Desarrollo de *Software* para Sistemas de Tiempo Real Basado en UML. Un Enfoque Formal Basado en Metamodelado

Tesis Doctoral

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, España

Presentada por

José María Álvarez Palomo

Director: Dr. Manuel Díaz Rodríguez

D. **Manuel Díaz Rodríguez**, Titular de Universidad del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga,

Certifica

que D. **José María Álvarez Palomo**, Ingeniero en Informática por la Universidad de Málaga, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Desarrollo de Software *para Sistemas de Tiempo Real Basado en UML. Un Enfoque Formal Basado en Metamodelado*

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo, y autorizo la presentación de esta Tesis Doctoral en la Universidad de Málaga.

En Málaga, 9 de febrero de 2006

Firmado: Manuel Díaz Rodríguez
Titular de Universidad
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Muchos años después, frente al pelotón de fusilamiento,
el coronel Aureliano Buendía había de recordar aquella
tarde remota en que su padre lo llevó a conocer el hielo.
Cien años de Soledad. Gabriel García Márquez

. . . , 'when the Queen jumped up and bawled out,
"He's murdering the time! Off with his head!"'
Alice in Wonderland. Lewis Carroll

— Una taza de café y un título de doctor no se le niega
a nadie, como decimos en la Universidad – insiste el joven.
La sonrisa etrusca. Javier Sampedro

Es ingenuo pensar que una obra que se desarrolla durante un período de varios años, como es el caso de esta tesis, sea el resultado de un trabajo individual. Son muchas las cosas que yo, como autor, he incorporado a este trabajo y que las he aprendido no sólo de las personas con las que he convivido en estos años, sino durante toda mi vida.

Siempre resulta injusto mencionar a unas personas concretas cuando la memoria hace que olvides a otras, quizá tan importantes, así que espero que aquellos quienes se consideren merecedores de estar en estas líneas y no se encuentren en ellas, no lo tomen como un desaire, sino que se unan a ellas con la seguridad de que estaré encantado de que así lo hagan.

Quiero, por tanto, agradecer de manera concreta y dedicar esta obra a las siguientes personas. A mi familia *antigua*: mi padre, José, mi madre, Rafi y mi hermana Belén, por su amor y la educación que he recibido. A mi familia *nueva*: mi mujer, Carmen, mi hijo, Héctor y mi hija, Carmen, también por el amor que me dan todos los días y por enseñarme a ser una persona mejor.

A mi director de tesis, Manolo, por la guía que me ha dado todos este tiempo y por su paciencia, ampliamente superior a la obligada. A mis compañeros de despacho, Luis y José, con los que he pasado tantos días de trabajo agradables, todo un lujo.

A mi querido amigo Paco, por su revisión del manuscrito, tan exhaustiva y precisa.

A Andy, Paul y Guiem, por darme la oportunidad de trabajar con ellos en York y hacerme sentir como en casa.

Y a Miguel Ángel, por sus consejos.

Índice general

CAPÍTULO 1

Introducción

modelo.

(Del it. *modello*).

4. m. Esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento.

Diccionario de la lengua española. Vigésima segunda edición. Real Academia Española.

1. Modelado de sistemas

1.1. ¿Qué es el modelado?

La creciente complejidad de los sistemas informáticos ha llegado a tal nivel que aquellos de mayor envergadura son comparables en dificultad y tamaño con las grandes obras de otras ramas de la ingeniería o la arquitectura. Esta complejidad comporta dos cuestiones fundamentales. Por un lado, es difícil llegar a construir un sistema tan sofisticado, especialmente si no se tiene experiencia previa ni información básica sobre su composición. Por otro lado, también es difícil establecer *a priori* si el sistema funcionará correcta-

mente una vez construido, lo que es especialmente grave en aquellos sistemas cuyo coste es muy elevado, o son especialmente difíciles de modificar una vez construidos, o llevan a cabo tareas muy delicadas o peligrosas. En estas otras ramas de las ciencias es tradicional el uso de modelos que permitan un análisis previo de las características y el funcionamiento del sistema.

El uso de modelos es una herramienta básica para tratar esta complejidad, ya que permite hacer una réplica más simple del sistema, de la que eliminan detalles que no son fundamentales, obteniendo así un objeto de estudio más sencillo de entender, manejar y que permite hacer predicciones sobre aspectos importantes del sistema real.

Un buen modelo debe tener varias características [109]:

- Debe permitir la abstracción, para poder obviar detalles irrelevantes en un momento dado para el análisis de ciertas propiedades concretas del sistema. Además el grado de abstracción debe ser variable y así permitir que el análisis sea a mayor o menor nivel de detalle, según sea necesario.

- Debe usar notaciones que permitan a un lector humano entender el sistema. Si la notación usada es oscura o difícil de entender el modelo será de poca utilidad, incluso para sistemas con un mínimo de complejidad.

- Debe mostrar las mismas características que el sistema final, al menos en aquellos aspectos que quieran ser estudiados o analizados antes de la construcción del sistema final.

- Debe tener una base matemática o formal que permita la demostración de propiedades, con el fin de poder predecir el funcionamiento del sistema una vez construido.

- Debe ser significativamente más fácil y económico de construir que el sistema final.

1.2. Necesidad del modelado

Aprovechando los avances en la construcción de ordenadores con cada vez mayor capacidad de cálculo y de almacenamiento de información, los sistemas informáticos son cada vez más grandes, se aplican a más campos y se depositan en ellos responsabilidades mayores. Esta situación provoca una mayor exigencia sobre los sistemas informáticos.

No sólo han de ser sistemas que proporcionen un resultado correcto, sino que tienen otra serie de requisitos entre los que se pueden citar la obligación de ser sistemas seguros o responder satisfactoriamente frente a situaciones no esperadas o de error. Se puede ilustrar esta situación usando ejemplos de la aplicación de la informática a un campo concreto como la sanidad. Las pruebas de radiodiagnóstico más avanzadas hoy en día, como la tomografía axial computerizada o la resonancia magnética están controladas por ordenador, de cuya correcta programación depende la exactitud de las pruebas. Más delicado aún es el ejemplo de algunas terapias antitumorales en las que un equipo informático controla la exposición de un paciente a isótopos radioactivos. En los años ochenta, cuatro personas murieron por haber sido sometidos a una dosis demasiado alta por culpa de un error informático [76]. En los sistemas sanitarios también se han introducido complejos sistemas informáticos en el área de la administración y se usan grandes bases de datos para almacenar, entre otros datos, los historiales clínicos de los pacientes. Estas aplicaciones tienen grandes ventajas, ya que permiten a los médicos un acceso inmediato, desde distintos lugares y, posiblemente, simultáneo, a la información histórica de pacientes a los que pueden no haber tratado antes. Sin embargo, los

requisitos de seguridad y de confidencialidad son una condición básica para evitar que esos datos tan sensibles puedan usarse indebidamente. Por su parte, los sistemas de apoyo vital a los enfermos que están en las unidades de cuidados intensivos han de ser capaces de seguir funcionando correctamente incluso si, por ejemplo, se produce una pérdida momentánea de suministro eléctrico.

Es, por tanto, clara la necesidad de construir sistemas informáticos libres de errores y que respondan a los requisitos con que se pensaron. El desarrollo de sistemas informáticos es una rama de la ingeniería reciente para la que aún no hay técnicas de desarrollo que conciten la aprobación generalizada de los desarrolladores o que se puedan aplicar universalmente a las diferentes clases de sistemas informáticos. Sin embargo, en lo que sí se está cada vez más de acuerdo es en la necesidad, y en los beneficios que conlleva, el usar modelos para guiar la construcción de los sistemas reales, de forma que se puedan analizar las propiedades del sistema final antes y durante su desarrollo.

Diferentes autores han propuesto múltiples modelos distintos, influenciados por el tipo de sistemas desarrollaban en ese momento, por el campo de aplicación para el que se proponían, etc. Aún hoy en día, la diversidad es grande y se está lejos de la unanimidad, o de la universalidad de los modelos, por lo que se sigue investigando ampliamente en el tema.

1.3. Los sistemas de tiempo real y su modelado

Los *sistemas de tiempo real* son una clase concreta de sistemas informáticos que se pueden definir de manera informal como aquellos sistemas *en los que el tiempo de respuesta es crucial para su funcionamiento correcto.* También se dice que en un sistema de tiempo real, un dato obtenido fuera de plazo, aunque sea correcto, es un dato inválido, que incluso puede provocar

que el sistema falle en su conjunto.

Uno de los ejemplos más habituales de los sistemas de tiempo real son los sistemas de control, en los que un sistema computerizado se encarga de controlar el funcionamiento de otro sistema, informático o no. Por ejemplo, en los automóviles actuales se encuentran multitud de estos sistemas, como el sistema de antibloqueo de los frenos (ABS). Este sistema se encarga de vigilar el funcionamiento de las ruedas del vehículo durante la frenada. Si se bloquean, se liberan momentáneamente las ruedas para que sigan girando y no se deslicen. En cuanto las ruedas han conseguido un giro mínimo, se vuelve a actuar sobre el freno para volver a pararlas. En este ejemplo se muestran algunas de las características habituales de estos sistemas: se monitorizan unos datos que llegan del entorno, se procesan y, como resultado, se actúa sobre dicho entorno. Si la respuesta llega tarde y las ruedas han empezado a patinar, el desbloqueo de los frenos puede ser inútil o incluso contraproducente.

Otro ejemplo de sistemas de tiempo real, de una naturaleza distinta, es el de los servicios de videoconferencia, donde se establece de forma remota una conexión entre dos o más extremos y se exige que los datos lleguen con una determinada velocidad y calidad a los otros extremos, incluyendo la consideración de posibles errores en el canal de comunicación. Si se producen retrasos o pérdidas de imagen o sonido momentáneas, es posible que se consiga mantener una calidad suficiente en la conferencia, pero para eso es necesario que la mayoría de la información llegue correctamente y con un retraso mínimo.

Una de las clasificaciones de los sistemas de tiempo real distingue entre sistemas *duros* (*hard real-time systems*), en los que ningún dato se puede producir fuera de plazo, y *blandos* (*soft real-time systems*), en los que se

puede permitir que algunos de los resultados se produzcan con un retraso mayor del establecido.

Los sistemas de tiempo real tienen unas características propias que hace que su desarrollo sea aún más difícil que el de la mayoría del resto de los sistemas informáticos:

- Son sistemas inherentemente concurrentes en los que hay varios flujos de control ejecutándose simultáneamente e interaccionando, accediendo a recursos comunes y comunicándose y sincronizándose entre ellos. El desarrollo de sistemas concurrentes es más complejo por la posibilidad de problemas adicionales como el bloqueo, la inversión de prioridades, etc.

- Interactúan directamente con sistemas físicos. Es muy habitual encontrar sistemas que tienen una relación a muy bajo nivel con dispositivos físicos para lectura de datos para monitorizar los sistemas controlados y para escritura de datos para su control.

- Su funcionamiento depende habitualmente de estímulos procedentes del entorno (se suelen clasificar dentro de los llamados *sistemas reactivos*, que actúan dando respuesta a un estímulo exterior). La frecuencia de los estímulos exteriores es unas veces periódica, otras sigue una distribución de probabilidad y, en ocasiones, es desconocida.

- Se desarrollan en arquitecturas físicas muy variadas, no sólo en ordenadores tradicionales, sino en otros dispositivos electrónicos autónomos, desde vehículos a teléfonos móviles, pasando por un amplio abanico. A este tipo de sistemas de tiempo real se les llama *empotrados* (*embedded real-time systems*). Es habitual que esos sistemas empotrados impongan fuertes restricciones en varios aspectos. Por un lado, los recursos

físicos con los que se cuenta, como memoria y capacidad de cálculo, suelen estar muy ajustados, lo que incide en una mayor dificultad para encontrar una solución viable. Los recursos *software*, como bibliotecas de funciones o sistemas operativos, pueden también estar limitados, ya que es habitual la ausencia de versiones para estos entornos no estándares.

- Tienen el requisito no funcional adicional de los plazos temporales de las respuestas. Este requisito hace necesario el análisis de la planificabilidad del sistema, que establece si se pueden cumplir o no los plazos temporales y, si no se puede, cuáles son los que fallan.

Estas particularidades de los sistemas de tiempo real impiden, o limitan, que los modelos y metodologías de desarrollo de sistemas informáticos en general se puedan aplicar a los sistemas de tiempo real o, al menos, que sean suficientes. De aquí surge la necesidad de complementar modelos generales o desarrollar otros nuevos para tener en cuenta las características adicionales que presentan los sistemas de tiempo real.

1.4. Objetivos

El objetivo fundamental de esta tesis es el desarrollo de una metodología que integre modelos que permitan tener en cuenta las características propias de los sistemas de tiempo real para llevar a cabo el análisis de los requisitos no funcionales y para construir sistemas correctos teniendo en cuenta las limitaciones expuestas en la sección anterior.

Para ello no se ha partido de cero, sino que se han estudiado metodologías y métodos ya propuestos tanto para sistemas informáticos en general como las específicas para sistemas de tiempo real. En ambos casos se ha intentado

identificar, en primer lugar, cuáles son las virtudes y la utilidad de cada una de ellas, cuáles son sus puntos fuertes y las ideas o herramientas que son útiles en este contexto. Lo que se ha contrastado es que las metodologías para sistemas generales son una buena base sobre la que partir y que incorporan conceptos más nuevos, pero que adolecen, como es de esperar, de especificidad para resolver problemas como el análisis de planificabilidad, la interacción con los dispositivos físicos o la concurrencia del sistema.

Por otro lado, las metodologías de desarrollo de sistemas de tiempo real existentes ofrecen herramientas para el estudio de los requisitos propios de estos sistemas, pero son más difíciles de aplicar o no incorporan el uso de los modelos más en uso actualmente como los modelos gráficos.

Se ha intentado, por tanto, desarrollar una metodología que integre claramente y como elementos de primer nivel los aspectos particulares de los sistemas de tiempo real junto con aquellos aspectos comunes a los sistemas generales. Con ese objetivo se ha hecho uso de experiencias concretas de desarrollo de sistemas de tiempo real industriales, se han identificado los principales puntos débiles de las metodologías genéricas y se han propuesto las modificaciones necesarias para la integración adecuada.

También ha sido necesaria la modificación de herramientas de modelado ya existentes, concretamente las máquinas de estados del *Lenguaje Unificado de Modelado* (*Unified Modeling Language, UML*, [24]), que son apropiadas para el desarrollo de sistemas reactivos, como lo son los de tiempo real, pero que no incorporan en su definición original el tratamiento del tiempo que es necesario para el análisis de la planificabilidad o el rendimiento del sistema. Asimismo, se ha definido una nueva teoría para proporcionar una base formal a la semántica de algunas de las herramientas usadas. Concretamente, tanto las máquinas de estados de UML como las acciones subyacentes no cuentan

con una semántica formal en la norma de definición.

1.5. Aportaciones

Las aportaciones fundamentales de esta tesis se pueden dividir en tres, cada una expuesta en uno de los siguientes capítulos: definición de una semántica formal para las acciones de UML, definición de una semántica y adaptación para la definición de sistemas de tiempo real de las máquinas de estados de UML y definición de una metodología de desarrollo de sistemas de tiempo real en la que se hace especial hincapié en la síntesis homogénea de aspectos funcionales y no funcionales del desarrollo en cada una de las fases.

Semántica de acciones

UML, como se explica en secciones posteriores, es un conjunto de lenguajes de modelado, cada uno de los cuales se adecua a una fase o a un aspecto del desarrollo del sistema. Algunos de los lenguajes, o tipos de diagramas, se encargan de definir aspectos estáticos del sistema, como los diagramas de casos de uso, los de clases y objetos o los de despliegue. En estos diagramas se muestra la relación estructural entre las diferentes partes en las que se divide el sistema. El resto de los diagramas especifican aspectos dinámicos del sistema, cómo va evolucionando éste a medida que pasa el tiempo y se van ejecutando las acciones que componen la respuesta del sistema a las entradas de datos.

Estas acciones son el concepto fundamental de la semántica dinámica. Sin embargo, en la definición de la norma de UML no son desarrolladas en amplitud, sino que, posteriormente, se ha creado una extensión de la norma específica para su definición, la semántica de acciones de UML.

Sin embargo, tanto ni la norma de UML [88] ni la semántica de acciones [4] incluyen una semántica formal de los elementos ni de los diagramas. Los

autores de la norma defienden esa posición argumentando que no hay una petición amplia de la comunidad de usuarios de UML y sólo ofrecen una semántica informal basada en lenguaje natural. No obstante, nosotros pensamos que, como se detalló en secciones anteriores, una de las características básicas de un buen modelo es la posibilidad que ofrece de analizar y verificar el cumplimiento, o no, de ciertas propiedades. Para que esta verificación se pueda hacer matemáticamente y ofrezca garantías absolutas, el modelo del sistema debe tener una base matemática que permita hacer ese análisis.

Otros autores han definido semánticas alternativas para las acciones basándose en distintos modelos formales ya existentes, pero la comunidad de creadores de UML es muy reacia al uso de formalismos matemáticos de difícil comprensión ya que, razonan, los ingenieros que deben usarlos para verificar los sistemas no son expertos en matemáticas y eso les produce un rechazo que acaba por hacer impopular su uso. En esta tesis hemos intentado tener en cuenta dicha limitación y hemos optado por un modelo matemático que debe ser más comprensible y cercano a los desarrolladores, un metamodelado basado en los conceptos de *clase* y *objeto* como nociones básicas sobre el que se construye la semántica.

La semántica oficial de UML también está basada en este tipo de metamodelado en el que los conceptos más concretos están relacionados con otros más abstractos de un nivel superior. No obstante, esta semántica sólo explica de manera formal la parte estructural de las relaciones entre las diferentes clases, mediante diagramas de clases y haciendo uso del lenguaje funcional OCL para añadir restricciones no expresables en los anteriores diagramas.

La semántica que proponemos no se queda en la parte estructural, sino que incluye la parte dinámica. La semántica se basa en una distinción fundamental entre los conceptos de la sintaxis abstracta y sus correspondientes

conceptos en el dominio semántico. La semántica se establece mediante las relaciones apropiadas entre elementos de ambos conjuntos. Una segunda distinción se hace entre los conceptos semánticos y su representación gráfica.

Basándonos en esta semántica de metamodelo hemos definido un conjunto de acciones mínimo que pueda servir de base a la definición de conjuntos más amplios en función de las necesidades de cada tipo de sistema. Este conjunto de acciones incluye, por ejemplo, acciones simples, secuenciales y concurrentes. El resultado de este trabajo ha sido publicado en [16].

Semántica y adaptación de las máquinas de estados de UML

Las máquinas de estados son un modelo adecuado para reflejar la naturaleza reactiva de los sistemas de tiempo real, que esperan la ocurrencia de un evento externo, reaccionan frente a él ejecutando una serie de acciones, que posiblemente incluya la generación de otros eventos, y vuelven a quedarse en otro estado estable de espera.

Las máquinas de estados de UML están basadas en los *statecharts* de Harel [57] y tienen un conjunto muy rico de operaciones y recursos. Son máquinas que se pueden definir jerárquicamente, expandiendo un estado compuesto en otros estados más simples. Permite la ejecución concurrente de varias líneas de control, la ejecución de actividades durante el tiempo que la máquina permanece estable en un estado, o la ejecución de transiciones que permitan salir o entrar de múltiples estados simultáneamente.

En primer lugar se ha especificado una semántica formal para un subconjunto completamente funcional de las máquinas de estados siguiendo la misma estrategia que con la semántica de acciones, el metamodelado separando los conceptos de la sintaxis abstracta, el dominio semántico y la relación entre ambos a través de asociaciones semánticas.

En segundo lugar se ha analizado la capacidad y las limitaciones para expresar propiedades de tiempo real y se han propuesto las extensiones necesarias para poder expresar dichas propiedades y requisitos evitando las anomalías frecuentemente relacionadas con estos sistemas como, por ejemplo, la inversión de prioridades.

Metodología de desarrollo de sistemas de tiempo real

El uso de las metodologías orientadas a objetos junto con los métodos de descripción formal se han revelado como una forma prometedora de tratar la complejidad de los sistemas de tiempo real actuales, altamente complejos. Estas metodologías están actualmente bien soportadas por un conjunto de herramientas que permiten la especificación, simulación y validación de los aspectos funcionales de estos sistemas.

No obstante, la mayoría de estas metodologías no tienen en cuenta los aspectos no funcionales tales como la interacción con los dispositivos físicos y las restricciones de tiempo real, que son especialmente importantes en el contexto de este tipo de sistemas. En esta tesis presentamos una metodología que se basa en la combinación de otras notaciones y metodologías (UML, OCTOPUS, etc.), junto con la integración de técnicas de análisis de planificabilidad en el contexto de técnicas de descripción formal.

La metodología presta especial atención a la transición del modelo de objetos al modelo de tareas, teniendo en cuenta aspectos de tiempo real y de integración con los dispositivos físicos.

En esta tesis se ha definido un conjunto reducido de acciones, pero que consideramos suficientemente ilustrativo de la viabilidad del método de trabajo propuesto. Igualmente, para las máquinas de estados sólo se ha definido un subconjunto de todas las propiedades que incluyen las máquinas definidas

en UML. También en este caso pensamos que el trabajo desarrollado es lo bastante amplio como para dejar clara su capacidad.

En ambos casos, por tanto, queda aún por definir en nuestro modelo una semántica formal para parte de la norma UML, tanto en el caso de las acciones, cuyo espectro es más amplio en el perfil de la semántica de acciones, como en el caso de las máquinas de estados, algunas de cuyas características, como las actividades o la ejecución concurrente de acciones en las transiciones, no han sido incluidas en este modelo.

Otro aspecto fundamental que falta por desarrollar es una mayor integración de ambos conceptos con los demás diagramas especificados en UML, como puede ser la relación entre las máquinas de estados y sus clases asociadas o entre esas mismas máquinas de estados y diagramas dinámicos, como los diagramas de secuencia que deben representar escenarios de ejecuciones concretas en el entorno de sistemas compuestos por objetos cuyo funcionamiento viene definido por máquinas de estados.

En el caso de la metodología, aunque ha sido usada para el desarrollo de sistemas reales, como el del un teléfono inalámbrico, sería aconsejable un uso más amplio y variado para comprobar la adecuación de las actividades planteadas para el desarrollo de estos sistemas y para ir modificando aquellos aspectos que sean completamente satisfactorios. También es necesario conseguir herramientas que permitan la ejecución automática de los análisis incluidos en la metodología, bien implementando herramientas nuevas o adaptando las ya existentes para incluir dichos análisis.

Los resultados de este capítulo han sido publicados en [10] y [15].

Trabajo relacionado

En [8], [9], [11], [12], [13], [117] y [14] se estudia en profundidad la adecuación de SDL para el desarrollo de sistemas de tiempo real y se proponen extensiones que permiten superar sus carencias y realizar análisis de los requisitos temporales. SDL cuenta con ciertas ventajas frente a las máquinas de estados de UML: fue pensado para sistemas de telecomunicación y dispone de una semántica formal subyacente que permite analizar propiedades del sistema.

2. Modelado formal de sistemas

Son numerosos los distintos métodos de modelar sistemas informáticos que han surgido desde los años sesenta. Algunos de ellos son formales porque incorporan un formalismo matemático subyacente que permite derivar de manera fiable propiedades de los sistemas modelados.

Algunos de esos métodos usan como base la lógica de predicados de primer orden, como los trabajos de C.A.R. Hoare [58] y E.W. Dijkstra [37, 38]. Hay otros que se basan en la teoría de conjuntos, como Z [115], B [3] y VDM [66], para describir los cambios de estados de los datos del sistema. Otros se basan en álgebras de procesos, como CSP [59], CCS [81] o LOTOS [60]. Otro grupo es el formado por los métodos basados en lógicas modales, generalmente lógicas temporales, que permiten modelar la evolución del estado del programa en el tiempo en función de la ocurrencia de eventos o de la ejecución de las acciones. Por su parte, otros métodos, como las redes de Petri [93], SDL [61], los *statecharts* [57] o las máquinas de estados de UML [102] están basados en máquinas de estados. Estos métodos cuentan como ventaja con su expresión gráfica, que los hace más asequibles a los desarrolladores y

son especialmente adecuados para modelar sistemas reactivos.

Cuando surgieron la mayoría de estos métodos algunas técnicas de programación, como la orientación a objetos, no estaban tan extendidas por lo que las metodologías no ofrecían apoyo para desarrollos basados en estos paradigmas. Posteriormente han aparecido extensiones o actualizaciones de algunos de estos métodos que sí tienen en cuenta la orientación a objetos. Igualmente ha ocurrido con la especificación de requisitos no funcionales, como los requisitos de tiempo, el rendimiento o aspectos de seguridad.

Como se pone de manifiesto a continuación, la mayoría de los métodos descritos en las siguientes secciones han sido desarrolladas inicialmente para modelar sistemas en los que no se tenían en cuenta requisitos relacionados con el tiempo, por lo que no ofrecían mecanismos adecuados para expresar propiedades temporales ni realizar análisis de estos requisitos y que no eran válidas para modelar y analizar sistemas de tiempo real. Algunas de ellas, además, presentaban deficiencias para expresar otras características representativas de los sistemas de tiempo real, como la concurrencia, y analizar propiedades relacionadas con ella, como la sincronización, envío de mensajes o ausencia de interbloqueo. Para la mayoría de los métodos que se incluyen han surgido ampliaciones que incluyen herramientas para especificar propiedades temporales y analizar el cumplimiento de los requisitos temporales. Estas ampliaciones son muy variadas, en función del formalismo sobre el que se han fundamentado y la facilidad y potencia que se hayan conseguido para el objetivo de cubrir los aspectos temporales del sistema.

2.1. Métodos axiomáticos

La axiomatización de los lenguajes de programación ha sido el primer paso en la formalización del diseño de los sistemas informáticos. Los prime-

ros trabajos elaborados para el desarrollo metódico de programas basándose en formalismos matemáticos son los de C.A.R. Hoare [58] y E.W. Dijkstra [38]. En ambos trabajos se usa la lógica de predicados de primer orden. Las fórmulas de estas lógicas de programación son las *tripletas*, que constan de una precondición, un programa y una postcondición ({P} S {Q}). La precondición y la postcondición son predicados lógicos sobre el estado del programa. El estado del programa viene representado por el conjunto de valores de las variables. La tripleta es verdad si cuando empieza la ejecución del programa S, la precondición P es cierta y, si acaba el programa, la postcondición Q también es cierta.

En la lógica se definen las reglas de derivación adecuadas sobre los constructores básicos de la programación secuencial —secuencia, selección e iteración— que permiten comprobar la validez de las tripletas. Las propiedades que se comprueban se dividen en dos grupos, las de *seguridad*, que indican que un programa no llega nunca a un estado indeseable, y las de *viveza*, que aseguran que un programa acaba llegando a un estado válido. Por ejemplo, un bucle sin fin no cumple la propiedad de viveza de acabar. La técnica de Dijkstra va un paso más allá y propone métodos para la construcción de programas de manera que esté garantizado el cumplimiento de las propiedades consideradas. En particular, define reglas para construir bucles y sentencias de selección correctos.

La programación concurrente, en la que varios procesos se ejecutan simultáneamente compitiendo por recursos comunes, presenta nuevas instrucciones y situaciones, como la comunicación y la sincronización entre los procesos. Las lógicas de programas se han ampliado para los lenguajes concurrentes incluyendo nuevas reglas de inferencia para las nuevas instrucciones, como la de ejecución en paralelo o la de espera.

Entre las propiedades especiales que se han de analizar para los programas concurrentes están la ausencia de bloqueo (*deadlock*), de esperas indefinidas, (*livelock*) y de postergación indefinida de procesos (*starvation*). Para ello se han de hacer nuevas suposiciones sobre el entorno de ejecución, como la naturaleza del planificador, si es cíclico o si soporta interrupciones, y sobre su imparcialidad (*weak fairness* y *strong fairness*).

El gran problema de estas técnicas es que la mayoría de sus usuarios potenciales las encuentra extremadamente difíciles de aplicar en la práctica, lo que se traduce en una imposibilidad económica de aplicarlas a desarrollos reales en la industria. Esta dificultad se ha visto incrementada por la falta de herramientas que permitieran aplicar las técnicas de forma automática en sistemas del tamaño y complejidad habituales en las aplicaciones actuales.

2.2. Técnicas basadas en teoría de conjuntos

La especificaciones basadas en la teoría de conjuntos se caracterizan porque el estado del programa se expresa de manera explícita mientras que el funcionamiento del programa está implícito. El estado del programa se puede deducir del estado inicial y de las operaciones que modelan los cambios de estados. En cambio, los métodos basados en álgebras de procesos definen de manera explícita el funcionamiento del sistema y es el estado el que está implícito.

Una de dichas técnicas es la *notación Z* ([115, 95, 120]), la cual está basada en la teoría de conjuntos y la lógica matemática. Los objetos matemáticos y sus propiedades pueden reunirse en *esquemas*, patrones de declaraciones y restricciones. El lenguaje de esquemas puede usarse para describir el estado del sistema y las formas en que el estado puede cambiar. También puede usarse para describir propiedades del sistema y para razonar sobre posibles

refinamientos del diseño.

Una propiedad característica de Z es el uso de *tipos*. Cada objeto del lenguaje matemático tiene un tipo único. Aunque Z es un lenguaje de especificación formal, las especificaciones también incluyen lenguaje natural. Los modelos construidos en Z se pueden refinar, obteniendo otro modelo coherente con el anterior, pero de mayor nivel de detalle. Z está pensado para describir las características funcionales del sistema y las características no funcionales como usabilidad, rendimiento o fiabilidad, quedan fuera de su ámbito.

Para ilustrar algunas de las características básicas de Z, se va a especificar un sistema muy simple en el que se anotan y recuerdan fechas de cumpleaños.

$$[NOMBRE, FECHA]$$
$$LibroCumpleanyos =$$
$$[conocidos : \mathbb{P} NOMBRE; cumple :$$
$$NOMBRE \pfun FECHA | \ conocidos = dom \ cumple]$$

En este esquema se han definido los tipos que se van a usar $NOMBRE$ y $FECHA$, el conjunto de nombres cuyo cumpleaños se sabe (*conocidos*) y una función parcial que asocia ciertos nombres con fechas (*cumple*). La restricción final establece una forma de calcular *conocidos*, como el dominio actual de la función *cumple*.

El siguiente esquema define una operación que modifica el espacio de estados definido en el anterior.

$$\Delta IncluirCumple =$$
$$[nombre? : NOMBRE; fecha? : FECHA |$$
$$nombre? \notin conocidos; cumple' = cumple \cup nombre? \mapsto fecha?]$$

En este esquema, Δ indica que la operación va a modificar el estado del sistema. $nombre?$ y $fecha?$ son los nombres de los argumentos de la operación. Como precondición el nombre de entrada no puede tener una fecha asociada. La otra precondición indica que, tras la ejecución, el estado de la

función parcial *cumple* (denotado por *cumple'*) varía como se indica en ese predicado.

A partir de esta especificación se pueden demostrar propiedades del sistema. Por ejemplo, el nuevo nombre pasa a formar parte de los conocidos: $conocidos' = conocidos \cup nombre?$.

El uso de Z en la especificación de sistemas reales puso de manifiesto que los mecanismos de estructuración disponibles hasta el momento (principalmente los esquemas) no eran suficientes para estructurar de manera adecuada aplicaciones de gran tamaño.

Para mejorar esta deficiencia, diversos grupos han propuesto alternativas en las que se integra la filosofía de la orientación a objetos en Z [116]. En algunas se propone el uso de Z con un estilo orientado a objetos, mientras que en otras se añaden nuevos elementos a Z que permitan la descripción de los elementos habituales en la orientación a objetos (clases, objetos, métodos, ...).

2.3. Técnicas basadas en álgebras de procesos

Los métodos basados en álgebras de procesos modelan la interacción concurrente entre procesos secuenciales. Todas siguen unos mismos principios básicos:

- La interacción entre procesos se hace a través de la participación de éstos en *eventos*. Los eventos se suelen considerar atómicos (acciones indivisibles). En el modelo general, no está limitado el número de procesos que pueden interactuar en un mismo evento.

- Todo evento se considera una interacción entre procesos, con lo que se consigue un modelo homogéneo. Por ejemplo, escribir un valor en un

registro es un evento en el que interaccionan el proceso que está escribiendo el valor y el registro o el sistema.

- El funcionamiento de los constructores de procesos debe depender exclusivamente de los procesos que toman parte en él. Como ejemplos de estos constructores están: *conjunción*, *disyunción*, *encapsulación*, *secuenciación* y *simultaneidad*. La diferencia entre ellos está los puntos de sincronización que fuerza en los procesos.

CSP

La idea básica del lenguaje CSP (*Communicating Sequential Processes*) [59] es definir la concurrencia mediante la comunicación de procesos que tienen un funcionamiento interno secuencial y se comunican y sincronizan a través de unos canales restringidos, de forma que cada proceso sólo tiene acceso a sus datos locales. El *proceso* es el elemento básico de CSP y denota el patrón de funcionamiento de una entidad. Cada proceso puede involucrarse en un conjunto concreto de *eventos*.

Un proceso se define recursivamente, $P = (e \rightarrow Q)$, donde el proceso P consta de la ejecución del evento e y, posteriormente, la ejecución del proceso Q. Se pueden concatenar acciones que se van a ejecutar secuencialmente. Por ejemplo, s:$(e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow P)$. Hay operadores de selección ($|$) y de concurrencia ($\|$). Cuando los procesos se ejecutan en paralelo están restringidos a la sincronización en eventos comunes.

Dos procesos se comunican datos sincronizándose a través de los *canales*, que son mecanismos de comunicación unidireccionales y punto a punto. Cuando un proceso ejecuta un evento de comunicación de salida mediante la operación !, escribe un dato en el canal de comunicación y otro proceso debe sincronizarse con él ejecutando simultáneamente un evento de entrada sobre

el mismo canal mediante la operación *?*, en la que se lee el dato presente en el canal. Con esos elementos básicos, se puede especificar un sumador que reciba dos números por diferentes canales y los descarte tras transmitir la suma por un tercer canal:

$$\begin{aligned} ADD = \quad & (in1?x \rightarrow in2?y \rightarrow out!(x+y) \rightarrow ADD \\ & \mid in2?y \rightarrow in1?x \rightarrow out!(x+y) \rightarrow ADD) \end{aligned}$$

La semántica original de CSP es un caso concreto de álgebras de procesos y su modelo más simple es el modelo de *trazas*. Cada posible patrón de funcionamiento de un proceso de denota mediante una secuencia de eventos, llamada traza. En el modelo de trazas sólo se pueden especificar propiedades de seguridad, pero no de viveza. El modelo de trazas es extendido con otros modelos para cubrir esas propiedades, como el modelo de *rechazos*, las *divergencias* o el modelo de *fallos*.

Timed CSP es una extensión que añade nuevos operadores a la sintaxis original de CSP para estudiar propiedades no funcionales de los sistemas, concretamente los tiempos de respuestas para garantizar sincronizaciones en las que el instante en el que se llega a la sincronización es fundamental. Entre los nuevos operadores se encuentran la *espera*, que modela un lapso de tiempo en el que el proceso no hace nada, el *prefijo con retraso*, el *timeout* y la *interrupción temporizada*.

LOTOS

LOTOS (*Language Of Temporal Ordering Specification*) [60] fue desarrollado por la ISO para la especificación formal de sistemas distribuidos abiertos. La parte de funcionamiento de LOTOS está basada en el concepto de acción atómica e instantánea. Si se precisa modelar acciones que tarden tiempo, se modela una acción de inicio y otra de fin.

Un sistema en LOTOS está constituido por subsistemas que se ejecutan concurrentemente e interaccionan entre sí. Los subsistemas pueden, a su vez, estar compuestos por otros subsistemas más simples formando una jerarquía. La composición paralela de estos subsistemas se hace de una manera muy simple: dos procesos interactúan cuando ejecutan una acción sobre un mismo punto de interacción (*gates*). En esta sincronización pueden intercambiar información.

El funcionamiento de un sistema se define mediante expresiones de funcionamiento como `stop`, `exit`, la composición secuencial (`;`) y la elección (`[]`). Se puede hacer uso de la recursión para definir el comportamiento de un subsistema. Hay varios operadores para componer en paralelo los subsistemas definidos: composición concurrente sin sincronización (`|||`), sincronización total (`||`) y sincronización selectiva (`|[...]|`). Un concepto fundamental en la composición jerárquica de subsistemas es el *ocultación*. El operador `hide` permite ocultar acciones de manera que ninguna otra acción de otro subsistema se puede sincronizar con ella.

E-LOTOS (*Enhancement to LOTOS*) [43] es una extensión de LOTOS que incorpora novedades como el concepto de tiempo cuantitativo, definición de datos con un método parecido a la programación funcional, modularidad, nuevos operadores para aumentar la potencia expresiva del lenguaje y algunas instrucciones habituales de los lenguajes imperativos de programación.

RT-LOTOS (*Real Time LOTOS*) [34] es otra extensión de LOTOS proporciona tres operadores temporales principales: el operador de retraso, *delay*, que retrasa de manera determinista la ocurrencia de acciones observables e internas, el operador de latencia, *latency*, que expresa una latencia no determinista, y el operador de restricción que limita el tiempo durante el cual una acción observable es accesible desde el entorno.

2.4. Lógicas temporales

Las lógicas temporales permiten expresar el funcionamiento de sistemas software en términos de fórmulas lógicas, que incluyen restricciones temporales, eventos y relaciones entre ambos. La capacidad expresiva de estas lógicas ha ido creciendo y muchas de ellas son capaces de especificar adecuadamente sistemas reactivos aunque no todas sirven para especificar sistemas de tiempo real [21].

Las lógicas temporales son una clase particular de lógicas modales en las que el conjunto de mundos posibles representa los instantes de un dominio temporal. Las más usadas en la especificación de sistemas software son extensiones de lógicas de predicados de primer orden. Generalmente se añaden, al menos, cuatro nuevos operadores sobre los tradicionales: *siempre en el futuro*, *alguna vez en el futuro*, *siempre en el pasado* y *alguna vez en el pasado*. Algunos casos añaden dos nuevos operadores, *desde* y *hasta*.

Para usar las lógicas temporales en la especificación de sistemas de tiempo real, éstas han de ser capaces de expresar las restricciones temporales, las cuales se pueden dividir en dos grandes grupos: (i) eventos y ordenación de eventos y (ii) restricciones temporales cuantitativas. Para la especificación y análisis de sistemas de tiempo real, en los que se necesita una cuantificación de las propiedades temporales, hace falta una métrica del tiempo. Una forma típica de hacerlo es definiendo operadores acotados en el tiempo. Por ejemplo, se puede definir un operador que afirme que una fórmula es siempre cierta entre los instantes 4 y 7:

$$\Box_{[4,7]}A$$

PTL (*Propositional Temporal Logic*) [94] extiende la lógica proposicional introduciendo los operadores temporales *siempre en el futuro*, *alguna vez en el futuro*, *a continuación* y *hasta*. Las proposiciones describen relaciones

temporales entre estados. PTL es una lógica basada en eventos y no proporciona métrica para el tiempo. Es especialmente adecuada para sistemas reactivos o basados en eventos como las máquinas de estados, pero no tanto para sistemas de tiempo real.

BTTL (*Branching Time Temporal Logic*) [22] es una extensión de PTL en la que el modelo de tiempo pasa de ser lineal a ser ramificado en el futuro, con lo que se pueden especificar sistemas no deterministas. Los operadores originales de PTL son ampliados para manejar estas ramificaciones.

RTCTL (*Real Time Computational Temporal Logic*) ([45]) es una extensión de CTL (*Computational Tree Logic*) ([32]) para sistemas de tiempo real que proporciona una métrica para el tiempo. Sin embargo, el problema de la satisfacibilidad es doblemente exponencial y el problema de la comprobación de modelos es de coste polinomial.

RTTL (*Real-Time Temporal Logic*) [91] añade una métrica para el tiempo usando un reloj global del sistema cuyo valor se aumenta periódicamente y que puede usarse en las fórmulas para incluir características temporales. La ordenación de los eventos es parcial porque aquellos que han ocurrido entre dos instantes son indistinguibles. Sin embargo, las fórmulas son muy flexibles a la hora de referirse al tiempo, lo que también provoca que sean más difíciles de entender y de manipular que las de otras lógicas.

TPTL (*Timed Propositional Temporal Logic*) [7] también incluye una métrica para el tiempo haciendo corresponder a cada instante un número natural y usando una función monótona que asocia un valor temporal con cada estado del sistema.

IL (*Interval Logic*) [107] es un lógica proposicional basada en intervalos. Su modelo de tiempo es lineal, acotado en el pasado y no acotado en el futuro. No presenta métrica del tiempo. Los intervalos están acotados por eventos y

cambios de estado del sistema.

EIL (*Extended Interval Logic*) ([80]) y RTIL (*Real-Time Interval Logic*) ([98]) extienden IL para permitir la especificación de restricciones cuantitativas típicas de sistemas de tiempo real.

LTI (Logic of Time Intervals) [5] es una lógica temporal de intervalos de segundo orden. Los intervalos se pueden dividir en subintervalos hasta llegar a *momentos*. La lógica también permite cuantificación de los intervalos. Permite especificar sin problemas restricciones sobre la ordenación de los eventos, pero no restricciones cuantitativas, ya que carece de métrica de tiempo.

RTL (Real-Time Logic) [65] extiende la lógica de predicados de primer orden con operadores para expresar restricciones típicas de un sistema de tiempo real, pero no es una lógica temporal en el sentido tradicional. Presenta un reloj absoluto para medir el progreso del tiempo, que puede ser referenciado en las fórmulas. El modelo de tiempo es lineal, discreto, acotado en el pasado, no acotado en el futuro y totalmente ordenado. El principal problema es la dificultad de las fórmulas por el hecho de que se referencia un tiempo absoluto, además a un nivel de abstracción muy bajo.

TLA (*Temporal Logic of Actions*) [73] es una lógica que se basa en definir la semántica de acciones de un programa, a diferencia de otras lógicas temporales, en la que las propiedades de los programas son definidas mediante fórmulas de la lógica de proposiciones o predicados con operadores temporales. En [2] los autores defienden la idea de que no es necesario describir una nueva semántica para especificar sistemas de tiempo real, sino que es suficiente con los métodos usados para especificar sistemas concurrentes. Basta con añadir una nueva variable de programa que se refiera al tiempo.

3. Técnicas y herramientas formales de análisis: Tau

El modelado formal de sistemas ofrece grandes ventajas. En primer lugar, permite eliminar la ambigüedad en la especificación de un sistema, reduciendo de esa forma la posibilidad de malentendidos entre los diferentes componentes del grupo de desarrollo. También posibilita la aplicación de técnicas de análisis formales que garanticen matemáticamente la corrección del sistema, frente a los métodos tradicionales de prueba y error, que son útiles para detectar la presencia de errores, pero no garantizan su ausencia.

La existencia de una semántica formal para el modelo en el que se ha descrito el sistema permite hacer uso de herramienta automáticas o semi-automáticas para realizar la simulación del funcionamiento del sistema, su verificación, la validación de posibles escenarios y la generación automática de código.

En las siguientes secciones introducimos brevemente, a modo de ejemplo de esta posibilidad, la herramienta automática de diseño Tau de Telelogic que, a partir de la descripción del sistema usando MSC [85] y SDL [108] permite realizar los análisis mencionados anteriormente.

MSC es un lenguaje gráfico orientado a objetos que se usa para describir *escenarios*, es decir, ejecuciones concretas del sistema. SDL permite expresar mediante máquinas de estados el funcionamiento de las clases del sistema.

3.1. Simulación

El primer paso en la simulación de un sistema es la generación de código a partir de su descripción en SDL. Tau permite escoger el compilador que se va usar y configurar múltiples opciones en la creación del código. El código generado incluye en el sistema un proceso *monitor* que permite controlar e

inspeccionar la ejecución del sistema generado.

El simulador permite realizar una ejecución controlada del sistema de manera similar a como lo hacen los depuradores tradicionales de los lenguajes de programación. Se puede, por ejemplo, ejecutar el sistema hasta la siguiente transición. Como los activadores de las transiciones son señales desde el entorno, el simulador permite simular el envío de señales externas. También se puede establecer un punto de ruptura en un símbolo del sistema y ejecutar el sistema hasta que llegue a dicho símbolo. Asimismo, se puede ejecutar hasta la expiración de un temporizador.

Existe la posibilidad de inspeccionar el estado interno del sistema, viendo la lista de procesos, las colas de señales y el valor de las variables de estados de las instancias de procesos. Se puede modificar el estado del sistema creando nuevas instancias de procesos, modificando su estado o el valor de sus variables y se pueden activar y desactivar temporizadores.

Durante la simulación se pueden obtener diferentes trazas de la ejecución. Se puede obtener también un registro en modo texto, se puede ir viendo según avanza la simulación cuáles son los estados SDL que se van activando o se puede generar un MSC que represente la historia de la ejecución simulada, mostrando los estados de los diferentes objetos involucrados y los mensajes intercambiados entre ellos. Por último, es posible seleccionar la cantidad de información que muestra el registro de la simulación.

3.2. Generación de código

Tau ofrece la posibilidad de generar código ejecutable de la aplicación a partir de la descripción SDL. La generación de código se compone de dos elementos, el generador de código C a partir de la especificación SDL y las bibliotecas predefinidas que incluyen el *runtime* de SDL y los elementos de

monitorización del sistema, por si el código generado se usa para simulación o validación. Las bibliotecas de *runtime* incluyen los mecanismos de comunicación del sistema con el entorno.

En realidad, lo que se genera es el código correspondiente al funcionamiento de las máquinas de estados. Dentro de cada acción de SDL se pueden incluir sentencias en C que se integrarán en el código generado.

Tau ofrece tres versiones del generador de código C a partir de SDL, CBasic, sólo para simulación y validación, CAdvanced, para cualquier tipo de aplicación, y CMicro, que genera versiones más pequeñas y optimizadas, más útiles cuando se trata de sistemas empotrados con restricciones fuertes de memoria y capacidad de procesamiento.

Cuando se genera código se incluyen macros para tratar la interacción con el entorno, como las llamadas al sistema y el envío y recepción de señales. Las macros dependen del nivel de integración, que puede ser ligera, *light integration*, o ajustada *tight integration*. El código generado con la integración ligera se ejecuta con una sola hebra del sistema operativo subyacente para toda la aplicación, por lo que la planificación corre a cargo del *run-time* incluido en el código generado, lo que permite su ejecución sin sistema operativo subyacente. Por su lado, el código que se genera con la integración ajustada contiene una hebra del sistema operativo por cada proceso de la aplicación. En el modo de integración ligera, los procesos no son interrumpibles, pero en ajustado sí lo son, dependiendo de las posibilidades del sistema operativo sobre el que se esté trabajando.

Con CMicro existe un modo adicional de generación de código, *bare integration*, que no incluye macros de manejo de dispositivos físicos ni de señales. Este modo permite personalizar el código resultante mediante el uso de funciones de la biblioteca de CMicro para definir esos manejadores.

3.3. Validación y verificación

Tau también ofrece una herramienta para validar sistemas. Una vez especificado el sistema en SDL se puede compilar con la ayuda de un compilador de C y generar un modelo ejecutable que se puede validar para encontrar errores o inconsistencias o verificar respecto a determinados requisitos. El sistema generado incluye un monitor, al igual que el sistema generado para la simulación, pero con un abanico de órdenes distintas.

El sistema SDL examinado con el validador se representa mediante un *árbol de funcionamiento*, en el que cada nodo representa un estado completo del sistema SDL. Es posible examinar el funcionamiento del sistema recorriendo el árbol y examinando los estados. El tamaño del espacio de estados explorado se puede configurar y de él depende el número de estados generados tras una transición y el número de posibles sucesores de un estado.

El validador permite seguir la traza del sistema paso a paso de manera similar a como se hacía en la simulación, pero ofreciendo información adicional, como la lista de posibles transiciones a ejecutar en un estado. De este modo, el validador puede informar sobre errores dinámicos, como el envío de una señal que no puede ser recibida por ningún proceso.

El validador puede mostrar un MSC con la secuencia de estados que lleva a un estado en el que se ha producido un error. Para los sistemas cuyo espacio de estados es demasiado grande para llevar a cabo un recorrido exhaustivo se puede hacer un recorrido aleatorio, escogiendo arbitrariamente un camino de entre los posibles en cada transición.

Otra funcionalidad proporcionada en Tau es la verificación de un posible escenario descrito mediante un MSC. En este caso, el validador recibe el MSC y el sistema como entrada y simula aquellos caminos del árbol de estados que pueden reproducir el MSC. La validación acabará informando de si ese MSC

es posible en el sistema o de si hay caminos que lo violen.

3.4. Comprobación de modelos

La comprobación de modelos (*model checking*) [68] es una técnica de verificación formal en la que las propiedades de funcionamiento de un sistema se comprueban a través del recorrido exhaustivo, implícita o explícitamente, de todos los estados alcanzables por el sistema y todos los funcionamientos que puede alcanzar durante ellos.

La comprobación de modelos ofrece dos ventajas sustanciales respecto a otros métodos de verificación: es completamente automático y su aplicación no requiere un conocimiento profundo de disciplinas matemáticas, y cuando el diseño no cumple una propiedad el proceso siempre genera un contraejemplo.

Una variante muy interesante es la comprobación de modelos simbólica en la que se puede hacer una enumeración implícita exhaustiva de un número de estados astronómicamente grande.

La comprobación de modelos tiene dos limitaciones fundamentales: es aplicable sólo a sistemas de estados finitos y el número de estados crece exponencialmente cuando hay varios procesos ejecutándose en paralelo.

La aplicación de esta técnica se divide en tres etapas: el modelado del sistema, la especificación de propiedades y la verificación.

Aunque idealmente la verificación de propiedades con la técnica de comprobación de modelos es un proceso automático, comúnmente tiene que contar con la supervisión humana, concretamente en el análisis de los contraejemplos en aquellas situaciones en que se ha detectado que el sistema no cumple la propiedad analizada.

La evolución del sistema se modela mediante un grafo en el que cada nodo representa un estado, o conjunto de estados, del sistema que se expresa como

una fórmula lógica que se satisface en dicho estado o conjunto de estados. Las propiedades a verificar se modelan como fórmulas lógicas y se comprueba si se cumplen o no en los nodos del árbol.

4. Lenguajes gráficos de modelado

En esta sección se presentan lenguajes de especificación y diseño basado en gráficos. Sus elementos de descripción son visuales, incrementando de esa manera la facilidad de aprendizaje y la comprensión por parte de un operador humano.

4.1. SDL

SDL, el *Specification and Description Language* [61] es un lenguaje en el que se describen sistemas como un conjunto de procesos concurrentes que se comunican entre sí y con el entorno. El funcionamiento de los procesos está descrito en base a máquinas de estados comunicantes extendidas. SDL está especialmente recomendado para sistemas distribuidos y reactivos.

Uno de los principales objetivos al definir SDL fue su facilidad de uso, por lo que se optó por un modelo gráfico, más intuitivo que uno textual. Las versiones iniciales no tenían una semántica formal subyacente, lo que provocaba problemas porque permitía diferentes interpretaciones, dificultaba el desarrollo de herramientas automáticas que ayudaran a su uso y hacía más complicada la descripción precisa de sistemas complejos. Estos inconvenientes llevaron a la definición de una semántica formal que se incluyó por primera vez en la versión de 1988.

SDL cuenta asimismo con una versión textual SDL/PR (la gráfica se denomina SLD/GR) que es semánticamente equivalente y que surgió por la necesidad de procesar los ficheros de manera automática.

SDL describe la estructura del sistema de manera jerárquica mediante los constructores *sistema*, *bloque*, *proceso* y *procedimiento*. Una descripción en SDL se compone de un sistema que muestra los límites y la comunicación entre el sistema descrito y su entorno. El sistema se divide en bloques que se comunican a través de mensajes y estos bloques se componen a su vez de procesos que son entidades que se ejecutan concurrentemente. Los procedimientos definen secuencias de código que se ejecutan secuencialmente en el ámbito de un proceso.

A nivel de sistema la descripción está compuesta por los bloques de más alto nivel, los canales (y su lista de señales) de comunicación entre ellos y los canales de comunicación con el entorno. Esta descripción se puede ir detallando para cada bloque que puede dividirse jerárquicamente en otros bloques o en procesos. A este nivel la descripción sigue siendo estructural mostrando cuáles son los componentes y los canales de comunicación entre ellos.

Es ya a nivel de procesos donde se entra en la descripción del funcionamiento del sistema. Los procesos son entidades que se ejecutan concurrentemente y su funcionamiento viene definido por máquinas de estados.

Los procedimientos representan partes del sistema que se pueden definir para aclarar la descripción o agrupar acciones que se repiten a menudo. Los procedimientos pueden definir sus propios valores, pueden ser recursivos e incluir estados, como los procesos, lo que les permite recibir señales externas. Sin embargo, no pueden devolver el control a un estado del proceso que los llamó, sino que debe acabar, llegando al final o ejecutando una sentencia `return`.

Los procedimientos remotos son un tipo especial de procedimientos que proporcionan una vía alternativa de comunicación entre procesos, síncrona

en este caso, y les permite acceder y modificar variables definidas en otros procesos. Estos procesos se declaran como remotos en el nivel adecuado (sistema o bloque) para que sea visible a todos los procesos que lo usan. El proceso que lo define lo ha de exportar para que los demás puedan usarlo. Si otro proceso quiere hacer uso de él, lo puede importar y lo tiene disponible para llamarlo como cualquier otro procedimiento propio.

El principal mecanismo de comunicación entre los componentes de un sistema descrito en SDL es de las *señales*. Las señales se pasan de un componente a otro a través de canales, para los que se especifica la lista de señales que se puede enviar a través de ellos. Las señales corresponden a un evento transitorio asíncrono, frente a las llamadas a procedimientos remotos que se hacen de manera síncrona. Para poder usar las señales han de estar previamente declaradas y sólo pueden usarse en su ámbito. Las señales pueden llevar parámetros asociados.

Para que una señal se pueda transmitir entre dos procesos ha de existir entre ellos una ruta de comunicación. Estas rutas se denominan *canales* y son medios de comunicación punto a punto y bidireccionales. Los canales se describen en la descripción estructural del sistema y pueden conectar el entorno, bloques o procesos. Cada canal puede transmitir un conjunto concreto de señales que se especifica con el canal.

Como ya se comentó anteriormente, el funcionamiento del sistema se hace a nivel de proceso mediante máquinas de estados. Estas máquinas de estados se llaman *comunicantes* porque intercambian señales con las demás máquinas de estados del sistema y *extendidas* porque usan variables para reducir el número de estados.

Los componentes principales de una máquina de estados son los *estados* y las *transiciones*. Una máquina permanece en un estado hasta que recibe

una señal (la expiración de un temporizador es un caso especial de recepción de una señal). Cuando recibe una señal la máquina efectúa una transición a un estado nuevo. En la transición el proceso puede realizar diferentes acciones, como enviar señales, llamar a procedimientos, operaciones condicionales, operaciones con temporizadores, etc.

Cada máquina de estados sólo puede efectuar una transición a la vez y esta transición se ejecuta (desde el punto de vista de la propia máquina) de manera atómica, es decir, que hasta que no llega al nuevo estado no acepta nuevas señales.

Las señales que llegan a un proceso se guardan en una cola donde se atienden en el mismo orden en el que llegaron. Cuando un proceso llega a un nuevo estado inspecciona la cola de señales que aún no han sido procesadas. Si la señal que está en el frente de la cola es atendida en ese estado, el proceso la consume y evoluciona. Si, por el contrario, la señal no es atendida en ese estado, ésta se descarta. Para evitar la pérdida de señales se puede especificar en los estados qué señales son guardadas para ser procesadas en otro estado.

Hay tipos especiales de transiciones, como las transiciones espontáneas, que no consumen ninguna señal y se usan principalmente para pruebas, las señales continuas, que son transiciones que no se activan por la llegada de una señal, sino porque una condición lógica se haga verdadera, y transiciones con prioridad —si un proceso puede escoger entre una transición normal y una con prioridad porque ambas señales están en la cola de señales del proceso, el proceso siempre escoge la transición con prioridad aunque su señal haya llegado después—.

Se pueden definir tipos de sistemas, bloques y procesos de los que se pueden tener múltiples instancias en el sistema. Entre los tipos se pueden establecer relaciones de herencia en los que cada tipo modifica el tipo padre

a través de la especialización. La especialización en SDL está definida de una manera muy liberal y prácticamente no hay ninguna restricción sobre qué elementos se pueden especializar y cómo.

A modo de ejemplo se va a especificar en SDL un sistema muy simple. Se trata de una máquina expendedora que acepta fichas para vender un producto. El sistema se compone de dos procesos, uno que inicia la venta y otro que realiza la cuenta del importe ingresado. Si quedan existencias, se espera a que se introduzcan nuevas fichas hasta que se complete el precio. El usuario tiene en cualquier momento la posibilidad de anular la compra.

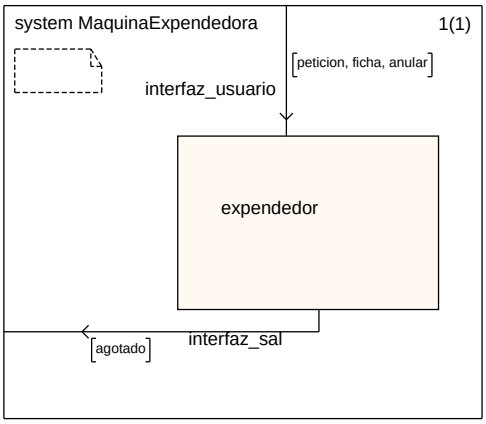

Figura 1.1: Descripción a nivel de sistema.

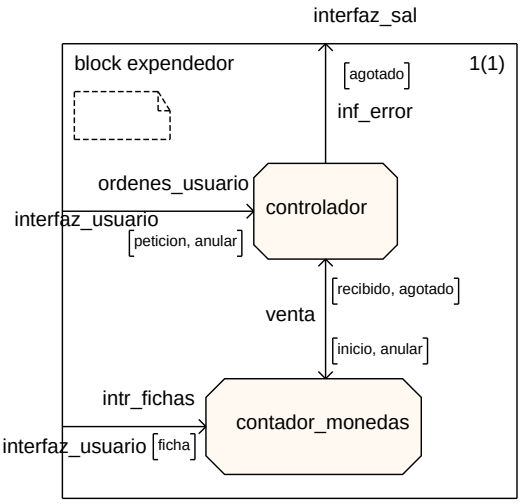

Figura 1.2: Descripción a nivel de bloques.

En la figura 1.1 se muestra la descripción del ejemplo a nivel de sistema en la que hay un único bloque que se comunica con el entorno a través de dos canales, para cada uno de los cuales se indican sus señales.

En la figura 1.2 se muestra la descripción a nivel de bloques en la que se ven dos procesos que se comunican entre sí y con el entorno. También se ve la correspondencia entre los nombres de las rutas de señales y los canales. Además de los canales de comunicación con el entorno, hay un canal interno,

no visible desde fuera del sistema, por el que se comunican los dos bloques.

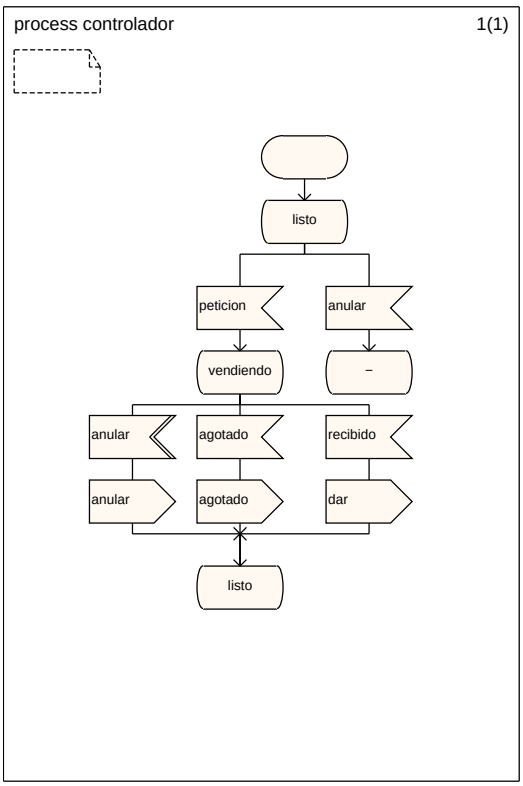

Figura 1.3: Descripción del proceso *controlador*.

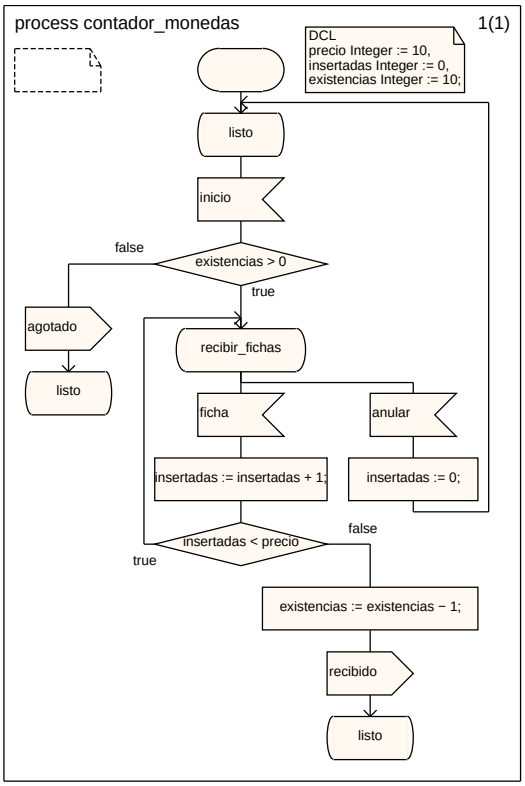

Figura 1.4: Descripción del proceso *contador*.

El proceso *controlador*, cuya máquina de estados se muestra en la figura 1.3, se encarga de iniciar y arrancar la venta. El proceso *contador*, figura 1.4, lleva a cabo la tarea de comprobar el pago. En ambos casos el funcionamiento viene desarrollado mediante una máquina de estados muy simple.

SDL para tiempo real

Aunque SDL tiene herramientas para especificar el paso del tiempo, como la operación *now* y los temporizadores, diversos trabajos de investigación han señalado que no son suficientes para especificar las restricciones habituales en los sistemas de tiempo.

Entre ellos se pueden citar [114], en el que se define SDL*, una extensión

de SDL que describe restricciones de diseño no funcionales. Con este SDL anotado y PMSC (una extensión de MSC con la misma filosofía) se puede partir el sistema en componentes físicos y lógicos para optimizar costes y cumplir los requisitos de tiempo. El proceso de diseño se puede automatizar para derivar un sistema de tiempo real optimizado. Las anotaciones son similares a las de PMSC y se integran como comentarios en el diseño SDL. Hay cinco tipos de directivas:

- de herramientas, que indican qué componentes se implementan y si hay que incluir ficheros con más anotaciones,
- de aplicación, que especifican sobre qué componentes físicos se desplegarán los componentes lógicos,
- de recursos, que indican la capacidad máxima de los recursos con límites en el sistema, como número de instancias, ancho de banda, etc.,
- de tiempo, que añaden información sobre el retraso en el envío de una señal o el *jitter*,
- de coste, que especifican el coste máximo que pueden tener los componentes del sistema.

En [86], se presenta una estrategia general para la especificación de sistemas de tiempo real usando SDL*. En primer lugar enumeran las deficiencias que, en su opinión, presenta el modelo de tiempo de SDL: temporizadores sin unidades, retrasos impredecibles en la recepción de las señales de expiración de los temporizadores e imposibles de acotar y semántica insuficiente de los relojes del sistema y la operación `now`. Clasifican las restricciones temporales en dos tipos: los requisitos temporales, que se han de validar durante las fases de diseño y prueba del sistema y condiciones temporales, que corresponden

a segmentos de código que se han de activar durante la ejecución del sistema en función de determinadas condiciones de validez asociadas. Proponen dos soluciones complementarias. Por un lado, la solución funcional, en la que desarrollan un patrón de diseño de sistemas de tiempo real que permite especificar reacción síncrona a componentes de acceso al medio, funcionamiento de tiempo real determinista y funcionalidad de nivel 2 OSI como multiplexores de paquetes o planificadores. La otra estrategia, la temporal, define relojes en base a una descripción matemática detallada, usando la notación descrita en esta propuesta. Cada sistema tiene al menos un reloj propio que puede ser asignado a cualquier elemento de la jerarquía del sistema y se define un nuevo tipo abstracto para los relojes con operaciones para acceder al valor del reloj y a operaciones sobre temporizadores sobre ese reloj.

En [26] Bozga *et al.* declaran que la semántica de tiempo de SDL y el que las señales de los temporizadores se encolen con las demás señales impiden especificar correctamente funcionamientos de tiempo real. SDL carece de una semántica de tiempo flexible y de más herramientas para expresar la parte no funcional del sistema o el entorno. Las características sobre el entorno y los aspectos no funcionales deben incluir la duración de las tareas internas, la periodicidad de los eventos externos y el funcionamiento esperable de los canales de comunicación. Para solucionarlo, se propone anotar el sistema con dos tipos de anotaciones: las *suposiciones*, que es conocimiento *a priori* del entorno, usado tanto para verificación como para generación de código, y las *afirmaciones*, que representan restricciones en forma de propiedades esperadas de los componentes del sistema. Para hacer suposiciones sobre el paso del tiempo en el sistema se definen *urgencias de transiciones* que especifican que una transición habilitada será ejecutada o inhabilitada antes de que pase una determinada cantidad de tiempo. En función de su urgencia las transiciones

se clasifican en *eager*, *lazy* y *delayable*. También se puede anotar la duración del tiempo que tarda una señal en llegar del emisor al receptor o el que consume una acción al ejecutarse. Asimismo se puede anotar la periodicidad esperada de los eventos externos. Respecto a los canales de comunicación, argumentan que en SDL siempre se consideran fiables, sin pérdida ni reordenación, consideración que siempre es realista, y por tanto proponen anotar los canales con propiedades que permitan indicar si es posible la pérdida o la reordenación de los mensajes enviados a través de los canales.

4.2. UML

UML, (*Unified Modeling Language*) [24], surge a mitad de los noventa como fusión fundamentalmente de tres métodos de desarrollo orientados a objetos, el método de Booch [23], el método OOSE [64] y el método OMT [101]. Cada uno de estos tres métodos era especialmente indicado en una de las fases del desarrollo y se intentó generar un único método que fuera útil durante todo el desarrollo y que eliminara los problemas de contar con diferentes nomenclaturas.

Aunque UML no es una metodología, sino un conjunto de lenguajes, su objetivo es visualizar, especificar, construir y documentar los elementos de sistemas con gran cantidad de software. Los lenguajes definidos en UML son fundamentalmente gráficos, para facilitar su estudio y comprensión. En UML se definen nueve tipos de diagramas, algunos de los cuales modelan diferentes vistas estáticas del sistema y otros modelas vistas dinámicas:

- Diagramas de clases. Éste es un diagrama estático en el que se muestran cuáles son las clases involucradas en el sistema y las relaciones entre ellas.

- Diagramas de objetos. Este otro diagrama estático está íntimamente

relacionado con el anterior. Muestra una vista de las instancias reales de las clases que están en ejecución en el sistema en un momento determinado.

- Diagramas de casos de uso. En estos diagramas se muestran conjuntos de casos de usos y actores y sus relaciones.
- Diagramas de secuencia. Estos diagramas muestran parte de la vista dinámica, concretamente una interacción entre un subconjunto de objetos, básicamente a través del envío de mensajes entre ellos. Estos diagramas priman la ordenación temporal de los mensajes.
- Diagramas de colaboración. Estos diagramas son isomorfos a los diagramas de secuencia, muestran la misma interacción, pero resaltando la organización estructural de los objetos.
- Diagramas de estados. Son máquinas de estados que especifican el funcionamiento de los objetos de una clase. Se centran en el comportamiento de sistemas reactivos dirigidos por eventos.
- Diagramas de actividad. Son un tipo especial de diagrama de estados que resaltan el flujo de actividad entre los objetos.
- Diagramas de componentes. Son diagramas estáticos que muestran la organización y las dependencias entre los componentes. Un componente suele corresponder a varias clases o interfaces.
- Diagramas de despliegue. Es una vista estática estructural en la que se relacionan los nodos de procesamiento con los componentes que residen en ellos.

En UML se definen cuatro tipos fundamentales de elementos:

- Estructurales. La mayoría son elemento estáticos e incluyen clases, interfaces, colaboraciones, casos de uso, clases activas, componentes y nodos.

- De comportamiento. Son las partes dinámicas del modelo e incluyen relaciones y máquinas de estados. Están asociados a uno o varios elementos estructurales.

- De agrupación. Son los elementos organizativos del modelo. Los únicos elementos de agrupación son los paquetes, que se pueden descomponer en elementos más simples.

- De anotación. Sirven para incluir explicaciones sobre los demás elementos del modelo. Las notas son los elementos explicativos. Son simplemente símbolos que introducen restricciones y comentarios.

Hay cuatro tipos de relaciones en UML:

- Dependencia. Es una relación semántica entre dos elementos en la que un cambio en un elemento puede afectar a la semántica en el otro.

- Asociación. Es una relación estructural que define un conjunto de conexiones entre objetos. La agregación es un tipo especial de asociación.

- Generalización. En una relación de generalización/especialización los elementos que especializan pueden sustituir al especializado. El caso más usual en la relación de herencia entre clases.

- Realización. Es una relación semántica entre clasificadores en la que un clasificador especifica un contrato que otro clasificador realizará. Esta relación se da entre interfaces y clases y entre casos de uso y diagramas de colaboración.

Las especificaciones en UML son la explicación textual de la sintaxis y la semántica de los elementos gráficos de UML. La especificación cubre los detalles que no se pueden representar en la notación gráfica, usualmente más simple y escueta. Las especificaciones se hacen en lenguaje natural habitualmente y, si se quiere una especificación más formal, se usa OCL (*Object Constraint Language*) [87]. Además de la especificación, la notación gráfica se puede enriquecer con *adornos*, símbolos gráficos concretos para indicar ciertos detalles del diagrama, como puede ser la visibilidad de los atributos.

UML ofrece la posibilidad de definir nuevos elementos mediante diversos mecanismos de extensión: los estereotipos, los valores etiquetados y las restricciones. Con los estereotipos se construyen nuevos bloques de modelado, derivados de los ya existentes pero específicos del dominio del problema. Los valores etiquetados extienden las propiedades de un bloque de construcción añadiendo nueva información en la especificación del elemento. La restricción extiende la semántica de un elemento de UML incluyendo nuevas reglas o modificando las existentes.

En el resto de la sección vamos a hacer un recorrido breve por los diagramas de UML que son más útiles para la descripción de los sistemas de tiempo real. En el aspecto estático incluiremos los diagramas de casos de uso y los de clases. En el aspecto dinámico incluiremos los diagramas de secuencia y de estados.

Diagramas de clases

Un diagrama de clases presenta un conjunto de clases, interfaces y colaboraciones, y las relaciones entre ellas. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos y se utilizan para describir la vista de diseño estática de un sistema. Los que incluyen clases activas se

utilizan para cubrir la vista de procesos estática de un sistema.

Los elementos de los diagramas de clases suelen ser:

- clases
- interfaces
- colaboraciones
- relaciones

Como en los demás diagramas, podemos añadir notas y restricciones. Si el sistema modelado por el diagrama es muy grande, se puede dividir jerárquicamente incluyendo paquetes o subsistemas en el diagrama.

Las clases se pueden utilizar para capturar el vocabulario del sistema que se está desarrollando, tanto para elementos software, hardware o puramente conceptuales. Las clases representan conjuntos de valores con propiedades comunes. Estas propiedades incluyen atributos, operaciones, relaciones y semántica.

Un atributo es una abstracción de un rango de valores (o estado) que puede incluir una instancia de la clase. Una operación es la implementación de un servicio que puede ser requerido a cualquier instancia de la clase para que muestre un comportamiento.

La representación gráfica de una clase es un rectángulo con compartimentos separados para los atributos y las operaciones.

Hay tres tipos fundamentales de relaciones: las dependencias, que representan relaciones de uso entre las clases, generalizaciones, que conectan clases generales con sus especializaciones, y asociaciones, que representan relaciones estructurales. Las asociaciones pueden conectar dos o más clases y suelen tener un nombre que identifica su naturaleza. Las clases involucradas en la

asociación tienen una función también identificada por su nombre. Cada clase puede estar presente en la asociación con un número diferente de instancias, posibilidad que se indica mediante la multiplicidad. Las asociaciones suelen representar relaciones de uso, aunque hay casos especiales, como las agregaciones y las composiciones, que representan una relación de *todo/parte*. Estas asociaciones indican aquellas situaciones en las que las instancias de una clase incluyen instancias de otras clases.

Un diagrama de objetos es un tipo particular de diagrama de clases en el que se muestran las instancias que conforman realmente el sistema. Representa un conjunto de objetos y sus relaciones y se utilizan para describir estructuras de datos correspondientes a instancias de los elementos encontrados en los diagramas de clases. Los diagramas de objetos cubren la vista de diseño estática o la vista de procesos estática de un sistema al igual que los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos.

Diagramas de casos de uso

Los casos de uso se emplean para capturar el funcionamiento deseado del sistema en desarrollo, sin tener que especificar cómo se implementa ese funcionamiento. Un caso de uso especifica el funcionamiento del sistema o de parte del mismo, y es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable de valor para un actor.

Un actor representa un conjunto coherente de funciones que los usuarios de los casos de uso juegan al interactuar con el sistema. Un actor es, típicamente, una persona, un dispositivo físico u otro sistema que interaccione con el modelado. Los actores se relacionan con los casos de uso a través de asociaciones que indican su función en el caso de uso. Un caso de uso describe

qué hace el sistema sin especificar cómo lo hace. Es importante tener clara la separación entre las vistas externa e interna.

Diagramas de interacción

Un diagrama de interacción muestra las acciones que se desarrollan entre un conjunto de objetos relacionados, fundamentalmente representados por el envío de mensajes entre los objetos. Hay dos tipos de diagramas de interacción, que son isomorfos: los de *secuencia* y los de *colaboración*. En los diagramas de secuencia se hace énfasis en mostrar la ordenación temporal de la sucesión de mensajes y en los diagramas de colaboración se incide más en la organización estructural de los objetos que realizan las acciones mostradas.

Diagramas de actividad

Un diagrama de actividad es un diagrama de flujo que muestra el flujo de control entre actividades. Las actividades son ejecuciones no atómicas en curso dentro de una máquina de estados. Las actividades producen acciones, compuestas de cálculos atómicos.

Los diagramas de actividad contienen estados de acción y de actividad, transiciones y objetos. Esots diagramas se pueden considerar como casos especiales de las máquinas de estados en los que los estados son estados de actividad y las transiciones se disparan por la terminación de la actividad del estado origen.

Las transiciones pueden incluir bifurcaciones dependientes de condiciones y divisiones y uniones para representar flujos de control concurrentes.

Cuando se modelan flujos de trabajos de varios objetos a la vez, se pueden modelar mediante calles (*swimlanes*), donde se separan gráficamente los flujos de cada objeto.

Máquinas de estados

Las máquinas de estados se usan para modelar los aspectos dinámicos de un sistema. Una parte fundamental del funcionamiento de un sistema orientado a objetos se puede definir de manera reactiva, las instancias de las clases del sistema responden con determinadas acciones a estímulos externos a la clase. Estos estímulos pueden ser una llamada a uno de los métodos de la clase o el envío de una señal. Si el objeto no realiza ninguna otra acción entre el fin de la respuesta a un estímulo externo y el inicio de la siguiente respuesta, se puede considerar que permanece en un determinado estado en ese intervalo de tiempo. Las máquinas de estados se adecúan perfectamente para definir este tipo de funcionamiento.

A diferencia de otros diagramas de UML, un diagrama de máquinas de estados no representa el funcionamiento de varias clases dentro del sistema, sino que se limita a representar el funcionamiento interno de las instancias de una clase concreta. Una máquina de estados está compuesta por varios elementos fundamentales: estados, transiciones, eventos y acciones.

Los estados representan las situaciones estables en la evolución de un objeto en la que se cumple alguna condición y se espera algún evento. Hay estados especiales, como el estado inicial, en el que se encuentra el objeto cuando se crea y los estados finales, que representan el final de la vida útil de un objeto cuando alguna transición hace que un estado final sea el estado actual del objeto. Los estados pueden dividirse jerárquicamente para representar un funcionamiento interno, bien secuencial, bien concurrente.

El paso de un estado a otro se hace mediante la ejecución de una transición. Una transición indica cuáles son las posibles evoluciones del estado de una instancia. Cada estado tiene su propio conjunto de transiciones, y cada una de ellas lleva a un nuevo estado. Las transiciones tienen asocia-

das condiciones, acciones y eventos. La elección de la transición que realiza una instancia como respuesta a un estímulo externo depende de cuáles están activadas (en base a su condición) y de los eventos asociados a las transiciones. Una transición sólo es elegible si su evento asociado coincide con el que ha recibido la máquina de estados. La ejecución de la transición implica la ejecución de las acciones asociadas.

Un evento es la especificación de un suceso significativo para el funcionamiento de la instancia que lo recibe. En las máquinas de estados de UML, los eventos pueden incluir señales asíncronas, llamadas a métodos, cambios de valores o el paso del tiempo. Cuando una instancia recibe un evento reacciona a él, típicamente ejecutando una transición, que suele involucrar un cambio de estado.

Las acciones especificadas en una máquina de estados representan los cálculos ejecutados por la instancia como respuesta al evento. Hay acciones que se ejecutan al entrar a un estado, otras que se ejecutan al salir de un estado, otras que se ejecutan al ejecutar una transición y otras que se ejecutan mientras la máquina permanece en un estado.

UML para tiempo real

UML tiene dos conceptos relativos al tiempo, las clases `TimeExpression` y `TimeEvent`. Los valores de la clase `TimeExpression` sirven para expresar valores relativos al tiempo en aquellas circunstancias en las que se necesiten, como, por ejemplo, las condiciones de habilitación de las transiciones de las máquinas de estados para etiquetar propiedades de los mensajes en diagramas de secuencia. Los elementos de la clase `TimeEvent` se usan para expresar la ocurrencia de un evento relacionado con el tiempo como la expiración de un temporizador. UML puede usar restricciones para establecer varias

propiedades de tiempo, como, por ejemplo, el tiempo de ejecución o el tiempo de bloqueo, o puede usar valores etiquetados para establecer la prioridad de una tarea o su plazo temporal de ejecución.

Estos mecanismos no son suficientes para especificar todas las restricciones y funcionalidades asociadas a los sistemas de tiempo real. Por ejemplo, no se puede expresar la frecuencia de ocurrencia de un evento como una función de probabilidad. Tampoco hay mecanismos para referirse a relojes de referencia o a los servicios temporales ofrecidos por capas inferiores de *software*, como el sistema operativo. La falta de una semántica formal de esos mecanismos impide que una herramienta que haga análisis de planificabilidad pueda contar con la representación de estos conceptos para su importación.

El *UML Profile for Schedulability, Performance, and Time Specification* (*Perfil de Planificabilidad, Rendimiento y Especificación del Tiempo)* [56] es una respuesta a una iniciativa del Grupo de Manejo de Objetos (*Object Management Group*, OMG), un grupo de promotores de la orientación a objetos en el desarrollo de sistemas. La finalidad de esta iniciativa es extender UML para que se pueda usar para la especificación y diseño de sistemas de tiempo real en toda su extensión y diversidad. El trabajo se ha realizado utilizando los mecanismos de extensión presentes en UML, los estereotipos y los valores etiquetados. En este perfil se han introducido en el metamodelo de UML algunas clases nuevas, como `TimeValue`, `Duration`, `Clock` y `Timer`. De esta forma se definen nociones tan importantes como el periodo de una tarea, el *jitter*, o el tiempo de transmisión de una señal. La estructura del perfil se divide en cuatro paquetes principales (figura 1.5), en los que se definen, respectivamente, los recursos generales, en los que se basan los demás paquetes, los conceptos relacionados con la definición del tiempo, los conceptos relativos al análisis de planificabilidad y los relativos al análisis del rendimiento.

Este perfil se revisa en profundidad en el capítulo 3, en el que se añaden a las máquinas de estados de UML elementos para especificar sistemas de tiempo real.

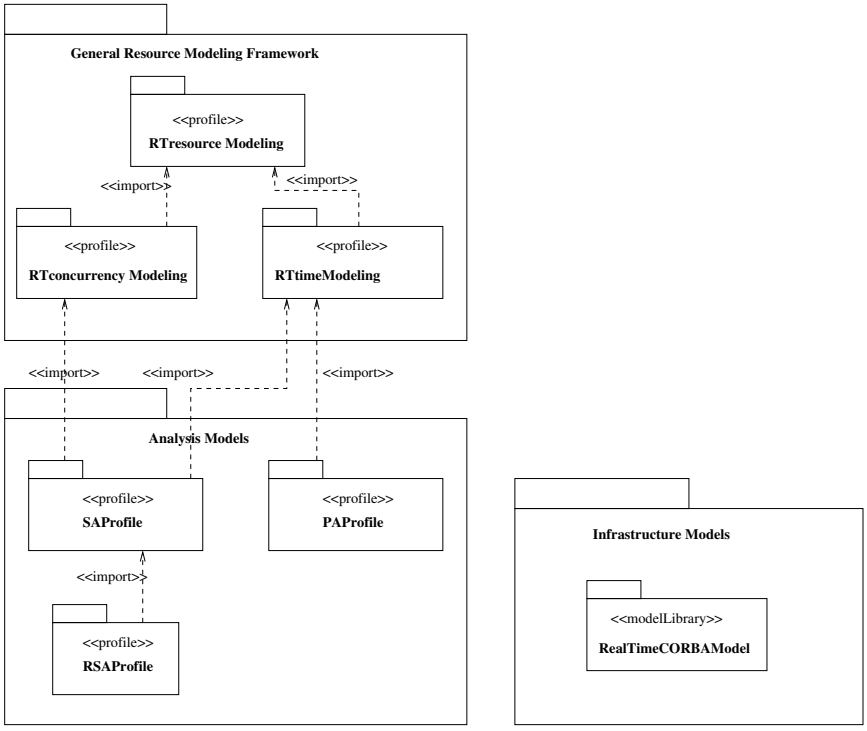

Figura 1.5: Estructura del perfil de planificabilidad, rendimiento y especificación de tiempo, tomada de [56].

Shankar y Asa [112] y Lavazza *et al.* en [75] sugieren estrategias similares para añadir capacidad de especificación de requisitos de tiempo real. En ambos casos optan por traducir el modelo UML a un modelo formal, concretamente lógicas temporales, sobre el que se pueden demostrar propiedades.

En [112] se centran en las máquinas de estados, a las que añaden constructores de tiempo real independientes de la semántica concreta de la máquina de estados. La base matemática es una lógica temporal proposicional bidimensional, en la que se diferencia el *microtiempo*, o puntos, para referirse a las transiciones instantáneas de estado de un objeto, del *macrotiempo*, para

referirse a las comunicaciones entre objetos. La lógica ofrece los operadores temporales usuales: *siempre en el futuro*, *alguna vez en el futuro* y *hasta*, con versiones acotadas. Las máquinas de estados que usan son planas y deterministas respecto a las transiciones, y se traducen a la lógica temporal siguiendo un determinado conjunto de reglas. Una vez que se tiene el modelo en la lógica se pueden verificar diferentes tipos de propiedades. Por un lado, las habituales de seguridad y viveza; por otro, si un evento o acción ocurre antes de t instantes de tiempo tras otro; y, finalmente, la validación del modelo en sí, incluyendo la comprobación de la consistencia entre diagramas de secuencia y máquinas de estados.

En [75] traducen las máquinas de estados de UML a TRIO (*Tempo Real ImplicitO*), una lógica de primer orden aumentada con operadores temporales. En el caso concreto del ejemplo con el que ilustran su propuesta, usan una herramienta automática de control de históricos. Para especificar el funcionamiento temporal, usan la sentencia `After` de UML y expresiones lógicas. La traducción a TRIO se hace identificando fragmentos del diagrama de estados y definiendo los correspondientes axiomas TRIO.

Aprville *et al.* proponen en [17] un nuevo perfil de UML, llamado TURTLE, específicamente pensado para la validación de restricciones de tiempo real. Para ello, se basan en tres pilares, una semántica formal para las asociaciones de clases, operadores temporales con retrasos no deterministas y la traducción de las clases y la descripción de su funcionamiento a RT-LOTOS. En TURTLE se usa un nuevo tipo de clase, `TClass`, que resulta de estereotipar el concepto usual de clase en UML. Las instancias de `TClass` se pueden comunicar a través de canales de comunicación llamados `TGates`, de los que hay de entrada y de salida. El funcionamiento dinámico de las clases se hace mediante diagramas de secuencias. Para cada constructor de los diagramas

de secuencias se define una traducción a RT-LOTOS. Con esta traducción se puede generar automáticamente una descripción del sistema en RT-LOTOS. La especificación RT-LOTOS queda oculta al usuario y puede ser simulada y validada mediante técnicas de análisis de alcanzabilidad.

5. Metamodelado

Cuando se define un lenguaje de modelado de sistemas, éste suele poder aplicarse a un conjunto más o menos amplio de casos, dependiendo de la generalidad del lenguaje. Si el lenguaje es muy específico puede quedar reducido a modelar sistemas de un tipo muy concreto como, por ejemplo, sistemas eléctricos, o cualquier otro. Si el lenguaje de modelado es suficientemente genérico y puede modelar una gama amplia de sistemas, puede llegar a modelar el propio lenguaje de modelado como un caso concreto de sistema modelado. En este caso, los elementos del lenguaje no se usan para describir los valores de los datos del sistema sino características de los datos del sistema.

Esta circunstancia del *modelado del modelo* en vez del *modelado de valores* se conoce como *metamodelado*. Esta *meta* relación no es exclusiva del modelado de sistemas. Bien al contrario, se puede encontrar en múltiples situaciones en el ámbito de la informática. Por ejemplo, en las bases de datos, los metadatos son datos sobre datos, los metaintérpretes son programas en cuya ejecución, se usan como datos ejecuciones de programas (por ejemplo los depuradores) o, en un sistema de información, se puede tener el modelo E-R del modelo E-R del sistema.

El metamodelado suele definirse en niveles. El lenguaje de modelado puede definirse mediante un metamodelo, donde los constructores presentes en el metamodelo se denominan metaelementos y su relación con los elemen-

tos del lenguaje de modelado es que éstos son instancias de aquellos, o sea, un elemento del lenguaje de modelado es un caso concreto de un metaelemento del nivel de metamodelado. Esta definición puede asimismo aplicarse al metamodelo, que puede ser descrito en términos de otro nivel superior, el metametamodelo, y cada metaelemento es una instancia de un metametaelemento. Esta jerarquía de definiciones se podría extender tantos niveles como se quisiera, pero en general, hay consenso sobre el uso de cuatro niveles distintos de abstracción:

- Metametamodelo. Describe la infraestructura para la arquitectura de metamodelado, incluyendo el lenguaje de descripción de metamodelos.
- Metamodelo. Describe el lenguaje de especificación de modelos.
- Modelo. Describe el lenguaje para describir sistemas.
- Elementos de usuario. Describe los valores concretos de un sistema particular.

Cuando un nivel es una instancia del nivel superior, se habla de metamodelado débil (*loose metamodeling*). Si se consigue que cada elemento de un nivel sea instancia de un solo elemento del nivel superior, se habla de metamodelado fuerte (*strict metamodeling*).

Uno de los casos más populares del uso de metamodelos en la definición de lenguajes de modelado es el que usa el OMG para describir diferentes lenguajes de modelado y su conexión. El OMG usa dos especificaciones distintas para modelar sistemas informáticos distribuidos y sus interfaces en CORBA:

- Lenguaje Unificado de Modelado (*Unified Modeling Language*, UML)
- Servicios de Meta-Objetos (*Meta-Object Facility*, MOF)

La especificación de UML define un conjunto de lenguajes gráficos para visualizar, especificar, construir y documentar los elementos de sistemas distribuidos orientados a objetos. La especificación incluye un metamodelo, una notación gráfica y un servicio de IDL en CORBA que permite el intercambio de modelos entre las herramientas que manejan el metamodelo de UML y repositorios de metadatos.

La especificación de MOF [90] define un conjunto de interfaces IDL en CORBA que pueden usarse para definir y manipular un conjunto de metamodelos y sus correspondientes modelos. Entre estos metamodelos están el metamodelo de UML, el metametamodelo de MOF y, según los planes actuales, estarán los metamodelos de las nuevas propuestas del OMG que usen metamodelado.

UML y MOF están basados en la arquitectura de metamodelado de cuatro niveles, por lo que se ha intentado que estén lo más alineado posible y que comparan la mayor parte de los elementos semánticos centrales. Este alineamiento permite reutilizar los elemento de UML en MOF para visualizar los elementos de los metamodelos. El metamodelo de UML puede ser considerado una instancia del As a metametamodelo de MOF. Como los objetivos de MOF y de UML son distintos, no se ha conseguido seguir un metamodelo fuerte, sino que se ha optado por el metamodelado débil. A pesar de estas diferencias, la similitud entre la estructura de ambos modelos es muy grande y la mayoría de los conceptos pueden ser traducidos de un nivel a otro de manera directa. Para aquellos elementos para los que no es posible esta aplicación inmediata, se definen reglas de transición de un nivel a otro.

6. Metodologías de desarrollo de sistemas de tiempo real

6.1. Metodologías estructuradas

McDermid [79] divide los métodos de diseño en informales, estructuradas y formales. Los estructurados se diferencian de los informales en que aquéllos usan métodos de representación, que pueden ser gráficos, bien definidos, construidos a partir de un pequeño número de componentes predefinidos interconectados de manera controlada. Sin embargo, a diferencia de los métodos formales, los lenguajes de los métodos estructurados carecen de la base matemática que permita realizar análisis formal de propiedades del sistema. Las metodologías estructuradas no incorporan fundamentos formales debido a la complejidad que presentan, que implica un coste muy alto en la formación de personal cualificado para su uso, especialmente en sistemas de gran complejidad.

HOOD [49], JSD [62] y MASCOT [113] son algunos ejemplos de metodologías estructuradas de diseño. Estas metodologías incorporan mecanismos para especificar las características no funcionales de los sistemas de tiempo real, como la concurrencia, los requisitos temporales, la tolerancia a fallos y la interacción con los dispositivos físicos.

Las metodologías estructuradas más usadas son las denominadas metodologías de diseño modular, que incorporan aspectos funcionales para la especificación de requisitos y abstracciones de datos en el resto del diseño.

A partir de los requisitos se obtienen diagramas de flujos de datos que identifican procesos secuenciales de cálculos y que pueden ser refinados hasta obtener transformaciones de datos de bajo nivel.

En el siguiente paso se tiene en cuenta la concurrencia inherente del siste-

ma y se agrupan los flujos de control secuenciales en hebras de acción concurrentes, en función de diferentes heurísticos. Por ejemplo, se pueden agrupar en función de los componentes físicos que usen, en base a consideraciones temporales o a su criticidad.

Seguidamente, se definen las tareas que van a componer el sistema a partir de las hebras de acción concurrentes previas. La división en tareas dependerá del estudio de la concurrencia de la aplicación, la conexión entre procesos y el acoplamiento y la cohesión entre las tareas. En esta definición se tiene en cuenta el modelo de tareas en el que se se van a implementar, para que dicha implementación sea lo más fácil posible.

Finalmente, a partir de las tareas, se crean las estructuras de datos que se van a usar en el programa, bien basándose en tipos abstractos de datos o en objetos. La característica diferenciadora es que en estas metodologías se tiene en cuenta la concurrencia antes que el modelo de datos.

6.2. Metodologías orientadas a objetos

La cualidad fundamental de las metodologías orientadas a objetos es que construyen el sistema desde un primer momento enfocando la atención sobre los datos, creando una red de entidades que se comunican, llamadas objetos. Estas metodologías se han mostrado especialmente adecuada en la definición de sistemas interactivos, por lo que se adecúan bien a los sistemas de tiempo real.

Hay dos modelos de concurrencia en los diseños orientados a objetos, el implícito y el explícito. En el implícito se supone en las primeras fases del diseño que cada objeto es una unidad concurrente de ejecución y los análisis se hacen en función de esa suposición. En las fases finales, el modelo de concurrencia se concreta en función de las posibilidades que ofrezca el

sistema subyacente. En el modelo explícito los objetos se agrupan desde un primer momento en procesos, que son los elementos concurrentes.

De entre estas metodologías destacamos HRT-HOOD, Octopus y ROOM que describimos brevemente en las siguientes secciones.

HRT-HOOD

HRT-HOOD [29] surge como una extensión de HOOD para incorporar los aspectos no funcionales de los sistemas de tiempo real. HRT-HOOD complementa las fases que considera habituales en otras metodologías de desarrollo de software —definición de requisitos, diseño estructural, diseño detallado, codificación y pruebas— para evitar que los aspectos claves en los sistemas de tiempo real se aborden demasiado tarde.

Para construir el sistema se describen *compromisos*, cada vez más específicos, que no se pueden modificar en niveles inferiores. Las *obligaciones* son los aspectos no cubiertos por los compromisos, y que el nivel inferior debe resolver. Este proceso de transformación de obligaciones en compromisos para el siguiente nivel está frecuentemente limitado por las *restricciones* que imponen los niveles inferiores.

Esos tres elementos influyen en gran medida en el diseño estructural. En el diseño de la parte lógica de la arquitectura se suelen tratar aspectos funcionales y es, en general, independiente de las restricciones del entorno físico. La parte física de la arquitectura tiene en cuenta los requisitos y otras restricciones y se encarga de los aspectos no funcionales. La arquitectura física es la base de los análisis de los requisitos no funcionales una vez que se hayan llevado a cabo el diseño detallado y la implementación. La arquitectura física es un refinamiento de la lógica, aunque ambas se van desarrollando en un proceso iterativo y concurrente.

Una vez terminado el diseño estructural se procede a realizar el diseño detallado, sobre el que habrá que hacer nuevas estimaciones del tiempo de respuesta del código conseguido, para ver si se sigue cumpliendo el análisis de planificabilidad que se hizo en el diseño de la arquitectura física, generalmente con tiempos estimados de manera menos precisa.

En el diseño de la arquitectura lógica sólo se permite un conjunto concreto de objetos: activos, pasivos, protegidos, cíclicos y esporádicos. Cada tipo está restringido en varios aspectos, como el tipo de llamadas que puede hacer o la concurrencia interna, para facilitar el análisis.

La fase de diseño físico lleva a cabo la aplicación de los elementos definidos en la arquitectura lógica a los elementos físicos sobre los que se construirá el sistema. Para realizar los análisis necesarios sobre las características no funcionales hace falta que el diseño se haya hecho de forma que permita el análisis y que se puedan conocer las características temporales de la plataforma sobre la que se ejecutará el sistema. En esta fase se han de realizar tareas de ubicación de objetos, planificación de procesos y redes y dependabilidad, incluyendo las cuestiones relativas al manejo de errores.

Los elementos del diseño se representan gráficamente mediante objetos, similares a los de HOOD. La única diferencia es que incluye nueva información sobre el tipo de objeto que es. Los objetos distinguen entre la interfaz y su implementación interna y pueden descomponerse a su vez en objetos más simples. Los requisitos no funcionales se expresan a través de atributos de tiempo real que indican cuestiones como el período, plazos temporales, etc.

HRT-HOOD ofrece reglas sobre cómo traducir los objetos especificados a código en un lenguaje de programación, preferiblemente Ada 95. También se ofrecen reglas sobre cómo habilitar los objetos en función de su tipo en los diferentes nodos de la red en un sistema distribuido para permitir la

planificabilidad.

En la figura 1.6 se muestra el esquema de mayor nivel de una bomba de extracción de agua en una mina especificado en HRT-HOOD.

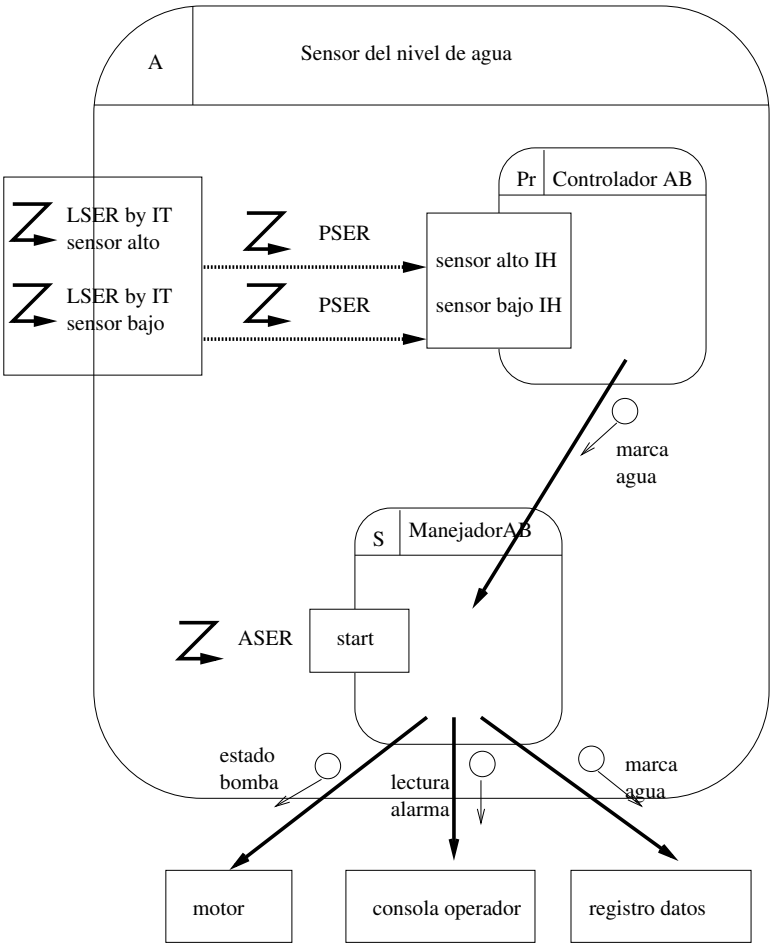

Figura 1.6: Parte del sistema de control de una mina especificado en HRT-HOOD (extraído de [29].

Octopus

Octopus [63] es una metodología basada en otras dos previas, OMT [101] y Fusion [33], que ha intentado mantener las partes positivas de ambas y añadir los elementos necesarios para el desarrollo de sistemas de tiempo real.

Octopus tiene modelos estructural, funcional y dinámico a nivel de sistema, subsistema, clases y objetos. Estos niveles de abstracción se reparten entre las diferentes fases: de requisitos, de arquitectura de sistemas, de análisis de subsistemas, de diseño de subsistemas y de implementación de subsistemas.

En la fase de especificación de requisitos se intentan identificar los requisitos relevantes para el software describiéndolos mediante casos de uso [64]. Los casos de uso se describen en lenguaje natural y se puede mostrar la relación entre ellos a través de diagramas de casos de uso. El paso final de esta fase en la construcción del diagrama de contexto del sistema, en el que se muestran las relaciones entre el entorno y el sistema.

En la fase de arquitectura del sistema, éste es dividido en subsistemas para poder abordar la complejidad del conjunto. La división en subsistemas facilita el desarrollo en paralelo y la reusabilidad, aunque obliga a definir de manera cuidadosa los interfaces entre los diferentes subsistemas. Para conseguir que el desarrollo del software sea independiente del hardware subyacente, en los proyectos en Octopus siempre hay una capa que aísla el hardware.

En la fase de análisis se describen los subsistemas como una composición de objetos. Se usa el modelo implícito de concurrencia. Los objetos se describen desde tres puntos de vista, el modelo de objetos, que describe la estructura estática en base a diagramas de clases, el modelo funcional, que describe la interfaz funcional en base a impresos de operaciones, y el modelo dinámico, que describe las operaciones del subsistema y aborda los aspectos reactivos y de tiempo real. Para los distintos pasos de esta fase se usan diferentes diagramas, entre los que podemos citar las máquinas de estados y MSC para los escenarios.

En la fase de diseño se pasa del modelo implícito de concurrencia a uno

explícito, asignando objetos a procesos. El proceso consta de varios pasos. Primero se establecen las hebras de interacción entre objetos y se depuran hasta que se convierten en hebras de eventos. Después se establece el tipo de conexión entre los objetos, síncrona o asíncrona. Finalmente, los objetos se agrupan y se perfilan los procesos asociados a los grupos de objetos. En esta fase se establecen qué objetos de los subsistemas responden a los eventos externos y cuáles son las secuencias de acciones que componen la respuesta total.

La fase de implementación depende del entorno final y del lenguaje de programación. Parte de la traducción al código ejecutable puede ser automatizada, como, por ejemplo, las cabeceras de las clases.

Esta metodología no incluye ningún mecanismo de análisis de propiedades temporales, como la planificabilidad.

Octopus/UML [96] es una versión mejorada de Octopus que cumple la terminología y la notación definida en UML. En algunos aspectos va más allá de UML, especialmente en los aspectos propios de los sistemas de tiempo real. Los elementos nuevos que no se adecuan a UML son denominados *Special Octopus Features (SOF)*.

ROOM

ROOM (*Real-Time Object Oriented Modeling*) [110] es otra metodología de desarrollo orientada a objetos y, según sus autores, cubre las deficiencias de otras metodologías anteriores para el desarrollo de sistemas de tiempo real. Para ello se centra en desarrollar sistemas de este tipo, incluyendo un conjunto básico de elementos de diseño suficiente para describir adecuadamente los sistemas de tiempo real, pero sin querer ser una metodología demasiado genérica para mantener su utilidad y evitar que sea demasiado complicada.

Los modelos usados son gráficos y se han intentando eliminar las discontinuidades entre las diferentes etapas para reducir la posibilidad de errores.

ROOM puede usarse para crear un modelo ejecutable orientado a objetos de un sistema de tiempo real y para soportar estrategias comúnmente usadas en la construcción de modelos.

El elemento básico de los modelos de ROOM son las *clases de actores*, que representan un conjunto de objetos con características y funcionamiento comunes. Estos objetos, u actores, son instancias de las clases de actores y representan máquinas lógicas, independientes y que funcionan concurrentemente.

Los actores del sistema se comunican a través del envío y recepción de *mensajes*. Los conjuntos de mensajes relacionados se agrupan en *clases de protocolos*. Estas clases de protocolos especifican las señales asociadas a los mensajes, la dirección de envío y los datos que se intercambian.

Los actores se relacionan entre sí asociando uno de los elementos de sus *interfaces*, los *puertos*, a los protocolos previamente definidos. Los puertos son, por tanto, vías de comunicación claramente identificados por los que fluye la información a través de un conjunto de mensajes bien definidos por el protocolo.

El funcionamiento de los actores de una clase se especifica mediante máquinas de estados (llamadas aquí *ROOMcharts*). En estas máquinas las transiciones entre estados son disparadas por la llegada de un mensaje a través de los puertos de la interfaz del objeto. Cuando una máquina de estados cambia de un estado a otro como respuesta a la llegada de un mensaje, se ejecutan varias acciones asociadas, respectivamente, al estado de salida, la transición y el estado de llegada. También se pueden definir variables de estado extendidas en términos de clases de datos.

El ámbito de complejidad de los actores puede ser tan grande como se necesite. Los actores complejos se pueden definir jerárquicamente en función de otros actores más simples, así como las máquinas de estados pueden expandir sus estados para definir el funcionamiento a mayor nivel de detalle.

Los diagramas de ROOM pueden incorporar instrucciones de otros lenguajes de programación para representar detalles de bajo nivel como el paso de mensajes o la manipulación de datos.

ROOM usa secuencias de mensajes para especificar escenarios, que se pueden usar para definir los requisitos que se esperan del sistema y contrastarlo con ejecuciones reales del sistema para comprobar si se cumplen aquéllos.

La máquina virtual de ROOM ofrece servicios de tiempo a través de un protocolo que permite acceder al instante actual de un reloj global, activar y desactivar temporizadores y generar señales de *timeout*.

ROOM proporciona la mayoría de las características de la orientación a objetos, como la herencia y la especialización y la creación dinámica de actores y relaciones. Sin embargo, estas cuestiones no han sido bien resueltas a día de hoy en lo relacionado con sus consecuencias sobre el funcionamiento de tiempo real del sistema.

ROOM asume un modelo de prioridades de eventos, pero sin prioridad garantizada. Aunque la idea de asignar prioridades a los eventos en vez de a los procesos también la defendemos en la metodología que proponemos, en ROOM no se garantiza la semántica de las prioridades para evitar tener que resolver cuestiones que pueden surgir, como la inversión de prioridades.

La validación se hace mediante ejecución de escenarios, especialmente a alto nivel de abstracción. La del código de bajo nivel se hace con métodos tradicionales.

ROOM tampoco incorpora mecanismos para realizar análisis de planifi-

cabilidad del sistema.

6.3. Metodologías basadas en SDL

SOMT

La tecnología SOMT [44] integra análisis orientado a objetos con diseño en SDL y se centra fundamentalmente en sistemas para los que consideran que estos modelos son especialmente apropiados, como los sistemas empotrados, los de tiempo real y los que tienen una alta carga de comunicación.

La metodología SOMT se divide en las cinco fases habituales: análisis de requisitos, análisis del sistema, diseño del sistema, diseño de objetos e implementación. Además de estas fases, la metodología añade como elemento fundamental guías para pasar de los modelos de una fase a los de la siguiente.

En la fase de análisis de requisitos se parte de un documento textual de requisitos y se intenta conseguir una definición formal de los mismos. Esta definición se hace a distintos niveles. Por un lado, se realizan diagramas de clases para obtener el modelo de objetos de los requisitos y, por otro, se establecen los casos de uso mediante texto estructurado o diagramas MSC. También se crea un diccionario de datos.

En el análisis de sistemas se describe la arquitectura del sistema y se identifican los objetos necesarios para implementar la funcionalidad requerida. Los objetos se representan mediante diagramas de clases y las relaciones entre ellos se especifican mediante escenarios descritos en MSC.

En el diseño del sistema se define la estructura de implementación y se identifican las estrategias globales de diseño. En esta fase se definen la estructura modular del diseño y la definición de la arquitectura mediante diagramas de sistemas y bloques en SDL y definición de interfaces basados en señales y llamadas remotas a procedimientos. También se establecen los

casos de uso del diseño mediante MSC.

En la fase de diseño de objetos se crea una descripción completa y formalmente verificable del funcionamiento del sistema, básicamente mediante diagramas de procesos en SDL. Esta fase también incluye tareas de prueba y verificación.

En la fase de implementación y pruebas se produce el producto ejecutable final. Las actividades de esta fase pueden incluir la generación automática de código, la integración en el entorno físico y la realización de pruebas.

Una de las herramientas que proporciona la metodología SOMT para guiar el paso entre los diferentes modelos de las sucesivas fases del desarrollo son los *implinks*, los enlaces de implementación, y la posibilidad de *pegarlos* en una fase posterior. Con los *implinks* se pueden marcar los objetos de las fases iniciales de análisis y diseño y posteriormente integrarlos y relacionarlos con elementos de los diagramas SDL, de tal manera que se facilita la navegación entre los conceptos relacionados en los diferentes diagramas y se puede comprobar mejor su coherencia, especialmente cuando se producen cambios.

The Codesign Methodology

Mitschele-Thiel y Slomka proponen en [83] una metodología de diseño mixto de sistemas con componentes físicos y lógicos basada en extensiones de SDL y MSC, reutilizando procesos de desarrollo con estos formalismos previos. Ambos son extendidos para incluir aspectos no funcionales y de implementación.

MSC es extendido para incluir la expresión de restricciones temporales. Esta extensión es denominada PMSC. SDL también es extendido en un lenguaje denominado SDL* (ver sección 4.1). En ambos casos las extensiones consisten en anotaciones sobre diagramas en los lenguajes originales que in-

cluyen información no funcional, como restricciones de tiempo o de despliegue en componentes físicos. Con estas extensiones se pretenden abordar los aspectos no funcionales y de implementación de manera formal.

La metodología sigue un desarrollo iterativo en el que cada ciclo corresponde a una fase del desarrollo: diseño inicial, formalización, diseño de la implementación e implementación final. Durante el diseño se realiza un refinamiento paralelo desde la especificación en PMSC y SDL* hasta la implementación en VHDL [19] y C.

La fase inicial de especificación de requisitos produce la descripción informal de los requisitos del sistema, tanto los funcionales como los no funcionales. En la fase de diseño inicial se modelan los casos de uso en PMSC, empleando las extensiones para incluir los requisitos temporales. Con alguna información adicional, sobre este primer diseño pueden llevarse a cabo análisis de rendimiento y contrastar sus resultados con las restricciones temporales especificadas. En esta fase se hace en SDL* la descripción estructural del sistema.

En la fase de formalización se hace un refinamiento paralelo de la estructura descrita en SDL* y de los casos de uso en PMSC que reflejen el refinamiento de SDL*. En esta fase se incluyen los aspectos no funcionales en el diseño SDL*, como los necesarios para realizar la división entre componentes físicos y lógicos. Los análisis de rendimiento de la fase anterior pueden servir ahora para centrarse en aquellas partes del sistemas que presentan mayores problemas en esta faceta. Los diseños refinados, tanto en PMSC como SDL*, también son analizados desde el punto de vista del rendimiento.

En la fase de implementación se llevan a cabo los estudios de verificación y simulación para, por ejemplo, estudiar la presencia de bloqueos o la planificabilidad. A continuación se genera el código de manera automática, en C

para la parte lógica y en VHDL para los componentes físicos, pudiendo escoger entre una biblioteca de componentes predefinidos o crear componentes específicos para la aplicación.

En la fase de pruebas se pueden generar bancos de pruebas de manera automática a partir de la especificación de los aspectos funcionales. Estas pruebas suelen estar descritas en TTCN [48]. Los autores han desarrollado una extensión de TTCN para tener en cuenta aspectos temporales especificados en PMSC.

La metodología está soportada por un conjunto de herramientas automáticas que tienen como datos de entrada la descripción de la arquitectura del sistema y de su funcionamiento dinámico en SDL* y PMSC, respectivamente, y que, en diferentes pasos, y teniendo en cuenta información adicional de las bibliotecas de componentes lógicos y físicos, acaba haciendo una optimización del sistema, que es otra descripción SDL* del sistema que cumple las restricciones especificadas.

6.4. Metodologías basadas en UML

ROPES

La metodología ROPES *Rapid Object Oriented Prototyping for Embedded Systems* [39] hace uso de UML para crear los modelos de las diferentes fases del desarrollo en que se divide: análisis, diseño, traducción y pruebas. ROPES sigue un ciclo de vida iterativo y fomenta la creación temprana de prototipos para verificar la calidad del sistema.

La fase de análisis se compone de tres subfases. La primera es la fase de análisis de requisitos, en la que se establecen los objetivos que ha de cumplir el programa. En esta fase se han de identificar los casos de uso y los actores y eventos externos, ya que el sistema se ve aún como una caja negra. Si los

casos de uso son muy complejos, se descomponen y se relacionan entre sí. También tienen que identificarse las restricciones, como las interfaces o los parámetros de rendimiento, entre los que se incluyen los requisitos de tiempo real de los eventos identificados. Para representar los casos de uso se usan inicialmente diagramas de casos de uso, en los que se incluyen los actores y las tareas que llevan a cabo. Si es necesario representar la dinámica de estos casos de uso se emplean diagramas de secuencia y de estados.

La siguiente subfase en aquellos sistemas cuya dimensión lo justifique es el análisis de sistema, donde se establece qué partes del sistemas serán implementadas mediante dispositivos físicos y cuáles serán programadas.

En la fase del análisis de objetos se identifican los objetos del sistema y las clases que los contienen. Se relacionan entre sí, se identifican sus operaciones, se define el funcionamiento dinámico y se traducen los requisitos temporales de los eventos externos en requisitos temporales sobre los objetos que toman parte en la respuesta al evento. En esta fase se usan fundamentalmente diagramas de clases y objetos. Para el análisis del funcionamiento dinámico de los objetos se usan diagramas de estados asociados a las clases y se definen escenarios mediante diversos diagramas, como diagramas temporales, de secuencias o de actividades.

En la fase de diseño, se ha de escoger una solución que implemente de manera eficiente, y cumpliendo las restricciones impuestas, el modelo que resulta de la fase de análisis. En esta fase emerge de manera explícita el modelo de concurrencia, ya que se distinguen los objetos activos y los pasivos. También se elige la política de planificación y de asignación de prioridades. Se asignan las clases y objetos a los componentes, en especial cuando se trata con sistemas distribuidos. Se implementan las relaciones y las máquinas de estados. En la metodología se divide el diseño en tres fases: estructural,

mecánico y detallado. En cada una de ellas las actividades que se llevan a cabo son cada vez de mayor nivel de detalle.

Tras el diseño se efectúa la traducción del modelo UML a código fuente. Y, con ayuda de un compilador, éste se traduce a código ejecutable. La traducción de UML a código fuente puede ser llevada a cabo en parte por herramientas automáticas, pero el programador ha de incluir todavía gran parte del código manualmente. Se incluyen pruebas de caja blanca, que pueden ser incrementales, en las que se van incluyendo módulos en el sistema, o de validación del sistema completo.

Finalmente, se propone la realización de bancos de pruebas basados principalmente en los escenarios identificados.

En el libro en que detalla la metodología, el autor reserva la última parte a hablar de modelado orientado a objetos de tiempo real avanzado, en el que detalla cuestiones relativas al modelo de tareas y su planificación, el modelado dinámico y los marcos de tiempo real.

Aunque, en general, la presentación de la metodología es de muy alta calidad desde el punto de vista de la ingeniería del software, por otro lado, da la impresión de que el uso que hace de UML es demasiado genérico, sin extenderlo ni adaptarlo para las características propias de los sistemas de tiempo real, aún cuando muchos autores han hecho notar la falta de adecuación de UML para algunas cuestiones de dichos sistemas.

UML para el modelado de sistemas complejos de tiempo real

Selic y Rumbaugh defienden en [111] la utilidad de la metodología ROOM (ver sección 6.2) en el desarrollo de sistemas de tiempo real, en especial los dirigidos por eventos y distribuidos, debido en parte a la presencia de elementos especializados en la definición de la arquitectura de este tipo de sistemas.

En este trabajo, actualizan los elementos de la metodología ROOM a los conceptos en los que se basa UML sin tener que definir nuevos elementos ni modificar el metamodelo, sino haciendo uso de los mecanismos de extensión usuales de UML. Por ejemplo, el concepto básico de *actor* de ROOM es capturado mediante una especialización del concepto de clase usando el estereotipo «`capsule`».

Los constructores usados para representar los conceptos de tiempo real se dividen en dos grupos, los de modelado estructural y los de modelado de funcionamiento.

La especificación completa de las estructura se consigue a través de una combinación de diagramas de clases y diagramas de colaboración. Los primeros muestran las clases del sistema y las relaciones permanentes entre ellas, independiente del uso concreto. En los diagramas de colaboración se explicitan las relaciones entre objetos de las clases determinadas por ciertos usos concretos del sistema. Los tres constructores principales del modelo estructural son las *cápsulas*, los *puertos* y los *conectores*.

Las cápsulas corresponden al concepto de *actor* de ROOM. Son entidades complejas, que interaccionan con el resto del sistema a través de objetos denominados *puertos*. Los puertos se encargan de procesar las señales de entrada/salida con las que la cápsula se comunica con su entorno. Cada puerto tiene un *protocolo* asociado que determina cuáles son las señales que pueden comunicarse por él. Al restringir de esta manera la comunicación de las cápsulas con su entorno se reduce el acoplamiento y se consiguen objetos más reutilizables. Los conectores, que corresponden al concepto ROOM de *ligaduras*, son vistas abstractas de los canales de comunicación que conectan dos o más puertos. Los puertos ligados por una conexión deben jugar un papel complementario en el protocolo. Los conectores se representan en

los diagramas de colaboración mediante papeles de asociación y capturan las comunicaciones claves entre clases, mostrando cuáles afectan a otras directamente.

Las cápsulas que corresponden a elementos complejos se pueden dividir jerárquicamente en diagramas de subcápsulas unidas mediante conectores. Esta división se puede repetir hasta conseguir cápsulas suficientemente simples. La relación entre la subcápsulas se representa con diagramas de colaboración de UML. El funcionamiento dinámico de una cápsula se expresa mediante máquinas de estados que se comunican con el exterior a través de los puertos finales, de los que recibe las señales externas. Cada subcápsula puede tener su propia máquina de estados. Las máquinas de estados pueden crear y destruir dinámicamente subcápsulas. Se pueden definir *papeles de conexión*, que son moldes genéricos que se pueden concretar *a posteriori* con subcápsulas concretas. Las máquinas de estados se pueden especializar según los mecanismos definidos en UML.

Los puertos de las cápsulas se dividen en dos categorías cuando se ven desde el interior, los *relay ports*, que se conectan a las subcápsulas y los *end ports*, que se conectan a la máquina de estados de la cápsula.

El modelado dinámico se hace a través de los *protocolos*, que especifican el funcionamiento deseado de los conectores. Un protocolo se compone de un conjunto de participantes, cada uno de los cuales juega un papel en el protocolo. Cada papel en el protocolo tiene una lista de las señales que puede enviar y otra de las que puede recibir. El funcionamiento dinámico del protocolo se puede especificar mediante una máquina de estados y, además, algunas interacciones concretas especialmente interesantes se pueden especificar mediante diagramas de secuencias, que han de ser compatibles con la máquina de estados.

Los servicios de tiempo que necesitan las cápsulas para ejecutar acciones relacionadas con el tiempo se adquieren a través de puertos de acceso a servicios concretos que proporcionan habilidades para disparar una transición en un instante absoluto o en uno relativo.

Los puertos se modelan en UML mediante el estereotipo «`port`» aplicado a la clase `Class`. Los puertos tienen una relación de realización respecto al papel de protocolo que implementan y una relación de composición respecto a la cápsula a la que pertenecen.

Los conectores se modelan mediante una asociación y se declaran a través de un papel de asociación en un diagrama de colaboración de cápsulas.

Las cápsulas se modelan mediante el estereotipo «`capsule`» aplicado a la clase `Class`. Las cápsulas tienen relaciones de composición con sus componentes, los puertos, las subcápsulas y los conectores internos. Los puertos de una cápsula aparecen listados en un compartimento propio en el dibujo de la cápsula. En los diagramas de colaboración, los puertos de las instancias de las cápsulas se suelen mostrar como un cuadrado negro (el puerto conjugado se muestra en blanco).

Los protocolos se modelan mediante el estereotipo «`protocol`» aplicado a la clase `Collaboration`. Un protocolo está asociado con todos sus papeles de protocolo. Los papeles de protocolo se modelan mediante el estereotipo «`protocolRole`» aplicado a la clase `ClassifierRole`. Cada papel de protocolo tiene asociaciones con la clase `Signal` para especificar las listas de señales de entrada y salida.

En las figuras 1.7 y 1.8 se muestran, respectivamente, los diagramas de clases y colaboración de un sistema simple en el que se ilustran algunos de los elementos de UML usados para modelar los conceptos de ROOM.

Aunque los autores de esta propuesta admiten la importancia de estable-

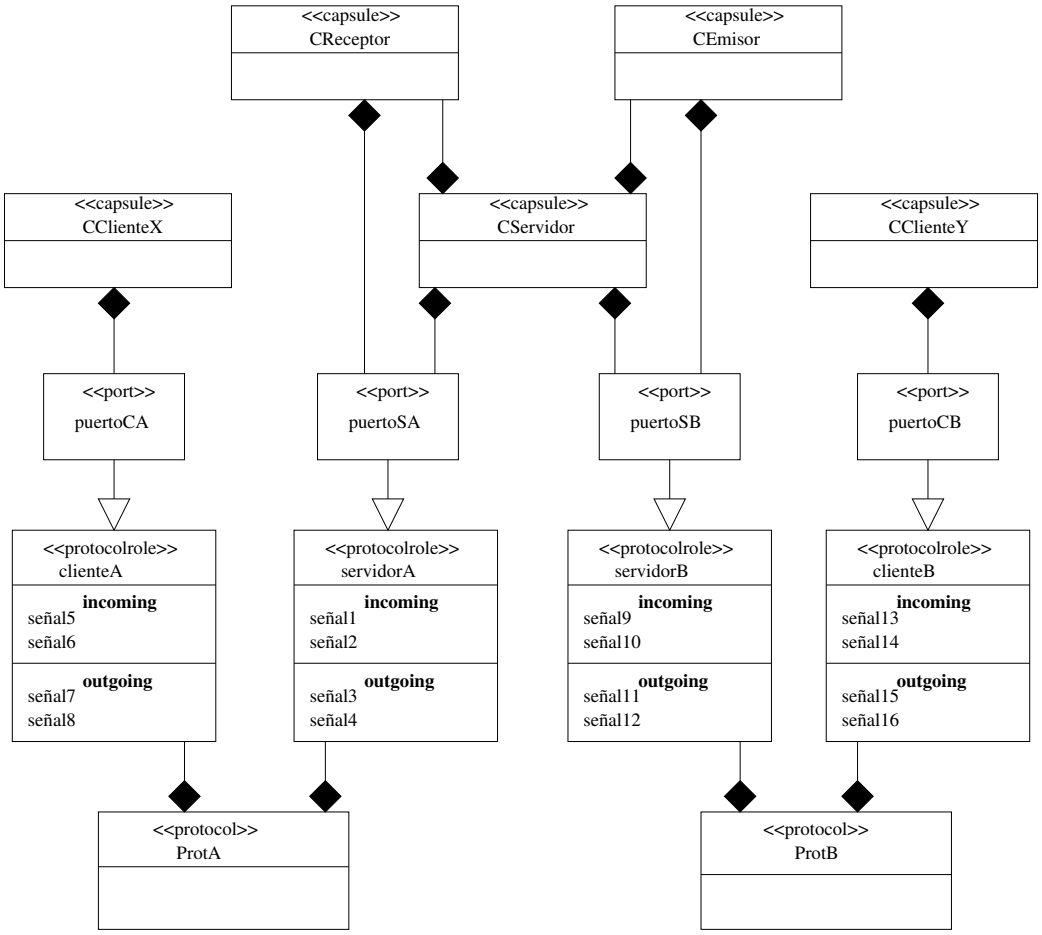

Figura 1.7: Diagrama de clases del sistema.

cer una semántica estática y dinámica, dicen que su definición está fuera del ámbito de la publicación.

COMET

La metodología COMET (*Concurrent Object Oriented and architectural design mEThod*) [54] es un método de diseño para aplicaciones concurrentes. En particular, aplicaciones distribuidas y de tiempo real.

El proceso de desarrollo del método COMET es un proceso orientado a objetos, compatible con el *Unified Software Development Process* y el modelo

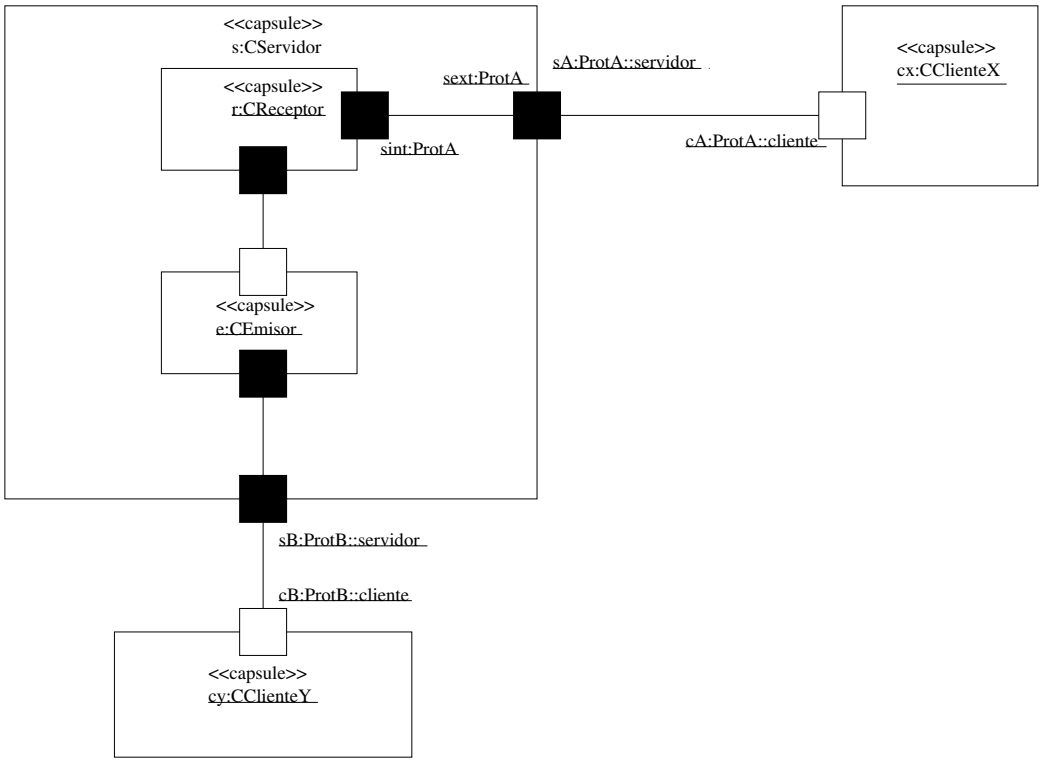

Figura 1.8: Diagrama de colaboración del sistema.

en espiral. El modelo COMET de ciclo de vida del *software* es un proceso de desarrollo altamente iterativo basado alrededor del concepto de caso de uso. Los requisitos funcionales del sistema se determinan en base a actores y casos de uso.

Los casos de uso se pueden ver a diferentes niveles de detalle. En el análisis de requisitos, los requisitos funcionales se definen en términos de actores y casos de uso. En el modelo de análisis, se refinan los casos de uso para describir los objetos que participan y cómo interaccionan. En el modelo de diseño, se desarrolla la arquitectura software, atendiendo las cuestiones relacionadas con la distribución, concurrencia y ocultación de información.

En la fase de requisitos se desarrolla el modelo de requisitos en base a casos de uso en los que participan los actores. Se incluye una descripción textual

de los casos de uso. Si los requisitos no están claros, se puede desarrollar un prototipo básico para intentar entenderlos mejor.

En la fase de modelado de análisis se desarrollan los modelos estáticos y dinámicos del sistema. El modelo estático se encarga de la relación estructural entre las clases del dominio del problema. Las clases y las relaciones se representan en diagramas de clases. Una vez determinados los objetos que se van a incluir en el sistema, se desarrolla un modelo dinámico en el que se refinan los casos de uso y se muestran qué objetos participan en cada uno y cómo se relacionan entre ellos. En esta fase se usan diagramas de colaboración y de secuencia para mostrar la dinámica de los casos de uso. También se pueden usar máquinas de estados para aquellos casos en los que el funcionamiento dependa del estado.

En la fase de modelado del diseño, se diseña la estructura *software* del sistema, en la que el modelo de análisis se aplica sobre el entorno operacional. Se pasa del dominio del problema, representado por el modelo de análisis, al dominio de la solución, representado por el modelo del diseño. Se crean los diferentes subsistemas en función de los criterios aceptados y se establece la comunicación entre ellos, especialmente importante en los sistemas distribuidos. En los sistemas concurrentes y los de tiempo real, además de tener en cuenta los aspectos normales en la orientación a objetos, como la herencia, la ocultación de información, etc., hay que considerar además los conceptos no funcionales inherentes a la concurrencia y los requisitos de tiempo.

Tras acabar el diseño de la estructura *software*, se sigue una estrategia de construcción incremental de *software*, que se basa en seleccionar un subconjunto del sistema a construir en cada incremento. El subconjunto se determina escogiendo los casos de uso que se van a construir y los objetos involucrados en ellos. Esta fase de construcción consta de diseño detallado,

codificación y prueba de las clases incluidas en el subsistema. Las diferentes unidades se van integrando hasta conseguir el sistema completo.

Durante la integración incremental de *software* se llevan a cabo las pruebas de integración de los subsistemas. Las pruebas se basan en los casos de uso seleccionados para el subsistema. Estas son pruebas de caja blanca en las que se analizan las interfaces de los objetos. El resultado de estas pruebas incrementales son prototipos incrementales en los que se van incluyendo cada vez nuevas funcionalidades. Si se detectan errores graves, será necesario volver atrás a las fases anteriores.

Las pruebas del sistema incluyen la prueba de los requisitos funcionales. Estas pruebas ven el sistema como una caja negra y son obligatorias antes de entregar un prototipo o el sistema final al usuario.

6.5. Herramientas automáticas de diseño

Diversas herramientas comerciales implementan estas metodologías, o al menos dan soporte a parte de ellas.

Una de las primeras herramientas de modelado basado en objetos que ofreció la posibilidad de generación automática de código fue ObjectTime Developer, de la empresa ObjectTime. ObjectTime Developer se basa en la metodología ROOM (ver sección 6.2). Tras la aparición y popularización de UML, ObjectTime Developer y ROOM se han fundido con Rational Rose y UML, dando lugar, respectivamente, a Rational Rose Real-Time y UML-RT, que usa los mecanismos de extensión de UML para modelar los conceptos de ROOM. Como se mencionó al hablar de ROOM, el concepto fundamental es un tipo de objeto activo, las *cápsulas*, que se comunican mediante mecanismos concretos, los *puertos*, y cuyo funcionamiento viene definido por máquinas de estados. Rational-Rose Real-Time incorpora la generación de código de

ObjectTime Developer, en la que cada objeto activo tiene su propia hebra de ejecución, que actúa concurrentemente con las otras hebras, y tiene su propia cola de eventos.

Rhapsody, [97] de i-Logix, también ofrece todo el ciclo de vida de UML, del análisis de requisitos a la generación parcial de código pasando por la especificación de objetos. Sin embargo, no ofrecen herramientas de análisis de propiedades de tiempo real estricto, como planificación, ni para la validación de sistemas.

Hay otras herramientas basadas en UML y SDL. Por ejemplo, Tau, de Telelogic, usa UML en el diseño de objetos y MSC para expresar el funcionamiento dinámico del sistema y SDL para la fase de diseño. Incluye traducción automática entre las máquinas de estados de UML y los diagramas de SDL. Ofrece generación automática de código y herramientas de simulación y validación. La validación permite verificar el funcionamiento dinámico especificado mediante diagramas MSC, detectando posibles bloqueos, *livelocks*, etc. Las herramientas de simulación permiten reproducir el funcionamiento que lleva a esa situación de manera fácil.

Todas estas herramientas carecen de mecanismos para el análisis de propiedades de tiempo real o de base formal para la verificación.

Igualmente, ninguna de estas herramientas ofrece la posibilidad de anotar características temporales a los modelos, ni de analizar cuestiones temporales, como el rendimiento o la planificación. Es más, tampoco ofrecen la posibilidad de usar directamente herramientas externas de análisis de planificabilidad.

CAPÍTULO 2

Una semántica de acciones para MML

1. Introducción

La semántica de acciones de UML ha sido adoptada recientemente por el *Object Management Group (OMG)* para "extender UML con un mecanismo compatible para especificar la semántica de las acciones de una manera independiente del software" [4]. Esta semántica define una extensión del metamodelo de UML 1.5 que incluye una sintaxis abstracta para un lenguaje de acciones. Este lenguaje proporciona un conjunto de constructores simples para las acciones como, por ejemplo, acciones de escritura, condicionales y compuestas, que pueden usarse para describir el funcionamiento computacional de un modelo UML. Una parte clave de la propuesta es la descripción en inglés de la semántica del funcionamiento de los objetos, basada en un modelo de la historia de las ejecuciones de los objetos.

Desafortunadamente, la propuesta de la semántica de acciones sufre de un problema frecuente cuando se desarrollan metamodelos grandes en UML —cómo estructurar el modelo de forma que queden claramente separados sus diferentes componentes—. Cuando no se consigue este objetivo, el me-

tamodelo resultante es difícil de entender y modificar, y en particular, de especializar y extender. Además, los metamodelos basados en la semántica actual de UML sufren de la falta de un núcleo con una semántica definida de manera precisa sobre el que construir el metamodelo. Esto significa que a menudo es difícil asegurar la corrección del modelo, y para superar esto, debe invertirse una gran cantidad de trabajo en aclarar la semántica antes de que se puedan hacer más progresos. En el lado positivo, el modelo semántico básico usado en la semántica de acciones, con sus nociones de *snapshots*, historias y cambios parece muy apropiado para definir los diferentes valores que van cambiando en el sistema. Además, las acciones definidas en la propuesta cubren de manera sistemática el amplio rango de acciones necesarias para un lenguaje de acciones útil. Así, si se puede de alguna forma reestructurar mejor lo que es un trabajo significativo, entonces habrá beneficios claros para los desarrolladores y usuarios del lenguaje.

En este capítulo vamos a mostrar cómo la definición de un núcleo con una semántica precisa y el uso de un mecanismo de extensión de paquetes al estilo de Catalisis [41] puede generar una definición mejor estructurada y adaptable de la semántica de acciones. El trabajo está basado en una extensión del Meta-Modelling Language (MML) [31], un lenguaje de metamodelado preciso desarrollado para conseguir que UML se pueda reconstruir como una familia de lenguajes de modelado. MML tiene muchas características en común con el metamodelo de UML. Además, es más pequeño y ya tiene una semántica modelada, siendo así una base útil para explorar la semántica de acciones en general. Sin embargo, hay que dejar claro que éste no es un intento de resolver el problema general de la ejecutabilidad de modelos, sino sólo una propuesta para describir la ejecutabilidad en el contexto de MML.

1.1. Los fundamentos del modelo MML

MML es un lenguaje de metamodelado pensado para reconstruir el núcleo de UML de tal manera que se pueda definir de una manera mucho más simple y que pueda ser extendido y especializado para diferentes tipos de sistemas. Entre otros conceptos básicos, MML establece dos distinciones ortogonales. La primera es una correspondencia entre los conceptos del modelo (la sintaxis abstracta) y las instancias (el dominio semántico). La segunda es la distinción entre los conceptos de modelado y la sintaxis concreta, y se aplica tanto a conceptos del modelo como a las instancias. La sintaxis es la forma en que estos elementos se concretan. Los conceptos del modelo y las instancias están relacionados a través de un paquete semántico que establece la relación de satisfacción entre los modelos y las instancias válidas. Entre los conceptos de modelado y la sintaxis concreta se establece una relación similar como se muestra en la figura 2.1).

Estas distinciones se describen en términos de un patrón de definición del lenguaje, ver figura 2.1. Cada componente del patrón es un paquete. Como en UML, un paquete es un contenedor de clases u otros paquetes. Además, los paquetes en MML se pueden especializar. Éste es el mecanismo clave por el que se generan definiciones extensibles del lenguaje, y es similar al mecanismo de extensión de paquetes definido en Catalysis [41]. Aquí, la especialización de paquetes se indica colocando una flecha de especialización de UML entre paquetes. El paquete hijo especializa (y, por tanto, contiene) todo el contenido del paquete padre.

Otro componente importante de MML es su paquete `core`, que define los conceptos de modelado básicos usados en MML: paquetes, clases y atributos. El paquete actual de conceptos del modelo de MML no proporciona un modelo dinámico del funcionamiento. Así, en las siguientes secciones se describe

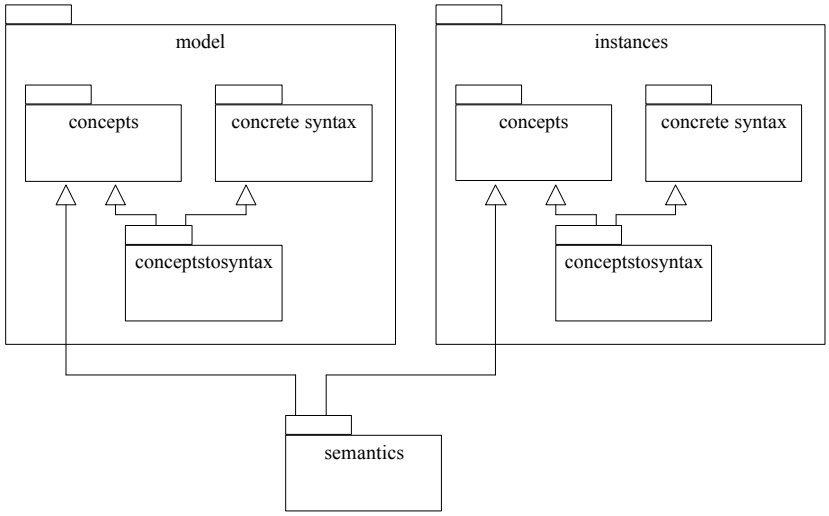

Figura 2.1: El método MMF.

un propuesta para la semántica de las acciones en MML, que se basa en una extensión del paquete `core` de MML.

2. Principios arquitectónicos

Dos han sido los objetivos principales de la definición de la semántica de acciones. El primero, incluir la semántica de acciones como el núcleo dinámico de MML. MML se centró originariamente en el modelado estático, por lo que una extensión natural es añadir características dinámicas. Esto implica la definición de vistas de modelo y de instancias y la separación que puede capturar tanto la noción de acciones como los conceptos y funcionamiento en términos de cambios en el estado de las instancias. El segundo objetivo es hacer uso de la simplicidad de MML y sus modelos de semánticas como

base para explorar la definición de acciones en un lenguaje del estilo de UML. Obviamente, esto es un primer hito para trabajos posteriores para incorporar modelos similares en UML (trabajo que actualmente se está haciendo como parte del proceso de envío de propuestas para UML 2.0 [1]). Finalmente, para conseguir el objetivo de la simplicidad, nos hemos centrado en la definición de un subconjunto de las acciones que se van a usar en la semántica de acciones. Estas acciones (`WriteVariable`, etc.) forman un núcleo de acciones primitivas que todos los lenguajes de acciones deberían incluir. Estas acciones son primitivas en el sentido de que la definición de acciones adicionales podría ser potencialmente descrita en términos de estas primitivas, facilitando así una estrategia de definición más estructurada en niveles. Una ventaja adicional de esto es que una herramienta que pueda ejecutar las acciones primitivas puede actuar potencialmente como motor de ejecución sobre el que se pudieran traducir las definiciones de un lenguaje más rico.

MML está pensado para ser extendido de manera consistente, por tanto la definición de las acciones se hará en otro paquete en MML, siguiendo la estructura del resto de paquetes, es decir, el paquete debería estar compuesto de paquetes para el modelo, las instancias y la semántica, con los paquetes de modelo y de instancias subdivididos a su vez en paquetes para los conceptos, la sintaxis concreta y la correspondencia entre conceptos y sintaxis.

3. Núcleo dinámico

El núcleo dinámico (*Dynamic Core*) intenta modelar la evolución de los valores de los elementos del sistema a lo largo del tiempo, en contraste con la vista del modelo estático que considera las instancias unidas a un único valor.

La estrategia escogida para definir el modelo dinámico considera que los

elementos mutables, los objetos, evolucionan a través de diferentes estados a lo largo de la vida del sistema según cambian sus valores las acciones que se ejecutan sobre ellos. Estos estados contienen todos los valores del elemento en un instante dado. El elemento asociado a ese estado se identifica por medio de un atributo, la identidad (`identity`), que es única en el sistema para ese elemento. Es decir, dos estados se refieren al mismo elemento si y sólo si tienen la misma identidad.

Esta relación entre los sucesivos estados de un elemento también se muestra por medio de la asociación `next`, que significa que un estado dado es el siguiente estado de un elemento sobre el que se ha ejecutado una acción.

Los conceptos del modelo y la semántica siguen siendo iguales que en el núcleo estático y es el paquete de los conceptos de instancias el que cambia. Cada instancia de la metaclase `object` representa un estado de un elemento mutable. El nuevo paquete para los conceptos de instancias del núcleo dinámico se muestra en la figura 2.2.

4. Acciones

En la definición de UML, OCL es un lenguaje formal para expresar restricciones sin efectos laterales. La evaluación de estas restricciones no cambia ningún valor del sistema.

MML tiene su propio lenguaje de restricciones basadas en OCL con unos pocos cambios semánticos. Este lenguaje proporciona todas las expresiones básicas del lenguaje de OCL. El concepto actual de `Expression` en MML cubre las expresiones básicas de OCL. Éstas incluyen expresiones lógicas (`and`, `not`, `equals`, `includes`), referencias a *slots*, variables e iteradores sobre tipos contenedores (`sequences`, `sets` y `bags`). Sin embargo, no define otras expresiones, como las aritméticas.

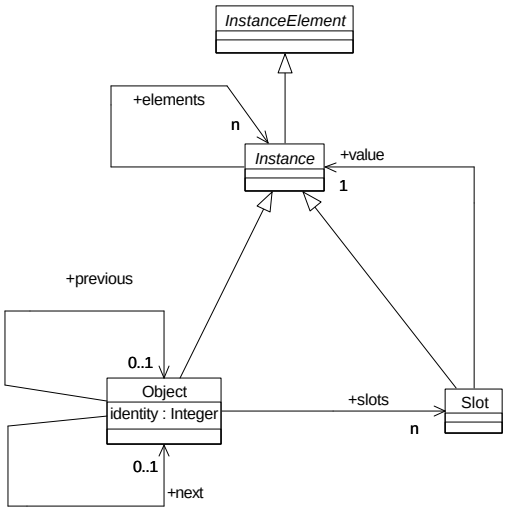

Figura 2.2: Conceptos de instancias del núcleo dinámico (paquete dynamic-core.instances.concepts).

Los métodos de MML se usan para definir procedimientos computacionales. Los métodos no tienen efectos laterales y simplemente sirven para evaluar un conjunto de parámetros contra una expresión OCL para obtener un resultado. Se puede definir un nuevo método simplemente cambiando la expresión.

Las acciones representarán los procedimientos computacionales que cambian el estado de un elemento del sistema. Las acciones no tendrán una expresión que especifique lo que hace la acción, sino que definirán las nuevas clases que especialicen el concepto de acción básica. Su funcionamiento particular se describirá por medio de reglas de corrección. Con esta estrategia, se tiene un conjunto de acciones básicas sobre las que se van a construir las nuevas acciones que sean necesarias.

En este trabajo sólo se va a definir un conjunto básico de acciones que resulte suficiente para permitir fácilmente la definición de las otras acciones

sobre ellas.

Se distinguirán las acciones primitivas de las compuestas. Las acciones primitivas son suficientemente simples y no necesitan definirse en términos de otros elementos más simples. Las acciones compuestas se definen como una composición de acciones más simples, tanto primitivas como compuestas.

4.1. Conceptos de modelo

La figura 2.3 muestra la definición de la clase abstracta `Action`. La clase `Action` especializa la clase `Classifier`. Cada acción tiene un conjunto de parámetros de entrada. Como el orden de los parámetros es significativo, estas asociaciones están ordenadas.

Los parámetros de entrada representan los valores que la acción necesita para ejecutarse. Estos valores pueden incluir el objeto sobre el que se aplica la acción.

Todas las acciones son ejecutadas por un objeto, que puede ser llamado el ámbito (`scope`) de la acción. La asociación entre las clases `Action` y `Class` muestra la clase a la que pertenece el objeto ámbito de la acción.

Métodos

1. El método `allActions()` devuelve el conjunto de todas las acciones de una clase, incluyendo la de sus progenitores. `allActions()` escoge sólo una acción en el caso de que haya múltiples padres con acciones con el mismo nombre. Ésta es tal vez la estrategia más simple para tratar el tema de las colisiones en la herencia múltiple, aunque se podría mejorar definiendo una regla de mezcla si fuera necesario.

   ```
   context uml.staticCore.model.concepts.Class
   allActions() : Set(Action)
   ```

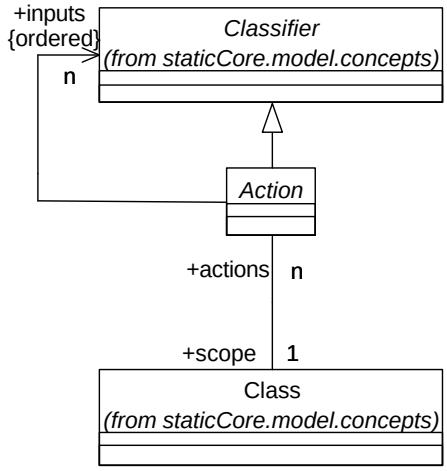

Figura 2.3: Paquete actions.model.concepts.

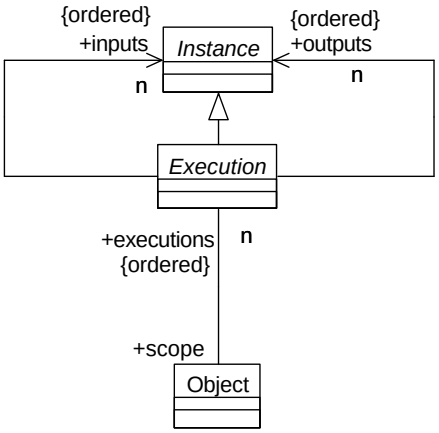

Figura 2.4: Paquete actions.instance.concepts.

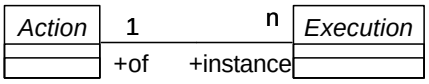

Figura 2.5: Paquete actions.model.semantics.

```
parents->iterate(p; s = actions | s ->union(p.allActions()->
       reject(c | actions->exists(c' | c'.name = c.name)))
```

2. El método `allSubActions()` devuelve todas las subacciones de una acción, anidadas a cualquier profundidad.

```
context uml.actions.model.concepts.Action
allSubActions() : Set(Action)

  self.subactions->
      union(self.subactions.allSubActions()->asSet())
```

4.2. Conceptos de instancias

La clase `Execution` está definida en el paquete de conceptos de instancias mostrado en la figura 2.4. Una instancia de la clase `Execution` representa una ejecución real de una acción.

Una instancia de `Execution` tiene una secuencia de valores de entrada sobre los que ejecutarse. Una entrada distinguida es el estado del elemento que se modifica en la ejecución. Una ejecución también tiene valores de salida. Entre ellos está el nuevo estado del elemento modificado.

Una instancia de `Execution` también tiene una asociación con la instancia de la clase `Object` que la ejecuta, la asociación `self`.

Aunque no hay noción de tiempo ni, por tanto, de ordenación temporal en el modelo dinámico, hay una relación causal entre las ejecuciones en el sistema. Para una ejecución puede haber una ejecución previa (`previous`) y otra siguiente (`next`).

También hay una relación causal entre la ejecución de la acción y los valores usados en ella. Como una acción no puede ejecutarse hasta que todos los valores de entrada hayan sido calculados, los valores producidos después de la ejecución de la acción no pueden ser usados como parámetros de entrada.

La semántica de este paquete se muestra en la figura 2.5.

Métodos

1. El método `allSubExecutions()` devuelve todas las subejecuciones de una ejecución, anidadas a cualquier profundidad.

```
context uml.actions.instance.concepts.Execution
allSubExecutions() : Set(Execution)

  self.subexecution->
      union(self.subexections.allSubExecutions()->asSet())
```

Reglas de corrección

1. Una salida de una ejecución no puede ser usada como entrada de esa ejecución.

```
context uml.actions.instance.concepts.Execution inv:

  inputs->forall(i| outputs->forall(o| i <>o))
```

5. Acciones primitivas

Una acción primitiva es una acción que no se descompone en acciones más simples. La característica común de las acciones primitivas es que no tienen subacciones.

`PrimitiveAction` es una clase abstracta que se especializa en tres subclases diferentes: la acción nula (`NullAction`), la creación de un objeto (`CreateObjectAction`) y la modificación del valor del atributo de un objeto (`WriteSlotAction`). Las figuras 2.6, 2.7 y 2.8 muestran, respectivamente, los paquetes de conceptos del modelo, conceptos de instancias y semántica.

Métodos

1. El método `subActions()` devuelve el conjunto de subacciones inmediatas de una acción. Para las acciones primitivas este conjunto es vacío.

```
context uml.actions.model.concepts.PrimitiveAction
subActions() : Set(Action)

  Set{}
```

2. El método `subExecutions()` devuelve el conjunto de subejecuciones inmediatas de esa ejecución. Para las ejecuciones de acciones simples

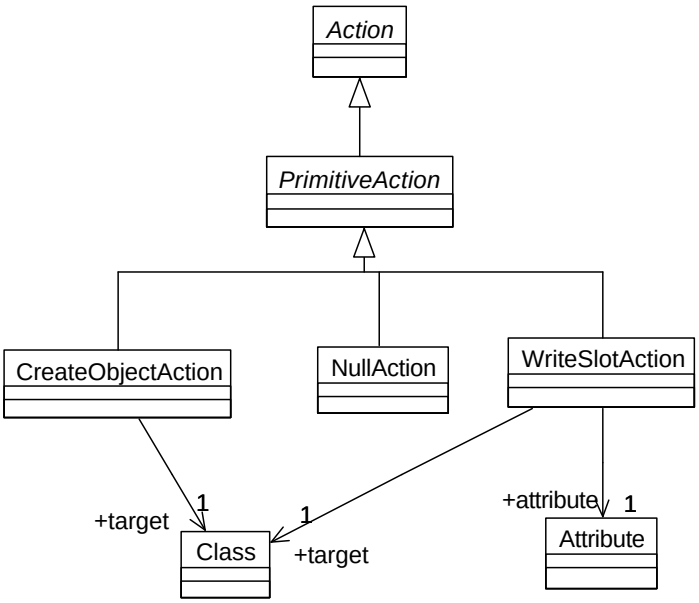

Figura 2.6: Paquete de conceptos del modelo de las acciones primitivas (primitiveactions.model.concepts).

ese conjunto es vacío.

```
context uml.actions.instance.concepts.PrimitiveExecution
subExecutions() : Set(Execution)
  Set{}
```

5.1. Acción nula

La acción nula, como su nombre indica, no hace cambios en el sistema. Se incluye porque las acciones compuestas se han definido para tener al menos una subacción.

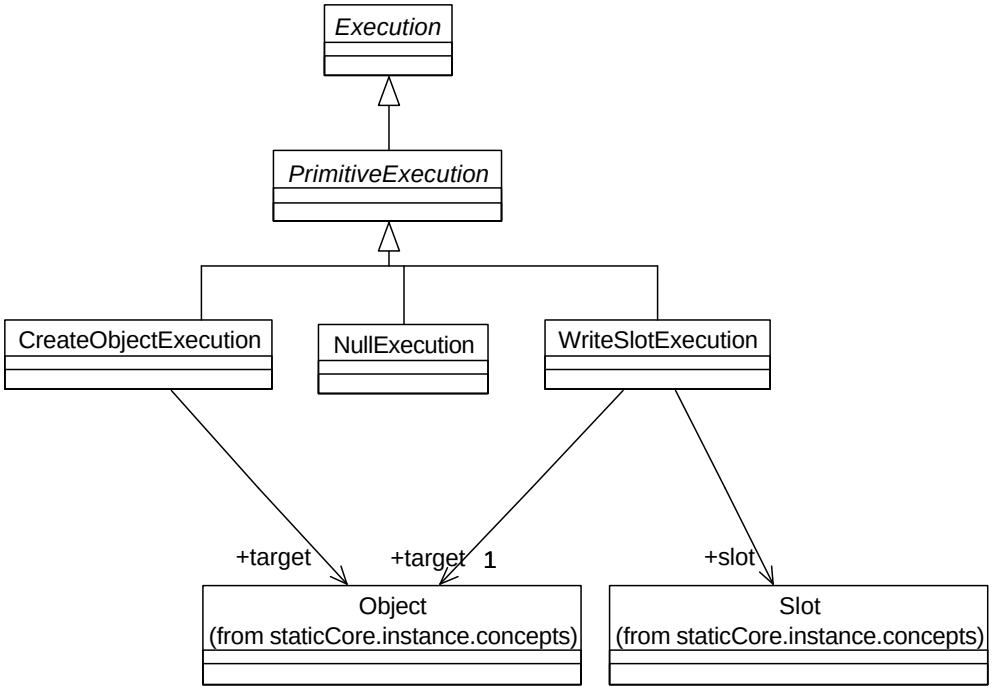

Figura 2.7: Paquete de los conceptos de instancia de las acciones primitivas (primitiveactions.instance.concepts).

Reglas de corrección

1. Una acción nula no tiene parámetros de entrada.

```
context uml.primitiveActions.model.concepts.NullAction inv:

  self.inputs->size = 0
```

2. Una instancia de `NullExecution` no tiene ni parámetros de entrada ni valores de salida.

```
context uml.primitiveActions.instance.concepts.NullExecution inv:

  self.inputs->size = 0 and self.outputs->size = 0
```

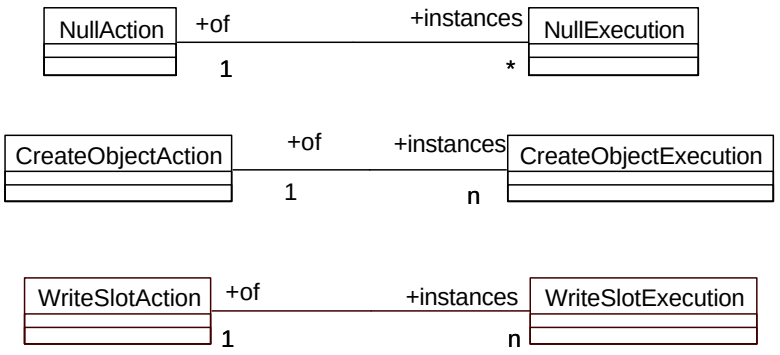

Figura 2.8: Paquete de la semántica de las acciones primitivas (primitiveactions.semantics).

5.2. Acción de crear un objeto

Una acción de la clase `CreateObjectAction` añade un nuevo objeto al sistema. En el lado de los conceptos de modelo, la clase `CreateObjectAction` tiene como entrada la clase del objeto que se va a crear. Esta única entrada se ha renombrado, por claridad, como `target`.

La ejecución de una acción de creación de un objeto tiene un resultado, el objeto creado. Esta acción no hace nada más, ni siquiera inicializa los valores de los atributos del nuevo objeto. Para asegurar que el objeto es nuevo, éste no puede tener un estado previo.

Reglas de corrección

1. Una instancia de `CreateObjectAction` tiene una única entrada.

```
context uml.primitiveActions.model.concepts.CreateObjectAction inv:

  self.inputs->size = 1
```

2. La entrada de una instancia de `CreateObjectAction` es la clase del objeto creado.

```
context uml.primitiveActions.model.concepts.CreateObjectAction inv:

  self.inputs.oclIsTypeOf(Class)
```

3. Una instancia de `CreateObjectExecution` no tiene parámetros de entrada y tiene un valor de salida.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution
inv:

  self.inputs->size = 0 and self.outputs->size = 1
```

4. La salida de una instancia de `CreateObjectExecution` es un objeto.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution
inv:

  self.outputs->at(1).IsTypeOf(Object)
```

5. La salida de una instancia de `CreateObjectExecution` es un objeto de la clase correspondiente a la acción.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution
inv:

  self.outputs->at(1).of = self.of.inputs->at(1)
```

6. La salida de una instancia de `CreateObjectExecution` es un objeto nuevo.

```
context uml.primitiveActions.instance.concepts.CreateObjectExecution
```

```
inv:

  self.outputs->at(1).previous->size() = 0
```

5.3. Acción de escritura de un atributo

Una acción de la clase `WriteSlotAction` cambia el valor de un atributo de un objeto. Los valores del resto de atributos permanecen inalterados.

Cada acción tiene dos entradas distinguidas: `target`, que muestra la clase del objeto a la que pertenece el atributo y `attribute`, que representa el atributo que se va a modificar. Éste debe ser un atributo válido de esa clase.

La ejecución de una acción de esta clase también tiene dos entradas distinguidas: `target` y `slot`, que representan el objeto y atributos concretos que se modifican. El atributo debe ser un atributo válido de ese objeto.

El nuevo valor del atributo es la otra entrada de la ejecución. Este valor debe coincidir con el valor del atributo en el estado del objeto después de la ejecución.

Reglas de corrección

1. Una instancia de la clase `WriteSlotAction` tiene tres parámetros de entrada.

   ```
   context uml.primitiveActions.model.concepts.WriteSlotAction inv:

     self.inputs->size = 3
   ```

2. `target` es una de las entradas de las instancias de `WriteSlotAction`.

   ```
   context uml.primitiveActions.model.concepts.WriteSlotAction inv:

     self.inputs->includes(self.target)
   ```

3. `attribute` es una de las entradas de las instancias de `WriteSlotAction`.

```
context uml.primitiveActions.model.concepts.WriteSlotAction inv:

  self.inputs->includes(self.attribute)
```

4. `attribute` debe ser un atributo válido de la clase.

```
context uml.primitiveActions.model.concepts.WriteSlotAction inv:

  self.target.attributes->includes(self.attribute)
```

5. El tipo del atributo modificado debe ser compatible con el tipo del valor de entrada.

```
context uml.primitiveActions.model.concepts.WriteSlotAction inv:

  self.inputs->at(3).oclIsKindOf(self.attribute.type)
```

6. Una instancia de `WriteSlotExecution` tiene tres parámetros de entrada y un parámetro de salida.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.inputs->size = 3 and self.outputs->size = 1
```

7. El objeto destino es una de las entradas de las instancias de `WriteSlotExecution`.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.inputs->includes(self.target)
```

8. El atributo que se va a actualizar es una de las entradas de las instancias de `WriteSlotExecution`.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.inputs->includes(self.slot)
```

9. El objeto destino debe ser de la clase con la que está asociada la correspondiente acción.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.target.of = self.of.target
```

10. El atributo del objeto debe coincidir con el atributo de la clase asociada a la correspondiente acción.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.slot.of = self.of.attribute
```

11. El atributo debe ser un atributo válido del objeto.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.target->slots->includes(self.slot)
```

12. El tipo del valor de entrada debe ser compatible con el tipo del atributo.

```
context uml.primitiveActions.instance.concepts.WriteSlotExecution
inv:

  self.inputs->at(3).oclIsKindOf(self.slot.type)
```

13. El objeto de salida es el siguiente estado del objeto de entrada.

```
context uml.actions.instance.concepts.WriteSlotExecution inv:

  self.inputs->at(1).next = self.outputs->at(1)
```

14. El valor del atributo actualizado tras una `WriteSlotExecution` es el valor de entrada.

```
context primitiveActions.instance.concepts.WriteSlotExecution inv:

  ouputs->at(1).slots->forall(s |
        s = self.slot implies s.value = inputs->at(3))
```

15. El resto de los atributos del objeto destino permanecen inalterados.

```
context uml.actions.instance.concepts.WriteSlotExecution inv:

  self.outputs->at(1)->forall(s |
      s <>self.slot implies s.value = self.target.slot.value)
```

6. Acciones compuestas

A diferencia de las acciones primitivas, las demás acciones que vamos a definir son complejas y pueden dividirse en subacciones. Vamos a definir dos tipos de acciones compuestas, las secuenciales y las paralelas. Las figuras 2.9, 2.10 y 2.11 muestran, respectivamente, los paquetes de conceptos del modelo, conceptos de instancias y semántica. Estas acciones no hacen directamente ningún cambio en el sistema, sino que éstos se hacen en las subacciones.

El modelo de concurrencia que se modela es un modelo de intercalado de acciones, donde no se ejecutan varias acciones simultáneamente, sino una detrás de otra.

Reglas de corrección

1. Una acción compuesta no tiene parámetros de entrada.

```
context uml.compositeActions.model.concepts.compositeAction inv:

  self.inputs->size = 0
```

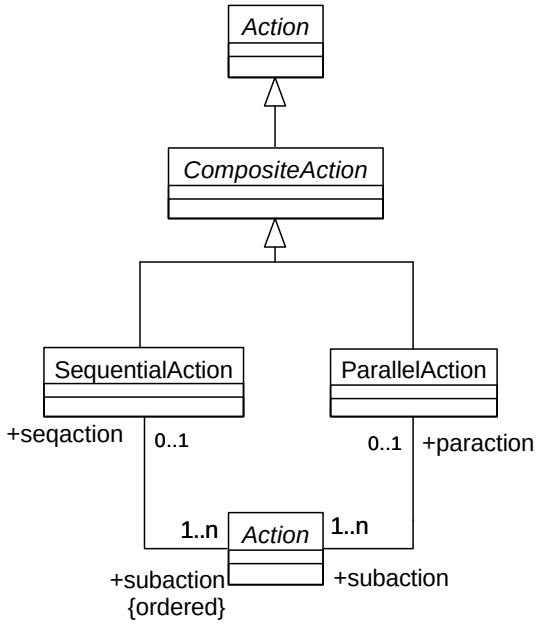

Figura 2.9: Paquete de conceptos del modelo de las acciones compuestas (compositeactions.model.concepts).

2. La ejecución de las subacciones de una ejecución compuesta se ejecutan secuencialmente. Esto implica que una ejecución no puede usar como entrada la salida de una ejecución posterior.

```
context uml.compositeActions.instance.concepts.CompositeAction inv:

  self.input->excludesAll(self.subsequents().
  allSubExecutions().outputs)
```

Métodos

1. El método `subExecutions()` devuelve las subejecuciones de una ejecución compuesta son sus subejecuciones.

```
context uml.compositeActions.instance.concepts.compositeExecution

subExecutions(): Seq(Execution)
```

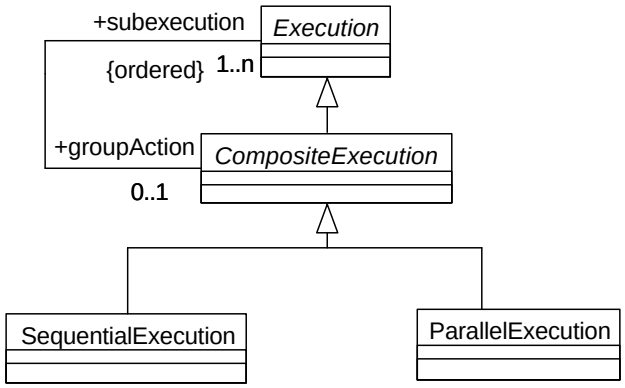

Figura 2.10: Paquete de conceptos de instancias de las acciones compuestas (compositeactions.instance.concepts).

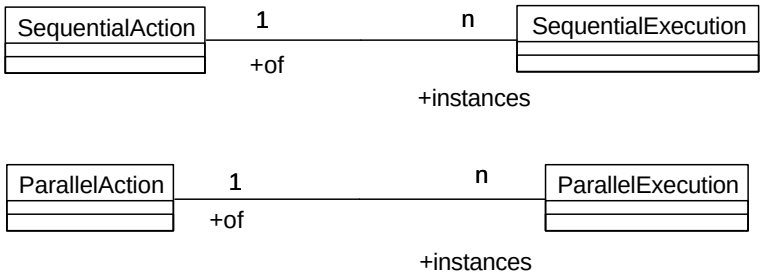

Figura 2.11: Paquete de la semántica de las acciones compuestas (compositeactions.semantics).

```
self.subexecution()
```

2. El método `pos` devuelve la posición de una ejecución en la secuencia de subejecuciones de una acción compuesta.

```
context uml.compositeActions.instance.concepts.compositeExecution
pos(): Integer

  self.posAux(self.groupExecution.subexecution, 1)
```

3. Método auxiliar para definir el método `pos`.

```
context uml.compositeActions.instance.concepts.compositeExecution
posAux(sa : Seq(Executions), i : Integer): Integer

  ifi >sa->size() then
    0
  else
    ifsa->at(i) = self then
      i
    else
      self.posAux(sa, i + 1) + 1
    endif
  endif
```

4. El método `subsequents()` devuelve el conjunto de subejecuciones posteriores a una subejecución dada en una ejecución compuesta, es decir, las subejecuciones que se ejecutan tras ella.

```
context uml.compositeActions.instance.concepts.compositeExecution
subsequents(): Seq(Execution)

  self.groupExecution.subexecution->
  subSequence(self.pos(), self.subexecution->size())
```

6.1. Acción Secuencial

Una instancia de la clase `Sequential Action` es un envoltorio de una secuencia de subacciones. Estas subacciones tienen que ejecutarse en el orden en el que están incluidas en la secuencia. Esto implica que una acción no puede usar como entrada una salida de una acción posterior.

Las ejecuciones de las subacciones también tienen que ejecutarse secuencialmente, por tanto, una ejecución, o un cálculo de un valor para esa ejecución, no puede acceder a un valor modificado por una ejecución previa.

Reglas de corrección

1. Las ejecuciones de las subacciones de una ejecución secuencial ocurren en el mismo orden en el que están definidas en la correspondiente acción secuencial.

```
context uml.compositeActions.instance.concepts.sequentialAction
inv:

  self.subaction->forall(a | self.subaction->forall(a2 |
       a.pos() <a2.pos() implies a.of.pos() <a2.of.pos()))
```

Métodos

1. Este método devuelve las subacciones de una acción secuencial son sus subacciones.

```
context uml.compositeActions.model.concepts.sequentialAction
subexecutions(): Seq(Action)

  self.subaction
```

2. Este método devuelve la posición de una acción en una acción secuencial.

```
context uml.compositeActions.model.concepts.sequentialAction
pos(): Integer

  self.posAux(self.groupAction.subaction, 1)
```

3. Método auxiliar para definir el método`pos`.

```
context uml.compositeActions.model.concepts.sequentialAction
posAux(sa : Seq(Actions), i : Integer): Integer
```

```
ifi >sa->size() then
  0
else
  ifsa->at(i) = self then
    i
  else
    self.posAux(sa, i + 1) + 1
  endif
endif
```

4. Este método devuelve el conjunto de las subacciones posteriores a una acción dada.

```
context uml.compositeActions.model.concepts.sequentialAction
subsequents(): Seq(Action)

  self.groupAction.subaction->
  subSequence(self.pos(), self.subaction->size())
```

6.2. Acción Paralela

En la ejecución de una acción paralela, las subejecuciones no tienen que seguir el orden en el que están definidas las subacciones en la acción paralela. Las acciones pueden ejecutarse concurrentemente y un mismo valor puede ser accedido para calcular valores para diferentes ejecuciones paralelas, pero sólo un valor dado puede ser modificado por una sola ejecución. Este modelo corresponde a una ejecución entrelazada de una acción paralela.

Métodos

1. Las subacciones de una acción paralela son sus subacciones.

```
context uml.compositeActions.model.concepts.parallelAction
subactions(): Set(Action)
```

```
self.subaction
```

7. Ejemplo de ejecución

Esta sección muestra la ejecución, en un ejemplo simple, de varias acciones sobre un objeto. Se define el método `child` para la clase `Dog`. Este método crea un nuevo objeto de la clase, inicializa sus atributos y lo asigna como el nuevo hijo del objeto que lo ha creado.

```
Perro::nacimiento()
{
    Perro p;

    p = nuevo Perro();
    par{
       p.edad = 0;
       p.pelo = corto;
    };
    hijo = p;
}
```

Este método consiste en un acción secuencial compuesta de tres subacciones: la creación de un nuevo objeto, la actualización de sus *slots*, y la asignación como nuevo hijo del padre. La actualización de los *slots* se hace en paralelo.

Los objetos de la clase `Perro` tienen dos atributos, uno que indica su edad y otro que indica la longitud de su pelo. Cuando se crea el objeto, el valor asociado a sus slots no es un valor válido, ya que la acción `CreateObjectAction`

no actualiza los valores de los *slots* del objeto creado. Para darles el valor adecuado hay que ejecutar sendas acciones de la clase `WriteSlotActions`.

En las figuras 2.12 y 2.13 se muestran la especificación en términos de la semántica de acciones propuesta anteriormente. Por claridad no se muestran el resto de los *slots* del perro padre.

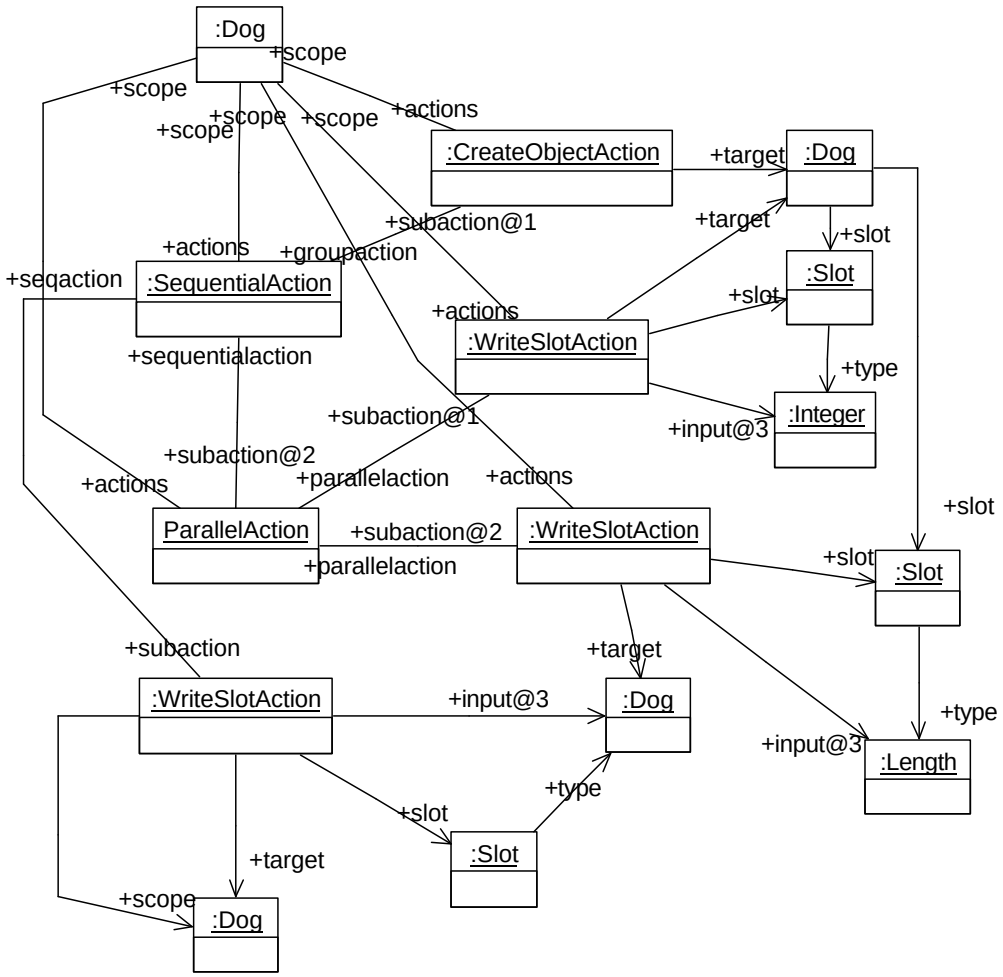

Figura 2.12: Vista del modelo de la ejecución del método

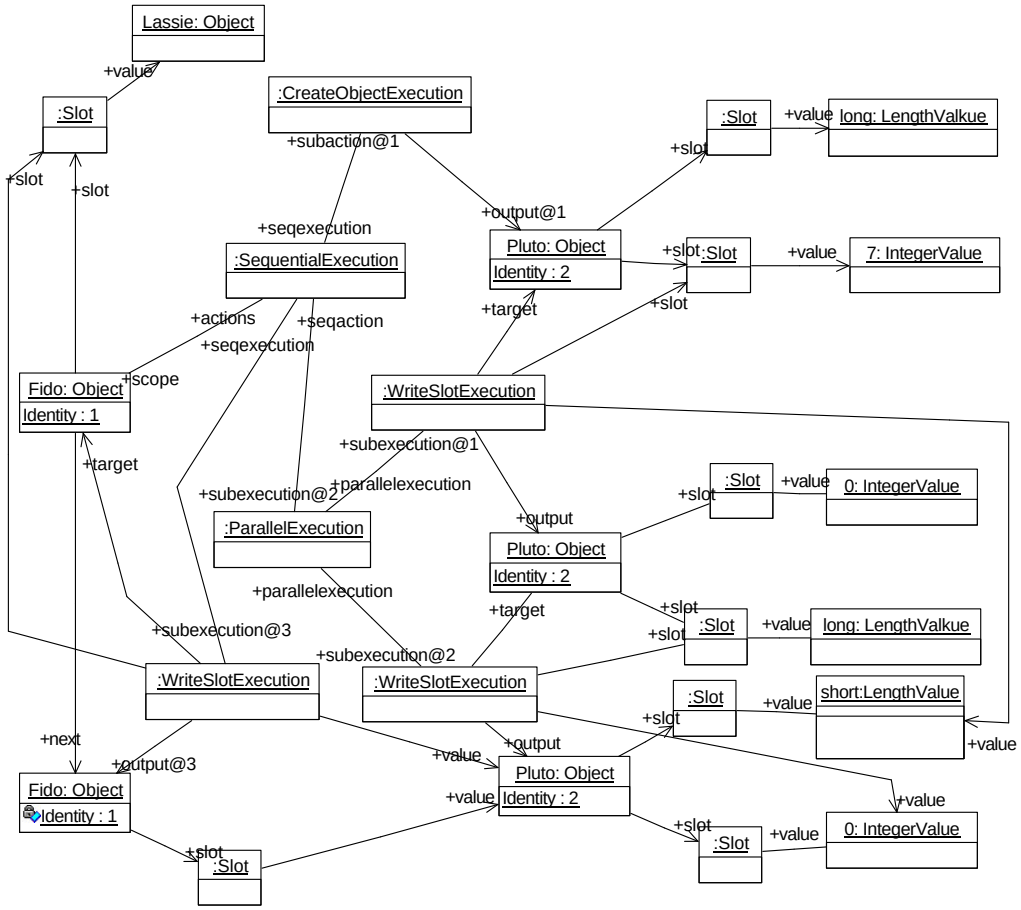

Figura 2.13: Vista de las instancias de la ejecución del método

8. La semántica de acciones en el ámbito de UML 2.0

La anterior semántica de acciones de propuso cuando la versión de trabajo de UML era la 1.4. Posteriormente, han aparecido las versiones 1.5 y 2.0.

En la versión 1.4 no aparece incluida la semántica de acciones en el manual de referencia sino que es un documento aparte, ([4]). En la versión 1.5 la semántica de las acciones ya aparece como un capítulo más del manual de referencia (capítulo 2, Semántica, parte 5, acciones), aunque el contenido de

este capítulo era exactamente igual que el del apéndice incluido en la versión anterior.

8.1. La propuesta adoptada finalmente para UML2.0

La versión 2.0 de UML, ([118] y [119]), sí supone un cambio cualitativo importante respecto a las versiones 1.x, como se conocen habitualmente. En la petición de propuestas se establece una larga lista de requisitos que tienen que cumplir las propuestas para la nueva norma. Uno de los requisitos fundamentales establece la separación en el metamodelo de UML entre los elementos básicos, el núcleo, y los elementos estructurales que dependen de este núcleo, por lo que las propuestas se han dividido en infraestructura y superestructura, respectivamente.

En la especificación adoptada finalmente no se hace referencia a las acciones en la infraestructura, sino que son postergadas a la superestructura. En esta versión se reestructuran los conceptos de acción y actividad, que ya estaban presentes en la versión 1.5. En la versión 2.0, las actividades son grafos de funcionamiento en los que los nodos en los que realmente se lleva a cabo la ejecución son las acciones.

La definición de las acciones es muy parecida en lo fundamental a lo que ya había. Aparecen dos acciones nuevas, `AcceptEventAction` y `SendObjectAction` y las acciones sobre atributos se generalizan a acciones sobre características estructurales (`AddStructuralFeatureValueAction`, `ClearStructuralFeatureAction`, `ReadStructuralFeatureAction` y `RemoveStructuralFeatureValueAction`). Desaparecen las acciones `ActionState`, `CallState` y `SubactivityState` por la nueva reestructuración en la que las acciones son parte de las actividades.

No obstante, la semántica de las acciones es similar al anterior con los mismos problemas que se mencionaron al principio del capítulo (1), lo que si-

gue haciendo válido dicho análisis y la justificación para la semántica descrita en este capítulo.

8.2. La propuesta enviada por 2U Consortium

Como se comentó al principio del capítulo (2), el grupo *2U Consortium* ha enviado su propia propuesta de definición de UML2.0 ([121], [122]). En ella, las actividades y las acciones se definen en una jerarquía similar a la de la propuesta aceptada, definiendo las actividades como un grafo de elementos de funcionamiento en el que las acciones los nodos que realmente ejecutan cálculos y modificaciones del estado del sistema.

Hay diferencias fundamentales, no obstante, entre ambas propuestas. En ésta, las acciones se definen como parte de la infraestructura y las actividades como parte de la superestructura. Como en la propuesta para la definición de UML 1.5, [4], la actual propuesta se basa en el uso de *MML (el Lenguaje de MetaModelado)*. Este lenguaje contiene elementos de metamodelado ya presentes en MOF ([84]): clases, atributos, operaciones de consulta, asociaciones, paquetes y un lenguaje de restricciones, OCL ([89]). Los otros dos elementos de MML son propios del lenguaje, la extensión de paquetes, que se usa en lugar de la importación y las plantillas de paquetes. Con estos dos elementos se pretende conseguir una especificación más compacta y coherente mediante la reutilización sistemática de patrones que se encuentran abundantemente repetidos en la definición del lenguaje, como la relación entre contenedor y elementos contenidos o entre un elemento generalizable y los elementos que lo especializan.

Como también ocurría en la propuesta para UML 1.5, se mantienen las separaciones básicas entre los elementos del lenguaje. La primera entre la sintaxis abstracta de un paquete y su dominio semántico, con la semántica

definida como la relación entre los elementos de ambos paquetes. La segunda entre los elementos de modelado y su representación, la sintaxis concreta. Para el análisis de la semántica de las acciones en esta nueva propuesta nos podemos centrar en los paquetes finales que resultan de la extensión e instanciación de los paquetes básicos y las plantillas de paquetes. Estos paquetes finales tienen un aspecto muy similar al de los paquetes de la anterior propuesta.

8.3. El paquete *Behaviour*

En el grafo de dependencias entre paquetes de la infraestructura, el paquete de las acciones, *Actions*, extiende directamente dos paquetes, *Behaviour* y *Expressions*. El paquete *Behaviour* representa la evolución de los estados del sistema a lo largo del tiempo. Su sintaxis abstracta es similar a la del paquete *Packages*: un paquete puede contener clases y subpaquetes.

En el dominio semántico aparece el concepto de *State*, que representa el estado, en un instante dado, de un elemento del sistema. Como subclases de *State* se definen *Snapshot*, *Object* y *Slot*, que corresponden con el estado de una instancia de un *Paquete*, una *Class* o un *Attribute*, respectivamente.

Otro concepto nuevo es el de *identidad*. Cada elemento del sistema tiene asociada una identidad y ésta permanece fija a lo largo de la evolución del sistema en los diferentes cambios de estado. La identidad tiene una asociación, en concreto, una secuencia ordenada de estados, llamada *filmstrip*, que permite seguir la evolución de ese elemento en el sistema.

Con estas clases y las operaciones definidas sobre ellas se puede saber, por ejemplo, si un estado de un elemento es posterior a otro o no.

8.4. El paquete *Actions*

Las acciones son cálculos que modifican el estado del sistema. Se usan en el cuerpo de las operaciones. El conjunto de acciones de esta propuesta es mucho más reducido que el conjunto de la propuesta finalmente aceptada. Sólo se definen, por un lado, dos acciones primitivas, *PrimitiveAction*: una que actualiza el valor de un atributo de un objeto, *WriteAttributeAction* y otra que crea una nuevo objeto de una clase *CreateObjectAction*. Por el otro lado, se definen dos acciones compuestas *CompositeAction*: una acción secuencial, *SequentialAction* y otra paralela, *ParallelAction*.

La clase *Action* cuenta con un atributo, *isPreemptable*, que indica si la evaluación de la acción en cuestión puede ser interrumpida. Para interrumpir una acción tiene que producirse la evaluación de un flujo, de la clase *Flow*, que está asociado con las acciones compuestas. El ámbito de una acción es el conjunto de variables accesibles desde dicha acción. Como la clase *Action* es una especialización de la clase *Expression*, las acciones tienen un valor de vuelta, que depende del tipo de acción que sea. Por ejemplo, la creación de un objeto tiene como resultado de vuelta dicho objeto.

Como en la semántica propuesta en la sección 2, el dominio semántico de las acciones son las evaluaciones de las mismas (*Evaluation*. Una evaluación tiene asociado un estado previo, sobre el que se aplica, y un estado posterior, que es al que se llega tras la ejecución.

Ventajas e inconvenientes de la nueva semántica de acciones

La semántica de acciones presentada en 2 es, como se puede comprobar, bastante similar a la que hemos ofrecido en este capítulo. No obstante, ofrece un serie de ventajas y resuelve algunas de las lagunas de mi propuesta. En primer lugar, aunque se trata de una característica que abarca a toda

la definición de UML, podemos citar el uso de las extensiones y los patrones de paquetes, que hace la definición global más compacta y coherente. También hay que apuntar la mejor definición de los conceptos que involucra el funcionamiento dinámico del sistema. Se han definido nuevas clases, como *Behaviour* y se ha redefinido otras como *Activity*, consiguiendo una mejor integración de los diferentes conceptos relacionados en la descripción dinámica del sistema. La definición del concepto de estado, que incluye *Snapshot*, *Object* y *Slot* y su diferenciación respecto a la identidad, que permanece fija, también aclara el dominio semántico. Las operaciones definidas sobre la clase *State* permiten definir mejor las reglas de corrección que implican dependencias causales (o temporales, como las definen los autores) entre las evaluaciones de las acciones y las expresiones asociadas.

En la nueva propuesta de la semántica de acciones se incluyen reglas de corrección que limitan el orden de evaluación entre una acción compuesta y sus subacciones, indicando que el estado previo de la evaluación de la acción compuesta debe coincidir con el estado previo de la evaluación de alguna subacción e igualmente para el estado posterior. Asimismo, las evaluaciones de las subacciones de una acción secuencial son ordenadas usando la relación entre sus estados previos y posteriores.

A pesar de todo, considero que aún hay aspectos que no han sido satisfactoriamente resueltos. Para ilustrar estas situaciones tomemos como ejemplo una acción compuesta, secuencial, por ejemplo, en la que todas sus subacciones son acciones de escritura de atributos. Las evaluaciones de las acciones compuestas y la acción de crear un nuevo objeto tienen como estados previos y posteriores una instancia de la clase *Snapshot*. La evaluación de una acción de escritura del valor de un atributo tiene como estados previo y posterior una instancia de la clase *Slot*. Al ser los estados previos y posteriores distintas

subclases de la clase *State* no es posible comparar si son el mismo[1], o si van antes o después.

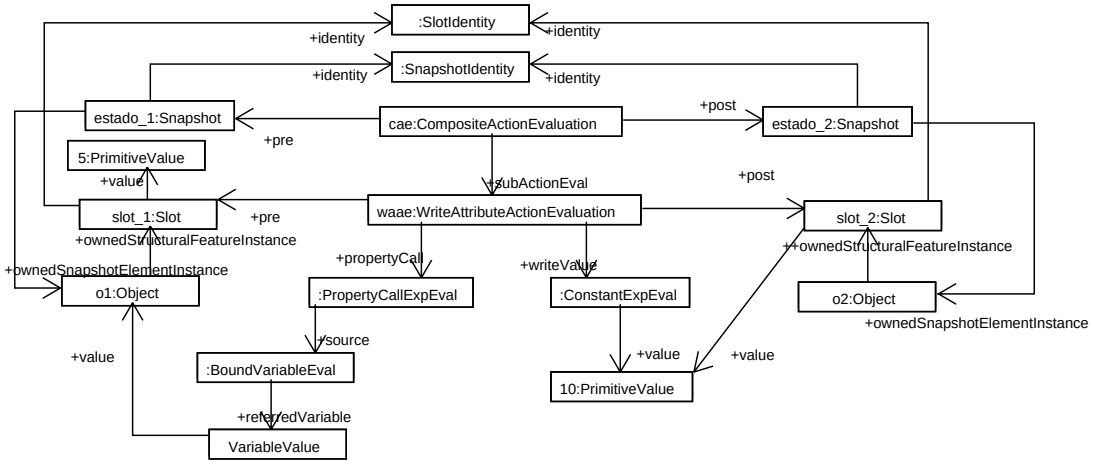

Figura 2.14: Evaluación en la que no coinciden ni el estado previo ni el posterior de una acción compuesta y una de sus subacciones

Otro cuestión muy relacionada que no queda clara es la relación de inclusión entre las diferentes clases de estados. Un *Snapshot* puede contener *Snapshots* u *Objetos* y éstos, a su vez, pueden contener *Slots*. Además, cada estado puede estar contenido directamente, a lo sumo, en un solo estado dentro de esta jerarquía. Por tanto, si tenemos varias *WriteAttributeActionEvaluation* consecutivas, según los diagramas de clases y las reglas de corrección no tenemos ni un nuevo *Snapshot* ni un nuevo *Object* como estado posterior para cada uno de los *Slot* intermedios que se van generando

Otra inconsistencia que se puede presentar afecta a la causalidad entre la

[1]En la propuesta hay varios errores. Uno de ellos es que se hace uso de la operación *isSameTime* sobre dos estados, que no ha sido definida previamente. En cambio, sí se ha definido la operación *isSame*, que simplemente comprueba si son el mismo objeto. El otro es que la figura que muestra el ejemplo de una evaluación de una *WriteAttributeActionEvaluation* el estado previo se define de la clase *Object*, no de la clase *Slot*, como se define en el diagrama de clases del dominio semántico.

evaluación de las expresiones asociadas a la evaluación de una acción. Cuando se evalúa una acción, los valores necesarios se calculan mediante la evaluación de expresiones. En esta semántica no se hace ninguna referencia a cómo deben estar relacionadas ambas evaluaciones. La restricción fundamental es que un valor usado en la evaluación de una expresión no puede tomarse de un estado posterior a la evaluación de la acción.

Una posible solución a estas discordancias es establecer que todas las acciones tienen como estados previo y posterior un *Snapshot* y añadir las reglas de corrección necesarias. En el caso de la escritura del valor de un atributo, la clase y el atributo sobre los que se aplica la acción se accede a través de la expresión *PropertyCallExpression* a través, respectivamente, de sus asociaciones *source* y *propertyCall*. Habría que añadir una regla de corrección que indique que el estado posterior tiene que haber un objeto con un *slot* cuyo valor coincida con el calculado en la expresión *writeValue*.

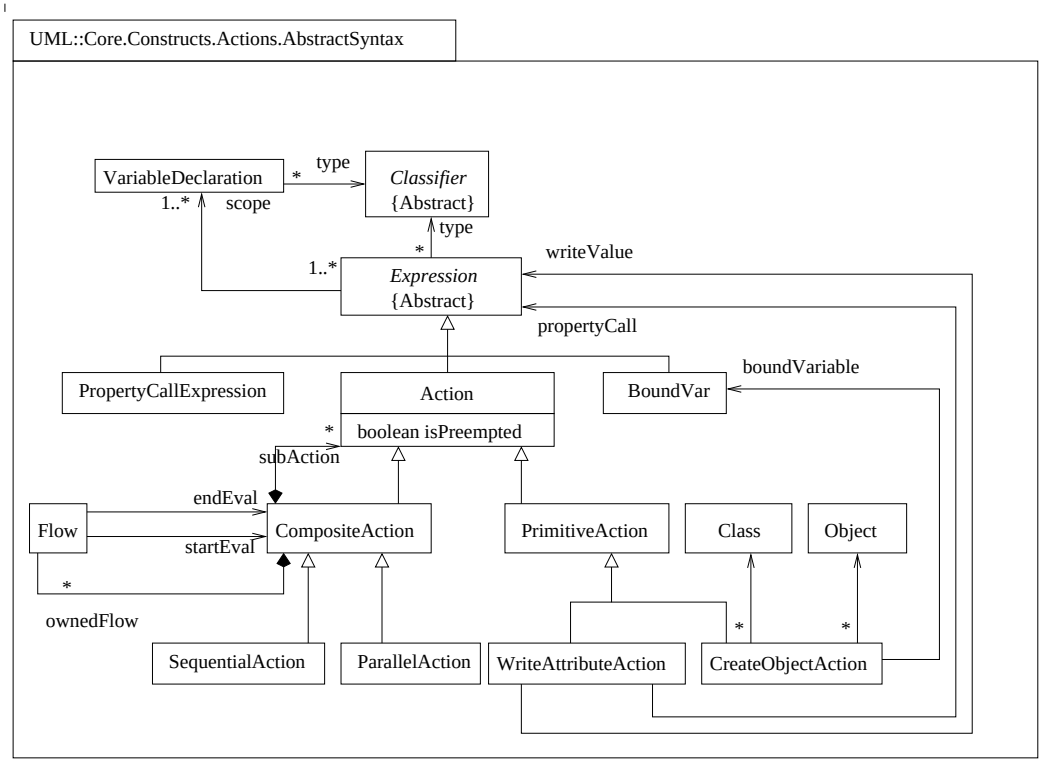

Figura 2.15: Sintaxis abstracta del paquete *Actions* donde los estados previos y posteriores de todas las acciones son de la clase *Snapshot*

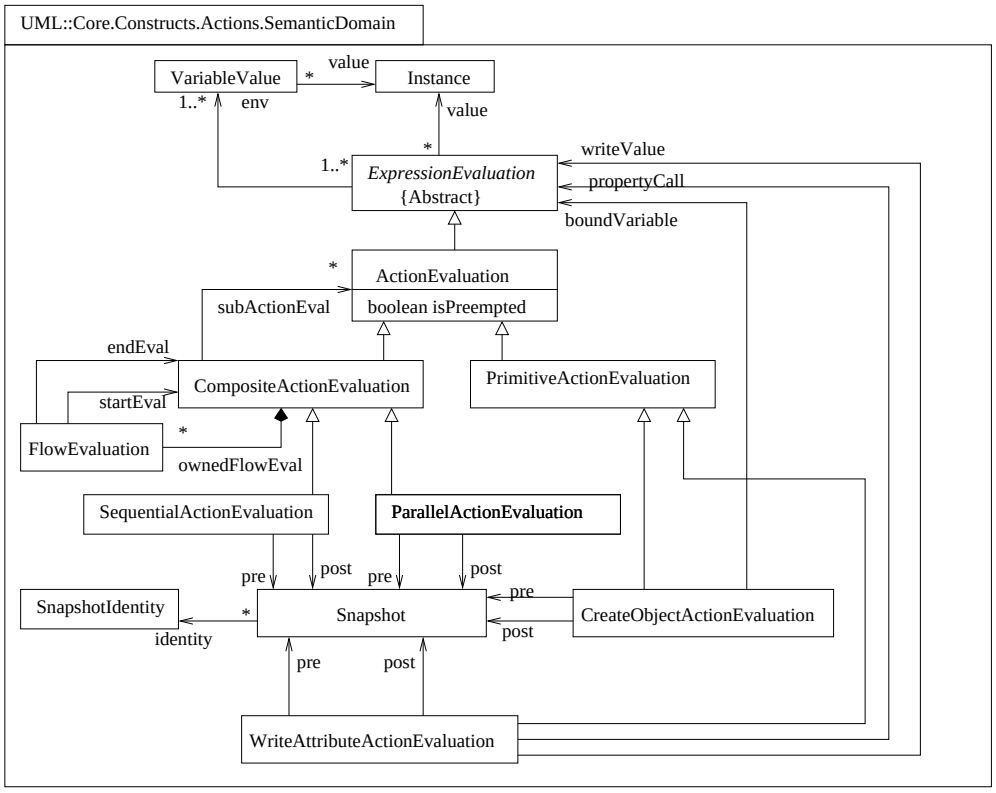

Figura 2.16: Dominio semántico del paquete *Actions* donde los estados previos y posteriores de todas las acciones son de la clase *Snapshot*

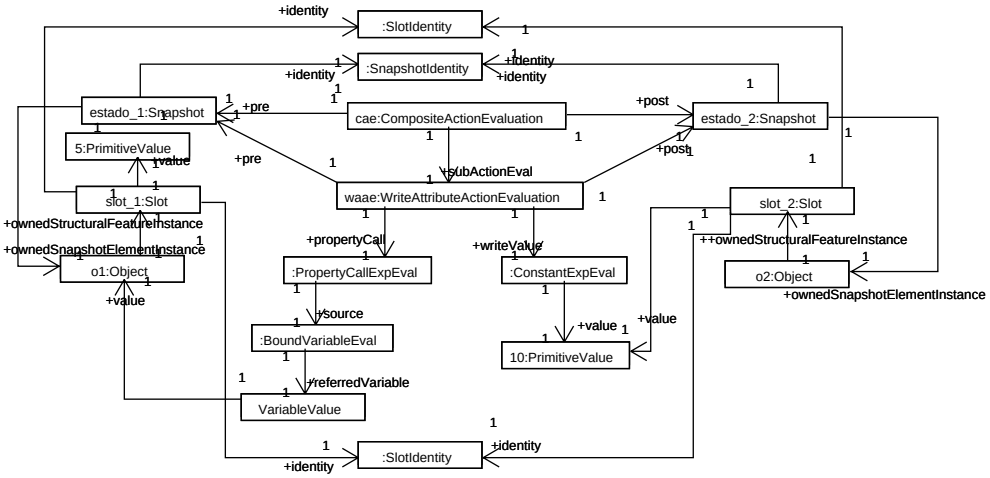

Figura 2.17: Evaluación en la que los estados previos y posterior de una *WriteAttributeActionEvaluation* son de la clase *Snapshot*.

Reglas de corrección

Snapshot

1. Las instancias de la clase *Object* contenidas en una instancia de la clase *Snapshot* son estados de objetos diferentes

```
context Snaphot inv:

  self.ownedSnapshotElementInstance->forall(oei, oei2 |
  oei <>oei2 implies <>oei.identity = oei2.identity
```

Object

1. Las instancias de la clase *Slot* contenidas en una instancia de la clase *Object* son estados de *slots* diferentes

```
context Object inv:

  self.ownedStructuralFeatureInstance->forall(osfi, osf2 |
  osfi <>osfi2 implies osfi.identity <>osfi2.identity
```

WriteAttributeActionEvaluation

1. En el estado posterior debe haber un objeto tal que uno de sus *slots* tiene el valor calculado en la evaluación de la acción.

```
context WriteAttributeActionEvaluation inv:

  post.ownedSnapshot.ElementInstance->exists(o | o.identity
  = self.propertyCall.source.referredVariable.value.identity
  and o.ownedStructuralFeatureInstance->exists(s |
  self.propertyCall.source.referredVariable.value.
  ownedStructuralFeatureInstance->exists(s2 |
  s2.identity = s.identity) and s.value = s.writeValue.value
```

Respecto a la evaluación de las expresiones se puede incluir una nueva regla de corrección que establezca que el estado del que la evaluación de la expresión obtiene sus valores coincide con el estado previo de la evaluación de la acción a la que está asociada.

Reglas de corrección

CompositeActionEvaluation

1. Para las evaluaciones de las subacciones directas de una acción compuesta, todos los valores de las expresiones asociadas pertenecen al estado previo de la acción.

```
context CompositeActionEvaluation inv:

  self.subActionEval.forall(sae | sae.ExpressionEvaluations()->
       forall(ee | ee.allSubExpressionEvaluations()->
             forall(see | see.ReferredInstances()->
                   forall(ri | ri.owningSnapshot = sae.pre)))
```

Métodos

AndExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *And* anidadas a cualquier nivel.

```
context AndExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.left->union(self.right)->
  union(self.left.allSubExpressionEvaluations())->
  union(self.right.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *And*.

```
context AndExpEval::
ReferredInstances():Set(Instances)

  self.left.ReferredInstances()->
  union(self.right.ReferredInstances())
```

BoundVariableEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *BoundVariable* anidadas a cualquier nivel.

```
context BoundVariableEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  Set{}
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *BoundVariableEval*.

```
context BoundVariableEval::
```

```
ReferredInstances():Set(Instances)

  self.referredVariable.value
```

EqualsExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *Equals* anidadas a cualquier nivel.

```
context EqualsExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.left->union(self.right)->
  union(self.left.allSubExpressionEvaluations())->
  union(self.right.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *Equals*.

```
context EqualsExpEval::
ReferredInstances():Set(Instances)

  self.left.ReferredInstances()->
  union(self.right.ReferredInstances())
```

GreaterThanExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *GreaterThan* anidadas a cualquier nivel.

```
context GreaterThanExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.left->union(self.right)->
  union(self.left.allSubExpressionEvaluations())->
  union(self.right.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *GreaterThan*.

```
context GreaterThanExpEval::
ReferredInstances():Set(Instances)

  self.left.ReferredInstances()->
  union(self.right.ReferredInstances())
```

IfExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *If* anidadas a cualquier nivel.

```
context IfExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.condition->union(self.thenExpression)->
  self.condition->union(self.elseExpression)->
  union(self.condition.allSubExpressionEvaluations())->
  union(self.thenExpression.allSubExpressionEvaluations())->
  union(self.elseExpression.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *If*.

```
context IfExpEval::
ReferredInstances():Set(Instances)

  self.condition.ReferredInstances()->
  self.thenExpression.ReferredInstances()->
  self.elseExpression.ReferredInstances()->
```

IncludesExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *Includes* anidadas a cualquier nivel.

```
context IncludesExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.source->union(self.condition)->
  union(self.source.allSubExpressionEvaluations())->
  union(self.condition.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *Includes*.

```
context IncludesExpEval::
ReferredInstances():Set(Instances)

  self.condition.ReferredInstances()->
  union(self.source.ReferredInstances())
```

IterateExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *Iterate* anidadas a cualquier nivel.

```
context IterateExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.source->union(self.body)->
  union(self.source.allSubExpressionEvaluations())->
  union(self.body.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *Iterate*.

```
context IterateExpEval::
ReferredInstances():Set(Instances)

  self.source.ReferredInstances()->
  union(self.body.ReferredInstances())
```

NotExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *Not* anidadas a cualquier nivel.

```
context NotExpEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.operand->
  union(self.operand.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *Not*.

```
context NotExpEval::
ReferredInstances():Set(Instances)

  self.operand.ReferredInstances()
```

PropertyCallExpEval

1. Esta operación devuelve las evaluaciones de todas las subexpresiones de la evaluación de una expresión *PropertyCall* anidadas a cualquier nivel.

```
context PropertyCallEval::
allSubExpressionEvaluations():Set(ExpressionEvaluation)

  self.source->union(self.referredProperty)->
  union(self.source.allSubExpressionEvaluations())->
  union(self.referredProperty.allSubExpressionEvaluations())
```

2. Esta operación devuelve todos los valores accedidos por la evaluación de una expresión *PropertyCall*.

```
context NotExpEval::
```

```
ReferredInstances():Set(Instances)

  self.referredProperty.value
```

CompositeActionEvaluation

1. Esta operación devuelve las evaluaciones de expresiones asociadas a la evaluación de una *CompositeActionEvaluation*.

```
context CompositeActionEvaluation::
ExpressionEvaluations():Set(ExpresionEvaluation)

  Set{}
```

CreateObjectActionEvaluation

1. Esta operación devuelve las evaluaciones de expresiones asociadas a la evaluación de una *CreateObjectActionEvaluation*.

```
context CreateObjectActionEvaluation::
ExpressionEvaluations(): Set(ExpresionEvaluation)

  self.boundVariable
```

WriteAttributeActionEvaluation

1. Esta operación devuelve las evaluaciones de expresiones asociadas a la evaluación de una *WriteAttributeActionEvaluation*.

```
context WriteAttributeActionEvaluation::
ExpressionEvaluations():Set(ExpresionEvaluation)

  self.propertyCall->union(self.callValue)
```

Esta regla de corrección implica que todas las acciones se evalúan de manera atómica, incluyendo la evaluación de sus expresiones. Esto no es una idea demasiado realista porque el espíritu de UML es precisamente la ejecución concurrente de objetos e, incluso en la misma propuesta, las acciones tienen un atributo que indican si son interrumpibles y las evaluaciones tienen otro que indican si han sido interrumpidas. Tampoco se estaría teniendo en cuenta la posibilidad de la intercalación de la evaluación de las expresiones de distintas acciones.

Podemos permitir la ejecución intercalada de acciones cambiando las reglas de corrección. La regla anterior se puede restringir a la evaluación de las acciones secuenciales (*SequentialActionEvaluation*), mientras que en las evaluaciones de las subacciones de una acción paralela (*ParallelActionEvaluation*), la evaluación intercalada significa que el estado del que toman los valores las expresiones asociadas a la evaluación de la acción no tiene por qué ser necesariamente el estado previo de la acción, sino uno anterior. Más concretamente, puede ser cualquier estado entre el estado previo de la acción paralela que contiene a la acción y el estado previo de la acción.

Reglas de corrección

ParallelActionEvaluation

1. Para las evaluaciones de las subacciones directas de una acción paralela, todos los valores de las expresiones asociadas pertenecen a un estado que está entre el estado previo de la evaluación de la acción paralela y el estado previo de la evaluación de la subacción a la que está asociada la evaluación de la expresión.

   ```
   context ParallelActionEvaluation inv:
   ```

```
self.subActionEval.forall(sae | sae.ExpressionEvaluations()->
forall(ee | ee.allSubExpressionEvaluations()->forall(see |
forall(see | see.ReferredInstances()->forall(ri |
(self.OuterParActionEval().pre.isEarlier(ri.owningSnapshot) or
self.OuterParActionEval().pre.isSame(ri.owningSnapshot)) and
(ri.owningSnapshot.isEarlier(sae.pre) or
ri.owningSnapshot.isSame(sae.pre)))))
```

2. Para las evaluaciones de las subacciones directas de una acción secuencial, todos los valores de las expresiones asociadas pertenecen al estado previo de la acción.

```
context CompositeActionEvaluation inv:

  self.subActionEval.forall(sae | sae.ExpressionEvaluations()->
      forall(ee | ee.allSubExpressionEvaluations()->
            forall(see | see.ReferredInstances()->
                  forall(ri | ri.owningSnapshot = sae.pre)))
```

Métodos

ActionEvaluation

1. Este método devuelve la acción paralela más exterior que contiene a una acción dada. Si la acción está incluida en una acción secuencial, el valor devuelto es la propia acción

```
context ActionEvaluation:

OuterParActionEval():ActionEvaluation

  if self.CompositeActionEvaluation->size() = 0 or
     self.CompositeActionEvaluation.of = SequentialAction then
     self
  else
     self.CompositeActionEvaluation.OuterParEvaluation()
  endif
```

Esta nueva de regla de corrección aumenta las posibilidades de ejecución permitiendo la intercalación en la evaluación de las acciones. No obstante, sigue sin contemplar la ejecución concurrente de dos acciones. Cada *Snapshot* sólo puede ser el estado previo de una evaluación, lo que impide que varias acciones se evalúen simultáneamente sobre el mismo estado y desemboquen en un mismo estado posterior. En este caso, puede ocurrir que dos acciones modifiquen concurrentemente el mismo valor y, por tanto, en el estado posterior el valor final del elemento modificado no coincidiría con el del valor evaluado en la expresión asociada.

Por tanto, para permitir la ejecución concurrente de varias acciones, habría que cambiar la cardinalidad de la asociación entre la clase *State* y las evaluaciones de las diferentes acciones para que cada estado pudiera ser el estado previo o el estado posterior de una acción o más y cada acción pudiera tener un solo estado previo y un solo estado posterior.

CAPÍTULO 3

Modelado de tiempo real de las máquinas de estados de UML

1. Introducción

Entre las principales contribuciones de UML (*Unified Modeling Language*) [88] está la unificación de varias notaciones previas que compartían los conceptos fundamentales, aunque diferían ligeramente en el modo en que las trataban. También hay que hacer notar que UML ofrece un conjunto de diagramas suficientemente amplio para permitir al desarrollador cubrir las diferentes partes o vistas de un sistema dado.

Las máquinas de estados se encuentran entre los denominados elementos de modelado dinámicos, que son los apropiados para describir el funcionamiento dinámico de los sistemas. Estos diagramas especifican el funcionamiento de los diversos elementos mediante un conjunto de estados estables donde los elementos están esperando la ocurrencia de un evento externo que hace que el sistema reaccione y, posiblemente, cambie su estado interno. Consecuentemente, estos diagramas son muy apropiados para describir sistemas reactivos [57].

El documento de especificación de UML 1.5 [88] define en la sección 2.12 la semántica de las máquinas de estados como una variante basada en objetos de los *statecharts* de Harel [57]. Su definición se hace de una manera semiformal. La sintaxis abstracta se define mediante diagramas de clases y la semántica estática se define usando reglas de corrección en OCL, que establecen ciertas propiedades que deben cumplir las instancias válidas de los diagramas de clases. Sin embargo, la semántica dinámica se describe en lenguaje natural, lo que impide la existencia de un modelo formal subyacente que permita a las herramientas de diseño hacer pruebas automáticas del cumplimiento de las propiedades del modelo.

Las máquinas de estados de UML tienen un gran número de diferentes elementos, lo que resulta en una semántica dinámica muy compleja. Por el contrario, algunos de estos elementos, como los estados *stub* y los estados *submáquina*, no añaden nueva semántica de por sí, sino que son construcciones sintácticas que pretenden facilitar el uso de las máquinas de estados.

Una semántica precisa para las máquinas de estados permitiría validar y verificar sistemas descritos en UML como hacen otras herramientas, como Telelogic Tau con sistemas basados en SDL [61]. Si hay una semántica precisa y correcta para las máquinas de estados, una herramienta podría representar el estado actual del sistema, por ejemplo como un diagrama de objetos, y podría comprobar si es una instancia válida del diagrama de clases que representa el sistema. La validación consistiría en generar el árbol de posibles evoluciones del sistema y analizar si los estados cumplen las restricciones establecidas. Estas restricciones se aplicarían a los estados mismos y a las transiciones de un estado a otro, como se comenta en el capítulo de la semántica de acciones (capítulo 2). Estas restricciones podrían expresarse en OCL.

Para la verificación, las herramientas deberían ser también capaces de

representar una traza de la ejecución definida por un diagrama de colaboración o de secuencia. Esto implica otro trabajo a hacer, ya que estos diagramas necesitan su propia semántica.

1.1. Trabajos relacionados

Son numerosos los trabajos que presentan semánticas formales alternativas para las máquinas de estados, demostrando de esa manera que es muy generalizada la opinión de que una semántica formal y precisa conlleva grandes beneficios, como ya hemos comentado. En [35], Crane y Dingel hacen un estudio muy actual y sistemático de diferentes propuestas de semánticas.

Son también muy variados los formalismos usados para construir dichas semánticas. Por un lado existen modelos que usan estructuras más similares a las máquinas de estados, como sistemas de transiciones etiquetadas, estructuras de Kripke, máquinas de estados abstractas, redes de Petri o grafos. La ventaja de estas propuestas es que la similitud entre los formalismos usados y las máquinas de estados de UML hace que la traducción de los conceptos entre ambos modelos sea más fácil.

Otro grupo de soluciones son las que se basan en lenguajes fuertemente matemáticos, como teoría de conjuntos y relaciones o lógicas temporales. Aunque estas soluciones facilitan soluciones completas y con propiedades demostrables, son más difíciles de entender por los usuarios de las máquinas de estados. Otras propuestas se basan en la transformación de las máquinas de estados en objetos de otros lenguajes apropiados para su análisis automáticos, como la comprobación de modelos.

Un tercer grupo de soluciones son aquellas que traducen las máquinas de estados de UML a otros lenguajes, ya sean lenguajes de especificación, como Z o PVS [92], lenguajes de entrada a herramientas de comprobación de

modelos, como PROMELA/SPIN, o lenguajes de programación imperativa, como Java.

En [25] Börger y otros usan máquinas de estados abstractas para representar la máquinas de estados de UML. Su trabajo incluye un estudio en profundidad de los aspectos más complicados del modelo, como el manejo de los eventos o las actividades internas de los estados, ausentes en muchos otros modelos.También hacen explícitos algunos de *puntos de variación semántica* de la norma UML.

En [20] Baresi y Pezzè ilustran a través de un ejemplo la traducción automática de un modelo UML, en el que las máquinas de estados son un parte, a redes de Petri de alto nivel. La sintaxis y la semántica de UML se hace sobre gramáticas de grafos. Una vez traducidas las máquinas de estados a redes de Petri se pueden usar varias herramientas automáticas ya disponibles para realizar análisis de exclusión mutua, ausencia de bloqueos, comprobación de modelos, etc, que se pueden traducir a su vez a notación UML para que el usuario las examine.

En [53] Gogolla y Presicce estudian un mecanismo muy intuitivo de transformación de máquinas de estados de UML a grafos aplanados mediante el uso de reglas de transformación de grafos. En [72], Kuske continúa el trabajo aplicando las técnicas de transformación de grafos para definir una semántica formad de las máquinas de estados, en la que las distintas configuraciones del sistema se representan como grafos y la ejecución de una transición corresponde a la aplicación de reglas de transformación de grafos.

En [18] Aredo proporciona un esquema general para traducir las máquinas de estados de UML a PVS-SL, el lenguaje de especificación del marco PVS. La traducción tiene en cuenta la sintaxis abstracta de las máquinas de estado y las reglas de corrección. Este método cuenta con la ventaja de

disponer de las herramientas automáticas de análisis para PVS, que incluyen comprobación de tipos y modelos y demostración de teoremas.

En [51] Gnesi y otros utilizan autómatas jerárquicos extendidos y estructuras de Kripke para darle un soporte formal a la semántica de las máquinas de estados y hacen uso de la herramienta JACK para hacer comprobación de modelos sobre ACTL, una lógica temporal de tiempo ramificado.

En [70] Knapp y Merz presentan HUGO, una herramienta que permite animar, comprobar modelos y generar código Java a partir de máquinas de estados de UML. La comprobación de modelos de las máquinas de estados se hace frente a diagramas de interacción en vez de fórmulas lógicas para facilitar su uso. HUGO hace uso de PROMELA y SPIN.

En [74] se estudia UML en su conjunto, representando los modelos de UML como teorías de una teoría de conjuntos extendida de primer orden. Las teorías se expresan en *Real-time Action Logic* y las máquinas de estados se formalizan como extensiones de las teorías para clases.

En [100] Rossi y otros usan LNint-e, una lógica temporal basada en puntos, intervalos y fechas para formalizar los aspectos fundamentales de la semántica de las máquinas de estados de UML, haciendo especial énfasis en los aspectos dinámicos. Algunas características como los eventos y las guardas no han sido modelados porque necesitan una lógica de primer orden.

Todas las propuestas anteriores se basan en formalismos matemáticos de los que la OMG, creadora de UML, prefiere no hacer uso por su dificultad de comprensión. Una limitación fundamental en la utilidad de estas propuestas es la extensión de las mismas para la definición de la semántica de los demás diagramas de UML, de manera que se pudiera usar toda la gama de diagramas de UML en el desarrollo de un mismo sistema. Si el formalismo no es adecuado para representar todos los diagramas, que son de muy distinta na-

turaleza, al menos deberían ofrecer mecanismos de traducción a la semántica de los otros diagramas para comprobar la coherencia las diferentes vistas de los modelos. Algunas propuesta, como [18] y [70] ofrecen herramientas para análisis automático de propiedades de las máquinas de estados que modelan, lo que demuestra su utilidad práctica.

En [46] Engels y otros usan el metamodelado para definir la semántica formal. Parten del metamodelo estático, basado en diagramas de clases) y lo extienden al modelo dinámico con diagramas de colaboración, que se usan para expresar la semántica dinámica de las máquinas de estados. Sin embargo, el conjunto de características de las máquinas de estados cubierto es muy reducido.

En [69], Kleppe y Warmer también usan el metamodelado para definir la semántica formal de UML. Este trabajo es, con toda probabilidad, el más similar al nuestro, pues también se basa en el trabajo desarrollado por el grupo pUML [27]. El modelo de máquinas de estados que se representa es muy simple, y sólo incluye estados simples, transiciones y acciones. No define la semántica de la relación entre los diferentes estados por los que pasa una máquina de estados a lo largo de su vida.

Finalmente, queremos destacar el hecho de que ninguna de esas propuestas estudia el tema del tiempo en las máquinas de estados, ni su aplicación a sistemas de tiempo real. Las que hacen mención a prioridades se refieren a las prioridades de las transiciones anidadas cuyo evento disparador coincide, pero no es ese el significado de prioridad necesario en los sistemas de tiempo real.

2. Semántica de las máquinas de estados

Como se discutió en el capítulo 2, la definición oficial de UML no diferencia claramente entre la sintaxis abstracta, o *conceptos de modelado*, y el dominio semántico, o *conceptos de instancia*. Por ejemplo, en la definición de la semántica se dice:

> El término "evento" se usa para referirse al tipo y no a la instancia del tipo. Sin embargo, en alguna ocasión, donde el significado se pueda deducir claramente del contexto, el término también se usará para referirse a una instancia del evento.

Esta falta de distinción también afecta al resto de los conceptos definidos en el paquete de las máquinas de estados y hace que conceptos más relacionados con la vista estática estén mezclados con otros más cercanos a la vista dinámica. Por ejemplo, aunque se habla ampliamente sobre el concepto de transición de un estado a otro, no hay ningún concepto que represente el procesamiento de una instancia de un evento, ni hay reglas de corrección OCL relativas a las transiciones.

En este capítulo proponemos una semántica para una versión simplificada de las máquinas de estados de UML. Esto significa que hay características, como las actividades, los estados de historia o la lista de eventos postergados, que no son contemplados. Para ilustrar la filosofía de la aproximación, se van a ofrecer diferentes modelos, cada vez con más elementos. Pensamos que el modelo finalmente considerado es suficientemente completo y lo bastante representativo de la validez del enfoque usado. El modelo de la sección 2.1 incluye estados simples y transiciones entre estados. El de la sección 2.2 introduce los estados compuestos, que se pueden describir en varios niveles de detalle. En la sección 2.3 se amplía el concepto de estado compuesto

añadiendo los estados concurrentes, donde hay varios subestados activos simultáneamente. Finalmente, en la sección 2.4 se añaden al modelo los eventos que disparan las transiciones y las acciones que se ejecutan en el paso de un estado a otro.

En cada sección el desarrollo del modelo es similar. Se definen los tres subpaquetes habituales en la definición de un paquete en MML. Cada uno de los subpaquetes incluye un diagrama de clases y las reglas de corrección necesarias para detallar los aspectos semánticos que no se pueden definir en el diagrama de clases. Asimismo se incluye un ejemplo para ilustrar los conceptos presentados. El ejemplo consta de una máquina de estados concreta especificada en UML con los elementos incluidos en cada sección y los diagramas de objetos correspondientes a la sintaxis abstracta y el dominio semántico del ejemplo.

Muchos de los conceptos presentados a continuación coinciden con otros definidos en la semántica oficial de las máquinas de estados de UML. En la presente propuesta, el subpaquete de conceptos de modelado contiene los elementos relativos a la descripción estática de la máquina de estados, como los posibles estados de la máquina o cuáles son las transiciones de salida de cada estado. Por otro lado, el subpaquete de conceptos de instancias define los elementos relativos al aspecto dinámico. Entre otros, este paquete incluye el procesamiento de una instancia de un evento y las ejecuciones involucradas en la transición de un estado a otro.

2.1. La máquina de estados plana

El modelo más simple de una máquina de estados consta, por un lado, de un conjunto de situaciones estables, denominadas *estados*, en uno de los cuales la máquina siempre permanece. En este modelo, los estados son indivi-

sibles y no se pueden desarrollar internamente. Debido a esta característica, denominaremos a estas máquinas como *máquinas de estados planas*.

Por otro lado, la máquina de estados comprende un conjunto de *transiciones*, que son enlaces entre los estados descritos previamente. Cada transición tiene un estado *origen* y un estado *destino*. La máquina puede cambiar de estado a partir de su estado actual. No obstante, el cambio no puede cambiar a cualquier otro estado, sino que para cambiar de un estado a otro debe haber una transición que tenga como estado origen al primer estado y como estado final al segundo.

Esta definición permite establecer varios paralelismos entre las instancias de las máquinas de estados y los objetos. Este paralelismo se refleja principalmente en el funcionamiento dinámico del sistema.

Una instancia de una máquina de estados es una entidad que, al igual que un objeto, pasa por diferentes estados según se van ejecutando acciones sobre el sistema. En el caso de las máquinas de estados, cada uno de estos estados por los que va pasando se denomina el *estado activo*. También tiene sentido establecer, como se hace en la definición de la infraestructura de UML 2.0 [121] para los objetos, una identidad para cada instancia de una máquina de estados. Esta identidad permanece fija durante la vida del sistema y actúa como nexo entre los sucesivos estados. Cada uno de estos estados debe pertenecer a un *Snapshot*, al igual que lo hacen las instancias de la clase *Object*, como se propone en [121].

Este paralelismo, no obstante, no es completo. Un objeto tiene características estructurales, los *slots*, que tienen asociados a su vez valores. Cada objeto tiene que estar asociado con una instancia, y sólo una, de cada uno de sus *slots*. Por su lado, cada estado activo de una instancia de una máquina de estados no tiene por qué estar asociado con una instancia de los estados

asociados a la máquina de estados, sino sólo con un subconjunto de ellos. En el caso de la máquina de estados plana sólo uno de los estados estará activo para cada estado de la instancia de la máquina de estados. Los estados de la máquina de estados no tienen asociado ningún valor, como es el caso de los *slots* de los objetos, al menos en este primer modelo simple.

Conceptos de modelado

En la figura 3.1 se muestra el diagrama de clases de los conceptos de modelado de las máquinas de estados. En él vemos cómo una máquina de estados tiene asociados un conjunto de estados y un conjunto de transiciones.

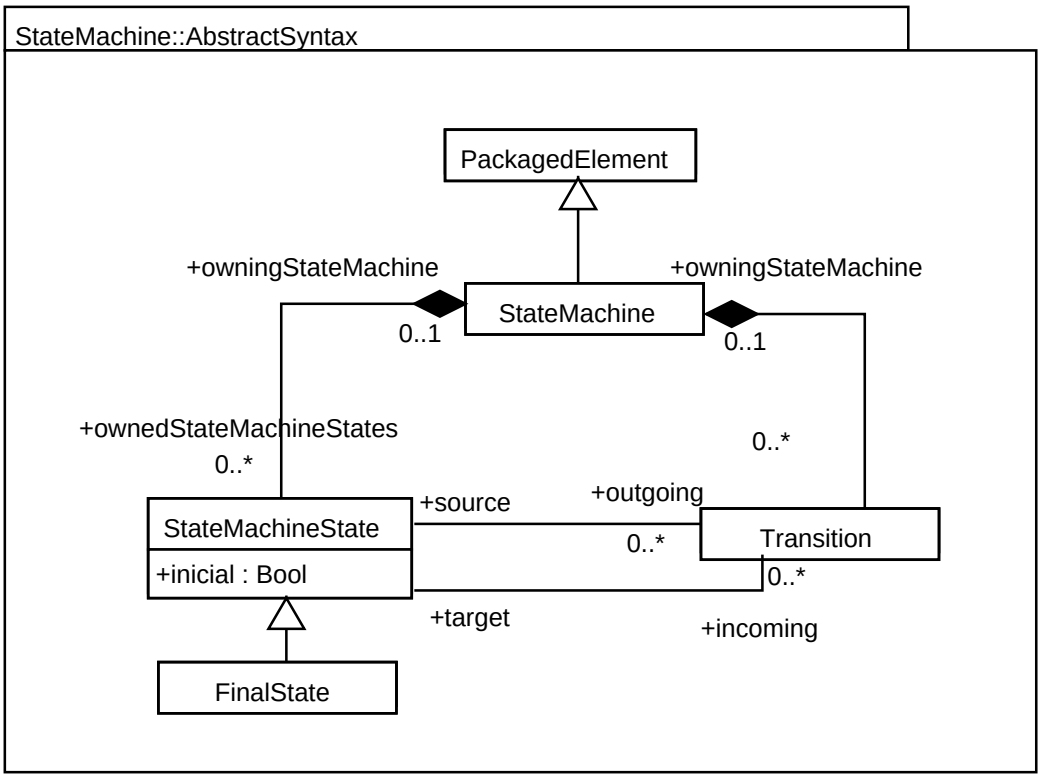

Figura 3.1: Conceptos de modelado de las máquinas de estados planas.

En esta primera aproximación todos los estados son simples, por lo que no se establece un relación de contención entre ellos.

Las transiciones, que definen los posibles cambios de estado en la máquina de estados, parten de un estado origen, *source*, y llegan a un estado destino, *target*.

En las máquinas hay dos estados distinguidos, el *initial*, que es el que se activa cuando empieza la vida de la máquina, y el *final*, que es un estado tal que cuando se transita a él, se acaba la actividad de la máquina. El estado final no puede, por tanto, tener ninguna transición de salida. En la definición de las máquinas de estados de UML, el estado inicial tiene una única transición de salida, que se *ejecuta* cuando empieza la vida de la máquina de estados.

Para no tener que definir una clase especial de estados, como se hace en la definición de las máquinas de estados de UML con los *pseudoestados*, que incluyen estados iniciales, de historia, *fork* y *join*, entre otros, nosotros vamos a añadir un atributo de tipo `Bool` a los estados que indique si es el estado inicial. Vamos a incluir una restricción que indique que sólo puede haber un estado inicial. Los estados finales los definimos como una especialización de los estados simples. En el caso de los estados finales no es necesario establecer ninguna restricción sobre su número ya que, al no poderse efectuar transiciones en ellos, no hay diferencia entre tener uno o varios estados finales.

Reglas de corrección

1. Sólo hay un estado inicial.

```
context StateMachine inv:

  ownedStateMachineStates->select(osms |
       osms.initial = true)->size() = 1
```

2. Los estados finales no pueden ser iniciales.

```
context FinalState inv:

  initial = false
```

3. Los estados finales no tienen transiciones de salida.

```
context FinalState inv:

  outgoing()->size() = 0
```

Dominio semántico

En la figura 3.2 se muestra el diagrama de clases correspondiente al dominio semántico de las máquinas de estados. En él se define una instancia de una máquina de estados como una especialización de la clase `SnapshotElementInstance`, como se describe en el capítulo de *Behaviour* de [121].

Allí, a su vez, `SnapshotElementInstance` se define como una especialización de la clase `State`. La identidad de una instancia de una máquina de estados, representada por `StateMachineInstanceIdentity`, está asociada con los sucesivos estados por los que pasa la instancia. Cada valor de la instancia de la máquina de estados estará asociada con un estado activo, una instancia de la clase `StateMachineStateInstance`. Los estados activos, como los *slots* de un objeto, tienen una identidad constante que no cambia durante la ejecución del sistema.

Aplicación semántica

La semántica de este primer modelo es muy simple, ya que en el dominio semántico sólo hay dos clases que se asocien con otras de la sintaxis abstracta (figura 3.3).

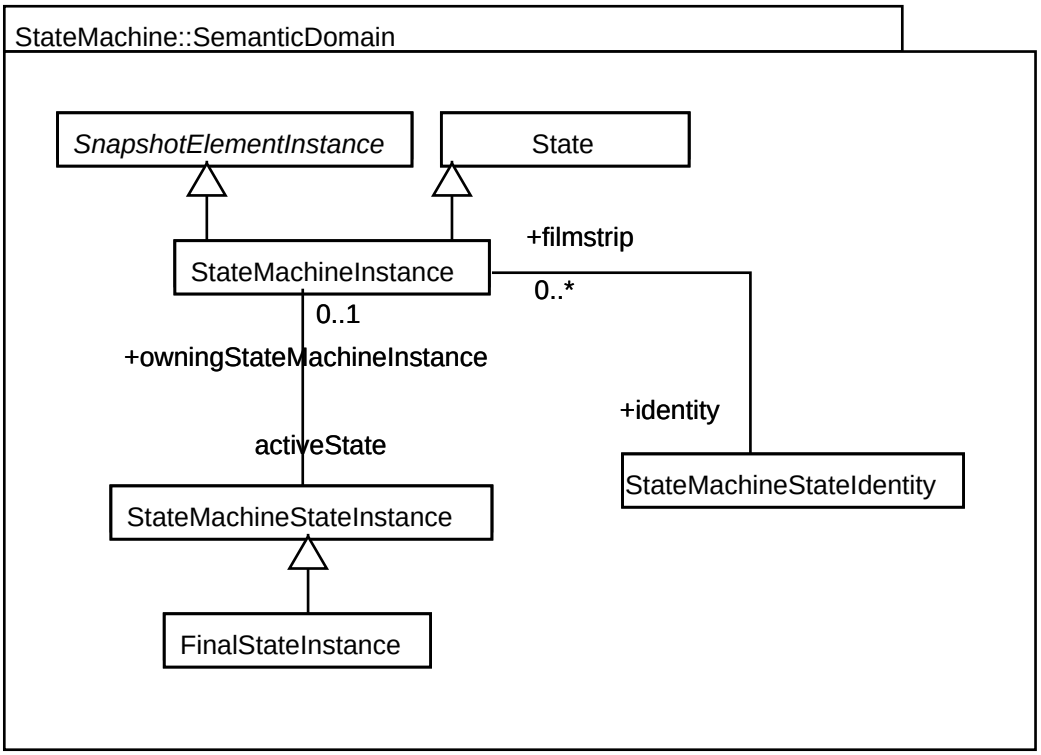

Figura 3.2: Dominio semántico de las máquinas de estados planas.

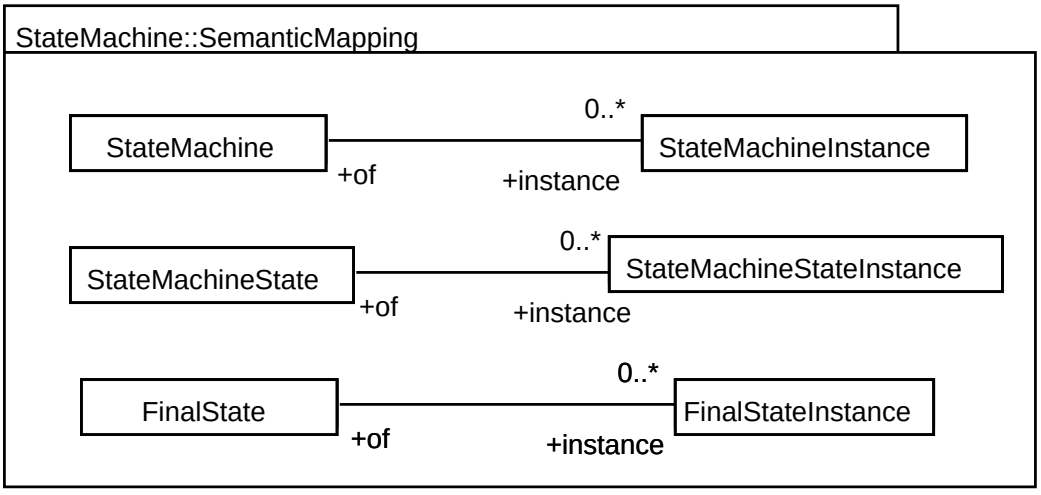

Figura 3.3: Aplicación semántica de las máquinas de estados planas.

Ejemplo

En la figura 3.4 se muestra una máquina con un estado inicial, uno final, tres estados *normales* y tres transiciones entre dicho estados. En la figura 3.5

se muestra el diagrama de objetos correspondiente a la sintaxis abstracta de dicha máquina. En la figura 3.5 aparecen algunos nombres que no aparecen en la figura 3.4, como los de las transiciones (*T_1*, *T_2*, *T_3* y *T_3*) y el del estado final (*Estado_f*). Esta circunstancia se debe a que el modelo UML de la máquina de estados hay elementos que no necesitan nombre, como las transiciones y los estados inicial y finales, entre otros, pero en el metamodelo, las clases que representan estos elementos son especializaciones de otras clases que cuentan entre sus atributos con un nombre, que es el que aparece en la figura 3.5.

En la figura 3.6 se muestra el diagrama de objetos correspondiente al dominio semántico de la máquina de estados del ejemplo de la figura 3.4. El diagrama de objetos representa una posible situación *instantánea* de la máquina de estados. Concretamente, la figura muestra una situación en la que el estado activo es el estado *Estado_1*.

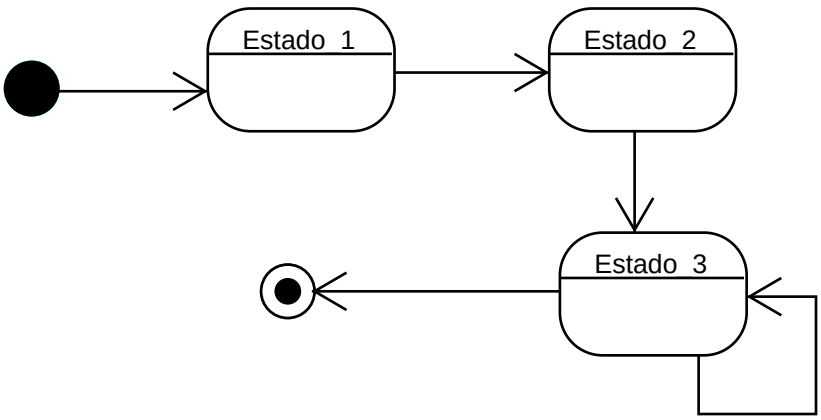

Figura 3.4: Ejemplo de máquina de estados plana.

2.2. La máquina de estados con estados compuestos

David Harel argumenta en [57] que una máquina de estados es un modelo cuya utilidad para el diseño puede mejorarse si el concepto de estado

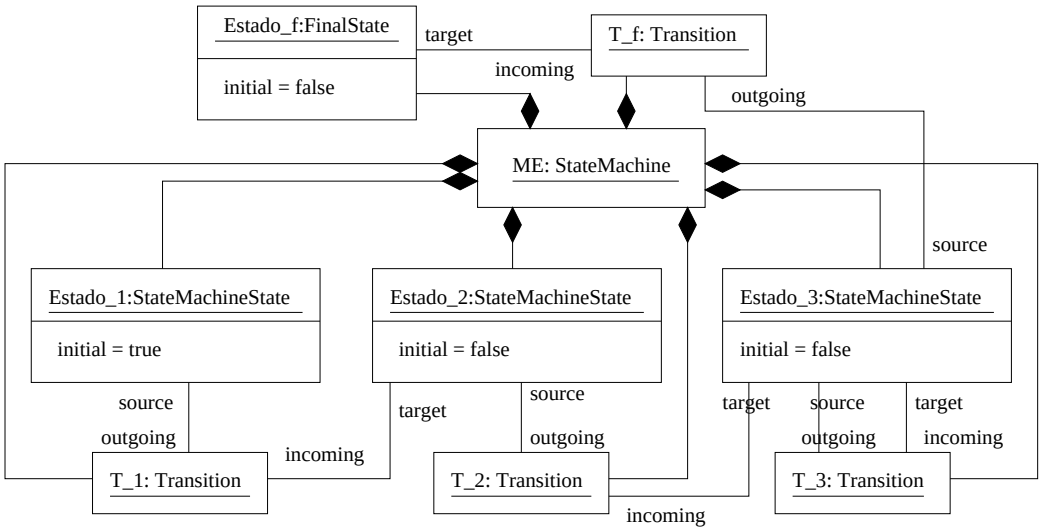

Figura 3.5: Diagrama de objetos de la sintaxis abstracta de la máquina de estados de la figura 3.4.

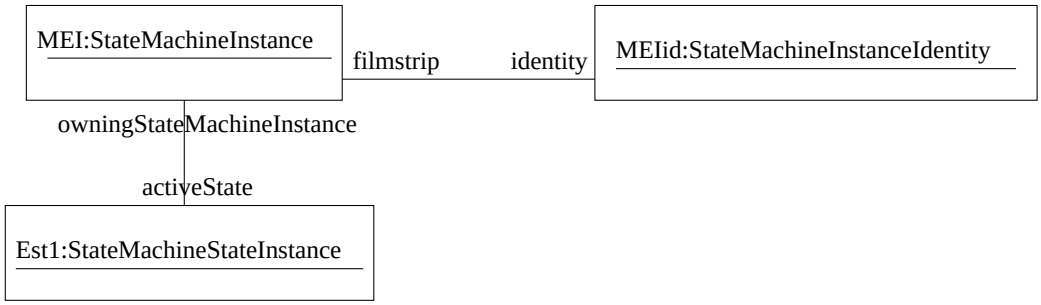

Figura 3.6: Diagrama de objetos del dominio semántico de la máquina de estados de la figura 3.4.

se amplía de tal forma que cada estado puede, a su vez, dividirse en otra máquina de manera jerárquica. Entre las ventajas están un mejor aprovechamiento del área de dibujo y diseños más cercanos al sistema modelado. La desventaja de ese modelo es que su semántica se complica porque aumentan las posibles relaciones entre los estados y las transiciones.

En esta sección vamos a considerar un modelo más completo que el de la sección anterior en el que los estados pueden no ser simples, sino también compuestos. Los estados compuestos (`CompositeState`) están divididos

jerárquicamente en subestados, lo que permite que el funcionamiento de la máquina de estados pueda definirse en un nivel mayor de detalle. Ésta es una relación de composición, por lo que un estado puede estar contenido a lo sumo en un estado compuesto. Esta relación tiene forma de árbol general donde el estado principal es la raíz. Dentro de los subestados hay uno distinguido, el estado inicial, que es el que se activa cuando se entra en el superestado.

La clase de los estados simples (`SimpleState`) representa aquellos estados que no tienen estructura interna, es decir, que no tienen subestados. Estos son los estados que hemos considerado en la sección anterior.

Las transiciones pueden partir de un estado anidado y llegar a otro estado anidado. Se considera que al ejecutar una transición de este tipo se sale de todos los estados que son ancestros del estado origen hasta el estado ancestro común con el estado destino más profundo, y se entra en todos los estados desde el mencionado ancestro común hasta el estado destino.

Conceptos de modelado

En la figura 3.7 se muestra el diagrama de clases de los conceptos de modelado para las máquinas de estados con estados compuestos. Una máquina de estados se compone de sus estados y sus transiciones. Los estados compuestos pueden contener a su vez otros estados.

Reglas de corrección

1. En el nivel más externo de la máquina de estados sólo hay un estado inicial.

```
context StateMachine inv:

  ownedStateMachineStates->select(osms |
       osms.initial = true)->size() = 1
```

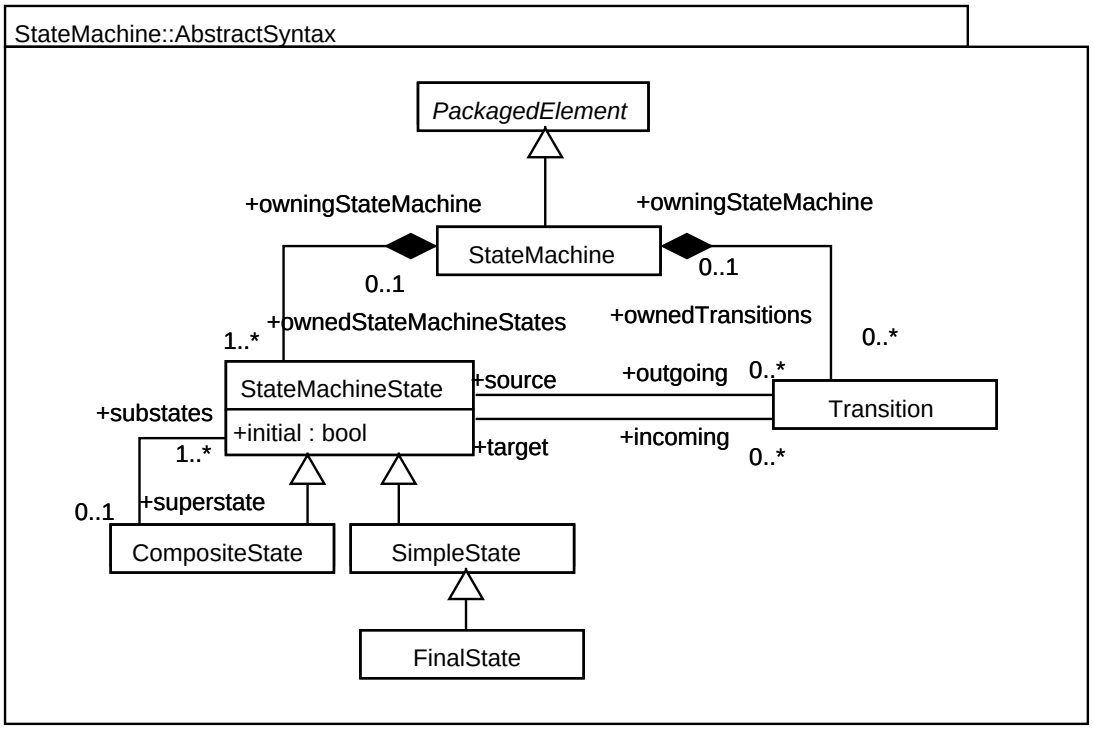

Figura 3.7: Conceptos de modelado de las máquinas de estados con estados compuestos.

2. Un estado compuesto sólo puede tener un subestado directo que sea inicial.

```
context CompositeState inv:

  subStates->select(ss | ss.initial = true)->size() = 1
```

3. Los estados finales no pueden ser iniciales.

```
context FinalState inv:

  initial = false
```

4. Los estados finales no tienen transiciones de salida.

```
context FinalState inv:
```

```
outgoing()->size() = 0
```

Dominio semántico

En la figura 3.8 se muestra el diagrama de clases correspondiente al dominio semántico de las máquinas de estados con estados compuestos. Como ya hicimos en la sección 2.1 se define una instancia de una máquina de estados como una especialización de la clase `SnapshotElementInstance`. El dominio semántico se amplía con dos nuevas clases para las instancias estados simples y compuestos, `SimpleStateInstance` y `CompositeStateInstance`, respectivamente.

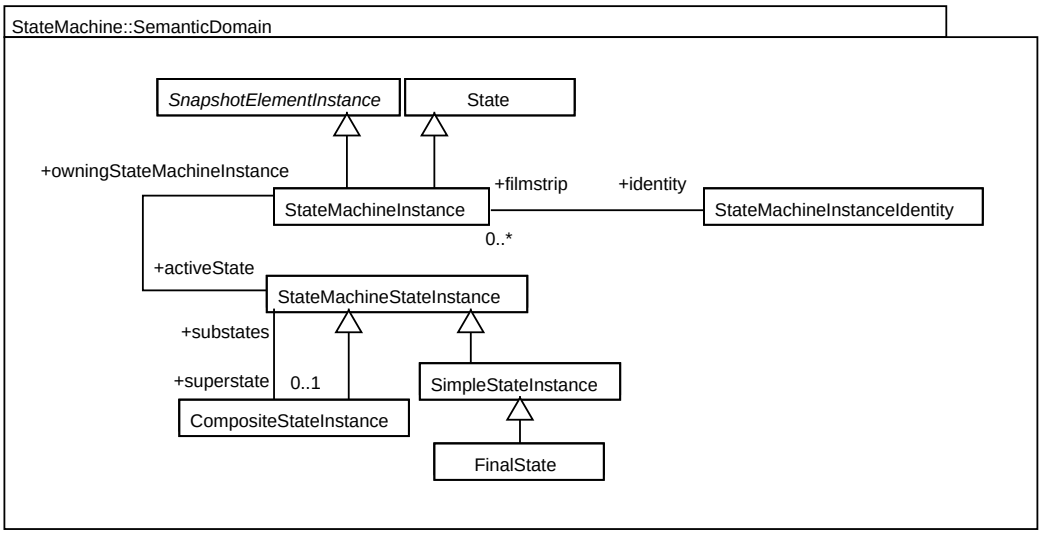

Figura 3.8: Dominio semántico de las máquinas de estados con estados compuestos.

Aplicación semántica

En la semántica de este modelo se incluyen respecto al de la sección anterior los diferentes tipos de estados, compuestos y simples (figura 3.9).

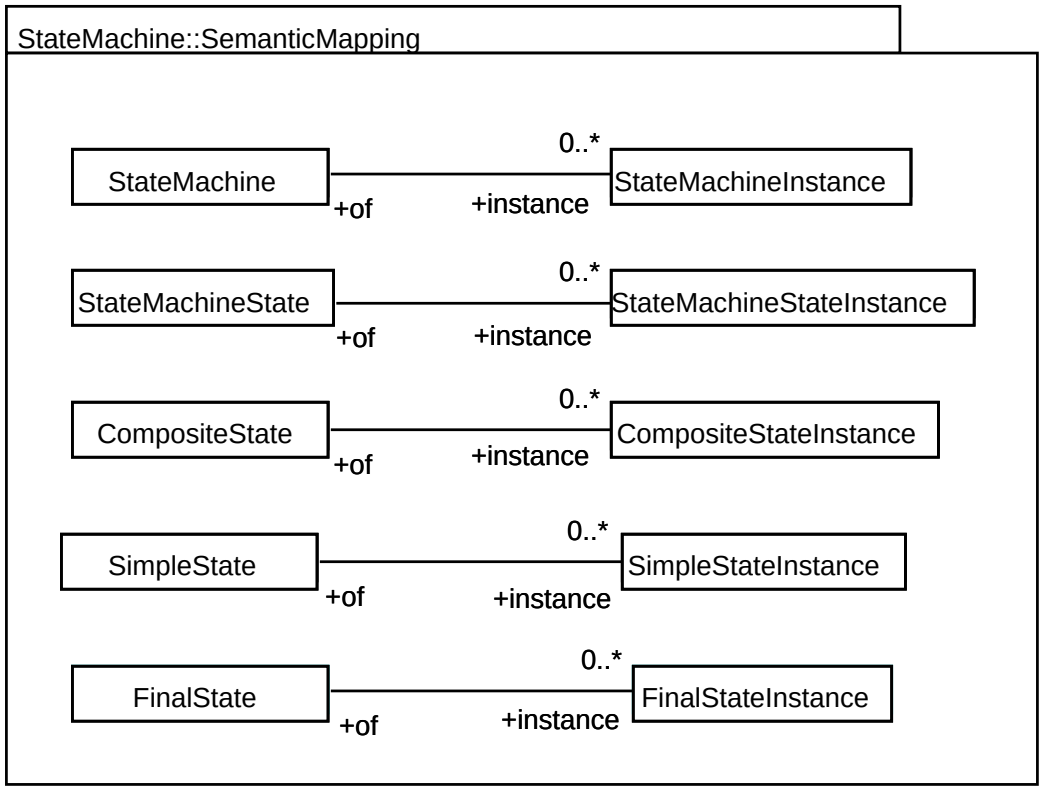

Figura 3.9: Aplicación semántica de las máquinas de estados compuestos.

Ejemplo

En la figura 3.10 se muestra una máquina que se compone, además del estado inicial y el final, de dos estados compuestos, cada uno de los cuales se refina en otra máquina con estado inicial, final y dos estados simples. En la figura 3.11 se muestra el diagrama de objetos correspondiente a la sintaxis abstracta de dicha máquina. En esta figura se han obviado las relaciones de composición entre la máquina de estados y las transiciones, definidas en la sintaxis abstracta, para mejorar la legibilidad.

En la figura 3.12 se muestra el diagrama de objetos correspondiente al dominio semántico, representando un posible estado activo de la máquina.

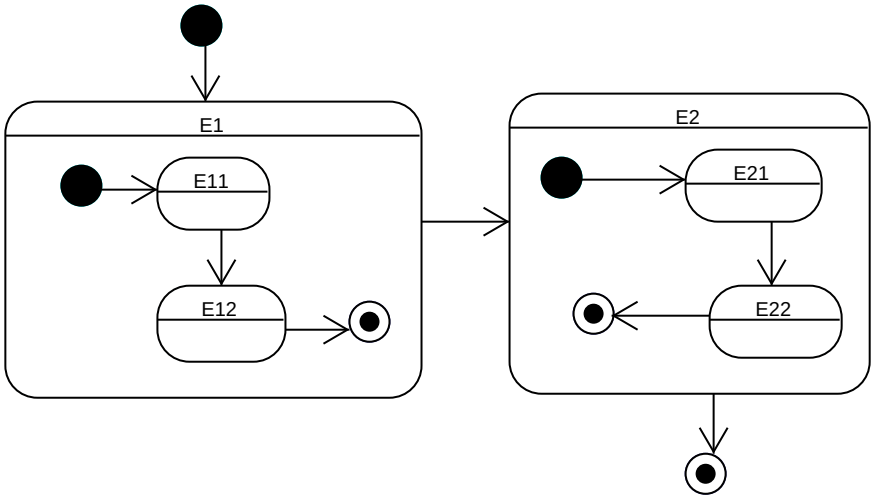

Figura 3.10: Máquina de estados con estados compuestos.

2.3. La máquina de estados con estados concurrentes

En esta sección hacemos una segunda ampliación del modelo de las máquinas de estados incluyendo un nuevo tipo de estado compuesto, los estados concurrentes. Los que hemos considerado en la sección anterior son los que se denominan secuenciales. La diferencia entre ambas categorías radica en la forma en que pueden ejecutarse los subestados contenidos. Cuando un estado secuencial está activo, sólo uno de sus subestados está activo. En cambio, cuando un estado concurrente está activo, todos sus subestados lo están también.

Los estados compuestos concurrentes se dividen en lo que se denominan *regiones*. Las regiones trabajan de manera concurrente. Debe haber al menos dos regiones en un estado concurrente y, cuando el estado concurrente está activo, todas las regiones también lo están. Las regiones también tienen que ser a su vez estados compuestos.

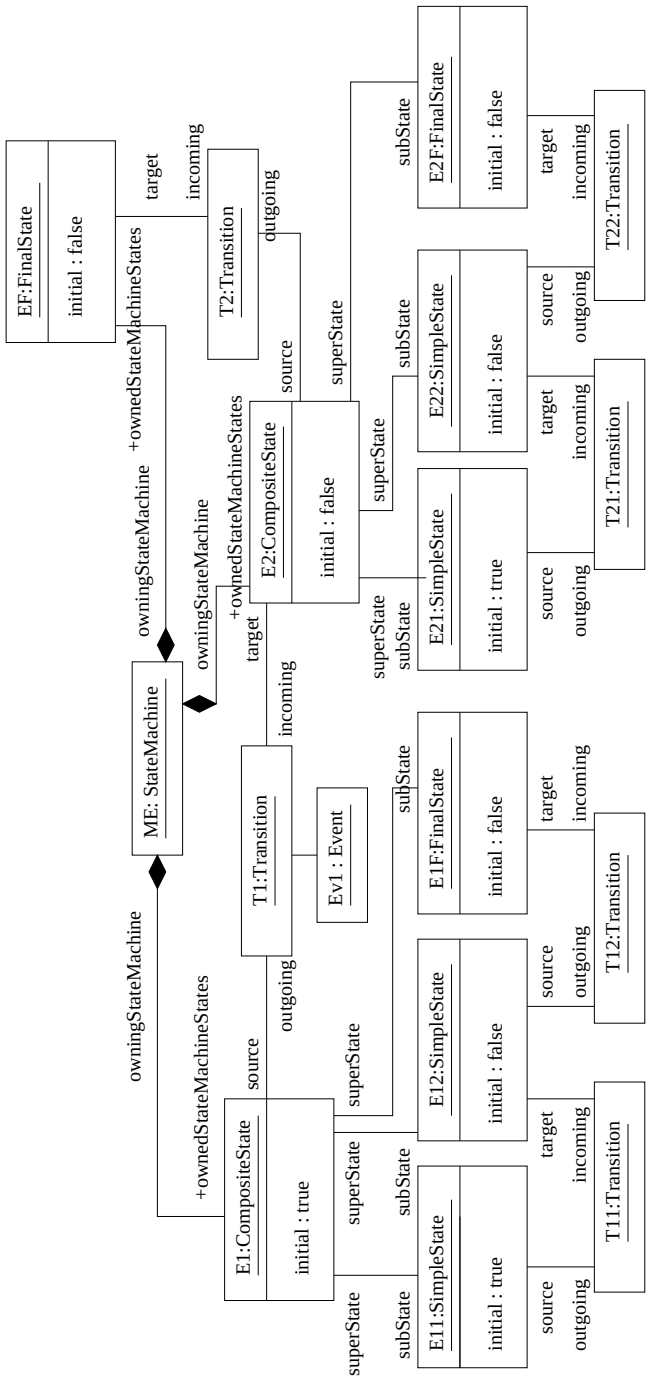

Figura 3.11: Diagrama de objetos de la sintaxis abstracta de la máquina de la figura 3.10.

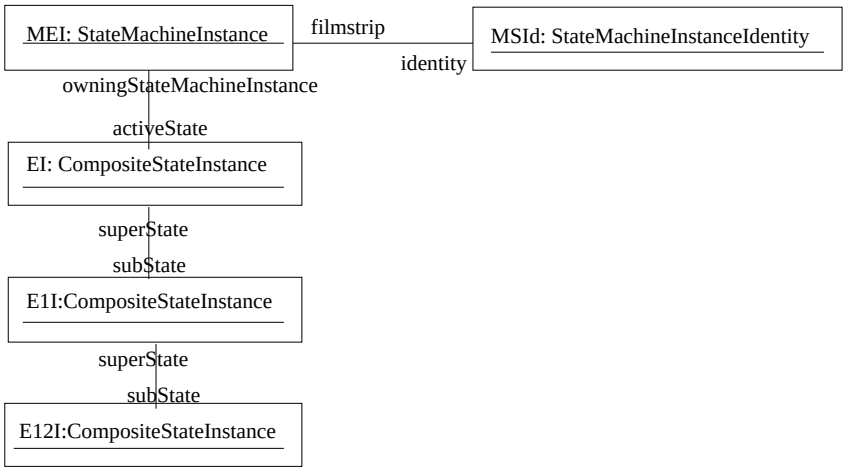

Figura 3.12: Diagrama de objetos del dominio semántico de la máquina de la figura 3.10.

Conceptos de modelado

En la figura 3.13 se muestra el diagrama de clases de los conceptos de modelado para las máquinas de estados con estados concurrentes. La principal diferencia en este paquete es que aparecen dos nuevas clases, `ConcurrentState` y `SequentialState`, que especializan `CompositeState`.

Reglas de corrección

1. Todos los subestados de un estado concurrente son estados compuestos.

```
context ConcurrentState inv

  substates->forall(sb | sb.oclIsKindOf(CompositeState))
```

2. Un estado concurrente tiene al menos dos subestados.

```
context ConcurrentState inv:

  substates->size >= 2
```

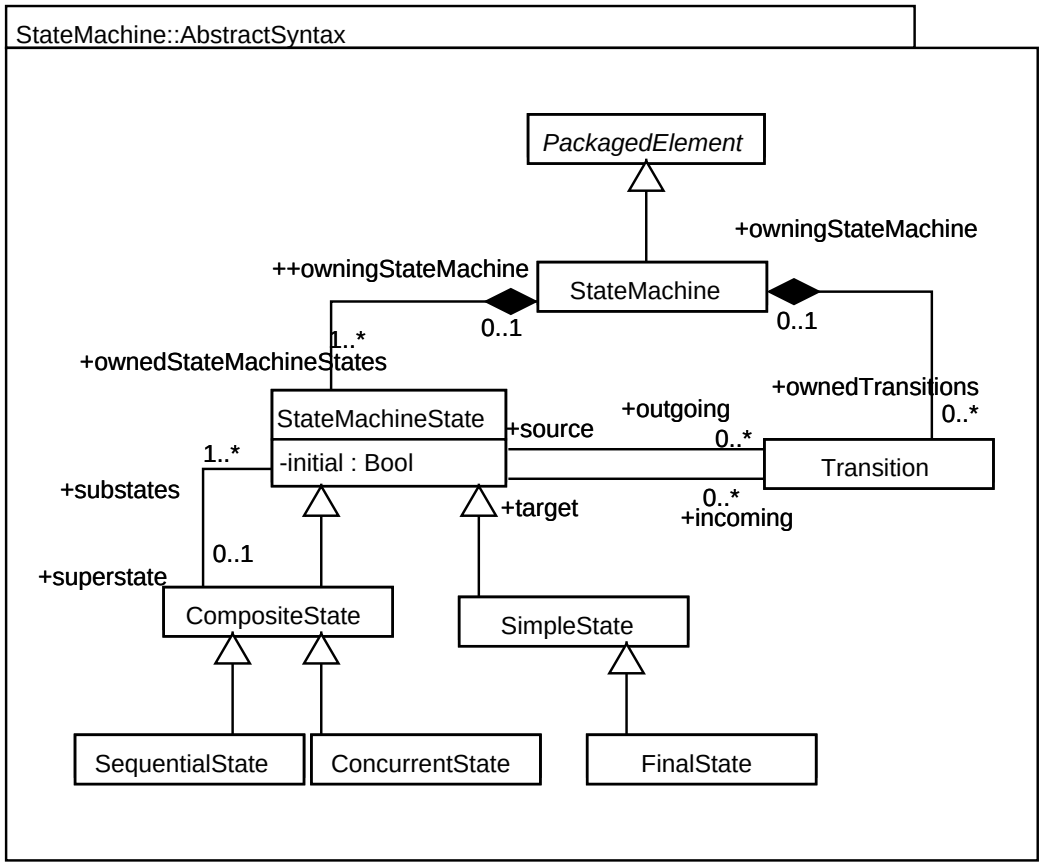

Figura 3.13: Conceptos de modelado de las máquinas de estados con estados concurrentes.

Dominio semántico

En la figura 3.14 se muestra el diagrama de clases correspondiente al dominio semántico de las máquinas de estados con estados concurrentes.

En este paquete también aparecen dos nuevas clases, `ConcurrentStateInstance` y `SequentialStateInstance`, que especializan la clase `CompositeStateInstance`. Otra diferencia es que en el modelo anterior, en el que sólo se consideraban los estados compuestos, la cardinalidad de la relación entre las clases `CompositeStateInstance` y `StateMachineStateInstance` era 1 en el extremo *substates* porque sólo podía estar activo un subestado del estado activo.

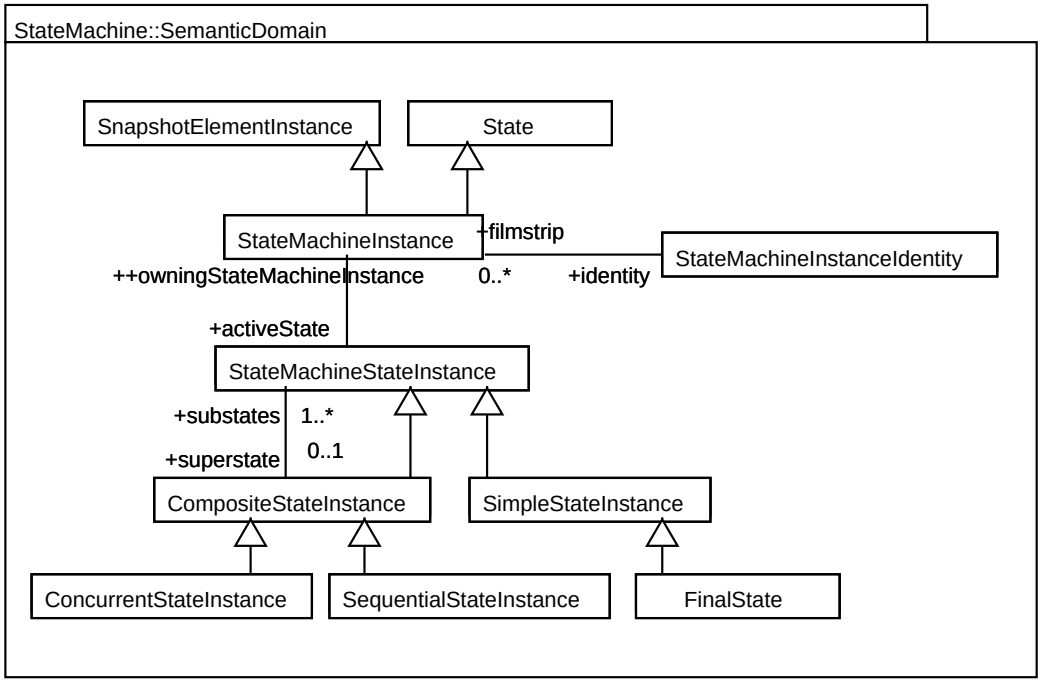

Figura 3.14: Dominio semántico de las máquinas de estados con estados concurrentes.

Ahora hay que añadir reglas de corrección para diferenciar cuántos subestados pueden estar activos en función de si el estado activo es secuencial o concurrente. Si es secuencial sólo habrá un subestado activo, y si es concurrente habrá uno por cada uno de sus subestados (*regiones*).

Reglas de corrección

1. Todos los subestados de un estado concurrente son estados compuestos.

   ```
   context ConcurrentStateInstance inv:

     substates->forall(sb | sb.oclIsKindOf(CompositeStateInstance))
   ```

2. Si el estado activo es compuesto y secuencial sólo hay un subestado activo.

   ```
   context SequentialStateInstance inv:
   ```

```
substates->size = 1
```

3. Si el estado activo es compuesto y concurrente hay un subestado activo por cada subestado definido en los conceptos de dominio.

```
context ConcurrentStateInstance inv:

  substates->size = self.of.substates.size
```

Aplicación semántica

En la semántica de este modelo se incluyen nuevas clases para los nuevos tipos de estados, los secuenciales y los concurrentes (figura 3.15).

Ejemplo

Para ilustrar el modelo de máquinas de estados concurrentes consideramos una máquina con dos estados, uno de los cuales es concurrente. En la figura 3.16 se muestra la máquina de estados. En la figura 3.17 se muestra el diagrama de objetos correspondiente a la sintaxis abstracta de dicha máquina. En la figura 3.18 se muestra el diagrama de objetos correspondiente a uno de los posibles estados. En este caso está activo el estado concurrente `EstCon`. Al ser concurrente también estarán activos un estado por cada uno de sus subestados. Concretamente, en la figura están activos los subestados `EC1-1` y `EC2-2`.

2.4. La recepción de un evento

En las secciones anteriores hemos ofrecido una visión estática de las máquinas de estados en las que se muestra un estado activo en el que permanece la máquina hasta que transita a otro estado. Las reglas de corrección presentadas se pueden incluir en lo que se denomina *semántica estática* en la definición de UML [118].

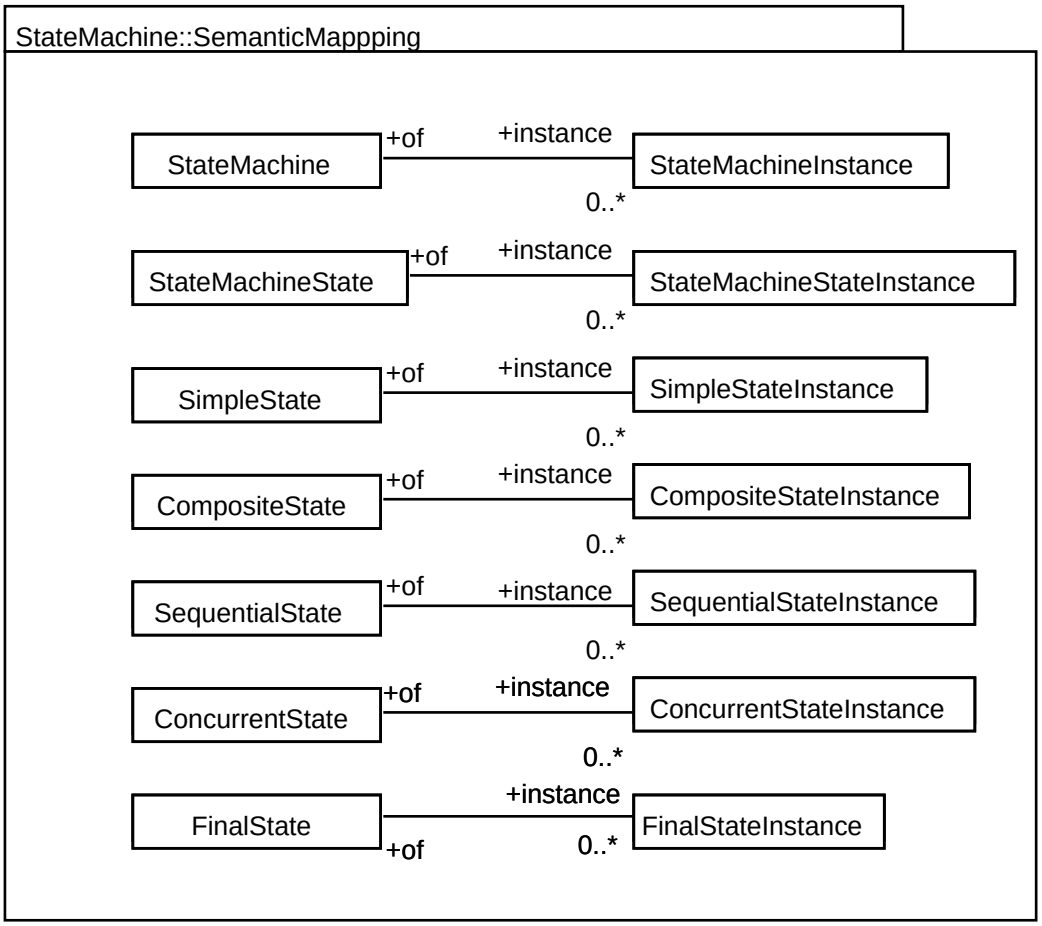

Figura 3.15: Aplicación semántica de las máquinas de estados concurrentes.

El aspecto verdaderamente dinámico de las máquinas de estados reside en cómo se cambia de un estado a otro cuando se produce la ocurrencia de un evento y qué acciones se ejecutan como respuesta.

Además de los que no se tienen en cuenta para simplificar el modelo, hasta ahora se han dejado aparte intencionadamente varios elementos. Estos elementos son: el *evento* que dispara una transición, la *guarda* que la habilita, las *acciones* asociadas, tanto a los estados como a las transiciones, y la recepción de un evento.

Una máquina de estados cambia de estado como respuesta a la llegada

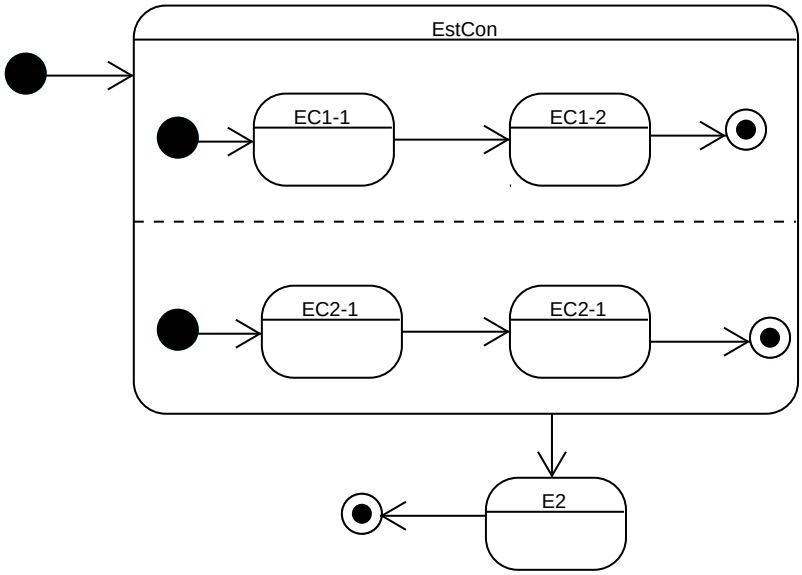

Figura 3.16: Máquina de estados con estados concurrentes.

de una instancia de un evento. Si en el estado actual de la máquina hay una transición cuyo origen es un estado activo, la máquina responde al evento disparando dicha transición. La máquina evoluciona entonces al estado destino de la transición.

Las transiciones responden a un solo evento, y para poder hacerlo han de estar habilitadas. Una transición está habilitada si su *guarda*, una expresión estática que devuelve un valor lógico, se evalúa a verdadero en el momento de la recepción del evento.

En una máquina de estados, los estados llevan asociadas, como parte de la respuesta de la máquina a la ocurrencia de un evento, varias acciones: la de entrada, la de salida y la llamada *actividad*[1]. También las transiciones pueden tener asociada una acción. Estas acciones se ejecutan secuencialmente como parte de la respuesta de la máquina a la ocurrencia de un evento. Primero se ejecutan las acciones de salida del estado del que se parte, después las

[1]Las *actividades* son una de las características que no vamos a considerar en nuestro modelo de la máquinas de estados.

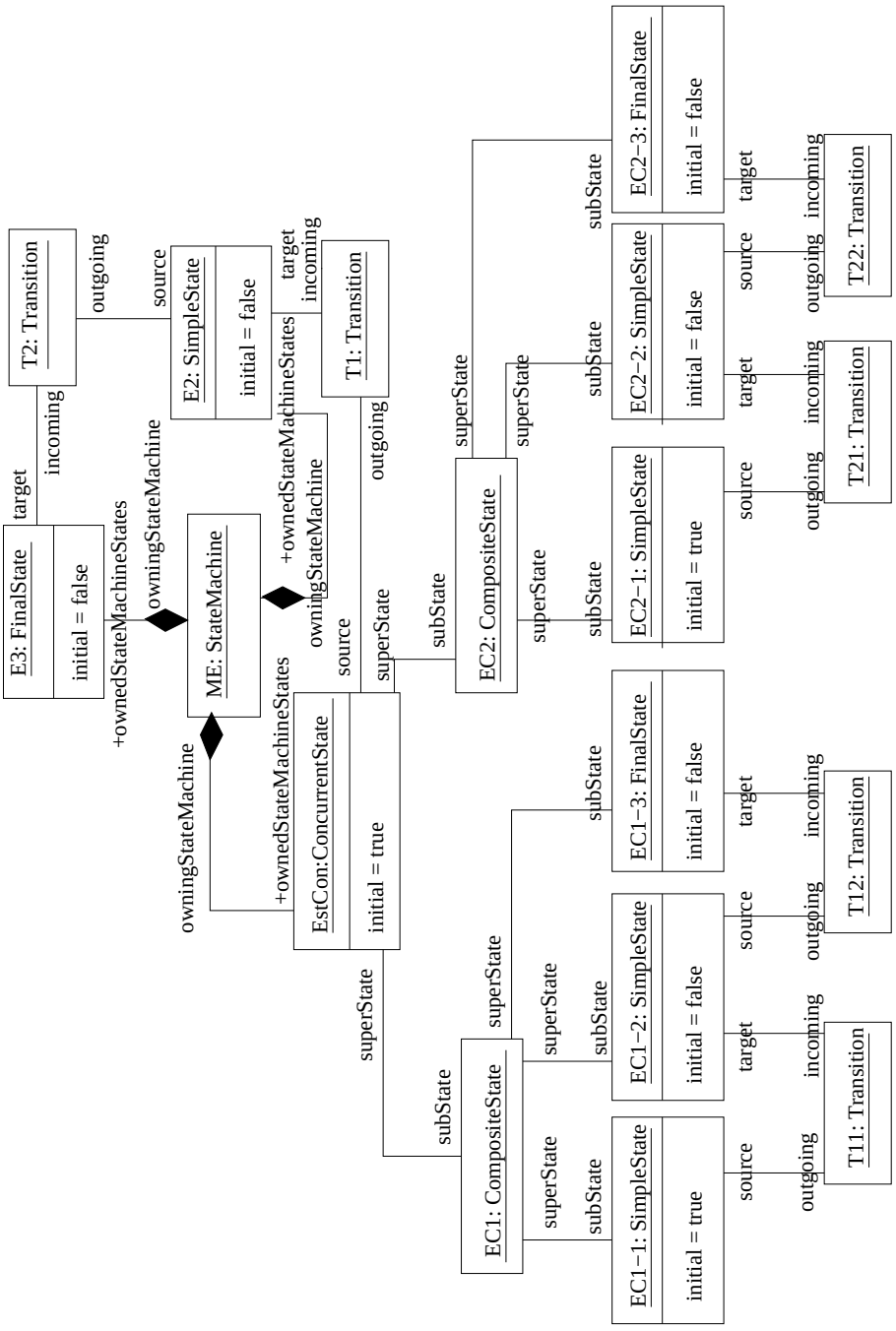

Figura 3.17: Diagrama de objetos de la sintaxis abstracta de la máquina de la figura 3.16.

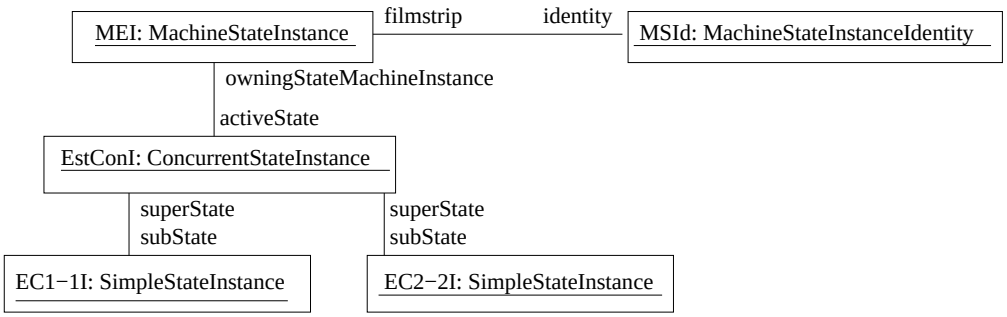

Figura 3.18: Diagrama de objetos del dominio semántico de la máquina de la figura 3.16.

acciones asociadas a la transición y, finalmente, las acciones de entrada del estado al que se llega.

Si la instancia de un evento llega cuando la máquina está en un estado en el que no hay ninguna transición para él, esa ocurrencia se consume sin respuesta. Cada estado puede, sin embargo, especificar lo que se denomina la lista de eventos postergados. Estos eventos no son tratados en el estado, pero tampoco se descartan, sino que se guardan para su tratamiento posterior cuando se cambie a un nuevo estado. La lista de eventos postergados tampoco la incluiremos en nuestro modelo.

En este modelo se incluyen los dos tipos de transiciones definidos en UML, las *externas* y las *internas*. La externas son las que se han tratado hasta ahora: reaccionan frente a la ocurrencia de un evento y pueden provocar el cambio de estado y la ejecución de las acciones asociadas. Por su lado, las internas no efectúan cambio de estado, sólo consumen la ocurrencia de un evento y sólo se ejecuta la acción asociada a la transición, no las de salida o entrada del estado.

El cambio de estado

Cuando se recibe un evento, si en el estado actual, o en alguno de sus subestados, hay alguna transición que se dispare con una ocurrencia de ese evento y dicha transición está activa, es decir, su guarda se evalúa a verdadero, se dispara esa transición.

Como los estados pueden estar anidados, tanto el origen como el destino de la transición que se ejecuta pueden ser subestados de otro u otros. Es posible, por tanto, que la ejecución de una transición provoque que se salga de más de un estado y se entre en más de un estado. En el estado final de la máquina de estados no puede aparecer ninguna instancia de los estados de los que se ha salido.

Para saber de qué estados se sale, se puede usar el concepto de *LCA*, *Least Common Ancestor*, definido en [118]. El LCA de dos estados es el ancestro común más interior del estado origen y el destino. Es decir, es un ancestro común tal que no hay ningún otro ancestro común que sea a su vez descendiente del LCA. Cuando se ejecuta una transición, se sale de todos aquellos estados activos que sean descendientes del LCA.

Los estados en los que se entran son los que cumplen las siguientes condiciones:

1. es descendiente del LCA de los estados origen y destino de la transición, y

2. *a*) es el estado destino, o

   b) es un ancestro del estado destino, o

   c) su superestado se activa en el estado destino y

      1) es el estado inicial de un superestado secuencial, o

2) el superestado es concurrente.

La última regla es la que hace referencia a aquellos estados que no son ascendientes directos del estado final pero que acaban activos porque pertenecen a alguna región de un estado concurrente que es tanto ascendiente del estado destino como descendiente del LCA de los estados origen y destino.

En la figura 3.19 se muestra una máquina de estados como ejemplo. En esta máquina en la que en el estado principal están el estado *A*, que es concurrente y se divide en dos regiones, y el estado B, que es simple. Estas regiones, a su vez, se dividen en otros estados compuestos, bien secuenciales o concurrentes. Por claridad, sólo se muestran las transciones a los estados iniciales y otras dos transiciones, una que parte del estado *A112* y llega al estado *B*, disparada por el evento *ev1* y otra que parte del estado *B* y llega al estado *A211*, disparada por el evento *ev2*.

Supongamos que, en un momento dado, el estado en el que se encuentra esta máquina implica que están activos los estados: *A*, *A1*, *A2*, *A11*, *A12*, *A21*, *A112*, *A121*, *A211*, *A212*, *A2112* y *A2121*. Si hay una ocurrencia del *ev1* y se ejecuta la transición asociada, el LCA de los estados *A112* y *B* es *EstadoPrincipal*, por lo que se sale de todos los estados activos excepto el estado principal. Por otro lado, si se produce una instancia del evento *ev2* siendo *B* el estado activo, se saldrá del estado *B* y se entrará en los estados *A*, *A1*, *A2*, *A11*, *A12*, *A21*, *A111*, *A121*, *A211*, *A212*, *A2111* y *A2121*. Los estados *A212* y *A2121* son ejemplos de estados que se activan por la última de las reglas enunciadas anteriormente. No son superestados del estado final, *A211*, pero pertenecen, a distinto nivel, a una región del estado concurrente *A21*, que es superestado del estado destino y descendiente del origen y el destino de la transición.

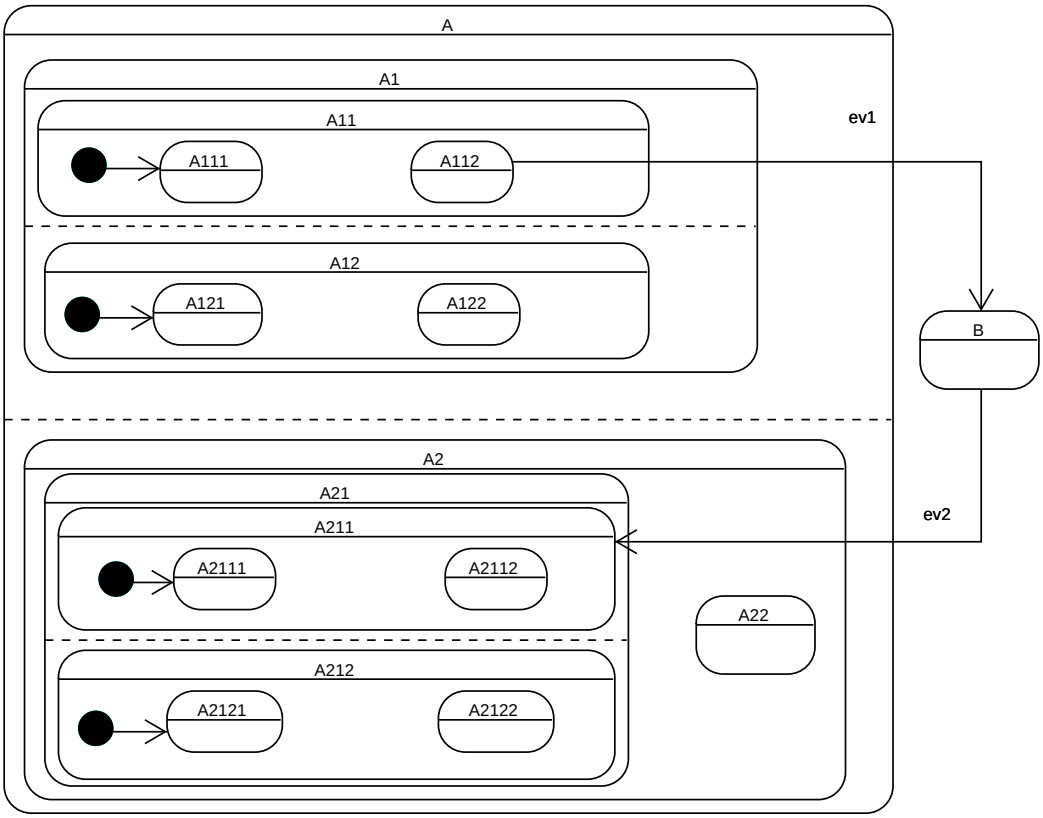

Figura 3.19: Máquina de estados con eventos.

La ejecución de las acciones

Durante la ejecución de una transición se ejecutan tres acciones secuencialmente y en este orden: la acción de salida del estado actual, la acción de la transición y la acción de entrada del estado destino de la transición.

Si consideráramos sólo el modelo de la máquina de estados plana, es muy fácil ver que el estado en el que está la máquina tiene que ser una instancia del estado origen, *source*, de la transición y que el estado en el que queda la máquina tiene que ser una instancia del estado final, *target*, de la transición.

Si pasamos a considerar estado compuestos, la cuestión se complica porque, como ya hemos visto, el estado actual de una máquina puede involucrar a más de un estado. Como la transición puede salir de varios estados anida-

dos, se ejecutan las acciones de salida de todos los estados de los que se sale. En la especificación oficial de UML 2.0 [118], se dice que se ejecutan secuencialmente empezando por la acción del estado más interno. Eso es válido para cuando los estados involucrados son secuenciales, pero no se dice nada sobre en qué orden se ejecutan las acciones de salida si se salen de las diferentes regiones de un estado concurrente.

Una posibilidad es considerar que se sale de las regiones de manera concurrente, es decir, que las acciones de salida de los estados de los que se sale de dentro de una región se ejecutan de manera concurrente con las acciones de salida de los estados de las otras regiones del mismo estado concurrente. Dentro de una región se ejecutarían secuencialmente desde el estado más interno como se especifica en [118].

Volvamos otra vez al ejemplo de la figura 3.19 y volvamos también a suponer que el estado en el que se encuentra dicha máquina implica que están activos los estados: *A*, *A1*, *A2*, *A11*, *A12*, *A21*, *A112*, *A121*, *A211*, *A212*, *A2112* y *A2121*. Si llega una instancia del evento *ev1* y se ejecuta la transición asociada hasta el estado *B*, se sale de todos los estados activos. La figura 3.20 muestra el diagrama de actividad que corresponde a la ejecución de las acciones según el modelo comentado en el párrafo anterior. En la figura 3.21 se muestra el objeto de la clase `Action` a la que corresponde dicha ejecución.

Lo mismo ocurre con las acciones de entrada: si una transición entra en más de un estado anidado, se ejecutan secuencialmente las acciones de entrada de todos los estados en los que se entra. En este caso, el orden es el inverso y se ejecuta primero la acción del estado más exterior. Si se entra en un subestado de un estado concurrente, se activan todas las regiones de ese estado concurrente. La especificación de UML no aclara cuál es el orden en el

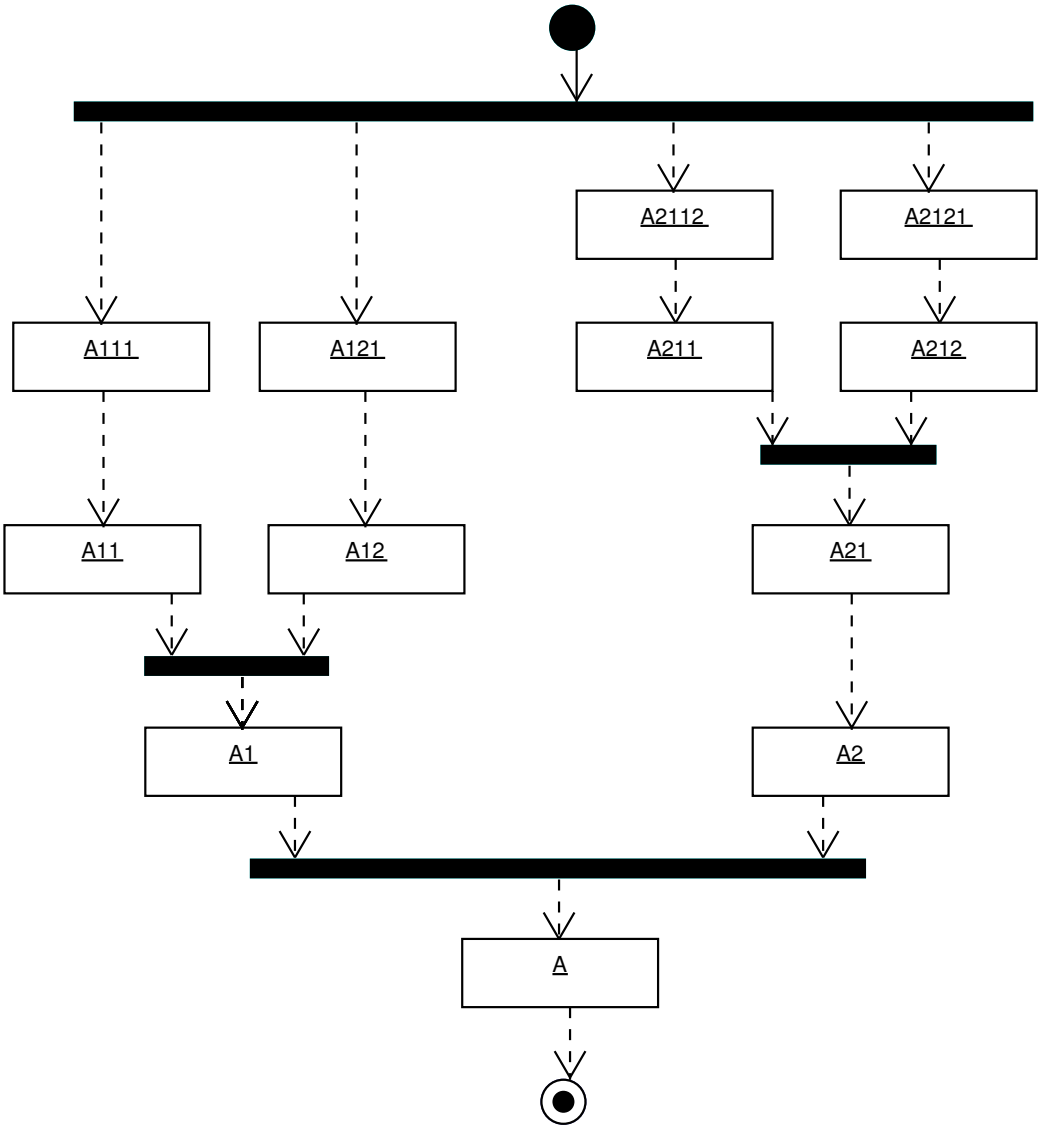

Figura 3.20: Diagrama de actividad de la ejecución de la transición del evento *ev1* en la figura 3.19 supuesto el estado indicado.

que se ejecutan las acciones de entrada de los estados activos de las regiones. Podemos suponer que se ejecutan como acciones concurrentes independientes todas las acciones de entrada que pertenezcan a la misma región.

En ese caso, si en el ejemplo de la figura 3.19 el estado B está activo y se produce una instancia del evento *ev2*, se efectuarán como acciones de

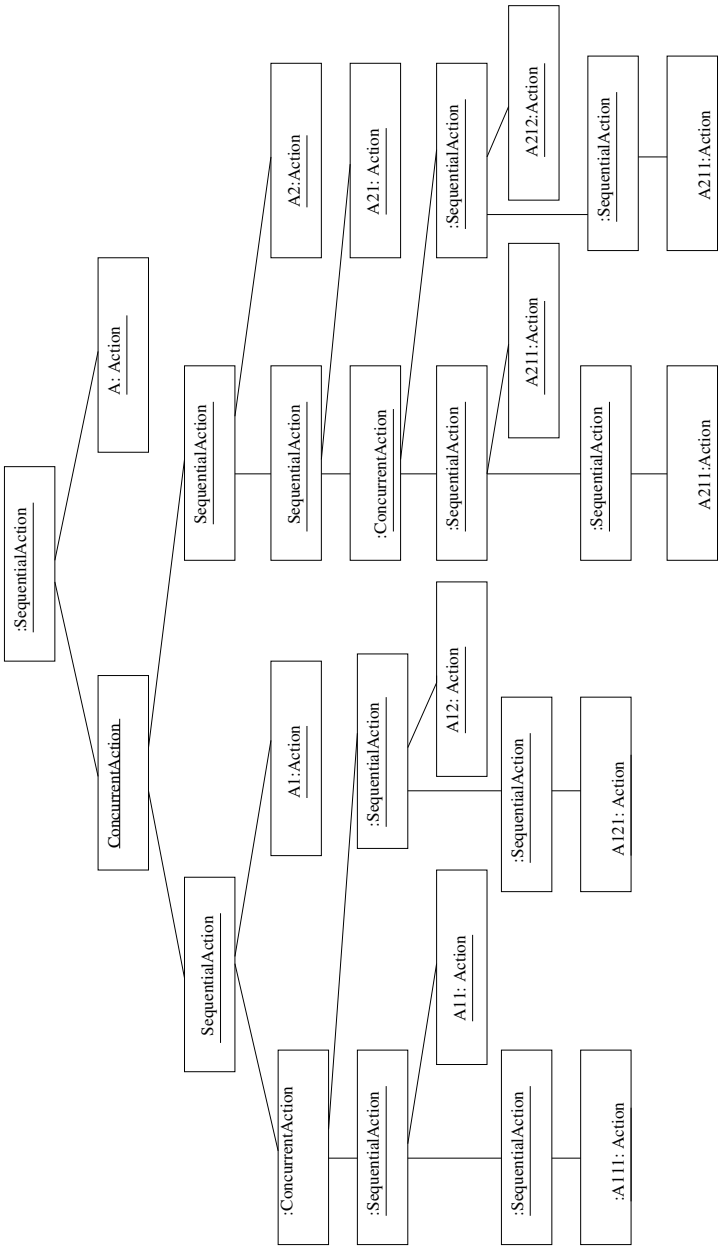

Figura 3.21: Evaluación de la acción compuesta correspondiente a la ejecución de la transición del evento *ev1* en la figura 3.19 en el estado considerado.

entrada las que se muestran en el diagrama de actividad de la figura 3.22. La acción compuesta correspondiente se muestra en el diagrama de objetos de la figura 3.23

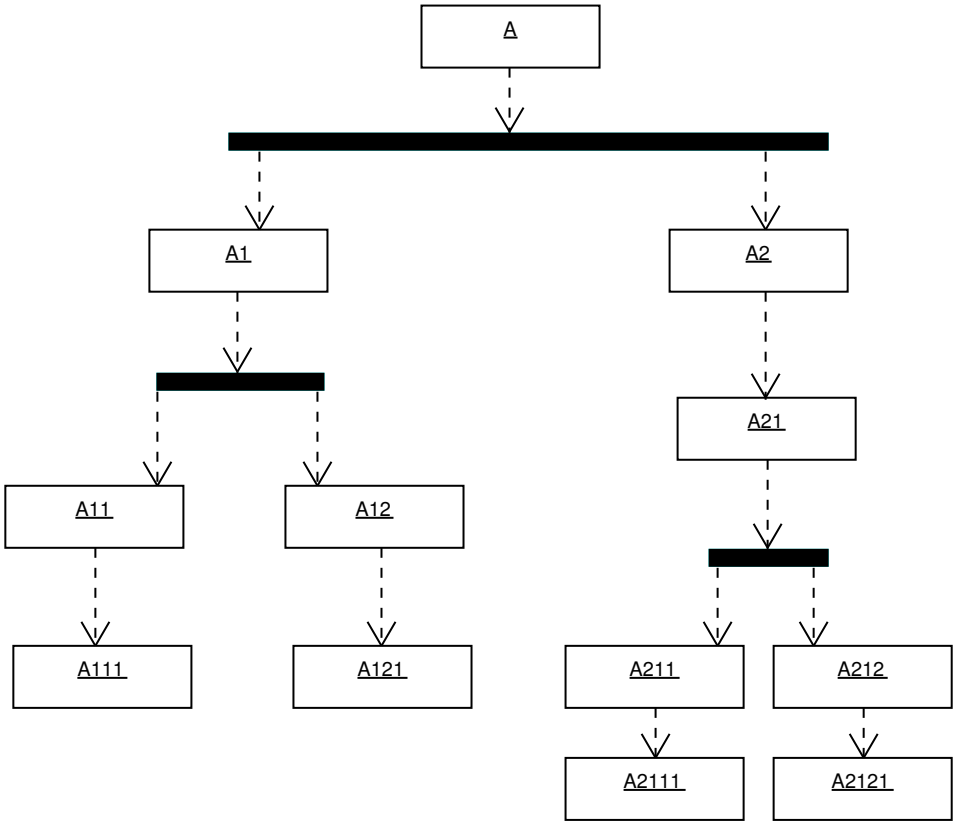

Figura 3.22: Diagrama de actividad de las acciones de entrada en la ejecución de la transición del evento *ev2* en la figura 3.19 suponiendo el estado indicado.

Conceptos de modelado

En la figura 3.24 se muestra el diagrama de clases de los conceptos de modelado para las máquinas de estados teniendo en cuenta las acciones.

En este paquete se añade a los estados las acciones que se ejecutan al realizar un cambio de estado, la acción de salida y la acción de entrada, representadas por las relaciones *exitAction* y *entryAction* con la clase `Action`.

Por su parte, las transiciones también se amplían con otra relación con la clase `Action`, *transitionAction*, que representa la acción asociada a la transición. La otra relación nueva de las transiciones es la guarda, representada por una instancia de la clase `StaticExpression`.

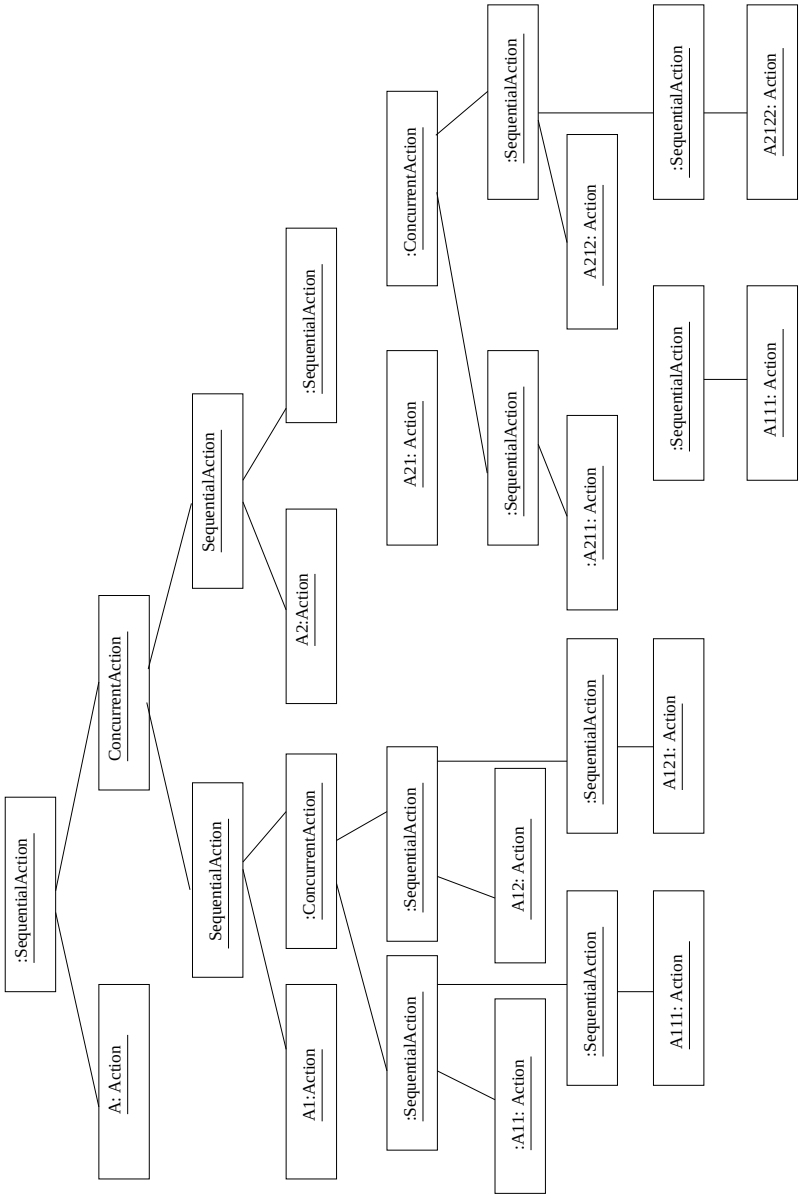

Figura 3.23: Diagrama de objetos correspondiente a la acción compuesta de la ejecución de la transición del evento *ev2* en la figura 3.19 en el estado considerado.

Reglas de corrección

1. El valor de la guarda de la transición es de tipo `Boolean`.

```
context Transition inv:
```

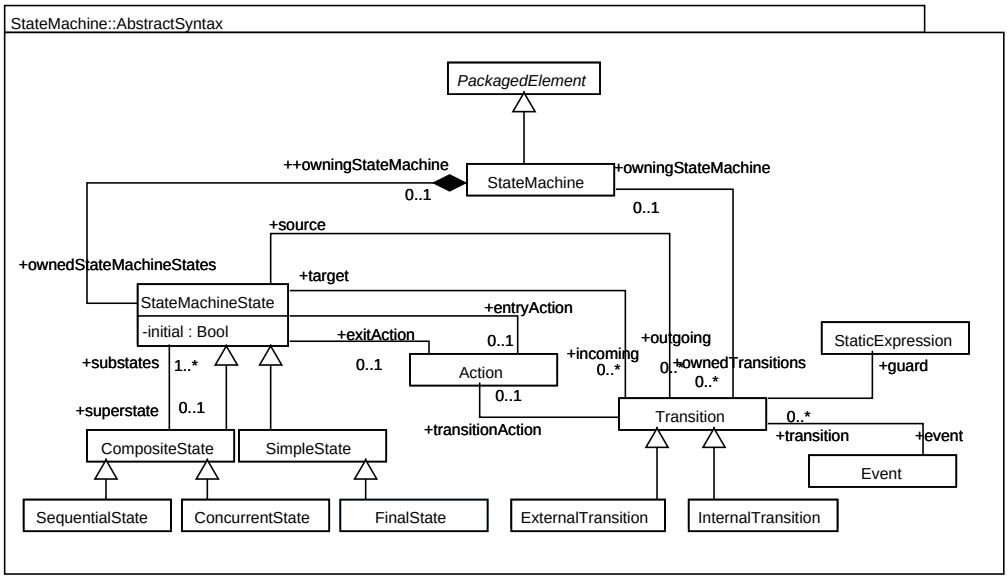

Figura 3.24: Conceptos de modelado de las máquinas de estados con acciones.

```
self.guard->type.isKindOf(Boolean)
```

2. Una transición no puede ir de una región a otra dentro del mismo estado concurrente.

```
context Transition inv:

  self.source.superState.isKindof(ConcurrentState) implies
  self.source.superState <>self.target.superState
```

Métodos

1. `ancestor(s1, s2)` comprueba si `s1` es un estado ancestro de `s2`.

```
context StateMachine:
ancestor (s1 : StateMachineState, s2 : StateMachineState) :
Boolean
```

```
if (s2 = s1) then
   true
else
   if (s1.superState->isEmpty) then
      true
   else
      if (s2.superState->isEmpty) then
         false
      else
         ancestor (s1, s2.superState)
      endif
   endif
endif
```

2. `LCA(s1,s2)` devuelve el mínimo estado común ancestro de `s1` y `s2`. Definida en [118] como:

```
context StateMachine:
LCA (s1 : StateMachineState, s2 : StateMachineState) :
CompositeState

  if self.ancestor (s1, s2) then
     s1
  else
     if self.ancestor (s2, s1) then
        s2
     else
        self.LCA (s1.superstate, s2.superstate)
     endif
  endif
```

Dominio semántico

En la figura 3.25 se muestra el diagrama de clases correspondiente al dominio semántico de las máquinas de estados teniendo en cuenta las acciones.

Una de las clases nuevas en este diagrama, `EventReception`, representa la

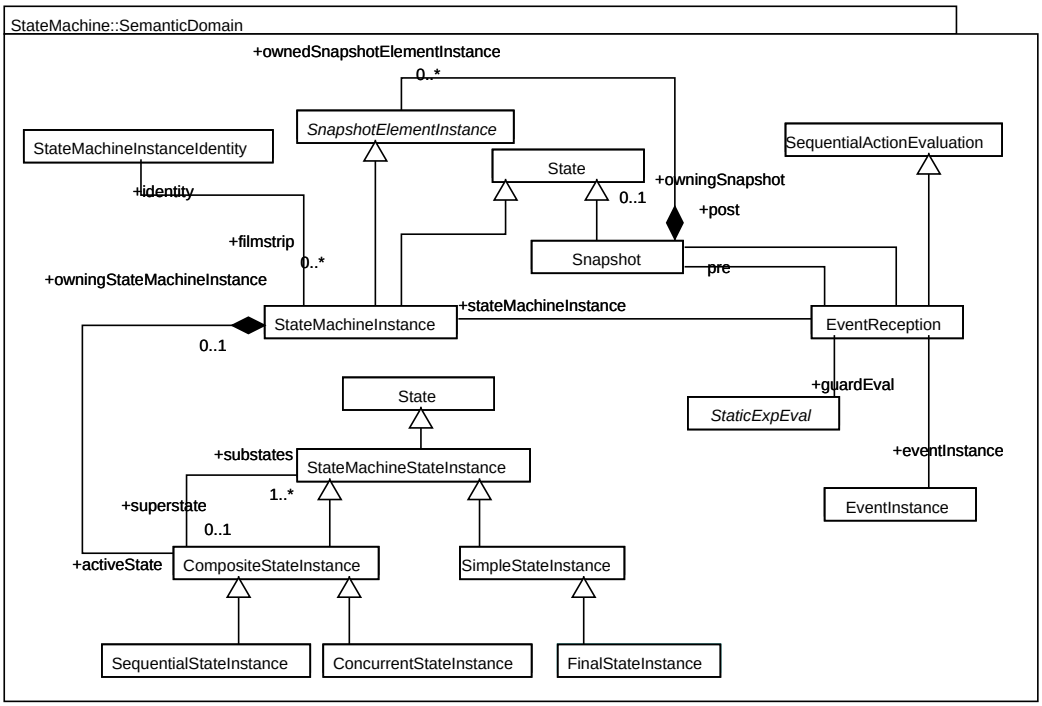

Figura 3.25: Dominio semántico de las máquinas de estados con acciones.

acción que se ejecuta en el ámbito de una máquina de estados cuando se recibe y procesa una ocurrencia de un evento. Como ocurre con las demás acciones descritas en el capítulo dedicado a las acciones de [121], una instancia de la clase `EventReception` está asociada con dos instancias de la clase `Snapshot`, *pre* y *post*, que representan, respectivamente, el estado en el que se encontraba la máquina de estados cuando se produjo la ocurrencia del evento y el estado en el que queda la máquina tras el procesamiento de dicho evento.

La recepción de un evento tiene más información relacionada. Por un lado, la instancia de la máquina de estados que recibe el evento, que es parte del *snapshot* de origen; por eso, la clase `StateMachineInstance` es una subclase de `SnapshotElementInstance`. Por otro lado, está asociada a la instancia del evento que se procesa y, finalmente, a las acciones que se ejecutan como parte del procesamiento de ese evento.

Reglas de corrección

1. La guarda de la transición se evalúa a verdadero para que la transición esté activa.

```
context EventReception inv:

  self.guardEval = true
```

2. La guarda de la transición se evalúa en el estado activo inicial.

```
context EventReception inv:

  self.guardEval.subActionEval.forall(sae |
  sae.ExpressionEvaluations()->forall(ee |
  ee.allSubExpressionEvaluations()->forall(see |
  see.ReferredInstances()->forall(ri |
  ri.owningSnapshot = self.pre)))
```

3. La recepción del evento implica la ejecución de tres acciones.

```
context EventReception inv:

  exec.subExecution->size = 3
```

4. Las transiciones internas no cambian de estado.

```
context InternalTransition inv:

  source = target
```

5. Si la transición es interna, sólo se ejecuta la acción de la transición.

```
context EventReception inv:

  self.of.isKindOf(InternalTransition) implies
  (previous = next and
  previous.allSubtates() = next.allSubtates() and
  self.subExecution->at(1).oclIsTypeOf(NullExecution) and
  self.subExecution->at(2).oclIsKindOf(self.of.transitionAction)
  and self.subExecution->at(3).oclIsTypeOf(NullExecution))
```

6. Si la transición es externa, la primera acción debe ser una acción compuesta que corresponda con las acciones de salida de los estados de los que se sale. El árbol de subestados activos de los que se sale depende del LCA de los estados origen y destino y del estado activo actual.

```
context EventReception inv:

  self.of.oclIsKindOf(ExternalTransition) implies
  self.pre.activeSubStatesInstancesTree(
  self.of.owningStateMachine.LCA(self.of.source,self.of.target))
  ->any(true).validExitEvaluation(self.subExecution->at(1)))
```

7. Si la transición es externa, la segunda acción se corresponde con la acción asociada a la transición.

```
context EventReception inv:

  (self.of.oclIsKindOf(ExternalTransition) and
  self.of.transitionAction->size = 1 implies
  exec.subExecution->at(2).of = self.of.transitionAction) and
  (self.of.oclIsKindOf(ExternalTransition) and
  self.of.source.transitionAction->size = 0 implies
  exec.subExecution->at(2) = nullActionExecution)
```

8. Si la transición es externa, la tercera acción debe ser una acción compuesta que corresponda con las acciones de entrada de los estados que acaban siendo activos al final de la transición. El árbol de subestados que se activan depende del LCA de los estados origen y destino de la transición y del estado activo final.

```
context EventReception inv:

  self.of.oclIsKindOf(ExternalTransition) implies
  self.post.activeSubStatesInstancesTree(
  self.of.owningStateMachine.LCA(self.of.source,self.of.target))
  ->any(true).validExitEvaluation(self.subExecution->at(3)))
```

9. El estado final no puede contener ninguno de los estados de los que se ha salido.

```
context EventReception inv:

  self.post.allSubStatesInstances.excludesAll(
  self.pre.activeSubStatesInstancesTree(
  self.of.owningStateMachine.LCA(self.of.source,self.of.target))
  ->any(true).allSubStatesInstances())
```

Métodos

1. El método `allSubstatesInstances()` devuelve el conjunto de todas las instancias de subestados de una instancia de un estado.

```
context StateMachineStateInstance:
allSubstatesInstances():Set(StateMachineStateInstance)

  {}
```

2. El método `allSubstatesInstances()` devuelve el conjunto de todas las instancias de subestados de una instancia de un estado compuesto.

```
context CompositeState:
allSubstatesInstances():Set(StateMachineStateInstance)

  substates->iterate(ss ac = substates | ac->union(ss.allSubStates()))
```

3. El método `allSubstatesInstances()` devuelve el conjunto de todas las instancias de subestados de una instancia de un estado simple.

```
context SimpleState:
allSubstatesInstances():Set(StateMachineStateInstance)

  {}
```

4. El método `postState()` devuelve el estado de la máquina de estados en el siguiente *snaphot*, es decir, el estado tras la ejecución de la recep-

ción del evento.

```
context EventReception:
postState() : StateMachineInstance
  self.post.ownedSnapshotElementInstance.select(osei |
  osei.identity = self.stateMachineInstance.identity)
```

5. El método `activeSubStatesInstancesTree()` devuelve el árbol de instancias de subestados cuya raíz es una instancia del estado que se pasa como parámetro. Si el estado sobre el que se aplica es una instancia del parámetro, entonces se devuelve como raíz del árbol el subestado que es ancestro del destino de la transición. En caso contrario, se busca entre los subestados. Este método se usa para calcular de qué estados se sale al ejecutar una transición aplicándolo sobre el estado origen de la transición y pasando como parámetro el LCA de los estados origen y destino de la transición.

```
context StateMachineStateInstance:
activeSubStatesInstancesTree(sms : StateMachineState)
: StateMachineStateInstance
  if self.of = sms then
     self.subStates()->select(ss | ss.of.ancestor(self.of.target))
  else
     self.subStates()->iterate(ac = Set{}, ss |ac = Set{},
     ac->union(ss.activeSubStatesInstancesTree(sms)))
  endif
```

6. El método `validExitEvaluation()` devuelve `true` si un árbol de ejecuciones de acciones es una ejecución válida de un árbol de estados del que se sale en la ejecución de una transición.

```
context StateMachineStateInstance:

validExitEvaluation(acteval: ActionEvaluation): Bool

  if self.isKindOf(simpleState) then
  acteval.of = self.exitAction
  else
     if self.isKindOf(sequentialState) then
        acteval.subEvals()->size = 2 and
        acteval.subEvals()->at(2).of = self.exitAction and
        acteval.subEvals()->at(1).
        isKindOf(sequentialActionEvaluation) and
        self.substates.validExitEvaluation(acteval.subevals()->at(1))
     else if self.isKindOf(concurrentState) then
        acteval.subEvals()->size = 2 and
        acteval.subEvals()->at(2).of = self.exitAction and
        acteval.subEvals()->at(1).
        isKindOf(concurrentActionEvaluation) and
        self.substates()->foreach(ss | actEval.subEvals()->
        one(se | ss.validExitEvaluation(se)))
     endif
  endif
```

7. El método `validEntryEvaluation()` devuelve `true` si un árbol de ejecuciones de acciones es una ejecución válida de un árbol de estados del que se sale en la ejecución de una transición.

```
context StateMachineStateInstance:

validEntryEvaluation(acteval: ActionEvaluation): Bool
```

```
if self.isKindOf(simpleState) then
acteval.of = self.exitAction and
acteval.subEvals()->size = 0
else
   if self.isKindOf(sequentialState) then
      acteval.isKindOf(seqAction) and
      acteval.subEvals()->size = 2 and
      acteval.subEvals()->at(1).of = self.entryAction and
      acteval.subStates()->at(1).validEntryEvaluation(
      acteval.subevals()->at(2))
   else // self.isKindOf(concurrentState) then
      acteval.isKindOf(seqAction) and
      acteval.subEvals()->size = 2 and
      acteval.subEvals()->at(1).of = self.entryAction and
      acteval.subEvals()->at(2).
      isKindOf(concurrentActionEvaluation) and
      self.substates()->foreach(ss | actEval.subEvals()->at(2)->
      subActionEvals()->one(se | ss.validEntryEvaluation(se)))
   endif
endif
```

Aplicación semántica

En la semántica de este modelo se incluyen, además de los previamente considerados, los conceptos relativos a los eventos y las transiciones (figura 3.26).

Ejemplo

En la figura 3.27 se muestra una máquina en la que sólo se han marcado algunas transiciones y acciones. En la figura 3.28 se muestra el diagrama de objetos correspondiente a la sintaxis abstracta de dicha máquina. En la figura 3.29 se muestra el diagrama de objetos correspondiente al dominio semántico. El diagrama corresponde a dos instantes en la vida del objeto, que estaba en los estados *E-1*, *E1-1*, *E13-1* y *E14-1* y tras recibir una instancia

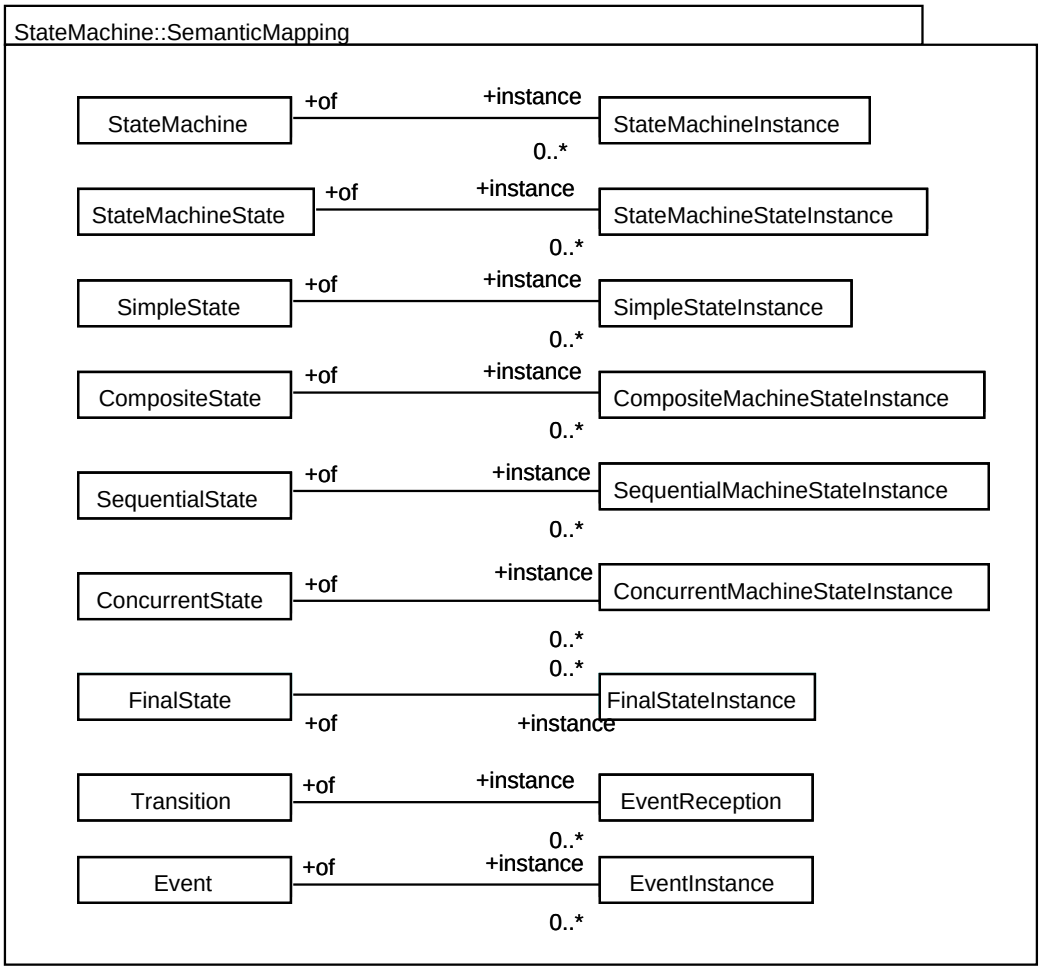

Figura 3.26: Aplicación semántica de las máquinas de estados con acciones.

del evento *et1*, ha transitado a los estados *E-2*, *E2-1*, *E21-1* y *E211-1*. En estas dos últimas figuras no se muestran los nombres de los extremos de las relaciones para evitar que la figura esté demasiado cargada. Tampoco se han incluido las relaciones composición entre las transiciones y la máquina de estados en la figura 3.28.

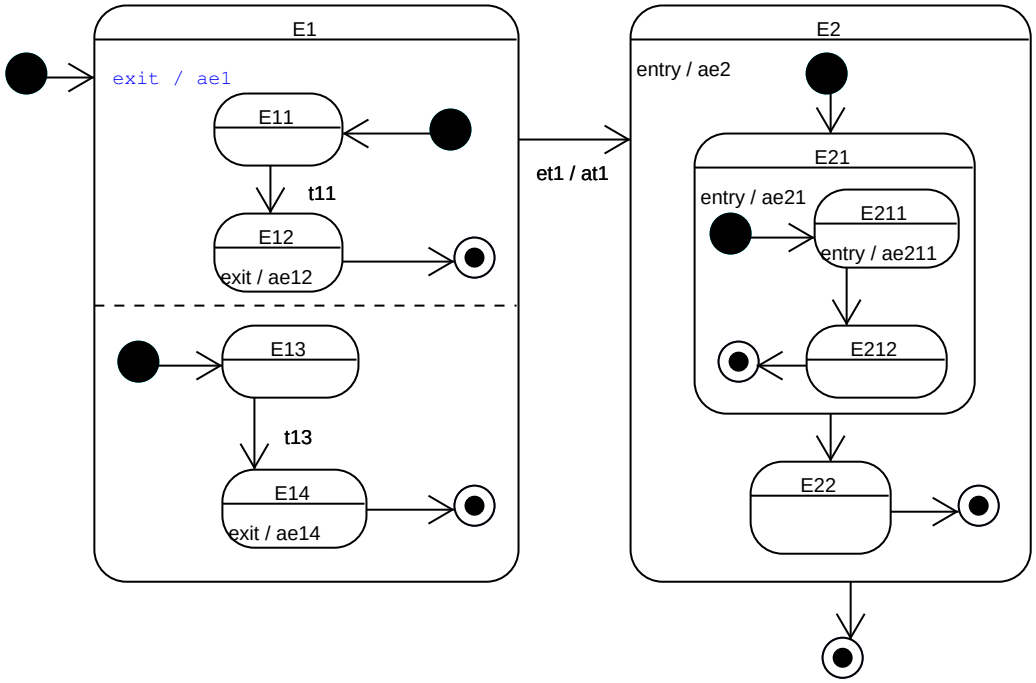

Figura 3.27: Máquina de estados con estados concurrentes y eventos.

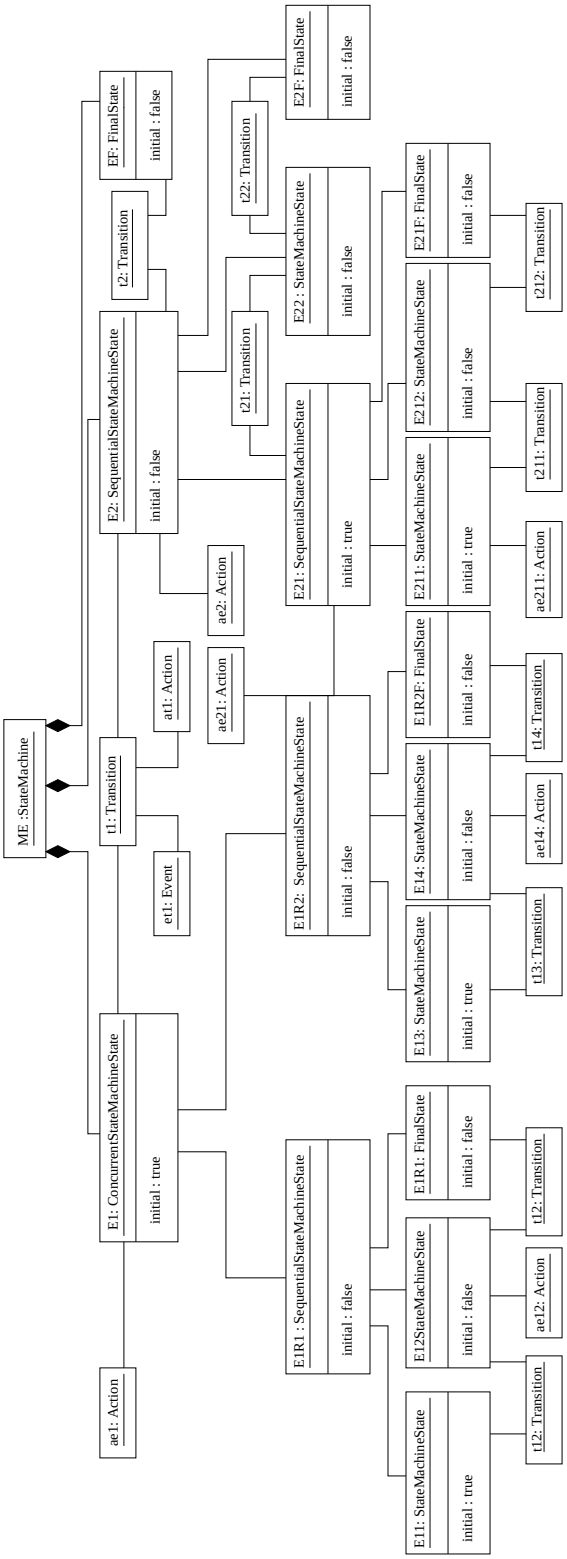

Figura 3.28: Diagrama de objetos de la sintaxis abstracta de la máquina de estados de la figura 3.27.

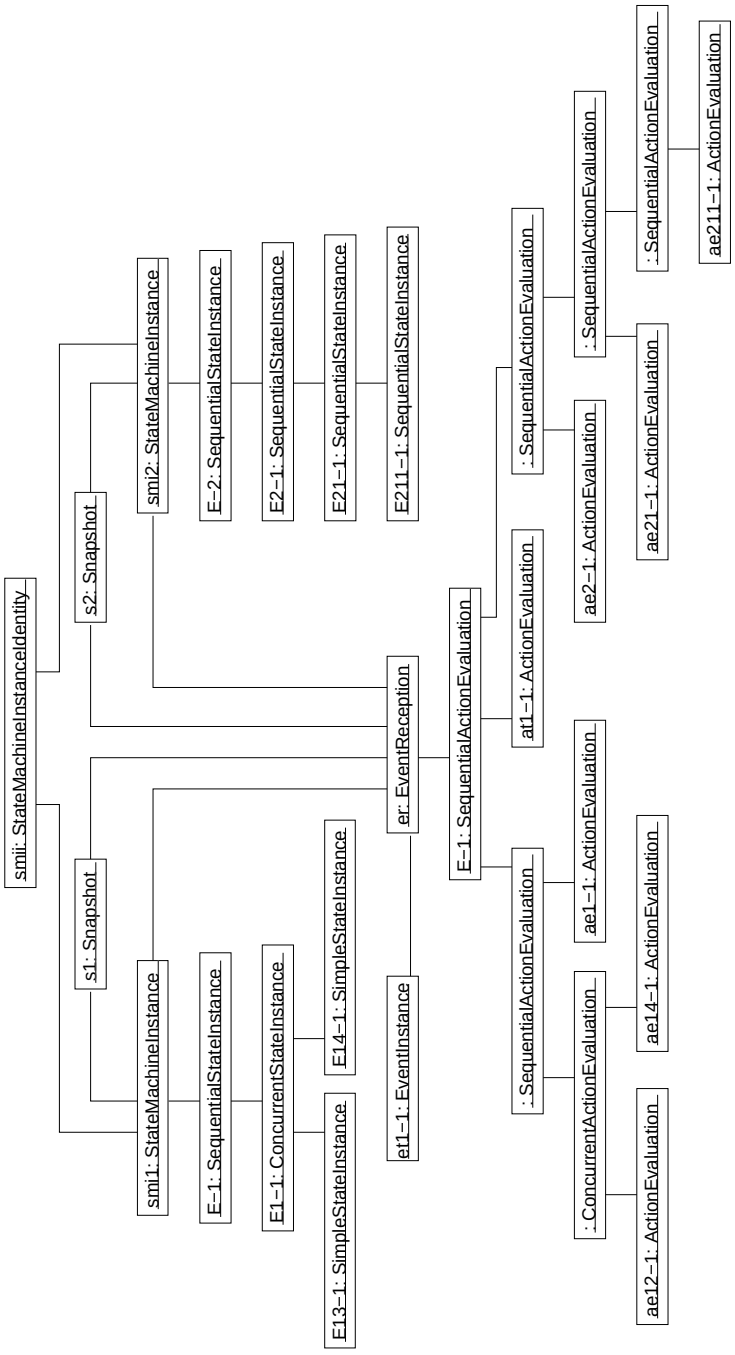

Figura 3.29: Diagrama de objetos del dominio semántico de la máquina de estados de la figura 3.27.

3. El tiempo real en las máquinas de estados

3.1. Perfil de Planificabilidad, Rendimiento y Especificación del Tiempo

El Perfil UML de Planificabilidad, Rendimiento y Especificación del Tiempo (*UML Profile for Schedulability, Performance, and Time Specification*, [56]) pretende extender UML para que se pueda usar para la especificación y diseño de sistemas de tiempo real en toda su extensión y diversidad. El grupo de trabajo que ha realizado este perfil afirma que las extensiones presentes en UML son suficientes para llevar a cabo la tarea.

Los autores de este perfil han intentado hacer un marco lo suficientemente amplio como para que cubra todas las distintas características de los sistemas de tiempo real pero que, a la vez, sea lo bastante flexible como para permitir futuras especializaciones.

Como indican los autores, no han pretendido desarrollar nuevas técnicas de análisis, sino anotar los modelos UML de forma que las técnicas de análisis, tanto las actuales como las que se puedan desarrollar, puedan aprovecharse de las ventajas del modelado en UML.

Entre los principios que han guiado la construcción de este perfil se cita la posibilidad de construir herramientas que, de manera automática, construyan modelos para distintos tipos de análisis concretos a partir de un modelo dado. Estas herramientas deberían poder leer el modelo, procesarlo y realimentar con sus resultados el modelo original.

A la hora de establecer cómo se ha de usar el perfil, los autores diferencian tres actores:

- El modelador, que construye modelos y está interesado en analizarlos para saber si cumplen los requisitos requeridos.

- El proveedor de métodos de análisis de modelos, que define métodos de análisis y herramientas para llevarlos a cabo.
- El proveedor de infraestructura, que proporcionan tecnologías como sistemas operativos de tiempo real o bibliotecas de componentes de tiempo real.

De entre los posibles usos que los autores esperan para el perfil, los que están pensados para los modeladores son:

- Sintetizar el modelo, que incluirá los estereotipos y valores etiquetados necesarios para llevar a cabo los análisis necesarios.
- Analizar el modelo, posiblemente desde varios puntos de vista y a diferentes niveles de detalle. Para el análisis se usarán la anotaciones añadidas al modelo UML.
- Implementar el sistema, que consiste en construir la aplicación que ejecute el modelo creado anteriormente.

Estructura

Debido a la gran diversidad de posibles sistemas de tiempo real, *hard* y *soft*, distribuidos y monoprocesadores, etc., es muy difícil establecer un conjunto canónico de conceptos de modelado y diseño que comprenda todos los sistemas.

Esta situación se agrava cuando se tienen en cuenta, además, las técnicas de análisis. A la hora de analizar un modelo, el analizador suele trabajar con una versión simplificada del sistema, del que se eliminan aquellas características que no son relevantes para el análisis. Una de las cuestiones que complica esta simplificación es el hecho de que es bastante usual el que las entidades

del modelo no se correspondan una a una con las entidades que es necesario conocer para poder llevar a cabo el análisis. Por ejemplo, las unidades de concurrencia en las que se base el modelo, procesos o hebras, pueden estar definidas sólo de forma implícita u ocultas en el modelo del sistema.

Otra dificultad añadida viene dada por el hecho de que sobre un mismo modelo de un sistema se pueden hacer diferentes análisis, cada uno de los cuales se puede centrar en un aspecto concreto del sistema como, por ejemplo, su planificabilidad o el consumo de memoria. Aspectos relevantes para un determinado análisis pueden no ser interesantes para otro.

Para superar ese problema los autores del perfil proporcionan un marco único que comprende los elementos comunes de los distintos métodos de análisis de sistemas de tiempo real. El núcleo de este marco lo constituye la definición de recursos de todo tipo y de sus calidades de servicio. Esta definición se hace en el paquete de Modelado de Recursos Generales. A partir de estas definiciones básicas, para cada método de análisis hay que definir los conceptos particulares que se necesiten. Estos conceptos se definirán como especialización de los ya definidos en los módulos de recursos generales.

Los modelos conceptuales se representan como paquetes y diagramas de clases UML en los que las clases representan conceptos del dominio, mientras que el perfil UML consta de estereotipos UML, que se corresponden con los conceptos de dominio. En un hipotético modelo de análisis, los conceptos propios de ese modelo serían paquetes y clases que especializarían los conceptos del Modelo de Recursos Generales. En el perfil UML habría clases estereotipadas que corresponderían a la especialización de elementos más generales del perfil. Entre los conceptos del modelo particular y las correspondientes clases del perfil de UML se establecería la relación adecuada.

No todos los conceptos de dominio tienen un estereotipo o un valor eti-

quetado correspondiente, porque algunos de esos conceptos son demasiado abstractos y nunca se van a usar directamente en ningún modelo de análisis, sino que algunos modelos usarán una especialización de ese concepto, y será para el concepto especializado para el que se defina el estereotipo o valor etiquetado correspondiente en UML.

Como ya se ha comentado anteriormente, los autores del perfil argumentan que los mecanismos de extensión ofrecidos por UML, estereotipos y valores etiquetados, son suficientes para la tarea encomendada. Ellos ven los estereotipos como un mecanismo de especialización de clases que permite añadir nuevos atributos a la clase especializada. Sin embargo, este mecanismo de estereotipado no permite agregar nuevas asociaciones en el metamodelo respecto a las que ya tenía la clase especializada. Las nuevas asociaciones que establezcan los elementos de modelo se pueden traducir de tres formas distintas:

- Algunas de estas asociaciones se corresponden exactamente con otra asociación ya existente en el metamodelo de UML.
- Otras asociaciones se corresponden con etiquetas asociadas con el estereotipo.
- En unos pocos casos, una asociación de dominio se representa mediante una relación del tipo ≪ taggedValue ≫ de los mecanismo de perfiles de UML.

Otro de los objetivos del perfil ha sido permitir más especializaciones de modelos de dominio concreto cuando hagan falta. Como aplicación ejemplo del perfil, los autores han desarrollado el subperfil de CORBA de tiempo real como especialización del perfil de análisis de planificabilidad.

Modelado de recursos

La parte fundamental del perfil es el paquete en el que se definen los recursos. Han intentado definir recursos suficientemente generales como para ser una base común suficiente para poder definir los demás recursos y conceptos propios de cada modelo particular que se defina posteriormente.

La cuantificación de las restricciones que afectan a los sistemas de tiempo real es uno de los aspectos fundamentales que se tratan en el perfil y, por tanto, las características de calidad de servicio son una parte integral de los recursos. Estas características de calidad de servicio tienen sentido en un contexto en el que se pueda indicar, además de la calidad de servicio ofrecida por un recurso, la calidad de servicio requerida por los elementos del sistema que actúan como clientes de dicho recurso. El análisis consistirá, en ese caso, en comparar ambas capacidades.

En el perfil han trabajado con lo que denominan dos *vistas* respecto a los recursos y los clientes. En la primera forma, los recursos y los clientes se encuentran en el mismo nivel de abstracción, por lo que la relación es *entre iguales* y se puede indicar mediante una asociación. La otra forma de uso de recursos es una interpretación jerarquizada, en la que el cliente está en un nivel y el recurso está en otro, generalmente para modelar una situación en la que el recurso implementa un servicio ofrecido por el cliente.

Modelado del tiempo

El modelado del tiempo se trata de varias formas en el perfil. Por un lado, se modela el tiempo como magnitud, para medir las restricciones. Sólo se define el tiempo métrico, ya que los autores argumentan que, habitualmente, los sistemas de tiempo real se ocupan de la cardinalidad del tiempo, como en el caso del cumplimiento de los plazos temporales. Encuentran útil distinguir

entre tiempo físico y tiempo virtual, o de simulación, que puede incrementarse de manera no monótona.

Un segundo aspecto que se modela sobre el tiempo son los mecanismos que se ofrecen al sistema, por ejemplo por parte de un sistema operativo de tiempo real, para tratar con él, como los temporizadores y los relojes.

Finalmente, también se hace referencia a los diferentes patrones que son esenciales en la planificabilidad o los distintos análisis, como la periodicidad, o la definición de plazos e intervalos temporales.

Modelado de la planificabilidad

El perfil se centra en la especificación de sistemas de tiempo real *duros*, es decir, aquellos en los que se han de cumplir todos los plazos, para lo que suele ser necesario poder establecer un límite superior en el tiempo de respuesta del sistema a los diferentes eventos. El objetivo principal del perfil en este aspecto ha sido cómo anotar el modelo para permitir diferentes modelos de planificabilidad, para determinar si se cumplen o no los plazos temporales.

Para evitar conflictos debidos a la nomenclatura, se han decidido por un marco común a todos los posibles análisis de planificabilidad en los que, en un principio, han incluido los más usados actualmente, como *Rate Monotonic Analysis (RMA)*, *Deadline Monotonic Analysis (DMA)* y *Earliest Deadline First (EDF)*. Sin embargo, los autores defienden que sus anotaciones son suficientemente flexibles como para poder usar otros métodos e, incluso, para poder definir extensiones propias si son necesarias para otros métodos de análisis.

Modelado del rendimiento

Según los autores del perfil, el análisis del rendimiento tiene muchas características comunes con el análisis de planificabilidad, por lo que comparten

como base común el Modelo de Recursos Generales. Consideran que el análisis del rendimiento es más adecuado para sistemas de tiempo real no estrictos porque se basa en técnicas estadísticas, como teoría de colas, que producen, asimismo, resultados estadísticos.

Por tanto, el modelo que proponen es mínimo, con la esperanza de que los desarrolladores de herramientas especialicen nuevos subperfiles concretos para análisis particulares. Aún así, afirman que lo que ofrecen es suficiente para hacer un análisis básico de rendimiento incluso de sistemas complejos.

Estructura del perfil

Al perfil se le ha dado una estructura modular para permitir que los usuarios no tengan que considerar todo el perfil en su conjunto, sino que puedan trabajar sólo con una parte de él. Esta división también está pensada para facilitar su especialización.

El perfil está dividido en una serie de subperfiles (figura 3.30), organizados en paquetes, dedicados a aspectos concretos y a técnicas de análisis de modelos. El núcleo del perfil es el conjunto de paquetes que representa el marco de modelado de recursos generales. A su vez, el modelo de recursos generales está dividido en varias partes, con la idea de que las posteriores especializaciones sólo tengan que usar la parte concreta que necesiten.

El paquete principal, el del Marco de Modelado de Recursos Generales, *General Resource Modeling Framework*, se divide en tres subpaquetes, de los que el principal es el de modelado de recursos, que se ha hecho de tal manera que sea lo más independiente posible de los modelos de concurrencia y tiempo que se definan. Tanto la concurrencia como el tiempo se definen en sus propios subpaquetes dentro del paquete principal.

El perfil define también un segundo paquete en el que se incluyen tres

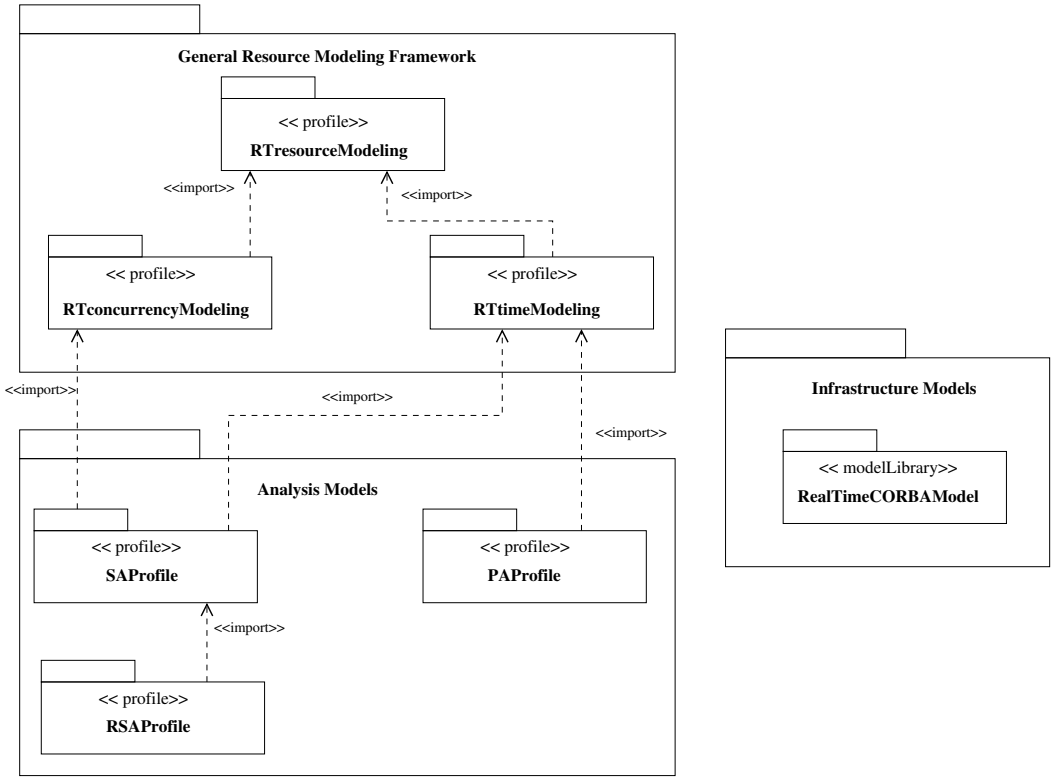

Figura 3.30: Estructura de paquetes del perfil de Planificabilidad, Rendimiento y Especificación de Tiempo.

tipos de análisis, el análisis de planificabilidad (*SAProfile*), el de rendimiento (*PAProfile*) y uno adicional que es una especialización del de planificabilidad, (*RSAProfile*), para analizar la planificabilidad de sistemas CORBA de tiempo real.

Junto con esos paquetes se ha definido un modelo de biblioteca con la definición de un modelo de CORBA de tiempo real. Este modelo pretende ser un ejemplo que avale el trabajo hecho en el perfil.

El paquete de modelado de recursos generales

El paquete de modelado de recursos generales, *General Resource Modeling Framework* (*GRM*), es la parte fundamental del perfil, en la que se definen

los fundamentos necesarios para el análisis cuantitativo de modelos UML. La noción básica del modelo es la de *calidad de servicio*, que proporciona una base uniforme para adjuntar información cuantitativa a los modelos UML. La información sobre calidad de servicio representa, de manera directa o indirecta, las propiedades físicas de los entornos *hardware* y *software* de la aplicación representada en el modelo.

Los diferentes aspectos del *GRM* se agrupan en paquetes individuales. La estructura interna del paquete *GRM* se muestra en la figura 3.31.

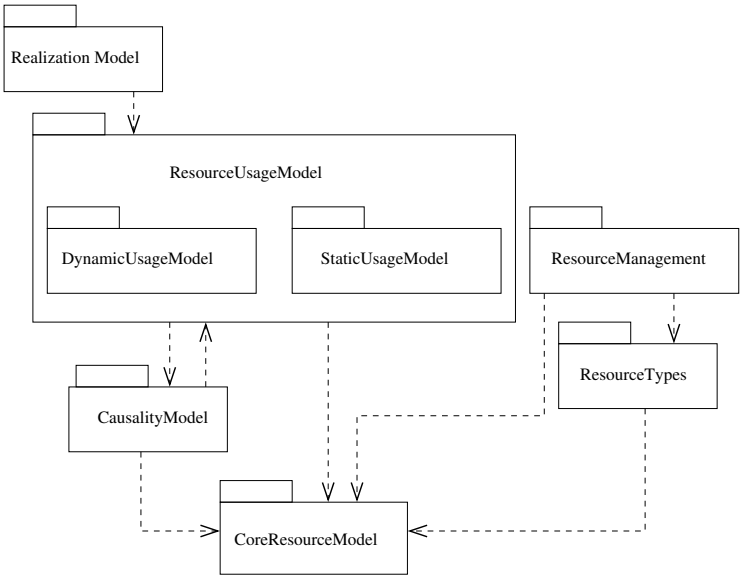

Figura 3.31: Estructura de paquetes del Marco de Modelado de Recursos Generales.

Los autores aclaran que para los propósitos de este perfil es fundamental distinguir entre elementos *descriptores*, como las clases y los tipos, y elementos *instancias de tiempo de ejecución*, como los objetos y variables, que se crean tomando como base los elementos descriptores. Esto es así porque prácticamente todos los análisis que se hacen en los sistemas de tiempo real tienen sentido sobre instancias concretas y sus asociaciones. Por tanto, es más real hacer los análisis sobre los diagramas de objetos que sobre los diagramas

de clases. Sin embargo, a veces es útil asociar características de calidad de servicio a descriptores, como ocurre en aquellas situaciones en las que todas las instancias de un servidor tienen el mismo valor para ese parámetro de calidad de servicio.

El paquete *Core Resource Model* define dos conjuntos de clases relacionados. Por un lado, las clases descriptoras y, por otro las clases de instancias. Por cada clase descriptora se define una clase de instancia entre la que se establece una relación de uno a muchos con los papeles *tipo* (`type`) e *instancia* (`instance`).

La clase principal en el grupo de los descriptores es la clase `Descriptor`, que representa un concepto muy general para cualquier elemento de diseño. La clase `Resource` es una especialización de `Descriptor`, y representa un recurso de un sistema de tiempo real. La clase `ResourceService` representa un servicio de los varios que puede ofrecer un recurso. Entre ambas clases se establece una relación de composición que representa esta idea. La clase `QoScharacteristic` representa la caracterización cuantitativa de un servicio ofrecido por un recurso. Entre las clases `ResourceService` y `QoScharacteristic` existe una asociación de muchos a muchos. Hay una asociación adicional de muchos a muchos entre las clases `ResourceService` y `QoScharacteristic` que, según los autores, puede usarse para simplificar el diseño del sistema cuando un recurso ofrezca claramente un único servicio.

El paquete *CausalityModel* es la base del funcionamiento dinámico del perfil. Modela la relación causa-efecto entre los eventos ocurridos en el sistema y las acciones que se ejecutan como respuesta.

Uno de los conceptos fundamentales en el modelo es el de ocurrencia de un evento, que corresponde a una instancia de la noción de evento de UML. De entre los diferentes tipos de eventos, los más interesantes para el propósito del

perfil son la *generación de estímulos* y la *recepción de estímulos*. Un estímulo es una instancia de una comunicación entre dos objetos. Cuando un objeto, el llamante, ejecuta una acción que involucra una comunicación con otro objeto, se crea una instancia de *generación de estímulos*. Cuando el estímulo llega al objeto llamado se crea una instancia de *recepción de estímulos*. Esta secuencia de comunicación genera, a su vez, la ejecución de un *escenario*, en el que, posiblemente, se ejecuten una secuencia de acciones como respuesta al estímulo, que puede incluir otros estímulos.

En el diagrama de clases del modelo, se incluye la clase *Estímulo*, (`Stimulus`), que tiene una asociación con la clase `StimulusGeneration` y con la clase `StimulusReception`. También aparecen las clase `Scenario`, e `Instance`. Esta instancia tiene un papel doble. Por un lado es el objeto en el que se ejecuta el escenario, y por otro es el receptor del estímulo.

El paquete *Resource Usage Model* representa el lado del cliente de la relación con un recurso. El uso de un recurso es un patrón que describe cómo un conjunto de clientes usa un conjunto de recursos y sus servicios. Este patrón se corresponde de una manera muy aproximada a la idea de caso de uso. Dependiendo del caso concreto, se distinguen dos tipo de uso: el *estático*, en el que sólo interesa conocer la relación estática entre los clientes y los servicios que usan de los recursos, y el *dinámico*, en el que se tienen en cuenta también aspectos temporales del uso, como el orden o la temporización de las acciones.

En este modelo se separa el uso de un recurso del evento que lo ha producido. Esto permite definir patrones de uso independientemente de los factores externos que han llevado a él. Para ayudar a determinar qué parte del modelo se va a analizar, se introduce el concepto de *contexto de análisis*, que consiste en un conjunto de usos de recursos con sus correspondientes densidades y las

instancias de recursos usadas.

El paquete *StaticUsageModel* se usa para representar una vista estática del uso de los recursos, aunque el uso en sí no sea un proceso estático. En ese uso, se asocia al cliente un conjunto de valores deseados de calidad de servicio por cada uno de los servicios de los recursos usados. Estos valores deseados de calidad de servicio se pueden contrastar con los valores ofrecidos para los mismos servicios por los recursos usados para dilucidar si las peticiones pueden satisfacerse.

El paquete *DynamicUsageModel* se usa para mostrar aquellas situaciones de uso de recursos en las que el orden y el instante del uso de los servicios de los recursos es relevante para el análisis. En ese caso, se usan instancias de escenarios, a los que hay asociados secuencias de ejecuciones de acciones, bien concurrentes, bien secuenciales. Las acciones se pueden representar a diferentes niveles de detalle y una acción compleja se puede dividir en acciones más simples.

El paquete *ResourceTypes* ofrece una colección de diferentes clases de recursos que se pueden clasificar en base a diferentes categorías. En base a su propósito se distinguen:

- *recursos procesadores*, que representan elementos físicos o virtuales con capacidad de cálculo y ejecución de instrucciones,
- *recursos de comunicación*, cuyo propósito principal es permitir la comunicación entre otros recursos y
- *dispositivos*, que representan al resto de recursos.

En base a su actividad, los recursos se clasifican en:

- *activos*, que son capaces de crear estímulos sin necesidad de haber recibido un estímulo externo previo, y

- *pasivos*, que sólo ejecutan acciones como respuesta a la recepción de un estímulo desde otro recurso.

En base a su protección, los recursos pueden ser:

- *protegidos*, que son los recursos que cuentan con una política de control de acceso para garantizar un uso exclusivo y correcto, y
- *no protegidos*, que son los que no cuentan con ninguna protección frente al uso simultáneo por parte de varios clientes.

Para acceder a un recurso protegido, los clientes han de efectuar en primer lugar una operación de adquisición del recurso, que puede ser de naturaleza bloqueante o no bloqueante. Cuando el cliente acaba de trabajar con el recurso, debe ejecutar una acción de liberación, de naturaleza no bloqueante.

El paquete *ResourceManagement* define dos funciones concernientes al manejo de los recursos, el *resource manager*, o *habilitador de recursos*, que se encarga de crear y destruir recursos en función de las peticiones de uso de los clientes, y el *resource broker*, *gestor de recursos*, cuya función es la de gestionar el acceso a los recursos por parte de los clientes.

El último subpaquete del paquete de modelado general de recursos es *RealizationModel*. En este paquete la realización es definida como la habilidad de desarrollar un recurso de un nivel superior mediante recursos de un nivel inferior, ya sea describiendo un recurso en base a otros de mayor nivel de detalle o describiendo un recurso *software* en base a recursos *hardware*. La herramienta básica para la aplicación de la realización es la construcción de sistemas por niveles. Uno de los casos particulares de esta realización es el refinamiento, del que señalan como ejemplo la especificación ISO RM-ODP, en la que en niveles superiores se encuentran elementos *software* completamente independientes de la tecnología concreta usada, que se especifica en

los niveles inferiores. Para ilustrar la realización como la construcción de un recurso en base a otros de niveles inferiores se señala el modelo OSI de siete capas para el desarrollo de sistemas de comunicación.

Para los autores del perfil, esta noción de realización es similar al concepto de *deployment* en UML: hay una aplicación de elementos de un nivel superior, usualmente software, a elementos de un nivel inferior, posiblemente un elemento del modelo del entorno. La relación de realización es similar a la que se establece entre un cliente y un recurso en el modelo general de recursos, siendo el elemento de mayor nivel de abstracción el que corresponde al cliente, y el elemento del nivel de abstracción inferior, que realiza al otro, el que corresponde al recurso.

El punto de vista UML del Paquete de Modelado General de Recursos

La forma habitual en que se introduce un nuevo concepto de modelado en una extensión UML es mediante un estereotipo. Este estereotipo se aplica a elementos del modelo UML, de tal manera que puedan ser reconocidos por herramientas automáticas de análisis o por personal especializado. Sin embargo, como en el paquete de modelado general de recursos la mayoría de los conceptos que se definen son abstractos y se especializarán con conceptos más concretos en diferentes subperfiles, no hay muchos estereotipos definidos para los conceptos de este paquete.

Uno de los conceptos para el que sí se define un estereotipo en UML es la realización, que se define como un tipo especial de la relación de abstracción, y para el que se define el estereotipo «*realize*». En el perfil se definen tres posibles especializaciones de esa relación:

- «*code*». Esta relación se usa con un elemento que contiene físicamente el código ejecutable de otro elemento.

- «*deploys*». Una relación más general que indica que las instancias del proveedor están contenidas en el cliente.

- «*requires*». Una especialización de la relación anterior e indica que el cliente es una especificación genérica del mínimo entorno de despliegue requerido por el proveedor.

El modelado general del tiempo

En el paquete *General Time Modeling* se definen los conceptos relacionados con la definición del tiempo y los mecanismos para manejarlos, como relojes y temporizadores. La definición se restringe al llamado *tiempo métrico* y no se incluyen modelos de tiempo lógico.

El diagrama de subpaquetes de este paquete consta de cuatro elementos en los que se definen, respectivamente, los conceptos para modelar el tiempo y los valores de tiempo, los conceptos para modelar eventos en el tiempo y estímulos relacionados con el tiempo, los conceptos para modelar mecanismos temporales (relojes y temporizadores) y los conceptos para modelar servicios temporales, como los que ofrecen los sistemas operativos de tiempo real.

El modelo de tiempo que se incluye en el perfil es un tiempo físico que se compone de una progresión continua y no acotada de instantes de tiempo físico que componen un conjunto totalmente ordenado y denso. La primera de estas dos características implica que los eventos están parcialmente ordenados y la segunda que el tiempo es denso. Como, habitualmente, los mecanismos de medición del tiempo tienen una precisión limitada, también se incluye el concepto de tiempo discreto.

La medida del tiempo se hace a través de la cuenta del número de ciclos expirados de un *reloj de referencia*, estrictamente periódico. Esta medida conlleva la discretización del tiempo. La cuenta de un instante de tiempo

particular, o su medida, se representa por un valor especial llamado *valor de tiempo*. El valor se podrá implementar mediante un número natural, real o cualquier otra estructura más compleja que sea necesaria.

Otros dos conceptos que aparecen en este subpaquete son el de *duración*, que es un valor temporal que representa el tiempo transcurrido entre dos instantes, y que también se representa con un valor temporal, y el de *intervalo*, compuesto por dos valores temporales, que representa un espacio de tiempo entre dichos instantes temporales.

Los mecanismos de manejo del tiempo que se definen en el perfil son elementos que miden el paso del tiempo y generan como resultado algún tipo de evento. Dichos mecanismos son una especialización del concepto de recurso definido en el paquete de modelado general de recursos.

Dos son los mecanismos que se definen: los *temporizadores* y los *relojes*. Los temporizadores generan instancias del evento *timeout* bien cuando se alcanza un determinado instante de tiempo o bien cuando ha transcurrido un intervalo de tiempo desde el instante en el que se activó el temporizador. Los relojes generan periódicamente instancias del evento *clock tick*. También pueden generar instancias del evento *interrupción del reloj*.

Todos los dispositivos de medida del tiempo tienen una serie de características, como el valor actual, su valor de referencia, el origen temporal, el valor temporal máximo, la resolución, etc. Por otro lado, ofrecen operaciones como establecer el tiempo, consultar el tiempo, reiniciar los servicios, y parar y reanudar el servicio.

En el modelo definido en el perfil, como en UML, las instancias de eventos son instantáneas. Es decir, ocurren en un instante de tiempo dado y no tienen duración. Para poder hacer el análisis temporal del sistema es necesario situar los eventos en el tiempo, por lo que se definen los *eventos temporales*. A cada

instancia de un evento temporal se le asocia un valor temporal relativo a un reloj, que denota el momento en el que ocurre dicha instancia.

Como se define en el paquete de modelado general de recursos, la ocurrencia de una instancia de un evento puede llevar aparejada la ocurrencia de una secuencia de estímulos. Para poder localizar estos estímulos en el tiempo, en el perfil se define el *estímulo temporal*, que lleva asociado dos valores temporales, el de inicio y, posiblemente, el de finalización.

También se definen las *acciones temporales*, a las que se les asocian sus propias marcas temporales, por un lado, un intervalo temporal, la duración de la acción, y, por otro, dos valores temporales, que corresponden a los instantes de ocurrencia de los eventos de inicio y fin, respectivamente, de la acción.

Finalmente, en este paquete se define un servicio temporal que funciona como un generador de relojes y temporizadores, pensado para facilitar el modelado de los servicios temporales ofrecidos por los sistemas operativos de tiempo real.

Punto de vista UML del modelado general del tiempo

En UML se definen los conceptos de `TimeExpression` y `TimeEvent`, para representar, respectivamente, valores de tiempo que se usan en tiempo de ejecución y ocurrencia de eventos, también en tiempo de ejecución. Los autores del perfil razonan que para expresar valores temporales en la fase de análisis, a veces es necesario establecer una distribución probabilística de ocurrencia de esos eventos por lo que el concepto `TimeExpression` de UML no es suficiente para modelar el concepto de valor de tiempo que pretende establecer el perfil. El mismo razonamiento se puede dar para el concepto de `TimeEvent`.

El tiempo físico no es definido en UML, sino sólo la medición de éste. Para especificar valores de tiempo se proponen dos maneras: un estereotipo, «*RTtime*», aplicable a los valores que representen valores de tiempo, y, en segundo lugar, usar instancias del tipo *RTtimeValue*, definido también en el perfil. Este tipo se basa en etiquetas de UML (*tags*) y describe diferentes formas de expresar valores temporales en base a distintas unidades y categorías. Los intervalos de tiempo, al ser formas especiales de valores temporales, también pueden especificarse de manera similar.

Los mecanismos de manejo de tiempo se definen mediante diversos estereotipos, como «*RTtimingMechanism*», «*RTclock*» o «*RTtimer*». Sus diferentes características se establecen mediante valores etiquetados, como *RTstability*, *RTresolution*, etc. Para las operaciones sobre esos mecanismos también se definen los correspondientes estereotipos.

Asimismo, para las acciones temporales también se añade otro estereotipo, *RTaction*, cuyas características temporales se especifican mediante valores etiquetados, bien indicando su duración, bien indicando los valores temporales de los eventos de inicio y fin.

Para modelar los eventos temporales, se define el estereotipo «*RTstimulus*», que se aplica a elementos UML que generen estímulos, como estímulos en sí o ciertas acciones: llamadas a funciones, envío de señales, etc. Dos estímulos concretos son los generados por los relojes y los temporizadores, caracterizados por los estereotipos «*RTclkInterrup*» y «*RTclktimeout*».

Modelado general de concurrencia

El paquete *General Concurrency Modeling* es una extensión del paquete de causalidad del modelado general de recursos. Se amplia el concepto de escenario como secuencia de ejecución de acciones como respuesta al envío

y recepción de estímulos. Cada objeto del sistema susceptible de iniciar una secuencia de ejecuciones se denomina una *unidad de concurrencia*.

Cuando se crea una unidad de concurrencia, ésta inicia un escenario correspondiente a su método principal. Este escenario puede incluir generación de estímulos, uso de servicios, etc. La ejecución de los escenarios de las unidades de concurrencia se ejecutan en paralelo.

A las unidades de concurrencia se les asocia colas donde almacenar temporalmente los estímulos que les llegan cuando aquéllas no están disponibles por estar procesando un estímulo previo. Desde el punto de vista del servidor, las peticiones se pueden atender de manera inmediata o postergada. Desde el punto de vista del receptor, las peticiones pueden ser síncronas o asíncronas.

Punto de vista UML del modelado general de concurrencia

Una entidad que se ejecute concurrentemente se describe con el estereotipo «*CRconcurrent*». Los estereotipos «*CRasynch*», «*CRsynch*», «*CRimmediate*» y «*CRdeferred*» sirven para indicar cómo se maneja un mensaje.

Modelado de planificabilidad

En el paquete *Schedulability Modeling* se describe un conjunto mínimo de anotaciones sobre conceptos de planificabilidad, con el propósito de que sea ampliable de manera sencilla para aquellos análisis que requieran un mayor grado de detalle o más información para poder realizar el análisis de planificabilidad.

Los autores hacen notar que el análisis de planificabilidad es inherentemente basado en instancias, por lo que no tiene sentido analizar diagramas con clases y asociaciones, sino diagramas de objetos con enlaces, en la que se sepa de manera concreta la cardinalidad de cada uno de los objetos.

El paquete consta de dos diagramas de clases en los que se definen, respectivamente, las clases correspondientes a los conceptos implicados en el modelado de sistemas que se quieran analizar y, por otro lado, elementos de la infraestructura necesaria para planificar procesos concurrentes, típicamente ofrecidos por el sistema operativo.

Los nuevos elementos de modelado son especializaciones de conceptos definidos en otros paquetes del perfil a los que se les han añadido atributos para indicar las propiedades que son necesarias para realizar el algoritmo de planificabilidad. Por ejemplo, la clase *SAction* es una especialización de la clase *TimedAction* del paquete *TimedEvents* con una serie de atributos nuevos para especificar las propiedades que se necesitan saber sobre una acción concreta en el análisis de planificabilidad como, por ejemplo, la prioridad, el tiempo de cálculo en el peor caso, el tiempo de retraso, el tiempo de bloqueo, etc. Otro ejemplo es la clase *SResource*, que es una especialización de la clase *ProtectedReource* del paquete *ResourceTypes* donde se añaden atributos para especificar la capacidad, el tiempo de adquisición y liberación, etc.

Punto de vista UML del modelado de la planificabilidad

La traducción a UML de los conceptos definidos para este subperfil también se hace definiendo nuevos estereotipos, «*SAaction*», «*SAengine*», «*SAowns*», «*SAresource*», «*SAschedRes*» y «*SAscheduler*» entre otros. La mayoría incorporan valores etiquetados para especificar los valores necesarios para llevar a cabo el análisis de planificabilidad.

Los autores del perfil incluyen guías y diagramas de ejemplo sobre cómo usar los conceptos definidos en este paquete. En los diagramas que se usan de ejemplo para modelar sistemas de tiempo real, los elementos se etiquetan con los estereotipos definidos anteriormente y los valores etiquetados necesarios

para indicar los factores involucrados en el análisis de planificabilidad, como la periodicidad, el plazo temporal o el tiempo de respuesta en el peor caso.

Modelado del rendimiento

Como en el caso del modelado de la planificabilidad, el paquete *Performance Modeling* proporciona un conjunto mínimo de anotaciones que permitan realizar un análisis de planificabilidad mínimo, con el propósito de que sea fácilmente extensible en aquellos casos en los que se necesite un análisis más detallado. Los conceptos incluidos en este paquete pretenden cubrir los requisitos de rendimiento que se le exigen al sistema, las características de calidad de servicio asociadas con los elementos del modelo, los parámetros de ejecución que permitan un análisis automático del rendimiento y la presentación de los resultados obtenidos a partir del análisis.

En el diagrama de clases definido en este paquete la clase central es *Performance Context*, que se refiere a uno o varios escenarios que describan diferentes situaciones dinámicas de varios recursos. Esta clase es una especialización de *AnalysisContext*. Los escenarios de este paquete se definen en la clase *PScenario*, una especialización de *Scenario* e incluyen nuevos atributos para especificar los datos necesarios para el análisis de rendimiento. Un escenario, como en el caso general, se compone de varios pasos más simples, definidos en la clase *PStep*.

Las actividades que se producen en el escenario se deben a las peticiones de uso por parte de los elementos del modelo, indicadas por instancias de la clase *Workload*, una especialización de *UsageDemand*. Estas cargas de trabajo pueden ser bien abiertas o cerradas, y cada una de ellas cuenta con sus propios atributos para especificar sus propiedades características.

También se define una nueva clase para los recursos, *PResource*, especiali-

zación de *Resource*, de cuyas instancias se puede especificar su utilización, su política de planificación y su caudal de salida. Se definen dos tipos concretos de recursos, los activos, o que producen procesamiento, *PProcessingResource*, y los pasivos, *PPassiveResource*.

Punto de vista UML del modelado del rendimiento

Debido a su naturaleza netamente dinámica, los escenarios juegan un papel central en el estudio del rendimiento. En UML, los escenarios se especifican fundamentalmente mediante de diagramas de colaboración o de actividad.

Tanto si se usan diagramas de colaboración como de actividad, se definen diferentes estereotipos para los conceptos definidos en este paquete: «*PAcontext*», «*PAstep*», «*PAhost*» y «*PAresource*».

3.2. Mecanismos de expresión del tiempo en las máquinas de estados de UML

Como se menciona en la sección 4.2 del capítulo 1, los únicos mecanismos relativos al manejo del tiempo en UML son los eventos de tiempo, `TimeEvent`, y las expresiones de tiempo, `TimeExpression`.

Un evento de tiempo modela el cumplimiento de un plazo temporal que se especifica mediante una expresión de tiempo. Las expresiones de tiempo no tienen un formato bien definido y se deja total libertad para su especificación. El tiempo expresado puede ser absoluto, una cantidad concreta de tiempo, o relativo, una cantidad de tiempo desde un determinado evento, que habitualmente es el instante de entrada a un estado.

Esta definición del tiempo es tremendamente imprecisa. No se hace mención a ningún reloj de referencia, ni a la posibilidad de múltiples relojes, ni se especifica cuáles son las unidades de tiempo, o si se puede modelar la

precisión de los relojes, o algunas otras de sus características.

Burns en [28] y Diethers *et al* en [36], modelan las máquinas de estados de UML mediante autómatas con tiempo. En estas propuestas los autómatas con tiempo disponen de un conjunto de relojes asociados; cada reloj se puede reiniciar a 0, avanza monótonamente y se puede consultar su valor para establecer condiciones que se basen en ellos. En ambos trabajos se aprovechan estas propiedades para hacer análisis de planificabilidad. Para modelar el tiempo de ejecución de las tareas llevadas a cabo en un estado, se reinicia a 0 un reloj cuando se entra al estado, se establece como invariante del estado que el valor del reloj es menor o igual que un tiempo y la condición de salida del estado se define como que el valor del reloj llegue al tiempo referenciado en el invariante del estado.

3.3. Características del entorno de tiempo

Álvarez *et al* en [14], describen el tiempo de ejecución de una acción de SDL asociando una etiqueta al icono de la acción, en vez de hacerlo de manera implícita, como los autores mencionados en la sección anterior. Con esa etiqueta, se puede analizar la estructura del sistema especificado para construir una red de tareas sobre la que hacer el análisis de planificabilidad. En ese caso hay que especificarlo como un comentario adicional porque no se tiene acceso al metamodelo de SDL y no se pueden agregar nuevas características a los elementos de SDL.

Para incluir el manejo del tiempo necesario para el análisis de planificabilidad que queremos hacer, nosotros vamos a seguir una estrategia similar a la seguida en [14] para SDL, aunque adaptándolo en función de las diferencias entre SDL y las máquinas de estados de UML. Una primera diferencia fundamental es que, en nuestro caso, en el que hemos definido un metamodelo

de las acciones y de las máquinas de estados, se puede añadir en ese metamodelo la información sobre el tiempo de ejecución y no hace falta hacer uso de comentarios adicionales.

Otra parte de la información necesaria para poder efectuar el análisis de planificabilidad es la relativa a las características temporales del entorno de las máquinas de estados. En ese entorno se incluyen, entre otras cosas, los mecanismos de manipulación del tiempo. Como se ha explicado en la sección 4.1 del capítulo 1, SDL incluye una definición precisa de los mecanismos que proporciona para el manejo del tiempo. En un sistema monoprocesador hay un reloj central que avanza monótonamente y cuyo valor puede ser consultado con la operación `now()`, que devuelve un valor de un tipo previamente definido, `TIME`, y los temporizadores, que son objetos que se pueden definir en un proceso, arrancar con la operación `SET` y reiniciar con la operación `RESET`. Los temporizadores generan un evento cuando llegan al final de su cuenta sin haber sido reiniciados.

Para incluir el entorno temporal necesario en las máquinas de estados, vamos a usar mecanismos temporales similares a los definidos en el *UML Profile for Schedulability, Performance, and Time Specification* [56].

Por tanto, vamos a definir los paquetes y los elementos necesarios usando el esquema que se ha seguido para definir las acciones y las máquinas de estados: por cada paquete se definirá un subpaquete con la sintaxis abstracta, otro con el dominio semántico y otro con la aplicación semántica.

En el paquete de recursos temporales generales vamos a incluir los relojes y los temporizadores. Los relojes ofrecen una operación `now()` que permite consultar el valor actual. Los temporizadores ofrecen operaciones para iniciarlos, `set`, y para pararlos, `reset`. Si un temporizador llega al final del tiempo establecido tras ser iniciado y sin haber sido parado, genera una instancia de

un evento temporal (figuras 3.32, 3.33 y 3.34).

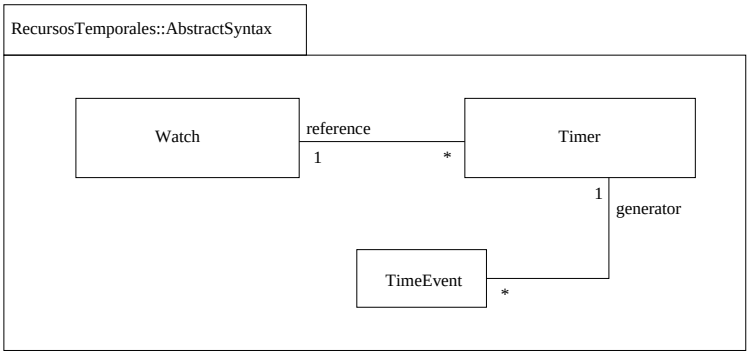

Figura 3.32: Sintaxis abstracta de los recursos temporales.

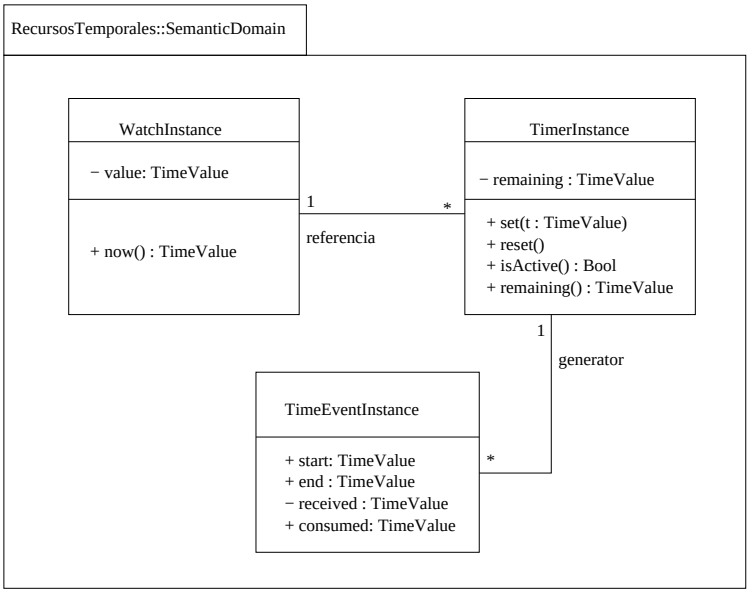

Figura 3.33: Dominio semántico de los recursos temporales.

Otro elemento que se debe introducir para tener la información necesaria para el análisis de planificabilidad es el tiempo de cálculo de una acción, el conocido como WCET (*Worst Case Execution Time*). Este dato se usa en métodos de análisis como el del cálculo del nivel de ocupación del procesador o el del cálculo del tiempo de respuesta. Para añadir esa información a las

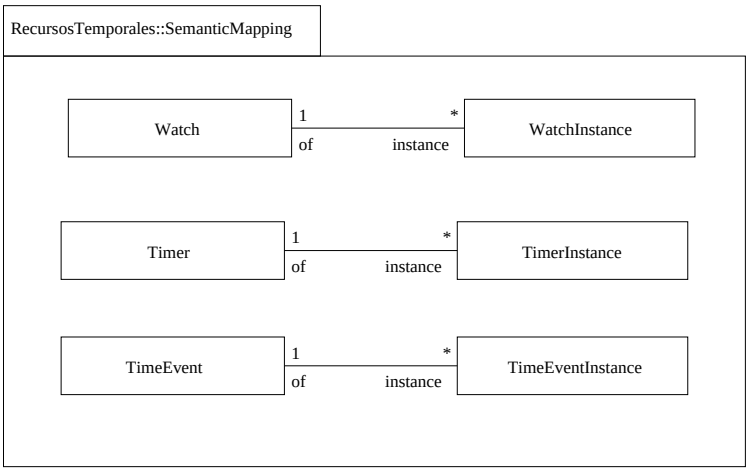

Figura 3.34: Aplicación semántica de los recursos temporales.

acciones vamos a definir dos nuevas clases a partir de las clases definidas en el capítulo 2, *TimedAction* y *TimedExecution*. *TimedAction* tiene asociado un valor de tiempo, el *wcet*, el tiempo de respuesta en el peor caso. *TimedExecution* tiene asociado otro valor de tiempo, *execTime*, que corresponde al tiempo que ha tardado dicha ejecución concreta de la acción.

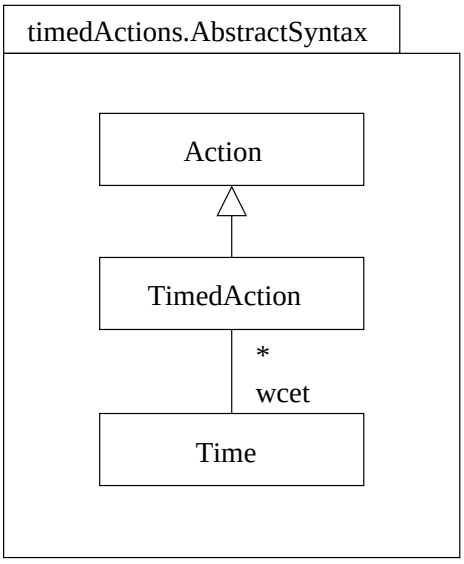

Figura 3.35: Sintaxis abstracta de las acciones con tiempo.

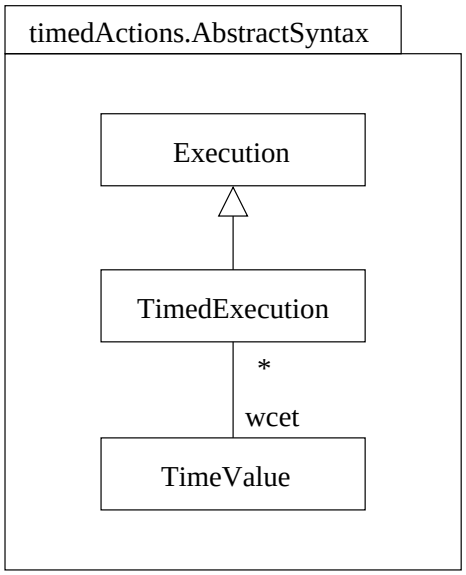

Figura 3.36: Dominio semántico de las acciones con tiempo.

Respecto al modelo de ejecución, vamos a considerar que todos los objetos cuyo funcionamiento viene definido por una máquina de estados se implementan con objetos activos, y que cada uno cuenta con su hebra de ejecución propia. Esta opción coincide con el concepto de *concurrency unit* que se define en el capítulo del modelado de la concurrencia del *UML Profile for Schedulability, Performance, and Time Specification*, [56].

El modelo de paralelismo implicará que los objetos activos se ejecutarán concurrentemente y que su ejecución es interrumpible. Un objeto podrá ser interrumpido por otro objeto cuando se produzca un evento que este otro objeto sea capaz de procesar y si el proceso que se activa tiene una prioridad mayor que el proceso que va a ser interrumpido. La cuestión de las prioridades se tratará a continuación.

En lo que concierne al manejo y postergación de los eventos recibidos por un objeto activo, se supondrá que cada objeto cuenta con una cola propia en la que se almacenan temporalmente las instancias de los eventos que el

objeto ha recibido pero no ha procesado aún.

3.4. Problemas de tiempo real

En [14] se estudian en profundidad los aspectos relacionados con el diseño de sistemas de tiempo real cuando se emplea SDL y sus mecanismos estándar de manejo de tiempo. En la sección anterior se ha propuesto una ampliación de las máquinas de estados de UML con la idea de reproducir el entorno de SDL, con, por ejemplo, relojes y temporizadores, con sus respectivas operaciones.

Es necesario, por tanto, considerar también en este caso si los problemas detectados en SDL aparecen asimismo cuando se usan las máquinas de estados de UML para diseñar un sistema de tiempo real.

Expresión de restricciones temporales

Una función usual para la que se utilizan los temporizadores es la de marcar una secuencia periódica, para, por ejemplo, activar una actividad periódicamente, como puede ser un sensor. Otra función es la de marcar un retraso, o la consumición de un período de tiempo, que, por ejemplo, indique el final del plazo de ejecución seguro asignado a un proceso.

En ambos casos, cuando el plazo de activación acaba, el temporizador genera un evento temporal que se envía a la máquina adecuada.

Ilustremos esta idea con un ejemplo. Supongamos una máquina de estados simple que cuenta con un único estado, figura 3.37, en el que se admiten dos señales, una corresponde a la expiración de un temporizador, t, que se activa con un determinado plazo temporal cada vez que se entra al estado, y la otra corresponde a la recepción de una instancia de un evento externo, e. Con este diseño se pretende conseguir una activación periódica del temporizador.

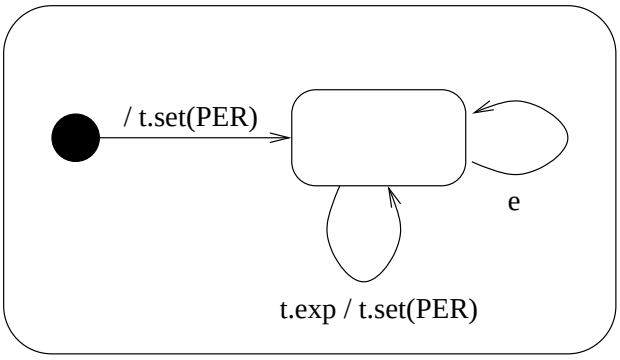

Figura 3.37: Máquina de estados con activación periódica de un temporizador.

Como se ha mencionado en la sección 3.3, los eventos que llegan a una máquina de estados se almacenan en una cola, con una política del *primero en entrar, primero en salir*. Si la máquina de estados está ocupada respondiendo a la recepción de otro evento cuando recibe una instancia del evento de expiración del temporizador, el procesamiento del evento de la expiración del temporizador se verá retrasado. Cuando la máquina de estados por fin atienda ese evento y llame a la función para consultar el valor del tiempo actual con la operación `now()` para volver a activar el temporizador, el valor devuelto por la función será el instante en el que se ha empezado a atender el evento, no el instante en el que se ha recibido. Eso significa que el retraso con el que el temporizador se activa la siguiente vez es mayor que el del período deseado.

Esa situación también se puede dar en otras circunstancias. Si la cola de eventos no está vacía, sino que contiene alguna instancia de la otra señal, en cuyo caso el tiempo que transcurre entre que se recibe la señal y ésta se procesa es aún mayor. Ocurre incluso que ese tiempo no está acotado porque no lo está el número de instancias de eventos que puede haber en la cola.

El otro motivo es que, con el modelo de concurrencia considerado, la máquina de estados puede ser interrumpida por otra máquina de estados

de mayor prioridad y esta interrupción puede ocurrir entre el instante de la recepción del evento y el inicio de su procesamiento, por lo que el valor devuelto por la función `now()` no coincidirá con el de la recepción de la señal, aunque la máquina atienda inmediatamente el evento.

Recursos compartidos

El uso de recursos compartidos es una situación habitual, y a veces inevitable, en un diseño concurrente. Por la naturaleza de UML, las variables no pueden ser directamente compartidas sino que tienen que estar encapsuladas en un objeto desde donde puede ser accesible a través de diversos mecanismos. En la figura 3.38 se muestra un sistema en el que hay tres máquinas de estados. El objeto al que corresponde la máquina de estados de la parte inferior de la figura contiene la variable x, a cuyo valor acceden los objetos a los que corresponden las máquinas de estados de la parte superior de la figura. En este caso el acceso a dicha variable se hace de manera ordenada a través de los dos eventos a los que responde la máquina de estados del objeto que contiene la variable.

Aunque las variables globales sean el caso más evidente de recurso compartido, hay otras situaciones donde también se da esta situación. Los dispositivos físicos son recursos compartidos a los que se accede desde distintos componentes del sistema. En los sistemas de tiempo real es muy habitual la interacción con estos dispositivos, por lo que su modelado dentro del sistema es fundamental. Además, la definición de un mecanismo de acceso a dispositivos físicos tiene una serie de beneficios: permite simular el sistema completo en la fase de diseño, lo que ayuda a tener una idea de la respuesta temporal del sistema, posibilita la realización de análisis de planificabilidad en la fase de diseño, incluyendo de esa manera los requisitos temporales de los disposi-

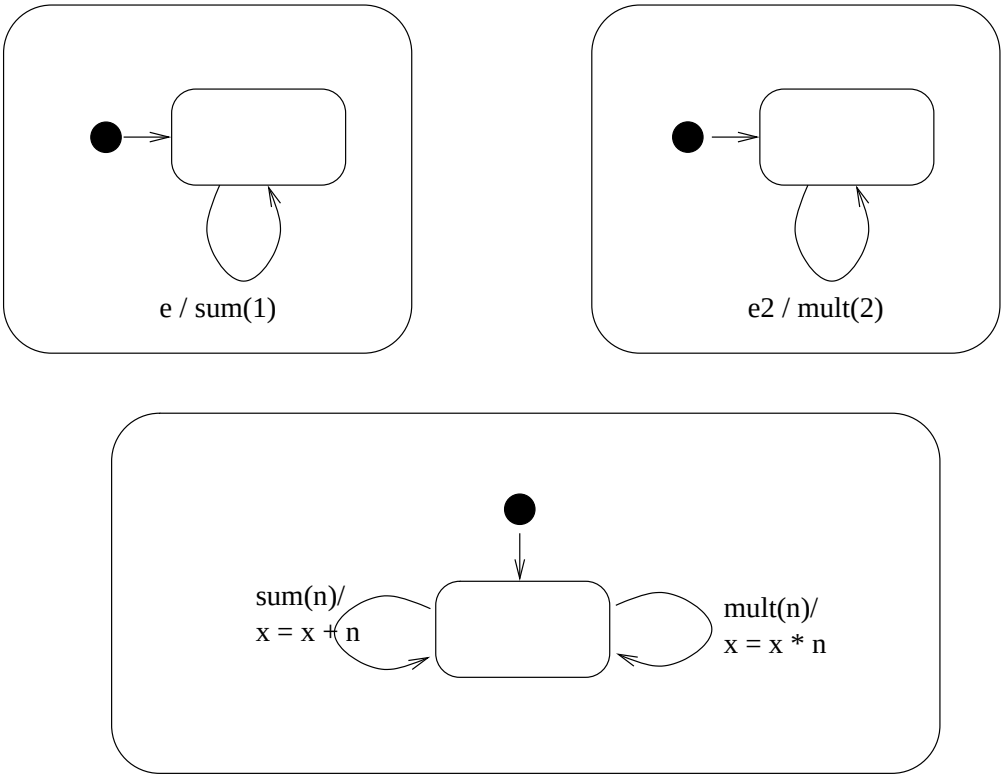

Figura 3.38: Variable compartida por dos máquinas de estados.

tivos físicos y ofrece un mecanismo común de interacción con los dispositivos físicos, lo que facilita la fase de implementación porque permite definir un esquema general de generación de código para acceso a los dispositivos físicos.

En la respuesta a un evento externo suelen estar involucrados varios objetos distintos. Entre estos objetos se crea una cadena de eventos, de tal manera que el objeto que recibe el evento externo inicial realiza una acción y genera otro evento, que será recibido por otro objeto. Sucesivamente, se generarán más eventos hasta que la respuesta al evento externo se complete. Puede ocurrir que un objeto esté involucrado en la respuesta a más de un evento externo. En ese caso podemos considerar ese objeto como un recurso compartido entre varias secuencias de eventos.

Inversión de prioridades

La inversión de prioridad es la situación que se produce cuando, en un sistema concurrente, un proceso de una determinada prioridad tiene que esperar porque hay otro proceso de menor prioridad que se está ejecutando y no puede interrumpirlo, aunque el modelo de concurrencia del sistema incluya la posibilidad de interrumpir los procesos en cualquier momento.

Cuando los procesos no comparten recursos se puede evitar la inversión de prioridad, pero si hay recursos compartidos no. Supongamos un sistema en el que varios objetos acceden a recursos proporcionados por un objeto *servidor*. El acceso a estos recursos implica el envío de un evento para la petición del servicio y la espera de otro evento de respuesta. Durante el tiempo entre ambos envíos, el proceso *cliente* está esperando a que el servidor acabe sus cálculos. Si hay un cliente con una prioridad mayor que la del servidor y está esperando la respuesta a un evento de petición de un servicio, otro proceso que tenga una prioridad menor que la del cliente, pero mayor que la del servidor puede activarse por la llegada de otro evento distinto e interrumpir al proceso servidor. En ese caso, el proceso cliente, con una prioridad mayor que la del tercer proceso, sufre una inversión de prioridad porque es postergado por un proceso de menor prioridad sin tener posibilidad de interrumpirlo.

El mismo ejemplo sirve para ilustrar otra situación de inversión de prioridad. Si el proceso servidor es accedido por múltiples clientes con distintas prioridades, es posible que, cuando un cliente con alta prioridad requiera un servicio por parte del servidor, se encuentre con que éste está atendiendo la petición de otro cliente con menor prioridad. Éste es otro escenario de inversión de prioridad. El tiempo que el proceso de mayor prioridad está postergado puede ser mayor porque la cola de eventos pendientes del proceso

servidor puede no estar vacía. Si la cola de eventos sigue una política de *primero en entrar, primero en salir*, se atenderán primero las peticiones de los procesos de menor prioridad.

El principal inconveniente en ambos casos no es la presencia de la inversión de prioridad, que es inevitable cuando se comparten recursos, sino que el tiempo en el que el proceso de mayor prioridad está bloqueado no es acotado. Esta circunstancia no permite hacer los análisis de planificabilidad habituales en los sistemas de tiempo real.

3.5. Extensión del entorno temporal de las máquinas de estados

Marcas temporales en los eventos

En la sección 3.4 se han presentado varias situaciones en la que una máquina de estados no puede saber el instante de tiempo en el que se ha recibido una instancia de un evento, provocando funcionamientos incorrectos desde el punto de vista temporal. Para posibilitar que una máquina de estados acceda a información temporal correcta sobre los eventos que procesa hemos definido la clase `TimedEventInstance`, mostrada en la figura 3.33, que representa las instancias de eventos con tiempo, en la que se incluyen cuatro atributos que representan instantes temporales de especial interés para dicha instancia del evento:

- `start`. Instante temporal en el que se ha generado la instancia del evento.
- `end`. Instante temporal en el que se ha terminado el procesamiento del evento. Definiremos el fin del procesamiento como el instante en el que la máquina de estados transita al estado destino de la transición disparada por el evento y está preparada para responder a otro evento.

- `received`. Instante temporal en el que el evento es colocado en la cola de eventos de la máquina de estados.

- `consumed`. Instante temporal en el que la máquina de estados empieza el procesamiento del evento.

Asignación de prioridades

En la definición de UML no se hace mención a la asignación de prioridades a objetos, máquinas de estados o transiciones. La única mención a las prioridades se hace cuando, en un estado compuesto, varios estados anidados responden al mismo evento, la transición correspondiente al estado más interno es la que se ejecuta antes que las otras.

En el *UML Profile for Schedulability, Performance, and Time Specification* [56], sí se hace referencia a la asignación de prioridades. En su capítulo 6, *Schedulability Modeling*, se trata el tema de la asignación de prioridades. Las acciones modeladas por la clase `SAction` tienen un atributo que indica la prioridad de la acción.

La asignación de prioridades directamente a las acciones no refleja, desde nuestro punto de vista, la filosofía que debe seguir la construcción de un sistema reactivo a eventos basado en máquinas de estados concurrentes. La asignación directa de prioridades a acciones tiene sentido cuando en el sistema todas las acciones se están ejecutando concurrentemente. Con la prioridad de cada acción el planificador sabe a qué acción tiene que asignarle el procesador y si una acción puede interrumpir a otra.

Sin embargo, cuando el sistema se basa en máquinas de estados, los objetos están esperando a que ocurra un evento al que ellos respondan. Si se produce un evento que activa una transición de un objeto cuando el procesador está ocupado ejecutando las acciones de respuesta a otro evento en otra

transición de otra máquina de estados, el planificador no puede saber si se puede interrumpir a esa otra máquina de estados o no, porque la transición no tiene una prioridad que comparar con la prioridad de la acción que se está ejecutando. Se puede argumentar que la prioridad de la transición es igual que la transición de la acción que ejecuta, pero tampoco es una solución clara, porque en una transición puede haber tres acciones involucradas: la acción de salida del estado, la asociada a la transición y la de entrada al nuevo estado.

En vez de asignar las prioridades a las acciones, nosotros las asignamos a las transiciones, como se propone en [14]. Las prioridades de los objetos, que serán normalmente los elementos que se implementen sobre una entidad planificable por el sistema, como un proceso, variarán en función de la transición que estén ejecutando. La planificación de los objetos se hará en función de la prioridad que tenga cada uno en ese instante. Esta posibilidad de cambiar de prioridad permite que un objeto que toma parte en la respuesta a varios eventos atienda cada uno a la prioridad adecuada.

Todas las transiciones involucradas en la respuesta a un evento deben compartir la misma prioridad para que el procesamiento del evento no se vea interrumpido por la llegada de otro evento de menor prioridad (en ese caso estaríamos frente a una situación de inversión de prioridades). Las transiciones compartidas, es decir, aquéllas que son parte de la respuesta a varios eventos externos tendrán como prioridad la mayor de entre las prioridades de los eventos a los que responde. La actualización de la prioridad de los objetos en base a la prioridad de la transición que están ejecutando está explicada en profundidad en [14].

Modelado de recursos compartidos

En esta sección proponemos una estrategia de diseño para el acceso a recursos compartidos. Los recursos estarán encapsulados en un tipo especial de objetos, que eviten la inversión de prioridad y posibiliten una implementación eficiente.

Estos objetos han de cumplir las siguientes propiedades:

- Son *servidores pasivos*, no empiezan ninguna secuencia de acciones por su propia iniciativa.
- El método de comunicación que ofrece es el de llamadas remotas a procedimientos.
- Su funcionamiento no implica estados, por lo que no tendrán asociados máquinas de estados.
- El tiempo de ejecución de sus procedimientos debe estar acotado, para que la inversión de prioridad sea limitada y, por tanto, aceptable para el análisis de planificabilidad.

En la sección 3.1.6, *The Resource Types Package*, del *UML Profile for Schedulability, Performance, and Time Specification* [56], donde se clasifican los recursos en función de diferentes características, se proponen objetos *activos*, que inician secuencias de estímulos, y *pasivos*, que sólo responden a estímulos, y, por otro lado, respecto a la protección, se distinguen recursos *no protegidos*, que no proporcionan mecanismos de protección frente a accesos simultáneos, y *protegidos*, que ofrecen servicios exclusivos. El atributo `accessControlPolicy` permite establecer el tipo de política de control de acceso al objeto. En el capítulo 6, *Schedulability Modeling*, del *UML Profile for*

Schedulability, Performance, and Time Specification [56], los recursos compartidos, modelados por la clase `SResource`, tienen un atributo que indica el techo de prioridad.

Con esas propuestas, podemos concluir que los recursos compartidos deberían modelarse con un objeto pasivo y protegido, para establecer la política de control de acceso y que ofreciera la posibilidad de asignar un techo de prioridad.

CAPÍTULO 4

Metodología de diseño de sistemas de tiempo real orientada a objetos

1. Introducción

En este capítulo presentamos una metodología de desarrollo de sistemas empotrados de tiempo real para el diseño de sistemas monoprocesadores.

Esta metodología es una adaptación del trabajo presentado en [10] y [15]. En esos trabajos se presenta una metodología cuyas características más sobresalientes son el uso sistemático de lenguajes gráficos de modelado para la especificación y diseño del sistema en sus diferentes etapas, la integración del análisis de planificabilidad en el entorno del uso de estos lenguajes gráficos y el modelo de interacción con los dispositivos físicos. La metodología incluye entre sus fases la simulación y validación del sistema. Para estas fases es necesario que el lenguaje en el que describe el sistema tenga una semántica precisa para generar una implementación ejecutable del sistema.

Para el análisis de planificabilidad se propone *Rate-Monotonic Analysis* (RMA) [47, 30, 77], que proporciona una colección de métodos cuantitativos para analizar y predecir el funcionamiento temporal de los sistemas de tiempo

real. Estos análisis pueden ayudar a organizar los procesos y los recursos en el diseño para hacer posible predecir el funcionamiento del sistema final. En este sentido, RMA ayuda a incorporar el análisis de tiempo real [67, 55, 78] y salvar el hueco entre el modelo de objetos y el modelo de tareas. Las tareas que se usan en el análisis de planificabilidad se obtienen a partir de la especificación del funcionamiento dinámico del sistema.

Tanto en [10] como en [15] se usa UML en las primeras fases, especificación y diseño del sistema, y SDL en la parte final, para el diseño estructural detallado, el diseño de procesos, la implementación de código y la simulación y validación del sistema. Una de las razones principales de la elección de SDL en [10] y [15] es que, gracias a su semántica, formal y claramente establecida, existen herramientas automáticas como Tau, de Telelogic, que incluye entre sus funciones la simulación y validación del sistema especificado y la generación de código. Por otro lado, hemos desarrollado herramientas que permiten aplicar RMA para realizar el análisis de planificabilidad de un sistema especificado en SDL, mediante su transformación en grafos de tareas.

En este capítulo se describe una modificación de la metodología propuesta en [10] y [15] en la que se sustituye SDL por UML en las etapas finales del diseño. La razón de esta sustitución es la tremenda popularidad que ha alcanzado UML para la construcción de sistemas de todo tipo, incluidos los sistemas reactivos y de tiempo real.

El uso de UML, no obstante, lleva acarreados inconvenientes que hay que solventar. Como UML carece de una semántica formal, no es posible aún llevar a cabo las últimas fases de la metodología sobre diagramas UML: generación de código, simulación, validación y análisis de planificabilidad.

Por este motivo, en los capítulos anteriores de este trabajo se ha desarrollado una semántica precisa para las acciones y para las máquinas de estados

de UML. Esta semántica es el primer paso para eliminar la ambigüedad presente en UML y y construir herramientas automáticas que proporcionen las funciones mencionadas en el párrafo anterior. En este trabajo no se ha incluido el desarrollo de estas herramientas.

1.1. Trabajo relacionado

Se han llevado a cabo algunos trabajos para intentar integrar este tipo de análisis en un modelo SDL. Por ejemplo, ObjectGeode [50] incluye un analizador de rendimiento en su simulador que hace uso de algunas directivas para extender SDL para indicar prioridades de procesos o retrasos temporales. En [71] se incluye una semántica *del plazo más inmediato* para SDL, de manera que permite una traducción desde una especificación en SDL a una red de tareas analizable. Sin embargo, no integran este análisis en el resto del ciclo de desarrollo, ni tienen en cuenta la interacción con el hardware o anomalías de tiempo real como la inversión de prioridades. Otra línea de trabajo en este contexto es la de suplementar SDL con modelos de carga, como se describe en [82], que usa teoría de colas para calcular los tiempos en las colas de los trabajos y los mensajes y las cargas de trabajo media y máximas de los procesadores. En [42] se presenta una estrategia nueva para la predicción temprana de rendimiento basada en sistemas especificados en MSC en el contexto de SDL. Pensamos que estos trabajos pueden ser complementarios y útiles en las primeras fases del diseño, pero que para los sistemas de tiempo real es necesario hacer un análisis final de planificabilidad y, para conseguirlo, es necesario proporcionar un modelo de ejecución planificable para UML. En [103] y [106] se puede ver cómo aplicar la teoría de planificabilidad en ROOM. Aunque estos trabajos han sido muy útiles para la propuesta que ahora presentamos, ROOM no es una técnica de descripción formal y no puede usarse

para incluir una fase de validación automática directamente desde el diseño. Pensamos que la validación es un tema importante para el diseño de los sistemas de tiempo real. En [105] se presenta una propuesta encaminada a la síntesis automática de implementaciones a partir de los modelos de diseño orientados a objetos de tiempo real. Éste es un trabajo muy interesante e incluye el tema del tiempo real y la respuesta en un plazo de tiempo en el proceso de diseño. [104] completa el trabajo previo y muestra cómo se puede integrar el análisis de planificabilidad con el diseño orientado a objetos.

En [6] se estudia una metodología para construir sistemas planificados restringiendo el funcionamiento de los procesos para garantizar dos tipos de restricciones: restricciones de planificabilidad y restricciones sobre los algoritmos de planificación tales como prioridades para procesos y posibilidad de interrupción. Aunque la metodología se ilustra sobre procesos periódicos, puede aplicarse a sistemas arbitrarios. Otra contribución de este trabajo es la descomposición de los requisitos de planificación en clases de requisitos que pueden ser expresados como restricciones de seguridad.

UML-MAST [40] es una metodología y un conjunto de técnicas de análisis para especificar, diseñar y estudiar sistemas de tiempo real haciendo uso de diversos diagramas de UML en herramientas automáticas, como diagramas de classes y objetos, de secuencias y de actividad. La metodología incluye cinco vistas del sistema, entre las que se encuentra la "Mast RT View", que incluye información sobre las características necesarias para realizar los análisis de tiempo real, como el funcionamiento temporal, las restricciones de rendimiento, etc. Esta información se pasa como entrada a diversas herramientas que analizan diferentes aspectos temporales del sistema, como análisis de planificabilidad, simulaciones o análisis de utilización. Los autores declaran que los nuevos componentes incluidos en UML para especificar

esta nueva vista disponen de un metamodelo formal. Este metamodelo se define se una manera similar a como se definen las clases del *UML Profile for Schedulability, Performance, and Time Specification*, discutido en la sección 3.1 del capítulo 3. La semántica estática se define mediante diagramas de clases, pero la semántica dinámica se hace en lenguaje natural.

Con respecto al trabajo relacionado con la traducción del modelo de objetos a modelo de procesos, UML/RT [111] propone una vista dinámica basada en máquinas de estados para cada objeto pasivo del modelo de objetos. Octopus, [63], propone el modelo de diseño como una extensión del modelo de objetos, aunque puede necesitar una descomposición ulterior en la fase de implementación. Con el objetivo de diseñar la concurrencia, Octopus propone el concepto de *hebra de interacción entre objetos* que representa el conjunto de objetos que pueden tomar parte en un evento. Octopus usa máquinas de estados para describir el funcionamiento en términos de cambios de estado causados por los eventos. Sin embargo, no los usan formalmente. Usa *class outlines* para grabar los detalles del diseño. Ése es el punto de arranque de la fase de implementación. Octopus combina dos conceptos, *objetos* y *procesos del sistema operativo*, para diseñar la concurrencia. La traducción directa de objetos a procesos del sistema operativo no es fácil y pensamos que el paso intermedio que proponemos la facilita.

El paso del modelo de análisis al modelo de diseño en ROPES [39] puede hacerse usando técnicas elaborativas o traductoras. Las técnicas traductoras proponen definir traductores aplicados al modelo de objetos para producir un sistema ejecutable. El traductor tiene dos partes: un marco de tiempo real y un generador de código. En este caso el modelo de objetos necesita estar más detallado que el que proponemos en nuestra metodología. El modelo elaborativo es más tradicional. El modelo de análisis es realizado con detalles

de diseño y puede puede ser mantenido como una entidad distinta del modelo de análisis. Esta línea está más cercana a nuestra propuesta.

HRT-HOOD [29] traduce el modelo de objetos a Ada. Para cada objeto se generan dos paquetes: el primero simplemente contiene una colección de tipos de datos y variables que definen los atributos de tiempo real del objeto; el segundo contiene el código del objeto. En este sentido esta metodología propone un enfoque traductivo.

Estas formas diferentes de pasar del modelo de objetos al modelo de diseño son muy interesantes, pero pensamos que es importante incluir en el modelo de diseño un formalismo que incluya una fase de validación para detectar situaciones indeseables en los diseños concurrentes como la inanición, bloqueos, etc.

2. La metodología

Nuestra propuesta está conceptualmente basada en alternativas que aparecen en otras metodologías (en particular SOMT y Octopus). Sin embargo, proporcionamos soluciones parciales a algunos de los problemas presentes en las metodologías mencionadas anteriormente, y también en otras metodologías de desarrollo orientadas a objetos. De hecho, nos ocupamos del salto existente entre el modelo de objetos y el modelo de procesos, que presenta un interés especial en el contexto de sistemas empotrados de tiempo real. Cubrimos ese hueco mediante dos estrategias ortogonales. Así, proporcionamos una estrategia para pasar del modelo de objetos al modelo de procesos, de tal manera que las restricciones de tiempo real pueden ser predecibles. Los criterios de diseño que pueden aplicarse cuando una clase (u objeto) del modelo de objetos tiene que ser traducida a uno o más procesos (descritos mediante máquinas de estados) se resumen en la sección 2.2. Esta forma de

traducir el modelo de objetos al modelo de procesos es un aspecto importante de nuestra propuesta, y proporciona una estrategia clara para convertir objetos en procesos, dando la posibilidad de analizar las propiedades de tiempo real. Sin embargo, hemos aprendido de nuestra experiencia en el desarrollo de sistemas empotrados de tiempo real otras dos lecciones. En primer lugar, el modelo de objetos puede ser refinado un poco más de lo que recomiendan otras metodologías, y en algunas ocasiones este diseño orientado a objetos más detallado es altamente recomendable. En segundo lugar, se puede conseguir un mejor provecho de este modelo más refinado si obtenemos una buena correspondencia entre los elementos del modelo de objetos y los elementos del modelo de procesos. Así, pensamos que esta traducción detallada entre ambos modelos presenta algunas novedades con respecto a las sugerencias hechas en otras propuestas.

El enfoque principal de esta metodología es sobre el sistema de desarrollo usando análisis y diseño orientados a objetos, pero proporcionando análisis de tiempo real basado en las máquinas de estados de UML. La metodología propuesta está básicamente dividida en cuatro fases *Análisis*, *Diseño*, *Validación* e *Implementación*. En cada fase se tienen en cuenta los aspectos funcionales y no funcionales en dos vistas del sistema separadas, pero complementarias. Esta distinción nos permite capturar las características de tiempo real y las peculiaridades de los dispositivos físicos del sistema, con relativa independencia de los requisitos funcionales (modelos de objetos y dinámicos), pero recordando en qué nivel han surgido los requisitos específicos.

La figura 4.1 muestra las diferentes fases de la metodología. Como se ha mencionado ya, esta figura muestra cómo cada fase se divide verticalmente en dos vistas, dando las perspectivas funcionales y no funcionales. Obsérvese que las expresiones usadas para denotar cada fase son diferentes dependiendo

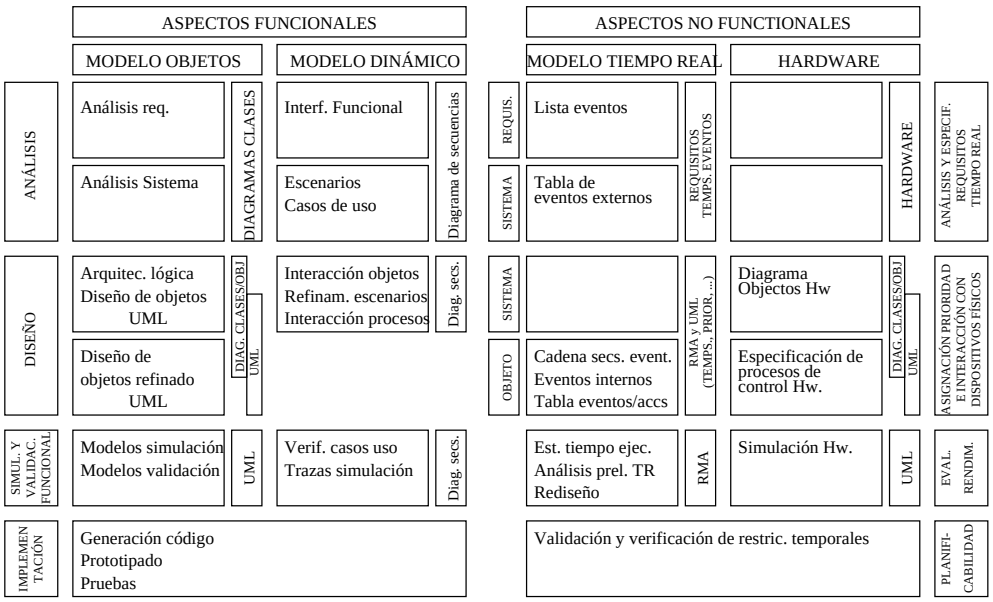

Figura 4.1: Fases de la metodología

de sobre qué aspecto nos queremos centrar en cada vista. La primera fase se denomina *Análisis* desde una perspectiva funcional, mientras que usamos el término *Especificación de requisitos de tiempo real* para la vista no funcional. Cada una de estas dos vistas está a su vez subdividida en otras dos vistas (llamadas modelos), distinguiendo así el modelo de objetos del modelo dinámico en la perspectiva funcional y el modelo de tiempo real de las restricciones físicas en la perspectiva no funcional.

En la fase de *análisis* se identifican y se analizan el dominio del problema y los requisitos del usuario del sistema. Básicamente, durante esta fase establecemos un primer modelo de objetos y los casos de usos preliminares del sistema. Usaremos dos notaciones gráficas para apoyar ambas actividades: los diagramas de clases de UML para capturar la descripción orientada a objetos, y diagramas de secuencias para describir el modelo dinámico.

Durante la fase de *diseño* los aspectos funcionales del sistema se estudian a dos niveles: sistema y objetos. El diseño se realiza básicamente mediante

diagramas de clases y objetos de UML. El diseño del sistema corresponde a un diseño de alto nivel, que se refina cuanto sea necesario hasta llegar a un diseño de objetos detallado.

En la fase de *simulación, validación y evaluación del rendimiento* se contrastan los resultados de las fases de análisis y diseño. La simulación y la validación del sistema se hacen usando la especificación del sistema antes de su implementación. Estas técnicas nos permiten detectar posibles errores generales de diseño tales como bloqueos, consumo implícito de señales, etc. Además, se pueden estudiar algunos escenarios particulares del sistema mediante la verificación del funcionamiento mediante diagramas de secuencia. Estos diagramas de secuencia pueden describir situaciones requeridas, o prohibidas, que pueden verificarse automáticamente por las herramientas existentes. En este sentido, es posible usar los diagramas de secuencia refinados en la fase previa para verificar parte de la funcionalidad del sistema.

Finalmente, en la fase de *implementación* obtenemos el código correspondiente a la parte del sistema específica de la aplicación. Este código específico de la aplicación se sostiene típicamente sobre servicios genéricos proporcionados por un nivel subyacente (sistema operativo o el sistema de tiempo de ejecución). Este nivel debe proporcionar ciertas características para permitir nuestro modelo de ejecución de tiempo real analizable. Al menos, debe proporcionar planificación de procesos completamente interrumpible basada en prioridades fijas y alguna forma de herencia de prioridad. Además, el modelo de dispositivos físicos debería adaptarse al proporcionado (si lo hay).

En las próximas secciones vamos a describir en detalle los nuevos aspectos de la metodología que presentamos, resaltando las fases donde se hace alguna contribución relevante. Para mantener la relación temporal entre aspectos funcionales y no funcionales, presentaremos cada fase dando las vistas

funcionales y no funcionales al mismo tiempo.

2.1. Análisis y especificación de requisitos de tiempo real

La parte funcional del análisis está dedicada a la obtención de los requisitos de usuario. En su parte funcional, se usan los diagramas de casos de uso y los diagramas de actividad para identificar la dinámica de uso del sistema. Con esta información, se crean los diagramas de clases que identifican los actores involucrados en las acciones descritas anteriormente, desde el punto de vista del usuario.

En esta fase también tiene una gran importancia la descripción de los aspectos no funcionales. En este nivel, el sistema se ve como una caja negra que responde a estímulos del exterior, por lo que toda la información que se recopile será relativa a eventos producidos fuera de los límites del entorno. Se tiene que tener en cuenta que el sistema habrá de responder simultáneamente a eventos externos cuya frecuencia puede ser distinta, o no estar definida y dar el resultado deseado dentro de los límites de tiempo impuestos. Así, esta fase se centrará en la detección de los requisitos temporales de los eventos externos (específicamente eventos de entrada) producidos por la interacción con el entorno, en el que incluyen los dispositivos físicos y otros subsistemas. El resultado de esta actividad es una tabla con los eventos externos, donde por cada evento se recopila la siguiente información: identificación del evento, clases involucradas en la respuesta del sistema y requisitos temporales (sobre período, plazo de respuesta, *jitter*, ...).

2.2. Diseño e interacción con dispositivos físicos y asignación de prioridades

En esta fase, una diferencia importante de nuestra propuesta respecto a otras metodologías (por ejemplo SOMT) es la posibilidad de reducir el hueco existente entre el diseño estructural (en diagramas de clases y objetos en UML) y el de procesos (máquinas de estados en UML). Esto es posible gracias a las actividades que se llevan a cabo durante esta fase (interacción con los dispositivos físicos y asignación de prioridades, véase sección 2.2) para capturar los aspectos no funcionales.

Nivel de objetos (Sistema)

En el aspecto funcional, en la fase de diseño se construye la estructura del sistema partiendo de los diagramas obtenidos en la fase de análisis.

A partir de los diagramas de clases del análisis se generarán nuevos diagramas de clases que reflejen la arquitectura lógica del sistema. Estos diagramas de clases se instanciarán en diagramas de objetos para reflejar la estructura concreta del sistema.

Nivel de objetos (Diseño)

Los diagramas de clases y objetos se usan hasta en los últimos pasos de esta fase. Así, durante este nivel del diseño, los diagramas de clases y objetos se van detallando hasta llegar a un diseño de objetos muy refinado.

Como en otras metodologías, los objetos se traducen a procesos siguiendo criterios claros tales como los siguientes: los objetos activos se describirán con una máquina de estados propia, al menos, para que tenga su propia capacidad de ejecución. En algunas situaciones tiene que considerarse la posibilidad de incluir en un objeto capacidad para procesamiento múltiple. Una de estas

situaciones se da cuando el objeto está involucrado en el procesamiento de varios eventos simultáneos que no pueden cumplir sus requisitos temporales. En este caso, puede ser necesario dividir el objeto en varios para atender por separado a los diferentes eventos. Otra posibilidad es la de usar máquinas de estados con objetos concurrentes. Los objetos pasivos no compartidos por varios objetos activos se modelan como un tipo de datos interno al proceso cliente. Si el objeto pasivo es compartido consideramos un proceso con las limitaciones establecidas en [15, sección 2.3.2]. Finalmente, también tratamos con las clases que modelan el funcionamiento de la parte física, y proponemos la distinción entre procesos activos (manejadores de dispositivos físicos) y procesos pasivos (que modelan directamente los dispositivos físicos), como se estudió en [15, sección 2.3.3].

A la vez que el diseño de los aspectos funcionales del sistema también se tienen en cuenta las características no funcionales. En esta fase, *Asignación de prioridades e interacción con los dispositivos físicos*, aún se mantienen en esta actividad dos niveles: el nivel de sistema y el nivel de objetos.

Interacción con los dispositivos físicos y asignación de prioridades (Nivel de sistema)

Respecto a los temas no funcionales, el diseño tiene que capturar cómo los dispositivos físicos interaccionan con el sistema. En este sentido, cada componente físico tiene que modelarse como un objeto; por tanto, hay que definir una clase para representar el componente físico correspondiente. Las actividades que se han de desarrollar en esta fase son similares a las propuestas en otras metodologías como, por ejemplo, en Octopus. Sin embargo, las guías para agrupar las clases que modelan los dispositivos físicos marcan una diferencia respecto a la estrategia propuesta por Octopus. De hecho, como se mencionó en [15, sección 2.3.3], en vez de agrupar todas estas clases en

un único subsistema, proponemos distribuir las clases de dispositivos físicos entre los diferentes subsistemas que han sido generados durante la fase de análisis. Esto promueve un diseño más natural y razonable, porque las clases que representan cada componente físico están definidos allí donde se necesitan. Básicamente, la notación que proponemos a este nivel son los diagramas de clases y objetos de UML.

Interacción con los dispositivos físicos y asignación de prioridades (Nivel de objetos)

El nivel de objetos de esta fase está dedicado a diseñar la interacción con los dispositivos físicos, como se trató en [15, sección 2.3.3], distinguiendo — para cada componente físico — un objeto pasivo y un objeto controlador. El primero modela el acceso al componente físico y el segundo implementa el protocolo demandado por los requisitos del sistema. Esta forma de diseñar los componentes físicos ayuda a decidir qué se va a diseñar como *software* y qué como *hardware*, y a simular el sistema en la fase de diseño. Los procesos físicos desaparecen en la fase de implementación.

Durante esta fase se llevan a cabo otras actividades. Por un lado, se completan las cadenas de secuencias de los eventos, asociando objetos a eventos, o fijando las prioridades de las transiciones (véase [15, sección 2.3.1]). Por otro lado, se asignan los techos de prioridad de los procesos pasivos usando métodos como el de techo de prioridad inmediato. Finalmente, se construye una tabla, que muestra la correspondencia entre cada evento y la correspondiente secuencia de transiciones. Además, cada transición tiene asociada cierta información, como el tiempo de retraso debido a los bloqueos (si lo hay), si es atómica o no, el tiempo de ejecución en el peor caso y su prioridad relativa.

2.3. Evaluación del rendimiento

En esta fase se usan los modelos de simulación para analizar cuantitativamente medidas tales como el *throughput* y el tiempo de respuesta. Sería deseable que este análisis se llevara a cabo con herramientas comerciales ya disponibles, aunque éstas no ofrecen la generación del código completo de un sistema a partir de una especificación UML.

En lo que respecta a la interacción con los dispositivos físicos, éstos tienen que ser simulados mediante elementos especificados en UML. Cuando el sistema esté implementado se accederá al dispositivo físico mediante funciones C integradas en el código generado a partir del diseño. Sin embargo, durante esta fase tenemos que completar la especificación con clases y objetos UML que modelen el funcionamiento de los dispositivos físicos. Esto nos permitirá la simulación y validación del sistema completo, teniendo también en consideración la interacción con los dispositivos físicos. Otro aspecto relevante es el que corresponde a los requisitos de tiempo real. Después del diseño, es necesario un análisis de tiempo real preliminar, que se basará en el análisis propuesto en [15, sección 2.4]. Usando este análisis es posible saber el funcionamiento temporal del sistema antes de la fase de implementación, y entonces hacer los cambios necesarios para mejorar el tiempo de respuesta del sistema. En este sentido, la fase de *evaluación del rendimiento* también incluye un paso de rediseño para refinar el sistema especificado en UML. Basándonos en nuestra experiencia, este paso propone una serie de heurísticos para cambiar el diseño y mejorar los tiempos de respuesta de los eventos del sistema. Este paso será especialmente útil cuando el sistema no cumpla los plazos de tiempo. Los cambios en el diseño afectarán a los parámetros de la ecuación del tiempo de respuesta (véase [15, sección 2.4]) como el bloqueo y la interferencia. Proponemos un conjunto de heurísticos que nos permiten:

- Rediseñar el sistema si no cumple los plazos de tiempo.
- Tener en cuenta los requisitos de tiempo real desde los primeros pasos del diseño.

Los heurísticos se pueden resumir en los siguientes:

- Transferencia de eventos, para reducir la interferencia con la respuesta a otros eventos dentro del mismo objeto.
- Creación de procesos, para reducir el tiempo de bloqueo debido a eventos de menor prioridad tratados por el mismo proceso.
- Eliminación de transiciones intermedias, eliminando transiciones entre objetos que no añadan ninguna información, sino que sólo sirvan como paso de eventos.

En [15] y [117] se estudian en profundidad las técnicas de rediseño aquí comentadas.

3. Un caso real: el diseño de un teléfono inalámbrico

El Eole 400 es un teléfono inalámbrico de la clase CT0 de Alcatel, que admite múltiples auriculares, con un contestador automático de estado sólido integrado en la base. Está basado en otros dos productos anteriores CT0, el France Telecom X1 (para el auricular y el circuito de radio de la base) y el Eole 300 (para el resto de la base). El sistema se compone de una parte fija (la base) y de uno o más auriculares. El *software* instalado en el teléfono proporciona las siguientes prestaciones principales:

- Intercomunicación entre la base y los auriculares.

- Rellamada automática al último número marcado.
- Activación del altavoz durante la comunicación.
- Encriptación de la comunicación base-auricular.
- Marcación por pulsos o multifrecuencia.
- Tres melodías programables en el auricular y opción de desconectar el timbre.
- Contestador automático con modos de funcionamiento simple y de grabación y acceso remoto autentificado con código de seguridad.

3.1. Descripción de la parte física

La base del EOLE 400 está compuesta de varios circuitos electrónicos, algunos de ellos programables:

- El microcontrolador NEC de 4 bits μPD75116, con 8192x8 bits de memoria de programa (ROM), 512x4 bits de memoria de datos (RAM), 34 puertos de E/S, un interfaz serie de 8 bits y un reloj a una frecuencia de 4,19 Mhz. El tiempo de ejecución mínimo de una instrucción es 0,95 μs. Las puertos de E/S conectan los diodos luminosos y el teclado al microcontrolador.
- Una memoria EEPROM de chip único, 24c01, con una capacidad de almacenamiento de 1KByte.
- Un circuito de radio, conocido como COMBO, basado en el chip Motorola MC13110.
- Un circuito conversacional con altavoz basado en el chip Motorola MC34216A.

- Un módulo contestador, conocido como ATISAM, basado en el chip MSP58C80 de Texas Instruments.

Para iniciar una llamada, el usuario ha de pulsar la tecla `line` del auricular, con lo que se envía una petición de línea a la base. El circuito de radio detecta el mensaje e informa al sistema. Si la base está en disposición de procesarlo, es decir, si no está ya en el estado de comunicación, se establece un enlace de radio entre la base y el auricular y se activa el *gancho* de la línea telefónica. Para volver al estado de inactividad, el usuario puede pulsar otra vez la tecla `line`. Si, por ejemplo, se pulsa la tecla `intercom` en la base, empiezan las acciones para comunicar una petición a todos los auriculares mediante la difusión de un mensaje a través del circuito de la radio. Si un auricular contesta la petición, pulsando la tecla `line`, se establece una intercomunicación entre ese auricular y la base, gracias al micrófono integrado y al altavoz. En caso contrario, si no hay respuesta en 15 segundos se supone que no hay nadie interesado en la intercomunicación y la base vuelve al estado inactivo. Aunque éste es un sistema distribuido entre la base y los auriculares, sólo vamos a modelar la parte fija del sistema, la de la base. Es decir, estamos tratando un sistema monoprocesador.

3.2. Aplicación de la metodología

En esta sección trataremos los modelos principales de nuestra metodología, aplicándolos al sistema telefónico desarrollado.

La primera parte de nuestra metodología es la fase de análisis. Los aspectos funcionales de esta fase pueden dividirse en los niveles de requisitos y sistema. Los documentos resultantes del nivel de requisitos son casos de uso desarrollados a partir del manual de usuario, especificación técnica y entrevistas con los ingenieros involucrados en el proyecto. A nivel de sistema se

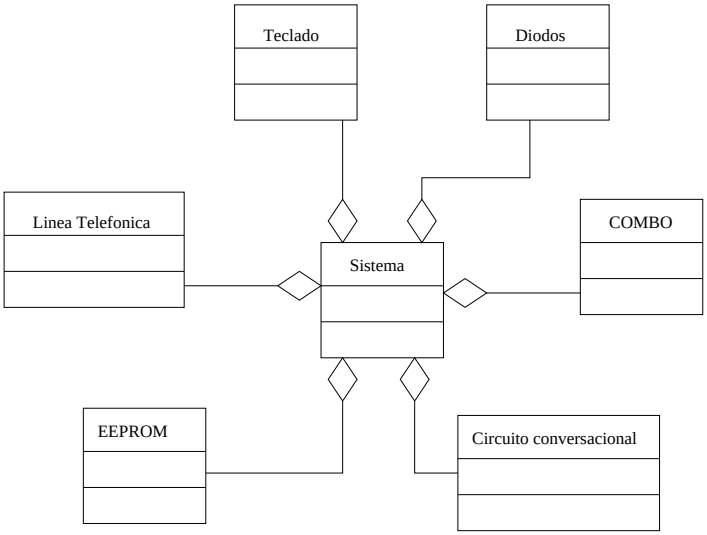

Figura 4.2: Arquitectura de objetos del sistema.

crea un primer diagrama de clases para mostrar las relaciones entre los objetos identificados en el estudio de los documentos de requisitos. Este diagrama se muestra en la figura 4.2[1]. Tras esto, se hacen diagramas de secuencias para estudiar el funcionamiento dinámico de los objetos en la posibles situaciones diferentes. En la figura 4.3 se muestra el diagrama de secuencia para la detección de una señal de llamada entrante desde la línea telefónica. Los aspectos no funcionales cubiertos en esta fase son los relativos a los eventos externos del sistema. Los eventos externos se muestran en la tabla 4.1 con sus nombres, requisitos temporales y los objetos a los que involucran, como se comentó en la sección 2.1.

La fase de diseño, interacción con el hardware y asignación de prioridades se divide en los aspectos funcionales y los asuntos de tiempo real y dispositivos físicos. A nivel de sistema hacemos una estructura lógica de las clases para diseñar la arquitectura del sistema que se refina para conseguir una definición

[1]La herramienta de modelado utilizada no permite el uso de caracteres acentuados, de ahí la falta de tildes en ésta y en otras figuras.

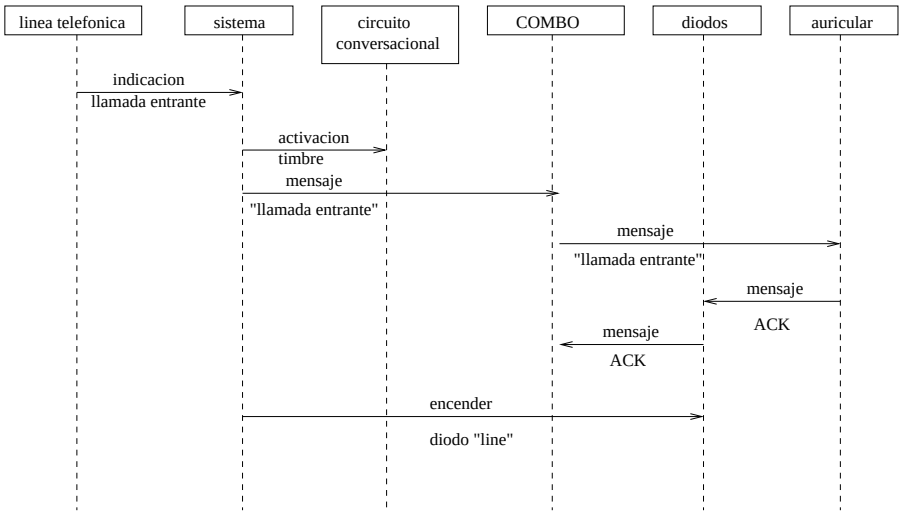

Figura 4.3: Diagrama de secuencias del escenario de llamada entrante.

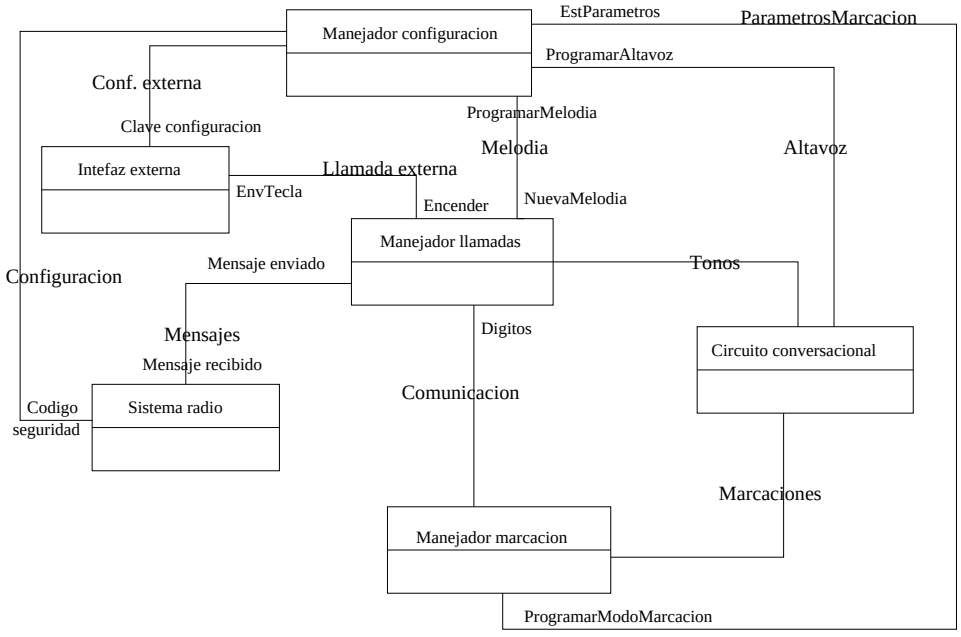

Figura 4.4: Diseño de objetos.

Evento externo	Objetos involucrados	Requisitos de tiempo
Llamada entrante	Sistema, Diodos, COMBO, Circuito conversacional, Línea telefónica	Buscar línea y señal no presente Periodo: 0.122 ms. Plazo:- Detectar señal en la línea. Periodo: - Plazo:700 ms
Llamada saliente	Sistema, Diodos, COMBO, Línea telefónica	Buscar en canales y no hay portadora. Periodo: 0.122 ms. Plazo:- Portadora detectada. Periodo: - . Plazo: 1100 ms
Intercomunicación	Sistema, Diodos, Teclado, COMBO	Analizar teclado y no hay tecla pulsada. Periodo: 7.82 ms. Plazo:- Tecla Intercom pulsada. Periodo: - . Plazo:1200 ms
Cambio de volumen	Sistema, Teclado, EEPROM, Circuito conversational	Analizar teclado y no hay tecla pulsada. Periodo: 7.82 ms. Plazo:- Tecla Volumen pulsada. Periodo: - . Plazo:2000 ms
Change melody	Sistema, Teclado, EEPROM, Circuito conversational	Analizar teclado y no hay tecla pulsada. Periodo: 7.82 ms. Plazo:- Tecla Intercom pulsada. Periodo: - . Plazo:2000 ms

Cuadro 4.1: Tabla de eventos externos.

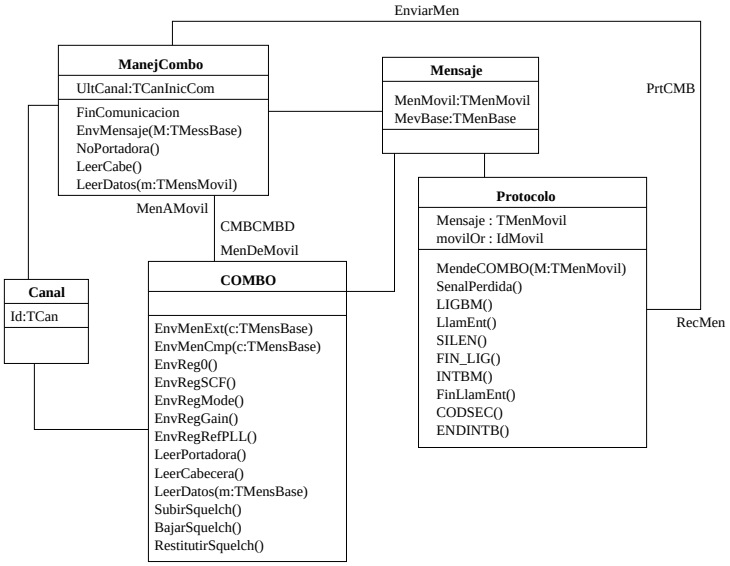

Figura 4.5: Diseño refinado del sistema de radio.

más precisa del sistema. Estos modelos se muestran en las figuras 4.4 y 4.5. En el nivel de sistema de los aspectos no funcionales hacemos un estudio más preciso de los objetos físicos como se comentó en la sección 2.2. Este estudio puede producir cambios en el diseño de objetos refinado. Como se comentó en [15, secciones 2.3.2 y 2.3.3] y en 2.2, los dispositivos físicos se modelan mediante dos objetos: un objeto pasivo que modela el dispositivo físico y que permite a los demás procesos acceder a sus recursos a través de llamadas remotas a procedimientos, y otro objeto activo, correspondiente al controlador, con un objeto controlador que es el que controla la comunicación entre el dispositivo físico y el resto del sistema. En nuestro sistema, el objeto `COMBO` modela el componente físico del sistema de radio y el objeto `DCombo` para el controlador. En la fase de análisis de planificabilidad, y en función de los árboles de tareas y los tiempos de respuesta de las acciones, se ve que hay eventos externos cuyo tiempo de respuesta es superior a su plazo máximo de respuesta.

Concretamente, esta situación se produce porque el objeto `DCombo` maneja eventos paralelos, ya que puede haber envío y recepción simultáneos de mensajes entre la base y los auriculares, y esta situación introduce un tiempo de bloqueo que impide que el sistema sea planificable. Este tiempo de bloqueo no se produce, por tanto, por un objeto externo que ejecute una acción de menor prioridad, sino por la naturaleza del sistema que impide que un objeto responda a dos eventos a la vez. Hay dos formas de conseguir eliminar ese tiempo de bloqueo en el procesamiento concurrente de los mensajes en ambos sentidos.

Una solución es la división del proceso en dos, ocupándose cada uno de ellos de los mensajes en un sentido. El inconveniente de esta solución es que los recursos del objeto original han de dividirse entre los dos nuevos objetos. Si cada uno de estos nuevos objetos no necesita acceder a los recursos del otro no hay ningún problema, pero si alguno de los recursos tiene que ser compartido por ambos objetos, hay que definir un nuevo objeto pasivo que simplemente actúe como contenedor de los recursos a compartir. Este objeto seguirá las reglas comentadas anteriormente para este tipo de procesos, sólo será accesible mediante llamadas remotas a procedimientos y su prioridad vendrá dada por el techo de prioridad de los objetos que acceden a él. En la figura 4.6 se muestran los dos objetos nuevos, `DComboEnt` y `DComboSal`, y el objeto pasivo, `CmbRecCmp`, encargado de encapsular los recursos compartidos que formaban parte del estado interno del objeto correspondiente al proceso controlador original. Estos recursos compartidos son la tabla de canales y los canales de transmisión y recepción. En las figuras 4.7 y 4.8 se muestran las respectivas máquinas de estados de los objetos `DComboEnt` y `DComboSal`.

Se podría pensar en una segunda alternativa, aprovechando las posibilidades de concurrencia intraobjetos que proporcionan las máquinas de estado de

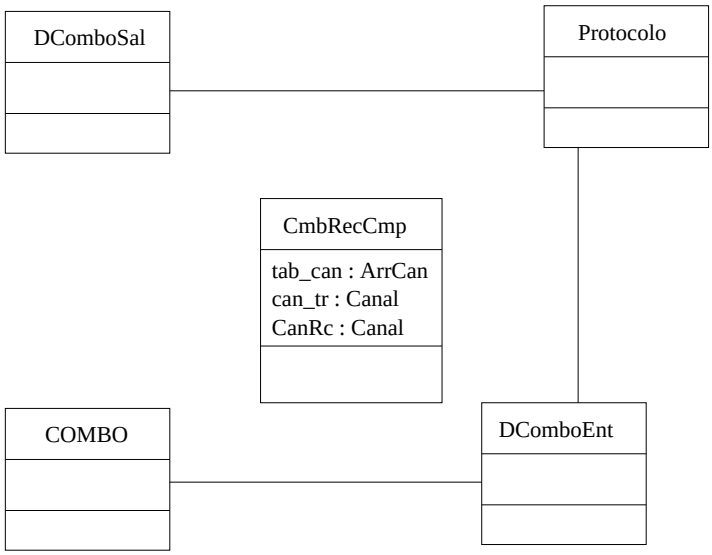

Figura 4.6: Especificación detallada del protocolo de radio

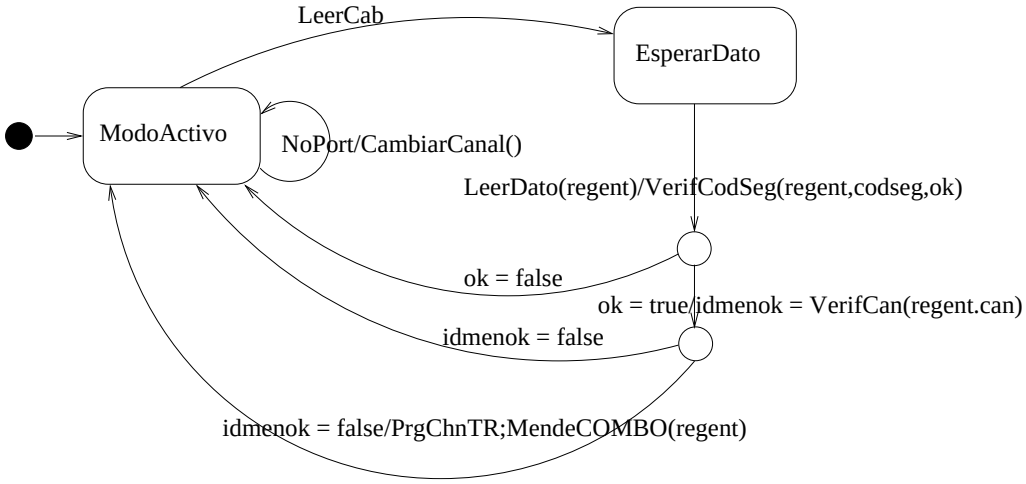

Figura 4.7: Máquinas de estados del objeto `DComboEnt`.

UML con los estados concurrentes. Los eventos que han provocado el tiempo de bloqueo excesivo se podrían procesar en diferentes regiones de un mismo estado concurrente. La principal ventaja de esta solución es que evita la complicación del diseño, ya que no se añaden nuevos objetos que no responden a necesidades de los requisitos funcionales, sino a restricciones no funcionales. Uno de los inconvenientes de esta solución es que, si la respuesta a los eventos

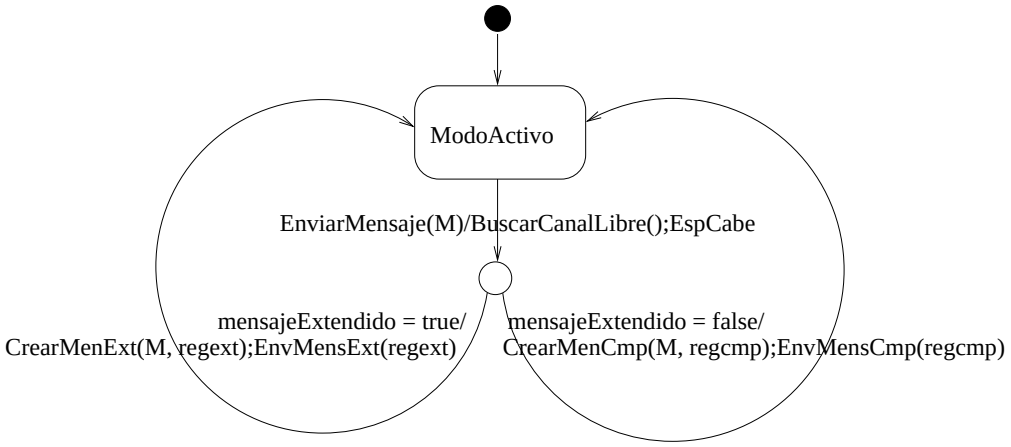

Figura 4.8: Máquinas de estados del objeto`DComboSal`.

implica el acceso a los mismos recursos, hace falta establecer un control de acceso a los recursos para que no se modifiquen concurrentemente de forma descontrolada. El otro inconveniente radica en si realmente las regiones de un estado concurrente pueden atender concurrentemente distintos eventos. En [88, sección 2.12.4.7] se indica que:

> es posible definir la semántica de las máquinas de estados de tal manera que la restricción de *ejecución hasta la finalización* se aplique concurrentemente a las regiones ortogonales de un estado compuesto en vez de a la máquina de estados en su totalidad. Esto permitiría relajar las restricciones sobre la serialización de eventos. Sin embargo, dicha semántica es muy sutil y difícil de implementar. Por tanto, la semántica dinámica definida en este documento se basa en la premisa de que un paso de *ejecución hasta la finalización* se aplica a toda la máquina de estados e incluye los pasos concurrentes llevados a cabo por regiones concurrentes de la configuración de estados activos.

Tampoco la semántica definida en el capítulo 3 permite la respuesta con-

currente a eventos.

Una de las tareas más importantes que hay que desarrollar en los aspectos no funcionales de la fase de diseño es el estudio de los eventos internos y la realización de la tabla de eventos/transiciones explicada en la sección 2.2. Distinguimos diferentes modos de operación y el análisis se lleva a cabo en el contexto de cada uno de los modos de operación. En la tabla 4.2 podemos ver la tabla de eventos/transiciones que contiene todos los eventos de nuestra aplicación simplificada. Los números relacionados con el tiempo son los especificados en los requisitos del sistema y dependen del microprocesador en el que se instale el software. Estos tiempos fueron suministrados por la empresa Alcatel.

Hemos considerado tres modos de trabajo: inactivo, llamada entrante y llamada saliente. Los eventos concurrentes implicados en los modos funcionales se muestran en la tabla 4.3.

La tabla 4.4 contiene toda la información necesaria para aplicar las técnicas de tiempo explicadas en [15, sección 2.4].

Como se comentó en la sección 1, la fase de simulación y validación no está aún disponible para UML. Este ejemplo se desarrolló también con SDL y en esa versión sí se realizaron las fases de simulación y verificación para solventar los errores del sistema desarrollado. Los modelos de validación ayudaron a completar el sistema detectando los fallos de diseño tales como bloqueos y consumo implícito de señales usando algoritmos de *bit state*. También verificamos nuestro sistema desarrollando casos de uso en MSC para comparar con los resultados de la simulación del sistema. Estas acciones se han comentado en la sección 2. En los aspectos no funcionales, obtuvimos un análisis de tiempo real preliminar usando las tablas de eventos/transiciones con los diferentes modos operativos de la fase de diseño. La tabla 4.5 mues-

Nombre del Evento	Tipo	Patrón de Ocurrencia	Requisitos de tiempo	Respuesta/transición
lectura de teclado	interno	periódico, 7.82 ms	Hard, 7.82 ms	leer tecla → leer tecla → leer tecla → leer tecla → verificar tecla
encender diodo	interno	periódico, 500 ms	Soft, 100 ms	esc on → activar registro → esc puerto
apagar diodo	interno	periódico, 500 ms	Soft, 150 ms	esc off → desabilitar registro → esc puerto
escanear canales	interno	periódico, 0.122 ms	Hard, 0.122 ms	preparar escaneo → establecer nivel de *squelch* → cambiar canal
enviar mensaje al auricular	interno	periódico, 1000 ms	Hard, 500 ms	buscar canal libre → esc cabecera → esc datos
comprobar línea	interno	periódico, 0.122 ms	Hard, 0.01 ms	comprobar línea
llamada entrante	externo	periódico, 700 ms	Soft, 700 ms	comprobar línea → (buscar canal libre → esc. cabe → esc. datos → leer cabe → leer datos → verificar canal → verificar código de seguridad) → (esc. *on* → activar registro → esc. puerto) ‖ programación de registros del altavoz
cambiar código de seguridad	externo	periódico, 600 ms	Soft, 400 ms	nuevo código → (modificar valor → (grabar EEPROM ‖ (esc cabe → esc datos)))
llamada saliente	externo	periódico, 1100 ms	Soft, 1100 ms	leer cabe → leer datos → pedir línea → buscar canal libre → esc. cabe → esc. datos → leer cabe → leer datos → (programar línea ‖ (esc. on → activar registro → esc. puerto))
intercomunicación	externo	periódico, 1200 ms	Soft, 1200 ms	preparar intercomunicación → ((buscar canal libre → esc. cabe → esc. datos → leer cabe → leer datos → verificar canal → verificar código seguridad) ‖ (esc. on → activar registro → esc. puerto))

Cuadro 4.2: Tabla de eventos/transiciones.

Modo	Nombre del evento
inactivo	lectura de teclado encender diodo apagar diodo escanear canales comprobar línea
llamada entrante	lectura de teclado encender diodo apagar diodo llamada entrante
llamada saliente	lectura de teclado encender diodo apagar diodo llamada saliente comprobar línea

Cuadro 4.3: Eventos para los modos operativos *inactivo*, *llamada entrante* y *llamada saliente*.

Id. Acción	Tiempo de bloqueo	Atómica	Tiempo Usado	Prioridad
leer tecla	n/d	S	0.010 ms	50
verificar tecla	n/d	S	0.008 ms	50
esc. off	0.083 ms	S	0.001 ms	40
inhabilitar registro	n/d	S	0.017 ms	40
esc. on	n/d	S	0.001 ms	39
habilitar registro	n/d	S	0.017 ms	39
esc. puerto	n/d	S	0.055 ms	39
preparar escaneo	n/d	S	0.001 ms	55
establecer nivel de *squelch*	n/d	N	0.065 ms	55
cambiar canal	n/d	N	0.050 ms	55
buscar canal libre	230 ms	N	1.770 ms	38
escribir cabecera	n/d	N	0.018 ms	38
escribir datos	n/d	N	0.028 ms	38
leer datos	n/d	S	230 ms	37
escribir cabecera	n/d	S	280 ms	37
verificar canal	n/d	N	0.133 ms	37
verificar código seguridad	n/d	N	0.400 ms	37
nuevo código	n/d	N	0.004 ms	33
modificar valor	n/d	N	0.001 ms	33
grabar EEPROM	n/d	S	120 ms	33
comprobar línea	n/d	S	0.002 ms	60
programar registros altavoz	n/d	N	0.040 ms	32
pedir línea	n/d	N	0.247 ms	30
programar línea	n/d	N	0.040 ms	30
preparar intercomunicación	n/d	S	0.001 ms	29

Cuadro 4.4: Tabla de transiciones.

Modo inactivo

Nombre del evento	Plazo	Tiempo de respuesta
lectura de teclado	7.82 ms	5.85 ms
encender diodo	100 ms	76.37 ms
apagar diodo	150 ms	77.59 ms
escanear canales	0.122 ms	0.121 ms
comprobar línea	0.01ms	0.005 ms

Modo de llamada entrante

Nombre del evento	Plazo	Tiempo de respuesta
lectura de teclado	7.82 ms	0.053 ms
encender diodo	100 ms	0.19 ms
apagar diodo	150 ms	0.20 ms
llamada entrante	700 ms	515.92 ms

Modo de llamada saliente

Nombre del evento	Plazo	Tiempo de respuesta
encender diodo	100 ms	0.156 ms
apagar diodo	150 ms	0.166 ms
comprobar línea	0.01ms	0.005 ms
llamada saliente	1100 ms	1067.37 ms

Cuadro 4.5: Tiempo de respuesta para las acciones de los modos operativos.

tra el análisis preliminar de tiempo real para los modos operativos inactivo, llamada entrante y llamada saliente.

CAPÍTULO 5

Conclusiones y trabajo futuro

El contenido de este trabajo está incluido en el marco de las investigaciones realizadas por parte de los miembros del grupo de Ingeniería del Software del Departamento de Lenguajes y Ciencias de la Computación, centrados en el uso y adaptación de técnicas avanzadas de Ingeniería del Software en el desarrollo de sistemas empotrados de tiempo real.

El objetivo fundamental de la metodología definida en el capítulo 4 es proporcionar a los desarrolladores de sistemas empotrados de tiempo real herramientas de especificación y diseño que incluyan las metodologías genéricas más modernas, como el empleo de la programación orientada a objetos, el uso sistemático de lenguajes gráficos de especificación y diseño y la utilización de herramientas automáticas para ayudar a establecer la corrección del sistema mediante la simulación y verificación del modelo y la generación automática de código.

La metodología presta especial atención a aspectos de desarrollo característicos de los sistemas empotrados de tiempo real y que no se contemplan en otras metodologías:

- El tratamiento de la parte física del sistema. En cada fase de la meto-

dología se detallan las acciones que se han de llevar a cabo para tener en cuenta sus características.

- Los aspectos no funcionales, con un mayor énfasis en los requisitos temporales. Los aspectos temporales se tienen en cuenta no sólo permitiendo y facilitando su especificación, sino también incluyendo herramientas para su análisis.

- Aspectos de concurrencia y prioridades, incluyendo el estudio de la asignación de prioridades a los eventos y los mecanismos necesarios para evitar situaciones incorrectas como la inversión de prioridades. Concretamente, se ha integrado el análisis de planificabilidad basado en prioridades fijas al ciclo de desarrollo.

Como se explica en el capítulo 4, respecto a la metodología original, propuesta en [10] y [15], se ha sustituido SDL por las máquinas de estados de UML que, al carecer de una semántica formal, no permiten aprovechar las ventajas derivadas del uso de herramientas automáticas. Por este motivo, en los capítulos 2 y 3 se han definido semánticas formales para las acciones y las máquinas de estados de UML. La estrategia seguida para la definición de estas semánticas es la del metamodelado, un método de modelado que, para definir los elementos del lenguaje de modelado, hace uso de los elementos del propio lenguaje (clases, objetos, asociaciones, etc.), pero a otro nivel de abstracción. El metamodelado es la opción escogida por la OMG para definir su semántica de UML. La diferencia fundamental entre nuestra semántica para las acciones y las máquinas de estados y la definida por la OMG es que la nuestra incluye la semántica dinámica, o dominio semántico, como uno de los tres aspectos básicos de la semántica. Los otros dos aspectos son la sintaxis estática, o sintaxis abstracta, y la aplicación semántica entre la sin-

taxis estática y la dinámica. En la definición de la semántica de la OMG, la semántica dinámica no está definida de manera rigurosa, sino que se expresa en lenguaje natural el funcionamiento esperado de los elementos del lenguaje de modelado.

Como el propósito fundamental de la definición de la semántica para las máquinas de estados y las acciones es su uso en una metodología de desarrollo de sistemas de tiempo real, se han ampliado ambos conceptos, tanto las acciones como las máquinas de estados, definiendo clases especializadas, que incluyen los conceptos y la información necesaria para que esos elementos puedan ser usados en los análisis de planificabilidad.

En el contexto de las máquinas de estados se han definido nuevas clases para poder especificar y tratar correctamente las cuestiones relativas al tiempo. También se han ampliado las clases relativas a los eventos con conceptos temporales.

Son muchos los puntos aún abiertos en este trabajo. Respecto a la metodología propuesta en el capítulo 4. Aunque se ha revelado eficaz en su aplicación industrial en un ejemplo real, como se explica en la sección 3 de ese capítulo, es necesario su uso en un conjunto amplio de sistemas, con características variadas, que confirmen su adecuación o que desvele aspectos que no estén bien cubiertos o que sean mejorables en función de las propiedades del sistema a desarrollar.

La semántica desarrollada en el capítulo 2 para las acciones tampoco se puede considerar un trabajo finalizado. Esta semántica puede ser un buen punto de partida para la definición de un conjunto de acciones más completo, cuya semántica detallada y final se base en este enfoque. El conjunto de acciones definido aquí puede tomarse como un conjunto básico sobre el que definir otras acciones, más concretas o especializadas, de manera que la

semántica del conjunto de acciones total del sistema se haga de una manera jerárquica. También pueden definirse nuevos conjuntos de acciones cuya semántica se defina con un esquema análogo al que presentamos aquí, de manera que cada conjunto de acciones se aplique a un determinado tipo de sistemas con características particulares.

La semántica desarrollada en el capítulo 3 para las máquinas de estados también tiene elementos susceptibles de mejorar. Como se explica en la sección 2 de dicho capítulo, algunos elementos de las máquinas de estados, como las actividades, los estados de historia o la lista de eventos postergados no han sido incluidos en el modelo de las máquinas de estados considerados para evitar presentar un modelo demasiado complejo. Obviamente, si se pretende que esta semántica sea de utilidad en el tratamiento de sistemas reales complejos, ha de definirse la semántica para todos los elementos de las máquinas de estados.

Otra cuestión pendiente, y de gran importancia, es el estudio de la relación entre la semántica de los elementos que hemos definido en este trabajo y la semántica de los demás diagramas definidos en UML. La cuestión de la coherencia entre distintos diagramas UML que cubren diferentes aspectos de los mismos elementos de un sistema ha sido apuntada por numerosos autores. Nuestra semántica toma como base la semántica definida para MML [27], que incluye diagramas estáticos como los diagramas de clases y los de objetos, razón por la que la coherencia entre la semántica propuesta en este trabajo y la de esos diagramas está bien establecida. Sin embargo, no hay ni en este trabajo, ni en [27] una semántica para, por ejemplo, los diagramas de actividad, que deben usarse en la simulación y validación de escenarios.

Hay otras facetas de sincronización que deben establecerse con más claridad, como la relación entre los aspectos estáticos y los dinámicos del sistema.

Uno de los elementos no considerados en nuestra propuesta de semántica para las máquinas de estados son las actividades, acciones que se ejecutan durante el tiempo en que una máquina de estados permanece en un estado. En el modelo presentado, cada instancia de una máquina de estados estará ligada a un objeto. La posibilidad de modificar el objeto de un estado a través de dos mecanismos en principio autónomos, las llamadas a métodos de la clase a la que pertenece el objeto el envío de instancias de eventos que serán procesados por la máquina de estados ligada al objeto, hace que sea necesario plantearse cuál es la forma en que una acción de un tipo repercute en el estado de ambos conceptos relacionados (objeto y máquina de estados). Por ejemplo, cómo tener en cuenta la ejecución de las acciones o de las actividades que modifican los valores de los *slots* del objeto al que se asocia la máquina de estados en la secuencia de *snapshots* del objeto en el dominio semántico.

Otro tema interesante a estudiar es el de la expresividad del metamodelado como mecanismo de modelado de la semántica de aspectos dinámicos (acciones y máquinas de estados) y compararla con otras técnicas de modelado, como las semánticas para máquinas de estados con otros formalismos nombradas en la sección 1.1 del capítulo 3. Entre las ventajas del metamodelado podemos citar, y de hecho es el principal motivo que cita la OMG para elegirlo en sus normas, la facilidad de comprensión, ya que no se basa en formalismos con una notación matemática compleja, sino en conceptos familiares, como clases, objetos, asociaciones, etc.

Entre los inconvenientes hay que resaltar que la base formal en la que se asienta el metamodelado no está tan desarrollada como la que usan otras propuestas (lógicas temporales, redes de Petri, máquinas de estados extendidas, etc.), lo que provoca que se sigan discutiendo distintas posibilidades y alternativas en el metamodelado (como, por ejemplo, la conveniencia del

metamodelado estricto, como sugiere la OMG).

Otro inconveniente destacable que hemos detectado durante la realización de este trabajo es la capacidad de establecer propiedades de viveza con los diagramas de clases y las reglas de corrección expresadas en OCL. Ciertas situaciones que se pueden expresar como *cuando se cumpla una condición, lo siguiente que debe ocurrir es esta acción* no son fácilmente expresables en OCL. Un ejemplo de estas situaciones se da en las transiciones espontáneas en las máquinas de estados. Cuando todas las regiones de un estado concurrente acaban, porque llegan a un estado final, el estado concurrente debe acabar ejecutando una transición espontánea. Una situación análoga se da cuando acaba la actividad que se ejecuta en un estado. El fin de la actividad fuerza, según la semántica de la OMG, la salida del estado a través de la ejecución de una transición espontánea. Pensamos que los diagramas de clases y las reglas de corrección en OCL que hemos empleado para definir el dominio semántico de los elementos dinámicos, y por tanto la semántica dinámica, han demostrado ser adecuados para especificar si un diagrama de objetos que representa una secuencia en el funcionamiento de un sistema es correcto o no desde el punto de vista de su definición semántica, pero que estos mismos elementos de modelado adolecen de herramientas para forzar la ocurrencia de secuencias como las mencionadas de las transiciones espontáneas.

Bibliografía

[1] pUML rfi submission with regard to UML 2.0, howpublished = http://www.cs.york.ac.uk/puml/papers/RFIResponse.PDF, key = 2Usubmission.

[2] M. Abadi and L. Lamport. And old-fashioned recipe for real time. *ACM Transactions of Programming Languages and Systems*, 5(16):1543 – 1571, sept 1994.

[3] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] Action semantics consortium: Response to omg rfp ad/98-11-01. action semantics for the UML. Revised September 5, 2000 (2000) www.umlactionsemantics.org.

[5] J. Allen and G. Ferguson. Actions and events in interval temporal logic. Technical Report TR-URCSD 521, University of Rochester, Rochester, NY, 1994.

[6] K. Altisen, G. Goesler, and J. Sifakis. A methodology for the construction of scheduler systems. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, Pune (India), sep 2000. Springer-Verlag.

[7] R. Alur and T. Henzinger. A really temporal logic. In *Proceedings of the 30th IEEE Conference on Foundations of computer Science*, Los Alamitos, CA, 1989. IEEE Computer Sociery Press.

[8] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. An analysable execution model for sdl for embedded real-time systems. In *Workshop on Real-Time Programming (WRTP99)*. Elsevier, 1999.

[9] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. Embedded real-time development using sdl. In *IEEE Real Time Systems Symposium (RTSS99). WIP sessions*, 1999.

[10] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. Integrating schedulability analysis and sdl in an object-oriented methodology. In *Proceedings of 9th SDL Forum*, pages 241 – 259, Montreal, jun 1999. Elsevier.

[11] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. Schedulability analysis in real-time embedded systems specified in SDL. In *Proceedings of 25th IFAC Workshop on Real-Time Programming*, pages 117 – 123, Palma de Mallorca (España), may 2000. Elsevier.

[12] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. Sdl and hard real-time: New design and analysis techniques. In *Workshop on SDL and MSC (SAM00)*, 2000.

[13] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. Deriving hard real time system implementations directly from SDL specifications. In *Proceedings of 9th International Symposium on Hardware/Software Codesign*, pages 128 – 134, Copenague, apr 2001. ACM press.

[14] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. Integrating schedulability analysis and design techniques in sdl. *Real Time Systems Journal*, 3(24):267 – 302, may 2003.

[15] J. Álvarez, M. Díaz, L. Llopis, E. Pimentel, and J. Troya. An object oriented methodology for embedded real-time systems. *The Computer Journal*, 46(2):123 – 145, 2003.

[16] J. M. Álvarez, T. Clark, A. Evans, and P. Sammut. An action semantics for the mml. In *UML 2001*, 2001.

[17] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A new UML profile for real-time system formal design and validation. In Gogolla and Kobryn [52], pages 287–301.

[18] D. Aredo. Semantics of UML statecharts in PVS. Technical Report Research Report 299, Department of Informatics, University of Oslo, 2000.

[19] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, San Francisco, 1995.

[20] L. Baresi and M. Pezzè. On formalizing uml with high-level petri nets. In G. Agha, F. de Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 2001.

[21] P. Bellini, R. Mattolini, and P.Ñesi. Temporal logics for real-time system specification. *ACM Computing Surveys*, 32(1):12 – 42, mar 2000.

[22] M. Ben-Ari, A. Pnuelli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 3(20):207 – 226, 1983.

[23] G. Booch. *Object-oriented analysis and design with applications*. Benjamin/Cummings Pub. Co., Redwood City, Calif., 1994.

[24] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Manual*. Addision-Wesley, 1999.

[25] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of uml state machines. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, volume 1912, pages 223–241, London, UK, 2000. Springer-Verlag.

[26] M. Bozga, S. Graf, L. Mounier, I. Ober, J.-L. Roux, and D. Vincent. Timed extensions for sdl. In Reed and Reed [99], pages 223–240.

[27] S. Brodsky, A. Clark, S. Cook, A. Evans, and S. Kent. A feasibility study in rearchitecting UML as a family of languages using a precise oo meta-modeling approach. Technical report, pUML group, 2000. http://www.puml.org/mmt.zip.

[28] A. Burns. How to verify a safe real-time system: The application of model checking and timed automata to the production cell case study. *Real Time Systems*, 24(2):135 – 151, mar 2003.

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

[30] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[31] A. Clark, A. Evans, and S. Kent. Engineering modelling languages: A precise oo meta-modeling approach. Technical report, pUML group, 2000. http://www.puml.org/mml.

[32] E. M. Clarke, E. A. Emerson, and A. Sistla. Automatic verification of finite-state concurrent sistems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 2(8):244 – 263, april 1986.

[33] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method*. Prentice Hall, 1993.

[34] J.-P. Courtiat, C. Santos, C. Lohr, and B. Outtaj. Experience with rt-lotos, a temporal extension of the lotos formal description technique. *Computer Communications*, 23(12):1104 – 1123, jul 2000.

[35] M. L. Crane and J. Dingel. On the semantics of uml state machines: Categorization and comparison. Technical Report 2005-501, School of Computing, Queen's University, Kingston, Ontario, Canada, 2005.

[36] K. Diethers, U. Goltz, and M. Huhn. Model checking UML statecharts with time. In J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 35–52. Technische Universität München, Institut für Informatik, 2002.

[37] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453 – 457, aug 1975.

[38] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[39] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Object Technology Series. Addison Wesley, 1999.

[40] J. Drake, M. G. Harbour, and J. L.Medina. Vista uml de tiempo real de sistemas diseñados bajo una metodología orientada a objetos. Technical report, Group of Computers and Real-Time Systems. University of Cantabria (Internal Report), 2001. http://mast.unican.es/umlmast.

[41] D. D'Souza and A. C. Wills. *Object Components and Frameworks with UML – The Catalysis Approach*. Addison-Wesley, 1998.

[42] W. Dulz, S. Grughl, L. Kerber, and M. Söllner. Early performance prediction of SDL/MSC specified systems by automatic synthetic code generation. In *Proceedings of the 9th SDL Forum*, pages 457 – 473, Montreal, Canada, jun 1999. Elsevier.

[43] *Enhancements to LOTOS (E-LOTOS)*, 2001.

[44] A. Ek. The somt method. Technical report, Telelogic AB, 1995.

[45] E. A. Emerson and J. Y. Halpner. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 1(33):151 – 178, jan 1986.

[46] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In A. Evans, S. Kent, and B. Selic, editors, *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[47] M. K. et al. *A Practitioner's Handbook for Real-time Analysis*. Kluwer Academic Publishers, 1993.

[48] E. T. S. I. (ETSI). *ETSI ES 201 873-1 V3.1.1 (2205-06) Methods for Testing and Specification(MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.*

[49] European Space Agency. *HOOD Reference Manual Issue 3.1, HRM/91-07/V3.1*, sep 1991.

[50] http://www.csverilog.com, 2001.

[51] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a uml statechart diagrams kernel and its extension to multicharts and branching time model-checking. *J. Log. Algebr. Program.*, 51(1):43–75, 2002.

[52] M. Gogolla and C. Kobryn, editors. *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*. Springer, 2001.

[53] M. Gogolla and F. P. Presicce. State diagrams in UML: A formal semantics using graph transformation. In *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*, 1998.

[54] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison-Wesley, 2000.

[55] M. González, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priorities. In *Proceedings of the IEEE Real Time Systems Symposium*, pages 116 – 128, San Antonio, dec 1991. IEEE Computer Society Press.

[56] O. M. Group. UML profile for schedulability, performance and time specification. version 1.1. formal/05-01-02. Technical report, Object Management Group, jan 2005.

[57] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[58] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, oct 1969.

[59] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666 – 677, oct 1978.

[60] ISO. Information processing systems – open systems interconection – lotos – a formal description technique based on the temporal ordering of observational behaviour. Technical Report IS-8807, ISO., Geneva, sep 1989.

[61] ITU. Specification and description language. Technical Report ITU-T Z.100, International Telecommunications Union, 1996.

[62] M. Jackson. *Principles of Program Design*. Academic Press Inc, 1975.

[63] M. Jackson. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*. Prentice Hall, 1996.

[64] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison Wesley, 1992.

[65] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 9(12):890 – 904, sep 1986.

[66] C. B. Jones. *Sistematic Software Development using VDM.* Prentice Hall, 1989.

[67] M. Joseph and P. Pandya. Finding response time in a real-time system. *The Computer Journal*, 29:390 – 395, 1986.

[68] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[69] A. Kleppe and J. Warmer. Unification of static and dynamic semantics of uml. a study in redefining the semantics of the uml using the puml oo meta modelling approach. Technical report, Klasse Objecten, 2001.

[70] A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. Technical Report Technical Report 2002-11, Institut fur Informatik, Universitat Augsburg, Reisensburg, Germany, 2002.

[71] T. Kolloch and G. Färber. Mapping an embedded hard real time systems SDL specification to an analysable task network - a case study. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 156 – 166, Montreal (Canada), jun 1998. Springer-Verlag.

[72] S. Kuske. A formal semantics of uml state machines based on structured graph transformation. In Gogolla and Kobryn [52], pages 241–256.

[73] L. Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 3(16):872 – 923, may 1994.

[74] K. Lano, J. Bicarregui, and A. Evans. Structured axiomatic semantics for uml models. In *Rigorous Object-Oriented Methods*, Workshops in Computing. BCS, 2000.

[75] L. Lavazza, G. Quaroni, and M. Venturelli. Combining UML and formal notations for modelling real-time systems. In *ESEC / SIGSOFT FSE*, pages 196–206, 2001.

[76] N. Leveson. *Safeware*. Addison-Wesley, 1995.

[77] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20:46 – 71, 1973.

[78] J. Mathai. *Real-time Systems. Specification, Verification and Analysis*. Prentice Hall International, 1996.

[79] J. McDermid. *Assurance in High Integrity Software*, chapter 10. Pitman, 1989.

[80] P. M. Melliar-Smith. Extending interval logic to real time systems. In *Proceedings of the Conference on Temporal Logic Specification*, pages 224 – 242, UK, 1987. Springer-Verlag.

[81] R. Milner. *Communications and Concurrency*. Prentice-Hall, 1989.

[82] A. Mitschele-Thiel and B. Müller-Clostermann. Performance engineering of SDL/MSC systems. In *Proceedinds of the 8th SDL Forum*, pages 23 – 26, Evry, France, sep 1997. Elsevier.

[83] A. Mitschele-Thiel and F. Slomka. Codesign with sdl/msc. In *First International Workshop on Conjoint Systems Engineering, Tölz, Bad*, 1997.

[84] *MetaObjectFacility(MOF) Specification. Version 1.4*, 2003. http://www.omg.org/docs/formal/02-04-03.pdf.

[85] *ITU recommendation Z.120. Message Sequence Chart (MSC).* Geneva (Switzerland), 2000.

[86] R. Münzenberger, F. Slomka, M. Dörfel, and R. Hofmann. A general approach for the specification of real-time systems with sdl. In Reed and Reed [99], pages 203–222.

[87] Object Management Group. *formal/03-03-13 (UML 1.5 chapter 6 - Object Constraint Language Specification)*, 2003.

[88] Object Management Group. *OMG Unified Modeling Language Specification*, March 2003. Version 1.5.

[89] *UML 2.0 OCL Specification*, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf.

[90] OMG. Meta-object facility. Technical Report Document ad/97-08-14, DSTC, OMG, sep 1997.

[91] J. S. Ostroff and W. Wonham. Modeling and verifying real-time embedded computer systems. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 124 – 132, Los Alamitos, CA, 1987. IEEE Computer Sociery Press.

[92] S. Owre, N. Shankar, and J. Rushby. The PVS specification language. Technical report, Computer Science Laboratory, SRI International, feb 1993.

[93] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. of IFIP Congress 62*, pages 386–390, Amsterdam, 1963. North Holland Publ. Comp.

[94] A. Pnuelli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46 – 57, Los Alamitos, CA, 1977. IEEE Computer Society Press.

[95] B. Potter, J. Sinclair, and D. Till. *An Introduction to Forma Specification and Z*. International Series in Computer Science. Prentice Hall, 1991.

[96] F. Rallis. Octopus/uml: Combining objects with real-time. pages 480 – 489, Washington, DC, USA, 1999. IEEE Computer Society.

[97] http://www.ilogix.com, 2001.

[98] R. Razouk and M. Gorlick. Real-time interval logic for reasoning about executions of real-time programs. *SIGSOFT Software Engineering Notes*, 8(14):10 – 19, dec 1989.

[99] R. Reed and J. Reed, editors. *SDL 2001: Meeting UML, 10th International SDL Forum Copenhagen, Denmark, June 27-29, 2001, Proceedings*, volume 2078 of *Lecture Notes in Computer Science*. Springer, 2001.

[100] C. Rossi, M. Enciso, and I. P. de Guzmán. Formalization of UML state machines using temporal logic. *Software and System Modeling*, 3(1):31–54, 2004.

[101] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[102] J. Rumbaugh, G. Booch, and I. Jacobson. *The Unified Modeling Language. User Guide*. Addison-Wesley, 1999.

[103] M. Saksena, P. Freedman, and P. Rodziewick. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *Proceedings of the 18th IEEE Real-Time System Symposium*, pages 240 – 251, San Francisco, EE.UU., dec 1997. IEEE Computer Society Press.

[104] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. In *Proceedings of 12th IEEE Euromicro Conference on Real-Time Systems*, pages 101 – 109, Estocolmo, Suecia, jun 2000. IEEE Computer Society Press.

[105] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multitasking implementations from real-time object oriented models. In *Proceedings of International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC00)*, pages 360 – 368. IEEE Computer Society Press, mar 2000.

[106] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewick. Schedulability analysis for automated implementations of real-time object-oriented methods. In *Proceedings of the 19th IEEE Real-Time System Symposium (RTSS98)*, pages 92 – 102, Madrid (Spain), dec 1998. IEEE Computer Society Press.

[107] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning. In *Proceedings of the Second ACM Symposium on Principles of Distributed Programming*, pages 173 – 186, New York, NY, 1983. ACM Press.

[108] *ITU recommendation Z.100. Specification and Description Language (SDL).* Geneva (Switzerland), 2000.

[109] B. Selic. *Models, Software Models and UML*, pages 1 – 15. Kluwer Academic Publishers, 2003.

[110] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley, 1994.

[111] B. Selic and J. Rumbough. Using UML for modeling complex real-time systems. Technical report, OjbecTime Limited and Rational Software Corporation, mar 1998.

[112] S. Shankar and S. Asa. Formal semantics of UML with real-time constructs. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 60–75. Springer, 2003.

[113] H. Simpson. The mascot method. *Software Engineering Journal*, 1(3):103 – 120, may 1986.

[114] S. Spitz, F. Slomka, and M. Dörfel. SDL* - an annotated specification language for engineering multimedia communication systems. In *Sixth Open Workshop on High Speed Networks, Stuttgart*, 1997.

[115] J. M. Spivey. *The Z Notation. A Reference Manual.* International Series in Computer Science. Prentice Hall, second edition, 1992.

[116] S. Stepney, R. Barden, and D. Cooper. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, 1992.

[117] L. M. L. Torres. *Integración de Análisis de Tiempo Real en la Técnica de Descripción Formal SDL.* PhD thesis, Universidad de Málaga, 2002.

[118] *UML 2.0 Infrastructure Specification*, 2003. http://www.omg.org/docs/ptc/03-09-15.pdf.

[119] *UML 2.0 Superstructure Specification*, 2003. http://www.omg.org/docs/ptc/03-08-02.pdf.

[120] J. Woodcock and J. Davies. *Using Z Specification, Refinement, and Proof.* International Series in Computer Science. Prentice Hall, 1996.

[121] www.2uworks.org. Unambiguous UML (2u) 3rd revised submission to UML 2 infrastructure rfp, version 1.0. omg document ad/2003-01-08. http://www.2uworks.org/uml2submission/1.0/uml2InfraSubmission1.pdf, jan 2003.

[122] www.2uworks.org. Unambiguous UML (2u) 3rd revised submission to UML 2 superstructure rfp, version 0.2. omg document ad/2002-12-23. http://www.2uworks.org/uml2submission/super0.2/uml2SuperSubmission02.pdf, jan 2003.