST obtenido del plan de despliegue inicial.
- La herramienta *AdaCCMApplicationGenerator* genera las particiones de código ejecutable (una por cada nodo de la plataforma) correspondiente a una aplicación descrita por su plan de despliegue. Utiliza como entrada el correspondiente plan de despliegue.
- La herramienta *AdaCCMLauncher* carga e inicia la ejecución en la plataforma de ejecución de una aplicación cuyas particiones se encuentran ya generadas.

Todas las herramientas están integradas en el entorno Eclipse, esto es, reciben como entrada la referencia a algún descriptor, el cual utilizan como guía para localizar en el repositorio toda la restante información que necesitan para realizar su operación. Los resultados que generan las herramientas se almacenan igualmente en la sección del repositorio a la que corresponden en función de su naturaleza.

6.2. Ejemplo de desarrollo de un componente Ada-CCM

Como ejemplo de aplicación de los diferentes aspectos que se han abordado en esta tesis referidos al desarrollo de un componente de tiempo real, se detalla el proceso de desarrollo completo del componente *ScadaEngine*, el cual se ha venido utilizando como ejemplo en capítulos anteriores.

Muchas de las aplicaciones desarrolladas siguiendo una metodología basada en componentes implementan arquitecturas cliente/servidor de 3 capas (*3-tier architecture)* [ECK95], en las que la estructura de la aplicación se divide en tres niveles:

- La capa cliente, o capa de presentación, representa el nivel más alto de la aplicación y está formada por componentes ligeros, que generalmente implementan la interfaz de interacción entre el usuario y la aplicación.
- La capa de aplicación o de lógica de negocio, en la que se implementa la funcionalidad característica de la aplicación dentro de un determinado dominio de aplicación. Los componentes que forman parte de esta capa suelen ser componentes pesados y complejos, pues son los que soportan la mayor parte de la lógica de la aplicación.

- La capa terminal, o capa de datos, engloba aquellos componentes terminales que dan acceso a recursos del sistema o a los datos que son utilizados por los componentes de la capa de negocio.

El componente *ScadaEngine* es un componente perteneciente a la capa intermedia, que implementa la funcionalidad básica de un sistema SCADA (*Supervisory Control and Data Acquisition*), esto es, supervisa periódicamente un conjunto de hasta 16 señales analógicas y registra, también de manera periódica, datos estadísticos sobre ellas con marcas temporales. Los periodos de muestreo y de registro son los mismos para todas las magnitudes supervisadas. El objetivo es diseñar el componente del modo más genérico o reutilizable posible, esto es, independiente del sistema SCADA en el que vaya a ser integrado. Para ello se independiza su comportamiento de los dispositivos a través de los que se capturan los datos, y también del método de almacenamiento de los valores estadísticos calculados.

La figura 6.9 muestra el proceso de desarrollo completo de este componente, destacando las fases del proceso, los artefactos (descriptores RT-D&C, módulos de código, etc.) concretos que se elaboran en él, los agentes que son responsables de cada uno de ellos y las herramientas que éstos utilizan en su elaboración. Se distinguen en la figura los artefactos que son generados a mano, de aquellos que se generan a través de herramientas. Las diferentes fases del proceso

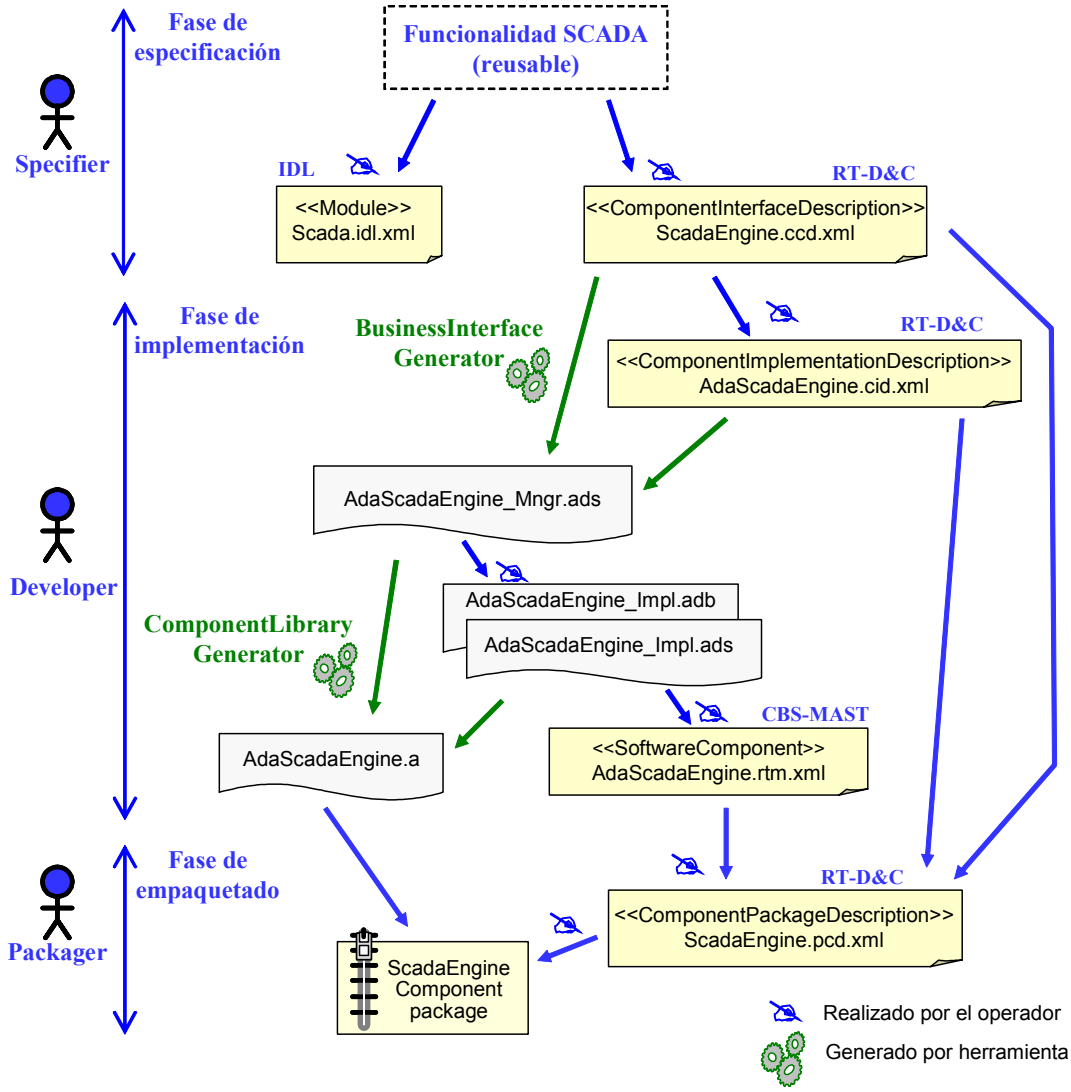


Figura 6.9: Proceso de desarrollo del componente *ScadaEngine* en Ada-CCM

(*Especificación*, *Implementación* y *Empaquetado*) se explican en detalle a continuación, mientras que los ficheros .xml completos correspondientes a los descriptores RT-D&C se pueden consultar en el anexo C.

6.2.1. Fase de especificación

El proceso de desarrollo del componente comienza cuando el agente *specifier* define la interfaz *ScadaEngine*, que describe la funcionalidad SCADA que se desea implementar. Para ello, en primer lugar, el *specifier* define la interfaz IDL que agrupa los servicios necesarios para implementar esta funcionalidad. La interfaz, denominada *ScadaControl*, define las operaciones básicas para establecer las magnitudes a supervisar y para acceder a los valores almacenados. Se incluyen operaciones que permiten ordenar y caracterizar una nueva tarea de supervisión (*supervise*), cancelar una de las tareas de supervisión establecidas (*cancel*), retornar los últimos datos adquiridos de una determinada magnitud (*getBufferedData*), etc. La interfaz, junto con los tipos de datos que se requieren para su definición, se formulan en el fichero *Scada.idl.xml*. Como se observa en la figura 6.10, el componente *ScadaEngine* ofrecerá esta funcionalidad a través de su única faceta, *controlPort*, que implementa la interfaz *ScadaControl*[1].

Con el objetivo de hacer el componente reutilizable e independiente de la aplicación, las actividades de adquisición de datos y de registro de valores estadísticos se realizarán a través de componentes independientes, a los que como se muestra en la figura 6.10, se accede a través de los dos receptáculos que declara el componente:

- *adqPort*: Receptáculo a través del cual el componente obtiene los valores de las magnitudes supervisadas. Los valores se obtienen a través de los servicios definidos en la interfaz *AnalogIO*, perteneciente al dominio *io*, que define operaciones para la lectura y establecimiento de señales analógicas de entrada y salida respectivamente. Como el componente va a soportar la supervisión simultánea de varias señales (hasta 16) este receptáculo se define como múltiple, pues cada señal puede ser adquirida a través de un dispositivo diferente.
- *logPort*: Receptáculo a través del cual el componente almacena de forma periódica los valores estadísticos calculados. Implementa la interfaz *Logging*, perteneciente al dominio *db*, que define operaciones para registrar datos de tipo *string* con una marca del instante de tiempo en que se producen. En este caso se trata de un receptáculo simple, pues únicamente se requiere un dispositivo de almacenamiento.

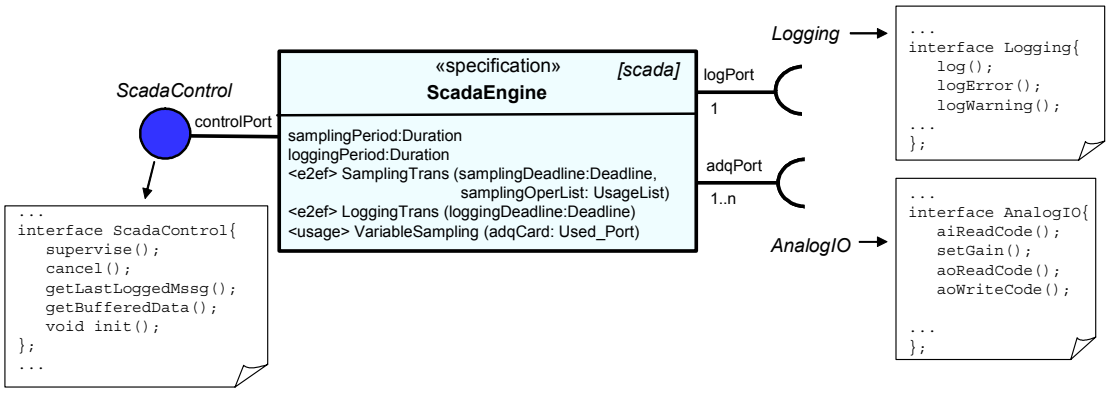


Figura 6.10: Interfaz externa del componente *ScadaEngine*

1. Por simplicidad, no se muestran en la figura las signaturas completas de los procedimientos de cada interfaz.

A diferencia de la interfaz *ScadaControl*, que ha sido definida por el *specifier* en esta fase, las interfaces *AnalogIO* y *Logging* se encontraban ya disponibles en el repositorio.

Por último, quedan por definir los parámetros de configuración de negocio del componente. En este caso se definen dos parámetros:

- *samplingPeriod*: Establece el periodo de supervisión de las magnitudes que supervisa el componente.
- *loggingPeriod*: Establece el periodo con el que se almacenan los resultados estadísticos de la señales previamente muestreadas.

Hasta este momento se han abordado los aspectos puramente funcionales de la interfaz del componente. El aspecto más importante desde el punto de vista de tiempo real consiste en identificar su reactividad, esto es, identificar la naturaleza de los eventos a los que el componente va a ser capaz de responder. Cada una de las respuestas posibles se declara como una transacción, indicando su nombre, su naturaleza y sus posibles parámetros. En este caso se identifican dos tipos de respuestas a eventos, derivadas de la funcionalidad del componente:

- *SamplingTrans*: El componente lleva a cabo una actividad periódica, de periodo *samplingPeriod*, a través de la que supervisa las magnitudes. Se considera que en cada ciclo se supervisan todas las magnitudes que se hayan configurado. La naturaleza de esta respuesta es temporizada, esto es, se ejecuta en respuesta a un evento periódico generado por el reloj del sistema.
- *LoggingTrans*: Representa la actividad periódica, de periodo *loggingPeriod*, a través de la cual el componente almacena de forma persistente las estadísticas calculadas sobre los valores de las magnitudes adquiridos en cada ciclo de supervisión. También se trata de una transacción temporizada, que se ejecuta en respuesta a un evento periódico generado por el reloj del sistema.

Las transacciones se representan en la figura con el estereotipo <e2ef>, y como se puede observar, definen como parámetros los requisitos temporales que se le pueden imponer. En el caso de *SamplingTrans* el requisito se impone en la finalización de la transacción, mientras que en el caso de *LoggingTrans* representa el plazo asociado al envío de los datos al logger, sin tener en cuenta la duración de la actividad de almacenamiento. La transacción *SamplingTrans* define otro parámetro, *samplingOperList*. El flujo de control de esta transacción es variable, pues las actividades ejecutadas dentro de ella dependerán del número de magnitudes a supervisar: por cada magnitud que se supervise será necesario realizar una actividad de adquisición del valor. Esta actividad se modela como una forma de uso, *VariableSampling*, que se declara también a nivel de la interfaz del componente. Está también parametrizada, pues dependerá del componente terminal concreto (de entre todos los conectados al puerto *adqPort*) sobre el que se realice la adquisición. Cuando se declare una instancia de la transacción *SamplingTrans* en un modelo de carga de una aplicación, será necesario asignarle una instancia de la forma de uso *VariableSampling* debidamente configurada, por cada magnitud supervisada.

Por último hay que incluir las propiedades de componibilidad de tiempo real del componente:

- Operaciones con modelo de tiempo real de la faceta *controlPort*:
  - *getLastLoggedMssg* => Referencia el modelo de la operación a través de la que se obtiene el último mensaje registrado relativo a una de las magnitudes supervisadas.
  - *getBufferedData* => Referencia el modelo de la operación a través de la que se requieren los valores adquiridos hasta el momento (desde el último almacenamiento) de una de las magnitudes supervisadas.

- Operaciones con modelo de tiempo real requeridas a través del receptáculo *logPort*:
  - *log* => Referencia el modelo de tiempo real de la operación del componente accedido a través de *logPort* utilizada para registrar los datos.
- Operaciones con modelo de tiempo real requeridas a través del receptáculo *adqPort*:
  - *aiReadCode* => Requiere que el componente que se conecte al receptáculo *adqPort* disponga de modelo para la operación con la que se lee el valor de una magnitud.

Toda esta información la formula el *specifier* en un fichero de nombre *ScadaEngine.ccd.xml*, cuyo elemento raíz es del tipo *ComponentInterfaceDescription* de RT-D&C. Es muy importante que el fichero incluya a través de etiquetas la descripción de la naturaleza del componente, así como de sus respuestas a eventos, pues será esa la información que gestionará el *assembler* posteriormente como base para decidir la utilización del componente en una aplicación. El fichero *ScadaEngine.ccd.xml* se puede consultar en la página 271 (anexo C).

6.2.2. Fase de implementación

Una vez definida la interfaz del componente, el *developer* es el responsable de elaborar una implementación de dicha interfaz. En este caso se trata de una implementación compatible con la tecnología Ada-CCM denominada *AdaScadaEngine*. El *developer* diseña las características de la implementación, principalmente el número de threads y de mecanismos de sincronización que va a requerir, y lo plasma en el descriptor de implementación, en este caso el fichero *AdaScadaEngine.cid.xml*. Una vez definido este descriptor, se usa como entrada de la herramienta *BusinessInterfaceGenerator*, que genera el código de la interfaz de negocio del componente. El *developer* usa esta interfaz como base para elaborar el código de la implementación. Una vez desarrollado, utiliza la herramienta *ComponentLibraryGenerator* para generar una librería .a con todos los ficheros fuente necesarios para ejecutar el componente, en este caso sólo los ficheros *AdaScadaEngine_Impl.ads/b y AdaScadaEngine_Mngr.ads*. Esa librería será la que se distribuya en el paquete del componente, a la que se puede acceder a través del propio descriptor de implementación.

Como se observa en la figura 6.11 la implementación *AdaScadaEngine* requiere del entorno dos threads con activación periódica: *samplingTh*, para realizar la lectura de las señales supervisadas, y *loggingTh,* para el registro de los resultados estadísticos. Se trata de una estrategia general a adoptar en el diseño de componentes RT-CCM: por cada transacción periódica identificada en la interfaz del componente, se va a requerir un thread periódico en la correspondiente implementación, y en consecuencia se va a declarar un puerto de activación de tipo *PeriodicActivation*.

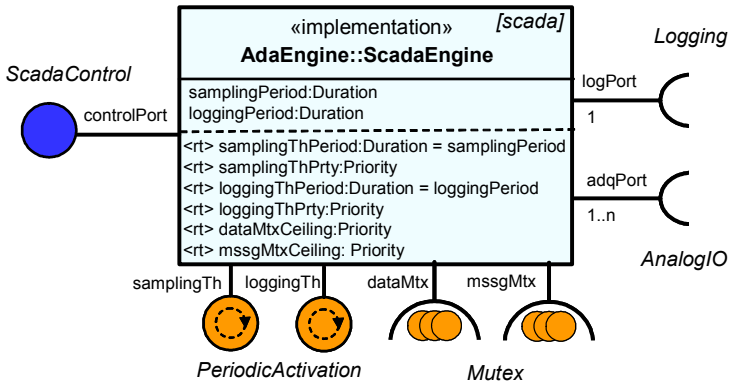


Figura 6.11: *AdaScadaEngine*: Implementación Ada del componente *ScadaEngine*

En cada ciclo de supervisión, el thread *samplingTh* lee de forma secuencial todas las magnitudes, que son almacenadas en una variable temporal. Los valores se adquieren a través del método *aiReadCode* invocado en el correspondiente receptáculo *adqPort*. Una vez adquiridos todos los valores, se actualizan las estadísticas. El thread *loggingTh* accede periódicamente a los resultados del cálculo de las estadísticas, genera el correspondiente mensaje y lo almacena, invocando el método *log* a través del receptáculo *logPort*. Para garantizar el acceso seguro de ambos threads a la estructura de datos interna donde se almacenan las estadísticas actualizadas, la implementación utiliza un mutex, requerido a través del puerto de sincronización *dataMtx*.

Por otro lado, cuando sobre el componente se invoca el método *getLastLoggedMssg*, éste debe devolver el último mensaje que se envió a través de *logPort*. Para evitar la generación del mensaje en cada invocación del método, cada mensaje enviado se almacena en una variable interna, cuyo valor es el que se devuelve cuando se invoca el método anterior. El acceso a dicha variable debe estar también protegido, por lo que se declara otro puerto de sincronización de tipo *Mutex*, denominado *mssgMtx*.

A consecuencia de la declaración de estos puertos de sincronización y activación, aparecen un conjunto de parámetros, específicos de esta implementación y relacionados con la planificación de las actividades ejecutadas en el código del componente. Todos ellos representan a su vez propiedades del modelo de tiempo real del componente (las propiedades de tiempo real se muestran en la figura separadas por una línea de puntos y con el estereotipo <rt>):

- *samplingThPeriod*: Periodo con el que se invoca el thread *samplingTh*, que corresponde al periodo de los eventos que lanzan la transacción *SamplingTrans*. Su valor es igual al valor del parámetro de configuración *samplingPeriod*, y así se asigna en la declaración del puerto de activación.
- *samplingThPrty*: La prioridad a la que se inicia la ejecución del thread *samplingTh*. Las herramientas de diseño calcularán su valor, que será asociado al valor de *stimulusId* de partida que se asigne al puerto *samplingTh* en el plan de despliegue.
- *loggingThPeriod*: Periodo con el que se invoca el thread *loggingTh*, esto es, el periodo de los eventos que lanzan la transacción *LoggingTrans*. Su valor se mapea al valor del parámetro de configuración *loggingPeriod*.
- *loggingThPrty:* La prioridad a la que se inicia la ejecución del thread *loggingTh*. Las herramientas de diseño calcularán su valor, que será asociado al valor de *stimulusId* de partida que se asigne al puerto *loggingTh* en el plan de despliegue.
- *dataMtxCeiling*: Techo de prioridad del mutex *dataMtx*. Su valor final se evaluará en el proceso de diseño de tiempo real y se la asignará directamente al puerto de sincronización.
- *mssgMtxCeiling*: Techo de prioridad del mutex *mssgMtx*. Su valor final se evaluará en el proceso de diseño de tiempo real y se la asignará directamente al puerto de sincronización.

Toda esta información se plasma en el descriptor de implementación, *AdaScadaEngine.cid.xml*, que se puede consultar en la página 272 (anexo C). El descriptor incluye además el localizador del modelo de tiempo real de la implementación, *AdaScadaEngine.rtm.xml,* cuya formulación se detalla en el siguiente apartado.

6.2.2.1. Elaboración del modelo de tiempo real

El modelo de tiempo real del componente *AdaScadaEngine* se formula de acuerdo a la metodología CBS-MAST. Como muestra la figura 6.12, el elemento raíz del modelo es un elemento de tipo *Software_Component*, que incluye los siguientes elementos:

- Los parámetros que se han declarado en la implementación del componente: *loggingThPeriod*, *loggingThPrty*, *dataMtxCeiling*, etc.
- Modelos de tiempo real de cada uno de los servicios ofrecidos a través de la faceta *controlPort*, al menos de aquellos que se identifican como ofertados en la interfaz *ScadaEngine*. En la figura 6.12 se muestran los modelos de los servicios *getBufferedData* y *getLastLoggedMssg*, ambos declarados dentro del elemento *controlPort*, de tipo *Provided_Port* (así se distingue la faceta a través del que se ofrece cada servicio).
- Modelos de los recursos de concurrencia o sincronización que se agregan al modelo por cada instancia del componente. Como regla general, todos los mutex que la implementación requiere del entorno deben incluirse en el modelo del componente. En este caso, como se muestra en la figura, se añade los elementos *dataMtx* y *mssgMtx*, de tipo *Shared_Resource*, cuyo valor de techo se iguala a los parámetros *dataMtxCeiling* y

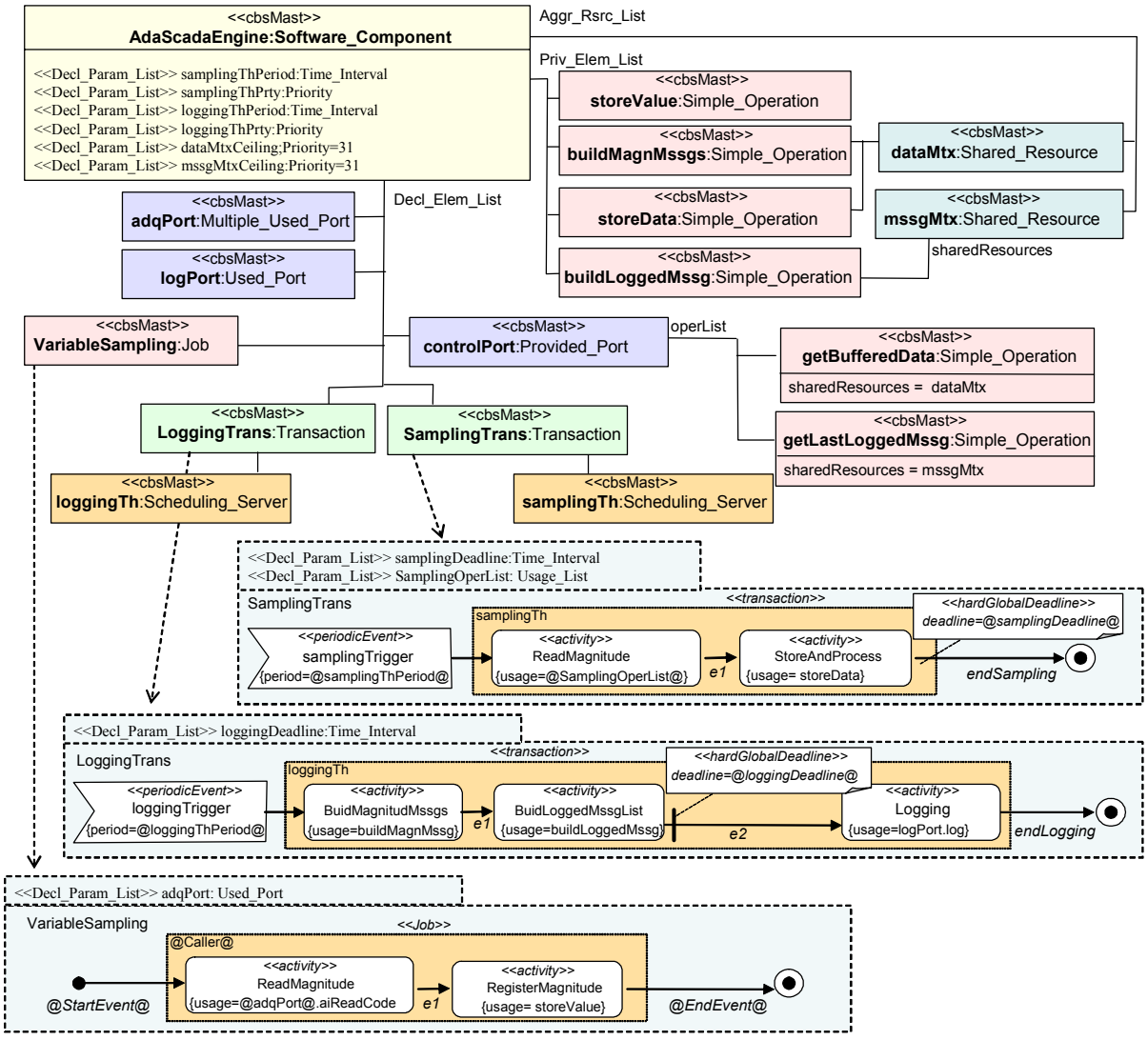


Figura 6.12: Modelo de tiempo real del componente *AdaScadaEngine*

mssgMtxCeiling. Estos recursos son utilizados en algunas de las operaciones ejecutadas por el componente, como *getBufferedData* y *getLastLoggedMssg,* respectivamente.

- Modelos de las transacciones y de las formas de uso que se identifican en la interfaz del componente, incluyendo cada uno de los parámetros allí declarados. En este caso se incluyen las transacciones *LoggingTrans* y *SamplingTrans*, así como la forma de uso *VariableSampling*. Para describir estos elementos se puede hacer uso de otros elementos también incluidos en el modelo, como por ejemplo la operación interna *buildLoggedMssg,* que modela la generación del mensaje que se va a almacenar. Como regla general, cada transacción periódica (asociada a un puerto de activación periódico) agrega su propio *Scheduling_Server*, *loggingTh* y *samplingTh* en este caso, que representa el thread que se obtiene a través del puerto de activación correspondiente, y que es el encargado de comenzar la ejecución de la transacción. Su prioridad se iguala al parámetro correspondiente, en este caso *loggingThPrty* y *samplingThPrty* respectivamente. El valor que se obtenga para dicha prioridad se mapeará al valor de *stimulusId* asignado al puerto, con lo que nos aseguramos que la transacción comienza siempre con dicho valor.

Un aspecto importante a destacar es que aquellos valores que influyen en la planificación de la aplicación, como *loggingThPrty* o *dataMtxCeiling*, son asignados a los correspondientes elementos del modelo, pero en todos ellos se asigna valor “NO” al atributo *preassigned*. Con ello se indica que dichos valores pueden ser modificados y calculados por las herramientas de asignación de prioridades de MAST. Los valores calculados serán reasignados a los correspondientes elementos en el plan de despliegue (en el caso de los techos de prioridad) o utilizados para configurar el servicio de planificación (en el caso de las prioridades). El fichero que representa el descriptor de modelo completo del componente *AdaScadaEngine*, *AdaScadaEngine.rtm.xml*, se puede consultar en la página 273 (anexo C).

6.2.3. Fase de empaquetamiento

En la última fase el *packager* recoge toda la información acerca del componente y lo empaqueta como un archivo .zip. Previamente, elabora un fichero denominado *ScadaEngine.pcd* (de tipo *PackageConfiguration*), donde recoge todos los metadatos necesarios para hacer uso de todas las posibles implementaciones del componente que se incluyan en el paquete. En este fichero se recoge tanto la información acerca de la interfaz del componente (fichero .ccd.xml) como de todas las implementaciones (ficheros .cid.xml). En el paquete .zip con el que se distribuye el componente se incluyen todos los descriptores RT-D&C así como el código del componente (como un librería .a) y el modelo de tiempo real (fichero .rtm.xml).

6.3. Ejemplo de desarrollo de una aplicación en Ada-CCM

Una vez visto el proceso de desarrollo de un componente, se explica el proceso de desarrollo de una aplicación en la que se hace uso de él. La aplicación de ejemplo se denomina *ScadaDemo*, y a lo largo de esta sección se explican las diferentes etapas que constituyen su desarrollo completo, desde la especificación hasta la ejecución.

6.3.1. Especificación de la aplicación ScadaDemo

La aplicación ScadaDemo, que se representa en la figura 6.13, tiene por objetivo supervisar un conjunto de magnitudes físicas analógicas, adquiridas a través de tarjetas de adquisición de datos. Supervisar una magnitud supone leer su valor a una determinada frecuencia configurable

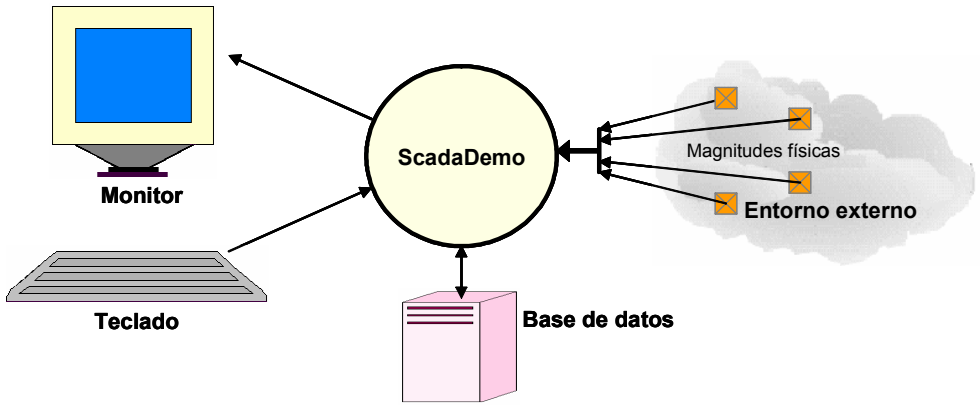


Figura 6.13: Aplicación de ejemplo *ScadaDemo*

y mantener una serie de registros estadísticos sobre los valores leídos. A una frecuencia más baja y también configurable se almacenan de forma persistente las estimaciones estadísticas evaluadas en ese periodo, junto con el instante de tiempo al que corresponden (la del momento de ser almacenadas). Asimismo, el último conjunto de datos almacenados para una determinada magnitud se muestra por pantalla de forma periódica, con periodo también configurable. La aplicación se controla a través del teclado, ofreciendo las siguientes posibilidades al operador:

- Elegir la magnitud de la que se muestran los datos almacenados.
- Mostrar por pantalla los valores adquiridos para una magnitud determinada desde la última vez que se realizó un envío a la base de datos.

Aplicando una metodología reactiva, la funcionalidad de esta aplicación se describe como el conjunto de respuestas a eventos que se ejecutan en ella de forma concurrente para implementar su funcionalidad. En este caso, se identifican cinco tipos de respuesta:

- De forma periódica (con periodo configurable) se realiza un ciclo completo de supervisión de todos las magnitudes programadas. Esta respuesta corresponde a un tipo de evento temporizado, generado por el reloj del sistema, y es de tiempo real, pues cada ciclo de supervisión debe ser ejecutado antes de que se produzca la siguiente activación.
- A una frecuencia más baja y también configurable, se almacenan en el logger todas las estimaciones estadísticas evaluadas para las magnitudes que se supervisan. De nuevo se trata de un evento temporizado generado por el reloj del sistema. El mensaje con los datos debe ser enviado al logger antes de la siguiente activación, sin embargo, no se imponen restricciones temporales a la actividad de almacenamiento en el propio logger.
- También de forma periódica, y de nuevo en respuesta a un evento procedente del reloj interno del sistema, se muestra en el monitor, para la magnitud elegida, el último conjunto de valores enviados al logger. Los datos deben ser mostrados en pantalla antes de la siguiente activación.
- En cualquier momento y a través del teclado, el operador puede modificar la magnitud cuyos datos se muestran en pantalla. En este caso el evento que lanza la transacción es de tipo externo, pues corresponde a la pulsación de una tecla por parte del operador. La modificación de la magnitud debe realizarse antes de que el operador ordene otra modificación.
- En respuesta a la pulsación de otra tecla, el sistema muestra por pantalla los valores recogidos para una magnitud elegida desde la última vez que sus estadísticas fueron almacenadas. Los valores deben ser mostrados antes de que el operador ordene otra actividad de muestra.

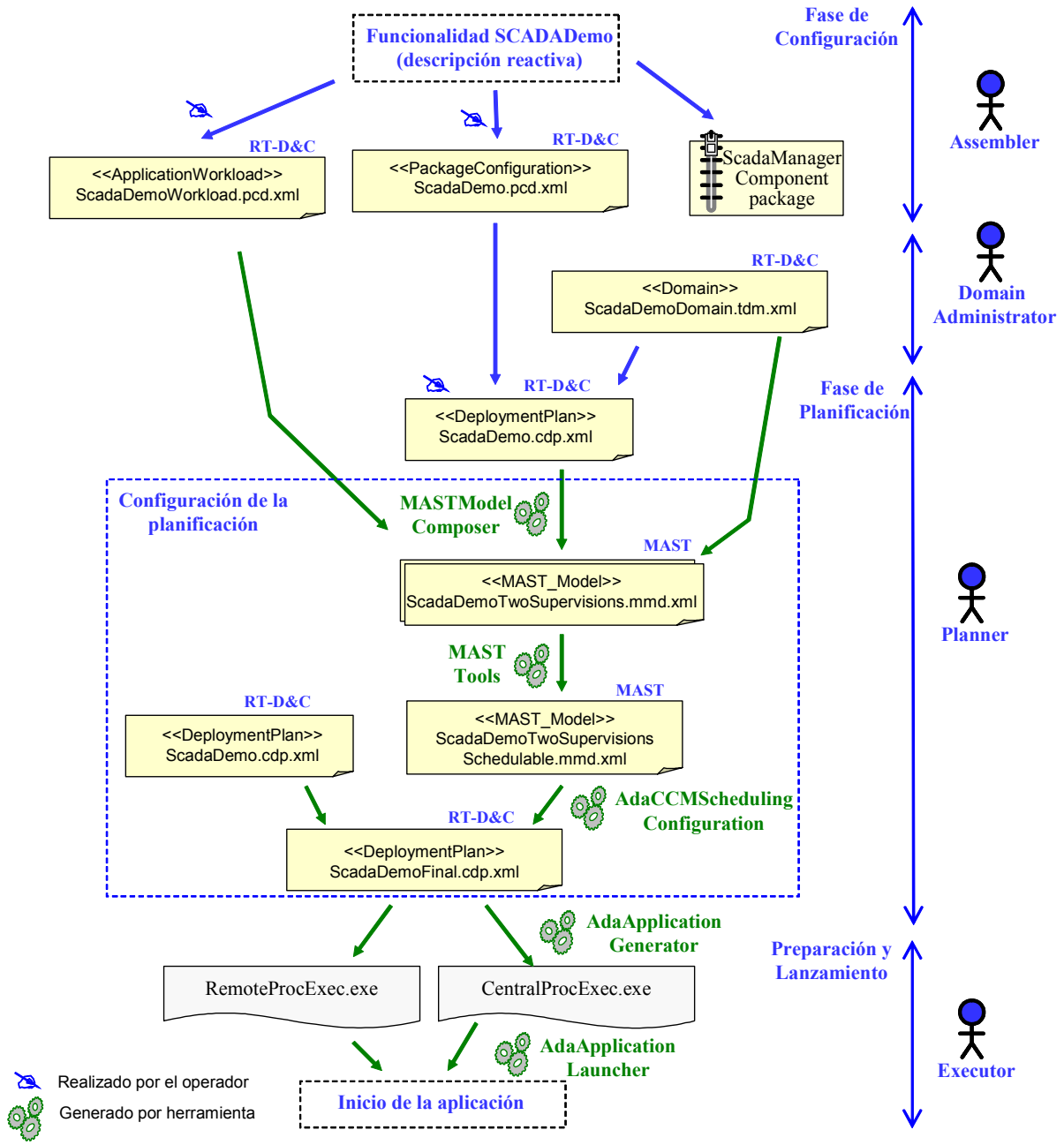


Figura 6.14: Proceso de desarrollo de la aplicación *ScadaDemo*

Esta especificación representa el punto de partida del proceso de desarrollo de la aplicación, como se muestra en la figura 6.14. Esta figura plasma el proceso completo de desarrollo, identificando las fases, artefactos, agentes y herramientas involucrados en él. A lo largo de la sección se explica cada fase en detalle, y de nuevo, algunos de los descriptores RT-D&C utilizados se pueden consultar en detalle en el anexo C.

6.3.2. Fase de configuración

La fase de configuración es responsabilidad del *assembler* y consiste en definir la aplicación como un ensamblado de componentes que satisfagan la funcionalidad requerida.

En una aplicación de tiempo real el aspecto clave consiste en satisfacer la reactividad de la aplicación, por lo que el proceso de búsqueda de componentes se centra en buscar aquellos componentes que responden a los eventos identificados en la especificación de la aplicación.

Definición de la arquitectura de la aplicación

Aplicando una arquitectura de tres capas a la aplicación *ScadaDemo*, se elige como componente central de la aplicación (aquél que soporta la mayor parte de su lógica de negocio) el componente *ScadaEngine*. A través del análisis del fichero *ScadaEngine.ccd.xml*, el *assembler* descubre que éste implementa algunas de las respuestas a eventos requeridas, en concreto las dos primeras. El siguiente paso consiste en encontrar componentes compatibles con sus receptáculos, tanto desde el punto de vista funcional (compatibilidad de interfaces) como de tiempo real (componibilidad de tiempo real). En este caso se eligen los componentes *Logger* e *IOCard*, cuyas especificaciones se pueden observar en la figura 6.15, en la que se muestra la arquitectura de componentes genérica de la aplicación:

- El componente *IOCard* pertenece al dominio *io*, y se trata de un componente destinado a la gestión de una tarjeta de adquisición de señales analógicas y digitales, definido con el objetivo de proporcionar el acceso a cualquier tarjeta con una interfaz estandarizada. Ofrece capacidad para leer y establecer valores en líneas tanto analógicas, a través de la faceta *analogPort*, como digitales, a través de la faceta *digitalPort*. En este caso, la funcionalidad que nos interesa es la ofrecida a través de *analogPort*.
- El componente *Logger*, perteneciente al dominio *db*, es un tipo de componente que se utiliza para almacenar de forma persistente información de texto, a la que se le asocia una marca del instante en que se registra. Esta es la funcionalidad ofrecida a través de la faceta *regPort*. Además, de forma opcional, la información se puede registrar bajo dos categorías: *Error* y *Warning*, generándose en cada caso un evento de tipo *DBChangeEvent* a través de las fuentes de eventos *errorEv* y *warningEv*, respectivamente. Los componentes interesados en estos eventos pueden registrarse en dichas fuentes para ser notificados cuando información de ese tipo es registrada en el *Logger*. En este caso no se hace uso de este mecanismo de notificación, lo cual es posible pues la multiplicidad de las fuentes de eventos es de 0..n.

Falta por definir el componente de la capa cliente, que en general es el que presenta más dependencias de la aplicación concreta que se está desarrollando. Dicho de otro modo, mientras que el resto de componentes son escogidos de entre aquellos componentes reutilizables disponibles en el catálogo, el componente cliente de la aplicación suele ser desarrollado

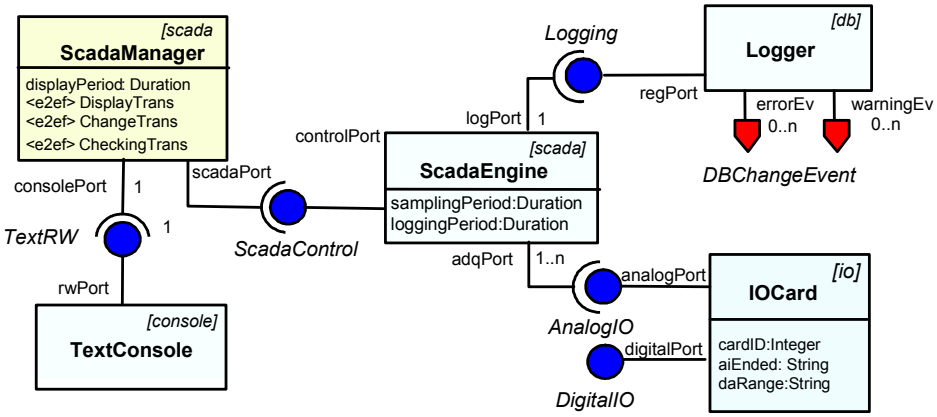


Figura 6.15: Arquitectura software de una aplicación SCADA

expresamente para la aplicación concreta. En este caso se desarrolla el componente *ScadaManager*, cuya especificación se muestra también en la figura. Desde él se manejan las tareas de supervisión, en concreto a través del receptáculo *scadaPort*. Parte de la funcionalidad de este componente consiste en mostrar por pantalla datos acerca de las magnitudes supervisadas, para lo cual hace uso de un componente previamente definido, *TextConsole*, que a través de su faceta *rwPort* ofrece la posibilidad de leer y escribir texto a través del teclado y la pantalla, respectivamente. El componente *ScadaManager* implementa el resto de las respuestas a eventos requeridas en la aplicación, a través de las siguientes transacciones:

- *DisplayTrans*: Representa la actividad periódica (de periodo *displayPeriod*) que muestra por pantalla los valores estadísticos calculados para una de las magnitudes supervisadas (la elegida por el operador). Define como parámetro el plazo asociado a su finalización.
- *ChangeTrans*: Representa la actividad que se ejecuta cuando el operador ordena modificar la magnitud de la cual se muestran datos en pantalla. Es una transacción de tipo esporádico, y define como parámetros tanto el periodo mínimo de activación, como el plazo de ejecución asociado a la finalización.
- *CheckingTrans*: Representa la actividad que se ejecuta cuando el operador ordena mostrar por pantalla los valores adquiridos para una determinada magnitud desde que se realizó el último almacenamiento de información. Se trata también de una transacción esporádica cuyo periodo mínimo y plazo de ejecución son definidos como parámetros.

La figura 6.15 representa la arquitectura genérica de una aplicación de tipo Scada, donde por ejemplo, la conexión entre el componente *ScadaEngine* y el componente *IOCard* es de multiplicidad 1..n, lo que representa que una instancia *ScadaEngine* puede estar conectada simultáneamente a una o más instancias de tipo *IOCard*. Sin embargo, el *assembler* debe definir una aplicación en la que se identifique el conjunto de instancias concretas que intervienen. La estructura de instancias para la aplicación *ScadaDemo* que utilizamos como ejemplo es la que se muestra en la figura 6.16. En ella se asocian dos instancias de componente *IOCard*, *sensorA* y *sensorB*, a la instancia de componente ScadaEngine, *engine*, lo que supone que las magnitudes se adquieren a través de dos tarjetas de adquisición diferentes. Completan la aplicación las instancias *manager*, *register* y *terminal*. El *assembler* refleja esta arquitectura en un nuevo descriptor de tipo *PackageConfiguration*, (.pcd.xml), que tiene la misma estructura que el de un componente cualquiera, salvo que en este caso la implementación se define como un ensamblado de componentes. Este fichero, denominado *ScadaDemo.pcd.xml*, se muestra en la página 275 (anexo C).

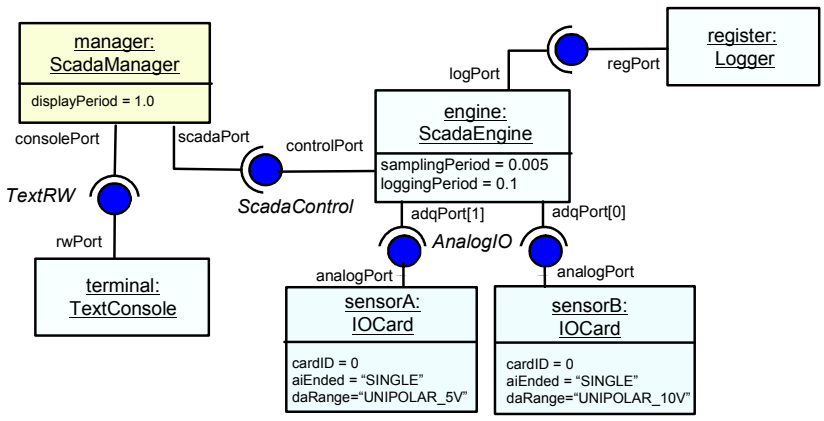


Figura 6.16: Estructura de instancias de la aplicación *ScadaDemo*

En dicho fichero, el *assembler* asigna también la configuración de negocio de la aplicación, que se muestra también en la figura 6.16. Se desea realizar la tarea de supervisión con un periodo de 5 ms (*engine.samplingPeriod* = 0.005), la tarea de almacenamiento de información con un periodo de 100 ms (*engine.loggingPeriod* = 0.1) y la de actualización de valores en pantalla con un periodo de 1s (*manager.displayPeriod* = 1.0). Además, las instancias de tipo *IOCard* reciben valores para las propiedades *cardID*, *aiEnded* y *daRange* definidas en sus interfaces.

Definición de las situaciones de tiempo real de la aplicación

Junto con la definición de la estructura de la aplicación, el *assembler* debe identificar las situaciones de tiempo real en las que la aplicación puede ejecutar, y definir para cada una de ellas la carga de trabajo que la describe y que sirve de base para el análisis de planificabilidad.

La carga de trabajo que soporta la aplicación *ScadaDemo* varía en función del número de magnitudes supervisadas. En este ejemplo se identifican dos posibles situaciones:

- *TwoSupervisions*: Representa la situación de tiempo real que corresponde a la supervisión de dos magnitudes: una a través de la instancia *sensorA* y otra a través de la instancia *sensorB*.
- *ThreeSupervisions*: Este caso añade al anterior la supervisión de una nueva magnitud a la que se accede a través de la instancia *sensorA*.

El fichero en que se declaran las posibles situaciones de tiempo real de la aplicación, *ScadaDemoWorkload.wld.xml*, se muestra en la página 276 (anexo C). En él, el *assembler* declara las correspondientes instancias de transacciones que se ejecutan en cada situación, debidamente configuradas. En ambas situaciones el número de transacciones, cuyas características se recogen en la tabla 6.2, es el mismo:

- *samplingTransInst*: Instancia de la transacción *SamplingTrans* iniciada en la instancia *engine*. Se asigna un valor de 0.005 (igual que el valor de *engine.samplingPeriod*) a su parámetro *samplingDeadline*.
- *loggingTransInst*: Instancia de la transacción *LoggingTrans* iniciada en la instancia *engine*. Se asigna un valor de 0.1 (igual que el valor de *engine.loggingPeriod*) a su parámetro *loggingDeadline*.
- *displayTransInst*: Instancia de la transacción *DisplayTrans* iniciada en la instancia *manager*. Se asigna un valor de 1.0 (igual que el valor de *manager.displayPeriod*) a su parámetro *displayDeadline*.
- *changeTransInst*: Instancia de la transacción *ChangeTrans* iniciada en la instancia *manager*. Se asigna un valor de 0.5 (tiempo mínimo estimado entre 2 pulsaciones consecutivas de una tecla) a sus parámetros *changeDeadline* y *changeTrigger*.

Tabla 6.2: Carga de trabajo y requisitos temporales de la aplicación *ScadaDemo*

Transacción	Tipo de disparo	Periodo	Deadline
samplingTransInst	Periódico	0.005	0.005
loggingTransInst	Periódico	0.1	0.1
displayTransInst	Periódico	1.0	1.0
changeTransInst	Esporádico	0.5	0.5
checkingTransInst	Esporádico	0.5	0.5

- *checkingTransInst*: Instancia de la transacción *CheckingTrans* iniciada en la instancia *manager*. Se asigna un valor de 0.5 a sus parámetros *checkingDeadline* y *checkingTrigger*.

La diferencia entre ambas situaciones de tiempo real se refleja en la definición de la instancia de transacción *samplingTransInst*. En su interfaz externa, el componente *ScadaEngine* declara como parámetro de dicha transacción la lista de actividades de lectura a realizar, que serán asignadas como una lista de instancias de la forma de uso *VariableSampling*. Por tanto, mientras que en la primera situación la lista está formada por 2 instancias de esta forma de uso, en el segundo caso se añadirá una instancia más. *VariableSampling* es a su vez parametrizada, pues depende del componente de tipo *IOCard* con el que obtiene los valores de las magnitudes: en cada caso habrá que asignar al correspondiente parámetro la referencia al puerto *analogPort* de las instancias *sensorA* o *sensorB*, según corresponda. En la figura 6.17 se muestra a través de un diagrama de secuencia el flujo de ejecución de la transacción *samplingTransInst*. La transacción se activa por la invocación del método *update* en el puerto de activación *samplingTh* de la instancia *engine*. El diagrama muestra la secuencia de actividades para ambas situaciones de tiempo real, tanto en el caso de dos magnitudes supervisadas, donde las invocaciones incluidas en el fragmento *opt* no son ejecutadas, como de tres supervisiones, en cuyo caso sí se ejecutan.

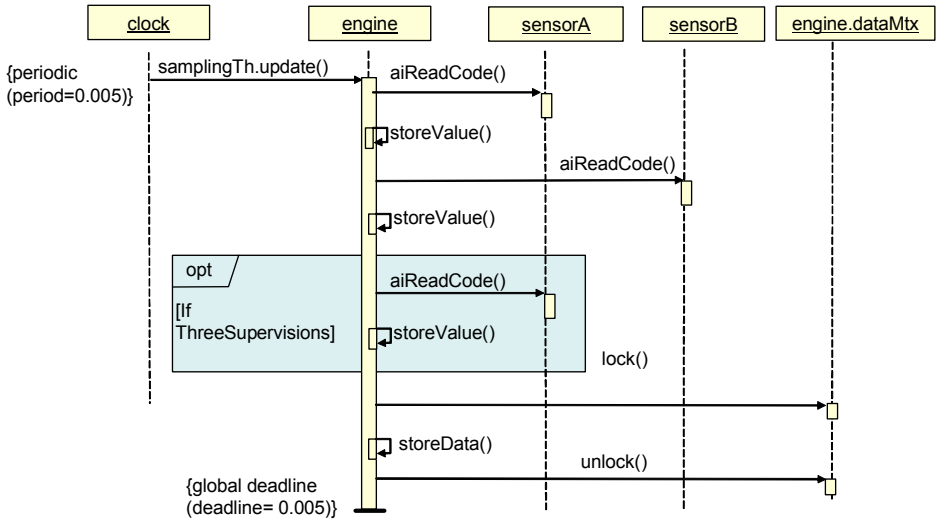


Figura 6.17: Actividad de la transacción *samplingTransInst*

El *assembler* es siempre responsable de generar los ficheros que describen la aplicación como ensamblado de componentes y su carga de trabajo. Si, como ocurre en este caso con el componente *ScadaManager*, es necesario desarrollar algún componente específico para la aplicación, el *assembler* es también responsable de elaborar su correspondiente paquete, como se mostraba en la figura 6.14, incluyendo en él sus descriptores RT-D&C, su código y su modelo de tiempo real. Para su elaboración, el *assembler* puede contar con la ayuda de expertos en los diferentes aspectos (elaboración del código, del modelo de tiempo real, etc.).

6.3.3. Fase de planificación

La fase de planificación se lleva a cabo una vez definida la estructura de instancias de componentes que forman la aplicación y la plataforma de ejecución. En ese momento, es responsabilidad del *planner* elegir implementaciones concretas para cada una de la instancias y

definir un despliegue para la aplicación, esto es, asignar cada instancia al nodo de la plataforma en el que va a ejecutar y elegir el tipo de mecanismo de comunicación a utilizar para cada conexión entre componentes.

La figura 6.18 recoge un posible despliegue que se propone para la aplicación *ScadaDemo*, en la que se muestran las implementaciones escogidas para cada instancia. En este caso se trata de aquellas que son compatibles con la tecnología Ada-CCM. En el caso de las instancias de tipo *IOCard* deben escogerse además implementaciones compatibles con la tarjeta de adquisición que se utiliza en la plataforma, en este caso, una tarjeta PCI9111DG de AdLink Technology Inc. Este es el caso de la implementación *PCI9111IOCard*. La información necesaria para encontrar dichas implementaciones se encuentra en sus correspondientes descriptores de implementación.

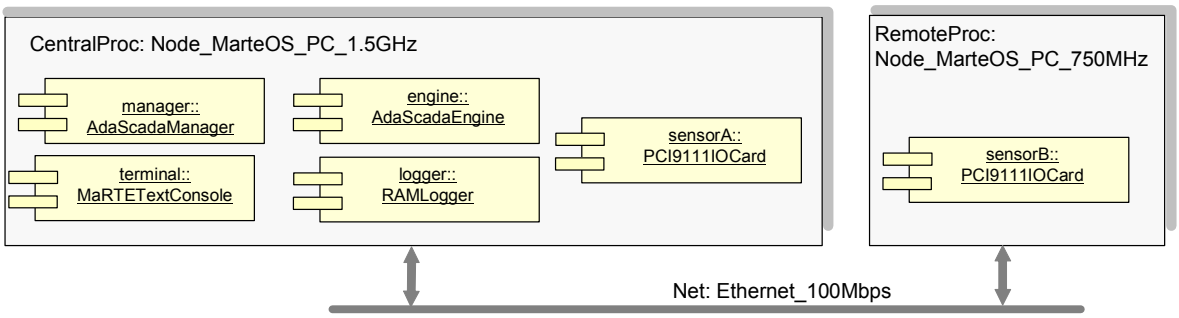


Figura 6.18: Despliegue de la aplicación *ScadaDemo*

Como muestra la figura 6.19, para tomar sus decisiones el *planner* utiliza el modelo que describe la plataforma de ejecución, *ScadaDemoDomain.tdm.xml,* que es elaborado por el agente *Domain Administrator* y que se puede consultar en la página 278 (anexo C). En este caso, la plataforma de ejecución está formada por dos nodos de tipo PC ejecutando el sistema operativo MaRTE OS, conectados a través de una red tipo Ethernet de 100 Mbps. La mayoría de las instancias se encuentran en el nodo central, *CentralProc*, mientras que la instancia *sensorB* se encuentra instalada en un nodo remoto, *RemoteProc*, al estar controlada desde él la tarjeta de adquisición asociada. Desde el descriptor de la plataforma se hace referencia a los modelos de tiempo real de todos sus elementos, que se deben encontrar almacenados en el repositorio. Para configurar dichos modelos, es necesario asignar valor a alguna de las propiedades de tiempo real declaradas en los descriptores. En el caso de los nodos, se ajusta su modelo a su velocidad real a través del parámetro *speedFactor*:

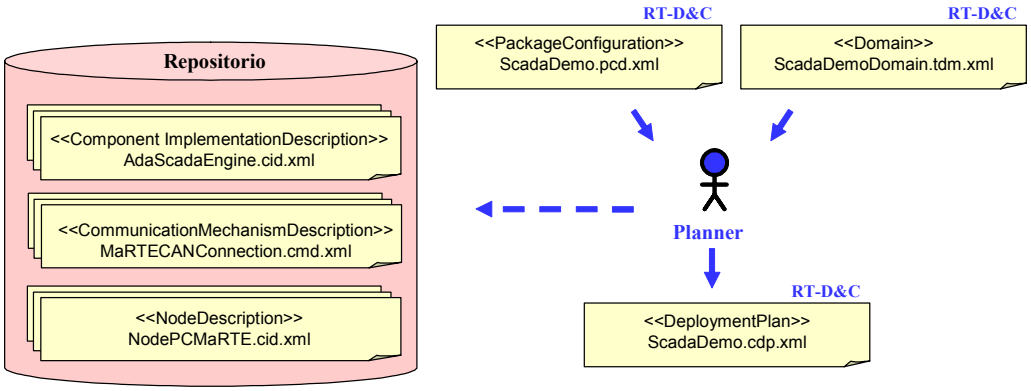


Figura 6.19: Artefactos y ficheros utilizados y generados por el *planner* para definir el plan de despliegue inicial

- el nodo *RemoteProc* corresponde a un procesador de 750MHz, que coincide con la velocidad del procesador que se toma como referencia y por tanto su valor de *speedFactor* es igual a 1.
- el nodo *CentralProc* corresponde a un procesador de 1.5 Ghz, y por lo tanto, le corresponde un valor de *speedFactor* igual a 2.

En el caso de la red, el valor de *speedFactor* es también igual a 1, pues se trata de la misma red que se usa como referencia. Como muestra del tipo de modelo de un elemento de la plataforma, se puede consultar el modelo de tiempo real del nodo de tipo PC de 750 MHz ejecutando MaRTE OS, *Node_MaRTEOS_PC_750MHz.rtm.xml*, que se encuentra en la página 279.

Definición del plan de despliegue

El *planner* formula el plan de despliegue inicial en el fichero *ScadaDemo.cdp.xml*, que se muestra en la página 280 (anexo C) y cuyo elemento raíz es del tipo RT-D&C *DeploymentPlan*. Para generar el fichero, comienza por declarar cada instancia de componente, tomando directamente la información del fichero *ScadaDemo.pcd.xml* elaborado por el *assembler*. En base a él, asigna a cada instancia su nombre y sus valores de configuración de negocio. La información que el *planner* añade consiste en:

- La implementación de componente elegida.
- El nodo en que va a ser instanciada, *CentralProc* o *RemoteProc*.
- Los valores de los parámetros de planificación o de tiempo real que son específicos de las implementaciones, o al menos aquellos que son necesarios para generar el modelo de tiempo real de la aplicación. Las propiedades que influyen en la planificación de la aplicación pueden recibir en este plan de despliegue valores por defecto (no siendo necesario asignarles ningún valor explícito), pues serán calculadas por las herramientas. En este ejemplo, no es necesario que el *planner* asigne valor a ninguna de las propiedades de este tipo pues están todas mapeadas a valores del modelo de tiempo real.
- Los valores de *stimulusId* correspondientes a los puertos de activación y a las propiedades de tipo *TransactionId* que se hayan definido. Este es un aspecto indispensable a incluir en el plan de despliegue inicial, pues las herramientas de configuración de la planificación necesitan estos valores como punto de partida para la obtención de los mapeados entre valores de *stimulusId* y entre valores de *stimulusId* y de parámetros de planificación.

La asignación de *stimulusId* realizada por el *planner* en la aplicación *ScadaDemo* se recoge en las 4 primeras columnas de la tabla 6.3, mientras que la columna de la derecha muestra las transacciones a las que las herramientas asocian internamente los valores de *stimulusId*, en base a la información disponible en los descriptores de los componentes. Como se observa, los puertos de tipo *OneShotActivation* (*notifyTh* de la instancia register y *keyboardTh* de la instancia manager) no corresponden a ninguna transacción, pues se utilizan con diferentes objetivos (ejecutar invocaciones en modo asíncrono y atender al teclado, respectivamente)

Una vez configuradas las instancias, el *planner* declara sus conexiones, para lo que utiliza de nuevo como base la descripción del ensamblado realizada por el *assembler*. De allí toma el nombre de la conexión y las instancias conectadas. El *planner* completa la declaración de cada conexión eligiendo el tipo de servicio de comunicación que se quiere utilizar para realizarla. Para ello, consulta en el modelo de la plataforma los servicios que se encuentran disponibles, y escoge uno que sea compatible con el tipo de red utilizado para la conexión. En el fichero de descripción del dominio *ScadaDemo*, se indica que los nodos soportan conexiones a través de

RT-EP y de CAN; sin embargo, al haberse escogido una red de tipo Ethernet, únicamente las conexiones a través de conectores RT-EP son compatibles. Consultado su correspondiente descriptor, el *planner* conoce qué propiedades debe asignar a la conexión para que las herramientas puedan generar su modelo de tiempo real y su código. Al igual que ocurre con las instancias de los componentes, aquellas propiedades que afecten a la planificación pueden recibir valores por defecto pues serán calculadas por las herramientas de análisis. En este caso, la única conexión remota es la que une las instancias *engine* y *sensorB*. Las propiedades que se definen para el tipo de conexión escogida son:

- *execDispatchPriority*: Prioridad de base con la que se atiende la llegada de invocaciones realizadas a través del correspondiente conector. Esta propiedad está mapeada a una propiedad del modelo de tiempo real, luego no es necesario asignarle valor.
- *channelId*: Canal a través del que el conector realiza sus envíos. Esta propiedad sí debe ser asignada por el *planner* para cada conexión. En este caso se le asigna valor 1..

Tabla 6.3: Asignación de *stimulusId* a puertos y propiedades realizada por el *planner*

Instancia	Puerto	Propiedad	StimulusId	Transacción
manager	keyboardTh		1	
	displayTh		2	displayTransInst
		changeTransId	3	changeTransInst
		checkingTransId	4	checkingTransInst
engine	samplingTh		5	samplingTransInst
	loggingTh		6	loggingTransInst
register	notifyTh		7	

Configuración de la planificación

Una vez definido el plan de despliegue y asignadas todas las propiedades de tiempo real necesarias para general el modelo, el *planner* hace uso de la herramienta de composición de modelos, *MastModelComposer*, y obtiene el modelo MAST de la aplicación. Se genera un modelo MAST para cada situación de tiempo real definida para la aplicación. A continuación, se explica el caso de la primera situación de tiempo real, *TwoSupervisions*.

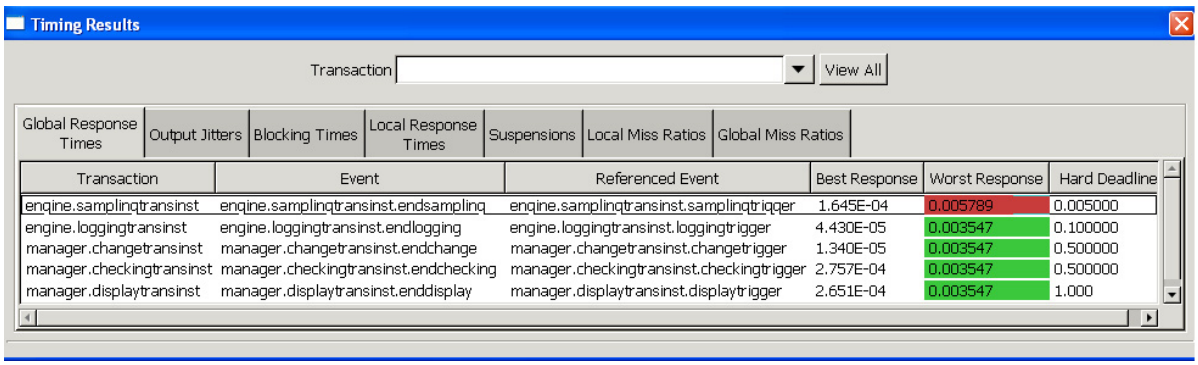
El ciclo de diseño y análisis del modelo de tiempo real comienza tratando de analizar el sistema con los valores por defecto de las propiedades de planificación. Sin embargo, el modelo generado para la aplicación *ScadaDemo* no es analizable directamente con las herramientas analíticas ofrecidas en el entorno MAST, debido a dos causas principales:

- Una de las transacciones es no lineal, incluye una bifurcación (modelada mediante un *Event_Handler* de tipo *Multicast*), y por tanto no es analizable con las técnicas actualmente disponibles en MAST.
- Como consecuencia de la estrategia para la obtención de las transformaciones de *stimulusId*, por cada nueva invocación realizada en un componente se incluye un nuevo *Scheduling_Server*. Si la invocación se realiza teniendo un recurso compartido (elemento de tipo *Shared_Resource*) tomado, el cambio de *Scheduling_Server* no está permitido en MAST (el recurso debe ser liberado previamente). Este tipo de situaciones no son difíciles de encontrar, especialmente cuando se realizan invocaciones remotas, por lo que

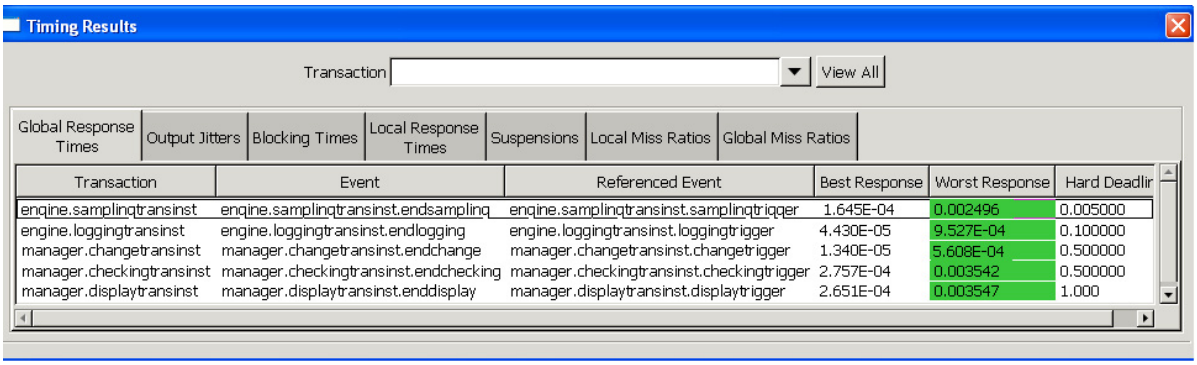
la eliminación de esta restricción es uno de los cambios que se han propuesto para la nueva versión de MAST.

Para mostrar la utilidad de las herramientas, linealizamos los modelos internos de los componentes, de manera que el modelo final resulte analizable. Una vez obtenido este modelo, se utiliza como entrada de las herramientas de análisis MAST. Se elige la técnica de análisis basada en *offsets* [PG99] (escogiendo la opción *Offset_Based_Optimized* en la interfaz de la herramienta MAST), que es la que ofrece unos resultados más exactos dentro de las técnicas disponibles para sistemas distribuidos. La aplicación resulta no planificable, como era de esperar al tomarse los valores por defecto de las prioridades y los techos de prioridad. En la parte superior de la figura 6.20 se muestra una captura de la pantalla de la herramienta MAST donde se ve como la transacción *engine.samplingTransInst* (que es la más restrictiva) no cumple su plazo.

Para intentar conseguir la planificabilidad de la aplicación se invoca a continuación la herramienta MAST con la opción de asignación de prioridades y cálculo de techos de prioridad. Como la aplicación es distribuida, debe elegirse alguno de los mecanismos de asignación adecuados a ese tipo de sistemas, como el HOPA o el "templado simulado". Elegimos en este caso el algoritmo HOPA, que generalmente obtiene mejores resultados. La herramienta genera un nuevo modelo, en el que se incluyen los nuevos valores de prioridades y techos. Con este nuevo modelo, se vuelve a invocar la herramienta de análisis, resultando en este caso la aplicación planificable. En la parte inferior de la figura 6.20 aparecen los resultados obtenidos por la herramienta MAST en este caso, donde se puede observar cómo todas las transacciones cumplen sus plazos.



(a) sin asignación de prioridades (valores por defecto)

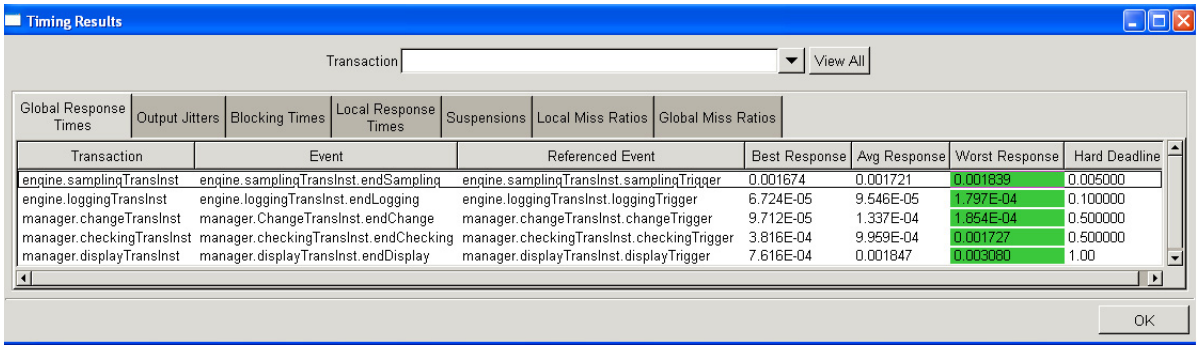


(b) con asignación de prioridades

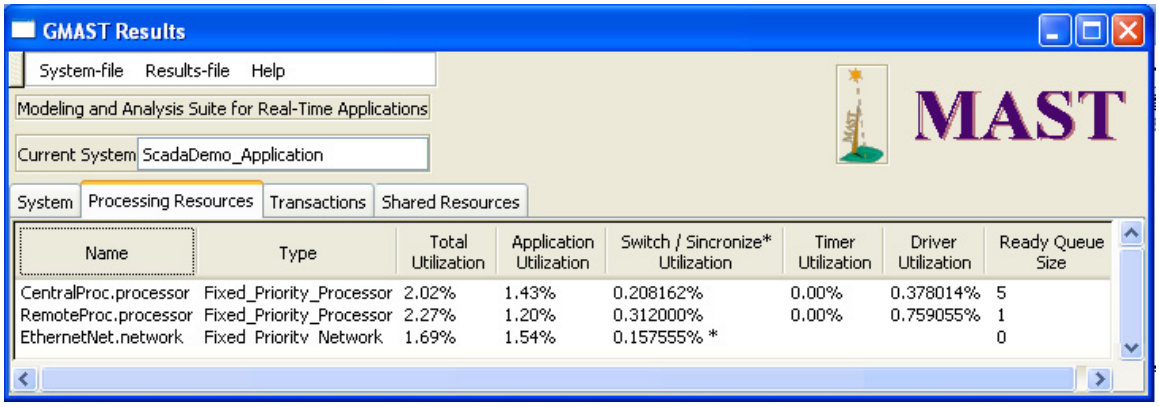
Figura 6.20: Resultados del análisis del modelo MAST de la aplicación *ScadaDemo*

Con los valores de prioridades y techos calculados por la herramienta de análisis, podemos analizar el modelo real de la aplicación (el modelo sin linealizar) haciendo uso de la herramienta de simulación incluida en MAST, SIM_MAST. Aunque no permita evaluar el cumplimiento de los requisitos temporales estrictos, con ella se puede comprobar si los tiempos de respuesta que se calculan por simulación son coherentes con los calculados con las técnicas de análisis, y evaluar así el pesimismo introducido por las herramientas de peor caso. La parte superior de la figura 6.21 muestra una captura de la pantalla con los resultados de la simulación, donde se observa como además del mejor y peor caso, se calculan valores promedio de los tiempos de respuesta de las transacciones. Comparando con los resultados anteriores, se observa como todos los resultados de la simulación para las diferentes transacciones se encuadran dentro del intervalo entre el mejor y el peor caso calculado por las herramientas analíticas, como era de esperar.

Además, la utilización de herramientas de simulación tiene como ventaja que permite calcular otros datos, como porcentajes de utilización detallados, el número de suspensiones máximo que ha sufrido cada thread (*Scheduling_Server*) o el número máximo de elementos que se han encolado a la espera de un recurso compartido. En la parte inferior de la figura 6.21 se muestra una captura de la pantalla de resultados donde se muestran los porcentajes de ocupación de cada recurso de la plataforma. Se distinguen los porcentajes debidos a las actividades propias de la aplicación, a actividades internas de los procesadores, etc. También se muestra para cada procesador el número máximo de *Scheduling_Server* que se encolan en el planificador a la espera de ser ejecutados (*Ready Queue Size*). Este tipo de resultados no pueden ser ofrecidos por las herramientas analíticas.



(a) Tiempos de respuesta de las transacciones



(b) Porcentajes de utilización de procesadores y redes

Figura 6.21: Resultados de la simulación del modelo MAST de la aplicación *ScadaDemo*

Como mostraba la figura 6.14, una vez obtenida una asignación de prioridades que lleve al sistema a satisfacer sus requisitos temporales (plasmada en el fichero MAST *ScadaDemoTwoSupervisionsSchedulable.mmd.xml*), se invoca con dicho modelo y el plan de despliegue inicial como entradas, la herramienta de configuración de la planificación en Ada-CCM. Ésta genera el plan de despliegue final, *ScadaDemoFinal.cdp.xml*, totalmente configurado para la ejecución. Este plan de despliegue se puede consultar en la página 281 (anexo C).

Aunque esta herramienta realiza el proceso de configuración de la planificación de forma oculta al *planner*, a continuación se exponen algunos aspectos del modo en que la herramienta obtiene la configuración. La tabla 6.4 muestra la asociación entre los puertos de activación y las propiedades de tipo *TransactionId* de cada instancia, sus correspondientes *StimulusId*, los *Scheduling_Server* del modelo MAST de los que se obtiene la prioridad que se asociará a dichos *StimulusId* de partida y las prioridades obtenidas del análisis. La herramienta de configuración se basa en estos mapeados para extraer los valores que deben ser asignados en el plan de despliegue final.

Tabla 6.4: Obtención de mapeados entre *stimulusId* y prioridades

Instancia	Puerto/ Propiedad	StimId	Scheduling_Server correspondiente en el modelo MAST	Prioridad
manager	keyboardTh	1	manager.keyboardTh	23
	displayTh	2	manager.displayTransInst.displayTh	1
	changeTransId	3	manager.changeTransInst.changeTh	5
	checkingTransId	4	manager.checkingTransInst.checkingTh	2
engine	loggingTransInst	5	engine.loggingTransInst.loggingTh	18
	samplingTransInst	6	engine.samplingTransInst.samplingTh	27
register	notifyTh	7	engine.notifyTh	14

Por su parte la tabla 6.5 muestra los *Scheduling_Server* de los que se obtiene la prioridad a la que se ejecuta cada invocación de un método de un componente, así como los mapeados de *stimulusId* que se aplican en cada invocación y la prioridad asociada a cada una. Se observa como para invocaciones procedentes de diferentes transacciones, el *Scheduling_Server* es distinto, y por tanto, la prioridad que se asocia puede ser diferente (como ocurre con el método *writeString* de la instancia console).

De la asignación obtenida por las herramientas MAST se extraen también los techos de prioridad que se han de asignar a los puertos de sincronización (propiedades de tipo *Mutex* o *Condition_Variable*) declarados por cada instancia. La tabla 6.6 muestra los puertos de estos tipos declarados por cada instancia, junto a los elementos de tipo *Shared_Resource* del modelo MAST de los que se toman su techo de prioridad y los valores obtenidos en este caso.

Todas estas asignaciones se plasman en el plan de despliegue final, siguiendo los formatos que se mostraron en el capítulo 3. Por ejemplo, los mapeados de *stimulusId* que se han mostrado en la tabla 6.5 son aplicados en tiempo de ejecución por los conectores, por lo que su configuración se realiza en la declaración de las conexiones entre componentes. La tabla 6.7 muestra la configuración que recibe cada conexión.

Tabla 6.5: Mapeado entre valores de *stimulusId* por operación de componente

Instancia	Método	StimId In	StimId Out	SchedulingServer correspondiente en el modelo MAST	Prioridad
engine	getLast LoggedMssg	2	8	manager_scadaPortToengine_control Port.manager.displayTransInst. getLastLoggedMssgServer	8
	getBuffered Data	4	9	manager_scadaPortToengine_control Port.manager.checkingTransInst. getBufferedDataServer	15
sensorA	aiReadCode	6	10	engine_adqPortTosensorA_analogPo rt.engine.samplingTransInst. aiReadCodeServer	21
sensorB	aiReadCode	6	11	engine_adqPortTosensorB_analogPor t.engine.samplingTransInst.aiReadCo deServer	1
logger	log	5	12	engine_logPortToregister_regPort.en gine.loggingTransInst.logServer	12
console	writeString	2	13	manager_consolePortToterminal_rw Port.manager.displayTransInst. writeStringServer	1
		4	14	manager_consolePortToterminal_rw Port.manager.checkingTransInst. writeStringServer	2
	readKey	1	15	manager_consolePortToconsole_rwP ort.manager.checkingTransInst. readKeyServer	17

Tabla 6.6: Configuración de los puertos de sincronización de los componentes

Instancia	Puerto	SharedResource correspondiente en el modelo MAST	Valor
manager	displayMutex	manager.displayMutex	2
engine	dataMtx	engine.dataMtx	27
	mssgMtx	engine.mssgMtx	18
register	logMutex	register.logMutex	14
	logCVMutex	register.logCVMutex	12
sensorA	aiMutex	sensorA.aiMutex	21
	aoMutex	sensorA.aoMutex	1
sensorB	aiMutex	sensorB.aiMutex	1
	aoMutex	sensorB.aoMutex	1

Tabla 6.7: Configuración de los mapeados de *stimulusId* en los conectores

Connection	Operation	InputId	OutputId
manager_scadaPortToengine_controlPort	getLastLoggedMssg	2	8
	getBufferedData	4	9
engine_adqPortToinnerSensor_analogPort	aiReadCode	6	10
engine_adqPortTosensor_analogPort	aiReadCode	6	11
engine_logPortToregister_regPort	log	5	12
manager_consolePortToterminal_rwPort	writeString	2	13
		4	14
	readKey	3	15

El plan final debe incluir también la configuración de los servicios del entorno. En la tabla 6.8 se muestra la configuración del *SchedulingService*, con los correspondientes mapeados entre valores de *stimulusId* y prioridades (cuyos valores y la forma en que se obtienen se han mostrado ya en las tablas anteriores).

Tabla 6.8: Configuración del *SchedulingService* en el nodo *CentralProc*

StimulusId	Prioridad	StimulusId	Prioridad	StimulusId	Prioridad
1	23	6	27	12	12
2	1	7	14	13	1
3	5	8	8	14	2
4	2	9	15	15	17
5	18	10	21		

La configuración del *CommunicationSchedulingService*, que se plasma en la tabla 6.9, es muy sencilla en este caso. Únicamente se realiza una invocación remota, por lo que sólo es necesario indicar los parámetros de planificación que corresponden al valor de *stimulusId* asignado a dicha invocación. Por cada *stimulusId* se asigna:

- una prioridad, que en el nodo cliente (*CentralProc)* corresponde con la prioridad del mensaje de envío de la invocación, mientras que en el nodo servidor (*RemoteProc*) corresponde a la prioridad del mensaje de retorno,
- el canal en el que la parte *proxy* del conector se queda a la espera de respuesta, y por tanto, el canal por el que la parte *servant* debe enviar dicha respuesta una vez ejecutada la invocación.

Tabla 6.9: Configuración del *CommunicationSchedulingService* en ambos nodos

Nodo	StimulusId	Prioridad	Canal
CentralProc	10	1	2
RemoteProc	10	128	2

La configuración del *ThreadingService* requiere únicamente asignar el tamaño del pool de threads que se utiliza, el cual debe ser suficiente como para cubrir todas las posibles peticiones. Los criterios que se utilizan para su cálculo son los siguientes:

- En el caso de que la aplicación sea monoprocesadora y que todas las invocaciones realizadas sean síncronas, el número de threads necesarios coincide con la suma de todos los puertos de activación de los componentes instanciados en el nodo.
- En el caso de una aplicación distribuida, y para cada nodo, hay que sumar al número anterior (número de puertos de activación) aquellos threads que son necesarios para atender las posibles invocaciones concurrentes que se reciban a través de los diferentes conectores. Para calcular este número se tienen en cuenta los siguientes aspectos:
  - Como mínimo, y debido al modo en que se han implementado los conectores RT-EP, se necesitan tantos threads como conectores de recepción (fragmentos de tipo *servant*) existan en el nodo. Esto es debido a que por cada conector se necesita un thread que atienda el canal de recepción de mensajes por el que llegan las posibles invocaciones (se usa un canal de recepción por cada conector).
  - Cuando se recibe una invocación, el thread que atendía el canal se encarga de su ejecución, por lo que se necesita otro thread que siga a la espera de nuevas invocaciones (que pueden ser más prioritarias).
  - El número máximo de threads que un conector puede necesitar simultáneamente corresponde al número de threads distintos desde los que puede recibir invocaciones más el thread que se queda a la espera.

  Como desde un mismo thread no se pueden realizar invocaciones simultáneas a varios conectores, el número total de threads que se necesitan en un nodo para atender las invocaciones recibidas tiene como cota máxima (podría ser optimizado) el número de threads distintos que realizan invocaciones en los conectores instalados en ese nodo. Este número se puede calcular fácilmente en base al modelo MAST final de la aplicación.

Aplicando estas reglas, el número de threads resultante en la aplicación *ScadaDemo* es 5 para el nodo *CentralProc* (por los 5 puertos de activación), y 2 en el caso del nodo *RemoteProc* (1 como consecuencia del único conector instalado y otro por el número de threads que pueden realizar invocaciones en él). En la tabla 6.10 se muestra el fragmento del plan de despliegue final donde se realiza la configuración de los servicios Ada-CCM para el nodo *CentralProc*.

6.3.4. Fase de preparación y ejecución

La fase de ejecución parte del plan de despliegue final. Con él como entrada, el *executor* invoca la herramienta *AdaApplicationGenerator*, que genera los ejecutables correspondientes a cada partición. En el caso de MaRTE, donde sólo se admite una partición por nodo, corresponden a los ejecutables que serán cargados en cada nodo. Internamente la herramienta sigue el siguiente proceso:

1. Procesa el plan de despliegue global, y lo divide en un conjunto de planes de despliegue parciales, uno por cada nodo, donde se incluyen sólo las instancias y las conexiones correspondientes a dicho nodo.
2. Por cada conector utilizado en la aplicación, se genera su código y se declaran las correspondientes instancias de fragmentos *proxy* o *servant* en el correspondiente plan de despliegue parcial.
3. Por cada despliegue parcial, se genera un fichero .adb que incluye el código necesario para configurar, instanciar, conectar y activar cada instancia de componente o fragmento

Tabla 6.10: Configuración de los servicios del entorno en la aplicación *ScadaDemo*

```
<qosConfiguration node="CentralProc">
     <serviceConfig name="ThreadingService">
       <threadingServiceConfig poolSize="5" poolPriority="31"/>
     </serviceConfig>
     <serviceConfig name="SchedulingService">
       <schedulingServiceConfig>
         <schedParamAssignment stimulusId="1" priority="23"/>
         <schedParamAssignment stimulusId="2" priority="1"/>
         <schedParamAssignment stimulusId="3" priority="5"/>
         <schedParamAssignment stimulusId="4" priority="2"/>
         <schedParamAssignment stimulusId="5" priority="18"/>
         <schedParamAssignment stimulusId="6" priority="27"/>
         <schedParamAssignment stimulusId="7" priority="14"/>
         <schedParamAssignment stimulusId="8" priority="8"/>
         <schedParamAssignment stimulusId="9" priority="15"/>
         <schedParamAssignment stimulusId="10" priority="21"/>
         <schedParamAssignment stimulusId="12" priority="12"/>
         <schedParamAssignment stimulusId="13" priority="1"/>
         <schedParamAssignment stimulusId="14" priority="2"/>
         <schedParamAssignment stimulusId="15" priority="17"/>
       </schedulingServiceConfig>
     </serviceConfig>
     <serviceConfig name="CommunicationService">
       <communicationServiceConfig>
         <schedParamAssignment stimulusId="11">
           <rtepParams channelId="2" MssgPriority="1"/>
         </schedParamAssignment>
       </communicationServiceConfig>
     </serviceConfig>
</qosConfiguration>
```

de conector, así como configurar los servicios Ada-CCM de cada nodo (previamente a la activación de los componentes).

4. Se compilan los ficheros .adb, con las librerías .a correspondientes a los componentes y a los conectores generados, y como resultado se obtienen los ejecutables a cargar en cada nodo de ejecución.

A continuación, el *executor* usa la herramienta *AdaApplicationLauncher*, que se encarga del lanzamiento de la aplicación. En el caso de nodos MaRTE, lo que hace la herramienta es copiar los ejecutables en el directorio de exportación, desde el que cada nodo de ejecución descarga el ejecutable que le corresponde una vez que es iniciado, y a continuación, iniciar los nodos de ejecución.

6.4. Conclusiones

En este capítulo se ha presentado un ejemplo que aplica el proceso de diseño y el resto de soluciones propuestas a lo largo de la tesis. El ejemplo se elabora de acuerdo a la tecnología Ada-CCM, que constituye una implementación de RT-CCM formulada enteramente en Ada 2005 y que se utiliza como método de validación de las propuestas realizadas en el capítulo 4.

7. Conclusiones y trabajo futuro

7.1. Conclusiones

Las aportaciones que se presentan en esta tesis constituyen en su conjunto una metodología de desarrollo de aplicaciones de tiempo real basadas en componentes software reutilizables. Con la metodología propuesta se pueden desarrollar aplicaciones distribuidas de tiempo real estricto por composición de componentes manejados de forma opaca, y en las que se puede analizar, y en su caso garantizar, el cumplimiento de los requisitos temporales que tengan establecidos en su especificación. A continuación se detallan las contribuciones más relevantes que se aportan.

Metodología modular de modelado de comportamiento temporal

Se ha definido la metodología de modelado Mod-MAST [LMD06], que permite describir de forma modular, independiente y reutilizable las características de los módulos software y hardware que influyen en el comportamiento temporal de los sistemas en los que están integrados. Es una metodología de modelado orientada a la componibilidad, que facilita la construcción de los modelos que predicen el comportamiento temporal de los sistemas de tiempo real por ensamblado de los modelos de los módulos que constituyen el sistema.

La metodología Mod-MAST se define formalmente a través de un metamodelo y su ámbito de aplicación es muy general: puede ser aplicada a diferentes estrategias de diseño modular y no específicamente a aplicaciones basadas en componentes.

Para aplicaciones basadas en componentes se ha definido la metodología CBS-MAST. Su metamodelo resulta de incorporar al metamodelo Mod-MAST nuevos elementos de modelado con la semántica propia del paradigma de componentes. CBS-MAST se utiliza en dos puntos clave del proceso de diseño definido para sistemas de tiempo real basados en componentes:

- Se utiliza para formular los modelos reutilizables que describen las características de comportamiento temporal de las implementaciones de los componentes. Son elaborados por los desarrolladores de los componentes software e incluidos en los paquetes con los que se distribuyen los componentes de tiempo real.
- Se utiliza también como base para construir el modelo de comportamiento temporal de un sistema basado en componentes por composición de los modelos de los componentes y de la plataforma que lo forman. Este modelo completo del sistema se genera a partir de la descripción de su estructura como ensamblado de componentes, despliegue y carga de trabajo, sin que el diseñador de la aplicación tenga que conocer los detalles de los modelos temporales ni el código de cada uno de los componentes o elementos de la plataforma que lo forman. Es decir, gracias a CBS-MAST se cumple el requisito de opacidad en el modelado de aplicaciones basadas en componentes.

Especificación de configuración y despliegue de aplicaciones de tiempo real basadas en componentes

Se ha definido el metamodelo RT-D&C [LCD10], que extiende el metamodelo de datos de la especificación D&C de OMG con objetivo de incorporar aquellos metadatos relativos al comportamiento temporal de los componentes y las aplicaciones que se necesitan en el proceso de diseño de tiempo real para garantizar el cumplimiento de los requisitos temporales impuestos en las aplicaciones. Las extensiones definidas en RT-D&C proporcionan a los diseñadores de aplicaciones capacidad para:

- Construir el modelo de tiempo real de la aplicación que permite predecir su comportamiento temporal y analizar su planificabilidad.
- Asignar en el plan de despliegue de las aplicaciones los valores de los parámetros de configuración de los componentes y de los recursos de la plataforma que controlan la planificabilidad de las aplicaciones, de forma que se cumplan los requisitos temporales de la aplicación. Gracias a las extensiones introducidas, estos valores pueden ser deducidos del análisis del modelo anterior.

La característica más relevante de la extensión propuesta es que distribuye la información relativa a comportamiento temporal entre los diferentes elementos que describen los componentes y las aplicaciones, de forma que cada agente que participa en el desarrollo dispone de la información que necesita para realizar su tarea y tomar sus decisiones sin necesidad de conocer los códigos ni los modelos de los componentes. Los agentes sólo deben saber manejar los metadatos que se definen en RT-D&C e interpretar los resultados que generan las herramientas que los procesan.

Proceso de diseño de aplicaciones de tiempo real basadas en componentes

Se ha definido un proceso de diseño completo de aplicaciones construidas por ensamblado de componentes y cuya especificación se realiza en base a un modelo reactivo sobre el que se imponen los requisitos temporales [LDM09][LCD10b]. Este proceso constituye la base de la metodología, ya que sobre él se integran el resto de contribuciones de la tesis, cuyo objetivo global es certificar el cumplimiento de los requisitos especificados en las aplicaciones.

El proceso se ha definido como una extensión del proceso estándar definido en la especificación D&C de OMG, de modo que pueda ser fácilmente adaptado a diferentes entornos de componentes. Para cada agente involucrado en el proceso se han definido las nuevas tareas de las que es responsable y que aparecen a consecuencia de la naturaleza de tiempo real de las aplicaciones. A su vez, para cada tarea se ha definido la información de entrada que se requiere y la información de salida que se genera, siempre desde un punto de vista lo más independiente de la plataforma posible.

En la tesis se muestra una adaptación del proceso definido al caso en el que se utilice una metodología de modelado basada en el entorno MAST, y una tecnología de componentes de tipo RT-CCM.

Tecnología de componentes de tiempo real

Se ha definido la tecnología de componentes de tiempo real RT-CCM [LBD10][LDP08][LDM08], que introduce los elementos necesarios para garantizar la predictibilidad temporal de las aplicaciones desarrolladas de acuerdo a ella. La tecnología se define de forma independiente de la plataforma de ejecución, por lo que la predictibilidad temporal sólo se alcanzará si es implementada sobre una plataforma de tiempo real.

La tecnología utiliza como punto de partida un modelo de componentes estándar y con bastante difusión, como es LwCCM de OMG. RT-CCM añade a este modelo una serie de elementos que permiten al contenedor controlar todos los aspectos del código del componente que influyen en la planificación de la aplicación. Con su uso, se puede realizar la configuración de la planificación de las aplicaciones sin necesidad de acceder al código de negocio de los componentes, que es opaco. Dentro del proceso completo de diseño, los valores a asignar en la configuración de la planificación se extraen del análisis del modelo de tiempo real de la aplicación.

Gracias al uso de conectores para implementar las conexiones entre componentes y a la generación automática del código de los contenedores, los diseñadores de componentes RT-CCM pueden elaborar su código independientemente de la plataforma de ejecución sobre la que vayan a ejecutar. El modelo de referencia de RT-CCM se define también de forma independiente de la plataforma de ejecución, por lo que se pueden elaborar implementaciones concretas adaptadas a diferentes plataformas, sin más que adaptar el código de los servicios del entorno y las herramientas de generación de contenedores y conectores.

Tecnología de componentes de tiempo real en lenguaje Ada

Se ha desarrollado la tecnología Ada-CCM [LDP08][LDM08]. Es una implementación concreta de RT-CCM, que utiliza el lenguaje Ada 2005 para desarrollar el código de los componentes y del entorno de ejecución. Aunque inicialmente se desarrolló como un medio para la validación de la tecnología RT-CCM, el resultado tiene interés por sí mismo, ya que constituye una tecnología de componentes muy simple, homogénea y ligera. Es idónea para utilizarse en plataformas embebidas de perfil mínimo al sólo requerir de los procesadores que dispongan de un entorno de ejecución Ada, y que los procesadores estén interconectados por una red de comunicación de tiempo real. La tecnología Ada-CCM se ha probado sobre una plataforma de tiempo real constituida por nodos con sistema operativo MaRTE OS e interconectados mediante una red Ethernet utilizando el protocolo RT-EP.

La inclusión en la versión Ada 2005 del concepto de interfaz ha sido clave en el desarrollo de una tecnología de componentes de tiempo real basada en lenguaje Ada.

Otros aspectos

Un aspecto importante a resaltar de las contribuciones de esta tesis es que en todas ellas se ha aplicado una formulación genérica e independiente de la plataforma:

- Mod-MAST y CBS-MAST son independientes de la tecnología de componentes, o criterio de descomposición modular, que se aplique para construir la aplicación.
- RT-D&C es independiente tanto de la tecnología de componentes como de la metodología de modelado de tiempo real utilizadas.
- RT-CCM es independiente de la plataforma de ejecución o del lenguaje de programación con el que se desarrollen los componentes.

Todas ellas son por tanto, adaptables a diferentes entornos de ejecución y/o de análisis temporal, siempre que estén enmarcados dentro de una metodología de diseño de tiempo real basada en describir la aplicación en base a un modelo reactivo o transaccional.

Los objetivos transversales que se plantearon en el capítulo inicial también se han alcanzado:

- Se consideran componentes con complejidad arbitraria. En todos los aspectos abordados (modelado, formulación de metadatos, tecnología, etc.) un componente puede tener definidos múltiples puertos, implementando a través de ellos diferentes servicios; y

además, no se impone ninguna restricción sobre la actividad interna del componente, pudiendo definirse tanto componentes activos (con flujos de control internos) como pasivos.

- Todas las herramientas requeridas para llevar a cabo las diferentes tareas del proceso de diseño se han definido en base a transformaciones de modelos, acordes a sus respectivos metamodelos. De esta manera, se simplifica su posterior adaptación a tecnologías MDE/ MDA.

7.2. Trabajo futuro

El trabajo desarrollado en esta tesis plantea un conjunto diverso de líneas de trabajo que se pueden abordar a partir de él, algunas de las cuales ya se han iniciado.

Alineamiento con otros estándares de modelado de OMG

La aprobación por parte del OMG del estándar de modelado MARTE conduce a la búsqueda de un mayor alineamiento de la metodología propuesta en esta tesis con dicho perfil. Para hacer la metodología lo más estándar posible, y con ello más fácilmente aplicable, los modelos de los componentes, plataformas y cargas de trabajo deberían formularse independientemente de las herramientas de análisis utilizadas. Para ello, tanto los modelos proporcionados con los componentes, como el modelo final de la aplicación, deberían formularse en base a las primitivas de modelado propuestas en MARTE.

Desde el punto de vista del modelado de bajo nivel, como ya se expuso en el capítulo 2, Mod-MAST y MARTE, en concreto su subperfil de análisis de planificabilidad (SAM), son muy similares, debido a que ambos se apoyan en un modelo reactivo del sistema basado en transacciones. Sin embargo, un aspecto que debería ser abordado para poder utilizar modelos MARTE en aplicaciones basadas en componentes es su capacidad para modelar, desde el punto de vista del comportamiento temporal, entidades de un mayor nivel de abstracción. Para ello, es necesario analizar el resto de elementos de modelado que MARTE ofrece, probablemente en conjunto con otros perfiles propuestos por OMG, como SysML [SysML] o directamente UML 2, con el objetivo de aportar componibilidad y reutilización a los modelos de tiempo real elaborados acorde a MARTE. De este modo, MARTE podrá ser utilizado para formular los modelos de tiempo real que se distribuyen con los componentes y que son referenciados desde los descriptores RT-D&C, y se podrá llevar a cabo el proceso de composición que lleva a la obtención del modelo de la aplicación final. Este modelo podría, o bien ser procesado por herramientas de análisis que admitan modelos MARTE como entrada, o bien ser transformado a un modelo conforme a las herramientas de análisis disponibles (MAST en nuestro caso).

Desarrollo de aplicaciones basadas en componente en plataformas abiertas

Una de las líneas de trabajo que ya ha sido iniciada consiste en adaptar la metodología propuesta en esta tesis a aplicaciones basadas en componentes que ejecutan en plataformas distribuidas abiertas.

Los métodos tradicionales de diseño de sistemas de tiempo real estricto se basan en conocer toda la carga de trabajo de la plataforma que pueda interferir con las tareas que tienen requisitos temporales. Sin embargo, en una plataforma abierta la carga de trabajo no se conoce de forma completa cuando se va a diseñar una aplicación, ya sea porque no se tiene información de todas las aplicaciones que se ejecutan en ella, o porque dicha carga evoluciona muy rápidamente o de forma imprevisible en el tiempo. En estos casos, se necesitan nuevos paradigmas de diseño de

tiempo real, y entre ellos, sobresale el diseño basado en reserva de recursos [RJM98][AGB06]. Esta estrategia se basa en la disponibilidad de un middleware en la plataforma que admite que las aplicaciones puedan establecer contratos de uso de los recursos, a través de los que especifican el tiempo de uso y las restricciones de acceso a los mismos que requieren para planificar sus actividades con las restricciones temporales que tienen establecidas. Si la plataforma (que conoce en todo momento la capacidad de trabajo que tiene comprometida) acepta los contratos de uso que le propone una aplicación, genera los correspondientes servidores internos que proporcionan el acceso a los recursos en los términos contratados, con lo que la aplicación tiene la seguridad de que va a cumplir sus requisitos temporales (siempre que los contratos se hayan diseñado correctamente).

La adaptación de la metodología propuesta en esta tesis a este tipo de estrategias se aborda desde diferentes puntos de vista. Entre ellos se encuentran:

- Extender la metodología de modelado y análisis de tiempo real para dar soporte a este tipo de estrategia. A partir de la información proporcionada por los componentes que forman una aplicación, debemos ser capaces de obtener el conjunto de contratos que la aplicación necesita para verificar sus requisitos temporales. Para ello se debe suponer una plataforma de ejecución virtual, esto es, una plataforma modelada como un conjunto de recursos virtuales que proporcionan la capacidad de procesamiento. Los contratos obtenidos de este modo serán los que posteriormente se negocien con la plataforma de ejecución real.
- Dar soporte a la ejecución de una aplicación basada en componentes sobre una plataforma que ofrezca un middleware basado en reserva de recursos. Gracias a las características de RT-CCM, su adaptación a este tipo de estrategia de planificación resulta sencilla ya que sólo requiere implementar los servicios de concurrencia, sincronización, etc., haciendo uso de los servicios ofrecidos por el middleware de reserva de recursos. El código de negocio de los componentes permanece inalterado respecto a la versión que trabaja sobre una planificación tradicional basada en prioridades fijas, pues son los contenedores los encargados de adaptarlo al nuevo tipo de planificación. Como middleware de reserva de recursos se utilizaría la implementación desarrollada en el grupo CTR dentro del proyecto europeo FRESCOR [FRSH], que implementa el framework de reserva de recursos sobre sistemas distribuidos de tiempo real estricto, esto es, ofrece capacidad para gestionar tanto la capacidad de procesamiento de los procesadores como la capacidad de envío de mensajes a través de las redes de comunicación.

Adaptación de la metodología a aplicaciones de tiempo real no estricto

En la actualidad son cada vez más comunes las aplicaciones en las que se mezclan requisitos temporales estrictos, con otros requisitos de tipo laxo o que exigen ciertos niveles de calidad de servicio o de prestaciones. En este tipo de sistemas, es necesario certificar el cumplimiento de los requisitos temporales estrictos pero también asegurar el cumplimiento de unos ciertos niveles de calidad de servicio. Para evaluar este tipo de requisitos se utilizan técnicas de análisis y metodologías de modelado diferentes de las que se emplean en el análisis de planificabilidad, pero que comparten muchas de sus características. De esto es prueba el hecho de que los subperfiles definidos en MARTE para análisis de planificabilidad (SAM) y para análisis de prestaciones, *Performance Analysis Model (*PAM), tienen ambos un antecesor común, el *Generic Quantitative Analysis Model* (GQAM), sobre el que cada perfil define sus propias extensiones.

Tanto la metodología de modelado como el metamodelo de especificación expuestos en esta tesis pueden ser adaptados para que den soporte al desarrollo de aplicaciones con estas características, tanto aquellas que mezclan requisitos estrictos y laxos, como aquellas que sólo exigen niveles de calidad de servicio.

Evolución hacia entornos integrados basados en tecnología MDA

La complejidad del proceso de desarrollo expuesto exige que sea llevado a cabo por herramientas automáticas, mientras que la continua evolución de las plataformas requiere una renovación continua de estas herramientas, lo cual incrementa los plazos de desarrollo y reduce la fiabilidad al no disponerse de herramientas estables.

Para dar solución a este problema, y como ya se ha expuesto en algunos puntos de la tesis, una de las líneas de trabajo futuro consiste en implementar todo el proceso de desarrollo propuesto en esta tesis utilizando paradigmas MDE/MDA. De este modo, las herramientas son generadas automáticamente en función de los metamodelos que se utilizan para describir las características tanto funcionales como no funcionales del sistema en cada fase del proceso de diseño, con lo que se simplifica su adaptación a nuevas plataformas. De hecho, la utilización de este tipo de estrategias simplificará la adaptación de la metodología a todo el resto de extensiones que se han planteado como trabajo futuro.

Anexo A. Metamodelo Mod-MAST

En este anexo se expone el metamodelo completo de la metodología Mod-MAST. En la figura A.1 se muestra la estructura de paquetes en que se divide el metamodelo, que ya se mostró en la figura 2.11 del capítulo 2.

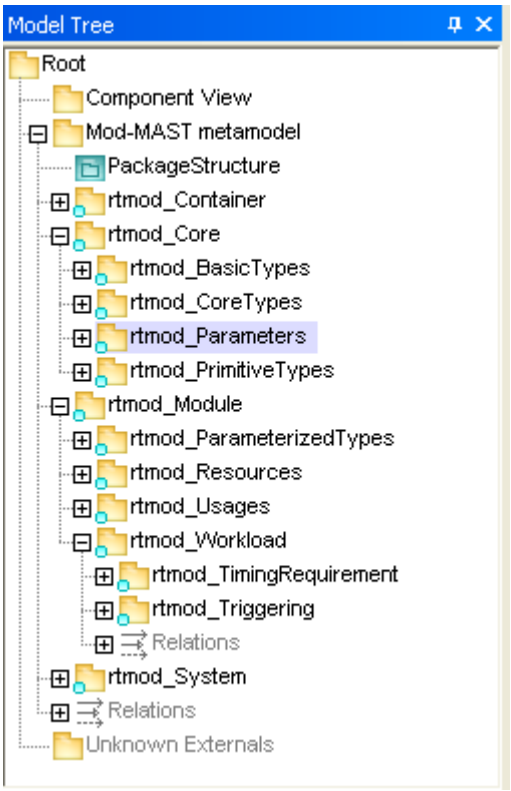
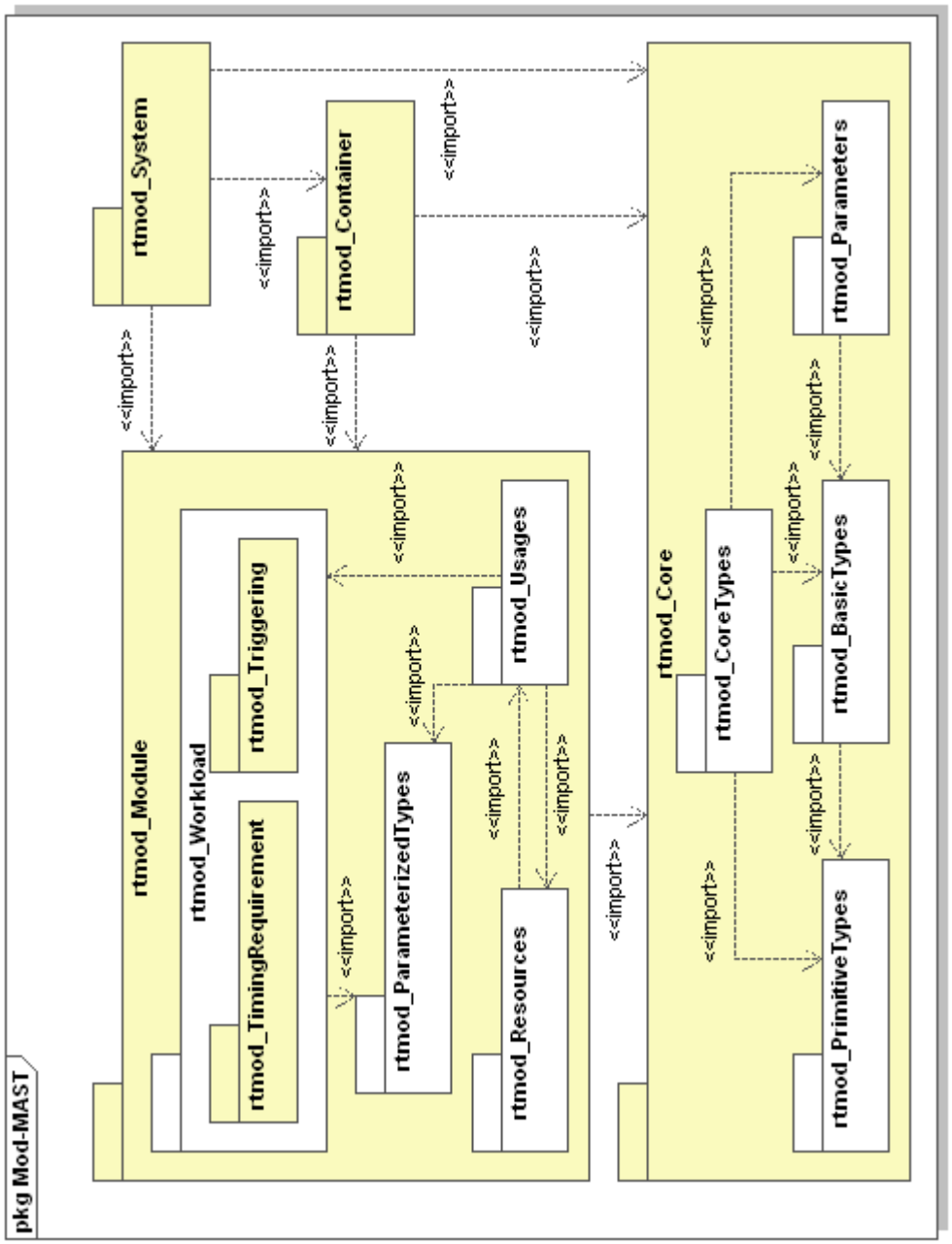
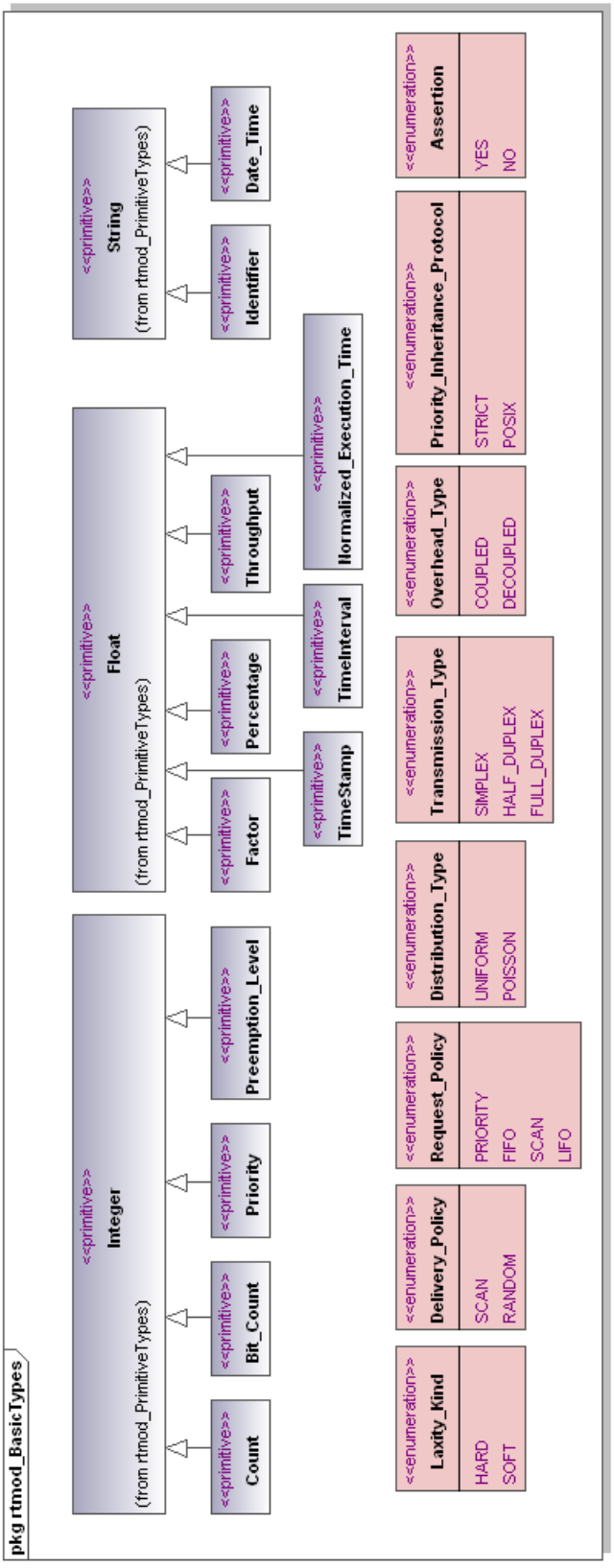


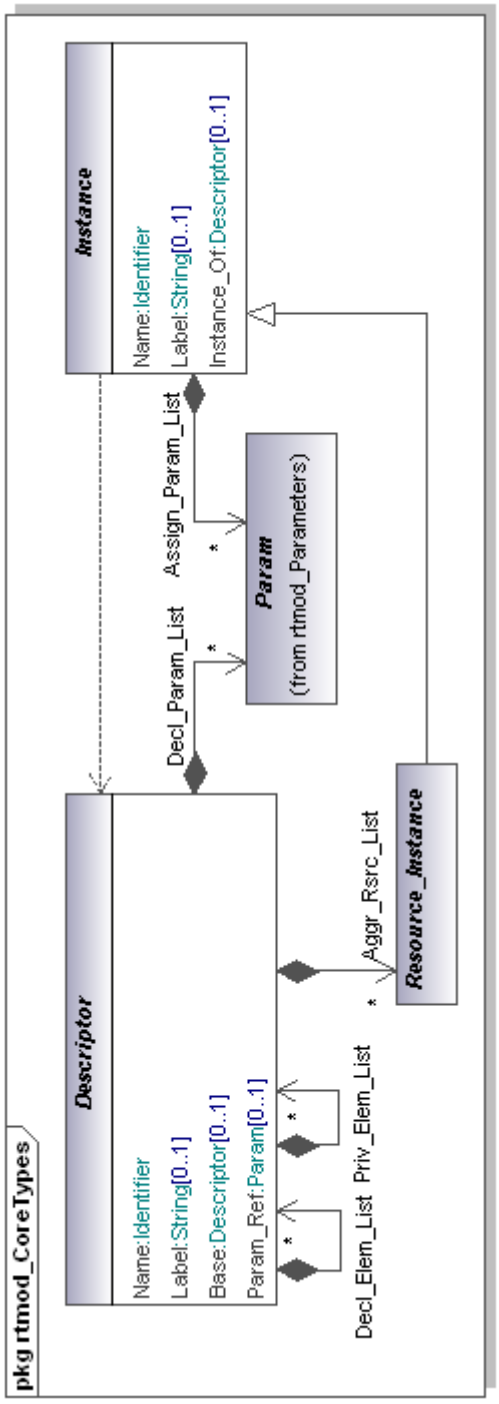
Figura A.1: Estructura de paquetes del metamodelo Mod-MAST

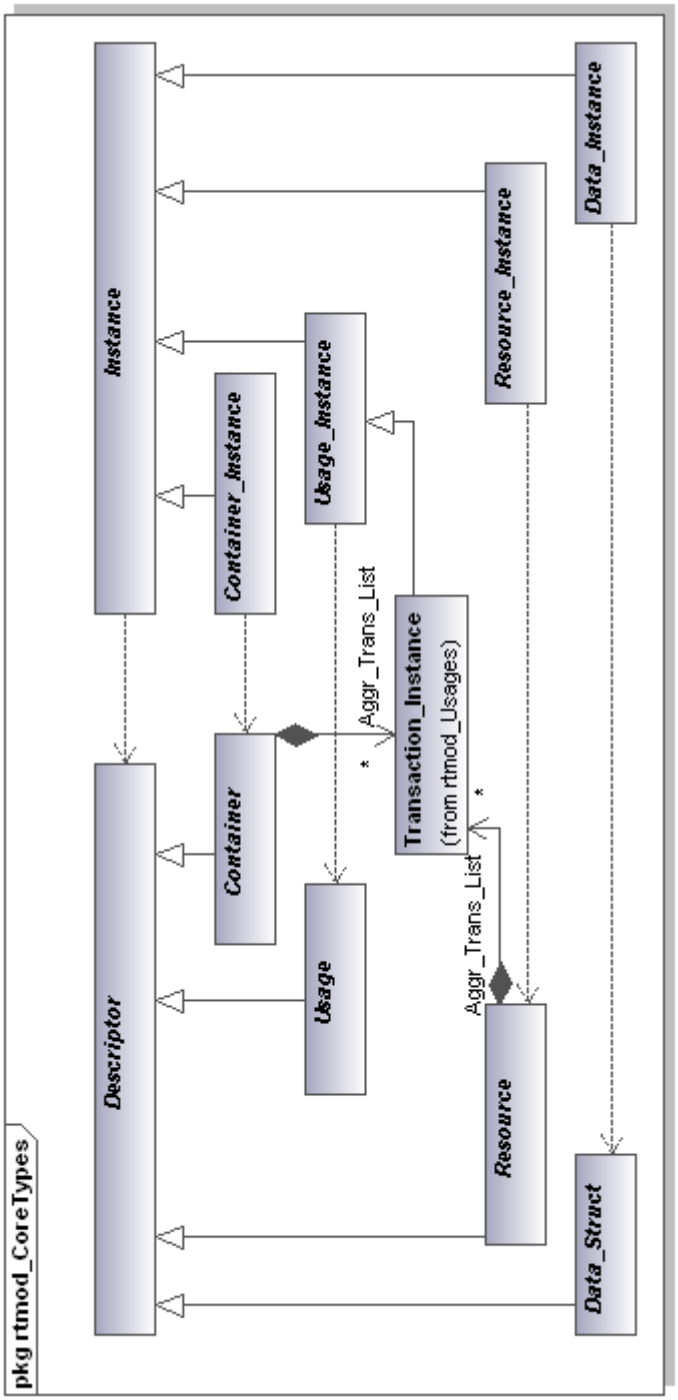
A lo largo del anexo se irán exponiendo las clases que forman el metamodelo a través de diagramas de clases. Como ya expuso en el capítulo 2, para reducir la complejidad y la extensión del metamodelo en este anexo, sólo se muestran los diagramas correspondientes a descriptores. Sus correspondientes instancias sólo se mostrarán explícitamente en aquellos casos en los que presenten alguna peculiaridad que las distinga del caso general. Además, los diagramas de clases donde se muestran los tipos de parámetros admitidos por cada elemento de modelado se muestran sólo para los elementos que se manejan exteriormente, esto es, todos los tipos de contenedores, así como las transacciones y los jobs.

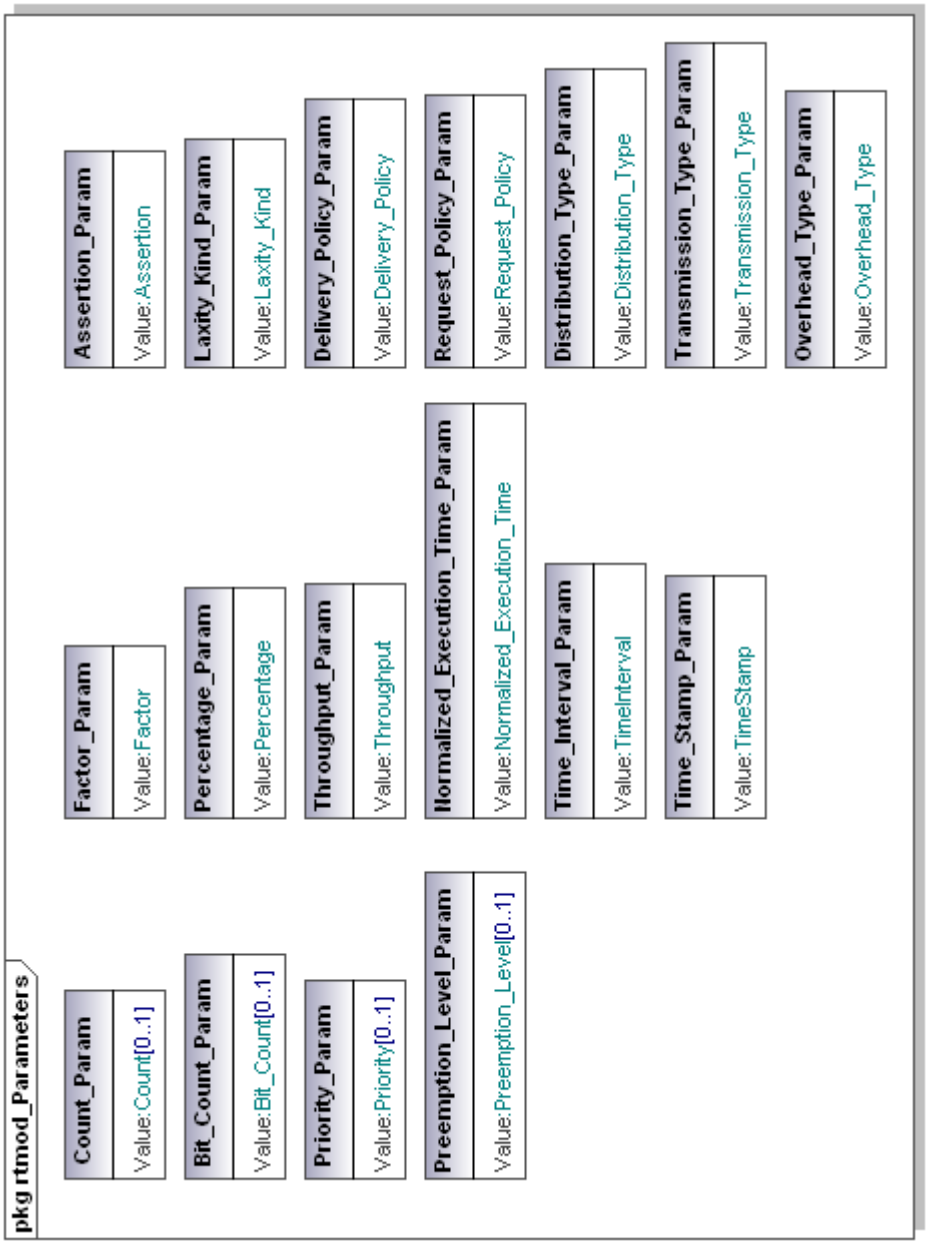
Los diagramas se muestran en forma apaisada para una mayor legibilidad. La lista de diagramas mostrados es la siguiente:

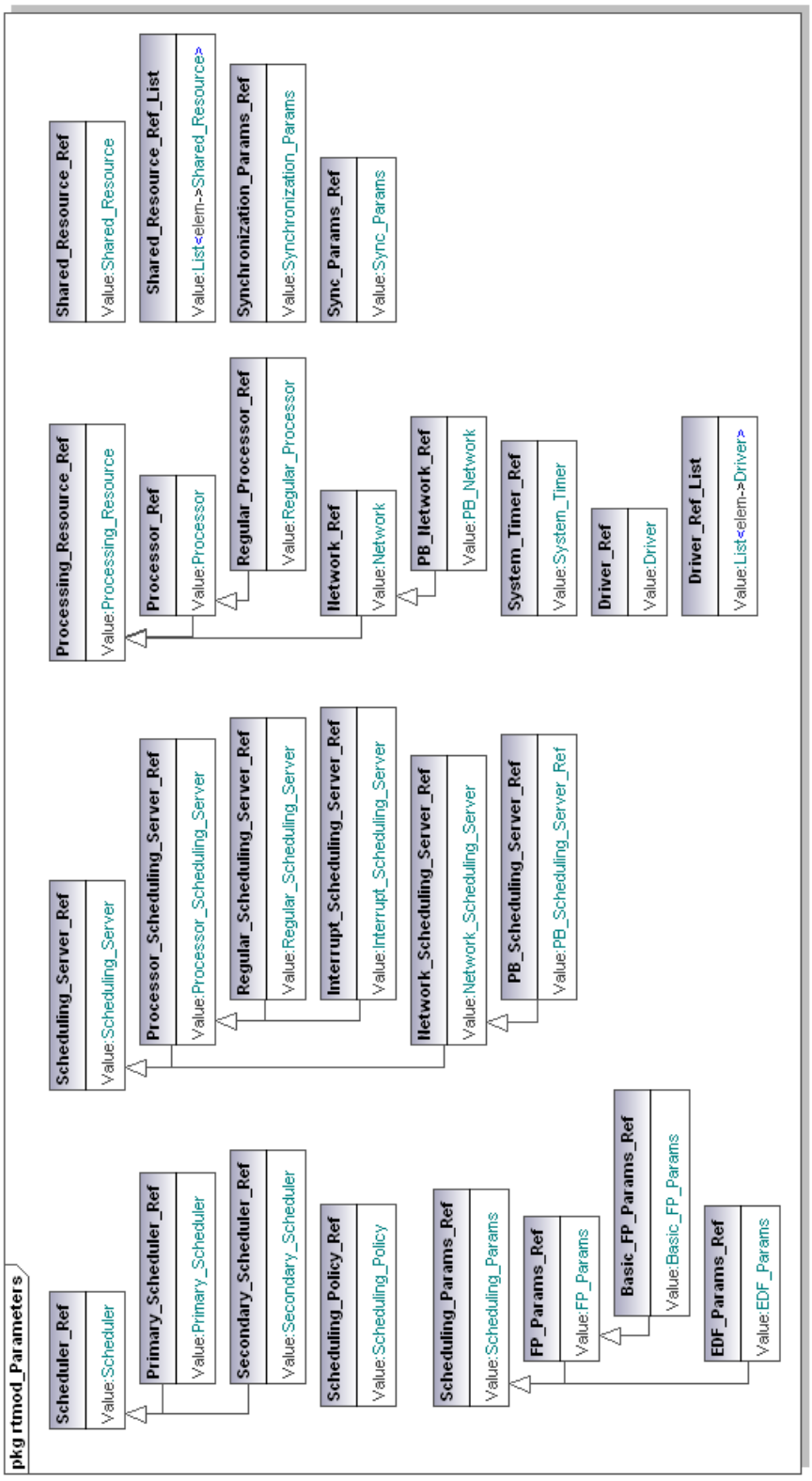

pkg Mod-MAST
rtmod_System
rtmod_Container
rtmod_Module
rtmod_Workload
rtmod_Triggering
rtmod_TimingRequirement
rtmod_ParameterizedTypes
rtmod_Usages
rtmod_Resources
rtmod_Core
rtmod_CoreTypes
rtmod_Parameters
rtmod_BasicTypes
rtmod_PrimitiveTypes
«import»

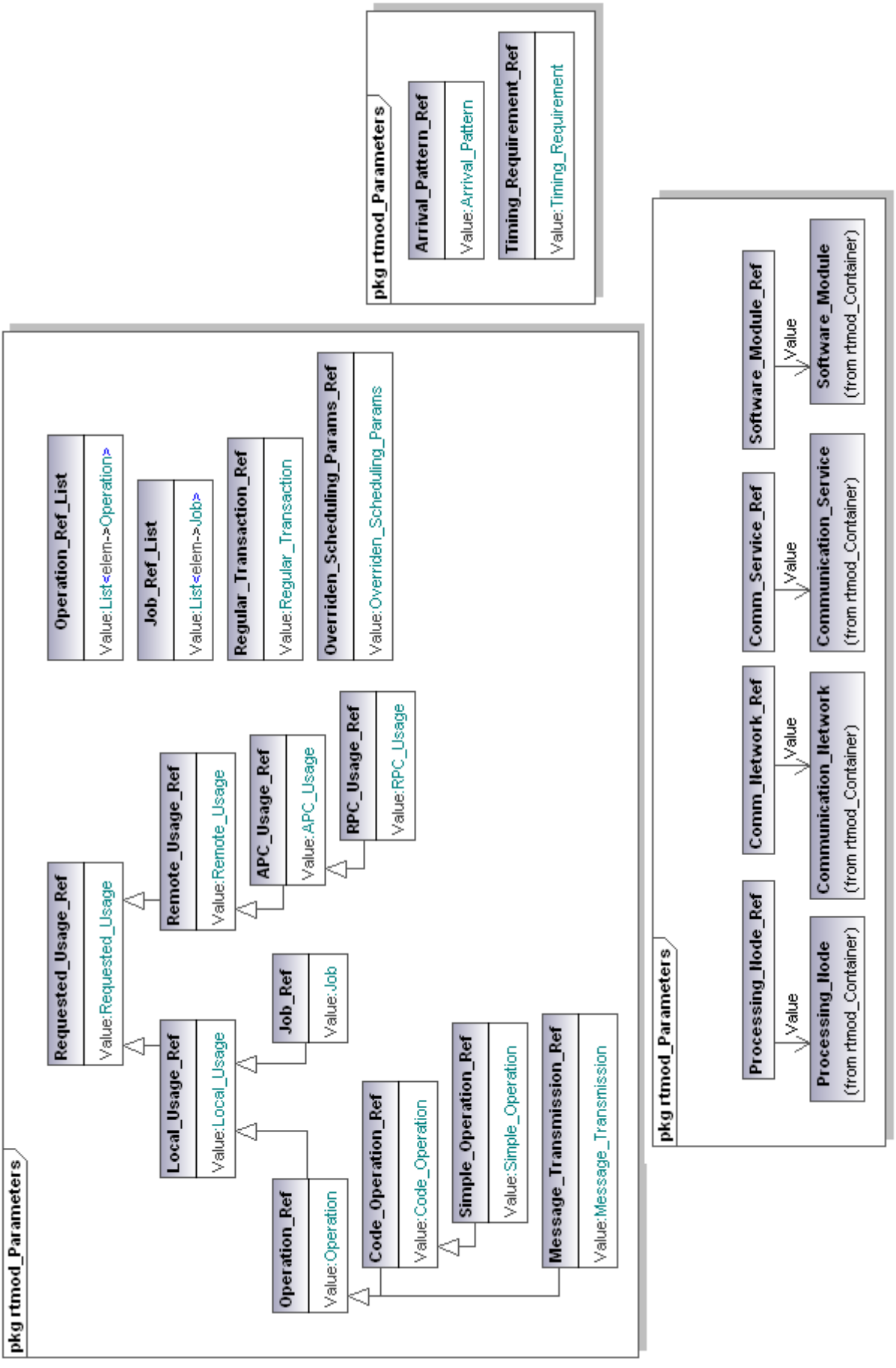

pkg rtmod_BasicTypes
«primitive» Integer
(from rtmod_PrimitiveTypes)
«primitive» Count
«primitive» Bit_Count
«primitive» Priority
«primitive» Preemption_Level
«primitive» Float
(from rtmod_PrimitiveTypes)
«primitive» Factor
«primitive» Percentage
«primitive» TimeStamp
«primitive» TimeInterval
«primitive» Throughput
«primitive» Normalized_Execution_Time
«primitive» String
(from rtmod_PrimitiveTypes)
«primitive» Identifier
«primitive» Date_Time
«enumeration» Laxity_Kind
HARD
SOFT
«enumeration» Delivery_Policy
SCAN
RANDOM
«enumeration» Request_Policy
PRIORITY
FIFO
SCAN
LIFO
«enumeration» Distribution_Type
UNIFORM
POISSON
«enumeration» Transmission_Type
SIMPLEX
HALF_DUPLEX
FULL_DUPLEX
«enumeration» Overhead_Type
COUPLED
DECOUPLED
«enumeration» Priority_Inheritance_Protocol
STRICT
POSIX
«enumeration» Assertion
YES
NO

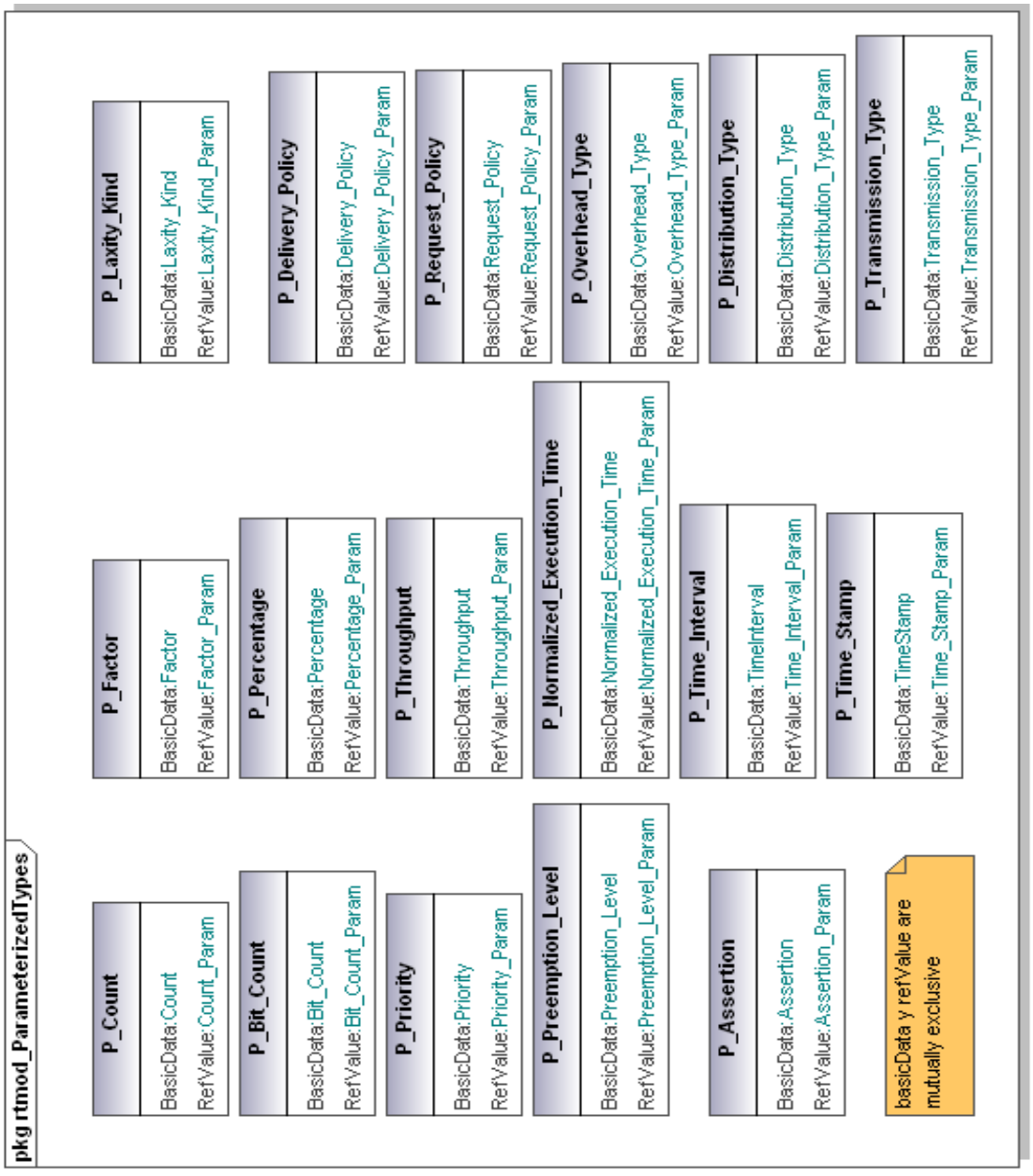

pkg rtmod_CoreTypes
Descriptor
Name:Identifier
Label:String[0..1]
Base:Descriptor[0..1]
Param_Ref:Param[0..1]
Instance
Name:Identifier
Label:String[0..1]
Instance_Of:Descriptor[0..1]
Param
(from rtmod_Parameters)
Resource_Instance
Assign_Param_List
Decl_Param_List
Aggr_Rsrc_List
Decl_Elem_List
Priv_Elem_List
*

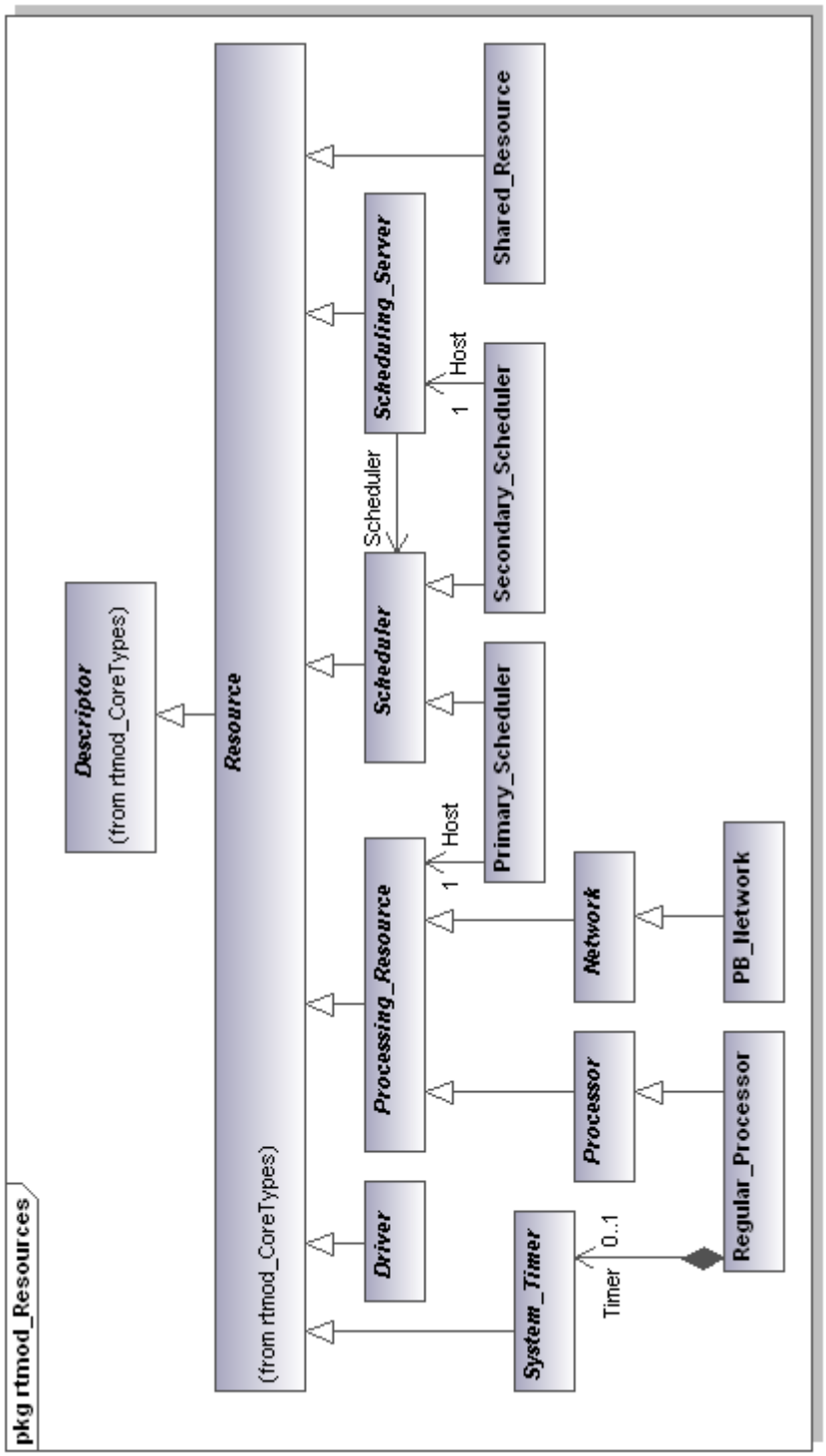

pkg rtmod_CoreTypes
Instance
Descriptor
Container_Instance
Container
Usage_Instance
Usage
Transaction_Instance
(from rtmod_Usages)
Aggr_Trans_List
*
Aggr_Trans_List
*
Resource_Instance
Resource
Data_Instance
Data_Struct

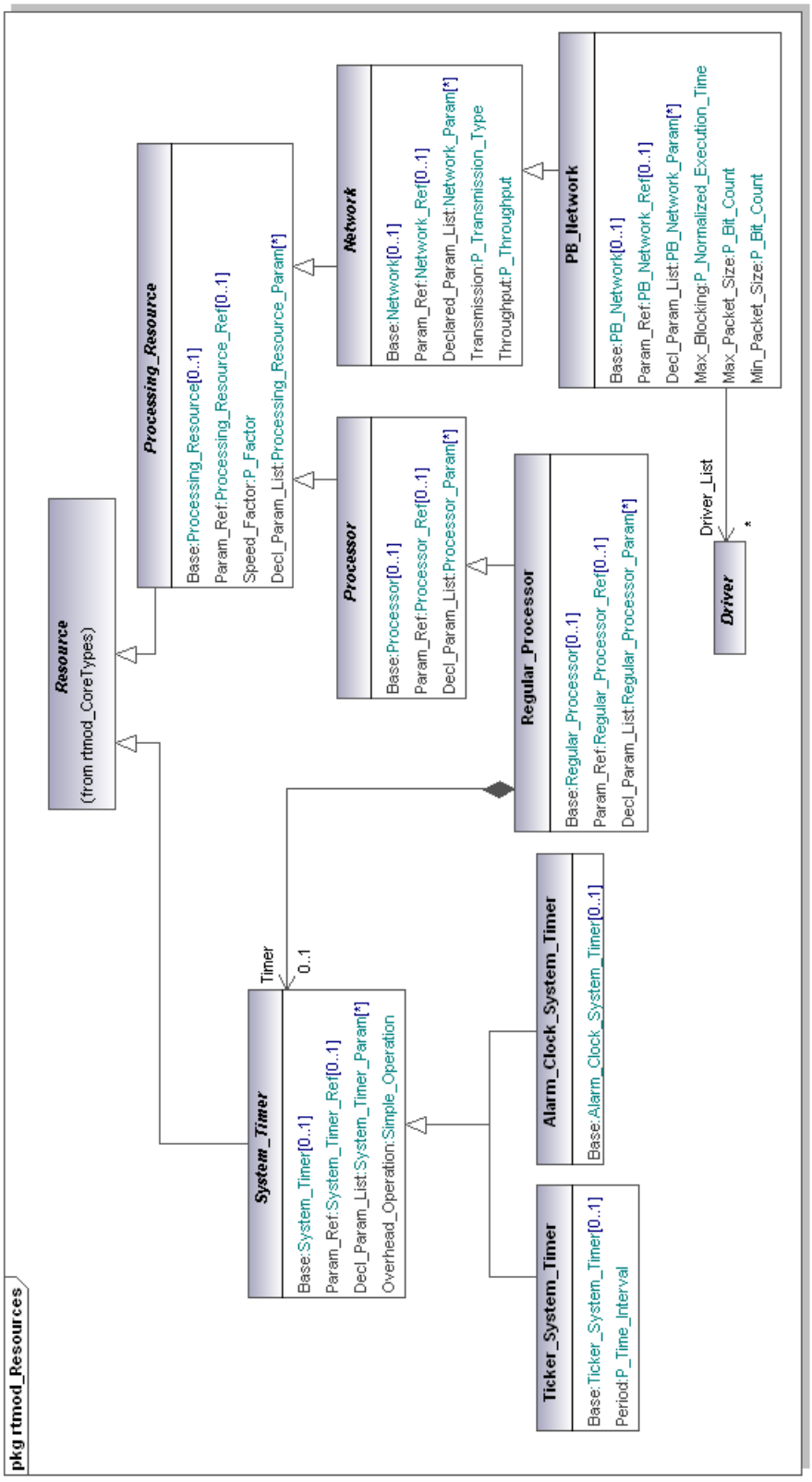

pkg rtmod_Parameters
Count_Param
Value:Count[0..1]
Bit_Count_Param
Value:Bit_Count[0..1]
Priority_Param
Value:Priority[0..1]
Preemption_Level_Param
Value:Preemption_Level[0..1]
Factor_Param
Value:Factor
Percentage_Param
Value:Percentage
Throughput_Param
Value:Throughput
Normalized_Execution_Time_Param
Value:Normalized_Execution_Time
Time_Interval_Param
Value:TimeInterval
Time_Stamp_Param
Value:TimeStamp
Assertion_Param
Value:Assertion
Laxity_Kind_Param
Value:Laxity_Kind
Delivery_Policy_Param
Value:Delivery_Policy
Request_Policy_Param
Value:Request_Policy
Distribution_Type_Param
Value:Distribution_Type
Transmission_Type_Param
Value:Transmission_Type
Overhead_Type_Param
Value:Overhead_Type

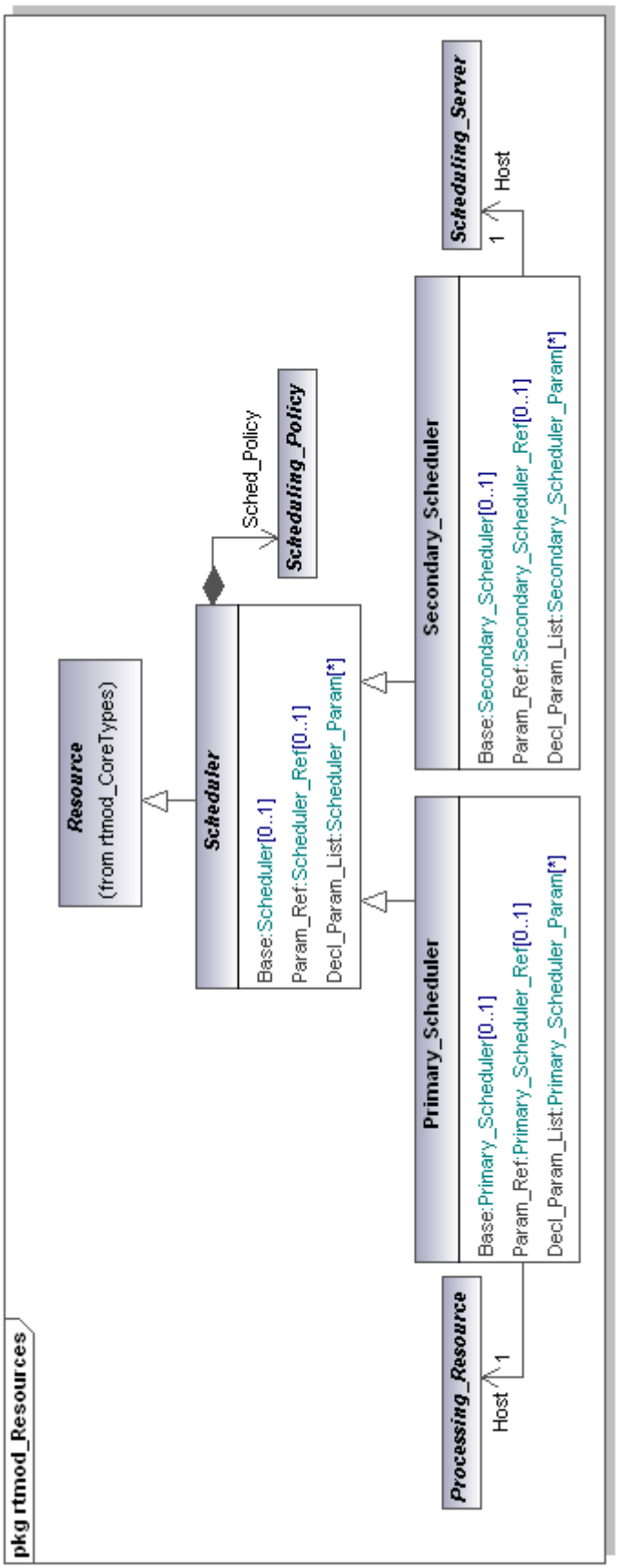

pkg rtmod_Parameters
Scheduler_Ref
Value:Scheduler
Primary_Scheduler_Ref
Value:Primary_Scheduler
Secondary_Scheduler_Ref
Value:Secondary_Scheduler
Scheduling_Policy_Ref
Value:Scheduling_Policy
Scheduling_Params_Ref
Value:Scheduling_Params
FP_Params_Ref
Value:FP_Params
Basic_FP_Params_Ref
Value:Basic_FP_Params
EDF_Params_Ref
Value:EDF_Params
Scheduling_Server_Ref
Value:Scheduling_Server
Processor_Scheduling_Server_Ref
Value:Processor_Scheduling_Server
Regular_Scheduling_Server_Ref
Value:Regular_Scheduling_Server
Interrupt_Scheduling_Server_Ref
Value:Interrupt_Scheduling_Server
Network_Scheduling_Server_Ref
Value:Network_Scheduling_Server
PB_Scheduling_Server_Ref
Value:PB_Scheduling_Server_Ref
Processing_Resource_Ref
Value:Processing_Resource
Processor_Ref
Value:Processor
Regular_Processor_Ref
Value:Regular_Processor
Network_Ref
Value:Network
PB_Network_Ref
Value:PB_Network
System_Timer_Ref
Value:System_Timer
Driver_Ref
Value:Driver
Driver_Ref_List
Value:List<elem->Driver>
Shared_Resource_Ref
Value:Shared_Resource
Shared_Resource_Ref_List
Value:List<elem->Shared_Resource>
Synchronization_Params_Ref
Value:Synchronization_Params
Sync_Params_Ref
Value:Sync_Params

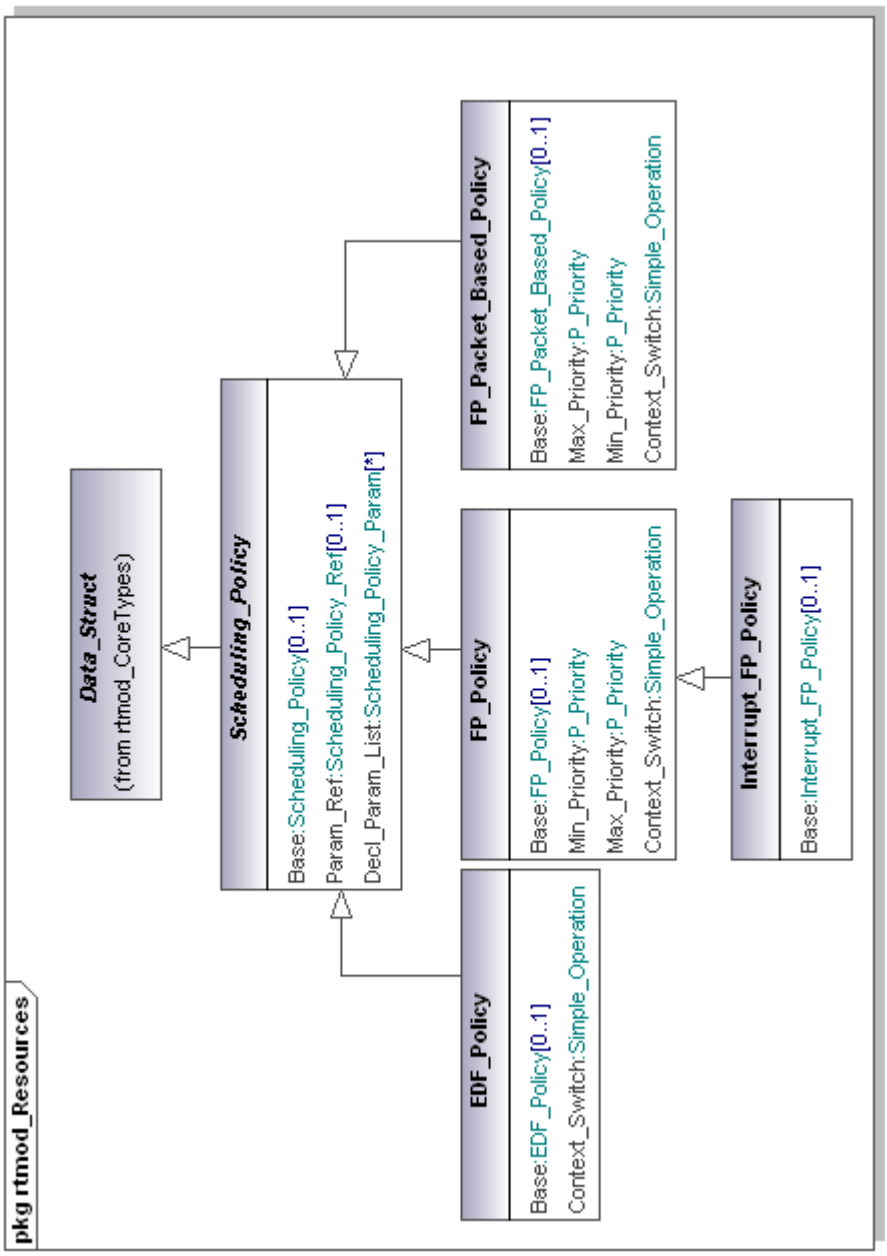

pkg rtmod_Parameters
Operation_Ref_List
Value:List<elem->Operation>
Job_Ref_List
Value:List<elem->Job>
Regular_Transaction_Ref
Value:Regular_Transaction
Overriden_Scheduling_Params_Ref
Value:Overriden_Scheduling_Params
Requested_Usage_Ref
Value:Requested_Usage
Remote_Usage_Ref
Value:Remote_Usage
APC_Usage_Ref
Value:APC_Usage
RPC_Usage_Ref
Value:RPC_Usage
Local_Usage_Ref
Value:Local_Usage
Job_Ref
Value:Job
Operation_Ref
Value:Operation
Code_Operation_Ref
Value:Code_Operation
Simple_Operation_Ref
Value:Simple_Operation
Message_Transmission_Ref
Value:Message_Transmission
pkg rtmod_Parameters
Arrival_Pattern_Ref
Value:Arrival_Pattern
Timing_Requirement_Ref
Value:Timing_Requirement
pkg rtmod_Parameters
Software_Module_Ref
Value
Software_Module
(from rtmod_Container)
Comm_Service_Ref
Value
Communication_Service
(from rtmod_Container)
Comm_Network_Ref
Value
Communication_Network
(from rtmod_Container)
Processing_Node_Ref
Value
Processing_Node
(from rtmod_Container)

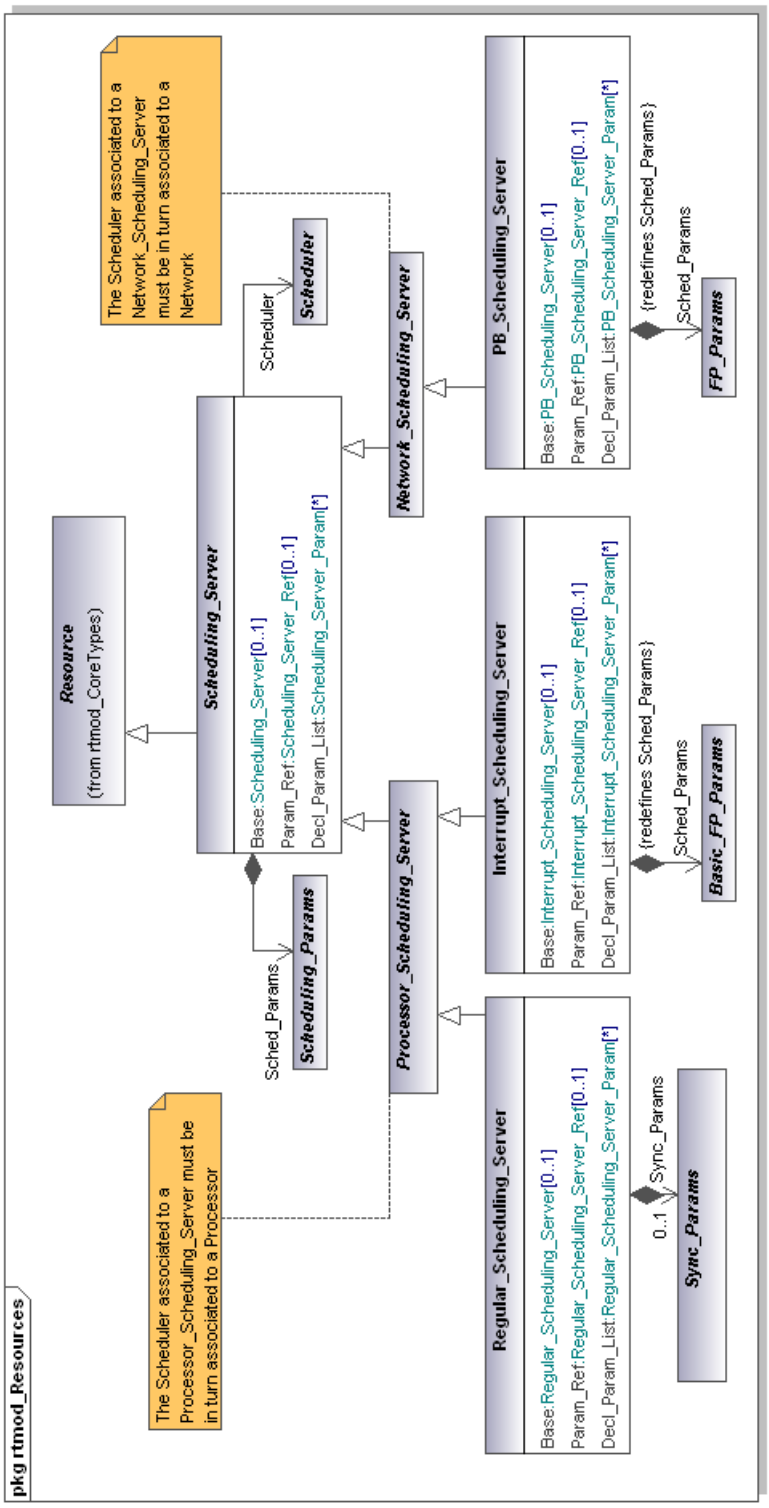

pkg rtmod_ParameterizedTypes
P_Count
BasicData:Count
RefValue:Count_Param
P_Factor
BasicData:Factor
RefValue:Factor_Param
P_Laxity_Kind
BasicData:Laxity_Kind
RefValue:Laxity_Kind_Param
P_Bit_Count
BasicData:Bit_Count
RefValue:Bit_Count_Param
P_Percentage
BasicData:Percentage
RefValue:Percentage_Param
P_Delivery_Policy
BasicData:Delivery_Policy
RefValue:Delivery_Policy_Param
P_Priority
BasicData:Priority
RefValue:Priority_Param
P_Throughput
BasicData:Throughput
RefValue:Throughput_Param
P_Request_Policy
BasicData:Request_Policy
RefValue:Request_Policy_Param
P_Preemption_Level
BasicData:Preemption_Level
RefValue:Preemption_Level_Param
P_Normalized_Execution_Time
BasicData:Normalized_Execution_Time
RefValue:Normalized_Execution_Time_Param
P_Overhead_Type
BasicData:Overhead_Type
RefValue:Overhead_Type_Param
P_Assertion
BasicData:Assertion
RefValue:Assertion_Param
P_Time_Interval
BasicData:TimeInterval
RefValue:Time_Interval_Param
P_Distribution_Type
BasicData:Distribution_Type
RefValue:Distribution_Type_Param
P_Time_Stamp
BasicData:TimeStamp
RefValue:Time_Stamp_Param
P_Transmission_Type
BasicData:Transmission_Type
RefValue:Transmission_Type_Param
basicData y refValue are mutually exclusive

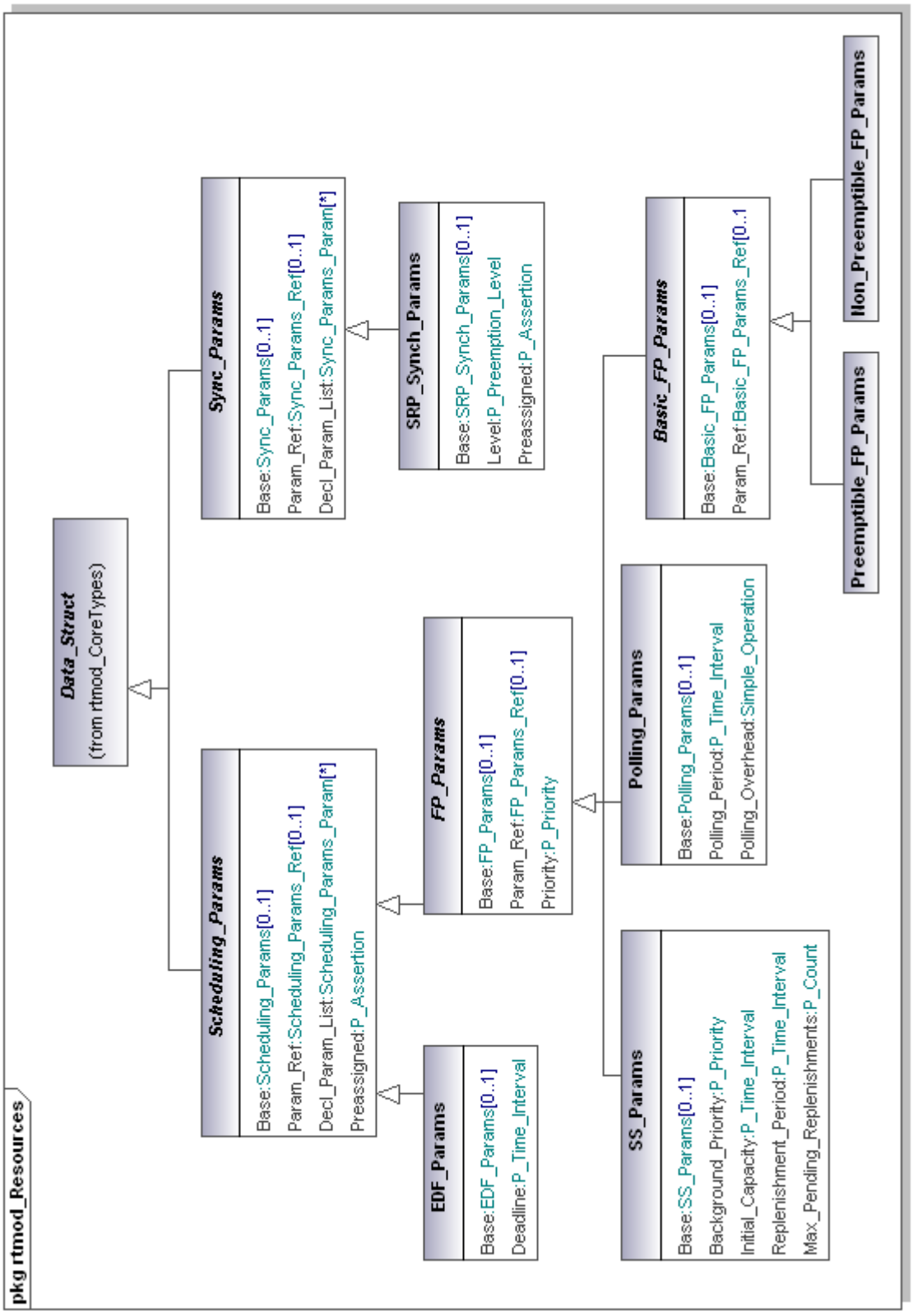

pkg rtmod_Resources
Descriptor
(from rtmod_CoreTypes)
Resource
(from rtmod_CoreTypes)
Shared_Resource
Scheduling_Server
Scheduler
Secondary_Scheduler
Primary_Scheduler
Host
1
Scheduler
Processing_Resource
Network
PB_Network
Processor
Regular_Processor
Driver
System_Timer
Timer
0..1

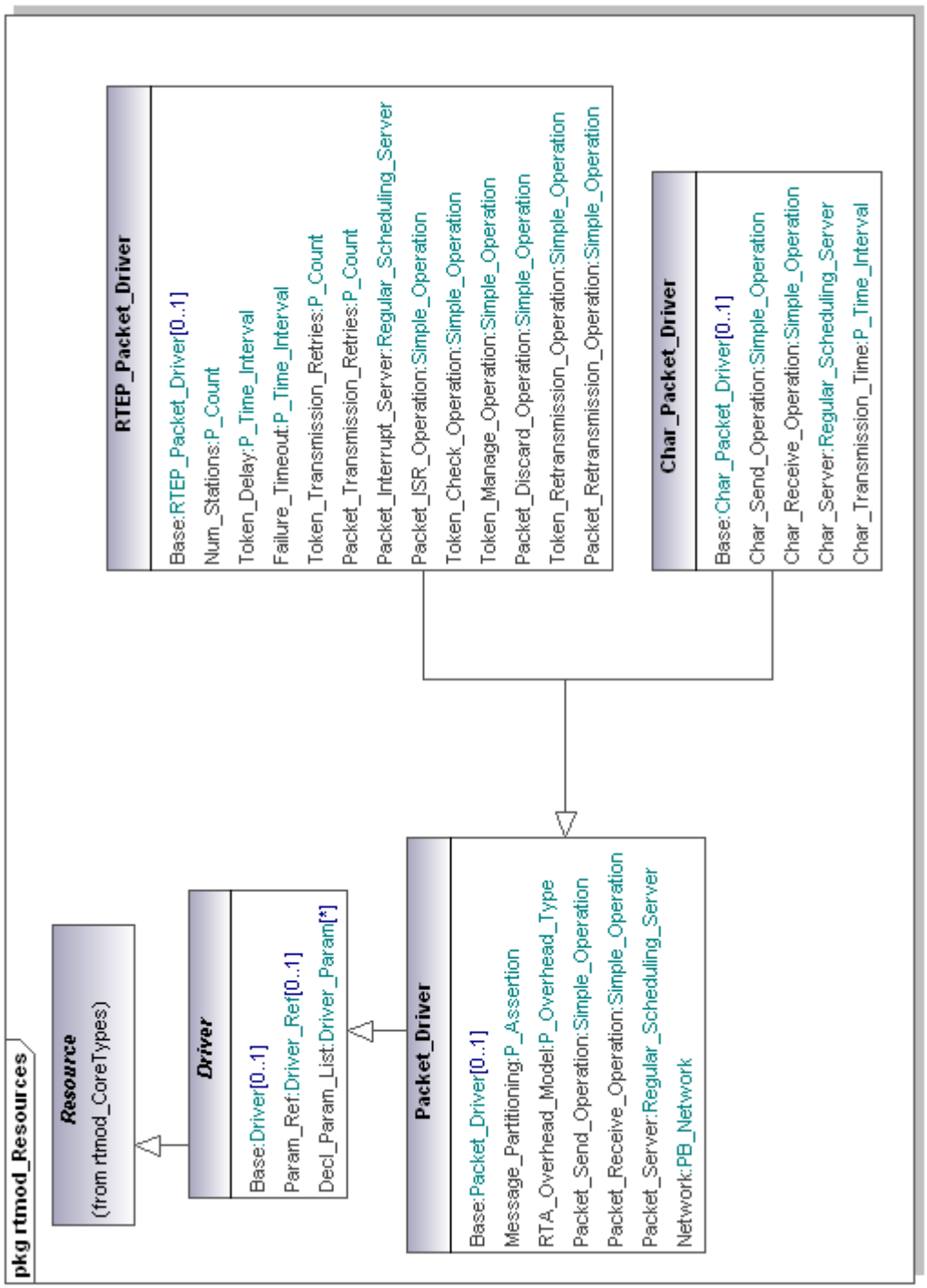

pkg rtmod_Resources
Resource
(from rtmod_CoreTypes)
Processing_Resource
Base:Processing_Resource[0..1]
Param_Ref:Processing_Resource_Ref[0..1]
Speed_Factor:P_Factor
Decl_Param_List:Processing_Resource_Param[*]
Network
Base:Network[0..1]
Param_Ref:Network_Ref[0..1]
Declared_Param_List:Network_Param[*]
Transmission:P_Transmission_Type
Throughput:P_Throughput
PB_Network
Base:PB_Network[0..1]
Param_Ref:PB_Network_Ref[0..1]
Decl_Param_List:PB_Network_Param[*]
Max_Blocking:P_Normalized_Execution_Time
Max_Packet_Size:P_Bit_Count
Min_Packet_Size:P_Bit_Count
Processor
Base:Processor[0..1]
Param_Ref:Processor_Ref[0..1]
Decl_Param_List:Processor_Param[*]
Regular_Processor
Base:Regular_Processor[0..1]
Param_Ref:Regular_Processor_Ref[0..1]
Decl_Param_List:Regular_Processor_Param[*]
Driver_List
*
Driver
Timer
0..1
System_Timer
Base:System_Timer[0..1]
Param_Ref:System_Timer_Ref[0..1]
Decl_Param_List:System_Timer_Param[*]
Overhead_Operation:Simple_Operation
Alarm_Clock_System_Timer
Base:Alarm_Clock_System_Timer[0..1]
Ticker_System_Timer
Base:Ticker_System_Timer[0..1]
Period:P_Time_Interval

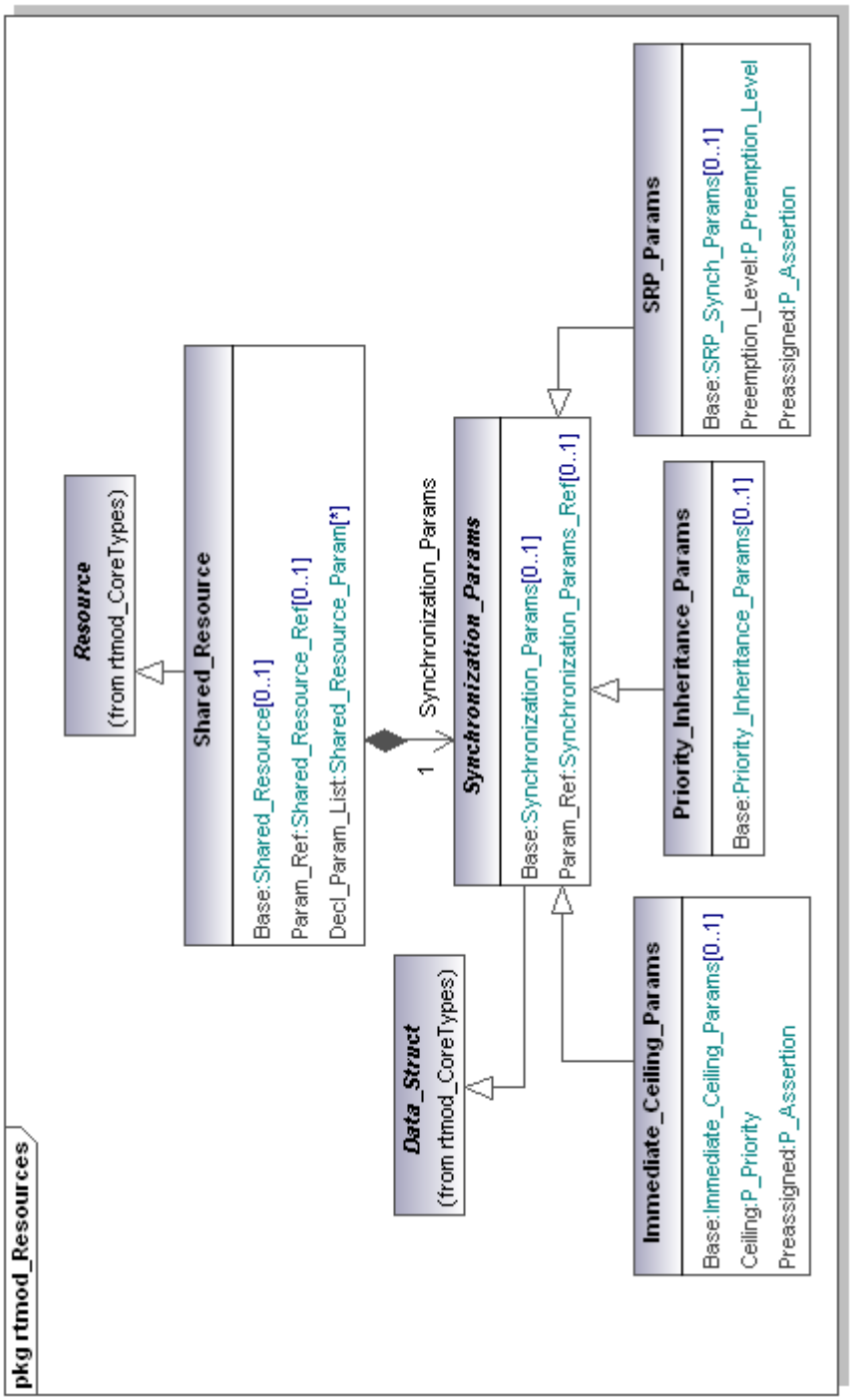

pkg rtmod_Resources
Resource
(from rtmod_CoreTypes)
Scheduler
Base:Scheduler[0..1]
Param_Ref:Scheduler_Ref[0..1]
Decl_Param_List:Scheduler_Param[*]
Sched_Policy
Scheduling_Policy
Primary_Scheduler
Base:Primary_Scheduler[0..1]
Param_Ref:Primary_Scheduler_Ref[0..1]
Decl_Param_List:Primary_Scheduler_Param[*]
Secondary_Scheduler
Base:Secondary_Scheduler[0..1]
Param_Ref:Secondary_Scheduler_Ref[0..1]
Decl_Param_List:Secondary_Scheduler_Param[*]
Processing_Resource
Host
1
Scheduling_Server
Host
1

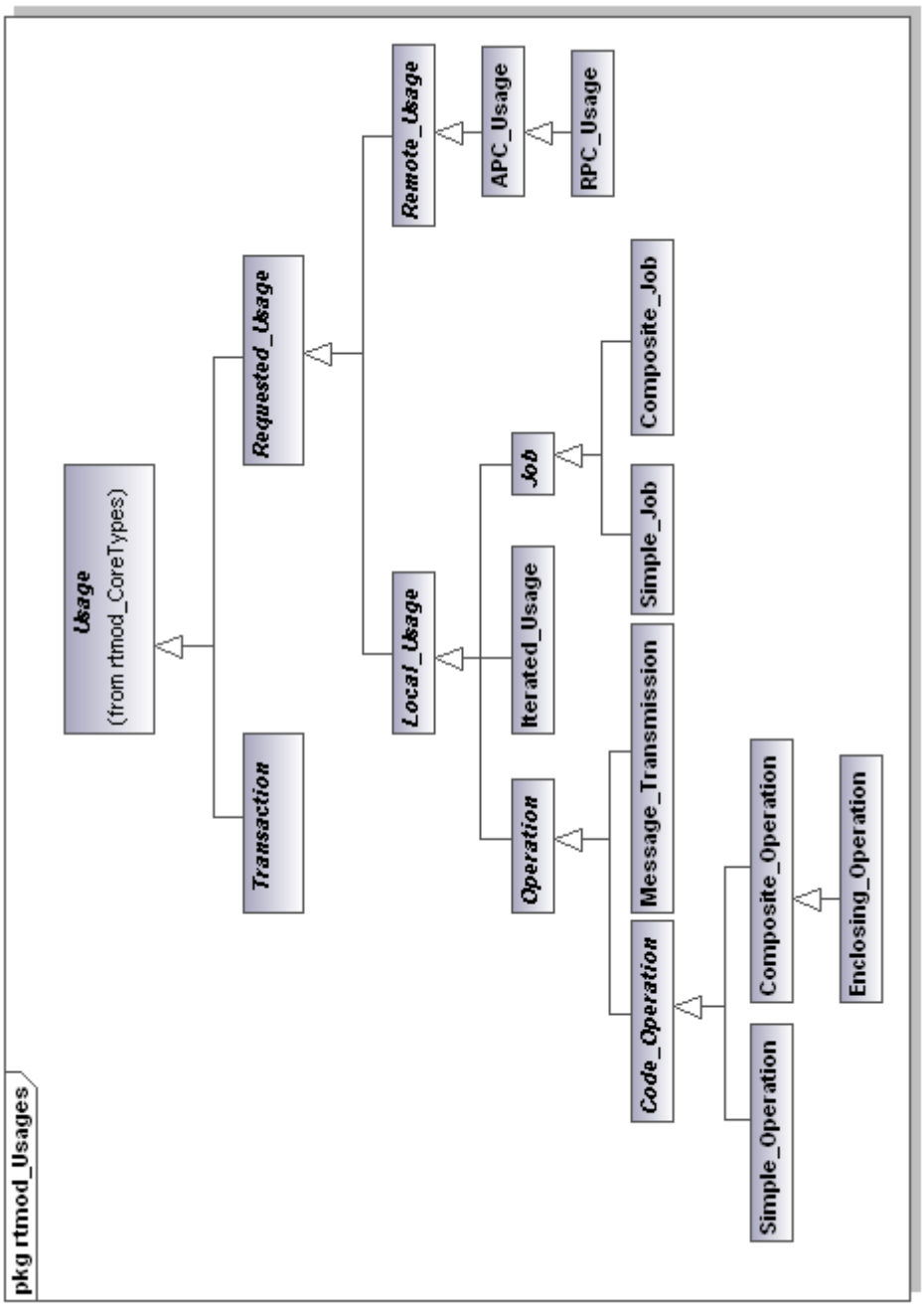

pkg rtmod_Resources
Data_Struct
(from rtmod_CoreTypes)
Scheduling_Policy
Base:Scheduling_Policy[0..1]
Param_Ref:Scheduling_Policy_Ref[0..1]
Decl_Param_List:Scheduling_Policy_Param[*]
FP_Packet_Based_Policy
Base:FP_Packet_Based_Policy[0..1]
Max_Priority:P_Priority
Min_Priority:P_Priority
Context_Switch:Simple_Operation
FP_Policy
Base:FP_Policy[0..1]
Min_Priority:P_Priority
Max_Priority:P_Priority
Context_Switch:Simple_Operation
EDF_Policy
Base:EDF_Policy[0..1]
Context_Switch:Simple_Operation
Interrupt_FP_Policy
Base:Interrupt_FP_Policy[0..1]

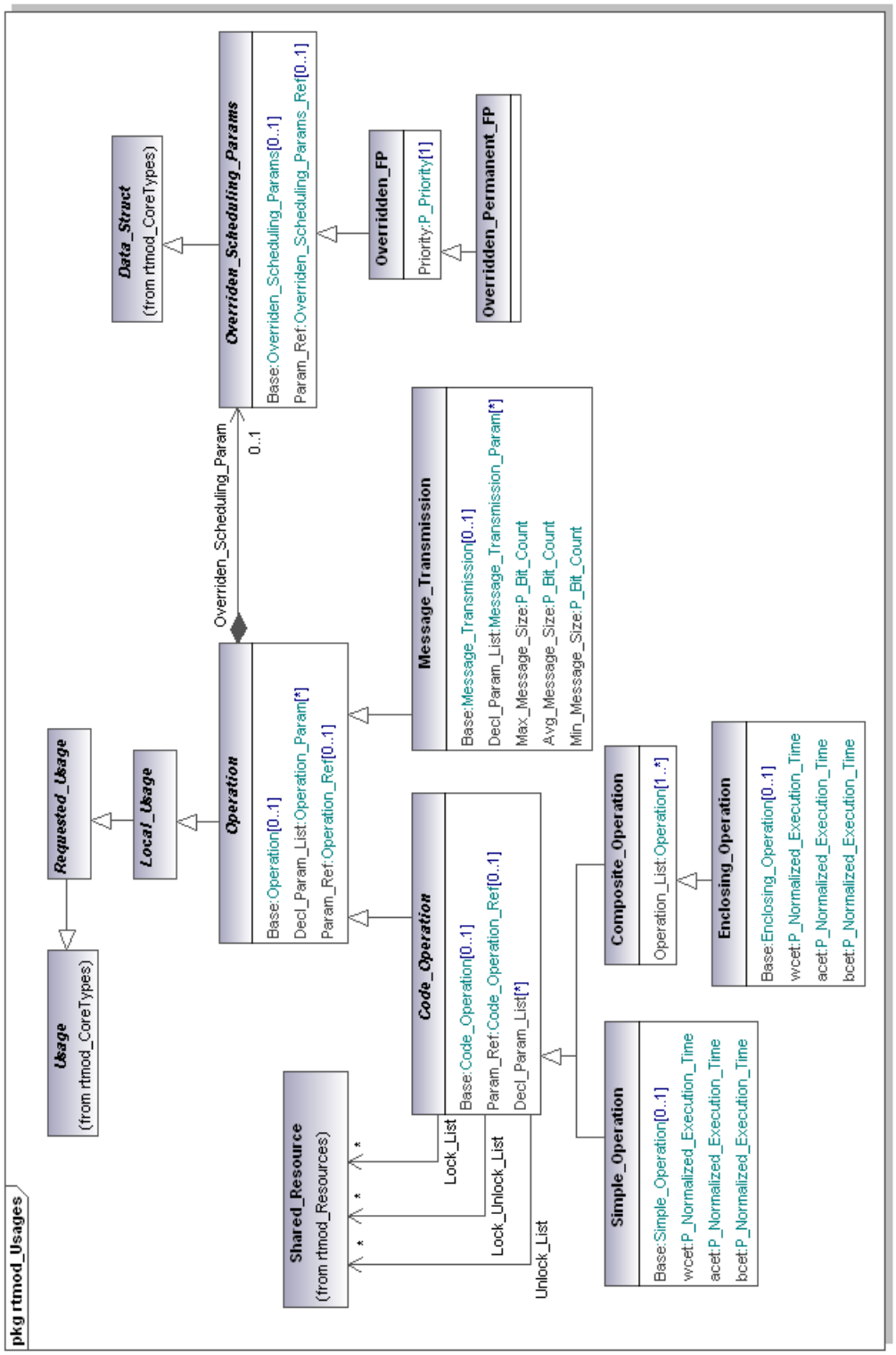

pkg rtmod_Resources
Resource
(from rtmod_CoreTypes)
Scheduling_Server
Base:Scheduling_Server[0..1]
Param_Ref:Scheduling_Server_Ref[0..1]
Decl_Param_List:Scheduling_Server_Param[*]
Scheduler
Scheduler
The Scheduler associated to a Network_Scheduling_Server must be in turn associated to a Network
Network_Scheduling_Server
PB_Scheduling_Server
Base:PB_Scheduling_Server[0..1]
Param_Ref:PB_Scheduling_Server_Ref[0..1]
Decl_Param_List:PB_Scheduling_Server_Param[*]
{redefines Sched_Params}
Sched_Params
FP_Params
Sched_Params
Scheduling_Params
Processor_Scheduling_Server
Interrupt_Scheduling_Server
Base:Interrupt_Scheduling_Server[0..1]
Param_Ref:Interrupt_Scheduling_Server_Ref[0..1]
Decl_Param_List:Interrupt_Scheduling_Server_Param[*]
{redefines Sched_Params}
Sched_Params
Basic_FP_Params
The Scheduler associated to a Processor_Scheduling_Server must be in turn associated to a Processor
Regular_Scheduling_Server
Base:Regular_Scheduling_Server[0..1]
Param_Ref:Regular_Scheduling_Server_Ref[0..1]
Decl_Param_List:Regular_Scheduling_Server_Param[*]
0..1
Sync_Params
Sync_Params

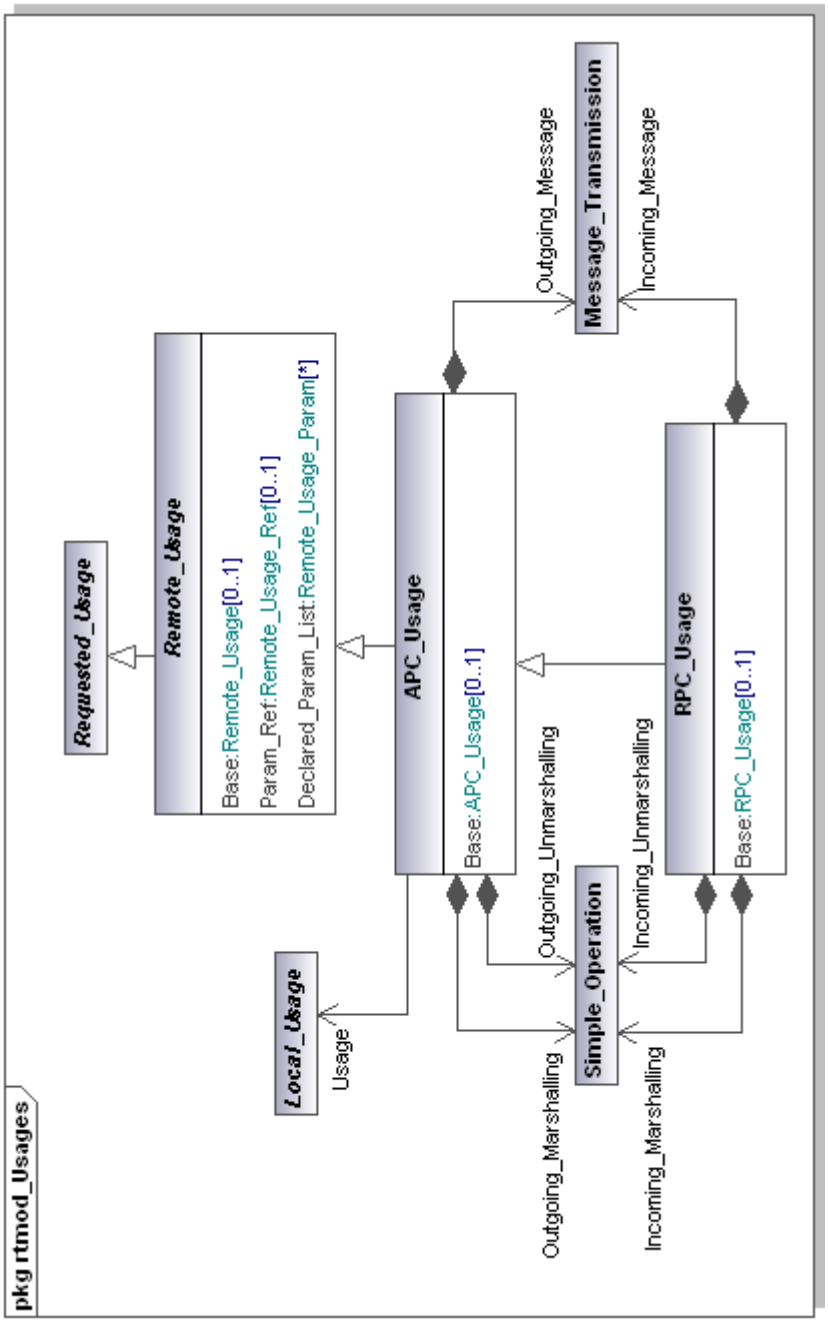

pkg rtmod_Resources
Data_Struct
(from rtmod_CoreTypes)
Sync_Params
Base:Sync_Params[0..1]
Param_Ref:Sync_Params_Ref[0..1]
Decl_Param_List:Sync_Params_Param[*]
SRP_Synch_Params
Base:SRP_Synch_Params[0..1]
Level:P_Preemption_Level
Preassigned:P_Assertion
Scheduling_Params
Base:Scheduling_Params[0..1]
Param_Ref:Scheduling_Params_Ref[0..1]
Decl_Param_List:Scheduling_Params_Param[*]
Preassigned:P_Assertion
FP_Params
Base:FP_Params[0..1]
Param_Ref:FP_Params_Ref[0..1]
Priority:P_Priority
EDF_Params
Base:EDF_Params[0..1]
Deadline:P_Time_Interval
Polling_Params
Base:Polling_Params[0..1]
Polling_Period:P_Time_Interval
Polling_Overhead:Simple_Operation
SS_Params
Base:SS_Params[0..1]
Background_Priority:P_Priority
Initial_Capacity:P_Time_Interval
Replenishment_Period:P_Time_Interval
Max_Pending_Replenishments:P_Count
Basic_FP_Params
Base:Basic_FP_Params[0..1]
Param_Ref:Basic_FP_Params_Ref[0..1
Preemptible_FP_Params
Non_Preemptible_FP_Params

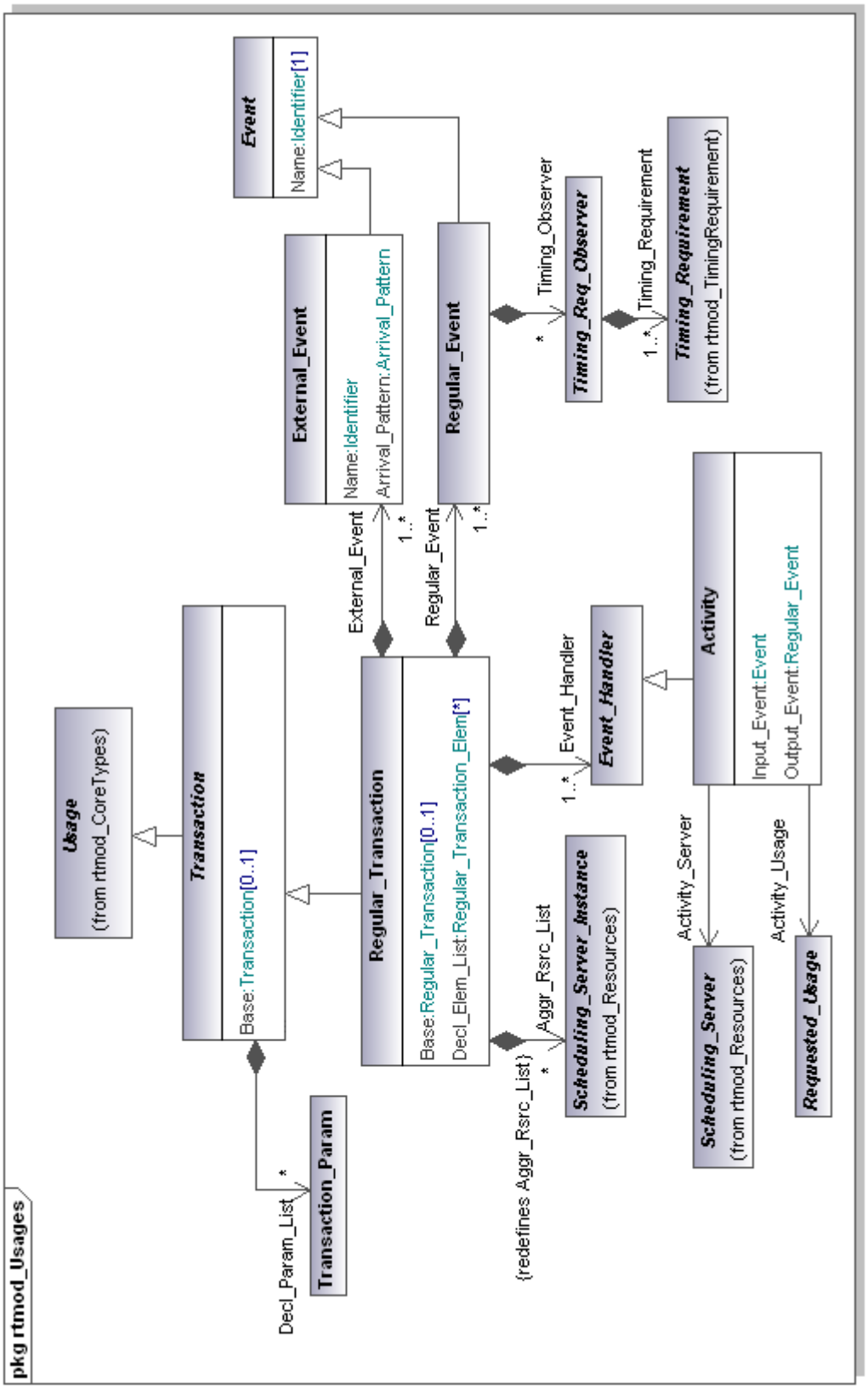

pkg rtmod_Resources
Resource
(from rtmod_CoreTypes)
Driver
Base:Driver[0..1]
Param_Ref:Driver_Ref[0..1]
Decl_Param_List:Driver_Param[*]
Packet_Driver
Base:Packet_Driver[0..1]
Message_Partitioning:P_Assertion
RTA_Overhead_Model:P_Overhead_Type
Packet_Send_Operation:Simple_Operation
Packet_Receive_Operation:Simple_Operation
Packet_Server:Regular_Scheduling_Server
Network:PB_Network
RTEP_Packet_Driver
Base:RTEP_Packet_Driver[0..1]
Num_Stations:P_Count
Token_Delay:P_Time_Interval
Failure_Timeout:P_Time_Interval
Token_Transmission_Retries:P_Count
Packet_Transmission_Retries:P_Count
Packet_Interrupt_Server:Regular_Scheduling_Server
Packet_ISR_Operation:Simple_Operation
Token_Check_Operation:Simple_Operation
Token_Manage_Operation:Simple_Operation
Packet_Discard_Operation:Simple_Operation
Token_Retransmission_Operation:Simple_Operation
Packet_Retransmission_Operation:Simple_Operation
Char_Packet_Driver
Base:Char_Packet_Driver[0..1]
Char_Send_Operation:Simple_Operation
Char_Receive_Operation:Simple_Operation
Char_Server:Regular_Scheduling_Server
Char_Transmission_Time:P_Time_Interval

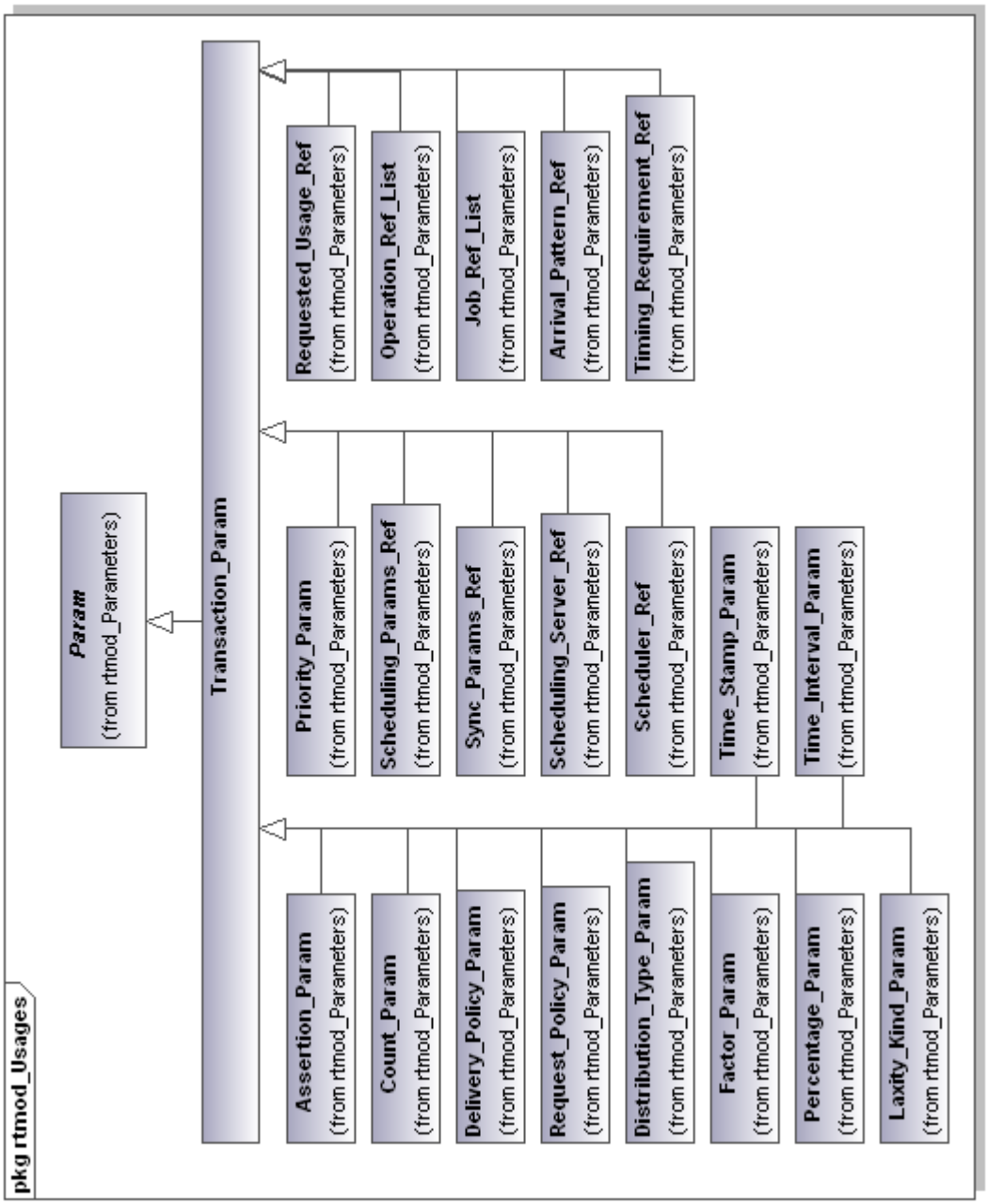

pkg rtmod_Resources
Resource
(from rtmod_CoreTypes)
Shared_Resource
Base:Shared_Resource[0..1]
Param_Ref:Shared_Resource_Ref[0..1]
Decl_Param_List:Shared_Resource_Param[*]
Synchronization_Params
1
Synchronization_Params
Base:Synchronization_Params[0..1]
Param_Ref:Synchronization_Params_Ref[0..1]
Data_Struct
(from rtmod_CoreTypes)
SRP_Params
Base:SRP_Synch_Params[0..1]
Preemption_Level:P_Preemption_Level
Preassigned:P_Assertion
Priority_Inheritance_Params
Base:Priority_Inheritance_Params[0..1]
Immediate_Ceiling_Params
Base:Immediate_Ceiling_Params[0..1]
Ceiling:P_Priority
Preassigned:P_Assertion

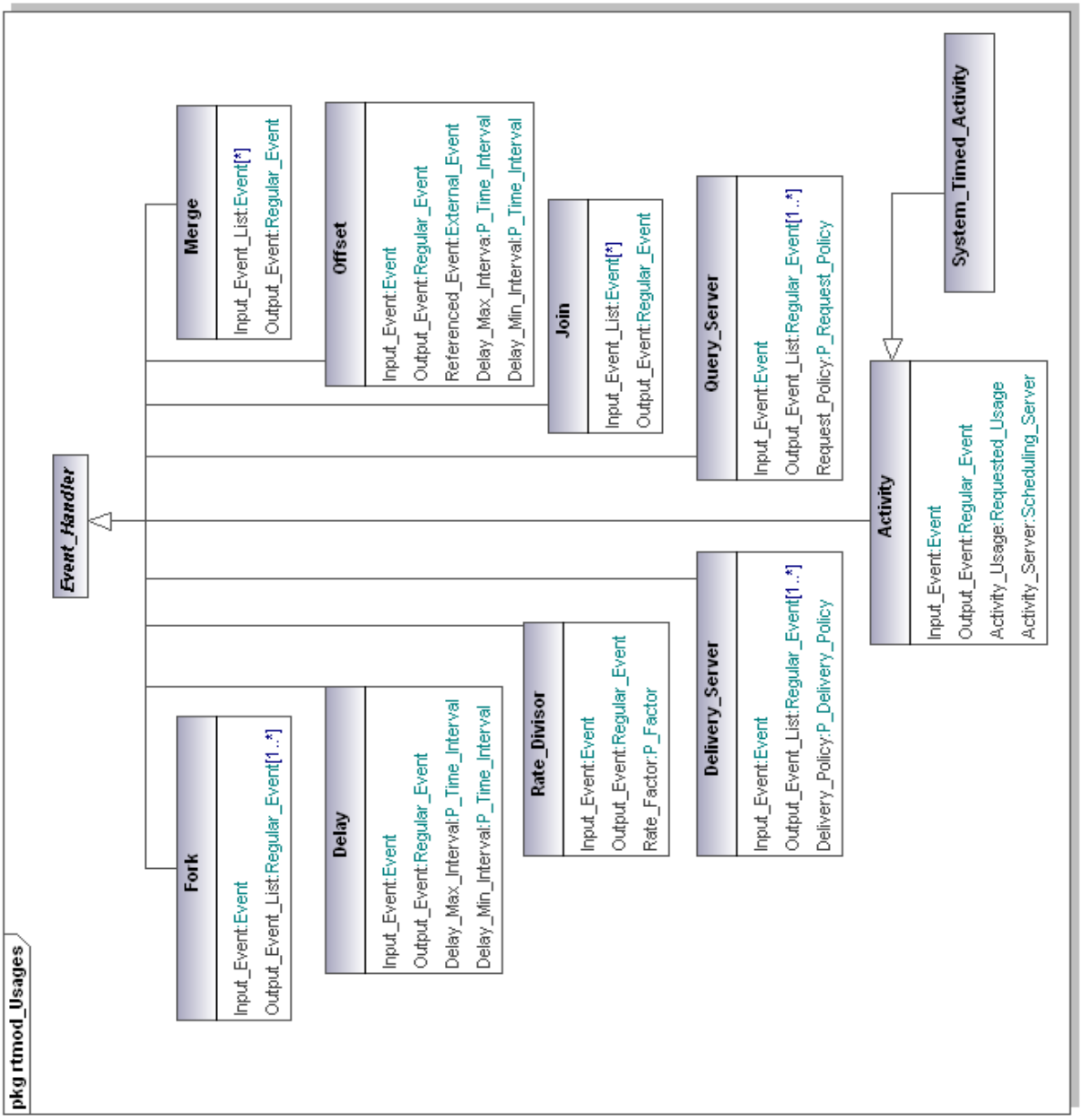

pkg rtmod_Usages
Usage
(from rtmod_CoreTypes)
Requested_Usage
Transaction
Remote_Usage
APC_Usage
RPC_Usage
Local_Usage
Job
Iterated_Usage
Composite_Job
Simple_Job
Operation
Message_Transmission
Code_Operation
Composite_Operation
Enclosing_Operation
Simple_Operation

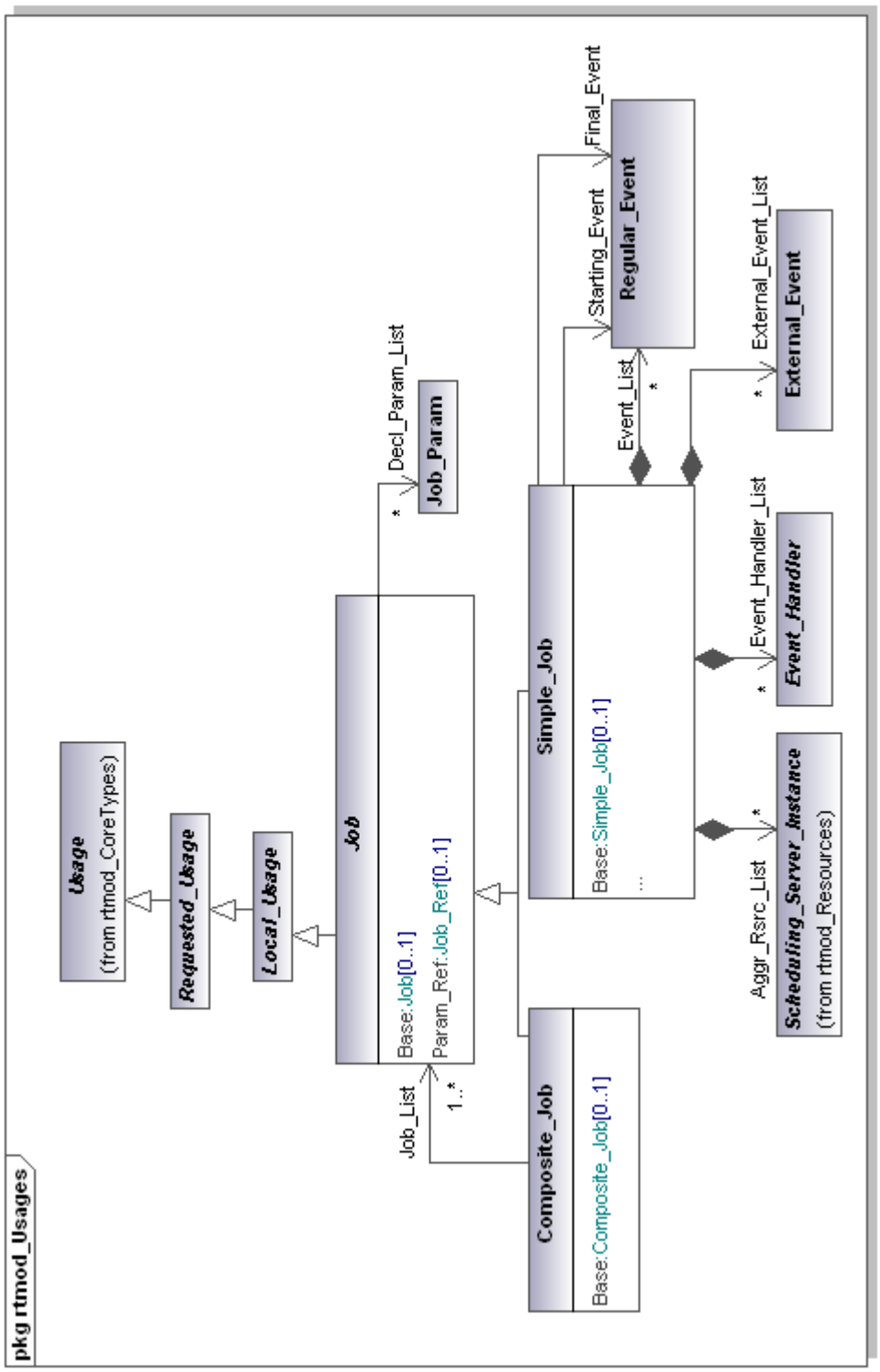

pkg rtmod_Usages
Usage
(from rtmod_CoreTypes)
Requested_Usage
Local_Usage
Operation
Base:Operation[0..1]
Decl_Param_List:Operation_Param[*]
Param_Ref:Operation_Ref[0..1]
Overriden_Scheduling_Param
0..1
Data_Struct
(from rtmod_CoreTypes)
Overriden_Scheduling_Params
Base:Overriden_Scheduling_Params[0..1]
Param_Ref:Overriden_Scheduling_Params_Ref[0..1]
Overridden_FP
Priority:P_Priority[1]
Overridden_Permanent_FP
Message_Transmission
Base:Message_Transmission[0..1]
Decl_Param_List:Message_Transmission_Param[*]
Max_Message_Size:P_Bit_Count
Avg_Message_Size:P_Bit_Count
Min_Message_Size:P_Bit_Count
Code_Operation
Base:Code_Operation[0..1]
Param_Ref:Code_Operation_Ref[0..1]
Decl_Param_List[*]
Shared_Resource
(from rtmod_Resources)
Lock_List
Lock_Unlock_List
Unlock_List
Composite_Operation
Operation_List:Operation[1..*]
Simple_Operation
Base:Simple_Operation[0..1]
wcet:P_Normalized_Execution_Time
acet:P_Normalized_Execution_Time
bcet:P_Normalized_Execution_Time
Enclosing_Operation
Base:Enclosing_Operation[0..1]
wcet:P_Normalized_Execution_Time
acet:P_Normalized_Execution_Time
bcet:P_Normalized_Execution_Time

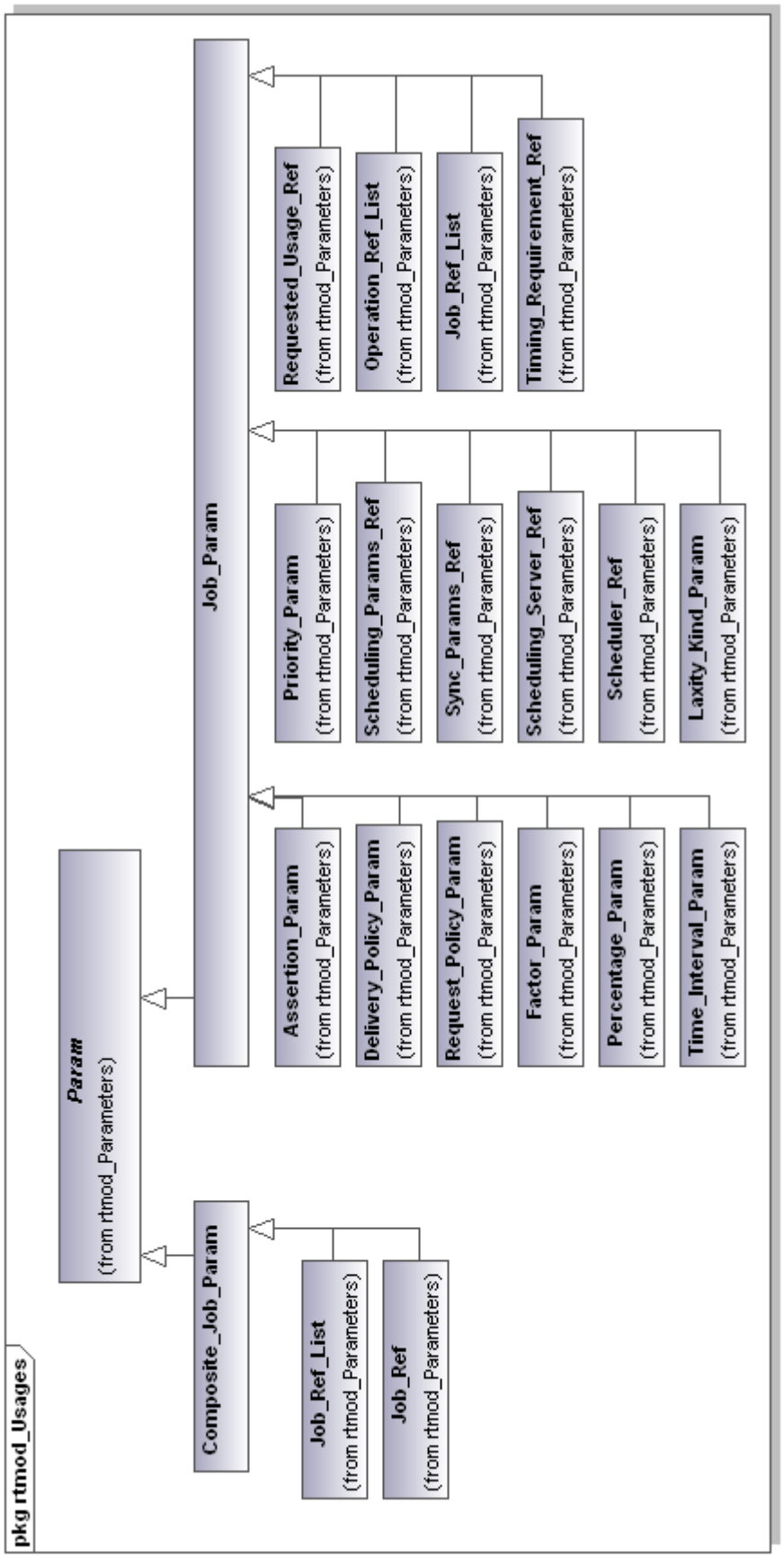

pkg rtmod_Usages
Requested_Usage
Remote_Usage
Base:Remote_Usage[0..1]
Param_Ref:Remote_Usage_Ref[0..1]
Declared_Param_List:Remote_Usage_Param[*]
APC_Usage
Base:APC_Usage[0..1]
RPC_Usage
Base:RPC_Usage[0..1]
Message_Transmission
Outgoing_Message
Incoming_Message
Local_Usage
Usage
Simple_Operation
Outgoing_Unmarshalling
Incoming_Unmarshalling
Outgoing_Marshalling
Incoming_Marshalling

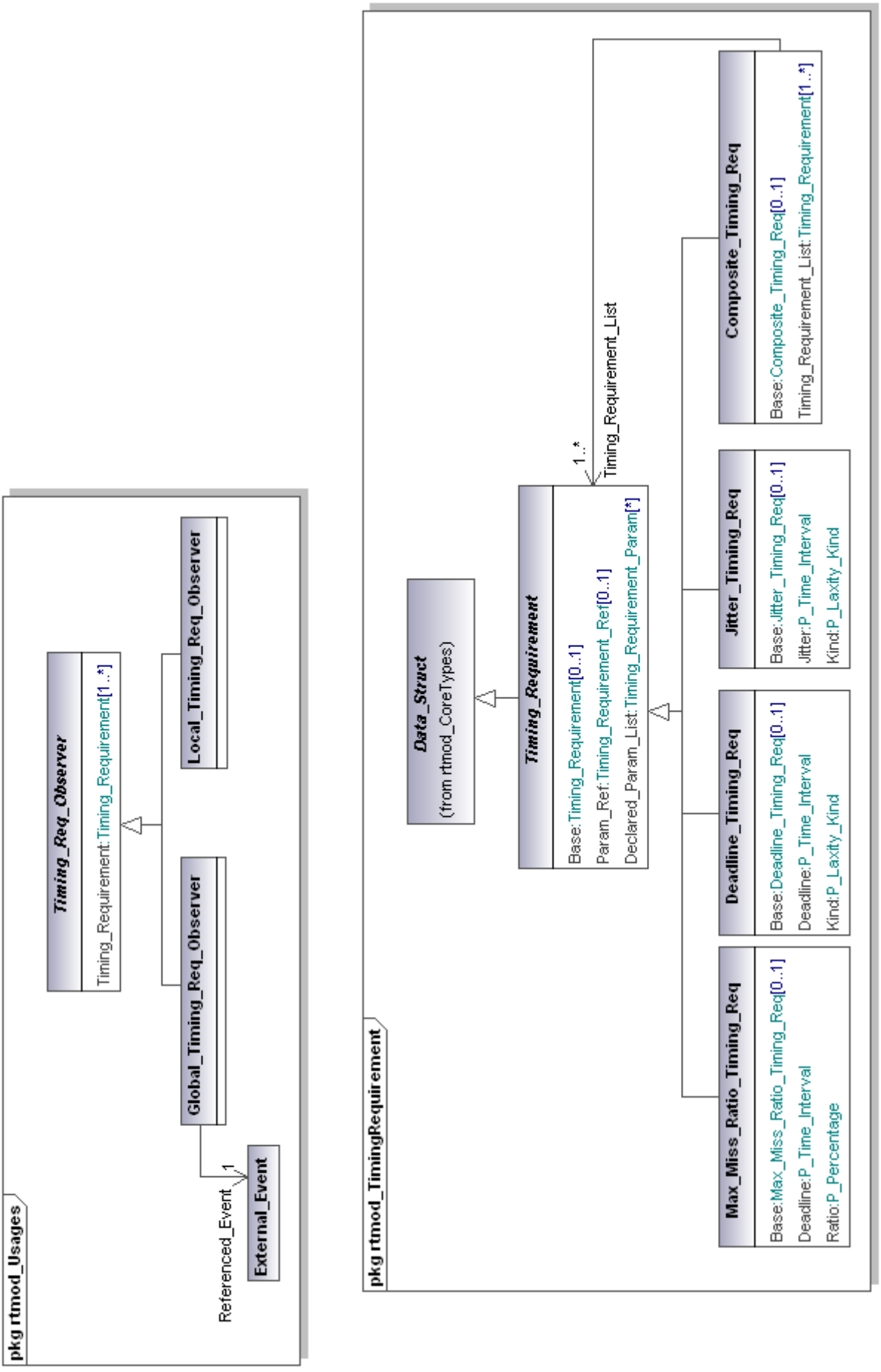

pkg rtmod_Usages
Usage
(from rtmod_CoreTypes)
Transaction
Base:Transaction[0..1]
Regular_Transaction
Base:Regular_Transaction[0..1]
Decl_Elem_List:Regular_Transaction_Elem[*]
Event
Name:Identifier[1]
External_Event
Name:Identifier
Arrival_Pattern:Arrival_Pattern
Regular_Event
Timing_Observer
Timing_Req_Observer
Timing_Requirement
Timing_Requirement
(from rtmod_TimingRequirement)
External_Event
Regular_Event
Event_Handler
Event_Handler
Activity
Input_Event:Event
Output_Event:Regular_Event
Decl_Param_List
Transaction_Param
{redefines Aggr_Rsrc_List}
Aggr_Rsrc_List
Scheduling_Server_Instance
(from rtmod_Resources)
Activity_Server
Activity_Usage
Scheduling_Server
(from rtmod_Resources)
Requested_Usage

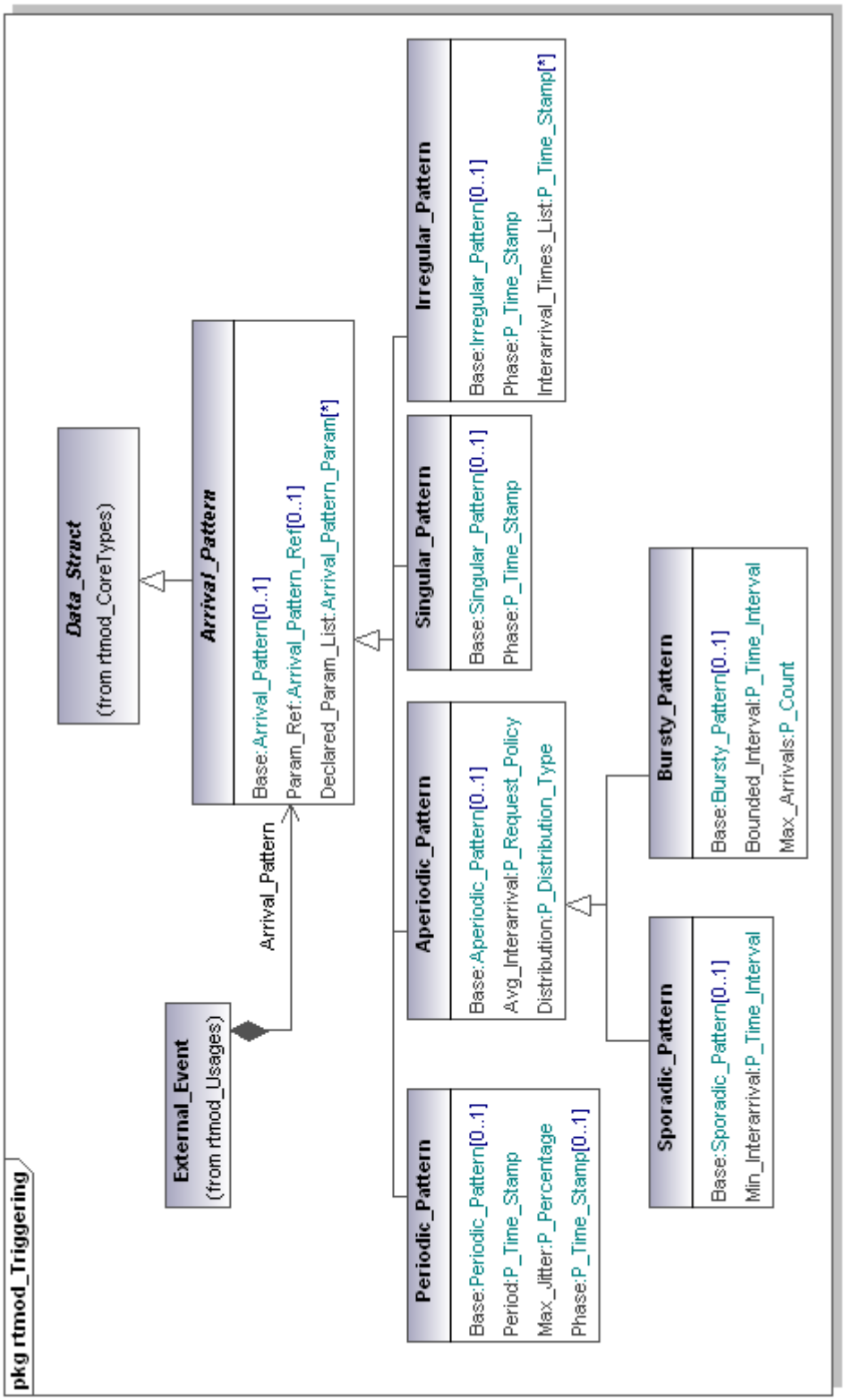

pkg rtmod_Usages
Param
(from rtmod_Parameters)
Transaction_Param
Requested_Usage_Ref
(from rtmod_Parameters)
Operation_Ref_List
(from rtmod_Parameters)
Job_Ref_List
(from rtmod_Parameters)
Arrival_Pattern_Ref
(from rtmod_Parameters)
Timing_Requirement_Ref
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Scheduling_Params_Ref
(from rtmod_Parameters)
Sync_Params_Ref
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Time_Stamp_Param
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Assertion_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)
Distribution_Type_Param
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Laxity_Kind_Param
(from rtmod_Parameters)

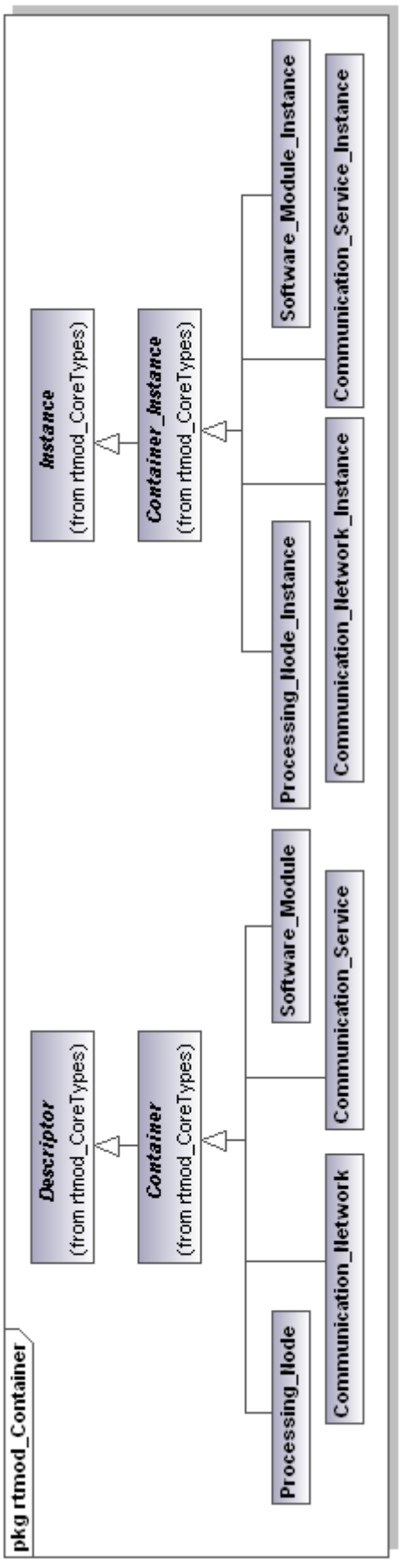

pkg rtmod_Usages
Event_Handler
Fork
Input_Event:Event
Output_Event_List:Regular_Event[1..*]
Merge
Input_Event_List:Event[*]
Output_Event:Regular_Event
Delay
Input_Event:Event
Output_Event:Regular_Event
Delay_Max_Interval:P_Time_Interval
Delay_Min_Interval:P_Time_Interval
Offset
Input_Event:Event
Output_Event:Regular_Event
Referenced_Event:External_Event
Delay_Max_Interval:P_Time_Interval
Delay_Min_Interval:P_Time_Interval
Rate_Divisor
Input_Event:Event
Output_Event:Regular_Event
Rate_Factor:P_Factor
Join
Input_Event_List:Event[*]
Output_Event:Regular_Event
Delivery_Server
Input_Event:Event
Output_Event_List:Regular_Event[1..*]
Delivery_Policy:P_Delivery_Policy
Query_Server
Input_Event:Event
Output_Event_List:Regular_Event[1..*]
Request_Policy:P_Request_Policy
Activity
Input_Event:Event
Output_Event:Regular_Event
Activity_Usage:Requested_Usage
Activity_Server:Scheduling_Server
System_Timed_Activity

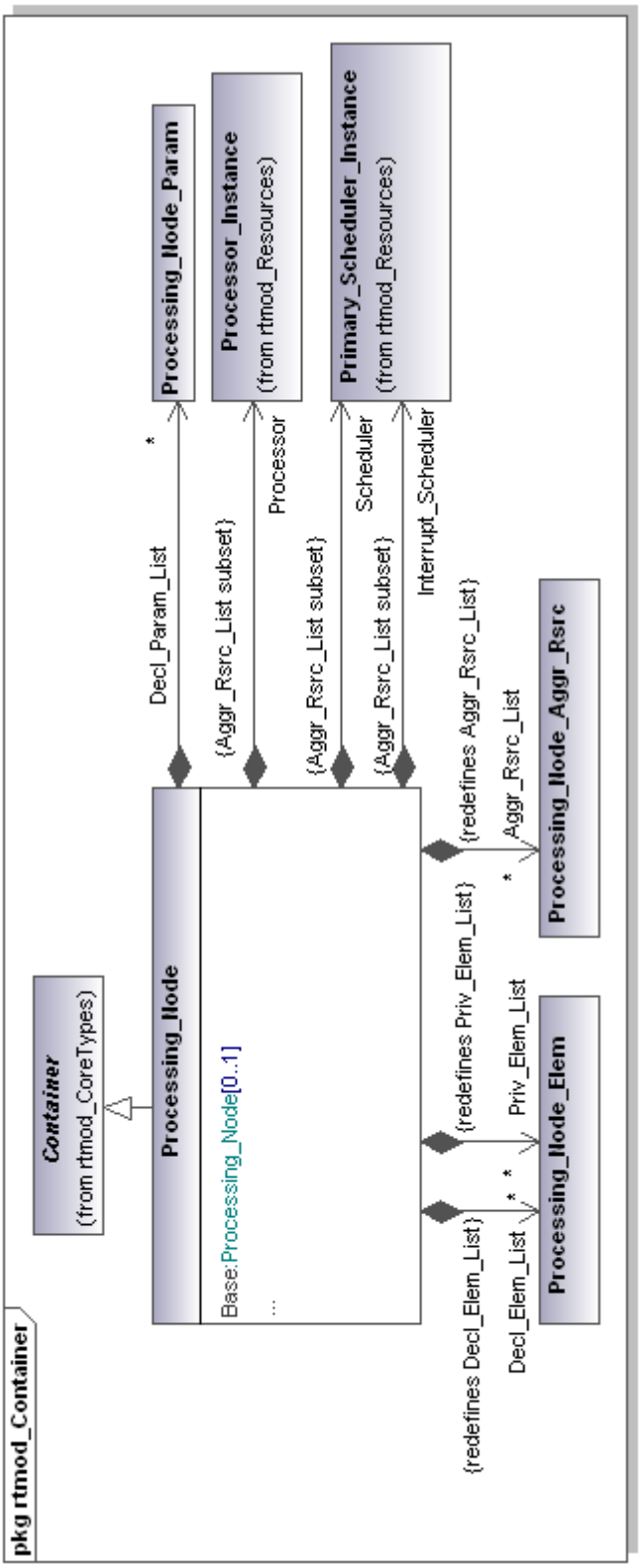

pkg rtmod_Usages
Usage
(from rtmod_CoreTypes)
Requested_Usage
Local_Usage
Job
Base:Job[0..1]
Param_Ref:Job_Ref[0..1]
Decl_Param_List
Job_Param
Job_List
1..*
Composite_Job
Base:Composite_Job[0..1]
Simple_Job
Base:Simple_Job[0..1]
...
Final_Event
Starting_Event
Regular_Event
Event_List
External_Event_List
External_Event
Event_Handler_List
Event_Handler
Aggr_Rsrc_List
Scheduling_Server_Instance
(from rtmod_Resources)

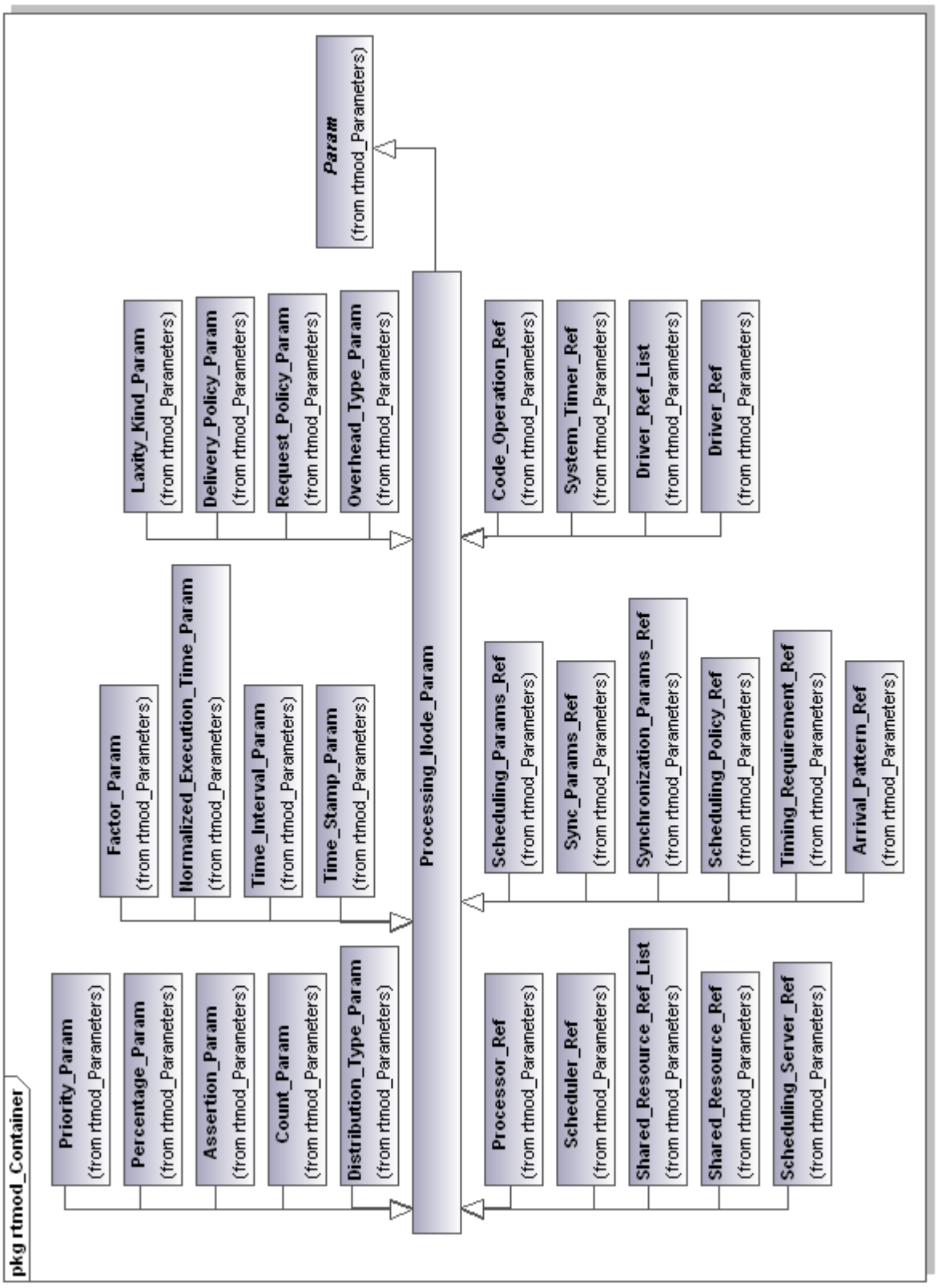

pkg rtmod_Usages
Param
(from rtmod_Parameters)
Job_Param
Requested_Usage_Ref
(from rtmod_Parameters)
Operation_Ref_List
(from rtmod_Parameters)
Job_Ref_List
(from rtmod_Parameters)
Timing_Requirement_Ref
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Scheduling_Params_Ref
(from rtmod_Parameters)
Sync_Params_Ref
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Laxity_Kind_Param
(from rtmod_Parameters)
Assertion_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Composite_Job_Param
Job_Ref_List
(from rtmod_Parameters)
Job_Ref
(from rtmod_Parameters)

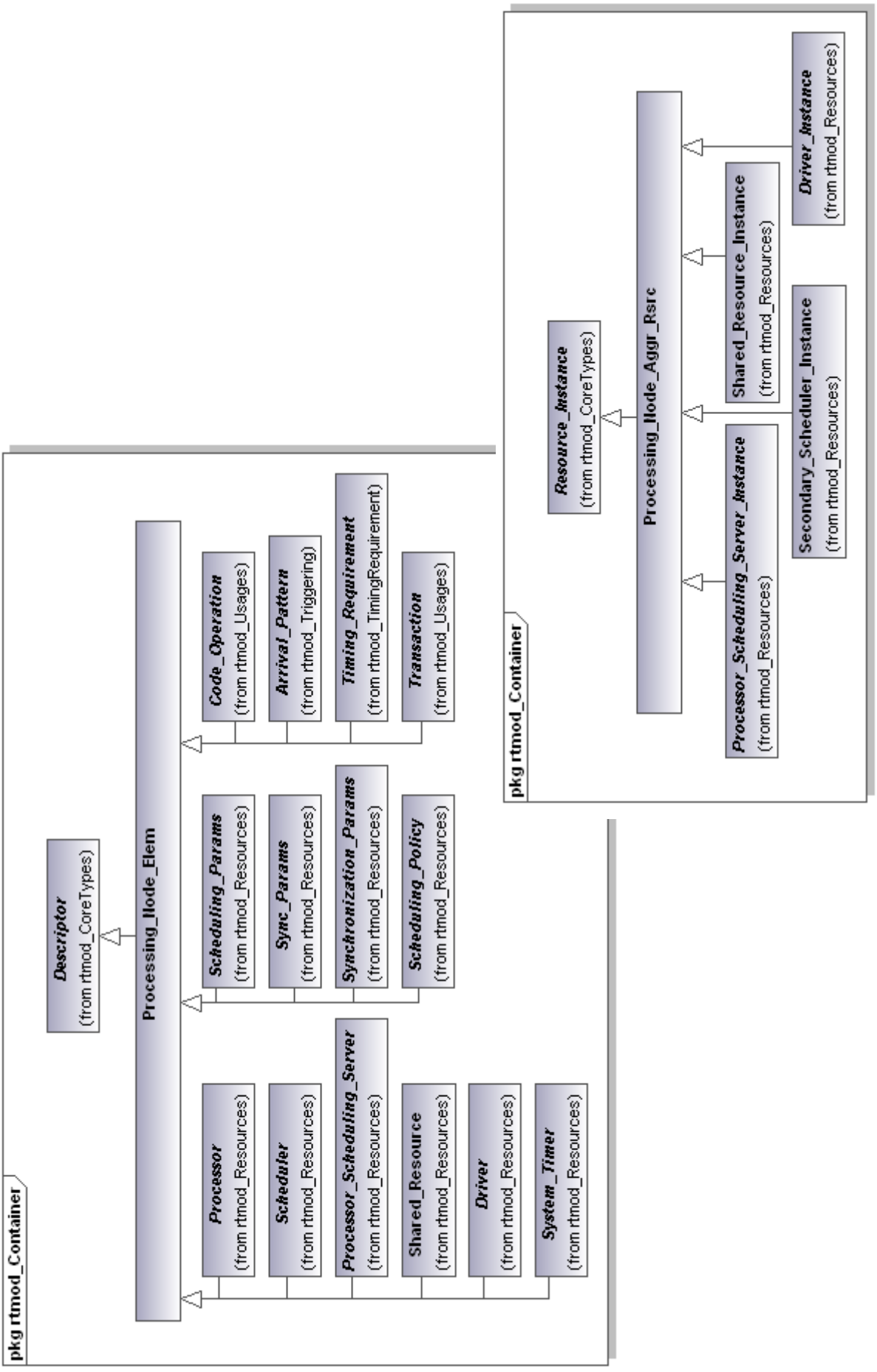

pkg rtmod_Usages
Timing_Req_Observer
Timing_Requirement:Timing_Requirement[1..*]
Local_Timing_Req_Observer
Global_Timing_Req_Observer
Referenced_Event
1
External_Event
pkg rtmod_TimingRequirement
Data_Struct
(from rtmod_CoreTypes)
Timing_Requirement
Base:Timing_Requirement[0..1]
Param_Ref:Timing_Requirement_Ref[0..1]
Declared_Param_List:Timing_Requirement_Param[*]
1..*
Timing_Requirement_List
Composite_Timing_Req
Base:Composite_Timing_Req[0..1]
Timing_Requirement_List:Timing_Requirement[1..*]
Jitter_Timing_Req
Base:Jitter_Timing_Req[0..1]
Jitter:P_Time_Interval
Kind:P_Laxity_Kind
Deadline_Timing_Req
Base:Deadline_Timing_Req[0..1]
Deadline:P_Time_Interval
Kind:P_Laxity_Kind
Max_Miss_Ratio_Timing_Req
Base:Max_Miss_Ratio_Timing_Req[0..1]
Deadline:P_Time_Interval
Ratio:P_Percentage

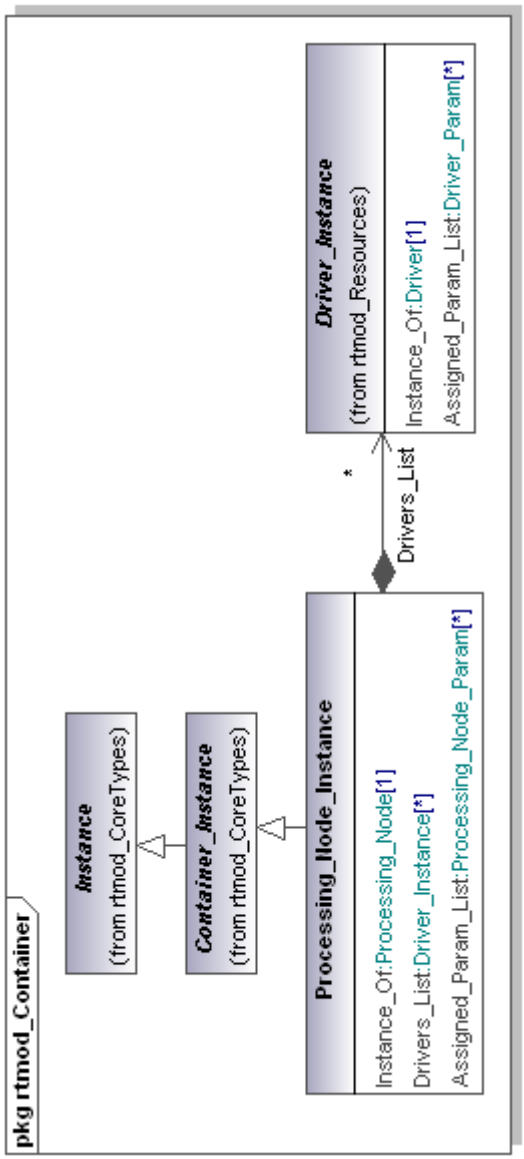

pkg rtmod_Triggering
Data_Struct
(from rtmod_CoreTypes)
Arrival_Pattern
Base:Arrival_Pattern[0..1]
Param_Ref:Arrival_Pattern_Ref[0..1]
Declared_Param_List:Arrival_Pattern_Param[*]
External_Event
(from rtmod_Usages)
Arrival_Pattern
Periodic_Pattern
Base:Periodic_Pattern[0..1]
Period:P_Time_Stamp
Max_Jitter:P_Percentage
Phase:P_Time_Stamp[0..1]
Aperiodic_Pattern
Base:Aperiodic_Pattern[0..1]
Avg_Interarrival:P_Request_Policy
Distribution:P_Distribution_Type
Singular_Pattern
Base:Singular_Pattern[0..1]
Phase:P_Time_Stamp
Irregular_Pattern
Base:Irregular_Pattern[0..1]
Phase:P_Time_Stamp
Interarrival_Times_List:P_Time_Stamp[*]
Sporadic_Pattern
Base:Sporadic_Pattern[0..1]
Min_Interarrival:P_Time_Interval
Bursty_Pattern
Base:Bursty_Pattern[0..1]
Bounded_Interval:P_Time_Interval
Max_Arrivals:P_Count

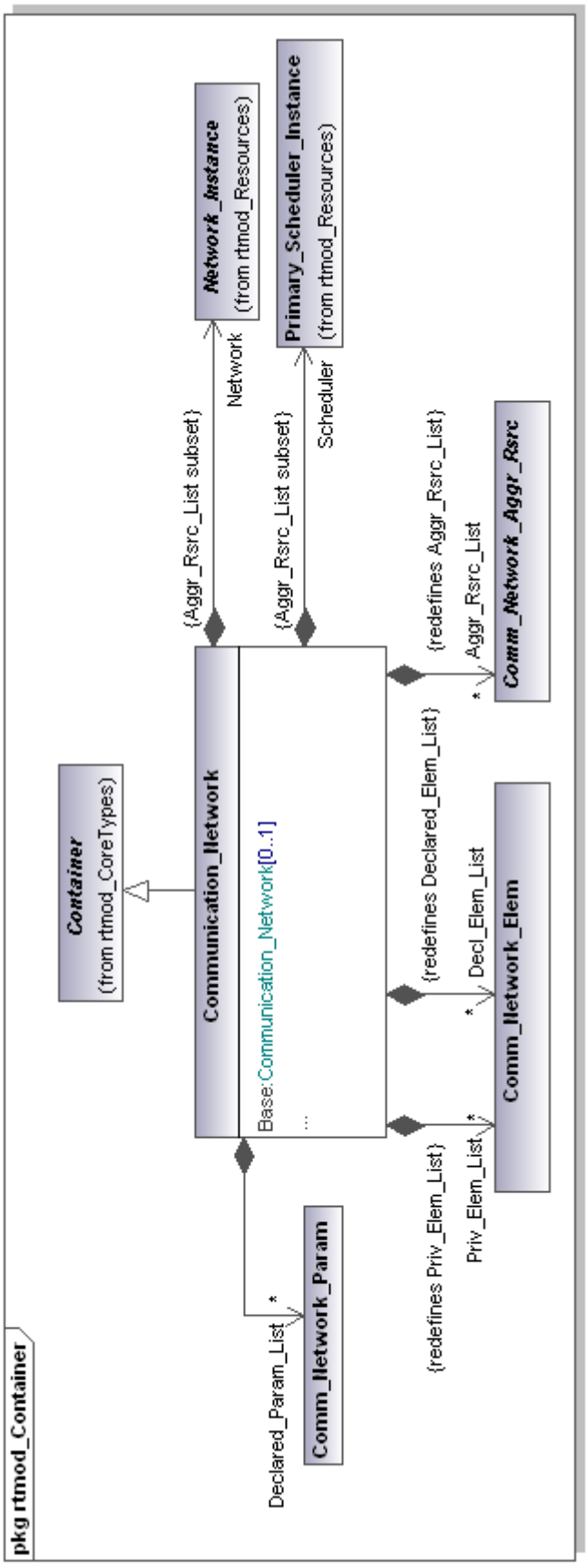

pkg rtmod_Container
Descriptor
(from rtmod_CoreTypes)
Container
(from rtmod_CoreTypes)
Processing_Node
Communication_Network
Software_Module
Communication_Service
Instance
(from rtmod_CoreTypes)
Container_Instance
(from rtmod_CoreTypes)
Processing_Node_Instance
Communication_Network_Instance
Software_Module_Instance
Communication_Service_Instance

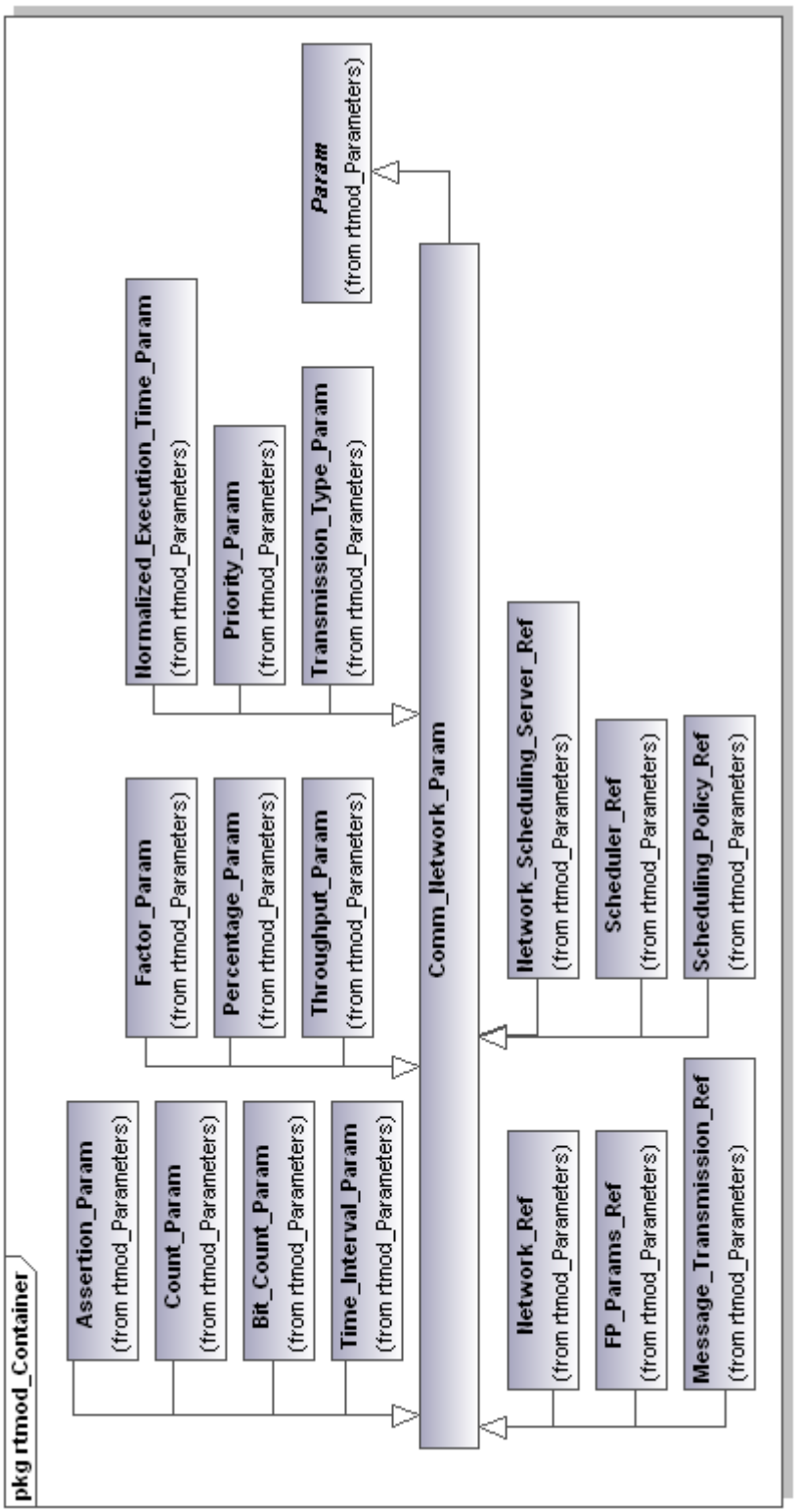

pkg rtmod_Container
Container
(from rtmod_CoreTypes)
Processing_Node
Base:Processing_Node[0..1]
...
Processing_Node_Param
Processor_Instance
(from rtmod_Resources)
Primary_Scheduler_Instance
(from rtmod_Resources)
Processing_Node_Aggr_Rsrc
Processing_Node_Elem
Decl_Param_List
*
{Aggr_Rsrc_List subset}
Processor
{Aggr_Rsrc_List subset}
Scheduler
{Aggr_Rsrc_List subset}
Interrupt_Scheduler
{redefines Aggr_Rsrc_List}
Aggr_Rsrc_List
*
{redefines Priv_Elem_List}
Priv_Elem_List
* *
{redefines Decl_Elem_List}
Decl_Elem_List

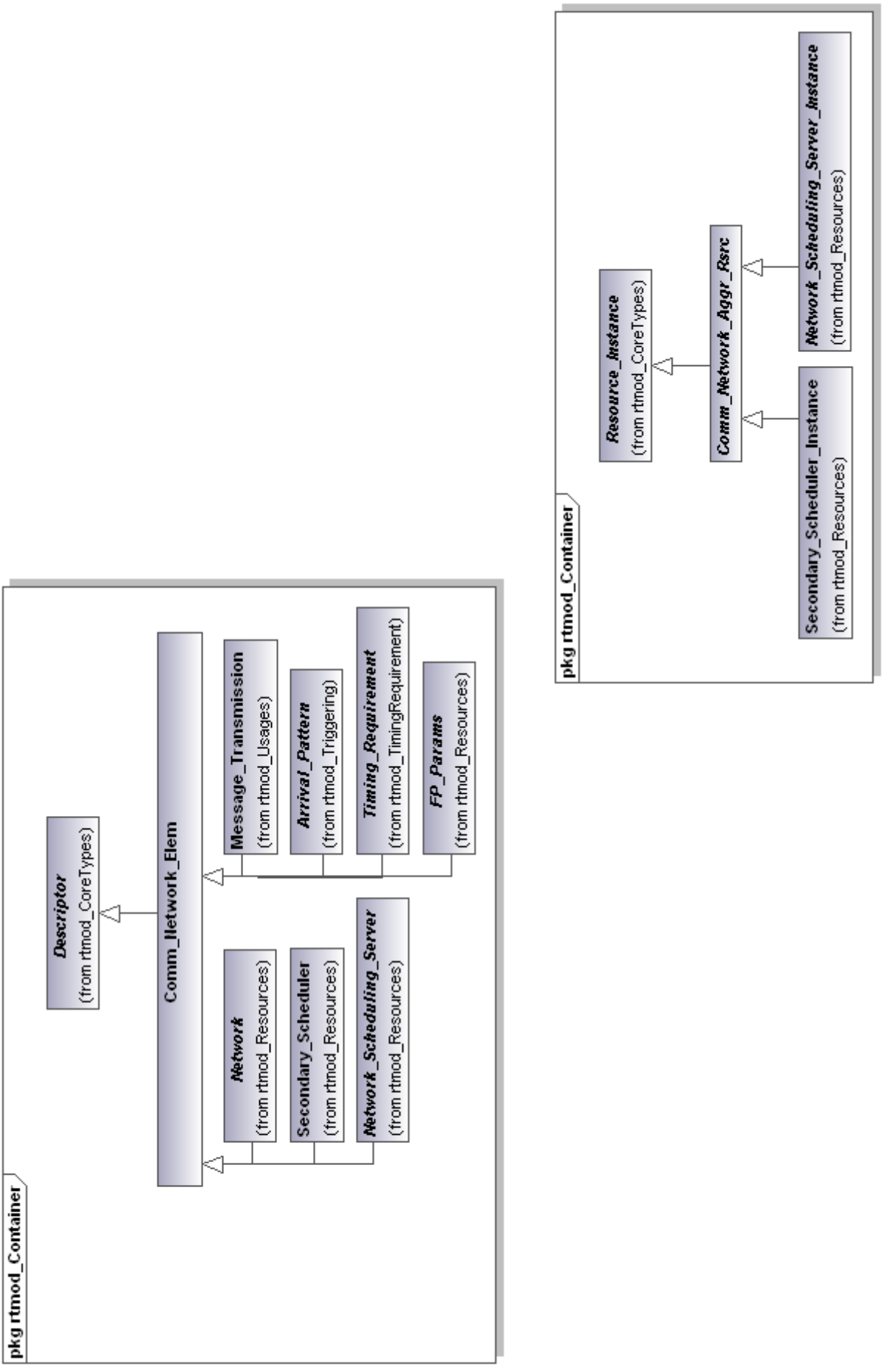

pkg rtmod_Container
Param
(from rtmod_Parameters)
Processing_Node_Param
Laxity_Kind_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)
Overhead_Type_Param
(from rtmod_Parameters)
Code_Operation_Ref
(from rtmod_Parameters)
System_Timer_Ref
(from rtmod_Parameters)
Driver_Ref_List
(from rtmod_Parameters)
Driver_Ref
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Normalized_Execution_Time_Param
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Time_Stamp_Param
(from rtmod_Parameters)
Scheduling_Params_Ref
(from rtmod_Parameters)
Sync_Params_Ref
(from rtmod_Parameters)
Synchronization_Params_Ref
(from rtmod_Parameters)
Scheduling_Policy_Ref
(from rtmod_Parameters)
Timing_Requirement_Ref
(from rtmod_Parameters)
Arrival_Pattern_Ref
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Assertion_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Distribution_Type_Param
(from rtmod_Parameters)
Processor_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Shared_Resource_Ref_List
(from rtmod_Parameters)
Shared_Resource_Ref
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)

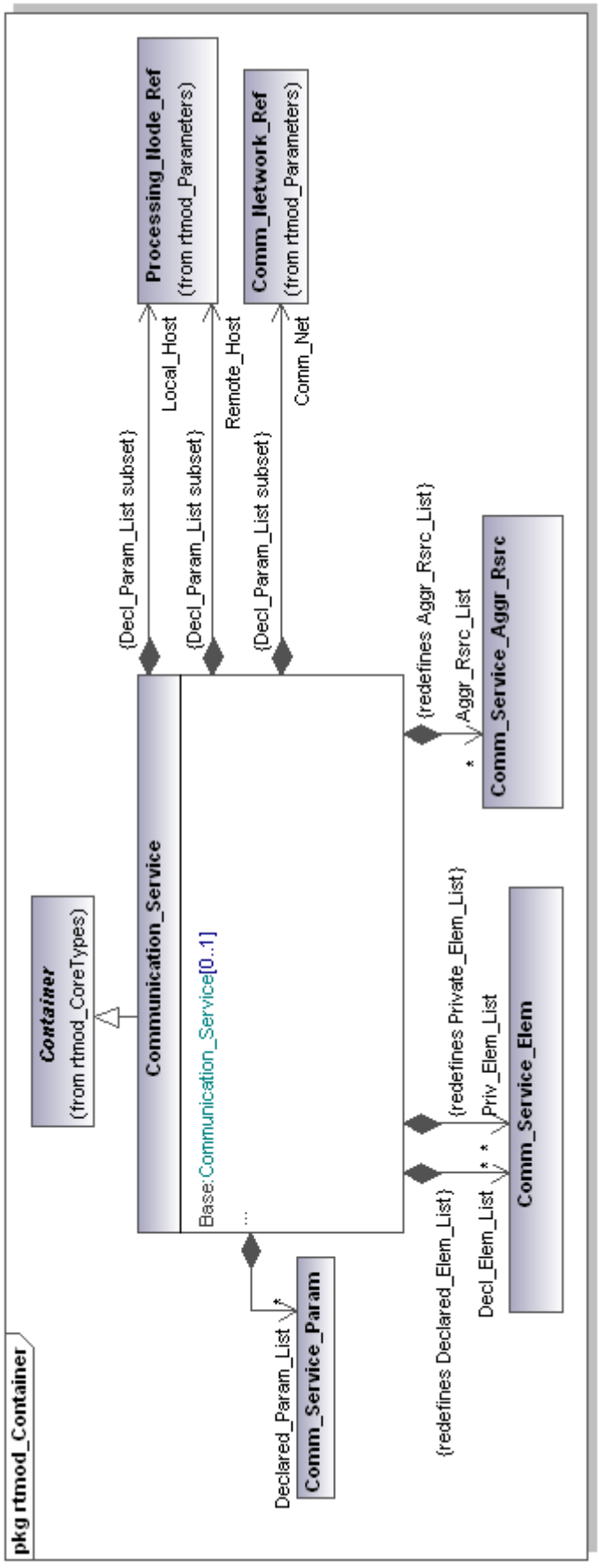

pkg rtmod_Container
Descriptor
(from rtmod_CoreTypes)
Processing_Node_Elem
Code_Operation
(from rtmod_Usages)
Arrival_Pattern
(from rtmod_Triggering)
Timing_Requirement
(from rtmod_TimingRequirement)
Transaction
(from rtmod_Usages)
Scheduling_Params
(from rtmod_Resources)
Sync_Params
(from rtmod_Resources)
Synchronization_Params
(from rtmod_Resources)
Scheduling_Policy
(from rtmod_Resources)
Processor
(from rtmod_Resources)
Scheduler
(from rtmod_Resources)
Processor_Scheduling_Server
(from rtmod_Resources)
Shared_Resource
(from rtmod_Resources)
Driver
(from rtmod_Resources)
System_Timer
(from rtmod_Resources)
pkg rtmod_Container
Resource_Instance
(from rtmod_CoreTypes)
Processing_Node_Aggr_Rsrc
Processor_Scheduling_Server_Instance
(from rtmod_Resources)
Shared_Resource_Instance
(from rtmod_Resources)
Secondary_Scheduler_Instance
(from rtmod_Resources)
Driver_Instance
(from rtmod_Resources)

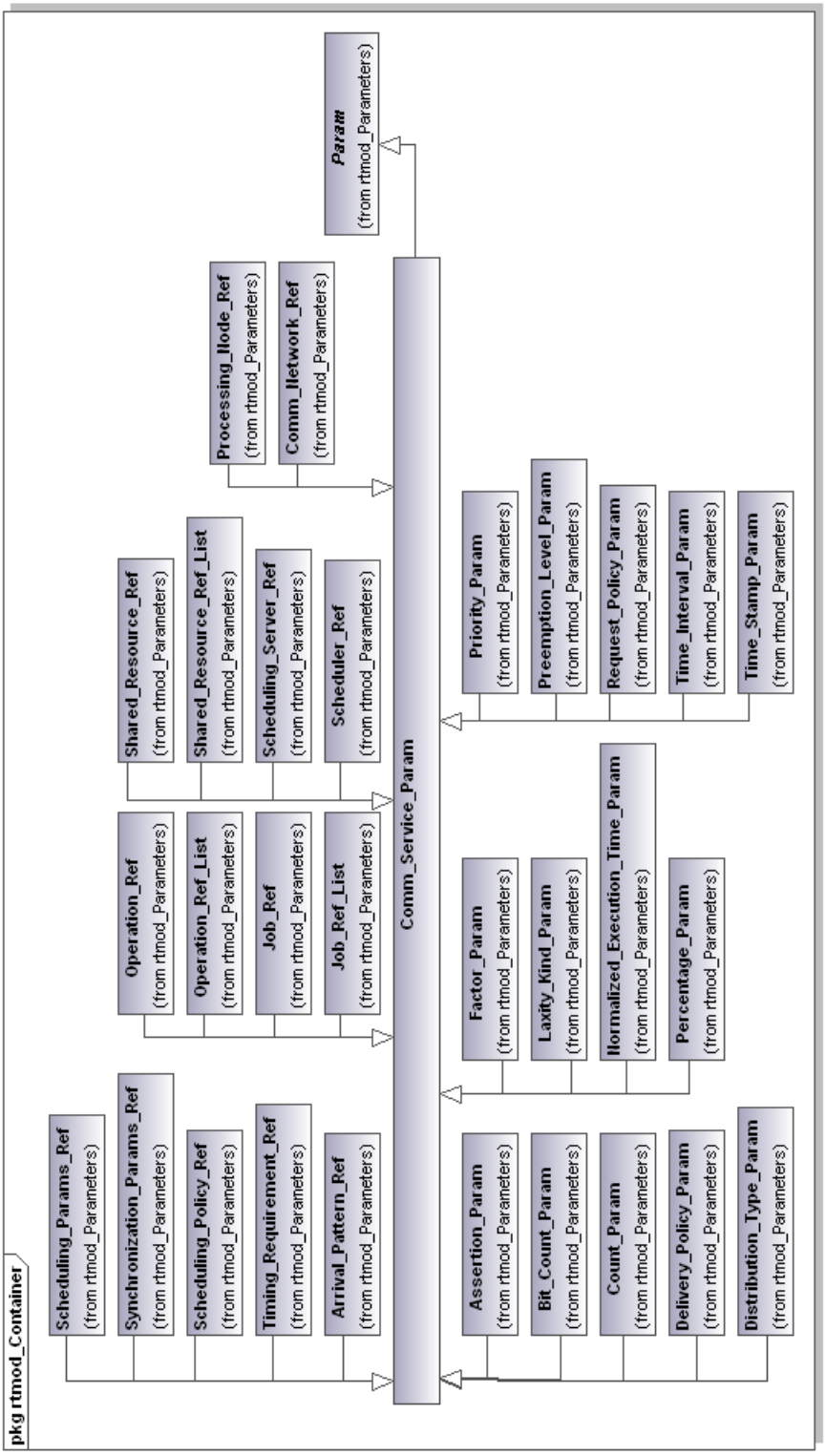

pkg rtmod_Container
Instance
(from rtmod_CoreTypes)
Container_Instance
(from rtmod_CoreTypes)
Processing_Node_Instance
Instance_Of:Processing_Node[1]
Drivers_List:Driver_Instance[*]
Assigned_Param_List:Processing_Node_Param[*]
Drivers_List
*
Driver_Instance
(from rtmod_Resources)
Instance_Of:Driver[1]
Assigned_Param_List:Driver_Param[*]


pkg rtmod_Container
Container
(from rtmod_CoreTypes)
Communication_Network
Base:Communication_Network[0..1]
...
Network_Instance
(from rtmod_Resources)
Primary_Scheduler_Instance
(from rtmod_Resources)
{Aggr_Rsrc_List subset}
Network
{Aggr_Rsrc_List subset}
Scheduler
{redefines Aggr_Rsrc_List}
Aggr_Rsrc_List
*
Comm_Network_Aggr_Rsrc
{redefines Declared_Elem_List}
Decl_Elem_List
*
Comm_Network_Elem
{redefines Priv_Elem_List}
Priv_Elem_List
*
Declared_Param_List
*
Comm_Network_Param


pkg rtmod_Container
Assertion_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Bit_Count_Param
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Throughput_Param
(from rtmod_Parameters)
Normalized_Execution_Time_Param
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Transmission_Type_Param
(from rtmod_Parameters)
Param
(from rtmod_Parameters)
Comm_Network_Param
Network_Ref
(from rtmod_Parameters)
FP_Params_Ref
(from rtmod_Parameters)
Message_Transmission_Ref
(from rtmod_Parameters)
Network_Scheduling_Server_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Scheduling_Policy_Ref
(from rtmod_Parameters)

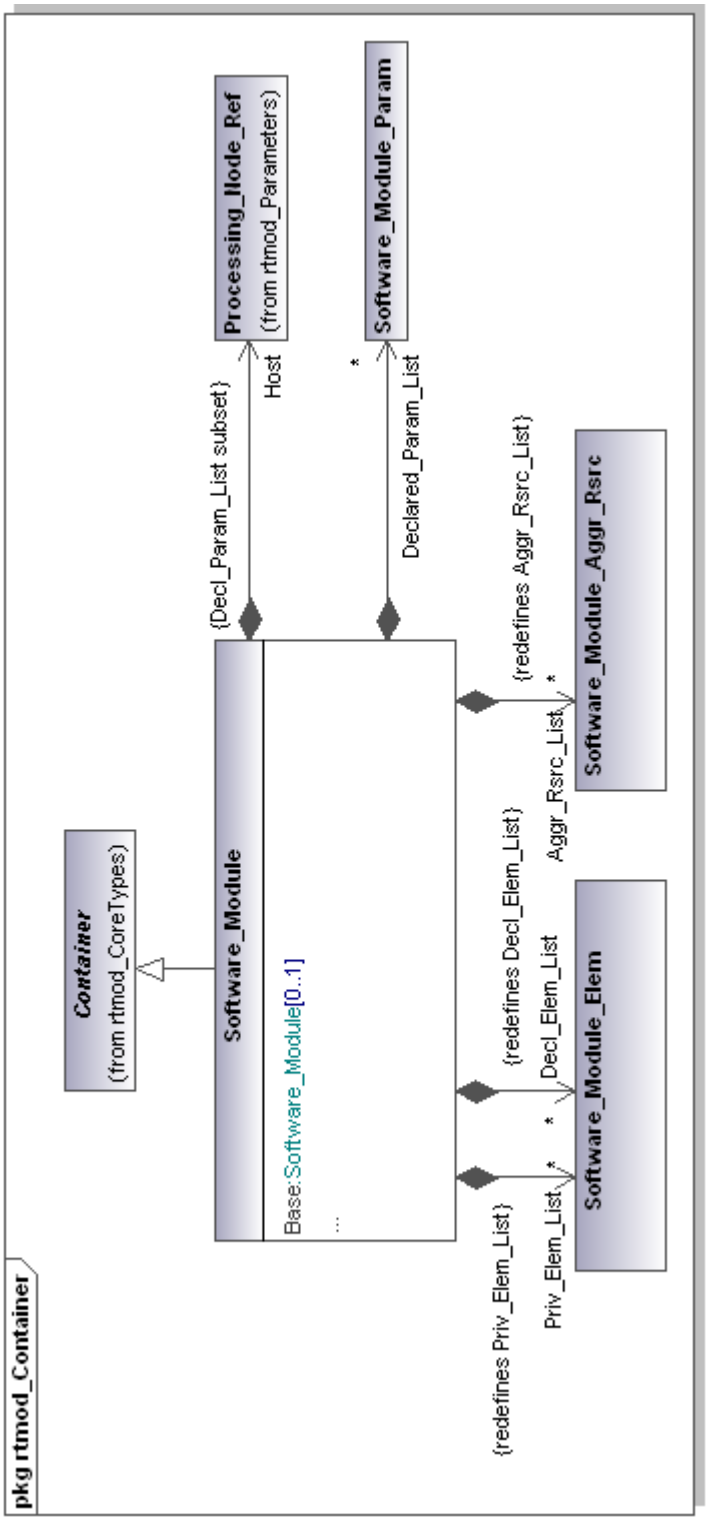

pkg rtmod_Container
Descriptor
(from rtmod_CoreTypes)
Comm_Network_Elem
Message_Transmission
(from rtmod_Usages)
Arrival_Pattern
(from rtmod_Triggering)
Timing_Requirement
(from rtmod_TimingRequirement)
FP_Params
(from rtmod_Resources)
Network
(from rtmod_Resources)
Secondary_Scheduler
(from rtmod_Resources)
Network_Scheduling_Server
(from rtmod_Resources)
pkg rtmod_Container
Resource_Instance
(from rtmod_CoreTypes)
Comm_Network_Aggr_Rsrc
Network_Scheduling_Server_Instance
(from rtmod_Resources)
Secondary_Scheduler_Instance
(from rtmod_Resources)

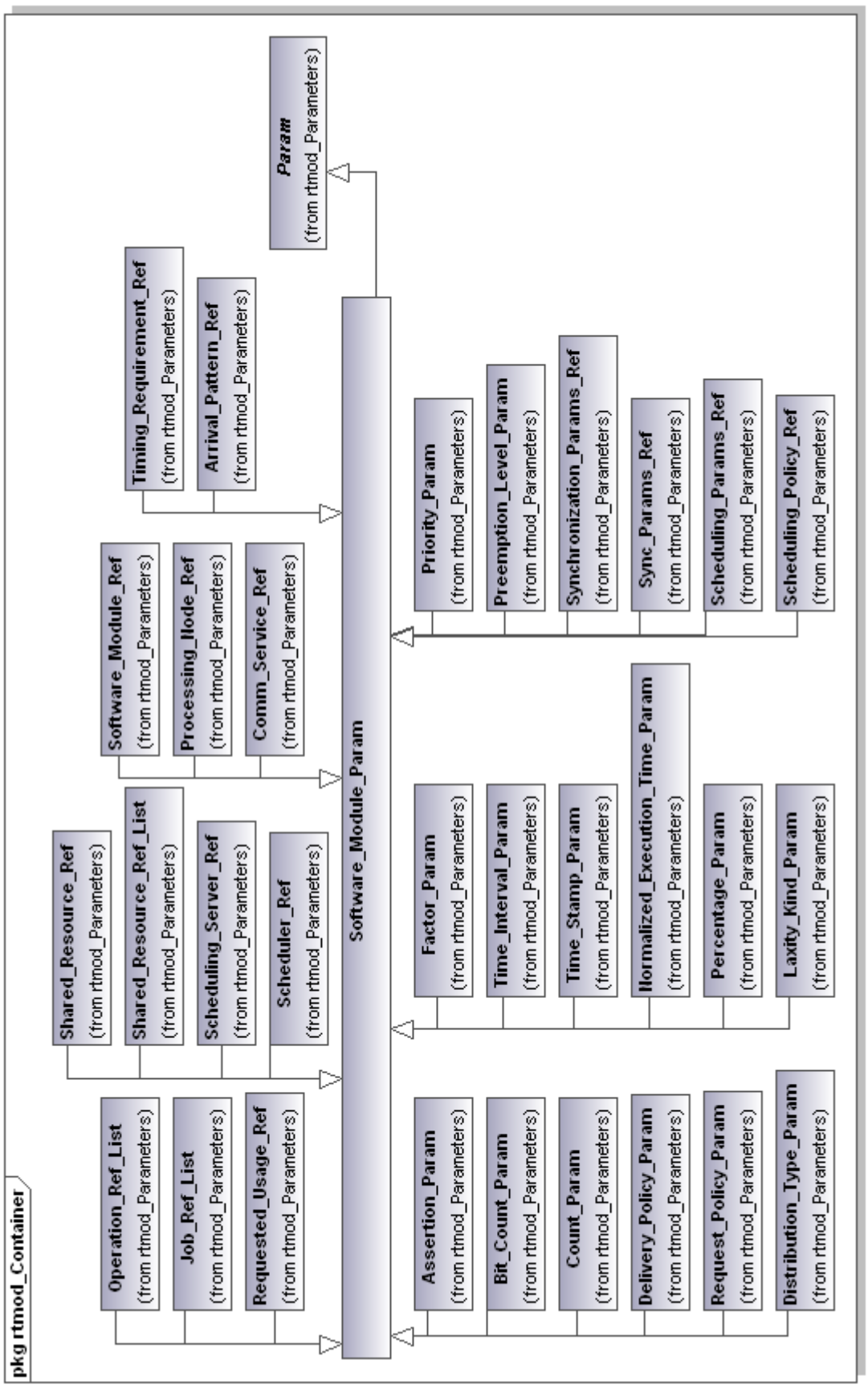

pkg rtmod_Container
Container
(from rtmod_CoreTypes)
Communication_Service
Base:Communication_Service[0..1]
...
Processing_Node_Ref
(from rtmod_Parameters)
Comm_Network_Ref
(from rtmod_Parameters)
{Decl_Param_List subset}
Local_Host
{Decl_Param_List subset}
Remote_Host
{Decl_Param_List subset}
Comm_Net
{redefines Aggr_Rsrc_List}
Aggr_Rsrc_List
*
Comm_Service_Aggr_Rsrc
{redefines Private_Elem_List}
Priv_Elem_List
* *
{redefines Declared_Elem_List}
Decl_Elem_List
Comm_Service_Elem
Declared_Param_List
*
Comm_Service_Param

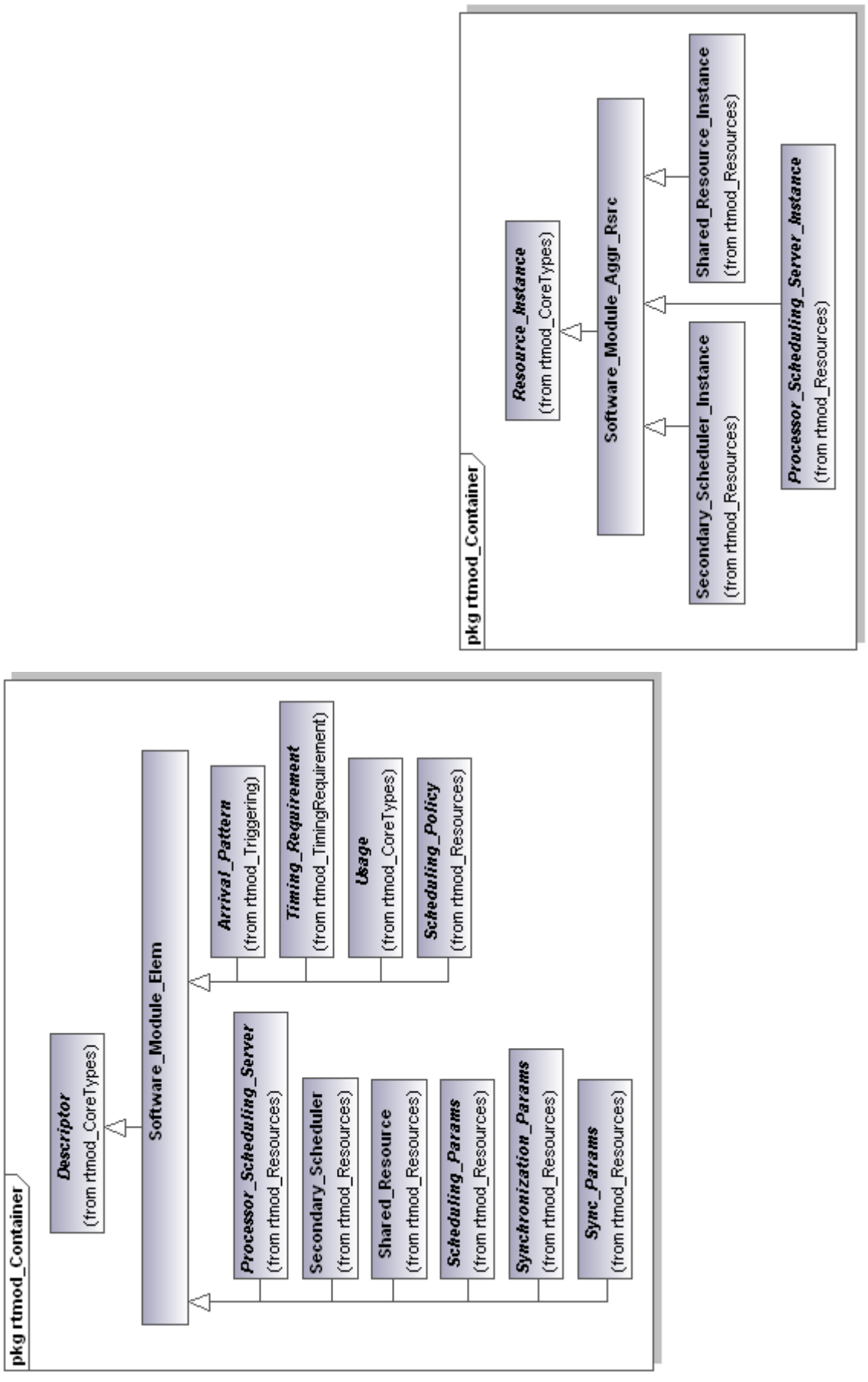

pkg rtmod_Container
Scheduling_Params_Ref
(from rtmod_Parameters)
Synchronization_Params_Ref
(from rtmod_Parameters)
Scheduling_Policy_Ref
(from rtmod_Parameters)
Timing_Requirement_Ref
(from rtmod_Parameters)
Arrival_Pattern_Ref
(from rtmod_Parameters)
Operation_Ref
(from rtmod_Parameters)
Operation_Ref_List
(from rtmod_Parameters)
Job_Ref
(from rtmod_Parameters)
Job_Ref_List
(from rtmod_Parameters)
Shared_Resource_Ref
(from rtmod_Parameters)
Shared_Resource_Ref_List
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Processing_Node_Ref
(from rtmod_Parameters)
Comm_Network_Ref
(from rtmod_Parameters)
Param
(from rtmod_Parameters)
Comm_Service_Param
Assertion_Param
(from rtmod_Parameters)
Bit_Count_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Distribution_Type_Param
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Laxity_Kind_Param
(from rtmod_Parameters)
Normalized_Execution_Time_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Preemption_Level_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Time_Stamp_Param
(from rtmod_Parameters)

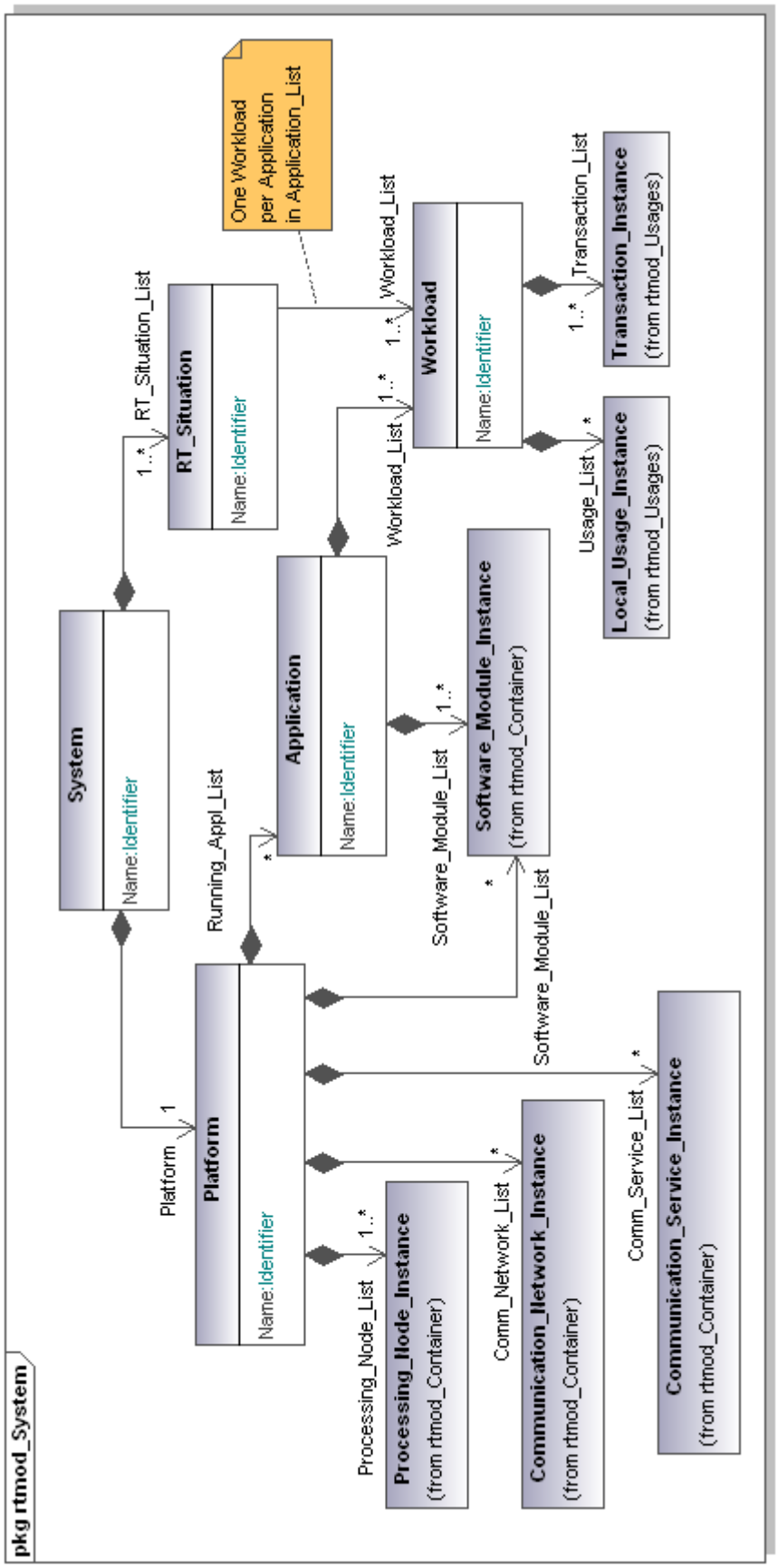
pkg rtmod_Container
Descriptor
(from rtmod_CoreTypes)
Comm_Service_Elem
Scheduling_Params
(from rtmod_Resources)
Operation
(from rtmod_Usages)
Arrival_Pattern
(from rtmod_Triggering)
Job
(from rtmod_Usages)
Scheduler
(from rtmod_Resources)
Scheduling_Server
(from rtmod_Resources)
Shared_Resource
(from rtmod_Resources)
Timing_Requirement
(from rtmod_TimingRequirement)
Transaction
(from rtmod_Usages)

pkg rtmod_Container
Resource_Instance
(from rtmod_CoreTypes)
Comm_Service_Aggr_Rsrc
Scheduler_Instance
(from rtmod_Resources)
Scheduling_Server_Instance
(from rtmod_Resources)
Shared_Resource_Instance
(from rtmod_Resources)


pkg rtmod_Container
Container
(from rtmod_CoreTypes)
Software_Module
Base:Software_Module[0..1]
...
Processing_Ilode_Ref
(from rtmod_Parameters)
Software_Module_Param
{Decl_Param_List subset}
Host
Declared_Param_List
{redefines Aggr_Rsrc_List}
Aggr_Rsrc_List
Software_Module_Aggr_Rsrc
{redefines Decl_Elem_List}
Decl_Elem_List
{redefines Priv_Elem_List}
Priv_Elem_List
Software_Module_Elem


pkg rtmod_Container
Param
(from rtmod_Parameters)
Timing_Requirement_Ref
(from rtmod_Parameters)
Arrival_Pattern_Ref
(from rtmod_Parameters)
Software_Module_Ref
(from rtmod_Parameters)
Processing_Node_Ref
(from rtmod_Parameters)
Comm_Service_Ref
(from rtmod_Parameters)
Shared_Resource_Ref
(from rtmod_Parameters)
Shared_Resource_Ref_List
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Operation_Ref_List
(from rtmod_Parameters)
Job_Ref_List
(from rtmod_Parameters)
Requested_Usage_Ref
(from rtmod_Parameters)
Software_Module_Param
Priority_Param
(from rtmod_Parameters)
Preemption_Level_Param
(from rtmod_Parameters)
Synchronization_Params_Ref
(from rtmod_Parameters)
Sync_Params_Ref
(from rtmod_Parameters)
Scheduling_Params_Ref
(from rtmod_Parameters)
Scheduling_Policy_Ref
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Time_Stamp_Param
(from rtmod_Parameters)
Normalized_Execution_Time_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Laxity_Kind_Param
(from rtmod_Parameters)
Assertion_Param
(from rtmod_Parameters)
Bit_Count_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)
Distribution_Type_Param
(from rtmod_Parameters)


pkg rtmod_Container
Descriptor
(from rtmod_CoreTypes)
Software_Module_Elem
Arrival_Pattern
(from rtmod_Triggering)
Timing_Requirement
(from rtmod_TimingRequirement)
Usage
(from rtmod_CoreTypes)
Scheduling_Policy
(from rtmod_Resources)
Processor_Scheduling_Server
(from rtmod_Resources)
Secondary_Scheduler
(from rtmod_Resources)
Shared_Resource
(from rtmod_Resources)
Scheduling_Params
(from rtmod_Resources)
Synchronization_Params
(from rtmod_Resources)
Sync_Params
(from rtmod_Resources)
pkg rtmod_Container
Resource_Instance
(from rtmod_CoreTypes)
Software_Module_Aggr_Rsrc
Shared_Resource_Instance
(from rtmod_Resources)
Secondary_Scheduler_Instance
(from rtmod_Resources)
Processor_Scheduling_Server_Instance
(from rtmod_Resources)


pkg rtmod_System
System
Name:Identifier
RT_Situation
Name:Identifier
RT_Situation_List
1..*
One Workload
per Application
in Application_List
Workload
Name:Identifier
Workload_List
1..*
Transaction_List
Transaction_Instance
(from rtmod_Usages)
Application
Name:Identifier
Workload_List
Local_Usage_Instance
(from rtmod_Usages)
Usage_List
Platform
Name:Identifier
Platform
1
Running_Appl_List
Software_Module_Instance
(from rtmod_Container)
Software_Module_List
Processing_Node_List
Processing_Node_Instance
(from rtmod_Container)
Comm_Network_List
Communication_Network_Instance
(from rtmod_Container)
Comm_Service_List
Communication_Service_Instance
(from rtmod_Container)

Anexo B. Metamodelo CBS-MAST

En este anexo se muestra los elementos que se añaden a Mod-MAST para adaptarlo al desarrollo de aplicaciones basadas en componentes, dando lugar al metamodelo CBS-MAST. Como muestra la figura B.1, éste importa todos los elementos de Mod-MAST y mantiene su misma estructura de paquetes. CBS-MAST incorpora algunos elementos nuevos, como los nuevos contenedores *Software_Component* y *Software_Connector*, o especializa alguno de los elementos de Mod-MAST, como es el caso de *CBS_System*. En este anexo se muestran únicamente aquellos elementos que son específicos de CBS-MAST.

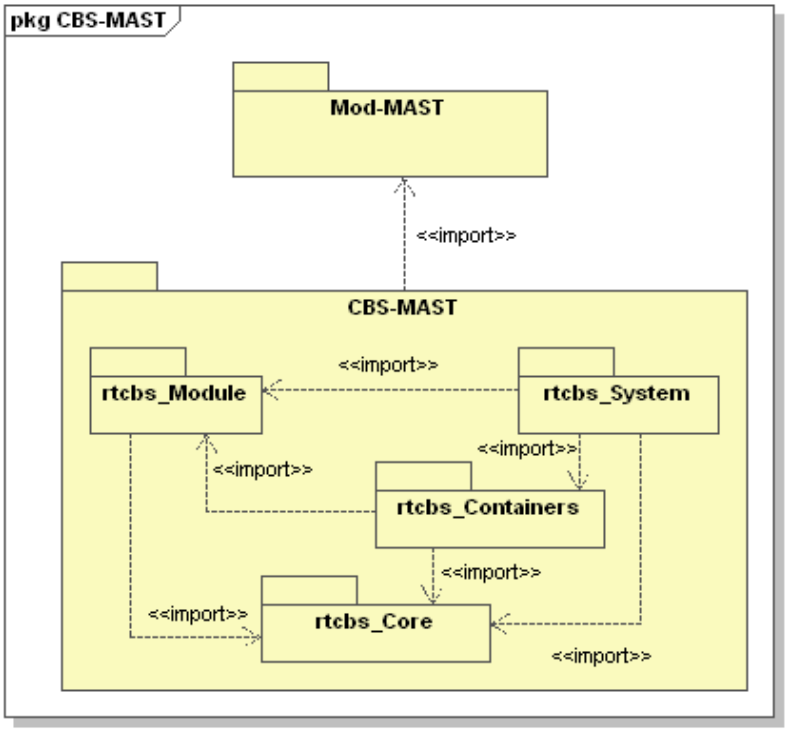
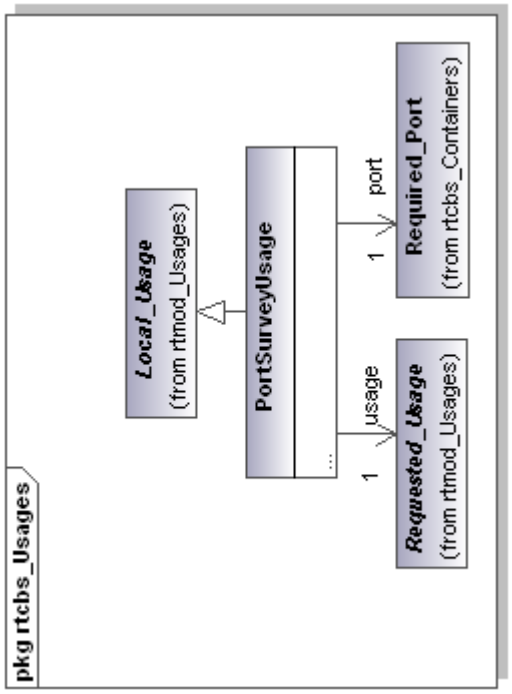
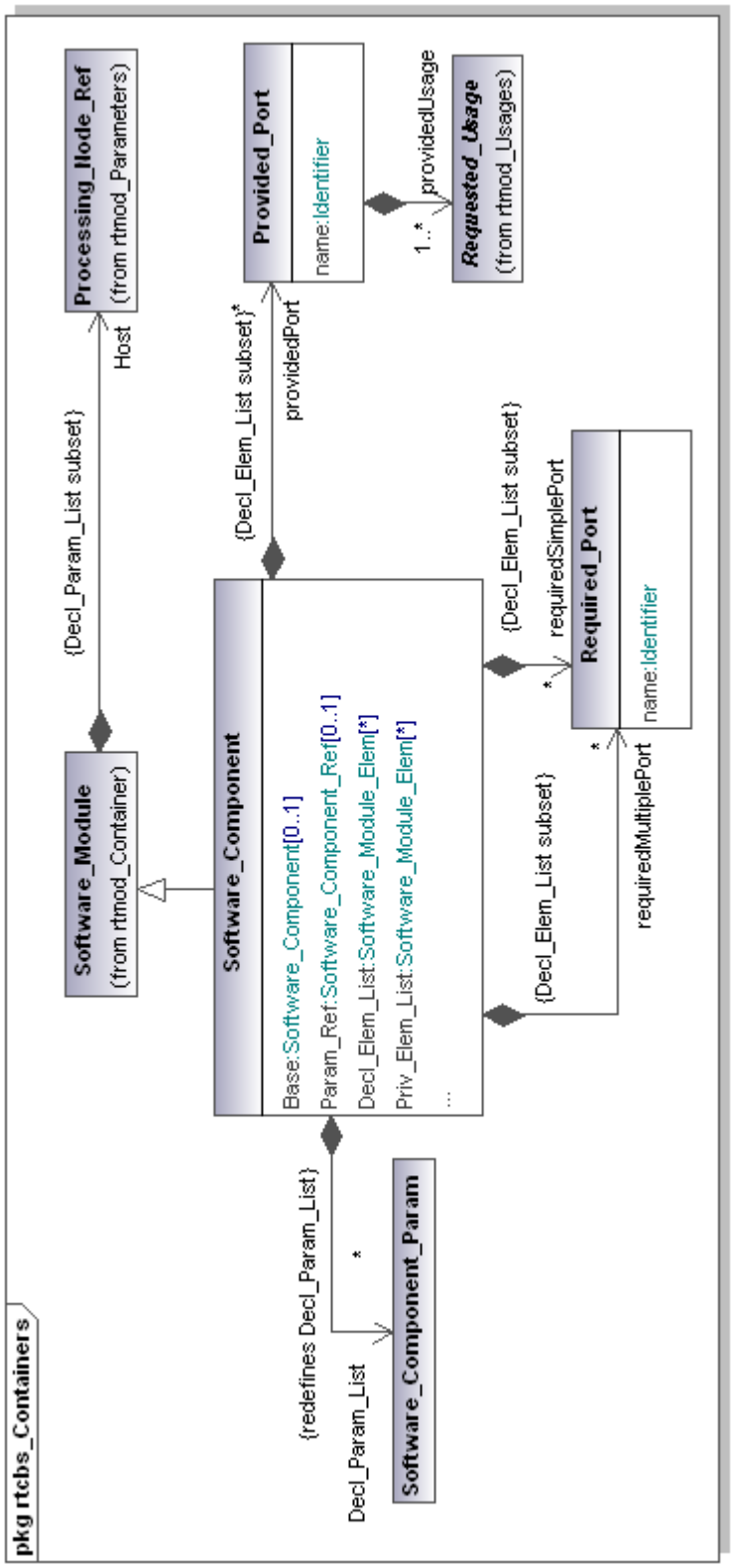
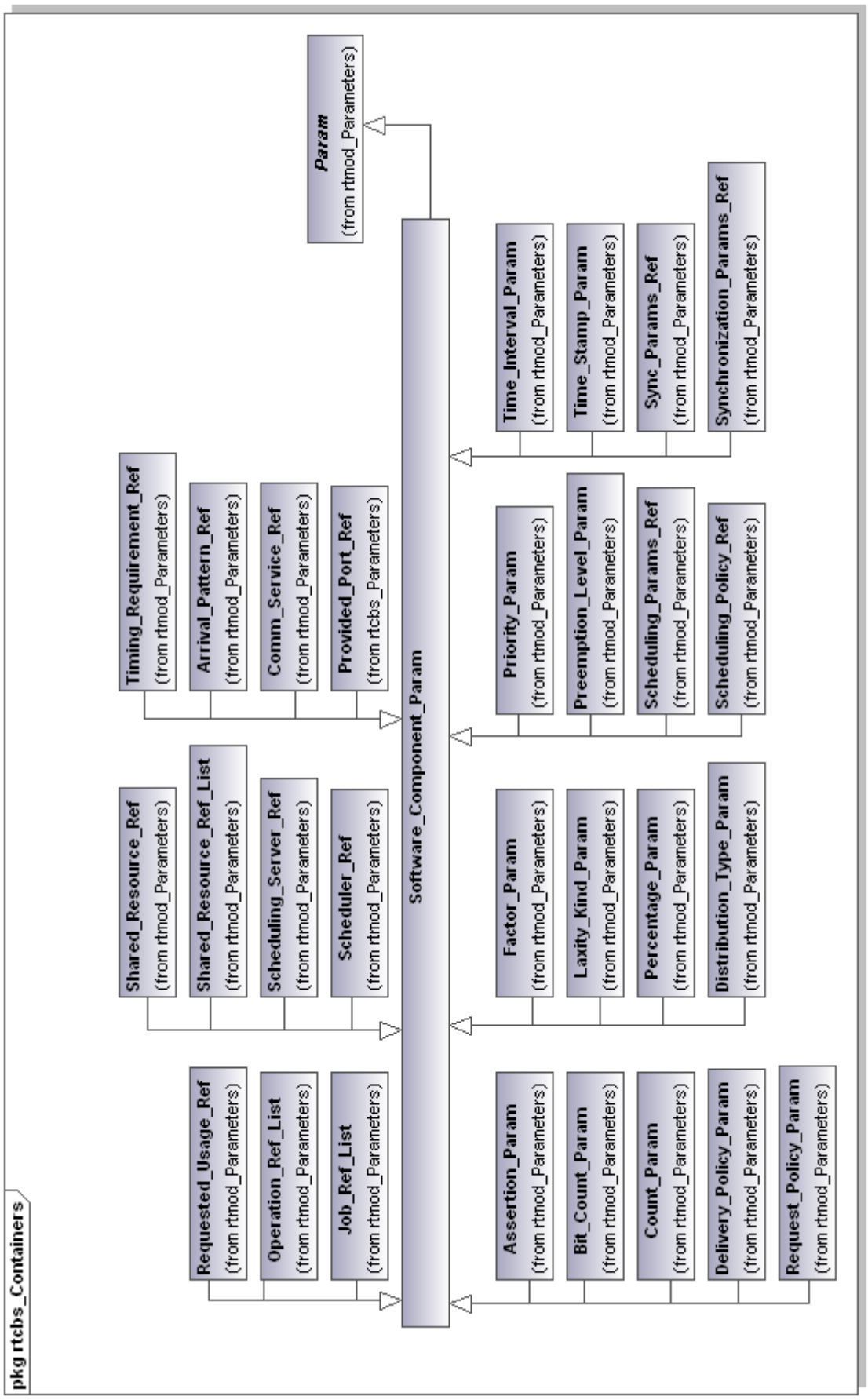


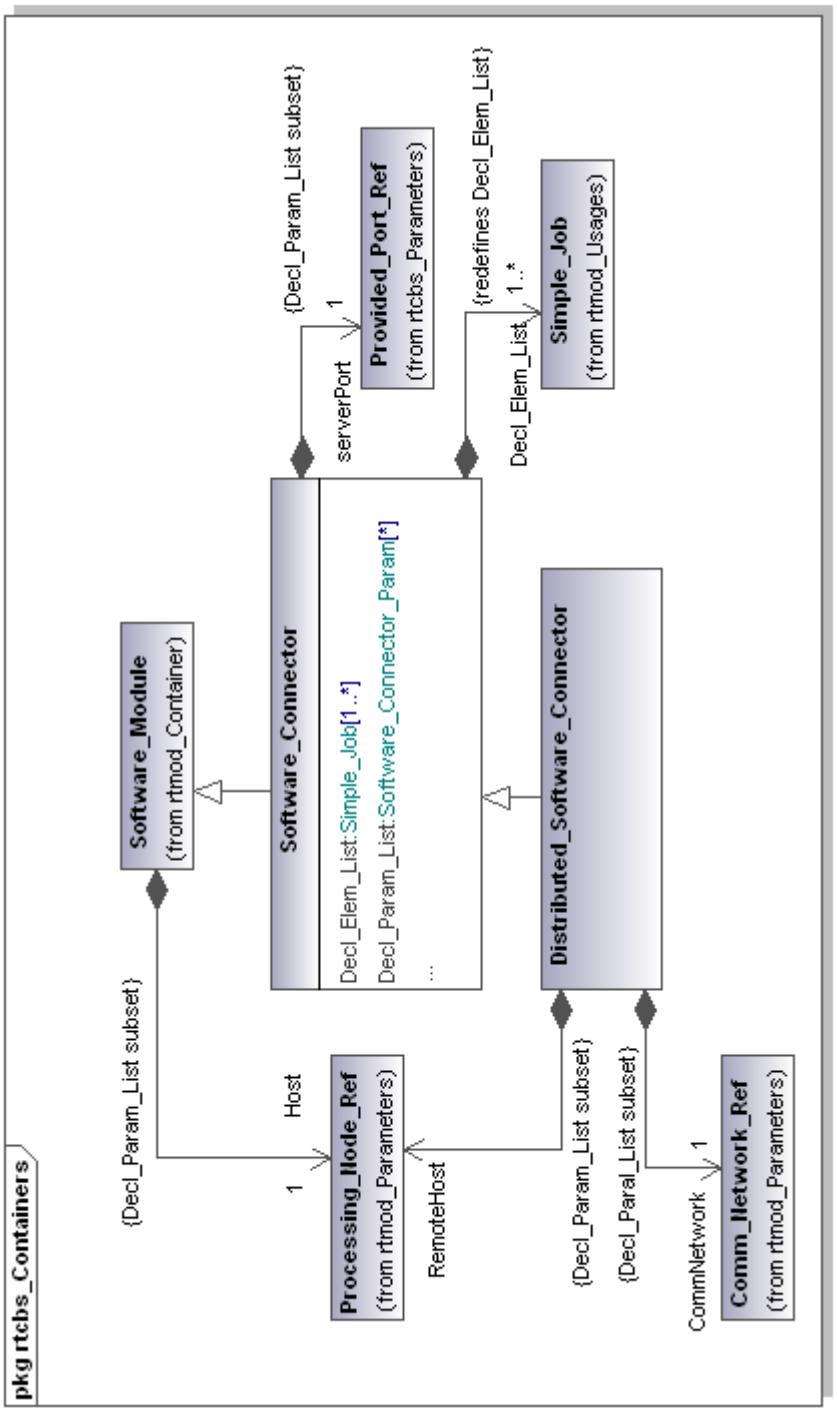
Figura B.1: Estructura de paquetes del metamodelo CBS-MAST

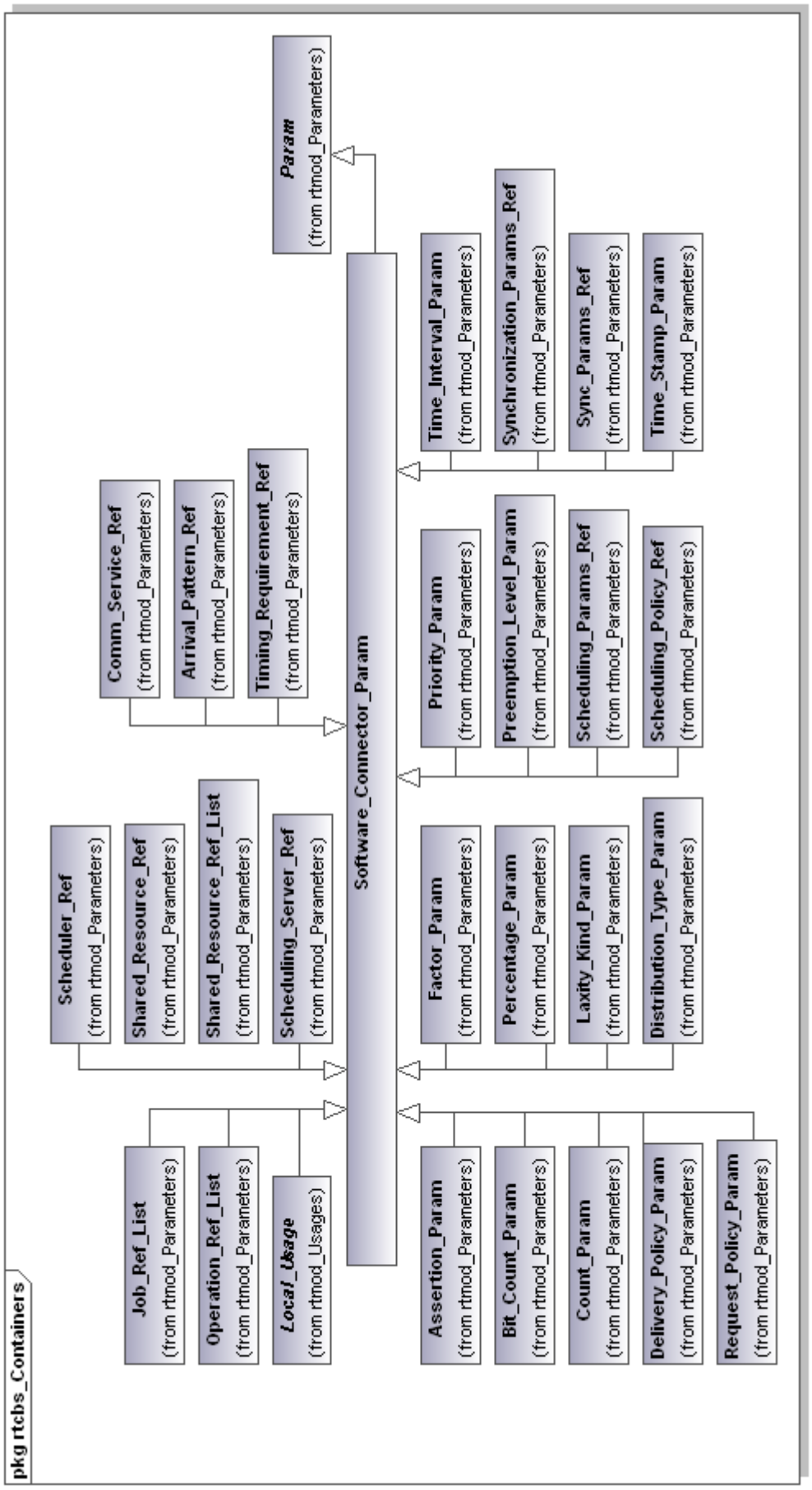
La lista de diagramas que se muestran en este anexo es la siguiente:

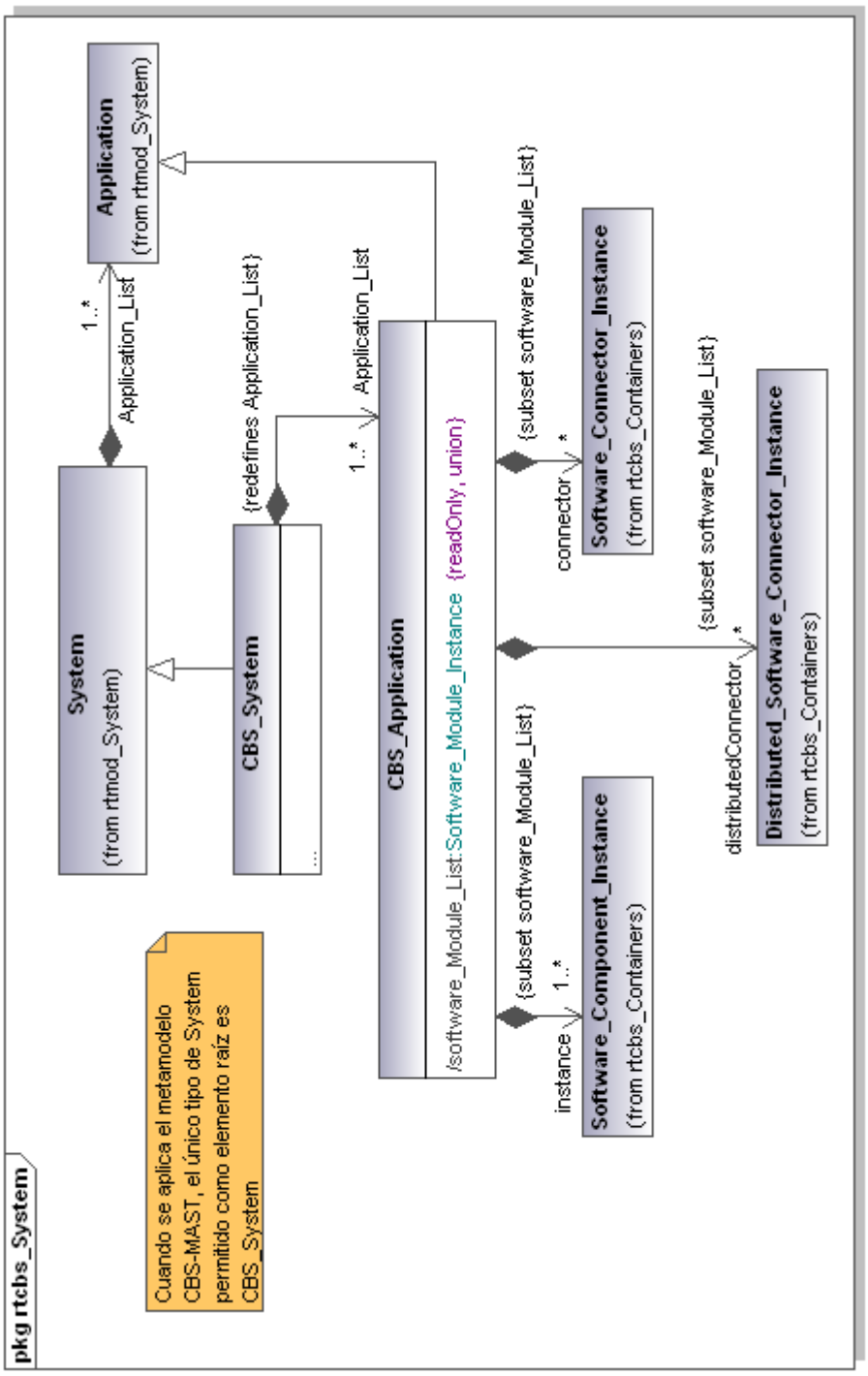

pkg rtcbs_Usages
Local_Usage
(from rtmod_Usages)
PortSurveyUsage
...
usage
1
port
1
Requested_Usage
(from rtmod_Usages)
Required_Port
(from rtcbs_Containers)


pkg rtcbs_Containers
Processing_Node_Ref
(from rtmod_Parameters)
Host
{Decl_Param_List subset}
Software_Module
(from rtmod_Container)
Provided_Port
name:Identifier
providedPort
{Decl_Elem_List subset}*
1..*
providedUsage
Requested_Usage
(from rtmod_Usages)
Software_Component
Base:Software_Component[0..1]
Param_Ref:Software_Component_Ref[0..1]
Decl_Elem_List:Software_Module_Elem[*]
Priv_Elem_List:Software_Module_Elem[*]
...
{Decl_Elem_List subset}
requiredSimplePort
*
Required_Port
name:Identifier
*
requiredMultiplePort
{Decl_Elem_List subset}
{redefines Decl_Param_List}
Decl_Param_List
*
Software_Component_Param


pkg rtcbs_Containers
Param
(from rtmod_Parameters)
Software_Component_Param
Timing_Requirement_Ref
(from rtmod_Parameters)
Arrival_Pattern_Ref
(from rtmod_Parameters)
Comm_Service_Ref
(from rtmod_Parameters)
Provided_Port_Ref
(from rtcbs_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Time_Stamp_Param
(from rtmod_Parameters)
Sync_Params_Ref
(from rtmod_Parameters)
Synchronization_Params_Ref
(from rtmod_Parameters)
Shared_Resource_Ref
(from rtmod_Parameters)
Shared_Resource_Ref_List
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Preemption_Level_Param
(from rtmod_Parameters)
Scheduling_Params_Ref
(from rtmod_Parameters)
Scheduling_Policy_Ref
(from rtmod_Parameters)
Requested_Usage_Ref
(from rtmod_Parameters)
Operation_Ref_List
(from rtmod_Parameters)
Job_Ref_List
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Laxity_Kind_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Distribution_Type_Param
(from rtmod_Parameters)
Assertion_Param
(from rtmod_Parameters)
Bit_Count_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)


pkg rtcbs_Containers
Software_Module
(from rtmod_Container)
Software_Connector
Decl_Elem_List:Simple_Job[1..*]
Decl_Param_List:Software_Connector_Param[*]
...
Distributed_Software_Connector
Provided_Port_Ref
(from rtcbs_Parameters)
Simple_Job
(from rtmod_Usages)
Processing_Node_Ref
(from rtmod_Parameters)
Comm_Network_Ref
(from rtmod_Parameters)
{Decl_Param_List subset}
{Decl_Param_List subset}
1
serverPort
{redefines Decl_Elem_List}
1..*
Decl_Elem_List
Host
1
RemoteHost
{Decl_Param_List subset}
{Decl_Paral_List subset}
CommNetwork
1


pkg rtcbs_Containers
Param
(from rtmod_Parameters)
Software_Connector_Param
Comm_Service_Ref
(from rtmod_Parameters)
Arrival_Pattern_Ref
(from rtmod_Parameters)
Timing_Requirement_Ref
(from rtmod_Parameters)
Time_Interval_Param
(from rtmod_Parameters)
Synchronization_Params_Ref
(from rtmod_Parameters)
Sync_Params_Ref
(from rtmod_Parameters)
Time_Stamp_Param
(from rtmod_Parameters)
Priority_Param
(from rtmod_Parameters)
Preemption_Level_Param
(from rtmod_Parameters)
Scheduling_Params_Ref
(from rtmod_Parameters)
Scheduling_Policy_Ref
(from rtmod_Parameters)
Scheduler_Ref
(from rtmod_Parameters)
Shared_Resource_Ref
(from rtmod_Parameters)
Shared_Resource_Ref_List
(from rtmod_Parameters)
Scheduling_Server_Ref
(from rtmod_Parameters)
Factor_Param
(from rtmod_Parameters)
Percentage_Param
(from rtmod_Parameters)
Laxity_Kind_Param
(from rtmod_Parameters)
Distribution_Type_Param
(from rtmod_Parameters)
Job_Ref_List
(from rtmod_Parameters)
Operation_Ref_List
(from rtmod_Parameters)
Local_Usage
(from rtmod_Usages)
Assertion_Param
(from rtmod_Parameters)
Bit_Count_Param
(from rtmod_Parameters)
Count_Param
(from rtmod_Parameters)
Delivery_Policy_Param
(from rtmod_Parameters)
Request_Policy_Param
(from rtmod_Parameters)


pkg rtcbs_System
Application
(from rtmod_System)
1..*
Application_List
System
(from rtmod_System)
CBS_System
{redefines Application_List}
...
1..*
Application_List
CBS_Application
/software_Module_List:Software_Module_Instance {readOnly, union}
{subset software_Module_List}
connector
*
Software_Connector_Instance
(from rtcbs_Containers)
{subset software_Module_List}
distributedConnector
*
Distributed_Software_Connector_Instance
(from rtcbs_Containers)
{subset software_Module_List}
1..*
instance
Software_Component_Instance
(from rtcbs_Containers)
Cuando se aplica el metamodelo CBS-MAST, el único tipo de System permitido como elemento raíz es CBS_System

Anexo C. Descriptores de la aplicación ScadaDemo

En este anexo se recogen algunos de los ficheros .xml correspondientes al ejemplo expuesto en el capítulo 6 (todos aquellos que se referencian explícitamente en el texto). Todos ellos son descriptores RT-D&C, correspondientes al metamodelo expuesto en el capítulo 3.

ScadaEngine.ccd.xml: Interfaz del componente ScadaEngine

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCcdm:componentInterfaceDescription ...
   xsi:schemaLocation="http://mast.unican.es/rtdnc/rtDnC_ComponentDataModel
   ..\..\schemas\rtdnc\rtDnC_ComponentDataModel.xsd">
   <description specificType="components/scada/ScadaEngine.pcd.xml"
      supportedType="components/scada/ScadaEngine.pcd.xml"
      label="ScadaEngine interface description"
      UUID="http://ctr.unican.es/rtccm/scada/ScadaEngine">
         <!-- Business port description -->
         <port name="controlPort"
            specificType="domains/scada/Scada.idl.xml::scada::ScadaControl"
            supportedType="domains/scada/Scada.idl.xml::scada::ScadaControl"
            provider="true"
            kind="FACET">
            <offeredPortRTModel>
               <rtUsage name="getLastLoggedMssg"/>
               <rtUsage name="getBufferedData"/>
            </offeredPortRTModel>
         </port>
         <port name="adqPort"
            specificType="domains/io/io.idl.xml::io::AnalogIO"
            supportedType="domains/io/io.idl.xml::io::AnalogIO"
            provider="false"
            exclusiveUser="false"
            optional="false"
            kind="MULTIPLEXRECEPTACLE">
            <requiredPortRTModel>
               <rtUsage name="aiReadCode"/>
            </requiredPortRTModel>
         </port>
         <port name="logPort"
            specificType="domains/db/db.idl.xml::db::Logging"
            supportedType="domains/db/db.idl.xml::db::Logging"
            provider="false"
            exclusiveUser="false"
            optional="true"
            kind="SIMPLEXRECEPTACLE">
            <requiredPortRTModel>
               <rtUsage name="log"/>
            </requiredPortRTModel>
         </port>
         <!-- Configuration properties -->
         <property name="samplingPeriod" type="FLOAT"
            label="Period used to sample of the supervised signals"/>
         <property name="loggingPeriod" type="FLOAT"
            label="Period used for storage activity"/>
         <!--EndToEnd Flow Declarations-->
         <rtWorkloadEntity name="SamplingTrans"
            label="Periodic activity executed every samplingPeriod for sampling the signals.
```

```
            It is executed in response to a system clock event">
          <property name="samplingOperList" type="USAGE_LIST"
            label="Set of activities executed in the transaction. An instance of the
                 VariableSampling usage must be included for each supervised signal"/>
          <property name="samplingDeadline" type="DEADLINE"
            label="Hard global deadline imposed on each transaction execution"/>
        </rtWorkloadEntity>
        <rtUsageEntity name="VariableSampling">
          <property name="adqCard" type="USED_PORT"
            label="Concrete receptacle used for acquiring the signal value"/>
        </rtUsageEntity>
        <rtWorkloadEntity name="LoggingTrans"
          label="Periodic activity executed every loggingPeriod to  store
            the statistical values in the component connected to the logPort.
            It is executed in response to a system clock event">
          <property name="loggingDeadline"  type="DEADLINE"
            label="Hard global deadline imposed on each transaction execution"/>
        </rtWorkloadEntity>

    </description>
</DnCcdm:componentInterfaceDescription>
```

AdaScadaEngine.cid.xml: Implementación AdaScadaEngine (implementación de la interfaz ScadaEngine para Ada-CCM)

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCcdm:componentImplementationDescription ...
 xsi:schemaLocation="http://mast.unican.es/rtdnc/rtDnC_ComponentDataModel
..\..\schemas\rtdnc\rtDnC_ComponentDataModel.xsd">
  <description label="Implementacion Ada para MaRTE OS del componente ScadaEngine"
    UUID="http://ctr.unican.es/rtccm/scada/AdaScadaEngine">
    <implements>
      <ref>components/scada/ScadaEngine.pcd.xml</ref>
    </implements>
    <!--MonolithicImplementationDescription-->
    <monolithicImpl rtModel="components/scada/AdaScadaEngine.rtm.xml">
      <primaryArtifact name="ADA_ScadaEngine">
        <referencedArtifact>
          <description location="c:/adaccm/rtccm/repository/components/scada/
            AdaScadaEngine.a">
            <deployRequirement name="run-time_Enviroment_Requirement"
              resourceType="run-time_Enviroment">
              <property name="type">
                <value> gtk-gpl-2[2].8.0-nt.exe</value>
              </property>
              <property name="version">
                <value>8.0</value>
              </property>
              <property name="provider">
                <value>AdaCore</value>
              </property>
            </deployRequirement>
          </description>
        </referencedArtifact>
      </primaryArtifact>
    </monolithicImpl>
    <!-- Specific properties -->
    <specificProperty name="samplingTh" type="PERIODIC_ACTIVATION"/>
    <specificConfigProperty name="samplingTh.period">
      <refValue property="samplingPeriod"/>
    </specificConfigProperty>
    <specificProperty name="loggingTh" type="PERIODIC_ACTIVATION"/>
    <specificConfigProperty name="loggingTh.period">
      <refValue property="loggingPeriod"/>
    </specificConfigProperty>
    <specificProperty type="MUTEX" name="dataMtx"/>
    <specificProperty type="MUTEX" name="mssgMtx"/>
    <!-- Mapping of ports to Transactions -->
    <transactionToPort transaction="LoggingTrans" activationPort="loggingTh"/>
    <transactionToPort transaction="SamplingTrans" activationPort="samplingTh"/>
```

```
    <!-- Real-Time Properties -->
    <rtProperty type="PRIORITY" name="samplingThPrty" sameValueOf="samplingTh.schedParam"/>
    <rtProperty type="DURATION" name="samplingThPeriod" takeValueOf="samplingPeriod"/>
    <rtProperty type="PRIORITY" name="loggingThPrty" sameValueOf="loggingTh.schedParam"/>
    <rtProperty type="DURATION" name="loggingThPeriod" takeValueOf="loggingPeriod"/>
    <rtProperty type="PRIORITY" name="dataMtxCeiling" sameValueOf="dataMtx.ceiling"/>
    <rtProperty type="PRIORITY" name="mssgMtxCeiling" sameValueOf="mssgMtx.ceiling"/>
  </description>
</DnCcdm:componentImplementationDescription>
```

AdaScadaEngine.rtm.xml: Modelo de tiempo real de la implementación AdaScadaEngine

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOFTWARE_COMPONENT ...
    xsi:schemaLocation="http://mast.unican.es/rtcbs/containers
        ..\..\schemas\rtcbs\rtcbsContainers.xsd">
  <!-- Parameter List -->
  <paramList>
    <param name="samplingThPeriod" type="TIME_INTERVAL"/>
    <param name="samplingThPrty" type="PRIORITY"/>
    <param name="loggingThPeriod" type="TIME_INTERVAL"/>
    <param name="loggingThPrty" type="PRIORITY"/>
    <param name="dataMtxCeiling" type="PRIORITY"/>
    <param name="mssgMtxCeiling" type="PRIORITY"/>
  </paramList>
  <!-- Aggregated element list -->
  <aggrRsrcList>
    <sharedResource name="dataMtx">
      <rtr:immediateCeilingParams preassigned="NO" ceiling="@dataMtxCeiling@"/>
    </sharedResource>
    <sharedResource name="mssgMtx">
      <rtr:immediateCeilingParams preassigned="NO" ceiling="@mssgMtxCeiling@"/>
    </sharedResource>
  </aggrRsrcList>
  <!-- Private element List -->
  <privateElemList>
    <simpleOperation name="storeValue"
        label="Almacena en una variable local el valor leído de la magnitud"
        wcet="1.8E-06" acet="1.3E-06" bcet="1.0E-06"/>
    <simpleOperation name="storeData"
        label="Procesa y almacena el conjunto de valores de magnitudes ya disponibles en
               variables locales"
        sharedResources="dataMtx"
          wcet="12.8E-06" acet="8.3E-06" bcet="5.1E-06"/>
    <simpleOperation name="buildMagnitudeMssgs"
          label="Construye los mensajes que corresponden a las magnitudes supervisadas
          y las almacena en variables locales"
          sharedResources="dataMtx"
          wcet="3.3E-05" acet="2.3E-05" bcet="1.8E-05"/>
    <simpleOperation name="buildLoggedMssg"
          label="Construye y almacena el mensaje que se va a registrar en el logger"
          sharedResources="mssgMtx"
          wcet="10.1E-06" acet="7.5E-06" bcet="6.8E-06"/>
  </privateElemList>
  <!-- Declared Element List -->
  <declElemList>
    <supportedPort name="controlPort"
        label="rt-model of the operation offered through the facet AnalogIO">
        <simpleOperation name="getBufferedData"
          label="Read the value os an input analog line"
          sharedResources="dataMtx"
          wcet="1.8E-06" acet="1.3E-06" bcet="1.0E-06"/>
        <simpleOperation name="getLastLoggedMssg"
          label="Set a value on an output analog line"
          sharedResources="mssgMtx"
          wcet="1.8E-06" acet="1.3E-06" bcet="1.0E-06"/>
    </supportedPort>
    <multipleUsedPort name="adqPort"/>
    <usedPort name="logPort"/>
    <job name="VariableSampling"
```

```
      label="Operación parametrizada que describe la lectura de una magnitud por el puerto
             adqPort">
        <rtuc:paramList>
          <rtuc:param type="USED_PORT" name="adqCard"/>
        </rtuc:paramList>
        <rtuc:event name="e1"/>
        <rtuc:activity name="ReadVariable_act"
             activityServer="@CALLER@"
             activityUsage="@adqCard@.aiReadCode"
             inputEvent="@START@"
             outputEvent="e1"/>
        <rtuc:activity name="RegisterVariable_act"
             activityServer="@CALLER@"
             activityUsage="storeValue"
             inputEvent="e1"
             outputEvent="@END@"/>
  </job>
  <transaction  name="SamplingTrans"
       label="Tarea periodica que consiste en leer los valores de las magnitudes
         supervisadas, procesarlos estadísticamente y almacenarlos en el buffer">
    <rtuc:paramList>
      <rtuc:param name="SamplingOperList" type="JOB_LIST"
           label="Lista de operaciones de lectura de las variables supervisadas. Corresponde
           a una lista de instancias de la operación parametrizada VariableSampling"/>
      <rtuc:param name="SamplingDeadline" type="TIME_INTERVAL"/>
    </rtuc:paramList>
    <rtuc:declaredElemList>
      <rtuc:declCompositeJob name="AllSamplingOper"  jobList="@SamplingOperList@"/>
    </rtuc:declaredElemList>
    <rtuc:aggrRsrcList>
      <rtuc:aggrRegularServer name="samplingTh"
           scheduler="@HOST@.scheduler">
        <rtr:preemptibleFPParams priority="@samplingThPrty@" preassigned="NO"/>
      </rtuc:aggrRegularServer>
    </rtuc:aggrRsrcList>
    <rtuc:externalEvent name="samplingTrigger" >
      <rtw:periodicPattern period="@samplingThPeriod@"/>
    </rtuc:externalEvent>
    <rtuc:event name="endSampling">
      <rtu:globalTimingReqObserver referencedEvent="samplingTrigger">
        <rtw:deadlineRequirement deadline="@samplingDeadline@" kind="HARD"/>
      </rtu:globalTimingReqObserver>
    </rtuc:event>
    <rtuc:event name="e1"/>
    <rtuc:activity name="ReadMagnitudesSet_act"
         activityServer="samplingTh"
         activityUsage="AllSamplingOper"
         inputEvent="samplingTrigger"
         outputEvent="e1"/>
    <rtuc:activity name="StoreAndProcess_act"
         activityServer="samplingTh"
         activityUsage="storeData"
         inputEvent="e1"
         outputEvent="endSampling"/>
  </transaction>
  <transaction  name="LoggingTrans"
       label="Se registran los resultados almacenados y se inicializan los estimadores
              estadísticos">
       <rtuc:paramList>
         <rtuc:param type="TIME_INTERVAL" name="loggingDeadline"/>
       </rtuc:paramList>
    <rtuc:aggrRsrcList>
      <rtuc:aggrRegularServer name="loggingTh"
           scheduler="@HOST@.scheduler">
        <rtr:preemptibleFPParams priority="@loggingThPrty@" preassigned="NO"/>
      </rtuc:aggrRegularServer>
    </rtuc:aggrRsrcList>
    <rtuc:externalEvent name="loggingTrigger" >
      <rtw:periodicPattern period="@loggingThPeriod@"/>
    </rtuc:externalEvent>
```

```
        <rtuc:event name="endLogging"/>
        <rtuc:event name="e1"/>
        <rtuc:event name="e2">
          <rtu:globalTimingReqObserver referencedEvent="loggingTrigger">
            <rtw:deadlineRequirement deadline="@loggingDeadline@" kind="HARD"/>
          </rtu:globalTimingReqObserver>
        </rtuc:event>
        <rtuc:activity name="BuildMagnitudeMssg_act"
             activityServer="loggingTh"
             activityUsage="buildMagnitudeMssgs"
             inputEvent="loggingTrigger"
             outputEvent="e1"/>
        <rtuc:activity name="BuidLoggedMssgList_act"
             activityServer="loggingTh"
             activityUsage="buildLoggedMssg"
             inputEvent="e1"
             outputEvent="e2"/>
        <rtuc:activity name="Logging_act"
             activityServer="loggingTh"
             activityUsage="@logPort@.log"
             inputEvent="e2"
             outputEvent="endLogging"/>
      </transaction>
    </declElemList>
</SOFTWARE_COMPONENT>
```

ScadaDemo.pcd.xml: Descripción de la aplicación ScadaDemo

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<DnCcdm:packageConfiguration ...
xsi:schemaLocation="http://mast.unican.es/rtdnc/rtDnC_ComponentDataModel
..\..\schemas\rtdnc\rtDnC_ComponentDataModel.xsd">
<!-- Component Package Description -->
<basePackage>
  <!-- ComponentInterfaceDescription-->
  <realizes>
    <description label="Descripcion de la interfaz de la aplicación ScadaDemo"
              UUID="http://ctr.unican.es/applications/scada/ScadaDemo"
              specificType="applications/scada/ScadaDemo.pcd.xml"/>
  </realizes>
  <!-- ComponentImplementationDescription-->
  <implementation name="ScadaDemo">
    <referencedImplementation>
      <description>
        <implements><ref>applications/scada/ScadaDemo.pcd.xml</ref></implements>
        <assemblyImpl>
        <!-- Instances declaration-->
          <instance>
            <description name="manager">
              <referencedPackage
                requiredUUID="http://ctr.unican.es/rtccm/scada ScadaManager"/>
              <configProperty name="displayPeriod">
                <value><float>1.0</float></value>
              </configProperty>
            </description>
          </instance>
          <instance>
            <description name="engine">
              <referencedPackage
                requiredUUID="http://ctr.unican.es/rtccm/scada/ScadaEngine"/>
                <configProperty name="samplingPeriod">
                  <value><float>0.01</float></value>
                </configProperty>
                <configProperty name="loggingPeriod">
                  <value><float>0.1</float></value>
                </configProperty>
            </description>
          </instance>
          <instance>
            <description name="sensorA">
```

```
                <referencedPackage requiredUUID="http://ctr.unican.es/rtccm/io/IOCard"/>
                <configProperty name="cardID">
                  <value><string>0</string></value>
                </configProperty>
                <configProperty name="aiEnded">
                  <value><string>SINGLE_ENDED</string></value>
                </configProperty>
                <configProperty name="daRange">
                  <value><string>UNIPOLAR_5V</string></value>
                </configProperty>
              </description>
            </instance>
            <instance>
              <description name="sensorB">
                <referencedPackage requiredUUID="http://ctr.unican.es/rtccm/io/IOCard"/>
                <configProperty name="cardID">
                  <value><string>0</string></value>
                </configProperty>
                <configProperty name="aiEnded">
                  <value><string>SINGLE_ENDED</string></value>
                </configProperty>
                <configProperty name="daRange">
                  <value><string>UNIPOLAR_10V</string></value>
                </configProperty>
              </description>
            </instance>
            <instance>
              <description name="terminal">
                <referencedPackage
                  requiredUUID="http://ctr.unican.es/rtccm/console/TextConsole"/>
              </description>
            </instance>
            <instance>
              <description name="register">
                <referencedPackage requiredUUID="http://ctr.unican.es/rtccm/db/Logger"/>
              </description>
            </instance>
            <!-- Connections declaration-->
            <connection name="manager_scadaPortToengine_controlPort">
              <internalEndpoint instance="manager" portName="scadaPort"/>
              <internalEndpoint instance="engine" portName="controlPort"/>
            </connection>
            <connection name="manager_consolePortToterminal_rwPort">
              <internalEndpoint instance="manager" portName="consolePort"/>
              <internalEndpoint instance="terminal" portName="rwPort"/>
            </connection>
            <connection name="engine_adqPortToinnerSensor_analogPort">
              <internalEndpoint instance="engine" portName="adqPort"/>
              <internalEndpoint instance="innerSensor" portName="analogPort"/>
            </connection>
            <connection name="engine_adqPortTosensor_analogPort">
              <internalEndpoint instance="engine" portName="adqPort"/>
              <internalEndpoint instance="sensor" portName="analogPort"/>
            </connection>
            <connection name="engine_logPortPorToregister_regPort">
              <internalEndpoint instance="engine" portName="logPort"/>
              <internalEndpoint instance="register" portName="regPort"/>
            </connection>
          </assemblyImpl>
        </description>
      </referencedImplementation>
      </implementation>
    </basePackage>
</DnCcdm:packageConfiguration>
```

ScadaDemoWorkload.wld.xml: Carga de trabajo de la aplicación ScadaDemo

```
<?xml version="1.0" encoding="UTF-8"?>
<rtWkld:applicationWorkload ..
```

```
  xsi:schemaLocation="http://mast.unican.es/rtdnc/rtWorkload
..\..\schemas\rtdnc\rtDnC_WorkloadModel.xsd">
  <workloadInstance name="BasicWorkload">
    <endToEndFlow instanceName="loggingTransInst" description="LoggingTrans" name="engine">
      <rtConfigProperty name="loggingDeadline">
        <value><timeInterval>0.1</timeInterval></value>
      </rtConfigProperty>
    </endToEndFlow>
    <endToEndFlow name="displayTransInst" instanceName="manager" description="DisplayTrans">
      <rtConfigProperty name="displayDeadline">
        <value><deadline>1.0</deadline></value>
      </rtConfigProperty>
    </endToEndFlow>
    <endToEndFlow name="changeTransInst" instanceName="manager" description="ChangeTrans">
      <rtConfigProperty name="changeInterarrivalTime">
        <value><deadline>0.5</deadline></value>
      </rtConfigProperty>
      <rtConfigProperty name="changeDeadline">
        <value><deadline>0.5</deadline></value>
      </rtConfigProperty>
    </endToEndFlow>
    <endToEndFlow name="checkingTransInst" instanceName="manager" description="CheckingTrans">
      <rtConfigProperty name="checkingInterarrivalTime">
        <value><deadline>0.5</deadline></value>
      </rtConfigProperty>
      <rtConfigProperty name="checkingDeadline">
        <value><deadline>0.5</deadline></value>
      </rtConfigProperty>
    </endToEndFlow>
  </workloadInstance>

  <workloadInstance name="TwoSupervisions" base="BasicWorkload">
    <usage name="Sampling1" description="VariableSampling" instanceName="engine">
      <rtConfigProperty name="adqCard">
        <value><componentPort instanceName="sensorA" portName="analogPort"/></value>
      </rtConfigProperty>
    </usage>
    <usage name="Sampling2" description="VariableSampling" instanceName="engine">
      <rtConfigProperty name="adqCard">
        <value><componentPort instanceName="sensorB" portName="analogPort"/></value>
      </rtConfigProperty>
    </usage>
    <endToEndFlow name="samplingTransInst" description="SamplingTrans" instanceName="engine">
      <rtConfigProperty name="samplingOperList">
           <value><usageList>
             <usage>Sampling1</usage>
             <usage>Sampling2</usage>
             </usageList>
           </value>
      </rtConfigProperty>
      <rtConfigProperty name="samplingDeadline">
           <value><deadline>0.005</deadline></value>
      </rtConfigProperty>
    </endToEndFlow>
  </workloadInstance>

  <workloadInstance name="ThreeSupervisions" base="BasicWorkload">
    <usage name="Sampling1" description="VariableSampling" instanceName="engine">
      <rtConfigProperty name="adqCard">
        <value><componentPort instanceName="sensorA" portName="analogPort"/></value>
      </rtConfigProperty>
    </usage>
    <usage name="Sampling2" description="VariableSampling" instanceName="engine">
      <rtConfigProperty name="adqCard">
        <value><componentPort instanceName="sensorB" portName="analogPort"/></value>
      </rtConfigProperty>
    </usage>
    <usage name="Sampling3" description="VariableSampling" instanceName="engine">
      <rtConfigProperty name="adqCard">
        <value><componentPort instanceName="sensorA" portName="analogPort"/></value>
```

```
      </rtConfigProperty>
    </usage>
    <endToEndFlow name="samplingTransInst" description="SamplingTrans" instanceName="engine">
      <rtConfigProperty name="samplingOperList">
          <value><usageList>
            <usage>Sampling1</usage>
            <usage>Sampling2</usage>
            <usage>Sampling3</usage>
            </usageList>
          </value>
      </rtConfigProperty>
      <rtConfigProperty name="samplingDeadline">
          <value><deadline>0.005</deadline></value>
      </rtConfigProperty>
    </endToEndFlow>
  </workloadInstance>
</rtWkld:applicationWorkload>
```

ScadaDemoDomain.tdm.xml: Descripción de la plataforma de ejecución

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<DnCtdm:domain ...
  xsi:schemaLocation="http://mast.unican.es/rtdnc/rtDnC_TargetDataModel
  ..\..\schemas\rtdnc\rtDnC_TargetDataModel.xsd">
  <!-- Node declaration-->
  <node name="CentralProc" source="platforms/MaRTEBaseProcessor.ndd.xml">
    <resource name="ADQCard0" resourceType="PCI9111"/>
    <resource resourceType="CommunicationDriver" name="rtep_support"
      source="platforms/marteEnvironment/MaRTEBaseProcessor.ndd.xml::rtepDriver">
      <rtConfigProperty name="network">
        <value><communicationNetwork>EthernetNet</communicationNetwork></value>
      </rtConfigProperty>
      <rtConfigProperty name="numStations">
        <value><positive>2</positive></value>
      </rtConfigProperty>
    </resource>
    <supportedConnection name="RTEPConnection"
      source="platforms/marteEnvironment/adaccm/AdaCCMRTEPConnection.cmd.xml"/>
    <supportedConnection name="LocalConnection"
      source="platforms/marteEnvironment/adaccm/AdaCCMLocalConnection.cmd.xml"/>
    <rtConfigProperty name="speedFactor"><value><float>2.0</float></value></rtConfigProperty>
    <connection>EthernetNet</connection>
  </node>
  <node name="RemoteProc" source="MaRTEBaseProcessorDescription">
    <resource name="ADQCard1" resourceType="PCI9111"/>
    <resource resourceType="CommunicationDriver" name="rtep_support"
      source="platforms/marteEnvironment/MaRTEBaseProcessor.ndd.xml::rtepDriver">
      <rtConfigProperty name="network">
        <value><communicationNetwork>EthernetNet</communicationNetwork></value>
      </rtConfigProperty>
      <rtConfigProperty name="numStations">
        <value><positive>2</positive></value>
      </rtConfigProperty>
    </resource>
    <supportedConnection name="RTEPConnection"
      source="platforms/marteEnvironment/adaccm/AdaCCMRTEPConnection.cmd.xml"/>
    <supportedConnection name="LocalConnection"
      source="platforms/marteEnvironment/adaccm/AdaCCMLocalConnection.cmd.xml"/>
    <rtConfigProperty name="speedFactor"><value><float>1.0</float></value></rtConfigProperty>
    <connection>EthernetNet</connection>
  </node>
  <!-- Interconnect declaration-->
  <interconnect name="EthernetNet"
    source="platforms/marteEnvironment/MaRTEEthernetNetwork.icd.xml">
    <rtConfigProperty name="speedFactor"><value><float>1.0</float></value></rtConfigProperty>
    <connect>CentralProc</connect>
    <connect>RemoteProc</connect>
  </interconnect>
</DnCtdm:domain>
```

Node_MaRTEOS_PC_750MHz.rtm.xml: Modelo de tiempo real de los nodos de la plataforma

```
<?xml version="1.0" encoding="UTF-8"?>
<PROCESSING_NODE_RTM ...
   xsi:schemaLocation="http://mast.unican.es/rtmod/containers
   ..\..\schemas\rtmod\rtmContainers.xsd>
   <!-- Parameters declaration-->
   <rtc:paramList>
      <rtc:param name="speedFactor" type="FACTOR"  value="1.0"/>
   </rtc:paramList>
   <!-- Aggregated resources declaration-->
   <rtc:aggrRsrcList>
      <rtc:processor name="processor" speedFactor="@speedFactor@">
         <rtr:alarmClockSystemTimer>
            <rtr:overheadOperation wcet="1.5E-6" acet="1.5E-6" bcet="1.5E-6"/>
         </rtr:alarmClockSystemTimer>
      </rtc:processor>
      <scheduler host="processor" name="scheduler">
         <rtr:fpPolicy minPriority="1" maxPriority="31">
            <rtr:contextSwitchOperation wcet="2.2E-6" acet="2.2E-6" bcet="2.2E-6"/>
         </rtr:fpPolicy>
      </scheduler>
      <rtc:interruptScheduler host="processor" name="interruptScheduler">
         <rtr:interruptFPPolicy minPriority="32" maxPriority="32">
         <rtr:contextSwitchOperation wcet="2.8E-06" acet="2.8E-06" bcet="2.8E-06"/>
      </rtr:interruptFPPolicy>
      </rtc:interruptScheduler>
   </rtc:aggrRsrcList>
   <!-- Declared elements declaration-->
   <rtc:declElemList>
      <rtc:rtepPacketDriver name="RTEPDriverMarteOS"
         numberOfStations="@numStations@" label="Driver RT-EP para PC sobre MarteOS"
         tokenDelay="80.0E-6" failureTimeout="250E-6" tokenTransmissionRetries="1"
         packetTransmissionRetries="1" messagePartitioning="NO" rtaOverheadModel="DECOUPLED"
         network="@network@">
         <rtr:paramList>
            <rtr:param name="numStations" type="COUNT"/>
            <rtr:param name="network" type="NETWORK"/>
         </rtr:paramList>
         <rtr:packetSendOperation wcet="32.39E-06" acet="19.27E-06" bcet="17.79E-06"/>
         <rtr:packetReceiveOperation wcet="46.20E-06" acet="18.68E-06" bcet="16.98E-06"/>
         <rtr:packetISROperation wcet="6.44E-06" acet="3.720E-06" bcet="2.4800E-06" />
         <rtr:tokenCheckOperation wcet="22.92E-06" acet="18.06E-06" bcet="14.96E-06"/>
         <rtr:tokenManageOperation wcet="20.50E-06" acet="10.34E-06" bcet="9.68E-06"/>
         <rtr:packetDiscardOperation wcet="12.69E-06" acet="3.87E-06" bcet="3.68E-06"/>
         <rtr:tokenRetransmissionOperation wcet="33.19E-6" acet="14.21E-6" bcet="13.36E-6"/>
         <rtr:packetRetransmissionOperation wcet="32.39E-6"acet="19.27E-6" bcet="17.09E-6"/>
         <rtr:packetServer scheduler="scheduler">
            <rtr:nonPreemptibleFPParams priority="31" preassigned="YES"/>
         </rtr:packetServer>
         <rtr:packetInterruptServer scheduler="scheduler">
            <rtr:nonPreemptibleFPParams priority="32" preassigned="YES"/>
         </rtr:packetInterruptServer>
      </rtc:rtepPacketDriver>
      <packetDriver name="CANDriverMarteOS" label="Driver CAN para PC sobre MaRTEOS"
         messagePartitioning="YES" rtaOverheadModel="COUPLED" network="@network@">
         <rtr:paramList>
            <rtr:param name="network" type="NETWORK"/>
         </rtr:paramList>
         <rtr:packetSendOperation wcet="1.0E-05" acet="1.0E-05" bcet="1.0E-05"/>
         <rtr:packetReceiveOperation wcet="1.5E-05" acet="1.5E-05" bcet="1.5E-05"/>
         <rtr:packetInterruptServer>
            <rtr:preemptibleFPParams priority="32" preassigned="YES"/>
         </rtr:packetInterruptServer>
      </packetDriver>
   </rtc:declElemList>
</PROCESSING_NODE_RTM>
```

ScadaDemo.cdp.xml: Plan de despliegue inicial de la aplicación

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCedm:deploymentPlan ...
     xsi:schemaLocation="http://mast.unican.es/rtdnc/rtDnC_ExecutionDataModel
        ..\..\schemas\rtdnc\rtDnC_ExecutionDataModel.xsd"
     rtWorkloadModel="applications/scada/ScadaDemoWorkload.rtw.xml">
   <!-- Implemented interface -->
   <realizes>
     <ref>applications/scada/ScadaDemo.pcd</ref>
   </realizes>
   <!-- Instances -->
   <instance name="manager" node="CentralProc" implementation="AdaScadaManager"
     source="components/scada/ScadaManager.pcd.xml::AdaScadaManager">
     <!-- Business Configuration Properties-->
     <configProperty name="displayPeriod">
          <value><float>1.0</float></value>
     </configProperty>
     <!-- Mandatory specific properties assignment-->
     <specificConfigProperty name="keyboardTh">
       <value><oneShotActivation stimId="1"/></value>
     </specificConfigProperty>
     <specificConfigProperty name="displayTh">
       <value><periodicActivation stimId="2" period="1.0"/></value>
     </specificConfigProperty>
     <specificConfigProperty name="changeTransId">
       <value><transactionIds>3</transactionIds></value>
     </specificConfigProperty>
     <specificConfigProperty name="checkingTransId">
       <value><transactionIds>4</transactionIds></value>
     </specificConfigProperty>
   </instance>
   <instance name="engine" node="CentralProc" implementation="AdaScadaEngine"
     source="components/scadal/ScadaEngine.pcd.xml::AdaScadaEngine">
     <configProperty name="samplingPeriod">
       <value><float>0.05</float></value>
     </configProperty>
     <configProperty name="loggingPeriod">
       <value><float>0.1</float></value>
     </configProperty>
     <specificConfigProperty name="loggingTh">
       <value><periodicActivation stimId="5" period="0.1"/></value>
     </specificConfigProperty>
     <specificConfigProperty name="samplingTh">
       <value><periodicActivation stimId="6" period="0.005"/></value>
     </specificConfigProperty>
   </instance>
   <instance name="terminal" node="CentralProc" implementation="AdaMaRTETextConsole"
     source="components/console/TextConsole.pcd.xml::AdaMaRTETextConsole">
   </instance>
   <instance name="register" node="CentralProc" implementation="AdaRAMLogger"
     source="components/db/Logger.pcd.xml::RAMLogger">
     <specificConfigProperty name="notifyTh">
       <value><oneShotActivation stimId="7"/></value>
     </specificConfigProperty>
   </instance>
   <instance name="sensorA"node="CentralProc" implementation="VirtualIOCard"
     source="components/io/AnalogIOCard.pcd.xml::VirtualIOCardImpl">
     <configProperty name="cardID">
       <value><string>0</string></value>
     </configProperty>
     <configProperty name="aiEnded">
       <value><string>SINGLE_ENDED</string></value>
     </configProperty>
     <configProperty name="daRange">
       <value><string>UNIPOLAR_5V</string></value>
     </configProperty>
   </instance>
   <instance name="sensor" node="RemoteProc" implementation="AdaPCICard"
     source="components/io/AnalogIOCard.pcd.xml::AdaPCI9111IOCardImpl">
```

```
      <configProperty name="cardID">
        <value><string>0</string></value>
      </configProperty>
      <configProperty name="aiEnded">
        <value><string>SINGE_ENDED</string></value>
      </configProperty>
      <configProperty name="daRange">
        <value><string>UNIPOLAR_10V</string></value>
      </configProperty>
    </instance>
    <!-- Connections -->
    <connection name="manager_scadaPortToengine_controlPort">
      <internalEndpoint instance="manager" portName="scadaPort" provider="false"/>
      <internalEndpoint instance="engine" portName="controlPort" provider="true"/>
    </connection>
    <connection name="manager_consolePortToterminal_rwPort">
      <internalEndpoint instance="manager" portName="consolePort" provider="false"/>
      <internalEndpoint instance="terminal" portName="rwPort" provider="true"/>
    </connection>
    <connection name="engine_adqPortToinnersensorA_analogPort">
      <internalEndpoint instance="engine" portName="adqPort" provider="false"/>
      <internalEndpoint instance="sensorA" portName="analogPort" provider="true"/>
    </connection>
    <connection name="engine_adqPortTosensorB_analogPort">
      <internalEndpoint instance="engine" portName="adqPort" provider="false"/>
      <internalEndpoint instance="sensorB" portName="analogPort" provider="true"/>
      <connectionConfiguration communicationMechanism="CentralProc.RTEPConnection">
        <configProperty name="channelId">
          <value>1</value>
        </configProperty>
      </connectionConfiguration>
    </connection>
    <connection name="engine_logPortPorToregister_regPort">
      <internalEndpoint instance="engine" portName="logPort" provider="false"/>
      <internalEndpoint instance="register" portName="regPort" provider="true"/>
    </connection>
    <!-- Implementations descriptions -->
    <implementation name="AdaScadaManager" artifact="AdaScadaManagerArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/scada/AdaScadaManager.a"
              name="AdaScadaManagerArtifact"/>
    <implementation name="AdaScadaEngine" artifact="AdaScadaEngineArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/scada/AdaScadaEngine.a"
              name="AdaScadaEngineArtifact"/>
    <implementation name="AdaPCI9111IOCard" artifact="AdaPCI9111IOCardArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/io/AdaPCI9111IOCard.a"
              name="AdaPCI9111IOCardArtifact"/>
    <implementation  name="AdaTextConsole" artifact="AdaMaRTETextConsoleArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/console/AdaMaRTETextConsole.a"
              name="AdaMaRTETextConsoleArtifact"/>
    <implementation   name="AdaRAMLogger" artifact="AdaMaRTERAMLoggerArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/console/AdaRAMLogger.a"
              name="AdaMaRTERAMLoggerArtifact"/>
</DnCedm:deploymentPlan>
```

ScadaDemoFinal.cdp.xml: Plan de despliegue final de la aplicación

```
<?xml version="1.0" encoding="UTF-8"?>
<DnCedm:deploymentPlan ...
   xsi:schemaLocation="http://mast.unican.es/rtdnc/rtDnC_ExecutionDataModel
   ..\..\schemas\rtdnc\rtDnC_ExecutionDataModel.xsd"
   rtWorkloadModel="applications/scada/ScadaDemoWorkload.rtw.xml">
   <!-- Implemented interface -->
     <realizes>
       <ref>applications/scada/ScadaDemo.pcd</ref>
     </realizes>
   <!-- Interfaces -->
   <instance name="manager" node="CentralProc" implementation="AdaScadaManager"
        source="components/scada/ScadaManager.pcd.xml::AdaScadaManager">
        <configProperty name="displayPeriod">
          <value><float>1.0</float></value>
```

```
      </configProperty>
      <specificConfigProperty name="keyboardTh">
        <value><oneShotActivation stimId="1"/></value>
      </specificConfigProperty>
      <specificConfigProperty name="displayTh">
        <value><periodicActivation stimId="2" period="1.0"/></value>
      </specificConfigProperty>
      <specificConfigProperty name="changeTransId">
        <value><transactionIds>3</transactionIds></value>
      </specificConfigProperty>
      <specificConfigProperty name="checkingTransId">
        <value><transactionIds>4</transactionIds></value>
      </specificConfigProperty>
      <specificConfigProperty name="displayMutex">
        <value><ceilingMutex ceiling="2"/></value>
      </specificConfigProperty>
</instance>
<instance name="engine" node="CentralProc" implementation="AdaScadaEngine"
     source="components/scadal/ScadaEngine.pcd.xml::AdaScadaEngine">
   <configProperty name="samplingPeriod">
     <value><float>0.005</float></value>
   </configProperty>
   <configProperty name="loggingPeriod">
     <value><float>0.1</float></value>
   </configProperty>
   <specificConfigProperty name="loggingTh">
     <value><periodicActivation stimId="5" period="0.1"/></value>
   </specificConfigProperty>
   <specificConfigProperty name="samplingTh">
     <value><periodicActivation stimId="6" period="0.005"/></value>
   </specificConfigProperty>
   <specificConfigProperty name="dataMtx">
     <value><ceilingMutex ceiling="27"/></value>
   </specificConfigProperty>
   <specificConfigProperty name="mssgMtx">
     <value><ceilingMutex ceiling="18"/></value>
   </specificConfigProperty>
</instance>
<instance name="terminal" node="CentralProc" implementation="AdaMaRTETextConsole"
     source="components/console/TextConsole.pcd.xml::AdaMaRTETextConsole">
</instance>
<instance name="register" node="CentralProc" implementation="AdaRAMLogger"
     source="components/db/Logger.pcd.xml::RAMLogger">
   <specificConfigProperty name="notifyTh">
     <value><oneShotActivation stimId="7"/></value>
   </specificConfigProperty>
   <specificConfigProperty name="logMutex">
     <value><ceilingMutex ceiling="14"/></value>
   </specificConfigProperty>
   <specificConfigProperty name="logCV">
     <value><ceilingMutex ceiling="12"/></value>
   </specificConfigProperty>
</instance>
   <instance name="sensorA" node="CentralProc" implementation="AdaPCICard"
     source="components/io/AnalogIOCard.pcd.xml::VirtualIOCardImpl">
   <configProperty name="cardID">
     <value><string>0</string></value>
   </configProperty>
   <configProperty name="aiEnded">
     <value><string>SINGLE_ENDED</string></value>
   </configProperty>
   <configProperty name="daRange">
     <value><string>UNIPOLAR_5V</string></value>
   </configProperty>
   <specificConfigProperty name="aiMutex">
     <value><ceilingMutex ceiling="21"/></value>
   </specificConfigProperty>
   <specificConfigProperty name="aoMutex">
     <value><ceilingMutex ceiling="1"/></value>
   </specificConfigProperty>
```

```
    <specificConfigProperty name="doMutex">
      <value><ceilingMutex ceiling="1"/></value>
    </specificConfigProperty>
  </instance>
  <instance name="sensorB" node="RemoteProc" implementation="AdaPCICard"
      source="components/io/AnalogIOCard.pcd.xml::AdaPCI9111IOCardImpl">
    <configProperty name="cardID">
      <value><string>0</string></value>
    </configProperty>
    <configProperty name="aiEnded">
      <value><string>SINGE_ENDED</string></value>
    </configProperty>
    <configProperty name="daRange">
      <value><string>UNIPOLAR_10V</string></value>
    </configProperty>
    <specificConfigProperty name="aiMutex">
      <value><ceilingMutex ceiling="1"/></value>
    </specificConfigProperty>
    <specificConfigProperty name="aoMutex">
      <value><ceilingMutex ceiling="1"/></value>
    </specificConfigProperty>
    <specificConfigProperty name="doMutex">
      <value><ceilingMutex ceiling="1"/></value>
    </specificConfigProperty>
  </instance>
  <!-- Connections -->
  <connection name="manager_scadaPortToengine_controlPort">
    <internalEndpoint instance="manager" portName="scadaPort" provider="false"/>
    <internalEndpoint instance="engine" portName="controlPort" provider="true"/>
    <connectionConfiguration communicationMechanism="CentralProc.LocalConnection">
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="getLastLoggedMssg">
            <schedData inputStimId="2" outputStimId="8"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="getBufferedData">
            <schedData inputStimId="4" outputStimId="9"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
    </connectionConfiguration>
  </connection>
  <connection name="manager_consolePortToterminal_rwPort">
    <internalEndpoint instance="manager" portName="consolePort" provider="false"/>
    <internalEndpoint instance="terminal" portName="rwPort" provider="true"/>
    <connectionConfiguration communicationMechanism="CentralProc.LocalConnection">
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="readKey">
            <schedData inputStimId="1" outputStimId="15"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="writeString">
            <schedData inputStimId="2" outputStimId="13"/>
            <schedData inputStimId="4" outputStimId="14"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
    </connectionConfiguration>
  </connection>
  <connection name="engine_adqPortTosensorA_analogPort">
    <internalEndpoint instance="engine" portName="adqPort" provider="false"/>
    <internalEndpoint instance="sensorA" portName="analogPort" provider="true"/>
```

```
    <connectionConfiguration communicationMechanism="CentralProc.LocalConnection">
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="aiReadCode">
            <schedData inputStimId="6" outputStimId="10"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
    </connectionConfiguration>
  </connection>
  <connection name="engine_logPortPorToregister_regPort">
    <internalEndpoint instance="engine" portName="logPort" provider="false"/>
    <internalEndpoint instance="register" portName="regPort" provider="true"/>
    <connectionConfiguration communicationMechanism="CentralProc.LocalConnection">
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="log">
            <schedData inputStimId="5" outputStimId="12"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
    </connectionConfiguration>
  </connection>
  <connection name="engine_adqPortTosensorB_analogPort">
    <internalEndpoint instance="engine" portName="adqPort" provider="false"/>
    <internalEndpoint instance="sensor" portName="analogPort" provider="true"/>
    <connectionConfiguration communicationMechanism="CentralProc.RTEPConnection">
      <configProperty name="execDispatchPriority">
        <value>16</value>
      </configProperty>
      <configProperty name="channelId">
        <value>1</value>
      </configProperty>
      <qosConfigProperty>
        <value>
          <schedulingConfiguration operation="aiReadCode">
            <schedData inputStimId="6" outputStimId="11"/>
          </schedulingConfiguration>
        </value>
      </qosConfigProperty>
    </connectionConfiguration>
  </connection>
  <!-- Environment services configuration -->
  <qosConfiguration node="CentralProc">
    <serviceConfig name="ThreadingService">
      <threadingServiceConfig poolSize="5" poolPriority="31"/>
    </serviceConfig>
    <serviceConfig name="SchedulingService">
      <schedulingServiceConfig>
        <schedParamAssignment stimulusId="1" priority="23"/>
        <schedParamAssignment stimulusId="2" priority="1"/>
        <schedParamAssignment stimulusId="3" priority="5"/>
        <schedParamAssignment stimulusId="4" priority="2"/>
        <schedParamAssignment stimulusId="5" priority="18"/>
        <schedParamAssignment stimulusId="6" priority="27"/>
        <schedParamAssignment stimulusId="7" priority="14"/>
        <schedParamAssignment stimulusId="8" priority="8"/>
        <schedParamAssignment stimulusId="9" priority="15"/>
        <schedParamAssignment stimulusId="10" priority="21"/>
        <schedParamAssignment stimulusId="12" priority="12"/>
        <schedParamAssignment stimulusId="13" priority="1"/>
        <schedParamAssignment stimulusId="14" priority="2"/>
        <schedParamAssignment stimulusId="15" priority="17"/>
      </schedulingServiceConfig>
    </serviceConfig>
    <serviceConfig name="CommunicationService">
      <communicationServiceConfig>
        <schedParamAssignment stimulusId="11">
          <rtepParams channelId="2" MssgPriortiy="1"/>
        </schedParamAssignment>
```

```
        </communicationServiceConfig>
      </serviceConfig>
    </qosConfiguration>
    <qosConfiguration node="RemoteProc">
      <serviceConfig name="ThreadingService">
        <threadingServiceConfig poolSize="2" poolPriority="31"/>
      </serviceConfig>
      <serviceConfig name="SchedulingService">
        <schedulingServiceConfig>
          <schedParamAssignment stimulusId="11" priority="1"/>
        </schedulingServiceConfig>
      </serviceConfig>
      <serviceConfig name="CommunicationService">
        <communicationServiceConfig>
          <schedParamAssignment stimulusId="11">
            <rtepParams channelId="2" MssgPriortiy="128"/>
          </schedParamAssignment>
        </communicationServiceConfig>
      </serviceConfig>
    </qosConfiguration>
    <!-- Implementations descriptions -->
    <implementation name="AdaScadaManager" artifact="AdaScadaManagerArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/scada/AdaScadaManager.a"
      name="AdaScadaManagerArtifact"/>
    <implementation name="AdaScadaEngine" artifact="AdaScadaEngineArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/scada/AdaScadaEngine.a"
      name="AdaScadaEngineArtifact"/>
    <implementation name="AdaPCI9111IOCard" artifact="AdaPCI9111IOCardArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/io/AdaPCI9111IOCard.a"
      name="AdaPCI9111IOCardArtifact"/>
    <implementation  name="AdaTextConsole" artifact="AdaMaRTETextConsoleArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/console/AdaMaRTETextConsole.a"
      name="AdaMaRTETextConsoleArtifact"/>
    <implementation   name="AdaRAMLogger" artifact="AdaMaRTERAMLoggerArtifact"/>
    <artifact location="c:/adaccm/rtccm/repository/components/console/AdaRAMLogger.a"
      name="AdaMaRTERAMLoggerArtifact"/>
</DnCedm:deploymentPlan>
```

Bibliografía

[ACF07] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, M. Tivoli, "The SAVE approach to component-based development of vehicular systems", Journal of Systems and Software, vol 80, Elsevier, May, 2007.

[ADA95] "Ada 95 Language Reference Manual", Intermetrics, Inc., Cambridge, Mass., January, 1995.

[ADA05] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy (Eds.), "Ada 2005 Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1". LNCS 4348, Springer, 2006.

[AG01] M. Aldea and M. González, "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications", Proc. of the Intl. Conf. on Reliable Software Technologies, Ada-Europe 2001, Leuven, Belgium, Springer LNCS 2043, May 2001.

[AGB06] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. González Harbour, G. Guidi, J.J. Gutiérrez, T. Lennvall, G. Lipari, J.M. Martínez, J.L. Medina, J.C. Palencia, M. Trimarchi, "FSF: A Real-Time Scheduling Architecture Framework", 12th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2006, San Jose (CA, USA), IEEE, April, 2006.

[AMH05] M. Åkerholm, A. Möller, H. Hansson y M. Nolin, "Towards a Dependable Component Technology for Embedded System Applications", Proc. 10th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems, 2005.

[AMS01] Object Management Group, "Ada Language Mapping Specification" - Version 1.2, OMG Document Number: formal/01-10-42, October 2001.

[ASZ08] C. Angelov, K. Sierszecki y F. Zhou, "A Software Framework for Hard Real-Time Distributed Embedded Systems", Proc. 34th Euromicro Conference Software Engineering and Advanced Applications, Parma, August 2008.

[AUD91] N.C. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times", Dept. Computer Science, University of York, December 1991.

[AUTO] AUTOSAR. Consortium, web page, www.autosar.org.

[BS88] T.P. Baker, A. Shaw, "The Cyclic Executive Model and Ada". Proceedings of the IEEE Real-Time Systems Symposium, December 1988.

[BAR07] J. Barnes, "Ada 2005 Rationale: The Language, The Standard Libraries", Lecture Notes In Computer Science; Vol. 5020, John Barnes Informatics.

[BAK91] T.P. Baker: "Stack-based Scheduling of real-time processes" Journal of Real-Time Systems, 3, 1991.

[BB05] L. Bulej and T. Bures, “Addressing Heterogeneity in OMG D&C-based Deployment”, Tech. Report No. 2004/7, Dep. of SW Engineering, Charles University, Prague, Nov 2004.

[BBB02] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, R. Seacord, K. Wallnau, “Volume II: Technical Concepts of Component Based Software Engineering”, 2nd Edition, Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, September 2002.

[BBS06] A. Basu, M. Bozga and J. Sifakis, “Modeling Heterogeneous Real-time Components in BIP”, 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), Invited talk, September, 2006.

[BCK98] L. Bass, P. Clements y R. Kazmena, “Software Architecture in Practice”, Reading Mass.:Addison-Wesley, 1998.

[BCW06] E. Bondarev, P.de With y M. Chaudron, “Compositional Performance Analysis of Component-Based Systems on Heterogeneous Multiprocessor Platforms”, Proc. 32th. Euromicro Conference on Software Engineering and Advanced Applications, Croatia, 2006.

[BCW04] E. Bondarev, P.de With y M. Chaudron, “Predicting Real-Time Properties of Component-Based Applications”, Proc. 30th. Euromicro Conference on Software Engineering and Advanced Applications, 2004.

[BCW07] E. Bondarev, P.de With y M. Chaudron, “CARAT: a Toolkit for Design and Performance Analysis of Component-Based Embedded Systems”, Design Automation and Test in Europe Conference, 2007.

[BGM06] S. Becker, L. Grunske, R. Mirandola and S.Overhage “Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective”, LNCS 3938, 2006, 169-192.

[BGO04] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, “The IF toolset”, in SFM, 2004.

[BHP06] T. Bureš, P. Hnetynka and F. Plášil, “SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model”, Proc. of SERA 2006, Seattle, USA, IEEE CS, Aug 2006.

[BKR07] S. Becker, H. Koziolek y R. Reussner, “Model-based performance prediction with the Palladio Component Model”, Proc. 6th International Workshop on Software and Performance, ACM Sigsoft, 2007.

[BKR09] S. Becker, H. Koziolek y R. Reussner, “The Palladio component model for model-driven performance prediction”, Journal of Systems and Software, v.82, 2009.

[BRO00] A.W. Brown, “Large-scale component-based development”, Upper Saddle River, N.J. : Prentice Hall PTR, 2000.

[BW95] A. Burns y A. Wellings, “HRT-HOOD: a Structured Design Method for Hard Real-Time Ada Systems”, Real-Time Safety Critical Systems, Elsevier, 1995.

[BW07] A. Burns y A. Welling, “Concurrent and Real-Time Programming in Ada”,Cambridge University Press, 2007.

[BWC04] E. Bondarev, P.de With y M. Chaudron, “Predicting Real-Time Properties of Component-based Applications”, In Proceedings of the 10th International RTCSA Conference, Gothenburg, Sweden, August 2004.

[CCM] Object Management Group: "CORBA Component Model Specification" OMG document number: formal/06-04-01, Abril 2006.

[CIAO] Component Integrated ACE ORB, web page: http://www.cs.wustl.edu/~schmidt/CIAO.html.

[CHE02] A.Cheng, "Real-Time Systems Scheduling, Analysis and Verification", John Willey & Sons, Inc., New Jersey, USA, 2002.

[COM] Microsoft, "Microsoft COM Technologies", http://www.microsoft.com/com.

[COMPS] COMPASS, "Component Based Automotive System Software", http://www.infosys.tuwien.ac.at/compass.

[CL02] I. Crnkovic and M. Larsson, "Building Reliable Component-Based Software Systems", Artech House Publishers, 2002. ISBN 1-58053-327-2.

[CLS06] S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele y E.Wandeler, "Interface-Based Rate Analysis of Embedded Systems", Proceedings of the 27th IEEE International Real-Time Systems Symposium, December 2006.

[CN02] P. Clements. & L. Northrop, "Software Product Lines: Practices and Patterns", Boston, MA: Addison-Wesley, 2002.

[CRN03] I. Crnkovic, "Component-based Software Engineering- New Challenges in Software Development", 25th Int. Conf on Information Technology Interfaces, Cavtat, Croacia, Junio 2006.

[CRN04] I. Crnkovic, "Component-based approach for embedded systems", Workshop of Component oriented programming, Oslo, Norway, June, 2004

[CLC06] I. Crnkovic, S. Larsson y M. Chaudron, "Component-based Development Process and Component Lifecycle", Intl. Conference on Software Engineering Advances, ICSEA'06, IEEE, Tahiti, French Polynesia, October, 2006

[D&C] Object Management Group: "Deployment and Configuration of Component-based Distributed Applications Specification" OMG document number: formal/06-04-02, Abril 2006.

[DBO05] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale,"DAnCE: A QoS-enabled Component Deployment and Conguration Engine", Proc.3rd Working Conference on Component Deployment, Grenoble, November 2005.

[DDS] Object Management Group: "Data Distribution Service for Real-Time Systems" OMG document number: formal/2007-01-01.

[DDSCCM] Object Management Group: "DDS for Ligthweight CCM Version Beta 1" OMG document number: ptc/2009-02-02.

[DGL08] M. Díaz, D. Garrido, L. Llopis, F. Rus, J.M. Troya, "UM-RTCOM: An analyzable componetn model for real-time distributed systems", Journal of Systems and Software, Volume 81 , Issue 5, Mayo 2008.

[DOU99] B. P. Douglas, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns", Reading Mass.: Addison-Wesley, USA, 1999.

[ECB06] J. Etienne, J. Cordry, and S. Bouzefrane, "Applying the CBSE Paradigm in the Real-Time Specification for Java". Proc. 4th Intl. Workshop on Java technologies for real-time and embedded systems, pages 218–226, 2006.

[ECK95] Eckerson, Wayne W, "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." Open Information Systems 10, 1 (January 1995).

[ECLIP] Eclipse project, web page: http://www.eclipse.org/.

[EJB] Sun Microsystems, “Enterprise JavaBeans Specification.” java.sun.com/products/ejb/docs.html, August 2001.

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

[GG99] J.J.Gutiérrez García and M. González Harbour, “Prioritizing Remote Procedure Calls in Ada Distributed Systems”, Proc. of the 9th Intl. Real-Time Ada Workshop, ACM Ada Letters, XIX, 2, pp. 67 72, Junio 1999.

[GG05] J. Gutiérrez García and M. González Harbour, "Optimized Priority Assignment for Tasks and Messages in Distributed Real-Time Systems", Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems, Santa Barbara, California, pp. 124-132, April 1995.

[GGP01] M. González Harbour, J.J. Gutiérrez, J.C. Palencia and J.M. Drake: “MAST: Modeling and Analysis Suite for Real-Time Applications”. Proceedings of 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[GOM84] H. Gomaa, “A Software Design Method for Real-Time Systems”, Communications of the ACM, Volumen 27, ACM, 1984.

[GOM89] H. Gomaa, “A Software Design Method for Distributed Real-Time Applications”, The Journal of Systems and Software, Elsevier Science Publishers, 1989.

[GOM93] H. Gomma, “Software Desing Methods for Concurrent and Real-Time Systems”, Reading Mass.: Addison-Wesley, 1993.

[GOM00] H. Gomma, “Designing concurrent, distributed, and real-time applications with UML”, ISBN 0-201-65793-7, Addison-Wesley, USA, 2000.

[GOP01] K. Gopalan, “Real-Time Support in General Purpose Operating Systems”, ECSL Technical Report TR92, Experimental Computer Systems Labs, Stony-Brook University, 2001.

[GS05] G. Gossler and J. Sifakis, “Composition for component-based modeling”. Sci. Comput. Program., 55(1-3):161–183, 2005.

[HAC01] D. Hammer and M. Chaudron, "Component-based software engineering for resource-constraint systems: what are the needs?”, Proc. Sixth International Workshop on Object-Oriented Real-Time Dependable System, 2001.

[HAC04] H. Hansson, M. Åkerholm, I. Crnkovic y M. Törngren, “SaveCCM- a component model for safety critical real-time systems”, Proc. 30th Euromicro Conference on Software Engineering and Applications, Septiembre 2004.

[HC01] G. Heineman y W. Councill, editors, “Component-based Software Engineering: Putting the pieces together”, Addison-Wesley, USA, 2001

[HCM08] J. Håkansson, J. Carlson, A. Monot, P. Pettersson, “Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT”, 6th International Symposium on Automated Technology for Verification and Analysis, p 252-257, Springer-Verlag, Seoul, October, 2008.

[HDJ08] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A.L. Sangiovanni-Vincentelli, M. Di Natale, “Software Components for Reliable Automotive Systems”, Design Automation and Test in Europe, DATE 2008, 549-554.

[HGC07] J. Hu, S. Gorappa, J. A. Colmenares, and R. Klefstad, “Compadres: A Lightweight Component Middleware Framework for Composing Distributed, Real-Time, Embedded Systems with Real-Time Java”. Proc. ACM/IFIP/USENIX 8th Intl Middleware Conference (Middleware 2007), 2007.

[HM06] T. A. Henzinger and S. Matic, “An interface algebra for real-time components”,Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2006.

[HMN08] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J.Lundback y K-L. Lundback, “The Rubus Component Model for Resource Constrained Real-Time Systems”, Internation Symposium on Industrial Embedded Systems (SIES 2008), June 2008.

[HMS03] S. Hissam, G. Moreno, J. Stafford y K. Wallnau, “Enabling Predictable Assembly”, Journal of Systems and Software: Special Issue on Component-based Software Engineering, 2003.

[IN02] D. Isovic and C. Norström, “Components in real-time systems”, Proceedings of the Eight International Conference on Real-Time Computing Systems and Applications (RTCSA'02), pages 135-139, Tokyo, Japan, March 2002.

[JBEAN] Sun Microsystems, “JavaBeans Technology”, http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp#learning.

[JMG07] X. Jin, H. Ma, Z. Gu, “Real-Time Component Composition Using Hierarchical Timed Automata”, Proc. 7th Intl.Conference on Quality Software, October 2007.

[KDK89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, Ralph Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," IEEE Micro, vol. 9, no. 1, pp. 25-40, Jan./Feb. 1989.

[KG08] A. Kavimandan y A. Gokhale, “Automated Middleware QoS Configuration Techniques for Distributed Real-time and Embedded Systems”, IEEE Real-Time and Embedded Technology and Applications Symposium, Abril 2008.

[KLM97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier y J. Irwin, "Aspect-Oriented Programming", Proc. European Conference on Object-Oriented Programming, vol.1241. pp. 220–242, 1997.

[KRP93] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour, A Practitioner's Handbook for Real-Time Systems Analysis, Kluwer Academic Pub., 1993.

[KSA07] X. Ke, K. Sierszecki, C. Angelov, “COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems”, Proc. 13th RTCSA Conference, August 2007.

[KZF91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner y W. Schuntz, “The design of real-time systems: from specificatio to implementation and verification”, Software Engineering Journal, vo. 6, no. 3, pp 72-82, Mayo 1991.

[LBD10] P. López Martínez, L. Barros y J. M. Drake, “Scheduling Configuration of Real-Time Component-Based Applications”, 15th Ada Europe Conference on Reliable Software Technologies, Lecture Notes on Computer Science, Vol: 5026, June 2010.

[LCD10] P. López Martínez, César Cuevas y J.M. Drake, "RT-D&C: Deployment Specification of Real-Time Component-Based Applications", In Proc. of 35th Euromicro Conference on Software Engineering and Advanced Applications, Lille, Septiembre 2010.

[LCD10b] P. López Martínez, César Cuevas y J.M. Drake, "Model-Driven Design of Real-Time Component-based Applications", In Proc. of 15th IEEE Internation Conference on Emerging Technologies and Factory Automation, Bilbao, Septiembre 2010.

[LDM04] P. López Martínez, J.M. Drake y Julio Medina, "Sim_MAST: Simulador de sistemas distribuidos de tiempo real", Actas de las XII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2004), Ávila, Junio 2004.

[LDM08] P. López Martínez, J. M. Drake, J.L. Medina y P. Pacheco, "An Ada 2005 Technology for Distributed and Real-Time Component-based Applications ", 13th International Conference on Reliable Software Technologies, Ada-Europe, Venice (Italy), in Lecture Notes on Computer Science, Springer, LNCS 5026, June, 2008.

[LDM09] P. López Martínez, J. M. Drake y J. L. Medina, "Enabling Model-Driven Schedulability Analysis in the Development of Distributed Component-Based Real-Time Applications", 35th Euromicro Conference on Software Engineering and Advanced Applications, Patras, Greece, August 2009.

[LDP08] P. López Martínez, J. M. Drake, P. Pachecho and J. L. Medina, "Ada-CCM: Component-based Technology for Distributed Real-Time Systems", 11th International Symposium on Component-based Software Engineering, CBSE 2008, Karlsruhe (Germany), in Lecture Notes on Computer Science, Springer, LNCS 5282, October, 2008.

[LG04] J. López Campos, J.J. Gutiérrez y M. González Harbour, "The Chance for Ada to Support Distribution and Real Time in Embedded Systems", Proc. of the Intl. Conf. on Reliable Software Technologies, Palma de Mallorca, Spain, in LNCS, Vol.3063, Springer, Junio 2004.

[LG06] J. López Campos, J.J. Gutiérrez y M. González Harbour, "Interchangeable Scheduling Policies in Real-Time Middleware for Distribution", Proc. of the 11th Intl. Conf. on Reliable Software Technologies, Porto (Portugal), in LNCS, Vol. 4006, Springer, Junio 2006.

[LL73] C. L. Liu y J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, Vol. 20, No. 1, pp 46-61, ,1973.

[LL82] J. Leung, and J.W. Layland. "On Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", Performance Evaluation 2, 237-50, 1982.

[LMD06] P. López, J.M. Drake, and J.L. Medina, "Real-Time Modelling of Distributed Component-Based Applications", In Proc. of 32h Euromicro Conference on Software Engineering and Advanced Applications, Croatia, August 2006.

[LR01] M.M. Lehman y J. F. Ramil, "EPICS: Evolution Phenomenology in Component-intensive Software" Workshop on Empirical Studies of Software Maintenance, Florencia, Noviembre 2001.

[LTG03] T. Lu, E. Turkay, A.Gokhale, D. C. Schmidt, "CoSMIC: An MDA Tool suite for Application Deployment and Configuration", Proc. ACM OOPSLA 2003

Workshop on Generative Techniques in the Context of Model Driven Architecture, Anaheim, CA, October 26, 2003.

[LIU00] Jane W.S. Liu: Real-Time Systems. ISBN 0-13-099651-3. Prentice Hall Inc., 2000.

[LW05] K.-K. Lau and Z. Wang, "A taxonomy of software component models," Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, Washington, DC, USA: IEEE Computer Society, 2005.

[MAF04] A. Moller, M. Akerholm, J. Fredikson y M. Nolin, "Evaluation of Component Technologies with Respect to Industrial Requirements", Proceedings of the 30th Euromicro Conference on Software Engineering and Applications, Rennes, France, Septiembre 2004.

[MARTE] Object Management Group, "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) Beta 2", OMG document number: ptc/2009-05-13, Mayo 2009.

[MAST] Modeling and Analysis Suite for Real-Time Applications, web page: http:// mast.unican.es.

[MASTd] J.M. Drake, Michael González Harbour, José Javier Gutiérrez, Patricia López Martínez, Julio Luis Medina y José Carlos Palencia, "Modeling and Analysis Suite for Real-Time Applications (MAST 1.3.7): Description of the MAST Model", Universidad de Cantabria, http://www.ctr.unican.es/mastdescription.pdf, 2008.

[MaROS] MaRTE OS, Minimal Real-Time Operating System, web page: http:// marte.unican.es/.

[MCCM] The MICO CORBA Component Project, web page: http://www.fpx.de/MicoCCM.

[MED05] Medina Pasaje, J., "Metodología y herramientas UML para el modelado y análisis de sistemas de tiempo real orientados a objetos", Tesis Doctoral, Septiembre 2005.

[MDA] Object Management Group: Architecture Board MDA Drafting Team, Model Driven Architecture - A Technical Perspective, OMG Document, ormsc/2001.

[MDE] D.C. Schmidt, "Model-Driven Engineering", IEEE Computer Society, Febrero 2006.

[MG05] J.M. Martínez y M. González Harbour , "RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet", Proc. of the 10th International Conference on Reliable Software Technologies, York (UK), in LNCS, Vol. 3555, Springer, Junio 2005.

[MGD01] J.L. Medina, M.González Harbour, J.M. Drake:" "MAST Real-time View: A Graphic UML Tool for Modeling Object_Oriented Real_Time Systems", Proceedings of Real-time systems symposium (RTSS), London, December, 2001.

[MH05] P. Merson y S.A. Hissam, "Predictability by construction", SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA05, San Diego, 2005.

[MDL06] J. Median, P. López y J.M. Drake, "Towards a UML Profile for Real-Time Modelling of Component-based Distributed Embedded Systems", Proceedings of IX Forum on Specification and Design Languages, Darmstadt, October 2006.

[MZS01] P. Müller, C. Zeidler, C. Stich and A. Stelter, "PECOS — Pervasive Component Systems", Workshop on "Open Source Technologie in der Automatisierungstechnik", GMA Kongress, Baden-Baden, Germany, 2001.

[NAD02] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A.Black, P. Müller, C. Zeidler, T. Genssler, R. van den Born, “A Component Model for Field Devices”, Proc. First International IFIP/ACM Conference on Component Deployment, June 2002.

[NAT08] M. Di Natale, “Design and Development of Component-Based Embedded Systems for Automotive Applications”, Proc. 13th Ada-Europe Intl. Conf. on Reliable Software Systems, Springer-Verlag, June 2008.

[NR68] P. Naur y B. Randell, editores, Proceddings of hte NATO Conference of Software Engineering, Garmisch, Germany, October 1968.

[NET] Microsoft, “.NET Home Page”, http://microsoft.com/net/.

[HIP05] S. Hissam, J. Ivers, D. Plakosh, and K. C. Wallnau. Pin component technology (V1.0) and its C interface. Technical Note CMU/SEI-2005-TN-001, Software Engineering Institute, Pittsburgh, PA, 2005.

[OCCM] OpenCCM, “The Open CORBA Component Model Platform”, http://openccm.ow2.org.

[OLK00] R. Ommering, F. Linden, J. Kramer: "The koala component model for con-sumer electronics software", IEEE Computer, IEEE (2000) 78-85.

[PACC] Predictable Assembly from Certifiable Code (PACC). Web page: http://www.sei.cmu.edu/pacc/.

[PG99] J.C. Palencia, and M. González Harbour, “Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems”. Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999.

[PJ09] H. Pérez and J. Javier Gutiérrez, “Experience in Integrating Interchangeable Scheduling Policies into a Distribution Middleware for Ada”, ACM SIGAda Annual International Conference on Ada and Related Technologies, November 2009, ACM.

[PMS08] A. Pläsek, P.Merle, L.Seinturier, “A Real-Time Java Component Model”, Proc. 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, 2008.

[PP99] A. Pasetti and W. Pree, “The component software challenge for real-time systems”, Proceedings First International Workshop oon Real-Time Mission-Critical Systems, Scottsdale, AZ, Nov 1999.

[PT00] L. Pautet y S. Tardieu, “GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems”, Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'00), Newport Beach, USA, Marzo 2000.

[QoSCCM]Object Management Group: Quality of Service for CORBA Components Specification, OMG Document number: ptc/06-04-15, Abril 2006.

[RBC03] Robocop: Robust Open Component Based Software Architecture, Public deliverables, http://www.hitech-projects.com/euprojects/robocop/deliverables.htm

[RJM98] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, “Resource kernels: A resource-centric approach to real-time and multimedia systems”, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.

[RTCRB] Object Management Group: “Real-Time CORBA Specification”, OMG document number: formal/2005-01-04, January 2005.

[RTP] RTP: Real Time Protocol. RFC 1889 (http://tools.ietf.org/html/rfc1889).

[RUBCM] http://www.arcticus-systems.com/html/tech-rubus-cm.html.

[RUBOS] http://www.arcticus-systems.com/html/prod-rubus-os.html.

[SDG07] V. Subramonian, G. Deng, C. Gill, J. Balasubramanian, L-J.Shen, W. Otte, D. C. Schmidt, A. Gokhale, and N. Wang, "The Design and Performance of Component Middleware for QoS-enabled Deployment and Conguration of DRE Systems", Elsevier Journal of Systems and Software, Special Issue Component-Based Software Engineering of Trustworthy Embedded Systems, 2007.

[SG96] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Engleewood Cliffs, N.J. Prentice Hall, 1996.

[SGW94] B. Selic, Garth Gullekson y Paul T. Ward, "Real-Time Object Oriented Modelling", John Willey and Sons, Inc., USA, 1994.

[SHP08] S. Sentilles, J. Håkansson, P. Pettersson, I. Crnkovic, "Save-IDE – An Integrated development environment for building predictable component-based embedded systems", Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), L'Aquila, Italy, September, 2008.

[SMAST] Simulator of MAST Models, web page: http://mast.unican.es/simmast.

[SPT] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification", Version 1.1, OMG document number: formal/05-01-02, 2005.

[SP97] C. Szyperski y C. Pfister, "First Workshop on Component-Oriented Programming, Summary", Special Issues in Object-Oriented Programming, Springer, 1997.

[SR88] J. Stankovic y K. Ramamritham, "Tutorial on Hard Real-Time Systems", Los Alamitos, CA: IEEE Computer Society Press, 1998.

[STA01] J. Stankovic, "VEST: a toolset for constructing and analyzing component based operating systems for embedded real-time systems", Proceedings of the Embedded Software First Internation Workshop (EMSOFT), 2001, LNCS 2211.

[STA03] J. Stankovic, et. al., "VEST: An Aspect-Based Composition Tool for Real_Time Systems", Real-Time Applications Symposium, May 2003.

[STA88] J. Stankovic, "Misconceptions about real-time computing", Computer, Octubre 1988, pg. 10-19.

[STE01] D. S. Stewart, "Designing software components for real-time applicationsÇ", Proc. Embedded System Conference, San Jose, CA, September 2000.

[SVB08] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, I. Crnkovic,"A Component Model for Control-Intensive Distributed Embedded Systems", Proc. 11th International Symposium on Component Based Software Engineering (CBSE2008), Karlsruhe, Germany, October, 2008.

[SysML] Object Management Group: "Systems Modelling Language", OMG document numberr: formal/2008-11-02, Diciembre 2008.

[SZY98] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley and ACM Press, ISBN 0-201-17888-5, 1998

[SZY02] C. Szyperski, D Gruntz y S. Murer, "Component Software: Beyond Object-Oriented Programming Second Edition", Addison-Wesley and ACM Press, ISBN 0-201-74572-0, 2002.

[TH04] A. Tesanovic y J. Hansson, “Structuring criteria for hte design of component-based real-time systems”, Proceedings of the IADIS International Conference on Applied Computing 2004, Marzo 2004.

[TIMMO] ITEA2 project TIMMO: “Timing Model”, www.timmo.org.

[TNH04] A. Tesanovic, D. Nyström, J. Hansson, and C. Norström, "Aspects and components in real-time system development: Towards reconfigurable and reusable software," Journal of Embedded Computing, February 2004.

[UML2S] Object Management Group: UML Superstructure Specification v2.2, OMG Document number: formal/2009-02-02, Febrero 2009.

[VHP04] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon, “PolyORB: a schizophrenic middleware to build versatile reliable distributed applications”, In Proceedings of the 9th International Conference on Reliable Software Technologies, volume 3063, Palma de Mallorca (Spain), June 2004. PolyORB web page, http:// polyorb.objectweb.org/.

[WAL03] K.C. Wallnau, “Volume III: A Technology for Predictable Assembly from Certifiable Components”, report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.

[WIL99] J. Williams, “Distributed Components and the Internet”, Application Development Trends, Diciembre 1999.

[WFP07] M. Woodside, G. Franks y D.C. Petriu, “The Future of Software Performance Engineering”, Future of Software Engineering, en International Conference on Software Engineering, Washington, DC, 2007.

[WGS04] N. Wang, C. Gill, D. C. Schmidt and V. Subramonian, "Configuring Real-time Aspects in Component Middleware", Proc. International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, October, 2004.

[WM85] P.T. Ward y S. J. Mellor, “Structured Development for real-time systems”, Yourdon Press, 1985.

[WRM05] S. Wang, S. Rho, Z. Mai, R. Bettati y W. Zhao, “Real-time Component-based Systems”, Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, Marzo 2005.

[WT05] E. Wandeler y L. Thiele, “Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling”, Proc. 5th ACM International Conference on Embedded software (EMSOFT), 2005.