Programa oficial de postgrado en
Ciencias, Tecnología y Computación
Máster en Computación
Facultad de Ciencias - Universidad de Cantabria

Trabajo de fin de Máster

Generación de código Ada para aplicaciones embebidas y de tiempo real desde modelos dinámicos UML

Alejandro Pérez Ruiz

perezruiza@unican.es

Tutor Julio Luis Medina Pasaje
Grupo Computadores y Tiempo Real
Dept. Electrónica y Computadores

Septiembre 2012
Curso 2011-12

Índice general

Índice de figuras

Capítulo 1

Introducción

Esta memoria de Trabajo de Fin de Máster muestra el desarrollo seguido para la construcción de un generador de código Ada para aplicaciones embebidas y de tiempo real desde modelos dinámicos UML. Como parte de este trabajo se han definido una serie de pautas y herramientas que permiten guiar al usuario o modelador en el paso de diseño a implementación en el marco de un proceso de desarrollo dirigido por modelos que utilice UML y el perfil MARTE para realizar aplicaciones de tiempo real. Este capítulo introduce el contexto del trabajo y presenta el desarrollo dirigido por modelos en el ámbito de los sistemas de tiempo real. Así mismo se describen las motivaciones, objetivos y el proceso seguido para llevar a cabo el desarrollo del proyecto.

1.1. Preámbulo

Actualmente, los sistemas informáticos van aumentando su complejidad exponencialmente. Se requieren tiempos de desarrollo de aplicaciones más cortos y procesos mucho más ágiles. Por lo que el desarrollo dirigido por modelos [Sch06] es considerado progresivamente por la industria actual como uno de las mayores objetivos para la Ingeniería del Software. Ayuda a crear y mantener los diferentes artefactos necesarios en todo proceso de desarrollo de software, además de facilitar la separación de los distintos módulos, incrementando la eficiencia del proceso, y finalmente mejorando la calidad del software.

Para las aplicaciones de tiempo real, una metodología basada en modelos nos puede ayudar a simplificar el proceso de construcción de los modelos de comportamiento de análisis temporal. Estos modelos constituyen la base para el diseño de tiempo real y el proceso de análisis de la planificabilidad. Con tal propósito, el diseñador debe generar, en sincronía con los modelos usados para generar el código de aplicación, un modelo adicional parametrizable adecuado para la validación temporal del sistema. El modelo de análisis abstrae el comportamiento temporal de todas las acciones que realiza, e incluye los parámetros de planificación, sincronización e información de los tiempos de ejecución de los recursos que es necesaria para predecir el comportamiento de tiempo real de las aplicaciones. En el enfoque que en este trabajo se presenta, estos modelos de análisis son automáticamente derivados desde un modelo de diseño de alto nivel, en el que tenemos anotadas un conjunto mínimo de

características de tiempo real tomadas de los requisitos de la aplicación en el entorno real en el que va a ser utilizada. De la misma forma en que se realiza la generación del código de la aplicación como una composición del código de sus partes constitutivas, el modelo completo de análisis de tiempo real de la aplicación también puede ser generado automáticamente a partir del conjunto de sub-modelos de tiempo real que lo componen.

El esfuerzo de investigación que se presenta en este trabajo considera la definición de los modelos de análisis de planificabilidad como parte del conjunto de herramientas y técnicas utilizadas en el proceso de ingeniería dirigido por modelos. A este nivel de abstracción, el paradigma concreto de modelado utilizado para concebir y elaborar los sistemas no se especifica, pero a efectos prácticos se asume que es posible expresarlo mediante UML [OMG11b]. Este es un lenguaje de modelado de propósito general, con él utilizaremos la extensión estándar para el modelado y análisis de sistemas de tiempo real, el denominado perfil MARTE [OMG11a].

El uso de técnicas de desarrollo dirigidas por modelo implica generalmente la generación de código a partir de modelos estructurales como los diagramas de clases. Con estos automatismos para la generación del código de las clases y paquetes que forman una aplicación, es por lo general fácil obtener su código estructural. Además, a fin de facilitar la ingeniería inversa se suelen utilizar comentarios codificados. Estos comentarios son colocados como marcas textuales que rodean las zonas del código generado que queden incompletas. Estas zonas son habitualmente escritas "a mano" por los programadores y corresponden al cuerpo de los paquetes o la descripción funcional de funciones y procedimientos, que especifican el comportamiento de las clases.

Un paso más en la generación de código es la utilización de las máquinas de estados para obtener el comportamiento de las clases. Ésto utiliza las operaciones de las clases como mensajes de control que desencadenan los eventos entre los diferentes estados. De este modo los mensajes procedentes de otros objetos pueden interactuar con el autómata de la clase, desafortunadamente este tipo de generadores de código no es consistente con la descripción del escenario requerido para realizar el análisis de planificabilidad, por lo que nos impediría tener una coherencia entre los modelos para la generación de código y los destinados al comportamiento temporal de la aplicación. Para este cometido los elementos de modelado adecuados son los diagramas de actividad, que son capaces de representar los flujos de ejecución paso a paso para las operaciones que componen las clases.

De este modo el presente trabajo constituye un avance en la metodología de desarrollo dirigido por modelos para sistemas de tiempo real al definir e implementar un nuevo tipo de generador de código. Con él no solo se genera el clásico esqueleto de las clases y sus operaciones UML, sino que se completan los cuerpos de las operaciones que tengan asociado un diagrama de actividad UML.

En la siguiente sección se describe el contexto en el que se desarrolla el presente trabajo.

1.2. Contexto

El generador de código desarrollado en este trabajo es una parte constitutiva de una metodología de desarrollo dirigida por modelos que permite realizar una análisis temporal

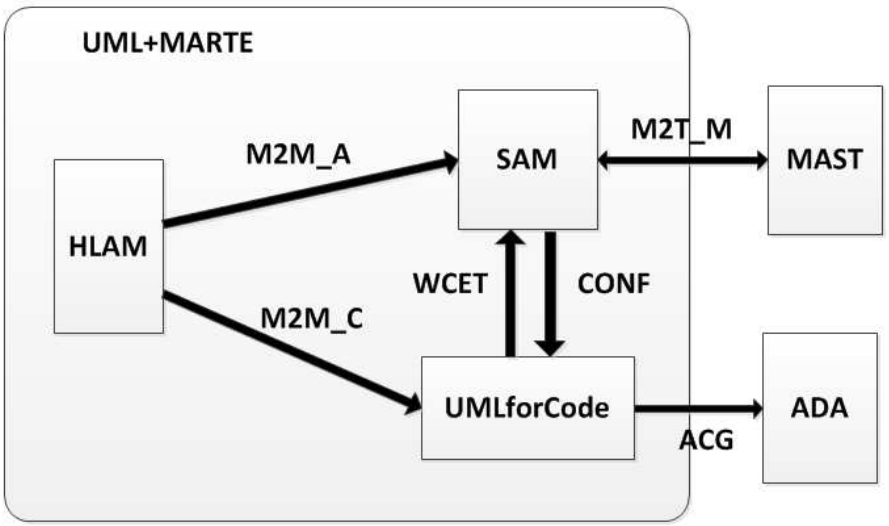

Figura 1.1: Modelos y transformaciones usados en el contexto del trabajo.

de las aplicaciones generadas. Esta metodología utiliza como lenguaje de modelado UML complementado con la extensión del perfil MARTE para el modelado y análisis de los sistemas embebidos y de tiempo real. El uso de dicho perfil es necesario para permitirnos especificar las características distintivas de los sistemas de tiempo real. De este modo en la figura 1.1 se ilustra esquemáticamente el contexto donde se sitúa el trabajo.

El modelo inicial usado para describir la aplicación y sus características de tiempo real se construye usando la extensión del perfil MARTE para el modelado de aplicaciones de alto nivel (High Level Application Modeling, *HLAM*). Desde este punto de partida se producen dos transformaciones modelo a modelo (model-to-model, *M2M*). La primera de ellas, *M2M_A*, es usada para crear una representación en UML del modelo de análisis temporal. Para este modelo se utilizan las capacidades para el modelado del análisis de la planificabilidad presentes en MARTE (*SAM*). La otra transformación, *M2M_C*, crea un modelo intermedio útil para la generación de código. Para esta metodología el lenguaje objetivo para la implementación es Ada, y el modelo intermedio, denominado *UMLforCode* en la figura 1.1, es un modelo típico orientado a objetos de UML que contiene la información estructural y de comportamiento de la aplicación. El comportamiento de las operaciones será expresado mediante los diagramas de actividad presentes en UML. La transformación de modelo a texto, llamada *M2T_M* en la figura 1.1, es necesaria para generar el modelo final para el análisis de la planificabilidad, y es parte de un trabajo realizado previamente [MC11], en el que se creó una herramienta [MC] para el entorno de desarrollo Eclipse para la generación de los modelos de análisis, la invocación de las herramientas de análisis y la recogida de resultados en el contexto de análisis de modelado. Dicha herramienta convierte modelos *SAM* en formalismos utilizados por MAST [HGGM01] para recoger los resultados en el modelo UML+MARTE.

Este trabajo presenta las técnicas y la herramienta para generar la implementación

del código Ada desde el modelo genérico orientado a objetos denominado *UMLforCode*. Esta generación se produce a través de una transformación modelo a texto (model-to-text) denominada *ACG* (Ada Code Generation) e ilustrada en la figura 1.1. El código obtenido tras aplicar las transformaciones *M2M_C* y *ACG* es consistente semánticamente con los modelos generados a través de las otras dos transformaciones *M2M_A* y *M2T_M*. Posteriormente, con el código Ada generado se obtendrán los tiempos de ejecución de peor caso (*WCET*) a través de la instrumentación del código, para ser recogidos luego en el modelo *SAM* y realizar el análisis de la planificabilidad. Estos resultados son finalmente anotados en el modelo *SAM* e incluyen datos de configuración de tiempo real tales como prioridades (o plazos relativos) para los elementos concurrentes, y techos de prioridad (o niveles de expulsión) para los recursos compartidos. Estos datos de planificación, llamados *Conf.* en la figura 1.1, se incluyen como parte de la información de configuración en el modelo *UMLforCode* a fin de ser utilizados para la generación del código final de la aplicación.

En la siguiente sección se pasará a detallar cuáles han sido las motivaciones para realizar un generador de código basado en nuevas técnicas para la obtención de código.

1.3. Motivación: La necesidad de una nueva técnica para generar código

Siguiendo trabajos anteriores que han estudiado el diseño de sistemas de tiempo real usando formalismo de orientación a objetos, podemos observar que la mayoría de ellos incluyen la especificación de la concurrencia usando modelos estructurales, habitualmente a nivel de diseño para implementación. Estos formalismos son utilizados con el objetivo de ayudarnos en la realización del análisis de planificabilidad a través de la especificación de un modelo simple de tareas y el uso posterior de las técnicas básicas de RMA [KRP$^+$93]. Desafortunadamente, la complejidad de los mecanismos usados para la generación de código hace que esta asunción no sea realista. Algunos ejemplos de metodologías que utilizan esta estrategia son ROOM [SGW94] [Sel98], Octopus/UML [EDZ99], ACCORD/UML [LGT98], Comet [Gom00] o los modelos de diseño limitados y monolíticos como HRT-HOOD [BW94] y OO-HARTS [MDN$^+$03]. En la mayoría de los casos el código generado corresponde simplemente a la descomposición modular en clases y métodos, dejando demarcado mediante comentarios específicos el espacio para introducir el código funcional de cada método, o generando este último mediante máquinas de estado.

Ensayando una suerte de sincretismo de estas metodologías mencionadas, y con el fin de facilitar la aplicación de técnicas sencillas de análisis de planificabilidad, las construcciones de alto nivel para el modelado de aplicaciones en MARTE (HLAM) también facilitan el uso de modelos estructurales para la especificación de la concurrencia. Sin embargo, las interacciones entre ellos (incluyendo la distribución) pueden tomar patrones complejos que requieren un modelo más rico para el análisis. Las técnicas de análisis basadas en offsets se adaptan mejor a estos escenarios que el modelo simple de tareas. HLAM propone dos bloques de construcción básicos, la unidad de tiempo real: *RtUnit* y la unidad pasiva protegida: *PpUnit*. En cuanto al comportamiento que define el código (el código que se encuentra dentro de las marcas o comentarios), debido a su complejidad natural, no es por lo general

solo código lineal pasivo que puede ser modelado como un cálculo de tiempo computacional, sino que incluye retardos e interacciones explícitas entre objetos y nodos, sobre todo cuando se convierten en parte de operaciones distribuidas (modelos de comportamiento). En estos casos, una máquina de estados no es directamente transformable en un modelo de análisis.

Desde la perspectiva de análisis, los modelos que son requeridos para aplicar las técnicas basadas en offsets, son fundamentalmente escenarios. Un escenario es una expresión del peor caso esperado para el diseño dado. Este es el principio fundamental empleado para tratar con la complejidad que distingue a las técnicas de análisis de planificabilidad basadas en RMA de otras estrategias basadas en autómatas de tiempo o lenguajes síncronos.

Como lenguaje de modelado para este dominio, la sección de modelado de análisis de planificabilidad de MARTE (SAM) sirve para expresar este tipo de escenarios, y por lo tanto, es un formalismo adecuado para utilizar con las correspondientes herramientas de análisis. Desafortunadamente estos escenarios no son necesariamente parte inicial de la especificación del comportamiento del sistema. Son un medio para expresar: los estímulos previstos, la carga de trabajo de alto nivel y los requerimientos temporales, pero no suelen ser los datos básicos utilizados para el propósito del diseño o la generación de código elaborada por los diseñadores.

La creación de estos modelos de análisis orientados a escenarios (habitualmente el peor caso) en consonancia con el código final es en realidad el objetivo principal y una alta responsabilidad de los diseñadores de tiempo real. Con el fin de ayudar en esta labor las herramientas necesitan que del modelo usado para la generación de código se puedan obtener los comportamientos de las operaciones expresadas en los escenarios. Por esta razón los modelos de entrada adecuados para la generación de código dentro de las operaciones en el modelo *UMLforCode* son los diagramas de actividad de UML. De este modo se tienen estructuras similares en ambos modelos y los modelos de análisis pueden ser caracterizados con sus tiempos de ejecución de peor caso instrumentando y estudiando el código final generado.

La siguiente sección describe los objetivos que se han propuesto para el desarrollo de este trabajo de fin de máster.

1.4. Objetivo

Como se ha comentado en la sección 1.2 estamos inmersos en un contexto de trabajo compuesto por diferentes metodologías y transformaciones por definir e implementar. De tal modo que se plantea como objetivo para el presente trabajo realizar una contribución significativa en el contexto descrito en dicha sección. El objetivo concreto de este proyecto es crear un generador de código Ada con las siguientes características:

1. Capaz de generar el código estructural de una aplicación a través de diagramas de clases UML.

2. Poder generar la implementación de las operaciones de las clases que tengan asociado un modelo de actividad UML.

3. Con la ayuda del perfil MARTE y el sub-perfil HLAM ser capaz de identificar en el modelo de entrada los elementos básicos de la programación concurrente tales como tareas y objetos protegidos.

4. Generar código que nos permita obtener su modelo de análisis temporal.

Cuando se plantean los objetivos anteriores para crear un generador de código en el ámbito de los sistemas de tiempo real, nace la necesidad de especificar modelos descritos como escenarios para realizar el análisis de la planificabilidad del sistema que se diseña. La creación de estos escenarios (habitualmente el peor caso) orientados al análisis, nos lleva a realizar la definición de éstos a través de los diagramas de actividad, los cuales describen el flujo de trabajo de las operaciones. El tener la descripción funcional de cada operación mediante diagramas de actividad facilitará más adelante instrumentar el código de forma automática para medir sus tiempos de ejecución.

Para satisfacer dichos objetivos, se realizaron las tareas que se describen a continuación:

1. **Estado del arte**: El objetivo de esta fase es adquirir los conceptos necesarios para la realización del proyecto, más concretamente, familiarizarse con el perfil MARTE y el sub-perfil HLAM. Dentro de esta fase también se incluye un estudio de las principales metodologías, técnicas y herramientas que permiten generar código para sistemas de tiempo real.

2. **Desarrollo**: En esta etapa del trabajo se han desarrollado una serie de transformaciones para trasladar los elementos básicos de los diagramas de clases descritos en UML y en el sub-perfil HLAM a código Ada. Para esta labor se ha utilizado el plugin para Eclipse denominado Acceleo [Obe] que implementa el estándar de la OMG, *MOF Model to Text Language (MTL)* [OMG08],y que permite crear generadores de código para cualquier metamodelo dado.

3. **Validación a través de casos de estudio**: En esta fase se analizará el generador de código a través de dos casos de uso, el primero de ellos nos permitirá verificar los elementos destinados a gestionar sistemas concurrentes, y el segundo utilizará las características básicas de la orientación a objetos.

Estas tareas se reportan en la memoria en los capítulos 2, 3 y 4 respectivamente. En el siguiente capítulo se realiza un estudio del estado del arte, se presentan conceptos clave mencionados aquí y se comparan las distintas herramientas que existen actualmente para la generación de código de sistemas de tiempo real.

Capítulo 2

Estado del arte y de la técnica

Para entender el presente hay que conocer el pasado.

En este capítulo se estudiará el estado del arte actual, por lo cual se presentará brevemente el perfil MARTE de UML para el modelado de sistemas embebidos y de tiempo real, y por otro lado se analizarán las metodologías y herramientas existentes para el modelado y generación de código que puedan ser utilizadas para el desarrollo de sistemas de tiempo real.

2.1. Perfil MARTE

Los perfiles son mecanismos que han sido definidos para extender UML a dominios específicos. Esta extensión debe garantizar la consistencia con el lenguaje y ser sintácticamente correcta. Los perfiles se apoyan en estereotipos caracterizados mediante propiedades, los cuales se definen sobre cualquier elemento del metamodelo.

El perfil MARTE *Modeling and Analysis of Real-Time Embedded Systems* [OMG11a] de UML se planteó para el análisis y modelado de sistemas embebidos y de tiempo real, por lo que realiza una extensión de UML a dos dominios específicos: sistemas embebidos y tiempo real, tanto para las labores de diseño como las de análisis. Para tal cometido MARTE nos otorga las siguientes posibilidades:

1. Modelar de forma independiente el software y el hardware del sistema.
2. Caracterizar los sistemas de forma cualitativa (comunicación, paralelismo...) o cuantitativa (prioridades, periocidad, plazos...).
3. Especificar los sistemas a través de constructores de alto nivel y bajo nivel respectivamente.
4. Soportar las tendencias actuales en tecnologías empleadas en sistemas de tiempo real, paradigmas de diseño y técnicas de análisis de modelos. Pero al mismo tiempo, está abierto a nuevos avances en la tecnología.

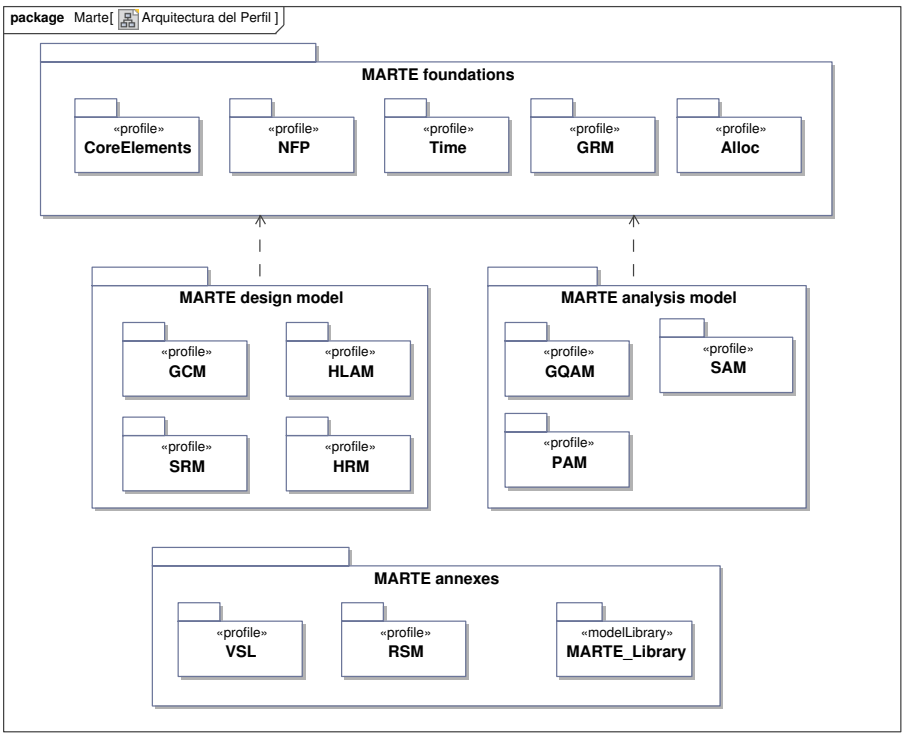

Figura 2.1: Arquitectura del perfil MARTE.

5. Análisis de las características software y hardware del sistema en forma cuantitativa y segmentada.

De tal modo que se permite modelar las características de estos sistemas y acotar los modelos de aplicación, de manera que se soporte el análisis de las propiedades del sistema ilustradas en la figura 2.1. Podemos observar que el modelado está soportado por el paquete *MARTE design model* y el análisis mediante el paquete *MARTE analysis model*. Estos dos paquetes se basan en las formas de especificación del tiempo y la utilización de recursos concurrentes, los cuales se encuentran en el paquete *MARTE foundations*. Este último paquete define los fundamentos del perfil:

1. *CoreElements*: Los elementos definidos en el core son la base para el resto de la especificación. Es útil para definir conceptos más elaborados, y todo está basado en dos paquetes centrales:

   a) *Foundations*: Contiene los elementos básicos para el modelado estructural del sistema.

   b) *Causality*: Describe los elementos básicos necesarios para el modelado del comportamiento y su semántica de tiempo de ejecución.

2. *NFP*: Contiene un marco de trabajo para el modelado de propiedades no funcionales (política de planificación, utilización de memoria...).

3. *Time*: Define los elementos necesarios para representar el tiempo y los conceptos relacionados con este.

4. *GRM*: Modela los recursos genéricos representados como una entidad física o lógica que ofrece un servicio.

5. *Alloc*: Incluye definiciones relevantes para la asignación de los elementos funcionales sobre los recursos disponibles(*Allocation*), esto es, sobre la plataforma de ejecución.

El paquete *MARTE analysis model* que se muestra en la figura 2.1 está diseñado para analizar y caracterizar el comportamiento del sistema y la planificación, pero el modelo de análisis es tan flexible que permite definir otros como consumo de energía, seguridad, uso de memoria, entre muchos otros. Para tal cometido el paquete contiene tres sub-perfiles:

1. *GQAM*: Modelo de análisis cuantitativo genérico que define los conceptos básicos de análisis y las propiedades no funcionales.

2. *SAM*: Marco de trabajo diseñado específicamente para el análisis de la planificabilidad, para ello describe un conjunto de propiedades comunes para el modelo basado en el análisis de la planificabilidad.

3. *PAM*: Contiene los elementos para el modelado del análisis del rendimiento, estando especialmente orientado a obtener las medidas de rendimiento de forma estadística.

Por otro lado el paquete *MARTE annexes* contiene descripciones adiciones de lenguajes de especificación, como por ejemplo *VSL - Value Specification Language* definido por el propio perfil MARTE y que es utilizado para especificar valores para las restricciones, las propiedades y los estereotipos relacionados con las propiedades no funcionales.

Por último tenemos el paquete *MARTE design model* que está orientado a modelar los sistema de tiempo real y embebidos a través de artefactos que permitan afrontar el modelado con un enfoque basado en componentes, esto es, el modelo de comportamiento genérico (*GCM*), el modelado detallado de recursos software (*SRM*) y hardware (*HRM*) y finalmente los conceptos de alto nivel para las aplicaciones (*HLAM*).

Debido a que el presente proyecto desarrolla un generador de código a través de modelos de alto nivel UML de sistemas de tiempo real, es muy relevante el sub-perfil *HLAM* para poder definir elementos de concurrencia en nuestro modelo de entrada, por ello la siguiente subsección describe brevemente algunos aspectos de dicho sub-perfil de MARTE.

2.1.1. Modelado de Aplicaciones de Alto Nivel (*HLAM*)

El objetivo del paquete *HLAM* es proveer conceptos de modelado de alto nivel para las características de los sistemas de tiempo real y embebidos. En comparación con el dominio de las aplicaciones habituales, los sistemas de tiempo real requieren características cuantitativas de modelado, como el plazo o el periodo, y por otro lado características cualitativas que

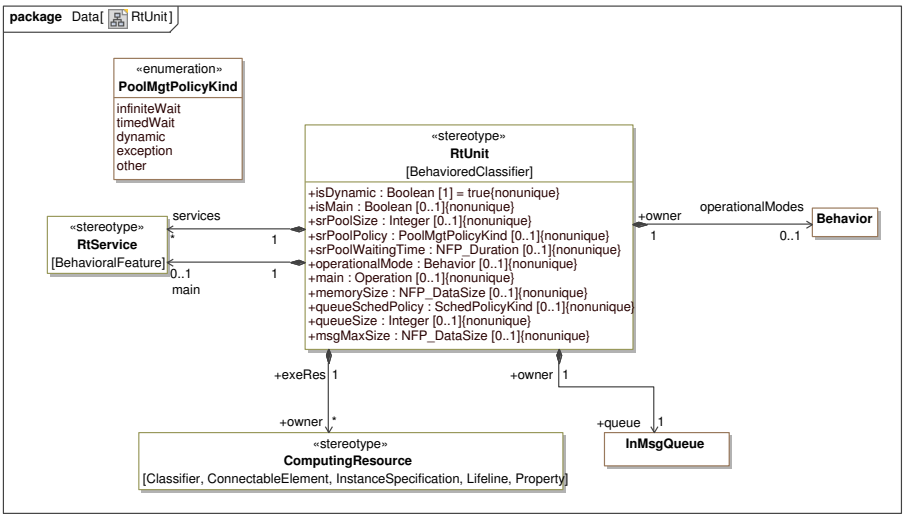

Figura 2.2: RtUnit del paquete HLAM.

están relacionados con el comportamiento, comunicación y concurrencia. Como uno de los temas más importantes para tratar cuando modelamos aplicaciones de tiempo real es la concurrencia, *HLAM* usa el concepto de *RtUnit* (ver figura 2.2) con el fin de manejar esta característica. Con *RtUnit* es posible crear elementos de alto nivel que permitan modelar sistemas de tiempo real y embebidos, ya que son similares a los objetos activos de UML, pero con un nivel de detalle semántico mayor.

Cada *RtUnit* puede poseer un conjunto de recursos de planificación o crearlos dinámicamente cada vez que se requiera, por lo tanto, una *RtUnit* puede verse como un recurso de ejecución autónomo, que puede manejar diferentes mensajes al mismo tiempo, por ejemplo, mensajes de concurrencia y de restricciones de tiempo real. Una *RtUnit* es una unidad de concurrencia que encapsula en una sola entidad los paradigmas de objeto y proceso. Cualquier unidad de tiempo real puede invocar servicios de otras unidades de tiempo real, enviando señales o datos, sin que esto perturbe las características de concurrencia del sistema.

Las aplicaciones deben tener al menos una *RtUnit* con el atributo *isMain* a verdadero. Después de la creación, cada *RtUnit* que tenga dicho atributo a verdadero empezará invocando a los principales servicios de tiempo real, los cuales ejecutarán hasta que la *RtUnit* finalice. Como cualquier otra unidad de tiempo real, el servicio principal de una unidad principal puede realizar acciones explícitas recibidas durante su ejecución, con el fin de aceptar los eventos recibidos. Una acción recibida por una unidad de tiempo real conduce a una activación directa de la especificación apropiada de un servicio. Durante la ejecución de un servicio, activado por la recepción de un mensaje, el principal servicio puede ser bloqueado o ejecutarse concurrentemente.

Otro aspecto clave al modelar la concurrencia, es el de la información compartida entre las unidades de tiempo real, aparece por tanto el concepto de *PpUnit* mostrado en la figura

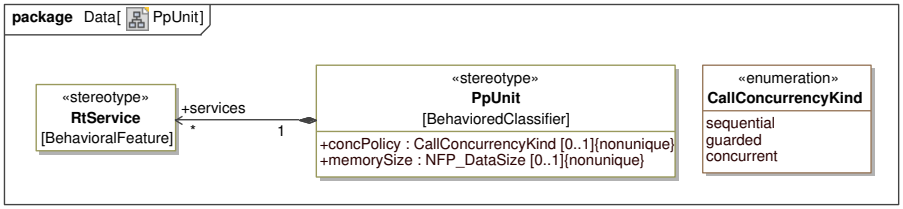

Figura 2.3: PpUnit del paquete HLAM.

2.3. La unidad pasiva protegida especifica su política de concurrencia de manera global para todos los servicios que ofrece (mediante el atributo *concPolicy*) o individualizada mediante el atributo *concPoliciy* correspondientede al estereotipo *RtService* aplicado sobre las operaciones que contiene. El tipo de ejecución puede ser *immediateRemote* o *deferred*, el cual en ambos casos es ejecución remota, ya que sus servicios se ejecutan en el contexto del *Thread* llamante.

Tener mecanismos y notaciones para describir un sistema de tiempo real como los mencionados es importante para poder llevar a cabo un desarrollo correcto. Estas propuestas no son adoptadas fácilmente por la industria y los desarrolladores en general si no tienen un respaldo en herramientas software disponibles. En la siguiente sección se describen algunas de las herramientas de modelado y generación de código que existen actualmente.

2.2. Herramientas de modelado para la generación de código

2.2.1. Rational Software Architect

Esta herramienta, antes conocida como *Rational Rose*, forma parte de la familia de software de IBM denominado *Rational Software* para el despliegue, diseño, construcción, pruebas y administración de proyectos en el proceso de desarrollo de software.

Rational Software Architect se puede definir como un entorno de modelado y desarrollo que utiliza UML para el diseño arquitectural de servicios web y aplicaciones. Está construido a través del framework Eclipse e incluye capacidades para el análisis del código arquitectural y el desarrollo dirigido por modelos a través de UML. Todas las herramientas *Rational* son distribuidas como plug-in para Eclipse bajo licencia privativa.

Sus principales características son el soporte de las siguientes transformaciones de modelos UML a código Java, C++, C#, VB.NET, XSD, CORBA Interface Description Language (IDL) o SQL (Structured Query Language). También cuenta con soporte para las transformaciones de código a modelo de Java, XSD, VB.NET, C++ o C# a UML.

2.2.2. Rational Software Rhapsody

Al igual que la herramienta anterior pertenece a la familia de herramientas *Rational Software* de IBM. En un principio dicha herramienta fue desarrollada por la compañía israelita I-Logix pero fue adquirida por IBM en el año 2008. Este software está creado para

facilitar la colaboración y el continuo desarrollo en sistemas embebidos y de tiempo real. Para tal cometido utiliza modelos gráficos (UML, SysML, AUTOSAR, DoDAF, MODAF, UPDM) para generar aplicaciones en diferentes lenguajes de programación(C, C++, Ada, Java y C#). Está orientada a la búsqueda de la alta fiabilidad y seguridad crítica de sistemas de tiempo real y embebidos, ya que permite utilizar estándares como MISRA-C:2004, Ada Ravenscar o el perfil MARTE de UML. Además en sus últimas versiones se integra con Mathwork Simulink [KA11], el cual es un software de simulación de modelos o sistemas, con cierto grado de abstracción de los fenómenos físicos involucrados en los mismos, en el que se hace hincapié en el análisis de sucesos a través de la concepción de sistemas (cajas negras que realizan alguna determinada operación).

En cuanto a la generación de código es una herramienta bastante completa, ya que por un lado puede generar el código estructural de las aplicaciones y por otro lado el código de comportamiento a través de los diagramas de estados de UML o SysML [FMS11]. Para algunos lenguajes como Ada permite utilizar los puertos que aparecen a partir de la versión 2 de UML. Los puertos en UML proveen un punto de interacción entre la clase y el mundo exterior, son una característica clave para el diseño basado en componentes, ya que los podemos describir como interfaces que requieren o proveen servicios. Otra característica a destacar es la posibilidad de realizar simulaciones de nuestro sistema a través de los diagramas de secuencia o de actividad. Además dispone de la capacidad de generar modelos a través de ingeniería inversa, es decir desde el propio código podemos obtener diagramas de clases, de actividad, de secuencia, entre otros.

2.2.3. BoUML

Es una herramienta para el diseño de diagramas UML y generación de código desarrollada por Bruno Pages. Hasta la versión 4.3 se distribuyó como software libre con licencia GPL, pero a partir de dicha versión pasó a convertirse en un programa de pago.

Entre sus características destacan la generación de código estructural a través de diagramas de clases UML a lenguajes como C++, Java, PHP, Python o IDL. También es posible a partir de un código escrito en C++, Java o PHP obtener su diagrama de clases. Además para un modelo UML del que hemos obtenido su código para los lenguajes C++ o Java es capaz, si modificamos alguna parte del código estructural, de detectar dichos cambios y aplicarlos al modelo UML. Por último posee la capacidad de generar código hacia un lenguaje objetivo que se defina mediante una serie de reglas que nosotros mismos establezcamos.

Su principal desventaja reside en que no posee una gran comunidad de desarrollo a sus espaldas, básicamente está siendo desarrollada en su totalidad por el autor.

2.2.4. Umbrello

Es una herramienta de software libre para crear y editar diagramas UML que ayuda en el proceso del desarrollo de software. Fue creada por Paul Hensgen y está diseñado principalmente para KDE, aunque funciona en otros entornos de escritorio.

Umbrello permite utilizar los diagramas estándar UML pudiendo crearlos, además de manualmente, a partir de código en Java, C++, Python, IDL, Pascal/Delphi, Ada, o tam-

bién Perl (haciendo uso de una aplicación externa). Así mismo, nos permite crear diagramas de clases y generar el código automáticamente en los lenguajes anteriormente mencionados, entre otros. La generación de código solamente está orientada al código estructural, es decir que únicamente es posible generar código a partir de diagramas de clases, dicha generación posee algunas carencias, ya que las dependencias entre clases no son trasladas al código generado.

2.2.5. UML2Ada2005

Desarrollado por AdaCore como un Add-in de Papyrus con licencia EPL. Para utilizar este plug-in en el desarrollo de sistemas de tiempo real han construido un perfil de UML2 denominado *HiUML profile*. Este perfil está basado en el modelo computacional Ravenscar, con el que se pueden definir operaciones en exclusión mutua, clases con comportamiento concurrente, entre otros. Parece que se ha dejado de dar soporte y actualizaciones de la herramienta por lo que no se ha podido probar con profundidad, pero a través de algún tutorial disponible se puede determinar que dicha herramienta únicamente es capaz de generar el código estructural de la aplicación.

2.2.6. WinA&D

Es una herramienta con licencia privativa de modelado que soporta la generación de código, la cual es desarrollada por la compañía estadounidense Excel Software. Es capaz de obtener el código estructural para los lenguajes C++, C#, Java, Delphi, REALbasic y PHP a través de los diagramas de clase de UML. Para Ada 95 con los modelos de clase UML permite al diseñador representar fácilmente los paquetes Ada y varios tipos de relaciones como la agregación (padre/hijo y padre/paquetes anidados), generalización de instancias y dependencias entre paquetes. De tal modo, que es capaz de generar el código estructural interpretando las clases de los diagramas como paquetes de Ada 95.

Por otro lado han desarrollado otra herramienta denominada WinTranslator capaz de generar diagramas por medio de código fuente. Genera modelos de datos SQL, modelos de clases UML con C++, C#, Java, Delphi y Ada 95, y para los lenguajes estructurados como C, Pascal, Basic y Fortran obtiene diagramas estructurales.

2.2.7. Sumario de herramientas analizadas

Se han analizado las principales herramientas existentes relacionadas con el modelado y generación de código automático. A partir de ello se ha realizado una tabla comparativa (ver figura 2.4) con algunos de los aspectos más importantes analizados.

De todas las herramientas analizadas, solamente una es capaz de generar código de comportamiento para métodos u operaciones, y lo hace a través de diagramas de estados. Además, ninguna de ellas es capaz de determinar directamente el análisis de planificabilidad de la aplicación, únicamente dos de ellas permiten modelar elementos concurrentes y aspectos temporales, a través de los cuales se podría estudiar la planificabilidad empleando alguna otra herramienta adicional externa que lo soporte.

Programa	**Modelo inicial**	**Generación de código estructural**	**Generación de código de comportamiento**	**Lenguajes objetivo**	**Ingeniería inversa (de código a modelo)**	**Modelo temporal y concurrencia**	**Licencia**
Rational Software Architect	UML	Sí	No	Java, C++, C#, VB.NET, XSD, IDL y SQL	Sí, para código Java, XSD, VB.NET, C++ y C#	No	Copyright
Rational Software Rhapsody	UML, SysML, AUTOSAR, DoDAF, MODAF, UPDM	Sí	Sí, a través de diagramas de estados	C, C++, Ada 2005, Java y C#	Sí	Sí, a través del perfil MARTE de UML	Copyright
BoUML	UML	Sí	No	C++, Java, PHP, Python o IDL	Sí, para código Java y C++	No	GPL
Umbrello	UML	Sí	No	Java, C++, Python, IDL, Pascal/Delphi, Ada y Perl	No	No	Open-source
UML2Ada2005	UML	Sí	No	Ada 2005	No	Sí, a través del perfil HiUML	EPL
WinA&D	UML	Sí	No	C++, C#, Java, Delphi, REALbasic, PHP y Ada 95	Sí	No	Copyright

Figura 2.4: Resumen de herramientas analizadas.

El análisis de planificabilidad requiere conocer los tiempos de ejecución de la aplicación, y desde luego ello exige tener el código final de la misma. Así, instrumentando adecuadamente el código se puede extraer estos valores mediante la ejecución de todos los segmentos aislados de código de la aplicación y la acumulación estadística de los valores medidos para sus tiempos de ejecución sobre la plataforma objetivo. Esta labor, así como la asignación de los valores a los segmentos independientes del modelo de análisis se facilita enormemente si ambos modelos (el de análisis de peor caso y el de ejecución) están descritos mediante un formalismo común. Es para ello que se propone el uso de modelos de actividad, que se ajustan tanto a la descripción de escenarios concretos de ejecución como a la representación de algoritmos de programación de los mismos.

A la vista de las características de las herramientas disponibles (ver figura 2.4) y de los requisitos propios del contexto de procesamiento de modelos en el que este trabajo se enmarca (ver Sección 1.2), se concluye que es oportuno y necesario el desarrollar un generador de código que además de extraer la estructura de clases y métodos en una aproximación orientada a objetos, sea capaz también de extraer el código para el cuerpo de las operaciones a partir de modelos de actividad.

Pasemos en la siguiente sección a estudiar las transformaciones de modelo a texto, una tecnología básica e indispensable para crear un generador de código.

2.3. Transformaciones de modelo a texto

Las transformaciones de modelo a texto constituyen una tecnología fundamental en los diversos enfoques del desarrollo software dirigido por modelos. Esta tecnología permite caracterizar artefactos capaces de traducir automáticamente modelos en formatos textuales, dando lugar a este tipo de transformaciones. En la actualidad existen numerosas tecnologías para realizar transformaciones de modelo a texto, sin embargo la manera de realizar el proceso es más o menos el mismo en todos los casos:

1. Existe algún tipo implícito o explícito de representación de un metamodelo(por ejemplo, UML).

2. Hay alguna clase de lenguaje imperativo, tales como lenguajes de programación o secuencias de comandos, que nos permiten escribir generadores de código.

Usualmente el lenguaje usado para la generación comparte propiedades con los lenguajes de programación habituales. La utilización de estos lenguajes mejora sustancialmente gracias al soporte que poseen algunas herramientas tales como el completado automático de código, una característica presente en la mayoría de los entornos de desarrollo integrados para los lenguajes de programación normales. Además según la OMG este tipo de lenguajes destinados a la generación de código deben presentar las siguientes características [ONG+05]:

1. *Estructurado*: El lenguaje debe soportar la estructuración y control de la generación del texto. Esto significa que debe ser posible especificar estructuras que permitan crear un conjunto de generaciones de grano más fino.

2. *Mecanismos de control*: Se deben proporcionar los mecanismos básicos de control de flujo. Esto implica que debe ser posible proporcionar el equivalente semántico de bucles e instrucciones condicionales.

3. *Mezclar código y el texto de salida*: Debe proporcionar un forma sencilla de combinar la transformación del código (la lógica), los datos del modelo y el texto de salida. También deberá permitir convertir los datos del modelo a texto para ser usados en el texto final de salida.

4. *Servicios del sistema*: Se debe ofrecer soporte para la manipulación de cadenas de caracteres. También es deseable que proporcione la capacidad de interactuar con servicios del sistema o funciones de librería, por ejemplo poder obtener la fecha y hora del sistema.

5. *Facilidad de uso*: El lenguaje debe mostrar similitud con los lenguajes de programación más habitualmente utilizados.

6. *Expresividad*: Por último, se debe proporcionar la expresividad necesaria para dar soporte a las necesidades del dominio concreto; la expresividad suficiente puede ser una solución de compromiso con respecto a la facilidad de uso.

Teniendo en cuenta las anteriores características citadas hemos analizado una serie de lenguajes y herramientas para la transformaciones de modelo a texto que pudieran ser válidas para el presente trabajo.

2.3.1. Acceleo

Es una herramienta para la creación de generadores de código, que implementa la especificación del lenguaje estándar para las transformaciones de modelo a texto desarrollado por la OMG y que se denomina MOFM2T [OMG08]. Es suministrado como un plugin de Eclipse con licencia EPL y desarrollado en lenguaje Java bajo el marco de trabajo Eclipse Modelling Framework (EMF) [SBP+08].

El lenguaje MOFM2T utiliza un enfoque basado en plantillas. A través de este enfoque, una plantilla es un texto que contiene una parte dedicada donde el texto se genera a través de los elementos que provienen de los modelos de entrada, en estas partes de la plantilla se nos proporcionan expresiones para seleccionar y extraer información de dichos modelos. Dentro de Acceleo, estas expresiones están basadas en la implementación de Eclipse del lenguaje OCL [WK03].

Algunas de las características más importantes que posee Acceleo son las siguientes:

1. Generación de código a través de cualquier tipo de metamodelo compatible con EMF como UML 1, UML 2 e incluso metamodelos adaptados a nuestro dominio específico[1].

2. Permite crear nuestro código encargado de la generación de un modo modular a través de las plantillas de MOFM2T.

3. Nos proporciona un editor con autocompletado, detección y refactorización de errores en tiempo real y resaltado de sintaxis.

4. Nos permite realizar llamadas al sistema y librerías externas a través del lenguaje Java.

En resumen es una herramienta muy completa que posee el punto fuerte de basarse en un estándar internacional de la OMG.

2.3.2. MOFScript

El objetivo de MOFScript es el desarrollo de herramientas y marcos de trabajo que soporten las transformaciones de modelo a texto. Para tal cometido nos permite utilizar cualquier tipo de metamodelo para dicha generación. El lenguaje utilizado en MOFScript fue presentado como candidato en la OMG para las transformaciones de modelo a texto, pero actualmente el estándar adoptado es el comentado en la herramienta anterior.

Esta herramienta se distribuye con licencia EPL y se presenta como un plugin de Eclipse, pero también es posible ejecutarlo como una aplicación Java independiente.

Las características más destacables que nos encontramos cuando trabajamos con esta herramienta son las siguientes:

[1] En inglés a este concepto se le denomina *Domain Specific Languages* (DSLs)

1. Generación de texto a través de cualquier modelo basado en MOF (Meta-Object Facility) de la OMG.
2. Posee la capacidad de especificar mecanismos básicos de control como bucles y sentencias condicionales.
3. Manipulación de cadenas de texto.
4. Editor con autocompletado, resaltado de sintaxis y detección de errores.
5. Invocaciones a funciones externas Java.

Como conclusión podemos decir que MOFScript es muy similar a Acceleo en cuanto a características disponibles pero tiene la desventaja de no utilizar un lenguaje estandarizado.

2.3.3. Xpand

Es un lenguaje de transformación de modelos a texto que se entrega junto a herramientas en forma de plugin para Eclipse con licencia EPL. Se basa en plantillas para la generación de código donde se pueden utilzar variables, bucles y sentencias de control; es muy similar a los dos lenguajes y herramientas analizados anteriormente. Una de las características más notables de Xpand consiste en que se distribuye junto con otra herramienta denominada Xtext destinada a la creación de lenguajes textuales de dominio específico.

Nuevamente nos encontramos con la desventaja de estar frente a una herramienta que no utiliza un lenguaje estandarizado basado en plantillas.

2.3.4. JET

Java Emitter Template (JET) es un subproyecto de Eclipse Modeling Framework (EMF) centrado en el proceso de generación automática de código a partir de plantillas para cualquier modelo basado en MOF. JET funciona a partir de plantillas muy similares a los JSP que, al igual que en esta tecnología, son traducidas a una clase Java para luego ser ejecutadas por medio de una clase generadora creada por el usuario. La principal diferencia que posee JET frente a las otras tres tecnologías analizadas consiste en que dentro de las etiquetas que describen los scripts de las plantillas se puede utilizar cualquier código Java.

2.3.5. Elección de herramienta

Tras haber analizado las principales alternativas para realizar transformaciones de modelo a texto hemos adoptado como la opción más conveniente para nuestro trabajo la de la herramienta Acceleo. Esta elección está justificada por dos razones principalmente, la primera de ellas ya la habrá podido intuir el lector al ir leyendo las descripciones anteriores, ya que Acceleo es la única que se apoya en un lenguaje estandarizado por la OMG, lo que nos proporcionará una mayor aceptación de nuestro trabajo por parte de la comunidad científica, y la segunda razón viene sugerida por la herramienta Marte2Mast [MC], creada en el contexto donde se engloba el presente trabajo (ver Sección 1.2), la cual fue desarrollada con Acceleo. Se ve conveniente por tanto el homogenizar las herramientas y sobre todo

las formas de uso que proponen el conjunto de servicios que se facilitan a la comunidad desarrollados en este enfoque.

Capítulo 3

Generación de código

Este capítulo describe los criterios del diseño y los conceptos base que son soportados por el generador de código, incluye también las decisiones tomadas para realizar las transformaciones del modelo UML a código Ada y explica brevemente la dinámica de implementación del generador con la herramienta Acceleo y su lenguaje MOF2T.

3.1. Principios fundamentales del diseño

Se enuncian algunos de los principios que han servido de guía para la especificación de las transformaciones a realizar y los criterios de diseño empleados.

3.1.1. Conceptos de ingeniería de software que son soportados

El objetivo fundamental de la utilización de modelos conceptuales para el diseño de sistemas de tiempo real es la gestión de la complejidad y el máximo aprovechamiento de los resultados y criterios empleados a lo largo del proceso de desarrollo. Las metodologías de concepción de sistemas de información orientadas a objetos son las que mejor se adaptan a ello y el uso de UML para su representación nos permite documentar y almacenar los diversos resultados intermedios del proceso de desarrollo de forma ágil y con ciertas facilidades para su reutilización.

Sin embargo no todas las facilidades de UML, ni todas las potencialidades conceptuales del modelado y análisis de sistemas orientados a objetos pueden ser plasmadas finalmente en el lenguaje de programación objetivo, en este caso Ada. Por ello se han seleccionado aquellos que mejor se pueden soportar en Ada y que facilitan tanto la reutilización de los modelos como la equivalencia semántica entre ambos formalismos.

Entre los principios fundamentales que han de ser soportados se encuentran: La clasificación o tipificación en clases con atributos y operaciones tanto de clase como de objeto, la asociación entre clases, la herencia, el polimorfismo, la ocultación de información (también llamada visibilidad), las dependencias entre clases, la definición de interfaces, y clases abstractas, y la de módulos de mayor entidad (paquetes).

Por otra parte de cara a la utilización de esta herramienta en el diseño de sistemas

concurrentes de tiempo real, se incluyen también las tareas y la definición de secciones críticas de acceso mutuamente exclusivo a recursos pasivos mediante objetos protegidos.

Finalmente, a fin de poder generar el código que define la arquitectura funcional de la aplicación en consonancia con su respectivo modelo de análisis, se debe extraer el código detallado de los modelos de actividad de cada operación y clase del sistema. La unidad base para la conformación de estos modelos es lo que se denomina la acción (action) en UML. A fin de facilitar la generación de código desde cualquier entorno de desarrollo se considera cada acción como una posible línea de código. Sin embargo a fin de facilitar la generación automática de código a partir de modelos que reflejen directamente algoritmos en el modelo de actividad, la herramienta deberá ser capaz de identificar y extraer el código correspondiente al encaminamiento condicional de líneas de flujo de control (*branches* o sentencias *If-then-else*) y la formación de lazos simples basados en tales sentencias condicionales (loops).

3.1.2. Creación de modelos para la generación de código

Como se ha mencionado, la base para la conformación del código correspondiente a cada modelo de actividad son las acciones que se encuentran encadenadas en secuencias de flujo de control. La semántica de cada acción de un modelo de actividad es significativamente variable, y en aras de la flexibilidad que es necesaria para la creación del software es desde luego deseable que sea así. Sin embargo debemos observar que esta flexibilidad abre una posible brecha en la garantía de analizabilidad del sistema y la equivalencia del código generado con su respectivo modelo de análisis.

Una forma de limitar y controlar este posible problema de consistencia semántica es el no permitir la introducción de acciones arbitrarias, sino más bien limitar las acciones a por ejemplo sólo llamadas entre operaciones del sistema. En cualquier caso esta forma de construcción del modelo se puede garantizar más fácilmente mediante el uso de las transformaciones de modelo a modelo que se proponen en el contexto de desarrollo previsto (ver sección 1.3). Así cualquier decisión en la generación de código que no sea suficientemente versátil o amigable para un usuario que introduzca el modelo de forma manual en UML puede ser gestionada o automatizada mediante la generación del modelo UML para la generación de código Ada a partir del modelo de alto nivel.

3.1.3. Modelos de alto nivel con el perfil MARTE

Así, un criterio básico para la definición del generador de código es que facilite tanto la automatización de la generación de código Ada funcional a partir de modelos UML introducidos manualmente por usuarios habituales de Ada, como la del correspondiente a modelos automáticamente generados para aplicaciones concurrentes de tiempo real que han de ser analizadas mediante herramientas de análisis de planificabilidad. UML es un lenguaje de modelado de propósito general, sin embargo es bastante rico en elementos de modelado propios de la definición de software, y en cierta forma por ello especializado en metodologías de modelado conceptual orientadas a objetos. Su utilización en multitud de otros dominios de la técnica se ha canalizado mediante la definición y estandarización de perfiles que lo

especializan puntualmente para cada uno de ellos. En el caso de los sistemas embebidos y de tiempo real la especialización correspondiente es el perfil MARTE. Así los conceptos y "vocablos"de modelado que este perfil introduce son los adecuados para la especificación análisis y programación de uso habitual en este dominio. En nuestro caso, es importante destacar los elementos de modelado del capítulo correspondiente al modelado de aplicaciones de alto nivel (HLAM) puesto que aportan los necesarios para modelar aspectos tales como la concurrencia y el acceso mutuamente exclusivo a recursos comunes. Por esta razón, aún para la generación de código Ada que no tenga requisitos de tiempo real pero que puede emplear estos conceptos, conviene considerar que se tienen a disposición los elementos básicos de modelado que este perfil aporta. Se empleará sin embargo lo mínimo indispensable para garantizar la coherencia semántica del código con el modelo original.

3.2. Modelo UML para la generación de código

3.2.1. Conceptos y elementos principales de los diagramas de clase en UML

Vamos a detallar a continuación la semántica de los diagramas de clases, las clases y los paquetes, para que los lectores menos familiarizados con el lenguaje de modelado UML entiendan los principales conceptos utilizados para la descripción estructural de un sistema.

Clases y diagramas de clase en UML 2.0

El elemento principal de este tipo de diagramas es la clase [OMG11b] y se define como una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica. Dichas clases son representadas gráficamente por cajas con compartimentos para:

1. Nombre de la clase.
2. Atributos.
3. Operaciones/métodos.

Cuando hablamos de atributos y métodos en las clases de UML debemos diferenciar dos tipos:

1. De objeto: Son aquellos que pertenecen a una instancia de la clase. Es decir, existirán siempre y cuando al objeto al que pertenecen exista.
2. Estáticos: Se definen como aquellos que son compartidos por todos los objetos, por lo tanto, todos los objetos verán la misma copia del atributo u operación en cuestión.

Una clase puede determinar que operaciones y datos serán revelados a las otras clases presentes en su diagrama. Esto lo hará a través del concepto de visibilidad. De tal modo que UML proporciona 4 tipos de visibilidad:

- Pública: Todos aquellos que puedan ser accedidos directamente desde cualquier clase.
- Protegida: Los elementos declarados como protegidos, podrán ser accedidos por métodos que son parte de su clase y también por elementos que hayan sido declarados en alguna clase que herede de ella.
- Paquete: Todos aquellos elementos que pueden ser accedidos por cualquier clase perteneciente al mismo.
- Privada: Solo la clase que contiene al elemento privado puede ver o trabajar con el atributo u operación declarada bajo esta visibilidad.

Las clases son utilizadas para describir la estructura de un sistema a través de los diagramas de clases. De este modo UML nos proporciona más mecanismos para que seamos capaces de definir correctamente un sistema, por ello un diagrama de este tipo se compone de las clases descritas anteriormente y de las relaciones que puedan existir entre ellas:

- Dependencia: Dicha relación se establece cuando alguna operación o método de una clase es utilizada por otra.
- Asociación: Indica referencia a objetos entre clases en forma de un atributo. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.
- Agregación: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye.
- Composición: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye.
- Generalización: Indica que una subclase hereda los métodos y atributos especificados por una clase padre [1], por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la clase padre (público y protegidos).

La representación gráfica en UML de las relaciones entre clases descritas anteriormente se ilustra en la figura 3.1.

En la siguiente sección vamos a describir el concepto de paquete en UML, que nos ayudará a encapsular y agrupar a las clases aquí descritas.

Paquetes en UML 2.0

Cuando la complejidad del diseño de un modelo crece, es normal agrupar las clases a través de los paquetes UML. La mayoría de los lenguajes orientados a objetos tienen una analogía de los paquetes UML para organizar y evitar colisiones de nombre entre clases. De tal modo, que las clases se agruparán en paquetes para indicar que corresponden a

[1] También denominada Super Clase

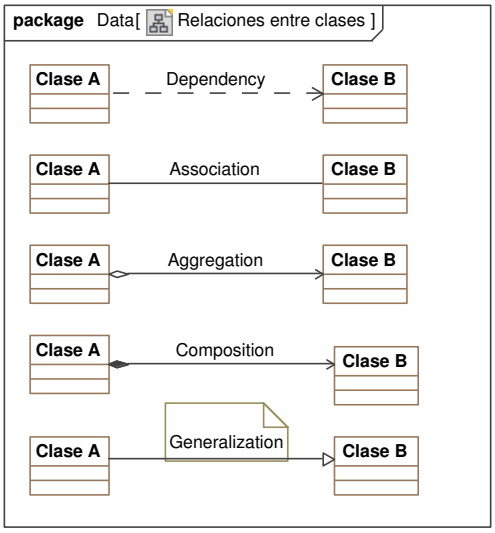

Figura 3.1: Representación gráfica en UML 2.0 de las distintas relaciones entre clases.

una característica específica del sistema, consiguiendo una modularidad y una reutilización mayor. Al igual que ocurría con las clases, los paquetes pueden presentar relaciones de dependencia entre sí.

Después de haber estudiado brevemente algunos de los elementos que nos permiten modelar la parte estructural de cualquier aplicación o sistema vamos a describir los principales elementos UML utilizados en los diagramas de actividad para el modelado del comportamiento.

3.2.2. Diagramas de actividad

Los diagramas de actividad de UML nos permiten especificar procesos, para ello representan paso a paso los flujos de operación de los componentes en un sistema. Son más expresivos que los diagramas de flujo y también heredan características de los diagramas de estado, los diagramas de flujo de datos y las redes de Petri.

Debido al alcance de este trabajo solo vamos a describir los principales elementos que se utilizan en el generador de código aquí desarrollado, para ello hemos ilustrado la figura 3.2 con un diagrama de actividad y sus elementos. En primera instancia tenemos el concepto de actividad que es un conjunto de acciones que modelan un proceso, se representa con un rectángulo con los bordes redondeados. Dentro de la actividad por cada paso tendremos lo que se denomina una acción, las acciones se conectan con otros elementos mediante flechas denominadas *control flows*. Además podemos tener *nodos de decisión*, que nos permiten especificar toma de decisiones en función de la *guarda* asociada a los control flows que salen del mismo, con este tipo de elementos se pueden modelar sentencias de control o bucles.

Tras haber descrito los ejes principales de cualquier diagrama de clases y diagrama de actividad, en el siguiente apartado se procederá a enumerar qué elementos UML son

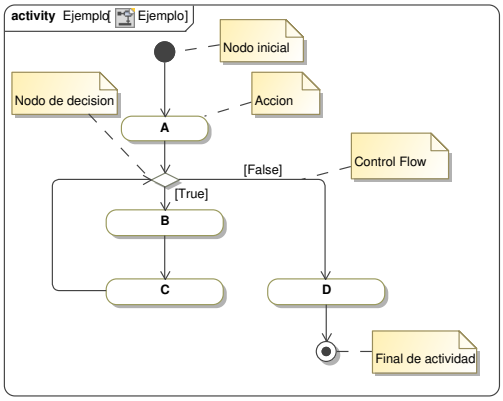

Figura 3.2: Ejemplo de diagrama de actividad con sus principales elementos nombrados.

soportados a través del generador de código desarrollado en este trabajo.

3.2.3. Elementos de UML soportados

Vamos a listar los elementos UML que son soportados por el generador de código para la parte estructural(diagramas de clases) y para la parte de comportamiento(diagramas de actividad).

Comenzamos con los elementos de los diagramas de clase:

1. Clase: dentro de ésta se soportan atributos y operaciones/métodos con sus correspondiente visibilidad. Además se soportan los siguientes tipos de clases:

   Clases abstractas.

   Clases activas.

   Clases con el estereotipo *PpUnit* del perfil Marte.

2. Paquetes: con la capacidad de encapsular otros paquetes o clases.

3. Relaciones entre clases: dependencia, asociación y generalización.

4. Interfaces: con su correspondiente relación denominada *realización* entre clase e interfaz.

Por otro lado en los diagramas de actividad se soporta:

1. Nodo inicial.

2. Acciones opacas.

3. Control flows.

4. Nodos de decisión.

5. Nodo final de actividad.

La siguiente sección explicará cómo trasladar los elementos UML vistos hasta ahora a código Ada 2005.

3.3. Transformaciones de UML a código Ada

3.3.1. Paquetes y Clases UML 2.0 en Ada 2005

Ada proporciona características de Orientación a Objetos de herencia y polimorfismo a través de los tipos etiquetados denominados *tagged types* en inglés. Con esta clase de tipos podemos implementar parte del concepto de clase presentado en UML 2.0. Concretamente con el tipo *tagged* podemos generar atributos y métodos de objeto, mientras que los estáticos, que están ligados a la clase y se almacenan en un espacio de memoria común para todos los elementos y objetos de la clase, no pueden ser generados a través del tipo *tagged*. Para ello debemos hacer uso del concepto de paquete en Ada. En Ada el concepto de paquete no es totalmente equivalente al de UML 2.0, pero puede ser utilizado para definir módulos que son colecciones de entidades (tipos, constantes, subprogramas...). Podemos utilizarlo por tanto para encapsular y definir las clases, de tal modo que por cada clase tendremos un paquete que encapsula un tipo *tagged* con todos sus atributos y métodos de objetos. Los elementos de clase estáticos serán definidos también a nivel de paquete para que se correspondan con el concepto de clase descrito en UML 2.0.

Teniendo en cuenta todo lo citado anteriormente debemos manejar además los aspectos de visibilidad ilustrados en la sección 3.2.1 y la visibilidad proporcionada en los paquetes Ada. De tal modo que vamos a detallar cuál es el ámbito de visibilidad que se nos presenta en éstos últimos:

- Los paquetes en Ada se describen mediante dos archivos, uno de ellos contiene la especificación [2] o también llamada interfaz y el otro posee el cuerpo en el que se especifica toda la funcionalidad [3].
- Todo lo que se encuentra dentro del cuerpo del paquete no puede ser accedido desde fuera del mismo. Por lo tanto tiene una visibilidad privada.
- Dentro de la especificación tenemos dos partes:
  - Una de ellas es pública, es decir, desde fuera del paquete podemos acceder a los elementos definidos en ella.
  - La parte que se encuentra por debajo de la cláusula *private* solo puede ser accedida desde fuera del paquete cuando el que intenta realizar el acceso es un elemento de un paquete hijo.

Para ilustrar de un modo más gráfico los diferentes ámbitos de visibilidad dentro de un paquete en Ada se ha añadido la figura 3.3.

[2] Se almacenan con la extensión .ads
[3] Se almacenan con la extensión .adb

Fichero name.ads

```
package name is
      --Visibilidad pública
private
      --Visibilidad protegida
end name;
```

Fichero name.adb

```
package body name is

      --Visibilidad privada

end name;
```

Figura 3.3: Ámbito de visibilidad en las diferentes partes de un paquete Ada.

Con los mecanismos que ofrece Ada 2005 no podemos tener una equivalencia directa con UML en el ámbito de la encapsulación de las clases a través de los paquetes y de la visibilidad (ya que no existe el concepto de visibilidad de paquete). Debido a esto hemos propuesto y valorado las siguientes soluciones para encapsular y trabajar con las clases:

1. *Toda clase contenida dentro de un paquete, será definida como un paquete hijo de éste:* Con esta solución conseguimos obtener una encapsulación similar a la expuesta en UML, pero aparecen problemas cuando existe herencia entre clases contenidas en distintos paquetes, ya que un paquete en Ada solo puede ser hijo de un solo paquete. Tal y como se puede ver en la figura 3.4 la clase B debería estar contenida en un paquete que fuese hijo del *packageS* que la contiene y de la clase A, de la cuál heredaría la parte protegida y pública. Por consiguiente, descartamos esta solución porque preferimos dar soporte a la herencia en vez de a la continencia.

2. *Toda clase contenida dentro de un paquete, será definida dentro de éste:* Esta solución puede parecer la más natural, pero aparecen nuevamente problemas cuando se realiza herencia entre clases contenidas en distintos paquetes. Utilizando el ejemplo ilustrado en la figura 3.4 nos damos cuenta que el paquete que define a la clase B debería ser hijo del paquete que conforma a la clase A, pero esto no es posible, ya que no se pueden crear paquetes hijos de paquetes contenidos dentro de otros. Como en la primera solución aportada nos ocurría, la descartamos por dar preferencia a la herencia frente a la encapsulación.

3. *Utilización de espacios de nombres para representar a los paquetes UML:* Con esta solución no creamos ningún paquete que contenga clases, únicamente se utiliza el concepto de paquete Ada para encapsular a las clases a través de su espacio de nombres. Por lo tanto, cuando nos encontremos una situación como la ilustrada en la figura 3.4, el paquete que forma la clase B, pasará a llamarse *packageS_Bs* y el paquete para la clase A pasaría a llamarse *packageR_As*. De este modo todas las clases que estén contenidas en paquetes tendrán un espacio de nombres común. Con esta solución podemos trabajar correctamente con todo tipo de herencia entre clases. Resuelto el tema de la encapsulación y la herencia aparece el problema de la visibilidad de paquete. Si quisiésemos trasladar esta visibilidad Ada necesitaríamos crear un nuevo paquete por cada paquete contenedor de clases para poder definir en su parte pública los métodos y atributos protegidos descritos en el modelo UML, esto dificultaría enormemente tener un diseño modular, reutilizable y verificable por lo que hemos decidido trabajar

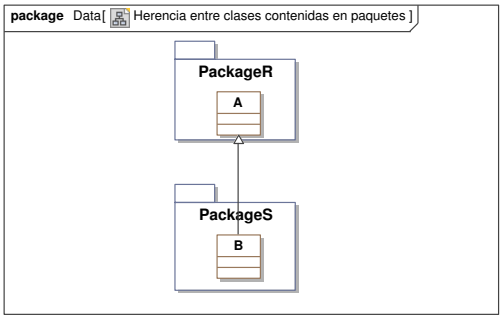

Figura 3.4: Herencia entre clases contenidas en distintos paquetes.

con los atributos y métodos con visibilidad de paquete como si fuesen públicos, salvo que delante de su nombre original le añadiremos el prefijo denominativo *package_* para indicar al usuario su visibilidad originaria. Esta alternativa de solución es la que hemos adoptado en este trabajo.

Teniendo presente la solución adoptada anteriormente y las características de visibilidad, vamos a describir el patrón utilizado para generar las clases:

1. Por un lado tenemos los elementos estáticos que serán establecidos del siguiente modo en base a su visibilidad:

   - Elementos públicos: Se colocarán en el fichero de especificación del paquete en su parte pública.
   - Elementos protegidos: Serán establecidos en el fichero de especificación en su parte privada.
   - Elementos privados: Serán dispuestos en el fichero que describe el cuerpo del paquete.
   - Elementos con visibilidad de paquete: Se colocarán igual que los elementos públicos, y se les renombra usando el prefijo *package_*.

2. Dentro de los elementos de objeto, debemos diferenciar entre métodos y atributos, ya que estos últimos son definidos en el *record* del tipo *tagged* y solamente se pueden establecer una única vez y con una visibilidad igual para todos ellos. Mientras que las operaciones o métodos pueden colocarse en diferentes lugares con distintas visibilidades.

   Teniendo esto presente para los atributos debemos seguir el siguiente patrón:

   En la especificación del paquete contenedor de la clase:

   a) Para los atributos públicos creamos un tipo *tagged* en la parte pública como el descrito a continuación:

```
01    Type Public_Part is abstract tagged record
02      --Todos los atributos públicos y no estáticos
03    end record;
```

b) Declaramos un nuevo tipo en la parte pública que será el encargado de contener a todos los atributos y que por ello heredará del declarado en el paso anterior, además se añadirá un puntero a este tipo declarado para posibles referencias que se puedan dar desde elementos externos.

```
01    Type Nombre_de_la_clase is new Public_Part with private;
02    Type P_Nombre_de_la_clase is access Nombre_de_la_clase;
```

c) Para declarar los atributos privados debemos crear un tipo sin definir (será definido en el cuerpo del paquete para cumplir con la visibilidad privada) en la parte privada que encapsula a la clase, a su vez es necesario definir un puntero a este tipo declarado y por último un nuevo tipo que contenga un elemento que sea del tipo puntero anterior. De este modo podemos conseguir que todos los elementos privados solo sean accesibles dentro de la clase. Por lo tanto, deberíamos generar un código como el siguiente:

```
01    Type Private_Part; --Se definirá por completo en el cuerpo
02    Type Private_Part_Pointer is Access Private_Part; --Puntero
03    Type Private_Component is record
04          P: Private_Part_Pointer;
05    end record;
```

d) En la parte privada debemos definir el *record* para el tipo que se encarga de contener a todos los atributos, por ello deberemos tener lo siguiente:

```
01    Type Nombre_de_la_clase is new Public_Part with record
02          -- Declaramos todos los atributos protegidos aquí
03          P: Private_Component; --Contendrá los elementos privados
04    end record;
```

Para el cuerpo del paquete debemos realizar lo siguiente:

a) Definimos el tipo para los elementos privados que declaramos en la especificación del paquete:

```
01    Type Private_Part is record
02          -- Declaramos todos los atributos privados aquí
03    end record;
```

También debemos de inicializar, ajustar y finalizar la parte privada, ya que la clase mostrará al exterior el tipo *Nombre_de_la_clase* como contenedor de los atributos privados, protegidos y públicos. Por lo tanto de algún modo debemos asignarle memoria y un valor al elemento del tipo *Private_component* que definimos. Para ello Ada nos ofrece un paquete denominado *Finalization* [TDB+06] que contiene una serie de procedimientos que nos permiten gestionar la inicialización, ajuste y finalización de las variables y tipos definidos en un paquete.

Al igual que hemos hecho con los atributos, también hemos seguido un patrón para generar los métodos descritos en el modelo:

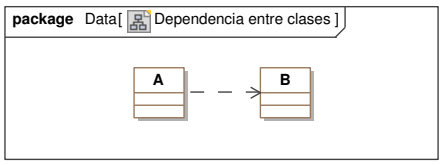

Figura 3.5: Dependencia entre clases.

En la especificación del paquete contenedor de la clase:

a) Todos los métodos públicos son colocados en la parte pública y se les pasa como parámetro un objeto del tipo *Nombre_de_la_clase* del tipo polimórfico *class-wide type* [TDB+06] definido anteriormente.

b) Todos los métodos protegidos son definidos en la parte privada y se les pasa como parámetro un objeto del tipo *Nombre_de_la_clase* definido anteriormente.

En el cuerpo del paquete contenedor de la clase:

a) Todos los métodos privados son definidos en el cuerpo y se les pasa como parámetro un objeto del tipo *Nombre_de_la_clase*.

3.3.2. Dependencias entre clases

Las clases por lo general necesitan unas de otras, de tal modo que en UML se nos presenta el concepto de dependencia entre clases. Este tipo de relación entre dos clases declara que una clase necesita conocer a la otra para utilizar objetos de esta última. En UML se representa mediante una flecha punteada direccional. A modo de ejemplo podemos ver en la figura 3.5 como la clase *A* depende de la clase *B* por lo que deberá conocer o tener acceso a los elementos que contenga. Además hay que tener en cuenta que la dependencia tiene visibilidad, es decir, puede ser de alguno de los cuatro tipos que ofrece UML (*public*, *private*, *protected* y *package*).

En este caso el modo de transformar las dependencias de UML a código Ada se puede realizar utilizando la cláusula *With* del lenguaje. Por lo tanto por cada dependencia que una clase tenga hacia otra tendremos que añadir la cláusula *With* teniendo en cuenta la visibilidad de la dependencia, por ello tenemos un pequeño inconveniente, ya que la visibilidad *protected* y *package* no se puede trasladar con el *With* de Ada. Por consiguiente decidimos que si tenemos una dependencia con visibilidad *protected* o *package* la trataremos como si fuese *public*. Habiendo solucionado este problema lo que haremos es añadir las dependencias con visibilidad pública, protegida o de paquete al comienzo de la especificación del paquete contenedor (en el archivo *.ads*) de la clase, mientras que las que tienen visibilidad privada se añadirán al comienzo del cuerpo del paquete[4].

Además de la dependencia explicitada gráficamente mediante la flecha punteada también tendremos en cuenta aquella que deriva de los argumentos de las operaciones de la clase

[4]En el archivo .adb

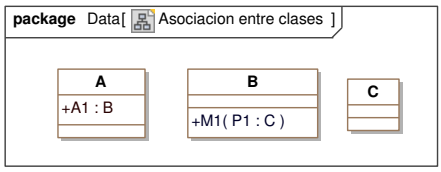

Figura 3.6: Asociación entre clases.

que sean del tipo de algún elemento del modelo descrito por el diseñador, es decir, que para este caso también se añadirá la cláusula *With* en la especificación del paquete contenedor de la clase hacia el tipo del argumento .

3.3.3. Asociación entre clases

Es otro tipo de relación entre clases con carácter más fuerte que las dependencias comentadas anteriormente. Esta relación significa que una clase contiene una referencia a un objeto, u objetos, de otra clase en forma de atributo. En UML la asociación se representa como una línea continua entre dos clases en la que sus extremos se describe la cardinalidad. En esta clase de relaciones puede existir bastante ambigüedad para obtener código fuente a través del modelo si no completamos correctamente la asociación (cardinalidad, dirección...). Por lo que en nuestro modelo de entrada hemos decidido que para simplificar la implementación de la asociación, ésta estará implícita en cualquier atributo de la clase que sea de un tipo de otra clase que esté en el modelo. De tal modo, que en la figura 3.6 podemos observar como la clase A tiene una asociación con la clase B por el hecho de que su atributo $A1$ es del tipo B. Por otra parte la clase B al mismo tiempo tiene una dependencia de la clase C por tener un argumento de sus operaciones del tipo C, es decir que sin necesidad de utilizar la notación gráfica de la línea presentada en UML podemos describir el mismo concepto.

Por consiguiente, por cada atributo que encontremos del tipo de otra clase que esté en nuestro modelo necesitamos añadir el nombre cualificado para ese tipo y la cláusula *With* de Ada para tener acceso a la clase.

3.3.4. Interfaces

En Ada 2005 [TDB+06, Bar06] se añade al lenguaje el concepto de *Interfaz*. Una interfaz no puede tener componentes y tampoco puede tener operaciones concretas, de tal modo que no podemos declarar objetos para una interfaz. Aunque no tengamos ninguna operación concreta podemos declarar procedimientos concretos a través de las clases específicas que implementen dichas interfaces.

En UML la semántica de una interfaz es parecida a la desarrollada en Ada 2005, ya que una interfaz declara un conjunto de operaciones pero no especifíca como serán implementadas, dejando este trabajo a las clases que la implementen. Tal y como se puede observar esto es idéntico en ambos casos, pero la diferencia reside en los atributos. Mientras que en Ada 2005 solo podemos definir procedimientos y funciones en las interfaces, en UML se nos

permite crear atributos. De tal modo, que hemos tenido que desarrollar una solución para adaptar ambas implementaciones.

Las interfaces en Ada son definidas dentro de un paquete del siguiente modo:

```
01  Package Package_interface
02     Type Int is Interface;
03     procedure Op (X: Int);
04   end Package_Interface;
```

Por lo tanto por cada interfaz definida en un diagrama de clases de UML generaremos un paquete denominado *Package_nombre_de_la_interfaz* con un *Type* del tipo *Interface* y los distintos procedimientos o funciones que correspondan. Aquellos métodos UML que estén definidos como estáticos no serán añadidos al paquete contenedor de la Interfaz, únicamente se tendrán en cuenta para las clases que implementen dicha interfaz, ya que se comprobará que los métodos estén definidos en la clase. Si en la interfaz de UML aparece algún atributo, tan solo se verificará que esté presente en las clases que implementen dicha interfaz debido a que las interfaces definidas en Ada 2005 no pueden contener ningún tipo de componente de datos, es decir que se comprobará que tenga los atributos, y en el caso de no ser así se añadirá un comentario advirtiendo al usuario de ello.

3.3.5. Clases abstractas

Una clase abstracta es aquella de la que no se pueden generar instancias, es decir objetos. Por ello forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma. Además algunos de los métodos de dicha clase pueden no tener definida su implementación. De este modo las subclases de una clase abstracta podrán implementar el cuerpo de las operaciones abstractas.

En Ada para definir un *tagged type* como abstracto debemos de añadrile la cláusula *abstract* en su definición, al igual que ocurre con las procedimientos y funciones. En el siguiente fragmento de código se ilustra un ejemplo:

```
01   type Object is abstract tagged ...;
02   function Function_Class_Member return Object is abstract;
03   procedure Procedure_Class_Member (X: Object) is abstract;
```

En la generación de código nos encontramos con un pequeño inconveniente, ya que en nuestro modelo UML podemos marcar que una operación de una clase sea abstracta y la clase no estar definida como abstracta. Si tradujésemos esto directamente a código obtendríamos un error de compilación, ya que en Ada cualquier procedimiento o función abstracta perteneciente a un *tagged type* implica que éste debe ser un tipo abstracto. Por consiguiente, debemos comprobar si una clase contiene operaciones abstractas y en tal caso definir automáticamente como abstracto el *tagged type* que la implementará.

3.3.6. Elementos concurrentes y de tiempo real

En esta parte vamos a describir las transformaciones desde el modelo hacia código Ada de dos elementos básicos de concurrencia y sistemas de tiempo de real como son las tareas y los objetos protegidos.

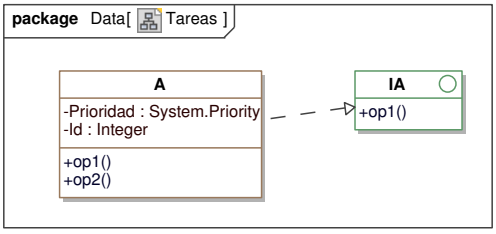

Figura 3.7: Ejemplo de clase activa en UML representando una tarea.

Tareas

Normalmente en los sistemas de tiempo real se ejecutan varias tareas, y cada tarea ejecuta una actividad de forma repetida, además debido a que este tipo de sistemas controlan actividades del mundo exterior que son simultáneas, las tareas se deben ejecutar en paralelo(concurrentemente). Por consiguiente, son un elemento esencial y necesario de cualquier sistema de tiempo real.

Habitualmente cuando modelamos sistemas con UML y no estamos trabajando bajo el paradigma de concurrencia nos encontramos que únicamente existe un elemento activo y con vivacidad por si solo, que usualmente es el programa principal encargado de arrancar y ejecutar todo el sistema. Pero como ya hemos dicho, en los sistemas de tiempo real necesitamos poder especificar tareas, para ello debemos buscar algún mecanismo de UML para realizar esta labor. En un primer momento la primera idea que se nos puede venir a la cabeza es la utilización del elemento *RtUnit* (ver sección 2.1) del perfil MARTE, pero hemos decidido buscar en el propio UML alguna característica que nos permita identificar que una clase contenga una tarea, ya que de este modo el generador será más versátil y permitirá ser utilizado más fácilmente en entornos en el que no se tienen requisitos de tiempo real.
Cuando definimos una clase en UML nos encontramos con la opción de determinar si es activa o no, de este modo consideraremos que cualquier clase que tenga está propiedad con valor a verdadero contendrá una tarea dentro de sus atributos.

Las tareas en Ada pueden tener una serie de discriminantes o argumentos de inicialización para establecer su identificador, prioridad, punteros a procedimientos o funciones, etc; por lo tanto si una clase UML es activa consideramos que todos los atributos definidos en ella, excepto los que vengan impuestos por la implementación de una interfaz si se da el caso, serán considerados como los discriminantes de la tarea. Además es importante tener en cuenta que para establecer la prioridad de una tarea debemos utilizar el concepto de *pragma* de Ada 2005 [BW07], de este modo el primer argumento de una tarea que sea del tipo *System.Priority* será considerado como la prioridad que queremos asignar.

Por otro lado es frecuente que las tareas interaccionen entre sí, por lo que necesitan de algún mecanismo para comunicarse y sincronizarse, debido a esto Ada presenta un mecanismo que se conoce como *rendezvous*[5] o punto de entrada de la tarea. La cita entre dos tareas se produce como consecuencia de la llamada de una tarea a un punto de entrada declarado

[5]Palabra adoptada del francés que se traduce como cita en castellano

en otra tarea. La sintaxis en Ada es similar a la de la declaración de los procedimientos salvo que ahora se encuentran dentro de una tarea y se designan con la palabra reservada *entry*. Al igual que con los atributos, cualquier operación de una clase activa y que no implemente ninguna de una interfaz será considerada como *entry* de la tarea.

Para clarificar los conceptos se ilustra en la figura 3.7 una clase activa que posee dos atributos y dos operaciones, una de ellas impuesta por la interfaz. Si seguimos las directrices citadas anteriormente nuestro generador creará el siguiente código(en la especificación de la clase) para el ejemplo de la figura:

```
01 ...
02  task type Task_A(Prioridad : System.Priority; Id : Integer)
03  is
04      pragma Priority(Prioridad);
05      entry op2;
06  end Task_A;
07
08  type P_Task_A is access Task_A;
09
10  type Public_Part is abstract tagged  record
11      T : P_Task_A;
12  end record;
13  type A is new Public_Part and IAs.IA with private;
14  type P_A is access A;
15  procedure op1 (Self : in out A'Class);
16  ...
```

Cabe destacar que para los argumentos de la tarea que se encuentren dentro de nuestro modelo, es decir que sean del tipo de alguna de las clases definidas por el usuario, se utilizarán los punteros que apuntan a la clase que se definieron internamente en ella.

Objetos protegidos

Otra forma de conseguir sincronización y a la vez exclusión mutua son los objetos protegidos que posee el lenguaje Ada. Este tipo de elementos encapsula datos y permite el acceso a ellos solo a través de procedimientos, funciones o entradas protegidas (*entry*), además el lenguaje nos garantiza que se ejecutarán de tal forma que los datos se actualizan en exclusión mutua.

UML no contiene ningún elemento que se pueda utilizar para representar el concepto de objeto protegido aquí descrito, por lo que en este caso es necesario utilizar el perfil MARTE y su estereotipo *PpUnit* (ver sección 2.1). De tal modo, que la clase que esté estereotipada con *PpUnit* implicará que todos sus elementos estarán protegidos, es decir, los atributos de la clase se traducirán como datos del objeto protegido y para las operaciones de la clase deberemos distinguir tres casos:

1. **Entry**: La característica de este tipo de operaciones en los objetos protegidos de Ada es que poseen una condición declarada con la cláusula *when* para poder ejecutarse. Por consiguiente hemos decidido que a través del diagrama de actividad asociado a la operación determinaremos si posee condición de guarda, por lo que si desde el nodo

inicial a la primera acción tenemos declarada alguna guarda, la operación será una *entry* con dicha guarda.

2. **Funciones**: Toda aquella operación que tenga un valor de retorno y todos sus argumentos tengan dirección de entrada será considerada una función del objeto protegido.

3. **Procedimientos**: Cualquier operación que no cumpla con las características de los dos puntos anteriores (ni condición de guarda, ni valor de retorno) será interpretada como un procedimiento en el objeto protegido.

Por último para este caso tendremos un *tagged type* que contendrá un puntero que referenciará al objeto protegido descrito anteriormente.

3.3.7. Procedimiento principal

En toda aplicación necesitamos de un procedimiento que sea capaz de inicializar todos los elementos para que el sistema puede comenzar a ejecutarse de un modo correcto. Este procedimiento debe tener vivacidad suficiente para poder realizar esta tarea, de tal modo que utilizaremos el perfil MARTE y su estereotipo *RtUnit* para detectar que una clase de UML se encargará de realizar tal cometido. Simplemente por el hecho de que una clase esté estereotipada con *RtUnit* no podemos decir que sea un procedimiento principal, por lo que *RtUnit* posee una propiedad denominada *isMain* (ver figura 2.2 de la sección 2.1). De este modo si *isMain* se encuentra a valor verdadero sabemos que el diseñador quiere que la clase sea un procedimiento principal, así la traducción a código Ada la haremos del siguiente modo, los atributos de la clase serán añadidos a la parte declarativa del procedimiento y el diagrama de actividad asociado a la clase se utilizará para obtener el cuerpo del procedimiento.

3.3.8. Comportamiento: Diagramas de actividad

Hasta ahora hemos descrito como pasar la parte estructural del modelo a código Ada. La característica distintiva de este generador es la posibilidad de crear modelos dinámicos, por ello si alguna operación de una clase, tarea u objeto protegido tiene asociado algún diagrama de actividad podremos sacar la implementación de dicha operación. De tal modo que hemos seguido una serie de pasos para hacer la traducción a código Ada:

1. En Ada las declaraciones de las variables deben realizarse en una sección específica de la operación, entre la cabecera y la cláusula *begin* o través de la sentencia *declare*. Por consiguiente, si en un diagrama de actividad existe algún comentario que comience con *–declare* o esté asociado al nodo inicial será considerado en su totalidad como el código escrito por el usuario que irá entre la cabecera de la operación y el *begin*. Se ha tomado esta decisión porque en UML no existe ninguna manera de especificar la parte declarativa de una operación.

2. Por cada acción opaca se obtendrá el texto asociado a su cuerpo y se tomará como sentencias Ada escritas por el usuario.

3. Los nodos de decisión se traducirán a sentencias de control (*if-else*) o bucles (*while*), para ello se diferenciarán dos casos; si el nodo de decisión solo tiene control flows de salida será considerado como una sentencia de control, mientras que si tiene control flows de entrada y salida será tomado como un bucle.

A continuación se describirá como se han implementado todas las transformaciones propuestas en esta sección a través de Acceleo y su lenguaje MOFM2T.

3.4. Implementación

Como ya se comentó brevemente en la sección 2.3.1 Acceleo utiliza el lenguaje MOFM2T para realizar las transformaciones de modelo a texto, además de permitir el uso de funciones externas Java. Este lenguaje estándar genera los artefactos de texto a través del uso de plantillas, las cuales están parametrizadas con elementos del modelo a transformar. Dentro de estas plantillas la principal forma de extraer la información del modelo es a través de consultas sobre las entidades del metamodelo. Esta información es convertida en fragmentos de texto usando expresiones del lenguaje para la manipulación de cadenas de texto. También es posible estructurar las transformaciones en módulos que contienen plantillas públicas o privadas. Teniendo en cuenta lo comentado hasta ahora vamos a ilustrar a través de un ejemplo muy sencillo y simplificado el funcionamiento práctico de las plantillas:

```
01  [template public classToAda(c : Class)]
02  package [c.name.concat('s')/] is
03         type [c.name/] is  tagged
04            record
05              [for (at : Property | c.attribute )]
06                   [at.name/] : [at.type.name/]
07              [/for]
08              end record;
09  end package [c.name.concat('s')/]
10  [/template]
```

Observamos como dentro de la plantilla todo lo que se encuentra dentro de los caracteres especiales '[', ']', '[/'y '/]'son elementos o funciones para extraer datos del parámetro de entrada de la plantilla, el resto del texto será imprimido en el fichero de salida tal y como está.

Tras esta breve introducción sobre el funcionamiento del lenguaje MOFM2T utilizado en Acceleo describiremos la estructura del código utilizada para la generación del código Ada. En primer lugar hemos dividido en seis módulos el generador con las siguientes finalidades:

1. Generación de clase.
2. Generación de clase con objeto protegido.
3. Generación de clase con tarea activa.
4. Generación de programa principal.

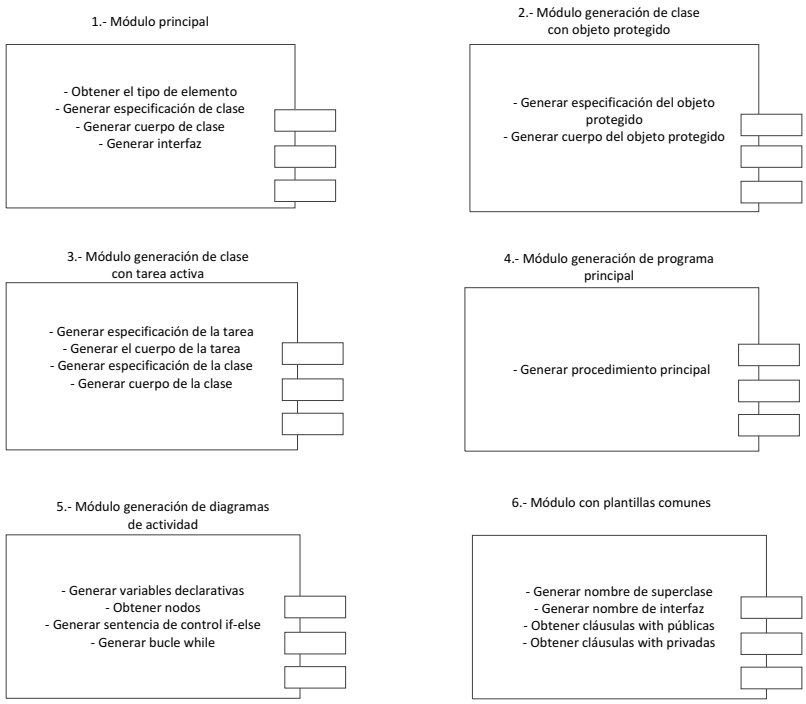

Figura 3.8: Arquitectura de los módulos con la funcionalidad de sus plantillas internas.

5. Generación de diagrama de actividad.

6. Plantillas comunes a los diferentes módulos.

Además de estos módulos se ha creado otro archivo con funciones Java para ser llamadas desde las plantillas de Acceleo y realizar operaciones complejas que en algunos casos eran difíciles o imposibles de implementar con MOFM2T.

El módulo principal es el encargado de recorrer todos los elementos del modelo y la generación de clases, clases abstractas o interfaces, en él existe una plantilla principal que recorre todos los paquetes UML del modelo de entrada, buscando en cada paquete los elementos contenidos para determinar a que plantilla llamar en función del elemento encontrado. Una vez que se ha elegido a que plantilla llamar se procede a generar los ficheros correspondientes siguiendo los patrones descritos en la sección anterior.

En la figura 3.8 se ilustra la estructura general del generador con los seis módulos de Acceleo por los cual está compuesto, y la funcionalidad de la plantillas que contiene cada uno de ellos.

Capítulo 4

Casos de estudio

En este capítulo se presentan dos casos de estudio para probar el generador de código desarrollado. El primero de ellos representa un problema típico de concurrencia en el que tendremos que utilizar tareas y hacer uso de la exclusión mutua a través de los objetivos protegidos de Ada. El segundo caso de estudio constituye un problema fuertemente ligado a la orientación a objetos y nos permite utilizar algunos elementos básicos de esta técnica de diseño y programación.

Los modelos de entrada utilizados para estos demostradores así como el código obtenido para ellos se encuentran desarrollados en su totalidad y se incluyen en el CD adjunto a esta memoria.

4.1. Problema de la cena de los filósofos

4.1.1. Especificación

Esta famosa metáfora para el manejo de recursos y bloqueos de procesos fue originalmente formulada por Edsger Dijkstra en 1971 [Dij71]. Cinco filósofos están sentados alrededor de una mesa y gastan su vida cenando y pensando alternativamente. Cada filósofo tiene un plato con comida china[1] y un palillo a la izquierda de su plato. Para comer son necesarios dos palillos y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo coge un palillo y el otro está ocupado, se quedará esperando, con el palillo en su mano, hasta que pueda coger el otro palillo, para luego comenzar a comer.

Si dos filósofos contiguos intentan tomar el mismo palillo al mismo tiempo, se produce una condición de carrera: los dos compiten por tomar el mismo palillo, y uno de ellos se quedará sin comer.

Si todos los filósofos tomasen el palillo que está a su derecha a la vez, entonces todos se quedarán esperando eternamente, porque alguien debería liberar el palillo que les falta. Ninguno lo hará porque todos se encuentran en la misma situación. Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo o deadlock.

[1] La formulación original de Dijkstra involucraba espaguetis y cubiertos de mesa, pero debido a que la mayoría de los filósofos pueden comer espaguetis con un tenedor individual muchos escritores utilizan la metáfora de la comida china en su lugar.

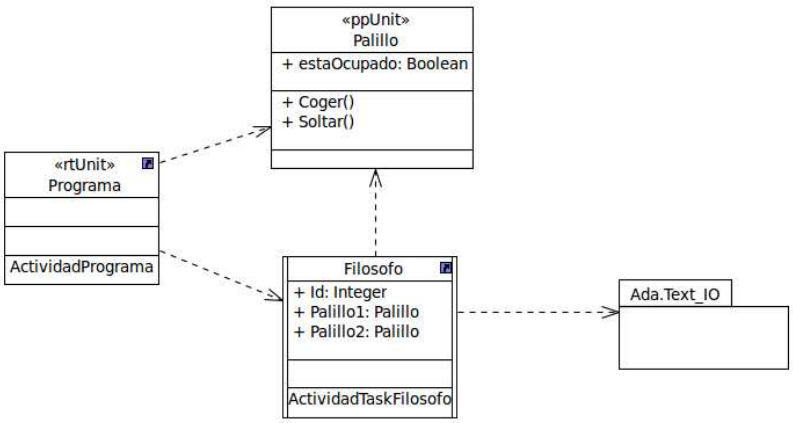

Figura 4.1: Diagrama de clases para el problema de los filósofos.

El problema consiste en encontrar una aplicación software que simule el problema y evite que los filósofos nunca mueran de hambre.

4.1.2. Análisis de la aplicación

Para llevar a cabo una solución al problema planteado en la sección anterior se ha decidido utilizar tareas para crear el comportamiento de cada uno de los cinco filósofos y la exclusión mutua para modelar el recurso compartido (el palillo). Ambos elementos son soportados por nuestro generador de código; por un lado las tareas son representadas como clases activas de UML y la exclusión mutua es alcanzada a través del estereotipo *PpUnit* del perfil MARTE (ver Sección 2.1).

En la figura 4.1 se ilustra el diagrama de clases utilizado para modelar la parte estructural del problema. Como ya se ha comentado tenemos una clase activa para representar a los filósofos, la cual tiene como atributos los dos palillos que posee cada uno de ellos y un identificador para diferenciarlos. Por otro lado los palillos son otra clase estereotipada con *PpUnit* para indicar que es un objeto protegido, además posee un atributo que sirve de barrera para acceder al recurso cuando está tomado por algún filósofo y dos operaciones para tomarlo o soltarlo. También se observa un paquete del que depende la clase *Filosofo*, el cual es un paquete externo necesario para poder imprimir texto por consola en lenguaje Ada. Y por último tenemos la clase *Programa* estereotipada con *RtUnit* con el objetivo de ser el programa principal que inicializa toda la aplicación.

Para definir el comportamiento de la aplicación hemos creado cuatro diagramas de actividad, dos de ellos para las operaciones del objeto protegido, otro para el comportamiento de cada uno de los filósofos y por último uno asociado al programa principal. A modo de ilustración se muestra en esta memoria uno de los cuatro diagramas[2], de tal modo que en

[2]En el CD adjunto a esta memoria se incluye el modelo completo así como el código obtenido.

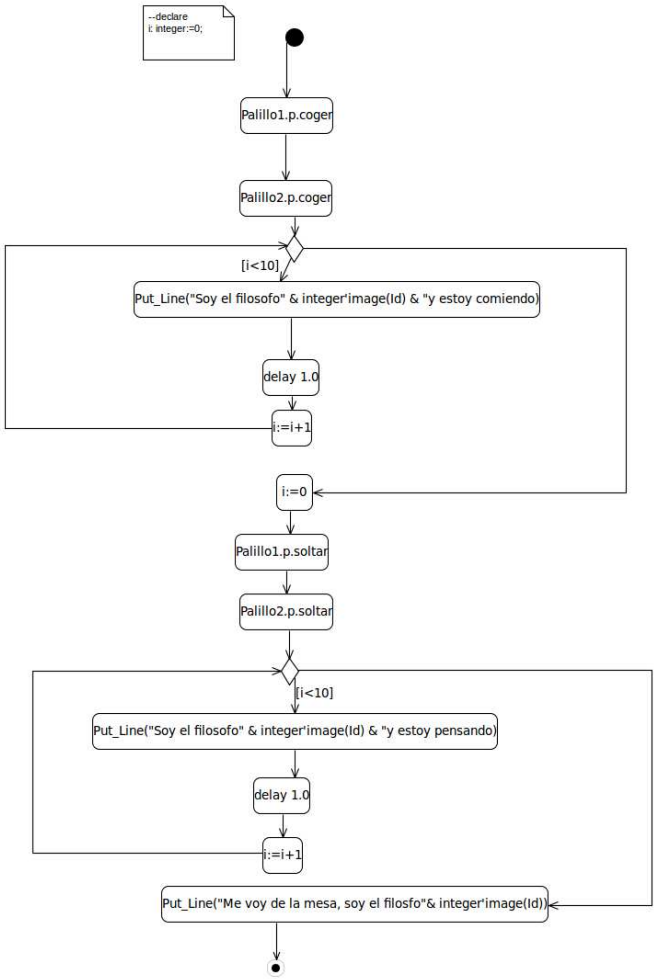

Figura 4.2: Diagrama de actividad asociado a la clase *Filósofo*.

la figura 4.2 se puede observar cómo es el flujo de trabajo asociado a la clase *Filósofo*, es decir, que este diagrama permite obtener la implementación para el cuerpo de la tarea que representa dicha clase.

4.1.3. Código obtenido

Tras tener todos los diagramas estructurales y de comportamiento asociados a las operaciones o tareas, automáticamente podemos generar código Ada compilable. Por ejemplo, para la especificación de la clase *Palillos* se ha creado el siguiente código:

```
01  package Palillos is
02      protected type protected_Palillo is
03          entry Coger;
04          entry Soltar;
```

```
05      private
06          estaOcupado: Boolean:= false;
07      end protected_Palillo;
08      type P_protected_Palillo is access protected_Palillo;
09      type Palillo  is tagged record
10          p : P_protected_Palillo;
11      end record;
12 type P_Palillo is access Palillo;
13  end Palillos;
```

Por un lado vemos un objeto protegido de Ada con dos entries, ya que ambas en su diagrama de actividad asociado tenían una condición de barrera, y por lo tanto han de ser entries y no simples operaciones.

Otra especificación generada que aquí mostraremos será la de la clase *Filosofo*, en ella podemos ver como se han añadido las dependencias presentes en el diagrama de clases a través de las cláusulas *with*:

```
01  with Ada.Text_IO;
02  with Palillos; use Palillos;
03  package Filosofos is
04
05      task type Task_Filosofo(Id:Integer;Palillo2:P_Palillo;Palillo1:P_Palillo);
06      type P_Task_Filosofo is access Task_Filosofo;
07
08      type Public_Part is abstract tagged  record
09          T : P_Task_Filosofo;
10      end record;
11      ...
```

4.2. Simulador lógico para circuitos combinacionales

4.2.1. Especificación

En este caso de estudio se pretende implementar un simulador lógico sencillo para circuitos combinacionales, es decir capaz de modelar sistemas digitales en el que sus salidas son función exclusivas del valor de sus entradas en un instante dado, sin que interfieran estados anteriores de las entradas o de las salidas. Las funciones que aquí desarrollaremos serán únicamente tres: OR, AND y NOT, no obstante el objetivo es utilizar la orientación a objetos para que se puedan añadir fácilmente nuevas características y mejoras al simulador sin necesidad de tener que rediseñar el sistema.

Para realizar el diseño nos basaremos en dos componentes básicos, el primero de ellos serán los *nudos* que representan las conexiones con valores lógicos (true o false), y el segundo serán las *puertas* que realizan combinaciones de los valores de entrada que se verán reflejadas en la salida. Ambos componentes deben ser fácilmente extensibles y por ello deberán contener operaciones y atributos que permitan identificarlos, conocer el valor actual o cambiar el valor lógico de un nudo en un instante dado, realizar la operación lógica de una puerta, entre otras que serán descritas en la siguiente subsección.

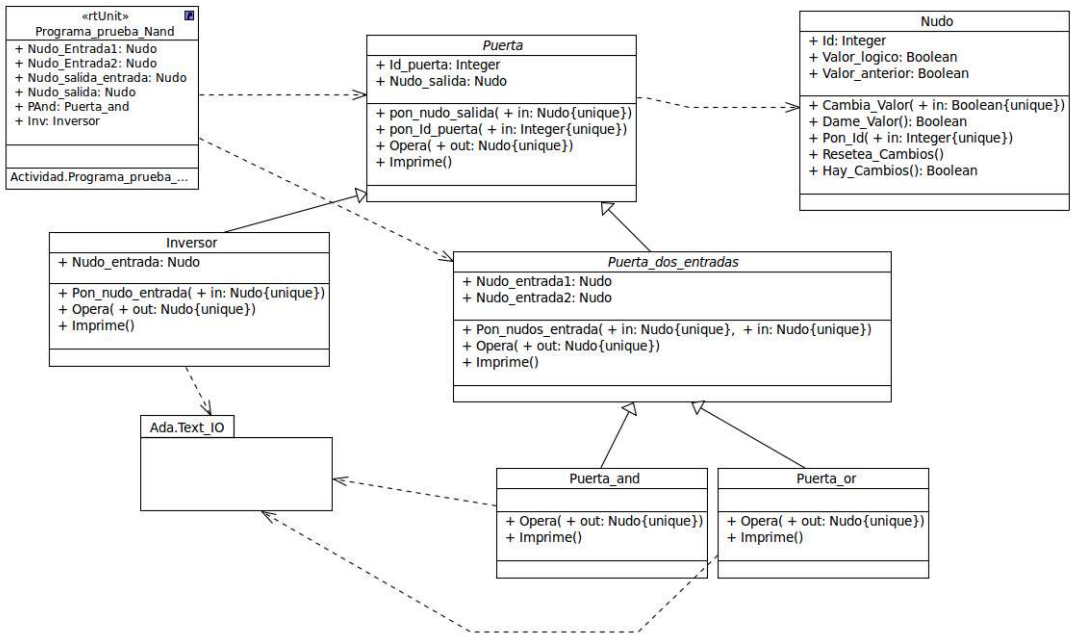

Figura 4.3: Diagrama de clases para el simulador lógico de circuitos combinacionales.

4.2.2. Análisis de la aplicación

Tal y como ya comentamos el diseño de la aplicación se apoyará en las puertas y en los nudos. En la figura 4.3 se ilustra el diagrama de clases que representa el diseño estructural del sistema, en él podemos ver la clase *Nudo* con una serie de operaciones que nos permiten añadir un identificador, cambiar o conocer el valor lógico actual, anotar a través de la operación *Resetea_Cambios* el valor lógico actual en el atributo *Valor_anterior*; ésta función junto a la denominada *Hay_Cambios* (encargada de comprobar si el valor actual y el anterior son el mismo) nos permitirían evaluar circuitos complejos desde alguna clase que se podría crear en una futura ampliación del simulador. La clase *Puerta* es abstracta, ya que alguna de sus operaciones serán implementadas por sus correspondientes clases hijas, debido a esto solamente dos de sus métodos, *pon_nudo_salida* y *pon_Id_puerta* que serán comunes a todas las clases que extiendan a *Puerta*, son definidos en ella a través de sus diagramas de actividad.

Las clases hijas de *Puerta* son especializaciones que van implementando nuevas funcionalidades según corresponda. Por ejemplo en la figura 4.4 se observa el diagrama de actividad asociado a la operación *Opera* de la clase *Inversor*.

Para comprobar el correcto funcionamiento se ha creado un procedimiento principal a través de la clase *Programa_prueba_Nand* estereotipada con *RtUnit* y con su valor etiquetado *isMain* a verdadero para indicar que se trata del programa principal. Los atributos declarados en esta clase serán utilizados por el generador para obtener la parte declarativa del procedimiento que se creará a través del diagrama de actividad.

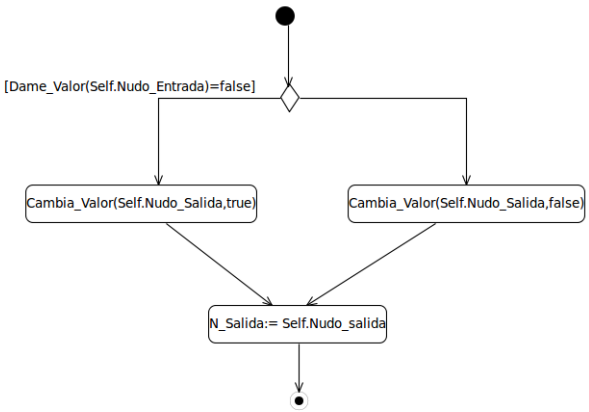

Figura 4.4: Diagrama de actividad para la operación *Opera* de la clase *Inversor*.

4.2.3. Código obtenido

Después de haber creado todos los diagramas de clase y actividad generamos automáticamente todo el código de la aplicación. A continuación mostramos parte del código obtenido para la clase *Puerta* en el que se observa que es declarada como una clase abstracta y como alguno de sus métodos también lo son:

```
01  with Nudos; use Nudos;
02  package Puertas is
03      type Public_Part is abstract tagged  record
04          Id_puerta : Integer;
05          Nudo_salida : Nudos.Nudo;
06      end record;
07      type Puerta is abstract new Public_Part  with private;
08 type P_Puerta is access Puerta;
09      -- Public Methods
10      procedure pon_nudo_salida (Self : in out Puerta'Class ;  Nudo : in   Nudos.Nudo);
11      procedure pon_Id_puerta (Self : in out Puerta'Class ;  Id : in   Integer);
12      procedure Imprime (Self : in out Puerta'Class ) is abstract;
13      procedure Opera (Self : in out Puerta'Class ;  N_salida :  out  Nudos.Nudo) is abstract;
14      ...
```

El siguiente código es el obtenido a través del diagrama de actividad mostrado en la figura 4.4 para el cuerpo de la clase *Inversor*:

```
01  ...
02  procedure Opera (Self : in out Inversor'Class;  N_Salida :  out  Nudos.Nudo) is
03  begin
04          if Dame_Valor(Self.Nudo_Entrada)=false then
05             Cambia_Valor(Self.Nudo_Salida,true);
06             N_Salida:= Self.Nudo_salida;
07          else
08             Cambia_Valor(Self.Nudo_Salida,false);
```

```
09             N_Salida:= Self.Nudo_salida;
10          end if;
11  end Opera;
12  ...
```

Capítulo 5

Conclusiones y trabajos futuros

El presente capítulo expone las conclusiones extraídas tras la realización del trabajo de fin de máster descrito en esta memoria, así como posibles trabajos futuros que complementen y mejoren el trabajo realizado.

5.1. Conclusiones

Este trabajo ha desarrollado un generador de código desde modelos UML a lenguaje Ada en forma de plugin para el entorno integrado de desarrollo (IDE) Eclipse. Normalmente los generadores de código obtienen únicamente la parte estructural de la aplicación que se quiere generar, en cambio nuestro generador es capaz de no solo extraer código para esta parte sino que con él se puede obtener el código correspondiente al comportamiento de operaciones o tareas a través de los diagramas de actividad. Además de esta innovadora característica, gracias a que los diagramas de actividad describen el flujo de trabajo de las operaciones, será posible en futuros trabajos instrumentar y obtener sus tiempos de ejecución de peor caso para poder analizar y determinar la planificabilidad del sistema.

Para la ejecución de este trabajo se han desarrollado una serie de reglas de equivalencia para realizar las transformaciones de UML a código Ada, debido a que Ada no es un lenguaje que permita fácilmente obtener una correspondencia directa con los modelos UML. Estas reglas pueden ser tomadas como un referente para cualquiera que requiera realizar una implementación de un sistema basada en el paradigma de orientación a objetos sobre este lenguaje de programación.

Los resultados obtenidos durante la realización de este trabajo de fin de máster han dado lugar a una publicación internacional [MR12][1] que incluye algunos de los resultados de mayor relevancia conseguidos, éstos han sido especialmente valorados por su potencial para el desarrollo dirigido por modelos de aplicaciones de tiempo real.

[1] El texto completo de la publicación está disponible en el CD adjunto.

5.2. Trabajos futuros

Todo trabajo es siempre susceptible de mejorar, existiendo una serie de características, que bien por limitaciones de tiempo, por su alta complejidad o por ser de un interés secundario, no fueron incorporadas al proyecto o quedaron por resolver. Y como este trabajo no es ninguna excepción, a continuación se muestra una lista de trabajos futuros que se podrán realizar:

1. Como ya se comentó en el capítulo 1 el presente trabajo se engloba en un enfoque en el que se necesitan desarrollar nuevas metodologías y herramientas para completar todas las transformaciones para conseguir obtener un desarrollo dirigido por modelos en el ámbito de los sistemas de tiempo real.

2. Añadir al generador la capacidad de instrumentar el código obtenido a través de los diagramas de actividad para obtener cotas estadísticas del tiempo de ejecución de las operaciones del sistema.

3. Ampliar el número de características de UML soportadas en el generador desarrollado, tales como dependencias circulares, relaciones de agregación o composición entre clases,y mayor variedad de nodos en los diagramas de actividad.

4. Utilizar el lenguaje OCL para definir restricciones en los modelos de entrada y poder detectar automáticamente situaciones no soportadas por el generador, como por ejemplo herencia múltiple entre clases.

5. Usar el lenguaje estándar de la OMG (ALF [OMG10]) para la especificación de las llamadas a operaciones dentro de las acciones de los modelos de actividad (*call actions*).

6. Proponer un método alternativo para la especificación del programa principal que no utilice el perfil MARTE, a fin de facilitar la explotación del generador por usuarios no interesados en los aspectos de tiempo real.

Apéndice A

Descripción de los contenidos del CD adjunto

A continuación se realiza una descripción del contenido del CD adjunto a esta memoria:

1. En la carpeta *plugin* se encuentra el plugin que contiene el generador desarrollado durante este trabajo.

2. En la carpeta *codigo_fuente_workspace* se encuentra disponible un *workspace* de Eclipse con el código fuente para el generador de código.

3. En la carpeta *casos_de_uso* están disponibles todos los artefactos desarrollados como demostradores y que se han presentado en el Capítulo 4. Se encuentran tanto el código generado como el modelo UML de entrada utilizado para los dos casos de estudio.

4. Dentro de la carpeta *contenido_adicional* se encuentra el manual de instalación del plugin creado, una copia del texto completo de la publicación realizada gracias al desarrollo de este trabajo y un enlace a la página web[1] creada para dar visibilidad y facilitar la evolución de este proyecto.

[1] `http://mast.unican.es/umlmast/uml2ada/`

Bibliografía

[Bar06] John Barnes. *Programming in ADA 2005*. Adison Wesley, 2006.

[BW94] Alan Burns and Andy J. Wellings. Hrt-hood: A structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, 1994.

[BW07] Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[EDZ99] R. Farfarakis E. Domiczi and J. Ziegler. Octopus supplement volume 1. nokia research center. `http://www.nrc.nokia.com/octopus/supplement/index.html`, 1999.

[FMS11] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML, Second Edition: The Systems Modeling Language (The MK/OMG Press)*. Morgan Kaufmann, 2011.

[Gom00] Hassan Gomaa. *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley, 2000.

[HGGM01] Michael González Harbour, J. J. Gutiérrez García, José C. Palencia Gutiérrez, and J. M. Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *ECRTS*, pages 125–134, 2001.

[KA11] Harold Klee and Randal Allen. *Simulation of Dynamic Systems with MATLAB and Simulink, Second Edition*. CRC Press, 2011.

[KRP+93] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M.G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, Kluwer Academic Publishers, 1993.

[LGT98] Agnes Lanusse, Sébastien Gérard, and François Terrier. Real-time modeling with uml: The accord approach. In *UML*, pages 319–335, 1998.

[MC] Julio L. Medina and Alvaro Garcia Cuesta. Marte2mast. `http://mast.unican.es/umlmast/marte2mast`.

[MC11] Julio L. Medina and Alvaro Garcia Cuesta. Model-based analysis and design of real-time distributed systems with ada and the uml profile for marte. In *Ada-Europe*, pages 89–102, 2011.

[MDN+03] Silvia Mazzini, Massimo DÁlessandro, Marco Di Natale, Andrea Domenici, Giuseppe Lipari, and Tullio Vardanega. Hrt-uml: Taking hrt-hood onto uml. In *Ada-Europe*, pages 405–416, 2003.

[MR12] Julio L. Medina and Alejandro Pérez Ruiz. Advances in the automation of model driven software engineering for hard real-time systems with ada and the uml profile for marte. In *Proceedings of International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2012.

[Obe] Obeo. Acceleo. `http://www.eclipse.org/acceleo`.

[OMG08] OMG. MOF Model to Text Transformation Language (MOFM2T), 1.0, formal/2008-01-16. Technical report, Object Management Group, 2008.

[OMG10] OMG. Action Language for Foundational UML (Alf) Concrete Syntax for a UML Action Language Beta 1, ptc/2010-10-05. Technical report, Object Management Group, 2010.

[OMG11a] OMG. Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems version 1.1, formal/2011-06-02. Technical report, Object Management Group, 2011.

[OMG11b] OMG. Object Management Group. Unified Modeling Language version 2.4.1, formal/2011-08-06. Technical report, Object Management Group, 2011.

[ONG+05] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Øyvind Aagedal, and Arne-Jørgen Berre. Toward standardised model to text transformations. In *ECMDA-FA*, pages 239–253, 2005.

[SBP+08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, and Randal Allen. *EMF: Eclipse Modeling Framework, Second Edition*. Addison-Wesley Professional, 2008.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Sel98] Bran Selic. Using uml for modeling complex real-time systems. In *LCTES*, pages 250–260, 1998.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling*. Wiley professional computing. Wiley, 1994.

[TDB+06] S. Tucker Taft, Robert A. Duff, Randall Brukardt, Erhard Plödereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries -*

International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1, volume 4348 of *Lecture Notes in Computer Science*. Springer, 2006.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional; 2 edition, 2003.