UNIVERSIDAD DE GRANADA
E.T.S. de Ingeniería Informática
Dpto. de Lenguajes y Sistemas Informáticos

Un modelo arquitectónico evolutivo para sistemas software basados en agentes

Por:

Dña. Patricia Paderewski Rodríguez

Dirigida por:

Dr. D. José Parets Llorca
Dra. Dña. Mª José Rodríguez Fórtiz

Tesis presentada para la
obtención del grado de Doctor

Granada, Abril de 2003

Un modelo arquitectónico evolutivo para sistemas software basados en agentes

TESIS DOCTORAL

Presentada por:
Dña. Patricia Paderewski Rodríguez

Dirigida por los doctores:
D. José Parets Llorca
Dña. Mª José Rodríguez Fórtiz

UNIVERSIDAD DE GRANADA
Dpto. de Lenguajes y Sistemas Informáticos

Granada, Abril de 2003

A mi madre y a Francis

Agradecimientos:

La verdad es que necesitaría mucho espacio y tiempo para poner a todas aquellas personas que, de un modo u otro, han influido en mi trabajo. Desde que empecé a trabajar en la universidad, he conocido mucha gente que me ha ayudado y me ha animado a seguir adelante. A veces, el camino se hace tan largo que es fácil pensar en coger otro que sea un poco más llevadero.

A los primeros que quiero expresar mi agradecimiento explícitamente son a mis directores de tesis, a José Parets Llorca y a Mª José Rodríguez Fórtiz sin los que este trabajo no hubiera llegado a su fin. Ha sido una suerte tenerlos de directores de tesis y poder trabajar con ellos.

A Francis, por el apoyo que siempre me ha prestado y por su meticulosa revisión del trabajo.

También deseo agradecer su colaboración a todos los miembros de mi grupo de investigación, GEDES, a Ana Anaya, Jesús Torres, Mavi Hurtado, Nuria Medina, Fernando Molina y Lina García. También a Germán Sánchez que, aunque ahora ya no es parte de GEDES, ha sido un gran compañero.

A muchos compañeros del departamento y a mis amigos, no hace falta decir explícitamente sus nombres, ellos ya lo saben.

Y por último a mi familia, los que siempre han estado a mi lado y siempre estarán aunque les haya dedicado menos atención, sobre todo últimamente.

P.P.R.
Granada, Abril de 2003

Indice General

Índice de Figuras, Tablas y Cuadros

Figuras

Tablas

Cuadros

Siglas utilizadas

AE Agente Elemental

ADL *Architectural Description Language*

AOP *Agent-Oriented Programming*

ASD Arquitectura del Software Dinámica

CBSE *Component-Based Software Engineering*

CSP *Communicating Sequential Processes*

E Evaluator

GEDES Grupo de Especificación, Desarrollo y Evolución de Software (Departamento de Lenguajes y Sistemas Informáticos de Granada)

HEDES Herramienta de Especificación, Desarrollo y Evolución de Software

LTP Lógica Temporal de Predicados

MEDES Método de Especificación, Diseño y Evolución de Software

MFH Historia Funcional del Metasistema (*Metasystem Functional History)*

MS MetaSistema

MSH Historia Estructural del Metasistema (*Metasystem Structural History*)

SGBD Sistemas de Gestión de Bases de Datos

SFH Historia Funcional del Sistema (*System Functional History*)

SOD Sistema Operativo Distribuido

SSH Historia Estructural del Sistema (*System Structural History*)

OcuR *Ocurrence Receiver*

OOP *Object-Oriented Programming*

PDN Notificador Dinámico basado en Pre-condiciones (*Pre-condition Dynamic Notifier*)

POO Programación Orientada a Objetos

RdPC Red de Petri Coloreada

SD Subsistema (o sistema) de Decisión (*Decision Subsystem*)

SG Sistema Genético (*Genetic System*)

SS Sistema Software

TDL Lenguaje de Descripción de Transacciones (*Transactions Description Language*)

TM *Transaction Manager*

UML *Unified Modeling Language*

CAPITULO 1

Introducción

En este primer capítulo vamos a introducir las principales motivaciones que han dado lugar al desarrollo de este trabajo. Como podremos comprobar, en él encontramos relacionados distintos temas de investigación como son: la evolución del software, las arquitecturas del software y los patrones de diseño, los agentes y los sistemas multi-agentes, y los modelos y lenguajes de coordinación. Esta diversidad de temas es necesaria cuando hablamos del desarrollo de sistemas software que pretenden modelar sistemas reales.

Desde hace años, la complejidad de los sistemas que necesitamos modelar ha crecido bastante, la aparición de Internet, la distribución de los distintos recursos tanto hardware como software dentro de las empresas, los continuos cambios en la tecnología, la necesidad de reutilizar sistemas antiguos y muchas otras causas, amplifican la complejidad del desarrollo de los sistemas software actuales.

Por otro lado, se pretenden crear sistemas software con calidad. Esta calidad se consigue gracias a la utilización de un proceso de desarrollo natural, flexible y evolutivo. Debemos desarrollar sistemas que sean capaces de adaptarse a los cambios que puedan sufrir mientras está en funcionamiento, siempre preservando su integridad y consistencia. Para ello se deben proporcionar a los equipos de desarrollo de software modelos y herramientas adecuados que permitan construir los sistemas

software de una forma gradual, de forma que puedan ser modificados en cualquier momento, según las necesidades de sus usuarios.

1.1 Motivaciones y Objetivos.

A continuación, vamos a centrarnos en los principales motivos que nos han llevado a realizar este trabajo para, después, especificar los objetivos que nos hemos planteado en la elaboración de esta tesis.

1.1.1 Motivaciones.

Frecuentemente, los sistemas software que se desarrollan no se corresponden totalmente con la realidad, ya que los sistemas del mundo real no son "funcionalmente" estáticos sino dinámicos y, normalmente, los productos generados y entregados a los usuarios son estáticos. Son muchos los motivos que hacen que un sistema software deba modificarse después de su puesta en funcionamiento y entrega al usuario:

- La producción de errores durante el desarrollo del software que hay que subsanar.
- La necesidad de mejoras en ciertos aspectos no funcionales del sistema como por ejemplo, el rendimiento y la usabilidad.
- La necesidad de añadir al sistema nuevas funciones que son solicitadas por los usuarios o de adaptar las ya existentes porque hayan variado los requisitos iniciales.
- Los cambios en el entorno y la necesidad de adaptación a nuevos usuarios con necesidades o habilidades especiales.

Hasta hace poco, todas estas modificaciones del sistema se encuadraban en la fase del ciclo de vida del software que llamamos mantenimiento. Sin embargo, el mantenimiento empieza después de la entrega y puesta en funcionamiento del producto. Nosotros pensamos que un producto siempre está modificándose, desde las fases iniciales de su ciclo de vida, y creemos que el desarrollo del software no acaba hasta que éste se desecha por sus usuarios. Nuestra principal motivación es considerar la evolución, en contraposición con el mantenimiento, como parte integrante de cada una de las fases del proceso de desarrollo del software. Así mismo, contemplamos un ciclo de vida iterativo e incremental en lugar de uno secuencial.

Parte de nuestras hipótesis de trabajo se han utilizado en el desarrollo del modelo MEDES (Método de Especificación, Diseño y Evolución de Software) [Parets95] [Rodríguez00a] y su herramienta asociada HEDES

[Parets99a] [Rodríguez99]. Nosotros partimos de estas ideas con el fin de elaborar un modelo que nos permita construir sistemas software evolutivos. Una parte central de MEDES es la utilización de un Metasistema para realizar las modificaciones sobre un sistema software. El equipo de desarrollo, a través del Metasistema, podrá hacer evolucionar al sistema software cuando sea necesario. Se utiliza el Metasistema como una herramienta CASE (que, en este caso, era HEDES) que permite crear, destruir y modificar los distintos elementos de un sistema software.

El modelo MEDES se basa en la *Teoría del Sistema General* [Le Moigne90] [Parets94] que defiende que el funcionamiento y la estructura de un sistema están íntimamente ligadas. De esta forma, cuando un sistema evoluciona, esta evolución provocará un cambio en su estructura, además de en su funcionamiento. Este cambio se llevará a cabo mientras el sistema está funcionando.

Por tanto, necesitamos definir la estructura y arquitectura de un sistema software con el fin de facilitar su evolución y controlar si los cambios propuestos en él se pueden o no llevar a cabo en el sistema de tal forma que éste quede en un estado íntegro, consistente. Necesitamos una arquitectura heterogénea, ya que las arquitecturas o estilos arquitectónicos existentes [Shaw96] no abarcan todos los aspectos arquitectónicos que contemplamos en un sistema software basado en agentes. Por otro lado, necesitamos que la arquitectura sea dinámica. Los lenguajes de descripción de arquitecturas (ADLs) dinámicas existentes tales como *Rapide* [Rapide], *Dynamic Wright* [Allen98], *Dynamic ACME* [ACME], y *Darwin* [Darwin], normalmente se centran en las modificaciones existentes en los conectores de los componentes de un sistema y no le prestan atención a las modificaciones de los componentes en sí. El Metasistema, del que hemos hablado anteriormente, debería ser el encargado de crear y modificar la arquitectura de un sistema software y de verificar que siempre sea consistente.

Así mismo, podemos observar que los sistemas del mundo real se componen de un conjunto de entidades que pertenecen a dos grupos distintos: entidades activas que son las que realizan acciones para el sistema y le hacen funcionar (un vendedor, un operario, un controlador, etc.), y entidades pasivas, sobre las cuales se realizan dichas acciones (un coche, un documento, etc.). Además, las entidades activas suelen caracterizarse por ser autónomas, independientes y reactivas.

Pero ¿cómo se puede representar a una entidad activa? Puede ser un objeto, pero los objetos no poseen ciertas características como la de reaccionar ante un cambio en el entorno o actuar de forma autónoma. Las técnicas orientadas a objetos como la herencia y las relaciones

entre objetos son demasiado estáticas y de "caja blanca" [Andrade02]. El uso de la herencia requiere conocer, comprender y modificar internamente al objeto. En muchas circunstancias esto no es posible o no es aceptable, por tanto, los objetos no son adecuados para implementar a las entidades activas que suelen existir en los sistemas reales. Tampoco nos sirven los componentes [Jennings99] ya que no tienen por qué ser activos. Sin embargo, la programación orientada a agentes (AOP-*Agent-Oriented Programming*) [Shoham93] sí proporciona un conjunto de características inherentes a los agentes que nos permite construir cualquier sistema software evolutivo. Así, se entiende que un agente:

- Percibe su entorno y realiza acciones sobre él
- Es autónomo
- Está integrado en su entorno de tal forma que es capaz de reaccionar dinámicamente a los cambios que se producen en él
- Tiene un estado

Por tanto, los sistemas software, objeto de nuestro interés, se compondrán de un conjunto de agentes concurrentes que actúan de forma independiente. Llamamos a estos sistemas, sistemas software basados en agentes, ya que se van a caracterizar por las propiedades inherentes a los agentes que acabamos de especificar. Existen distintas clases de agentes [Brenner98]. En concreto, los agentes que vamos a necesitar deben ser cooperativos y adaptativos.

Pese a que existen metodologías para desarrollar sistemas basados en agentes [Wooldridge01] ninguna de ellas se preocupa de los aspectos de evolución, ven los agentes como una unidad de un alto nivel de abstracción y no profundizan en su composición. Algunas de ellas sí contemplan la creación y sustitución de un agente pero esto limita mucho las posibles modificaciones que puede sufrir un sistema software durante su vida y que le permiten adaptarse a las nuevas circunstancias de su entorno. Sin embargo, sí se centran más en la forma y en los lenguajes de comunicación entre agentes.

A veces, en un sistema se realizan tareas más complejas que requieren la intervención de más de un agente. Es necesario, por tanto, proporcionar un mecanismo de comunicación y de coordinación entre agentes. Este mecanismo no debe atentar contra la autonomía e independencia de los agentes. Pensamos que, para facilitar la evolución, al mismo tiempo que garantizamos la independencia, los agentes deberían comunicarse de forma no explícita y se debería registrar la historia de su funcionamiento y evolución de forma individual. Existen distintos modelos y lenguajes de coordinación [Papadopoulos98] [Omici01] pero ninguno de ellos se adapta completamente a nuestras necesidades. No debe ser un mecanismo donde sea necesario que los

agentes se conozcan unos a otros, pero se requiere que los agentes sepan lo que ocurre en el sistema para poder actuar en consecuencia. Por ello, se debe almacenar el estado del sistema en una estructura de datos común accesible por todos los agentes. Nos interesa un modelo de coordinación que combine la notificación de eventos con la comunicación indirecta de los agentes a través de una estructura de datos central.

A la hora de diseñar un nuevo modelo de comunicación y coordinación entre agentes, coincidimos con Gelernter y Carriero [Gelernter92] en que es mejor separar la funcionalidad propia de los agentes de la forma en la que los agentes se coordinan. Gracias a esto se obtiene una mayor portabilidad (reutilización), un soporte para la heterogeneidad de las entidades que deben ser coordinadas y se facilita la evolución.

Además, puesto que en un sistema se realizan acciones simples, que involucran a un único agente, y acciones complejas o transacciones, que involucran la realización de acciones por parte de varios agentes, creemos que los agentes deben contemplar ambos tipos de acciones y que éstas tendrán un tratamiento diferente. Las acciones complejas o transacciones deben cumplir las propiedades ACID *(Atomicy Consistency Isolation Durability)* [Tanembaum96].

1.1.2 Objetivos.

Una vez vistas las motivaciones que nos llevan a realizar este trabajo, vamos a concretar y comentar qué objetivos nos fijamos al comienzo y durante la realización de esta tesis. En el transcurso del trabajo desarrollado han surgido nuevos problemas y, por tanto, nuevos objetivos que no nos habíamos planteado inicialmente y que complementan y enriquecen la presente tesis.

1. Puesto que partimos del modelo MEDES, uno de los objetivos es ampliar este modelo y adaptarlo al desarrollo de sistemas basados en agentes. Para ello se ha de incorporar la entidad *agente* con las características que la definen: autonomía, reactividad, pro-actividad y sociabilidad.

2. Proporcionar un modelo de comunicación y coordinación entre agentes independiente del funcionamiento de los propios agentes mediante la utilización del concepto de separación de aspectos. Para ello se debe realizar un estudio de los distintos modelos de comunicación y coordinación existentes y comprobar si se adaptan o no a nuestras necesidades.

3. Incorporar a los agentes las acciones complejas o transacciones. Hasta ahora, en MEDES, las entidades activas, llamadas procesadores, sólo realizaban acciones simples. Esto implica incorporar los mecanismos necesarios para especificar y llevar a cabo las transacciones dentro de un sistema software.

4. Debido a la incorporación de las transacciones, modificar la estructura plana de los sistemas software generados por HEDES a una estructura en grafo. En este grafo debe existir un nodo, un agente, que represente al sistema y del cual descienden todos los demás agentes. Un agente puede delegar la realización de sus acciones en sus agentes hijos. Además, una transacción definida en un agente se constituye de acciones que realizan sus agentes hijos.

5. Se ha de facilitar la reutilización. Un sistema debe verse como un componente que puede integrarse fácilmente dentro de otro sistema software.

6. Modificar el sistema de decisión [Rodríguez00a] y adaptarlo a la creación y modificación de sistemas software basados en agentes. Las modificaciones deben mantener a los sistemas en un estado consistente, íntegro. Con este fin, se añadirá una lista de invariantes que se deben cumplir en cualquier sistema software.

7. Facilitar la tarea del equipo de desarrollo proporcionando una herramienta gráfica que permita tanto crear un sistema software como hacerlo evolucionar. Para ello se seguirá un ciclo de vida iterativo e incremental basado en prototipos.

8. Definir un lenguaje que permita describir la estructura de un sistema software basado en agentes según el modelo y la arquitectura propuesta.

1.2 Organización de la memoria.

A continuación vamos a comentar cómo se ha organizado el contenido del resto de esta tesis. Además de éste, se presentan nueve capítulos más que describimos brevemente:

Capítulo 2: Evolución del Software

La investigación de la evolución del software se centra en dos aspectos fundamentales [Lehman02]: el *cómo* se logra la evolución y el *qué /por qué* ocurre dicha evolución. Nuestras investigaciones se centran más en el *cómo*, y por tanto, nos interesa estudiar los modelos, los métodos y

los mecanismos necesarios para que un sistema pueda modificarse adecuadamente en respuesta a los cambios que surjan mientras éste está en funcionamiento.

Una vez definido el término evolución, ambiguo según el ámbito en el que se use, y las distintas causas que lo hacen necesario, estudiamos las distintas tendencias de investigación existentes relacionadas con la evolución del software. Continuamos describiendo los distintos mecanismos y modelos de evolución posibles en los sistemas software [Torres02], y determinamos cuáles de ellos están más relacionados con nuestro trabajo.

Capítulo 3: Estudio de Arquitecturas del Software y Patrones

Todo sistema software tiene una estructura que determina su funcionamiento. Esta estructura se ajusta a una arquitectura determinada y, por ello, se realiza un estudio de las distintas arquitecturas o estilos arquitectónicos que usualmente se utilizan en la construcción de sistemas software. Este estudio nos permitirá valorar cuál de ellas, si existe alguna, se adapta mejor a los sistemas software basados en agentes.

Puesto que dentro de un sistema existen aspectos que requieren un nivel de abstracción menor, también realizamos una revisión a los distintos patrones de diseño, concretamente, a aquellos que utilizan la tecnología orientada a objetos sobre la cual trabajamos.

Acabamos el tema centrándonos en las arquitecturas dinámicas y sus lenguajes de descripción de arquitecturas. Esto se debe a que la estructura de cualquier sistema software evolutivo ha de ser dinámica.

Capítulo 4: Agentes y Modelos de Comunicación y Coordinación

En este capítulo se hace una revisión del concepto de agente con el fin de determinar las características básicas que debe tener un agente, ya que no existe una definición totalmente consensuada.

También, y puesto que un sistema está compuesto de un conjunto de componentes que necesitan comunicarse y cooperar, estudiamos los diferentes modelos y lenguajes de comunicación y coordinación existentes. La finalidad de este estudio es elegir alguno de ellos para incorporarlo a nuestro modelo, o en su defecto, comprender qué características serían necesarias en un nuevo modelo de coordinación.

Capítulo 5: Definición del Modelo

Definimos de forma general el modelo del proceso de desarrollo de sistemas software basados en agentes. Para ello, primero se describen los principales conceptos que aporta el modelo MEDES.

A continuación se define lo que, en nuestro modelo, es un sistema software junto con la descripción de cada uno de los elementos que van a estar presentes en su estructura.

Presentamos un conjunto de problemas de funcionamiento de los agentes que componen un sistema y las soluciones propuestas. Uno de los problemas detectados origina la necesidad de definir un modelo de coordinación y comunicación entre agentes. Esta solución general da lugar a un nuevo patrón de diseño que llamamos PDN.

Se aborda el concepto de transacción y los mecanismos necesarios que se incorporan en el modelo para su gestión.

Capítulo 6: Funcionamiento de los Agentes

Una vez definido el modelo, nos centramos en el funcionamiento de los agentes. Puesto que este funcionamiento es dirigido por el agente llamado *Controller,* se especifica detalladamente su comportamiento. Esto nos lleva a estudiar el comportamiento de sus sub-agentes: *Ocurrence Receiver* y *Evaluator*.

A continuación nos centramos en el agente *Transaction Manager* que se encarga de controlar la realización de una transacción.

Una vez establecido el modelo básico, se estudian diversas optimizaciones que harían que el funcionamiento del sistema software fuera más eficiente.

Finalizamos presentando un lenguaje de descripción de sistemas basados en agentes.

Capítulo 7: Evolución de los Agentes

Primero se introduce el proceso de evolución dentro de nuestro modelo. Para ello, se estudia con detalle el funcionamiento del Metasistema y de los Sistemas Genéticos.

Se describe detalladamente el conjunto de acciones estructurales o de evolución y la lista de los invariantes que deben cumplirse en un sistema software.

Se exponen los aspectos a tener en cuenta para mantener consistente la información de un sistema cuya estructura es dinámica.

Capítulo 8: Caso práctico: Empresa de alquileres de coches

Se presenta un sistema real con el fin de construir un sistema software que lo modele. Definimos el problema y la jerarquía de agentes que constituye el sistema software resultante.

Se describen para cada uno de los agentes, las acciones que realiza, las pre-condiciones asociadas a éstas y la información que mantiene su *Controller*.

Terminamos presentando varios ejemplos de funcionamiento y de evolución que ayudan a comprender el modelo presentado.

Capítulo 9: Prototipo

En este capítulo nos centramos en el prototipo que hemos desarrollado para crear sistemas software basados en agentes y permitir su evolución.

Mostramos el diseño de clases que se ha usado para la construcción de la herramienta. En este diseño están presentes todos los conceptos que forman parte de la arquitectura de un sistema software basado en agentes.

Capítulo 10: Conclusiones y Trabajos Futuros

Se presentan las principales conclusiones de este trabajo de investigación, enfatizando las aportaciones obtenidas.

Finalizamos con la exposición de distintas líneas de trabajo futuras que mejorarían y complementarían los resultados obtenidos en esta tesis.

Apéndices

En el apéndice I se muestra el lenguaje L desarrollado por M.J. Rodríguez [Rodríguez00a].

En el apéndice II se presenta el proceso que seguido para realizar las consultas sobre la historia.

En el apéndice III recogemos el estudio realizado de los distintos mecanismos de comunicación y sincronización existentes.

En el apéndice IV se presenta un glosario de términos.

Bibliografía

Por último, se muestra la bibliografía utilizada y referenciada en esta tesis. Como se puede observar, en ella aparecen referencias bibliográficas que carecen de fecha, esto es debido a que no se corresponden con publicaciones en artículos, libros o revistas sino que son referencias a páginas web.

CAPITULO 2

Evolución del Software

La evolución es un problema crucial en el desarrollo de software, que lleva estudiándose desde hace casi dos décadas y para el cual se han propuesto diversas soluciones. Después de realizar una revisión de la literatura existente sobre el tema, hemos podido observar que existen visiones muy diferentes, tanto en la propia definición del concepto como en los enfoques propuestos para abordar los aspectos evolutivos.

El término *evolución* ha sido utilizado por muy escasos autores. A partir de 1990 este término se ha ido haciendo más común cada día, debido principalmente a la concepción del proceso de desarrollo del software como un proceso iterativo que tiene lugar durante toda la vida del software [Boehm86]. Desde esta perspectiva, el desarrollo del software comienza en el instante en que se establece una lista inicial de requisitos que debe cubrir el sistema software, y finaliza cuando el sistema deja de usarse [Fisher91]. Durante la concepción del sistema, y, posteriormente durante su uso, la funcionalidad y la estructura del sistema pueden cambiar. De hecho, en los sistemas que están en funcionamiento, surge la necesidad de introducir cambios (adaptativos, correctivos o de mejora) con el fin de que pasen a comportarse de forma diferente [Sommerville96]. Al proceso de cambio continuo que se realiza iterativamente durante toda la vida del software, le llamamos evolución.

2.1 Evolución frente a mantenimiento.

Como opinan algunos autores [Rodríguez01b], *históricamente, el mantenimiento se puede considerar como la primera forma de evolución de los sistemas*. El mantenimiento se ha considerado tradicionalmente como la última etapa del ciclo de vida del software y siempre ha sido una de las más costosas. Debido a esto, se está prestando especial interés en investigar cómo se puede optimizar dicha etapa y qué herramientas y métodos son necesarios para minimizar el coste del mantenimiento de un software que sufre continuas modificaciones a lo largo de su vida.

Algunas definiciones de mantenimiento son:

- ANSI/IEE: *Las modificaciones de los productos software después de su entrega para corregir fallos, mejorar el rendimiento u otros atributos o adaptar el producto a un cambio de entorno.*

- ISO/IEC: *Un producto software soporta una modificación en el código y su documentación asociada para la solución de un problema o por la necesidad de una mejora. Su objetivo es mejorar el software existente manteniendo su integridad.*

- [Pressman02] *La fase de mantenimiento se centra en el cambio que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software y a cambios debidos a las mejoras producidas por los requisitos cambiantes de los clientes.*

En todas ellas podemos comprobar que el mantenimiento comienza cuando ya se ha iniciado el funcionamiento del sistema software y es una etapa aislada donde, probablemente, es necesario parar el sistema para poder realizar sobre él las operaciones de mantenimiento necesarias. Otros autores, como Sommerville [Sommerville01], ya no hablan de mantenimiento, sino de evolución del software. Estos autores opinan que las etapas de desarrollo de software y de mantenimiento no son dos procesos separados. Es más realista considerar a la ingeniería de software como un proceso evolutivo en el cual el software se cambia continuamente durante su periodo de vida como respuesta a los requisitos cambiantes y a las necesidades del usuario.

Nosotros estamos de acuerdo con este punto de vista aunque está claro que, ante el problema de los *legacy systems*, el mantenimiento es la única solución. Es un software ya construido y, a no ser que volvamos a construirlo, no podremos aplicarles los métodos necesarios para permitir que pueda evolucionar de forma controlada. Sólo en la construcción de nuevo software podremos aplicar los modelos, métodos

y herramientas necesarios para preparar dicho software con el fin de que evolucione sin problemas y a un coste más bajo que el que tendría con el mantenimiento tradicional.

Por tanto la evolución no sólo comprende el mantenimiento de un software que ya ha comenzado a funcionar sino que se funde con el resto del proceso de desarrollo viendo la construcción y funcionamiento del software como un proceso continuo.

Daremos finalmente la definición de evolución que hemos adoptado [Parets98] y de la que partimos en este trabajo:

> *La evolución de un Sistema Software consiste en la transformación de la estructura (patrones de acción, decisión y memorización) a lo largo del tiempo, realizada por el Sistema de Desarrollo. La evolución del Sistema Software es la funcionalidad fundamental del Sistema de Desarrollo.*

2.2 Diferentes tendencias de investigación en evolución del software.

Son varios los autores que han trabajado en evolución del software, y también son variados los puntos de vista desde los que se aborda el propio concepto de evolución y sus implicaciones. En [Parets96] [Parets99b] [Rodríguez01a] se agrupan los trabajos de estos autores según la concepción que presentan de la evolución. A continuación presentamos las principales tendencias detectadas considerando los objetivos planteados por sus autores.

2.2.1 Modelado de la dinámica del software.

El trabajo pionero de Belady y Lehman [Belady76] considera la evolución de software como "dinámica de la evolución de programas", identificando esta evolución con los continuos cambios sufridos por los programas, cambios que eran contemplados como fuente de problemas más que como una necesidad de adaptación de los programas al entorno. Posteriormente, Lehman [Lehman80] trata de establecer leyes cuantitativas sobre el comportamiento de un sistema software y su proceso de desarrollo. Estos trabajos se desarrollaron en la línea clásica de la dinámica de sistemas tratando de descubrir los parámetros de esta evolución concebida como dinámica parametrizable.

Actualmente, Lehman y Ramil, en sus trabajos más recientes, [Lehman00][Lehman01a][Lehman01b][Lehman02] se preocupan más por el "qué" y el "por qué" de la evolución que por el "cómo" (métodos y herramientas que permiten crear sistemas software que puedan evolucionar y adaptarse a su entorno). Les interesa estudiar la naturaleza del fenómeno de la evolución, a quién se dirige y el impacto que esta produce. Por esa razón, durante más de 30 años (1969-2001) han recogido datos sobre la evolución de distintos sistemas de diferentes tamaños, para distintos dominios de aplicación, desarrollados por organizaciones diferentes y con variados grupos de usuarios. Una vez obtenidos los datos necesarios, los autores proponen una teoría de evolución del software basada en dos niveles: el nivel teórico (*Theorical Level*) y el nivel de observación (*Observational Level*). El nivel de observación incluye las observaciones cualitativas y cuantitativas realizadas durante muchos años sobre el comportamiento de distintos sistemas. De algunas de ellas se pueden obtener los invariantes de comportamiento de los sistemas. A partir de este estudio se sacan unas generalizaciones o leyes. Una vez que las tengamos, comienza la formalización de la teoría y como resultado, la determinación de un conjunto de reglas y guías. Todo este proceso es cíclico, es un sistema realimentado (*feedback*). En la figura siguiente se representa el desarrollo de la teoría.

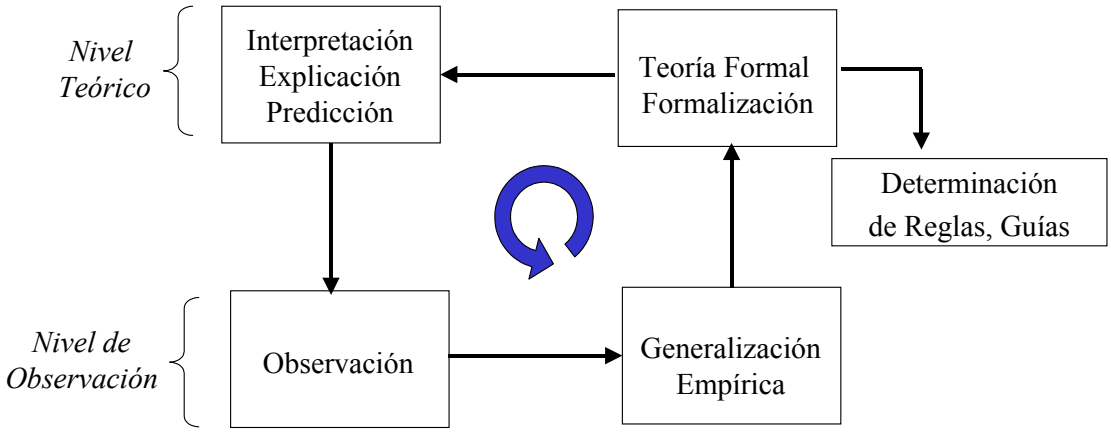


Figura 2.1 Desarrollo de la teoría de Lehman y Ramil

2.2.2 Modelado del proceso de desarrollo de software.

En la década de los 80, distintos autores investigan sobre los modelos de producción de software (modelos de ciclo de vida). Estos modelos de ciclo de vida tratan de incorporar la iteración en el proceso de desarrollo del software. El sistema software se desarrolla haciendo uso de iteraciones, cada una de las cuales lleva asociada tareas de

especificación, diseño, codificación o prueba. Dentro de este grupo de modelos cabe destacar el modelo de prototipos [Agresti86], el modelo en espiral de Boehm [Boehm86] y el modelo de Henderson [Henderson90].

La elaboración de este tipo de modelos, tratando de abandonar la pretendida perfección de un proceso secuencial, e incorporando la posibilidad de retroalimentación, ha servido de base para que se plantee la necesidad de la representación, a ser posible automatizable, del proceso de desarrollo, controlando el flujo de trabajo (*workflow*) y gestionando el control de versiones. Con esta finalidad, a finales de los 90 se lanza el *Rational Unified Process*, RUP [Jacobson00] [RUP] que presenta un proceso de desarrollo de software integrado, centrado en la arquitectura, dirigido por los casos de uso (desde la especificación al mantenimiento) y donde se sigue un ciclo de vida iterativo e incremental. El ciclo de desarrollo de software se divide en cuatro fases: preparación inicial (*Inception*), preparación detallada (*Elaboration*), construcción (*Construction*) y transición (*Transition*). Este proceso de desarrollo, al estar inmerso dentro de las metodologías orientadas a objetos, es, hoy en día, uno de los más extendidos.

2.2.3 Transformación de programas.

Los programas pueden transformarse para generar nuevas versiones que consideren automáticamente las modificaciones en las especificaciones de los requisitos [Kozaczynski92] o que preserven el significado original y se mejoren aspectos como la eficiencia del código[Berzins93]. Berzins y sus colaboradores opinan que "*La mayor parte de la investigación en transformación asume que el programa transformado P' debe cumplir los requisitos del programa original P. Estas transformaciones se denominan transformaciones preservadoras del significado".*

Muchos autores, como Said [Said01] llevan a cabo transacciones automáticas basadas en factores de calidad.

2.2.4 Modelado de la dinámica en sistemas de información: modelado dinámico.

Tardieu [Tardieu92] define los principios básicos del modelado dinámico:

> *"El modelado dinámico consiste en capturar los invariantes del sistema de información que no se expresan en el modelado estático [...] por oposición a la cinemática, no se concentra en las*

transiciones de estados, pero considera las causas de la transición [...] por oposición a la evolución, no cambia la estructura del sistema de información. Cualquier cambio es reversible. El beneficio del modelado dinámico reside en el hecho de que describe los productos y los procesos, lo que conduce a una mejor comprensión de la organización modelada.".

Ejemplos típicos de esta tendencia son los diagramas de transición de estados utilizados en OMT [Rumbaugh91] y en UML [Booch99]. En definitiva, en este caso se trata de modelar la sucesión de los diferentes estados por los que pasa un sistema y las acciones que producen los cambios entre los diferentes estados.

2.2.5 Modelado de la dinámica del proceso de concepción-desarrollo de software.

Desde esta perspectiva se concibe un sistema software como dotado de una estructura determinada que puede evolucionar y madurar. La evolución implica la transición desde un estado del sistema a otro gracias a la labor del equipo de desarrollo. El estudio de la evolución implica, pues, la elaboración de modelos de estas transiciones estructurales.

En este campo cabe destacar los trabajos realizados por autores relacionados con el paradigma orientado a objetos. Banerjee [Banerjee87] y Casais [Casais90] consideran la evolución de una estructura de clases, las operaciones que modifican esa estructura, las restricciones a la hora de realizar esas modificaciones y los efectos de dichas modificaciones en la estructura de clases y en las instancias existentes de esas clases. En el trabajo desarrollado por [Lieberherr93] [Lieberherr94] dentro del proyecto DEMETER, se introduce la historia de evolución de una estructura. Esta historia describe una secuencia de fases de desarrollo. También introducen la noción de *programa adaptativo* que es capaz de amoldarse a ciertos cambios en la estructura de clases. Con el mismo sentido, autores como Heckel [Heckel01], Wermenlinger [Wermenlinger01] y Mens [Mens01] proponen como herramienta formal, usada a nivel de meta-modelo, la utilización de la reescritura de grafos (graph rewriting) para modificar la estructura de los programas, resolviendo los problemas de evolución.

Nuestro trabajo puede encuadrarse dentro de esta última tendencia con un enfoque similar al utilizado por [Banerjee87] [Casais90] [Lieberherr96] [Carsí99]. Al igual que estos autores, nosotros consideramos la evolución como un proceso de maduración. El proceso de maduración comienza cuando empezamos a crear un sistema y está presente durante toda la vida del sistema. Concebimos un *sistema* como

un conjunto de elementos en interacción y un *modelo* como una representación simbólica de éste utilizando algún medio. Según esta visión, un sistema software pasa, a lo largo de su desarrollo, por diferentes estadíos de maduración. El cambio de un estadío a otro está motivado por la intervención del equipo de desarrollo para modificar el(los) modelo(s) del sistema.

2.3 El origen del cambio en los Sistemas Software.

En la fase inicial del ciclo de vida se realiza una primera actividad consistente en la obtención de los requisitos del sistema a partir de las necesidades de los usuarios. Esta actividad, conocida habitualmente como *elicitación de requisitos*, es un proceso iterativo, durante el cual se establecen requisitos que se validan para descubrir errores o detectar nuevos requisitos ocultos hasta el momento. Es un medio para que los desarrolladores y usuarios lleguen a una comprensión del sistema. Las técnicas utilizadas en este proceso son muy variadas y suelen implicar, de una u otra forma, el uso del lenguaje natural. Técnicas como los casos de uso, integrados en la metodología UML [Booch99] o modelos como *WinWin* [Boehm98], permiten obtener los requisitos a partir de la interacción de los usuarios y el equipo de desarrollo.

Una vez establecidos los requisitos, el equipo de desarrollo describe el sistema creando un modelo (o modelos) con un mayor grado de formalismo que el lenguaje natural. Con mucha frecuencia, estas necesidades o requisitos no están claros, son ambiguos, entran en conflicto o son demasiado generales, provocando, incluso, cambios durante el proceso de modelado. Esta dificultad viene ocasionada por alguna de las siguientes situaciones:

- Problemas de comunicación y comprensión entre usuarios y desarrolladores: dificultad para expresar los requisitos mediante alguna metodología que ambos comprendan y sobre la que se puedan hacer validaciones [Duran99].
- Las necesidades y las prioridades de los usuarios pueden cambiar. Tanto éstos como los desarrolladores pueden tomar decisiones que llevarán a la realización de modificaciones sobre los requisitos y a la propagación de los cambios en cascada [Rumbaugh91].
- El entorno del sistema software puede variar e inducir modificaciones en el propio sistema [Meyer97].
- El conocimiento sobre el sistema va ampliándose, con lo que se irá detallando la especificación y concretando la estructura.

La experiencia nos ha enseñado que es imposible evitar estas situaciones y que siempre se realizarán cambios durante el proceso de desarrollo del software. En ese caso, la mejor alternativa es facilitar al desarrollador la labor de especificación y gestión de requisitos cambiantes mediante el uso de herramientas CASE que proporcionen mecanismos para ello. Bajo esta perspectiva resulta interesante la utilización de modelos de proceso iterativos, como el ciclo de vida del prototipado [Sommerville96], o el modelo en espiral [Boehm86] que permiten integrar el cambio, de forma más natural, en el proceso de desarrollo. El uso simultaneo de herramientas y de un proceso adecuado permite:

- Ayudar a los usuarios a concretar sus necesidades reales y sus requisitos.

- Ayudar a los desarrolladores a concretar, de forma gradual, las necesidades y requisitos del usuario, incorporando sus propias ideas, tomando decisiones sobre la permanencia de algunos requisitos y sobre sus características y las de otros requisitos con los que estén relacionados.

- Ayudar a ambos a entenderse mutuamente.

La evolución del modelo permitirá hacer cambios sobre él, para que tanto usuarios como desarrolladores puedan evaluarlo (comprobarlo y validarlo), proporcionando un *feedback* (retroalimentación) adecuado.

Un sistema nunca dejará de evolucionar hasta que se deseche. Esto significa que el proceso de modelado no termina cuando se entrega un producto, sino que continúa mientras que se esté usando. Por tanto es necesario incorporar mecanismos que permitan al desarrollador cambiar un sistema durante toda su vida. Será necesario proporcionar modelos conceptuales y herramientas que garanticen ciertas propiedades durante la vida de los sistemas software.

2.4 Mecanismos y modelos de evolución.

En [Torres02] se presenta un amplio estudio de los mecanismos y modelos de evolución que se usan en la actualidad y que se han utilizado anteriormente en Biología y su aplicación, teniendo en cuenta el tipo de evolución factible para los Sistemas Software.

La evolución está relacionada de forma natural con los sistemas software. Las necesidades de evolución para un sistema software se presentan:

- durante el proceso de desarrollo, desde el instante que el desarrollador se plantea la obtención de un modelo que represente al sistema real.
- cuando el sistema software ya está funcionando y se produzcan cambios en su entorno, en los requisitos con los que inicialmente fue concebido o en ambos.

Tanto los mecanismos como los modelos de evolución están relacionados con la actividad del sistema de desarrollo y del sistema software en funcionamiento. La diferencia reside en que los mecanismos son actividades de evolución abstractas que, pueden implicar o no cambios en la estructura del sistema. Los modelos son una representación simbólica de cómo evoluciona un sistema software mediante la utilización de uno de los mecanismos comentados.

Para hacer operacionales los mecanismos y los modelos de evolución se debe llevar a cabo una formalización que permita el uso de herramientas de representación e implementación para obtener sistemas software concretos.

Se proponen dos tipos de mecanismos de evolución de software:

1. Mecanismo de *Adaptación*: para definir la forma básica en la que el sistema modifica su estructura o funcionamiento de acuerdo a las necesidades del entorno. Puede ser de dos tipos:

    - *Acomodación-Aprendizaje*: Este tipo respondería a los cambios que sufre un sistema para adaptarse mediante aprendizaje de la forma en la que podría utilizar más adecuadamente su estructura.
    - *Mutación-Diferenciación*: Se plantea para afrontar la necesidad de llevar a cabo cambios estructurales en el sistema mediante la modificación o diferenciación de sus componentes, la inserción de nuevos elementos o la eliminación de los existentes.

2. Mecanismo de *Herencia de Caracteres Adquiridos*: forma en la que se genera un nuevo sistema software, a partir de otros ya existentes, transmitiendo a los nuevos sistemas las características adquiridas durante la "vida" del sistema original.

Los modelos consisten en representaciones simbólicas de las posibles formas en las que pueden modificarse los sistemas software y los Metasistemas (sistemas utilizados por el modelador o desarrollador para modificar los sistemas software). Se identifican los siguientes modelos:

1. *Metateleología Dirigida por el Modelador*: modela la modificación del Metasistema a través del modelador. Utiliza el mecanismo de adaptación mediante mutación-diferenciación.

2. *Teleología Dirigida por el Modelador*: trata la evolución del sistema software por influencia del modelador a través del Metasistema. Utiliza el mecanismo de adaptación mediante mutación-diferenciación.

3. *Herencia de Metacaracteres Adquiridos*: genera un nuevo Metasistema mediante clonación. De esta forma, el nuevo Metasistema hereda las adaptaciones adquiridas por el Metasistema del cual clona. Utiliza el mecanismo de herencia.

4. *Herencia de Caracteres Adquiridos*: modela la producción de un nuevo sistema que hereda las adaptaciones adquiridas por el antiguo (a través de una operación de clonación). Utiliza el mecanismo de herencia.

5. *Autoadaptación Metasistema-Sistema Software*: el Metasistema interactúa con el sistema software para completar el desarrollo y evolución de éste último, de forma independiente del modelador. Utiliza los mecanismos de adaptación mediante mutación-diferenciación y/o acomodación-aprendizaje.

6. *Autoadaptación del Sistema Software*: el sistema software puede desencadenar, autónomamente, algunas acciones adaptativas de forma independiente del modelador. Utiliza el mecanismo de adaptación mediante acomodación-aprendizaje.

Los modelos deben formalizarse. Esta formalización se consigue [Torres00] [Torres02] aplicando un conjunto de operadores para realizar las modificaciones, tanto funcionales como estructurales, en los sistemas. Estos operadores se dividen en dos clases:

- Aquellos que modifican la estructura. Se utilizarán por los modelos de herencia de caracteres adquiridos y por los que utilizan mecanismos de adaptación mediante mutación-diferenciación.
- Aquellos que no modifican la estructura. Se aplicarán por los modelos que utilicen el mecanismo de adaptación mediante acomodación-aprendizaje.

Además, dichos operadores se aplicarán teniendo en cuenta la definición de distintas condiciones que deben cumplirse para poder aplicarlos.

En nuestro trabajo, centrado en la evolución de los sistemas software basados en agentes, utilizamos los siguientes dos tipos de modelos:

- El modelo de la Herencia de Caracteres Adquiridos: la forma normal de creación de los agentes es mediante clonación.
- El modelo de Teleología Dirigida por el Modelador: implementamos el Metasistema como una herramienta CASE. El modelador interactúa con el Metasistema para hacer evolucionar al sistema software. El mecanismo de adaptación utilizado es el de mutación-diferenciación.

2.5 Conclusiones.

Con el fin de no confundir el concepto tradicional de mantenimiento y el de evolución del software, hemos aclarado ambos términos. Para nosotros, la evolución de un sistema software comienza al principio del proceso de desarrollo y termina cuando se desecha el sistema. Se deben proporcionar los modelos, métodos y mecanismos necesarios para que un sistema pueda modificarse adecuadamente en respuesta a cambios tanto adaptativos como correctivos o perfectivos (de mejora).

Hemos distinguido distintas tendencias de investigación referentes a la evolución del software. De todas ellas, nuestra concepción de la evolución del software se aproxima más al *modelado de la dinámica del proceso de concepción-desarrollo de software*. Nos basamos en la idea de que el equipo de desarrollo produce modelos del sistema software que, a lo largo del tiempo, irán refinándose. Cada uno de estos refinamientos implica un estadío en la estructura del sistema. La evolución se encontrará, pues, con dos problemas fundamentales que abordaremos en el resto de los temas:

- La manipulación y gestión de estos cambios de un estadío a otro manteniendo la integridad y la consistencia de los modelos propuestos.
- La propagación del cambio a las estructuras ya fijadas y en funcionamiento (en general, los datos manejados por el usuario).

Una vez estudiadas las razones por las que los sistemas software pueden cambiar, hemos abordado los distintos mecanismos y modelos de evolución posibles en los sistemas software. De ellos, los que están relacionados con nuestro trabajo son los modelos *de Herencia de Caracteres Adquiridos* y *Teleología Dirigida por el Modelador*.

CAPITULO 3

Estudio de Arquitecturas del Software y Patrones

Todo sistema software tiene una estructura que determina su funcionamiento. Esta estructura se ajusta a una arquitectura determinada y, por ello, hemos realizado un estudio de las distintas arquitecturas o estilos arquitectónicos que usualmente se utilizan en la construcción de sistemas software. Este estudio nos permite valorar cuál de ellas, si existe alguna, se adapta mejor a los sistemas software basados en agentes.

Como veremos, la arquitectura determina a un nivel alto de abstracción el conjunto de componentes y el tipo de interconexiones existentes entre éstos. Sin embargo, existen otras cuestiones que necesitan un nivel de detalle más concreto, como por ejemplo, el modelo de coordinación existente entre los componentes del sistema. Con el fin de determinar si el modelo de coordinación que necesitamos está resuelto mediante algún patrón de diseño, hemos estudiado distintos patrones de diseño y valorado si se adaptan o no a nuestras necesidades.

3.1 Arquitectura del software.

La Arquitectura del Software (AS) [Shaw96][Sartipi97][Addy98] es una disciplina "relativamente reciente" para los ingenieros del software. Surgió, de forma natural, junto con la evolución de las abstracciones de diseño (patrones). Este nacimiento tiene su origen en la búsqueda de los ingenieros para obtener un software más comprensible con el fin de facilitar la construcción de sistemas software, cada vez más grandes y complejos. Esta facilidad en la construcción de nuevo software se basa en la reutilización, no sólo del código (o partes de él), sino también del diseño.

Dentro de las metodologías de desarrollo de software, podemos hablar del *diseño arquitectónico*, cuya idea básica es ver al sistema como un todo. En el diseño arquitectónico se decide, a un nivel de abstracción elevado, cual será la arquitectura del sistema basándose en sus características. En esta decisión influye mucho el dominio de aplicación en el que se enmarque el sistema.

En cualquier ingeniería se separa el conjunto de tareas involucradas en el diseño en dos grupos: diseño "*rutinario*" y diseño "*innovador*" [Shaw96]. El diseño rutinario está relacionado con la obtención de soluciones para problemas familiares, cotidianos. En éste, es normal la reutilización de soluciones antiguas y efectivas (o grandes partes de ellas). El diseño innovador se relaciona con la búsqueda de nuevas soluciones para problemas no familiares. Los diseños innovadores suelen ser menos necesarios que los rutinarios. La arquitectura del software, está, quizás, más orientada al diseño rutinario. Sin embargo, se puede observar que el software, en muchos dominios de aplicación, se trata más como innovador que como rutinario, más de lo que sería necesario. Si capturáramos y organizáramos lo que ya sabemos se podría incrementar la productividad en la creación de sistemas software, identificando las aplicaciones que pueden ser rutinarias y desarrollando el soporte adecuado. Actualmente, la reutilización está enfocada a la captura y organización del conocimiento existente de una parte particular del problema: el conocimiento expresado en forma de código. Pero este conocimiento no es útil si los programadores no lo conocen o no se animan a usarlo (los componentes de una librería requieren más cuidado en el diseño, implementación y documentación que otros componentes simplemente insertados en un sistema).

Debido al incremento del tamaño y complejidad de los sistemas software (SS), el diseño y la especificación de la estructura del sistema llegan a ser más importantes que las clases de algoritmos y estructuras de datos que se utilizarán internamente.

En un sistema simple, el diseño consiste en:

- Elegir los algoritmos apropiados.
- Elegir las estructuras de datos necesarias.

En un sistema complejo, el diseño, respecto a la estructura, consiste en:

- Organizar el sistema como un conjunto de componentes.
- Elegir las estructuras de control globales.
- Decidir los protocolos de comunicación, sincronización y acceso a los datos.
- Asignar la funcionalidad para diseñar elementos.
- Realizar la composición de los elementos de diseño.
- Realizar la distribución física, determinar el escalado y el rendimiento del sistema.
- Tener en cuenta los aspectos de mantenimiento y evolución.
- Seleccionar entre distintas alternativas de diseño.

Dada la complejidad del proceso, el tiempo y coste del desarrollo de un sistema software complejo es muy grande. Se debe intentar minimizar en lo posible las distintas tareas que se realizan en el desarrollo de estos tipos de sistemas.

Como conclusión diremos que, actualmente, los profesionales reconocen la necesidad de establecer formas de compartir la experiencia a un nivel mayor de abstracción, a nivel de diseño de arquitecturas que sean comunes en distintos dominios de aplicación. Una vez identificada la arquitectura adecuada, se puede ganar tiempo reutilizando las decisiones de diseño que la acompañan.

3.1.1 ¿Qué se entiende por arquitectura del software?

De forma abstracta, la arquitectura del software supone la descripción de los elementos de los sistemas construidos, las interacciones entre dichos elementos, los patrones que guían su composición y las restricciones sobre dichos patrones [Shaw96], es decir, "*la arquitectura de un sistema software define a dicho sistema en términos de componentes computacionales y las interacciones entre ellos*". Los componentes pueden ser clientes y servidores, bases de datos (BD), filtros y capas de un sistema jerárquico. Las interacciones pueden ser tan simples y conocidas como una llamada a un procedimiento o un acceso a una variable compartida. Pero también pueden ser tan complejas y semánticamente ricas como los protocolos cliente-servidor o los protocolos de acceso a una BD. Casi todos los autores que trabajan

en este campo tienen claro la división de los elementos arquitectónicos de un sistema en componentes y conectores o interacciones. La mayoría de la investigación actual está orientada a una o a las dos clases de elementos.

Existe una amplia variedad de definiciones asociadas al término *Arquitectura del Software*. En un documento del SEI (*Software Engineering Institute*) de *Carnegie Mellon University* [CMU] se muestran numerosas definiciones de arquitectura del software, tanto de personalidades relevantes en el campo de la ingeniería del software como de investigadores noveles. De la lectura de todas ellas, concluimos que existe un núcleo común que también está presente en la anterior definición y es el hecho de que una arquitectura se constituye de un conjunto de componentes y conectores y éstos serán más o menos complejos dependiendo del tipo de sistema que se quiera modelar.

Además de especificar la estructura y topología del sistema, la arquitectura muestra la correspondencia entre los requisitos del sistema y los elementos construidos y proporciona las razones que justifican las decisiones de diseño. Un sistema no es sólo los componentes y sus interacciones, tiene unas propiedades asociadas que debe cumplir.

En general, los modelos arquitectónicos tratan de aclarar las diferencias estructurales y semánticas entre los componentes y las interacciones. La arquitectura establece las especificaciones para los elementos individuales, que pueden ser en sí mismos redefinidos como subsistemas arquitectónicos o implementados en un lenguaje de programación convencional.

El decir que esta disciplina es “relativamente nueva” no es totalmente cierto. Si miramos hacia atrás, es algo que se ha ido haciendo desde que se hace software. La relativa informalidad y el alto nivel de abstracción que actualmente se hace en la descripción de las arquitecturas podría, inicialmente, sugerir que las descripciones arquitectónicas no son importantes para los ingenieros del software. Pero hay dos razones por lo que esto no es cierto:

1) Los ingenieros han desarrollado a través del tiempo una colección de lenguajes, patrones y estilos de organizaciones de sistema software que les sirven de vocabulario común.

2) Aunque las estructuras arquitectónicas son abstractas, proporcionan un entorno natural para obtener una mayor comprensión en temas relacionados con el sistema, tales como patrones de comunicación, estructuras de control de la ejecución, escalabilidad y evolución del sistema.

La descripción de la arquitectura de un sistema sirve para obtener las propiedades de dicho sistema y, así, encontrar sus requisitos más fuertes. Esto no quiere decir que sean innecesarias las notaciones formales y las técnicas de análisis rigurosas para la descripción de arquitecturas. En efecto, se podría ganar mucho si el diseño arquitectónico (y con ello la arquitectura) estuviera soportado con mejores notaciones, teoría y técnicas analíticas. De hecho, la investigación en el campo de la arquitectura del software se deriva hacia la utilización de notaciones formales que permitan comprobar si dicha arquitectura cumple las propiedades del sistema. Este es uno de nuestros principales objetivos, presentar un modelo que nos permita construir sistemas software con una determinada arquitectura de tal forma que se pueda comprobar fácilmente que el sistema cumple un determinado conjunto de propiedades.

Una arquitectura del software debe contener la información necesaria para su comprensión, evaluación y toma de decisiones. Las cuestiones [Addy98] que deben considerarse en cualquier arquitectura del software incluyen:

- ¿Cuál es la naturaleza de los componentes?
  - ¿Cuál es el significado de la separación de componentes?
  - ¿Se ejecutan los componentes de forma independiente?
  - ¿Se ejecutan en tiempos distintos?
  - ¿Los componentes constan de procesos, hebras o ambos?

- ¿Cuál es el significado de los enlaces?
  - ¿Se comunican los componentes entre ellos?
  - ¿Se controlan entre ellos?
  - ¿Se envían datos?
  - ¿Se utilizan unos a otros, se invocan explícitamente?
  - ¿Se sincronizan unos con otros?
  - ¿El enlace indica un trabajo de descomposición de estructura?

- ¿Cuál es el significado de tener niveles?
  - ¿Por qué hay componentes en diferentes niveles?
  - ¿Cuál es la relación entre componentes de diferentes niveles?

- ¿Cómo trabaja la arquitectura en tiempo de ejecución?

- ¿Cómo fluyen los datos y el control a través del sistema?

3.1.2 El estado de la Arquitectura del Software.

Una buena base en la arquitectura del software beneficia tanto al desarrollo como a su evolución. Respecto al desarrollo, cada vez está más claro que se necesita facilitar el diseño arquitectónico del software. Las razones son:

- Es importante poder reconocer los paradigmas comunes. Si se comprenden las relaciones a alto nivel entre los sistemas, se pueden construir nuevos sistemas como variaciones de sistemas antiguos.
- Es crucial conseguir la arquitectura correcta para obtener un buen diseño de un sistema software, una equivocación puede llevarnos a resultados desastrosos.
- El comprender detalladamente las distintas arquitecturas del software existentes permite a los ingenieros elegir entre distintas alternativas de diseño.
- La representación de un sistema arquitectónico normalmente es esencial para el análisis y la descripción de las propiedades de alto nivel de un sistema complejo.
- El manejo en el uso de notaciones para describir paradigmas arquitectónicos permite al ingeniero comunicar nuevos diseños de sistemas a los demás.

Inicialmente, en el campo de la arquitectura del software se utilizaban diagramas y párrafos informales como herramientas de descripción. Desgraciadamente, esto es ambiguo. En estos diagramas y párrafos están implícitas las intuiciones y la experiencia de su creador, de tal forma que otra persona distinta, al carecer de éstos, no les da el mismo significado. Además, los diseñadores generalmente carecen de conceptos, herramientas y criterios de decisión adecuados para seleccionar y describir estructuras de sistemas que sean apropiadas para el problema.

La elección de la arquitectura más apropiada para un problema determinado (o dominio) sigue siendo un problema abierto. Las reglas del estilo arquitectónico determinan, normalmente, cómo empaquetar componentes, por ejemplo, procedimientos, objetos o filtros. Los componentes normalmente no pueden intercambiarse entre estilos: el código no puede reutilizarse porque su interfaz realiza suposiciones incompatibles.

Actualmente no se tiene una taxonomía bien aceptada de los distintos estilos arquitectónicos existentes. Tampoco existe una teoría completamente desarrollada de la arquitectura del software. Pero se puede identificar un conjunto de arquitecturas, o estilos

arquitectónicos, que actualmente forman la base de una arquitectura software [Shaw96]. Algunas de éstas están cuidadosamente documentadas y ampliamente difundidas. Por ejemplo, el sistema *X Window* sigue una arquitectura cliente-servidor.

Una comunidad cada vez mayor de investigadores trabaja en temas relacionados directamente con la arquitectura del software. Han surgido nuevos lenguajes de descripción de arquitecturas (ADL-*Architecture Description Language*) [NIST], formalismos, patrones, etc. Los ADLs proporcionan modelos, notaciones y herramientas que permiten describir los componentes y conectores presentes en la estructura de una determinada aplicación. Por su importancia, retomaremos el tema de los ADLs en el punto 3.3 cuando hablemos de las arquitecturas dinámicas. Las cuestiones arquitectónicas se están estudiando en áreas tales como lenguajes de interfaces de módulo, arquitecturas específicas del dominio, reutilización del software, codificación de patrones de organización para el software, lenguajes de descripción arquitecturales y entornos de diseño arquitectónico.

Así mismo, muchos campos de investigación en ingeniería del software están relacionados con la arquitectura del software. Uno de ellos es la Programación Orientada a Aspectos (AOP-*Aspect Oriented Programming*) [AOSD], que se basa en separar los distintos aspectos que pueden observarse en un elemento o componente de un sistema: coordinación, distribución, funcionalidad, seguridad, etc. Los aspectos se implementan de forma independiente y la agrupación de distintos aspectos da lugar a un componente del sistema que tiene ciertas propiedades y características.

La Ingeniería del Software Basada en Componentes, CBSE (*Component-Based Software Engineering*) [ISBC], también está muy relacionada. A veces también nos referimos a la Programación Orientada a Componentes (*Component-oriented programming*) [COP]. La programación orientada a componentes [Nierstrasz95] es una extensión de la programación orientada a objetos (OOP) para entornos abiertos. Se intenta tener una bolsa de componentes global reutilizables en el desarrollo de un sistema software. Para ello se tiene especial interés en diseñar componentes con una interfaz clara que indique las operaciones que es capaz de realizar, así como las que necesita de otros componentes cuando esté en ejecución (sus restricciones). Los lenguajes basados en componentes describen los sistemas como configuraciones de módulos que interactúan de una forma predeterminada (llamada a procedimiento remoto, mensajes o eventos) cumpliendo unos patrones de organización específicos. Estos lenguajes proporcionan nuevas formas de describir interacciones entre componentes en grandes sistemas, pero utilizan, normalmente, conjuntos pequeños de paradigmas de comunicación y descripciones a

nivel de programación o están orientados a una organización de propósito único, especializado. Esto los hace inapropiados para expresar un amplio rango de diseños arquitectónicos. La diferencia entre los lenguajes de descripción de arquitecturas y los lenguajes basados en componentes, es el distinto nivel de abstracción en el que trabajan. Los lenguajes basados en componentes trabajan a un nivel más bajo de abstracción sobre los componentes de un sistema, se centran en el diseño y la implementación de los componentes (su funcionalidad, dependencias internas y externas, interfaz, etc.).

Hay que tener en cuenta que, aunque cada sistema tiene una arquitectura, no siempre es conocida y la arquitectura utilizada puede divergir de su diseño. La arquitectura para un sistema puede ser buena o mala, según se adecue la arquitectura al sistema, a su propósito. Por tanto, el análisis y la evaluación de la arquitectura es también una cuestión importante.

Antes de comenzar describiendo las distintas arquitecturas, resumimos los objetivos más destacables por los cuales nace el campo de investigación de la arquitectura del software:

- Comprender y manejar la estructura de las aplicaciones complejas.
- Reutilizar estructuras ya existentes (o partes de ellas) para resolver problemas parecidos.
- Planificar la evolución de la aplicación, identificando las partes susceptibles de modificación. De esta forma se pueden abaratar los costes relacionados con los cambios.
- Analizar el buen funcionamiento de la aplicación y el grado de cumplimiento de sus requisitos, tanto iniciales como aquellos que pueden surgir durante su funcionamiento.
- Permitir el estudio de una o más propiedades específicas del dominio de la aplicación.

3.1.3 Descripción de distintas arquitecturas. Ventajas e inconvenientes.

Desde el momento en el que se reconoció, por parte de los ingenieros de software, la existencia de principios y de estructuras de organización específicas para ciertas clases de software, y de que lo pusieran en conocimiento de sus compañeros, se han estado reutilizando estas estructuras como estilos arquitectónicos ya validados. Aunque se está lejos de una clasificación de estilos arquitectónicos totalmente aceptada, si existe una cierta taxonomía inicial [Shaw96].

Un estilo arquitectónico define una familia de sistemas según un patrón de organización estructural. Es decir, un estilo define un vocabulario de

tipos de componentes y conectores y un conjunto de restricciones o reglas sobre cómo deben combinarse. Para muchos estilos puede existir uno o más modelos semánticos que especifican cómo se determinan las propiedades del sistema completo a partir de las propiedades de sus partes.

Podemos estudiar los estilos arquitectónicos específicos respondiendo a las siguientes preguntas:

- ¿Cuál es el vocabulario de diseño (los tipos de componentes y conectores)?
- ¿Cuales son los patrones estructurales permitidos?
- ¿Cuál es el modelo computacional subyacente?
- ¿Cuales son los invariantes esenciales del estilo?
- ¿Cuales son algunos ejemplos comunes de su uso?
- ¿Cuales son las ventajas y desventajas del uso de ese estilo?
- ¿Cuales son algunas especializaciones comunes?

A continuación vamos a exponer brevemente distintos estilos arquitectónicos respondiendo a las cuestiones que se han planteado anteriormente.

3.1.3.1 Cauces y filtros.

En un estilo *cauces-y-filtros* (*pipes-and-filters*) cada componente tiene un conjunto de entradas y un conjunto de salidas. Un componente lee cadenas de datos de sus entradas, realiza alguna transformación local de dichos datos y el resultado se envía por sus salidas. Las salidas pueden comenzar antes de que se hayan consumido completamente los datos de las entradas. Los componentes se denominan filtros y los conectores cauces. Los cauces conectan la salida de un filtro como entrada de otro.

Podemos destacar los siguientes invariantes en este estilo:

- Los filtros deben ser entidades independientes. No deberían compartir el estado con otros filtros.
- Un filtro no conoce la identidad de los otros filtros con los que está conectado. Sus especificaciones podrían limitarse a lo que aparece en las entradas o establecer garantías acerca de lo que se produce en los cauces de salida, pero ellos no pueden identificar a los componentes que hay al final del cauce. Además, la correctitud de la salida de una red de cauces-y-filtros no debería depender del orden en el que los filtros realizan su procesamiento (aunque se asuma una planificación justa).

Dentro de las especializaciones de este estilo más comunes se incluyen los *pipelines* (vea la figura siguiente), los cuales restringen las topologías de interconexiones posibles a secuencias lineales de filtros; los *cauces limitados*, que restringen la cantidad de datos que pueden residir en un cauce; y los *cauces con tipo* que requieren que los datos pasados entre dos filtros tengan un tipo bien definido.

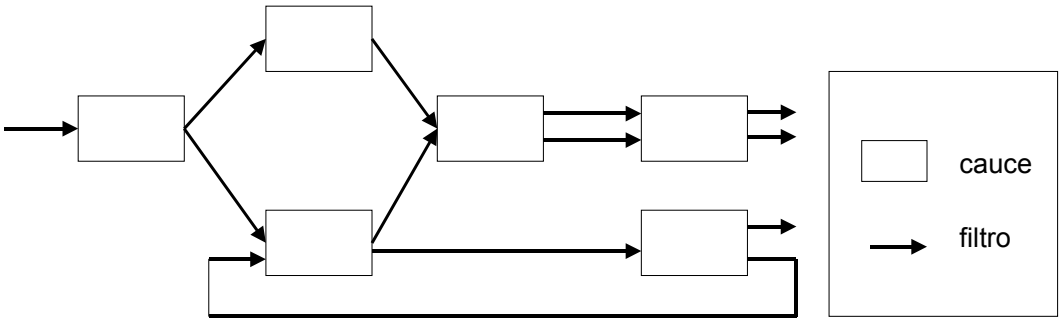


Figura 3.1 Pipeline

Como ejemplos de esta arquitectura tenemos los programas que se escriben en el *shell* de Unix. Unix soporta este estilo proporcionando una notación para la conexión de componentes (procesos Unix). Otro ejemplo son los compiladores, las distintas fases se pueden ver como un conjunto de filtros. Encontramos más ejemplos en las aplicaciones pertenecientes a dominios como el procesamiento de señales, la programación paralela, la programación funcional y los sistemas distribuidos.

Las ventajas de los sistemas cauce-y-filtro son:

- Permiten al diseñador comprender el comportamiento completo de E/S de un sistema como una composición simple de los comportamientos individuales de los filtros.
- Soportan reutilización: cualesquiera dos filtros pueden conectarse poniéndose de acuerdo en los datos que se transfieren.
- Los sistemas son fáciles de mantener y extender: pueden añadirse nuevos filtros o sustituir los antiguos por otros.
- Permiten ciertas clases de análisis especializados tal como el análisis de rendimiento y de interbloqueo.
- Soportan, naturalmente, la ejecución concurrente. Cada filtro puede implementarse como una tarea independiente y ejecutarse potencialmente en paralelo con otros filtros.

Las desventajas son:

- La organización por lotes (*batch*) del procesamiento de los cauces. Este estilo no es bueno para aplicaciones interactivas debido a su

carácter transformacional. Este problema es mas serio cuando se necesita que se mantenga actualizada la pantalla o la ventana.
- Hay que mantener las correspondencias entre dos *streams* (cadenas de datos) separados pero que vayan por un mismo cauce. Esto puede ser un estorbo porque es posible que haya que tener un cuidado especial con la información que llega a un filtro y saber de dónde procede.
- Dependiendo de la implementación, puede forzar a realizar la transmisión de los datos de una determinada forma, añadiendo mas trabajo a cada filtro para analizar y actualizar sus datos. Esto, en cambio, puede hacer que se pierda rendimiento y se incremente la complejidad en la escritura de los filtros.

3.1.3.2 Abstracción de datos y organización orientada a objetos.

En este estilo, las representaciones de los datos y sus operaciones primitivas asociadas están encapsulados en un tipo de datos abstracto (TDA) u objeto. Los objetos interactúan mediante invocaciones a funciones y procedimientos. Dos aspectos importantes de este estilo son:

- Un objeto es responsable de preservar la integridad de su representación (normalmente, manteniendo algún invariante sobre él).
- La representación de los datos de un objeto se oculta a otros objetos.

Cada vez está más extendido el uso de tipos de datos abstractos y de sistemas orientados a objetos. Hay muchas variaciones, por ejemplo, hay sistemas que permiten a los objetos ser tareas concurrentes (actores) y otros permiten a los objetos tener múltiples interfaces.

Las ventajas que nos ofrecen los sistemas orientados a objetos son:

- Ya que un objeto oculta su representación a sus clientes, es posible cambiar su implementación sin que afecte a dichos clientes.
- El ligar las rutinas de acceso con los datos que manipulan permite a los diseñadores descomponer los problemas en colecciones de objetos que interactúan.
- La reutilización tanto de clases como de objetos existentes.

Desventajas:

- La desventaja más importante es que los objetos necesitan conocer a priori la identidad de los otros objetos con los que interactúan. Por tanto, si cambia la identidad de un objeto, es necesario modificar todos aquellos objetos que lo invocan. En un lenguaje orientado a

módulos, esto implica cambiar la lista de importación de cada modulo que utilice el modulo cambiado.

3.1.3.3 Invocación implícita basada en eventos.

En un sistema orientado a objetos, los componentes interactúan mediante la invocación explícita de las funciones y procedimientos que se encuentran en la interfaz de cada componente. Recientemente, se ha prestado un especial interés en una técnica de integración alternativa conocida como invocación implícita, integración reactiva o *broadcast selectivo* (indistintamente). Este estilo tiene sus raíces en los sistemas basados en actores, satisfacción de restricciones, demonios y redes de intercambio de paquetes.

La idea tras la invocación implícita es que un componente, en vez de invocar directamente a un procedimiento, puede anunciar (o repartir) uno o más eventos. Otros componentes del sistema pueden declarar que tienen interés en un evento asociando uno de sus procedimientos al evento. Cuando se anuncia el evento, el sistema invoca automáticamente a todos los procedimientos que habían estado interesados en dicho evento. De esta forma, el anuncio de un evento origina la invocación implícita (no directa) de procedimientos en otros componentes.

Los componentes en este estilo son los módulos cuyas interfaces proporcionan tanto un conjunto de procedimientos (y tipos de datos abstractos) como un conjunto de eventos.

Por ejemplo, en el sistema *Field* [Reiss90], herramientas tales como editores y monitores variables se interesan por los eventos de los puntos de ruptura (*breakpoint*) del depurador. Cuando un depurador para en un punto de ruptura, anuncia un evento que permite que el sistema invoque automáticamente a los procedimientos de dichas herramientas interesadas. El depurador no sabe qué otras herramientas o acciones (si hay) están relacionadas con el evento que anuncia, o lo que se hará cuando lo anuncie

Los sistemas de invocación implícita se usan en entornos de programación para la integración de herramientas, en SGBD para asegurar las restricciones de consistencia (uso de *triggers*), en interfaces de usuario para separar la presentación de los datos de las aplicaciones que manejan dichos datos y en editores dirigidos por la sintaxis para soportar la comprobación semántica incremental.

Las ventajas que nos ofrece este estilo son:

- No es necesario que los componentes que interactúan se conozcan. Basta que un componente sepa los eventos que existen en el sistema para interesarse o no en ellos, sin importar qué componente lo anunciará. No se utiliza comunicación directa.
- Gracias a la modularidad e independencia de los componentes es fácil tanto modificar como añadir, borrar o sustituir eventos y/o componentes. En un sistema pueden existir varios componentes que anuncien el mismo evento, si uno de ellos se borra, los componentes interesados en dicho evento no se ven afectados.

Principales desventajas:

- No se sabe a qué componentes afectan los anuncios de los eventos ni en qué orden se realizarán los procedimientos asociados. Por esta razón, muchos sistemas de invocación implícita también incluyen invocación explícita como forma de interacción complementaria.
- Cuando un componente anuncia un evento, no puede suponer que otros componentes responderán a él.
- A veces se pueden pasar datos junto con el evento, pero en otras situaciones los sistemas de eventos dependen de un repositorio compartido para la interacción. En esos casos, el rendimiento global y la gestión de recursos puede llegar a situaciones críticas si no se aborda este problema.
- El razonamiento acerca de la correctitud puede ser problemático, ya que el significado de un procedimiento que anuncia eventos dependerá del contexto en el que sea invocado. Es decir, no se sigue el proceso de razonamiento clásico de la invocación de procedimientos que necesita conocer pre y post-condiciones antes de la activación del procedimiento.

3.1.3.4 Sistemas en capas.

Un sistema en capas está organizado jerárquicamente, de tal forma que cada capa (componente) proporciona un servicio a la capa superior y se comporta como un cliente para la capa inferior. En algunos sistemas en capas, las capas más internas están ocultas y sólo son visibles las adyacentes, excepto para algunas funciones cuidadosamente elegidas para su exportación. Los componentes implementan una máquina virtual en cada nivel de la jerarquía. Los conectores están definidos por los protocolos que determinan cómo interactuarán las capas.
Los ejemplos más conocidos son los protocolos de comunicación por capas, como el protocolo OSI. En éstos cada capa proporciona una parte de la comunicación a un determinado nivel de abstracción. Otras áreas incluyen las Bases de Datos y los Sistemas Operativos.

Los sistemas en capas tienen varias ventajas:

- Soportan diseños basados en niveles de abstracción incrementales: esto permite partir un problema complejo en un conjunto de problemas más pequeños y, probablemente, más sencillos de resolver.
- Facilidad de extensión, simplemente se añade una nueva capa.
- Permiten la reutilización.

Las desventajas son:

- No todos los sistemas pueden estructurarse fácilmente en capas, pueden existir partes del sistema que no pertenezcan a ninguna capa o que deban estar en varias.
- Puede ser difícil encontrar los niveles correctos de abstracción.

3.1.3.5 Repositorios.

En este estilo existe una estructura de datos central, llamada repositorio, y una colección de componentes independientes que operan sobre el almacenamiento de datos central. Las interacciones entre el repositorio y sus componentes externos puede variar bastante entre diferentes sistemas.

La elección de una disciplina de control nos lleva a dos grandes subcategorías:
- Si los tipos de transacciones a la estructura de datos determinan el o los procesos a ejecutar, el repositorio puede ser una base de datos tradicional.
- Sin embargo, si el estado actual de la estructura de datos central es el disparador principal para la selección de procesos a ejecutar, el repositorio puede ser una pizarra (*blackboard).*

El modelo *blackboard* [Corkill91]presenta normalmente tres grandes partes:

- Las *fuentes de conocimiento* (KS: *knowlegde source*): son parcelas independientes del conocimiento propio de la aplicación. La interacción entre las fuentes de conocimiento tiene lugar solamente a través de la pizarra.
- La *estructura de datos* del *blackboard*, la pizarra: contiene los datos del estado de resolución del problema, organizados dentro de una jerarquía dependiente de la aplicación. Las fuentes de conocimiento realizan cambios en la pizarra y estos cambios van llevando al sistema a la obtención de una solución del problema.

- El *Control*: dirigido por el estado de la pizarra. Las fuentes de conocimiento sólo responden cuando se realizan cambios en la pizarra que les afectan. En la siguiente figura se muestra una arquitectura *blackboard*.

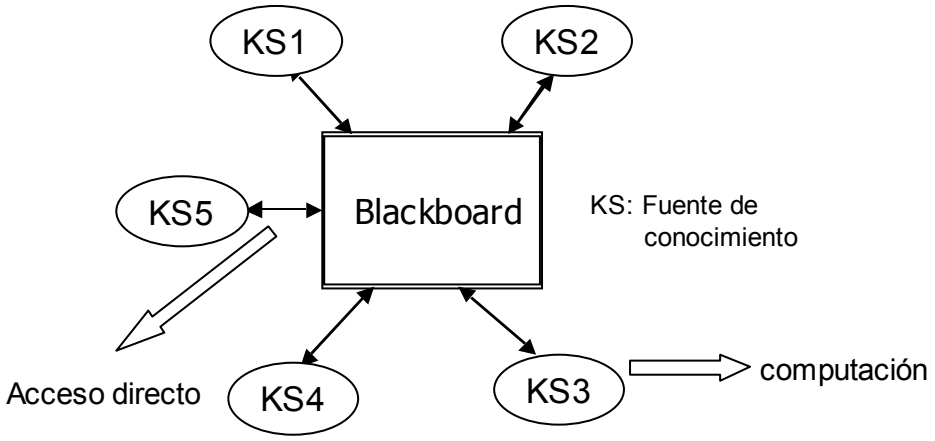


Figura 3.2 Arquitectura blackboard

La invocación de una fuente de conocimiento (KS) se dispara por el estado de la pizarra. Los puntos de control actuales y, por tanto, su implementación, pueden estar en las fuentes de conocimiento, en la pizarra, en un módulo separado o en una combinación de ellos.

Los sistemas *blackboard* se han utilizado tradicionalmente para aplicaciones que requieren interpretaciones complejas del procesamiento de señales, así como en el reconocimiento de patrones. También aparecen en otras clases de sistemas que involucran acceso compartido de datos con agentes débilmente acoplados.

Naturalmente, hay muchos otros ejemplos de sistemas de repositorio. Los sistemas *batch-secuencial* con BD globales son un caso especial. Los entornos de programación se organizan frecuentemente como una colección de herramientas junto con un repositorio compartido de programas y fragmentos de programas. Incluso aplicaciones que han sido consideradas tradicionalmente como arquitecturas *pipeline*, pueden ahora interpretarse como sistemas de repositorios.

Las ventajas de este estilo son:

- Aunque los componentes de un sistema sean independientes, el acceso a la estructura de datos compartida consigue mantener la integridad del sistema.
- Permiten un tratamiento incremental de la información, conforme la estructura de datos va siendo consultada y modificada.

Las desventajas que presenta este estilo son:

- La única forma en la que pueden interactuar los componentes es a través de la estructura de datos compartida, por lo que cualquier modificación de ésta afectará a los componentes.
- Se deben adaptar los nuevos componentes que se añadan al sistema para trabajar con la estructura de datos compartida.

3.1.3.6 Control de procesos (sistemas de control).

En estos sistemas se tienen variables de entrada, variables de salida, valor de variables fijo, y componentes que relizan operaciones para regular el proceso y mantener las variables con unos valores fijos o dentro de un intervalo.

En un sistema de control se toma la información de las variables de entrada y si éstas no cumplen una cierta propiedad (por ejemplo, un determinado valor de temperatura), se realiza una o más acciones (por ejemplo, encender la calefacción) para obtener unas variables de salida con las propiedades deseadas (por ejemplo, temperatura a 20 grados). En particular, las variables están asociadas con elementos que pueden cambiar el sistema de control con el fin de regular el proceso que implementa.

El propósito de un sistema de control es mantener las propiedades especificadas de las salidas de un proceso según unos valores de referencia concretos, denominados "*set points*". Nótese que se está hablando de procesos en el entorno de sistemas de control.

Cuando la ejecución de un sistema software está influenciada por perturbaciones externas, fuerzas o eventos que no son directamente visibles o controlables por el software, entonces podríamos considerar un sistema de control (en general) como una arquitectura software.

Podemos distinguir dos tipos de sistemas de control:

- Sistemas de control de bucle abierto: existe un proceso que transforma el valor de unas variables de entrada en un valor que debe tener unas variables de salida. Las variables de entrada y las de salida sólo están relacionadas por el proceso que las transforma.
- Sistemas de control de bucle cerrado: existe un ciclo en el proceso, las variables de salida pueden ser de entrada en el sistema (*feedback*) o las variables de entrada intervienen en el valor de la salida (*feedforward*).

En esta arquitectura las partes esenciales son:

- Elementos computacionales para manejar las variables, definición de procesos y algoritmos de control.
- Elementos de datos: variables, sensores, valores fijos de referencia para el control de las variables.
- Paradigma del bucle de control: establece qué procesos se deben ejecutar según el estado o valor de las variables, obteniéndose un nuevo estado tras su ejecución.

Estas tres partes se corresponden con un tipo especial de arquitectura de flujo de datos. Los componentes interactúan pasándose datos y cada componente se ejecuta cuando le llegan los datos. Los componentes siempre están funcionando y verificando valores.

Las ventajas de este tipo de arquitectura son:

- Tienen en cuenta elementos externos.
- Reconocen la naturaleza reactiva de los sistemas de control y el hecho de que siempre están en funcionamiento.

La principal desventaja es que esta arquitectura es demasiado específica para los sistemas de control, no es adecuada para otros tipos de sistemas donde no tengan sentido los sensores ni el comportamiento cíclico del sistema.

3.1.3.7 Intérpretes.

Un intérprete incluye el programa en pseudocódigo que va a ser interpretado y el motor de interpretación. El programa incluye el programa en sí y a su estado de ejecución (registros de activación). El motor de interpretación incluye la definición del intérprete y el estado actual de su ejecución. Un intérprete tiene generalmente cuatro componentes:

- Un motor de interpretación para hacer el trabajo.
- Una memoria que contiene el pseudocódigo a interpretar.
- Una representación del estado del motor de interpretación.
- Una representación del estado actual del programa que está siendo interpretado.

Los intérpretes se utilizan normalmente para construir máquinas virtuales que llenen el hueco existente entre el motor de computación esperado por la semántica del programa y el motor de computación disponible en el hardware. En la siguiente figura se representa la estructura de un intérprete.

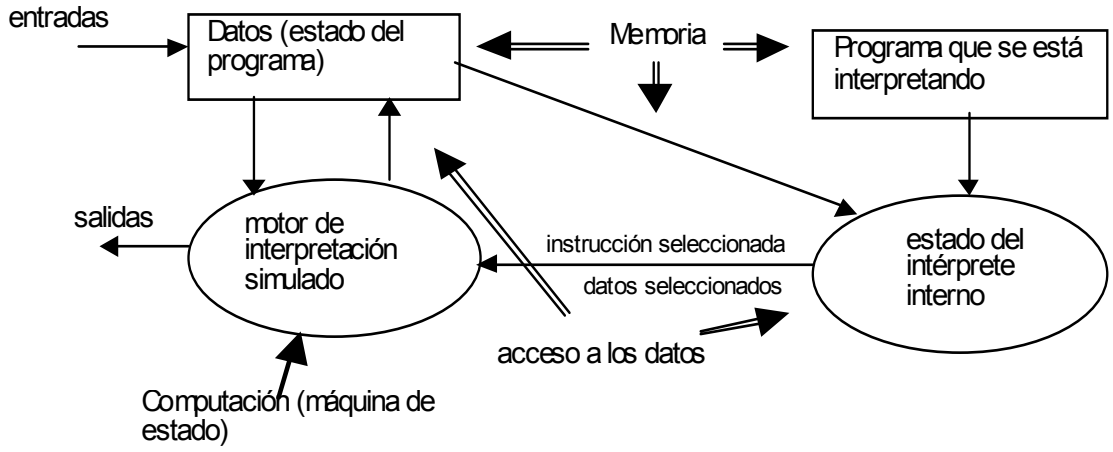


Figura 3.3 Estructura de un intérprete

3.1.3.8 Otras arquitecturas frecuentes.

Existen otros muchos estilos arquitectónicos. Algunos son generales y otros son específicos para determinados dominios. A continuación citamos algunos de ellos:

- *Procesos distribuidos.* Se han desarrollado numerosos sistemas distribuidos para sistemas multiprocesadores. Algunos se caracterizan por sus topologías, tal como "en anillo" o "en estrella". Otros se caracterizan mejor por la clase de protocolos de comunicación entre procesos que utilizan.

  Una forma común de arquitectura de sistemas distribuidos es la organización llamada *cliente-servidor*. En dichos sistemas un servidor representa a un proceso que proporciona servicios a otros procesos (los clientes). El proceso servidor está siempre en ejecución, esperando peticiones de los clientes, encolándolas, procesándolas y respondiendo a ellas. Normalmente, el servidor no conoce a priori las identidades o número de clientes que se comunicarán con él en tiempo de ejecución. Sin embargo, los clientes sí conocen la identidad del servidor y acceden a él a través de una llamada a procedimiento remoto (RPC). También conocen los servicios que éste proporciona.

- Organización de *programa-principal/subrutinas*: Esta organización se corresponde con la organización clásica de los programas a bajo nivel. El programa principal (*main*) actúa dirigiendo y organizando la secuencia de control (normalmente en un orden secuencial) y las llamadas a las subrutinas. Suelen utilizarse métodos como las cartas de estructura para modelar la relación entre los componentes, mostrando no sólo flujos de control, sino también flujos de datos entre subrutinas.

- *Arquitecturas software específicas del dominio (DSSA)*: Proporcionan una estructura organizacional a medida, específica, para una familia de aplicaciones (por ejemplo, un sistema de gestión de vehículos, movilidad de robots, interfaces de usuario estándares). Es decir, se utilizan como patrones de diseño de aplicaciones específicas.

  Además, en muchos casos la arquitectura está tan detallada y es tan específica que se puede generar un sistema ejecutable automáticamente o semiautomáticamente desde su descripción arquitectónica.

- *Sistemas de transición de estados*: Una organización común para muchos sistemas reactivos es el sistema de transición de estados. Se definen en función de un conjunto de estados y un conjunto de transiciones con nombre que hacen evolucionar al sistema de un estado a otro. Tienen asociados métodos de verificación para comprobar en qué estado se está y detectar si se darán o no determinadas transiciones en función del estado actual.

3.1.3.9 Arquitecturas heterogéneas.

En muchos sistemas se puede observar que existe una combinación de distintos estilos. Los estilos arquitectónicos pueden combinarse de distintas formas. Una forma es a través de la jerarquía. Un componente de un sistema organizado en un estilo arquitectónico puede tener una estructura interna que se ha desarrollado en un estilo completamente diferente. Los conectores también pueden descomponerse jerárquicamente (aunque esto sorprenda más). Por ejemplo, un conector cauce puede estar implementado internamente con una arquitectura tipo cola *FIFO* a la cual se accede mediente operaciones de inserción y borrado.

Una segunda forma de combinar los estilos es permitir que los componentes utilicen una mezcla de conectores arquitectónicos. Por ejemplo, un componente podría acceder a un repositorio desde parte de su interfaz pero interactuar como cauce con otros componentes del

sistema y aceptar información de control a través de otra parte de su interfaz (por ejemplo, una BD activa).

Una tercera forma sería elaborar completamente un nivel de descripción de la arquitectura en un estilo arquitectónico completamente diferente.

De todas las arquitecturas descritas, nosotros nos decidimos por una arquitectura hereogénea que combine los siguientes estilos: el estilo de repositorios y, en concreto, su variante *blackboard*, y el estilo de invocación implícita. Como veremos en los siguientes capítulos, gracias a ambos estilos y realizando algunas variaciones, conseguimos la coordinación, comunicación e independencia entre los agentes que componen un sistema software, tanto a nivel de funcionamiento como de evolución.

En la siguiente tabla mostramos de forma resumida distintos estilos arquitectónicos, sus componentes y conectores.

Arquitectura	**Componentes**	**Conectores**
Cauces y Filtros	Filtros	Cauces
Abstracción de datos y objetos	Instacias de TDA u objetos	Invocación de funciones y procedimientos
Invocación implícita	Actores, programas o partes de programas	Anuncios de eventos
Sistemas en capas	Capas	Invocación de funciones y procedimientos de capas adyacentes
Repositorios	Estructura de datos central y programas	Contenido de la estructura de datos central (BD o pizarra)
Control de procesos	Procesos, sensores, controladores y variables	Dispositivos de entrada y salida
Intérpretes	Programa fuente y motor de interpretación	Memoria donde residen el código del programa y los datos del estado actual de su traducción
Sistema Cliente-Servidor	Procesos clientes y servidores	Mensajes

Tabla 3.1 Componentes y conectores de distintos estilos arquitectónicos

3.2 Patrones y Patrones de diseño.

Antes de construir un determinado patrón de diseño para solucionar un problema particular dentro de un sistema, es necesario estudiar los distintos patrones de diseño existentes. Aunque ninguno de ellos resuelva nuestro problema, sí es posible que resuelva parte de él. Concretamente, uno de nuestros objetivos es estudiar el problema de comunicación y coordinación entre los agentes que constituyen un

sistema software. Aunque tengamos clara la arquitectura o arquitecturas que mejor se adaptan a las particularidades del los sistemas software basados en agentes, necesitamos concretar más, bajar de nivel de abstracción, y esto es lo que se persigue con el siguiente estudio.

Por tanto, en este punto vamos a definir los patrones de diseño, vamos a exponer qué es necesario para describir un patrón y qué cualidades debe tener para que se facilite el proceso de búsqueda del patrón más idóneo ante un determinado problema de diseño.

3.2.1 Definición de Patrón y de Patrón de diseño.

El término patrón (*pattern*) y en concreto, patrón de diseño (*design pattern*), se ha extendido mucho en el mundo de la ingeniería del software [Sane]. En concreto, ha tenido especial relevancia entre los diseñadores de software dentro del paradigma de la programación orientada a objetos. Debido a su importancia y estrecha relación con la arquitectura del software, vamos a estudiar la utilidad de los patrones.

Aunque no está clara la diferencia entre los términos "arquitectura del software" y "patrón de diseño", los hemos distinguido por encontrarlos a distinto nivel de abstracción dentro del diseño de software, como se explicará más adelante. De todas formas, todavía existe una gran confusión, y se relaciona el término patrón de diseño (arquitectónico) como sinónimo de estilo arquitectónico.

3.2.1.1 ¿Qué es un patrón?

Según Christopher Alexander (considerado el antecesor de este concepto) [Lea] "cada patrón describe un problema que ocurre recurrentemente en nuestro entorno añadiéndole el núcleo de la solución para dicho problema de tal forma que se pueda utilizar la solución un millón de veces más, sin hacer lo mismo dos veces". Aunque Alexander hablaba acerca de patrones en la arquitectura de ciudades, esto es aplicable a los patrones dentro del desarrollo de software. Nuestras soluciones las expresamos en términos de objetos e interfaces, en vez de muros y puertas como lo hace Alexander, pero el núcleo de ambas clases de patrones es una solución al problema en el contexto. De todas formas, la definición de patrón todavía no está muy madura, no existe una estandarización del término "patrón de diseño" en el mundo del desarrollo de software. Su interpretación afecta a lo que se dice que es o no un patrón. Lo que puede constituir un patrón para un diseñador, no lo es para otro.

Los patrones no son construcciones teóricas [Rising96], sino artefactos descubiertos en un sistema existente formados por unos componentes (con una función específica) y unas relaciones entre éstos. Los patrones forman una base flexible para la reutilización.

Para Jim Coplien [Coplien] un buen patrón debe:

- Resolver un problema: los patrones captan soluciones no obvias, no estrategias o principios abstractos.
- Ser un concepto probado: un patrón capta soluciones ya utilizadas y demostradas, no son teorías o especulaciones.
- Describir una relación: los patrones no describen módulos, sino estructuras y mecanismos del sistema.
- Recoger el componente humano, lo cual es importante.

3.2.1.2 ¿Qué es un patrón de diseño?

Ya que utilizamos como metodología básica la programación orientada a objetos, nos vamos a centrar en los patrones de diseño para un entorno orientado a objetos. Los patrones de diseño son [Gamma94] descripciones de objetos y clases que se comunican y que son fabricados para resolver un problema de diseño general en un contexto particular. Un patrón de diseño nombra, de forma abstracta, o identifica los aspectos clave de una estructura de diseño común que es útil para crear un diseño orientado a objetos reutilizable. El patrón identifica las clases e instancias participantes, sus roles y colaboraciones y la distribución de responsabilidades.

Cada patrón se enfoca sobre un problema o característica de diseño. Describe cuándo se puede usar, si puede ser aplicado junto con otras restricciones de diseño y las consecuencias y efectos de su utilización. A veces, se utiliza código para ilustrar la implementación de un patrón.

En general, un patrón tiene cuatro elementos esenciales:

- *El nombre del patrón*: el nombrar a un patrón aumenta nuestro vocabulario de diseño y permite diseñar a un nivel más alto de abstracción. Encontrar buenos nombres es una de las tareas más difíciles en la descripción de un patrón.

- *El problema que describe cuándo aplicar el patrón*: explica el problema y su contexto. Describe problemas de diseño específicos tal como cómo representar algoritmos usando objetos. Describe las estructuras de clases u objetos que son sintomáticas de un diseño flexible. Algunas veces el problema incluye una lista de condiciones que deben encontrarse antes de que tenga sentido aplicar el patrón.

- *La solución*: describe los elementos presentes en el diseño, sus relaciones, sus responsabilidades y colaboraciones. La solución no describe un diseño o implementación concreta porque un patrón es como una plantilla que puede aplicarse en diferentes situaciones. Sin embargo, el patrón proporciona una descripción abstracta de un problema de diseño y cómo una disposición de elementos (clases y objetos, en nuestro caso) lo soluciona.

- *Las consecuencias*: son los resultados y efectos de aplicar el patrón. Aunque las consecuencias no se oyen frecuentemente cuando se describen decisiones de diseño, son críticas para la evaluación de las alternativas de diseño y para comprender los costes y beneficios de aplicar el patrón. Las consecuencias para el software se refieren frecuentemente al espacio y tiempo dedicado. Ellas pueden dirigir las características del lenguaje o implementación. Desde que la reutilización es un factor frecuente en el diseño orientado a objetos, las consecuencias de un patrón incluyen su impacto sobre la flexibilidad, extensibilidad o portabilidad del sistema. El presentar esas consecuencias explícitamente nos ayuda a comprender y evaluar.

3.2.1.3 ¿En qué se diferencia el concepto de arquitectura del software del de patrón de diseño?

Lógicamente se ven muchas similitudes entre la descripción del concepto de arquitectura del software visto anteriormente y el concepto de patrón de diseño. Es difícil precisar si hablan de lo mismo o son distintos y distinguibles aunque estén muy relacionados.

En nuestra opinión, hablamos de arquitectura del software a un nivel de abstracción más alto, un sistema completo tiene una determinada arquitectura (simple o no). Para describir sus elementos (componentes o relaciones) utilizamos patrones de diseño, que ofrecen un nivel de abstracción más bajo y permiten entrar en detalles. Por tanto, los patrones en sí no describen un sistema completo sino una parte de él. Un sistema no es un único problema, es un conjunto de problemas relacionados de alguna forma pero, a la vez, independientes. Cada uno de esos problemas podría resolverse siguiendo un determinado patrón de diseño.

Kamram Sartipi [Sartipi97] comparte la misma opinión, él opina que *"los patrones de diseño y los estilos arquitectónicos proporcionan distintos niveles de abstracción ... Un estilo arquitectónico es un modelo de generación de distintos sistemas, mientras que un patrón ofrece una solución a un problema particular de un sistema ... los patrones se pueden utilizar para construir partes de un estilo arquitectónico".*

En [Pree95] nos recuerda que Gamma y su colaboradores (1993) definen a los patrones como "*micro-arquitecturas reutilizables que contribuyen a una arquitectura de sistema completa*" que *"... proporciona un vocabulario común para el diseño*". Esto puede avalar nuestro razonamiento, esas micro-arquitecturas no son arquitecturas porque no describen completamente (aunque a un alto nivel de abstracción) todo el sistema. De hecho la clasificación de tipos de patrones, que veremos a continuación, también nos da a entender que la diferencia entre arquitectura (patrón arquitectónico) y patrón de diseño que nosotros entendemos es, por lo menos, aceptable.

Además, [Appleton] opina que muchas veces se utiliza el término patrón o patrón de diseño relacionado con temas de arquitectura del software, de diseño o de programación. Basándose en la diferencia entre ésos tres niveles conceptuales se pueden clasificar los patrones en:

- *Patrones arquitectónicos*: aquellos que expresan una organización estructural fundamental para los sistemas software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye las reglas y directrices para la organización de las relaciones entre ellos.

- *Patrones de diseño*: proporcionan un proyecto para refinar los subsistemas o componentes de un sistema software, o las relaciones entre ellos. Describe una estructura normalmente repetida de comunicación entre componentes que resuelve un problema de diseño general dentro de un contexto particular.

- *Lenguajes (idioms*): es un patrón a bajo nivel específico para un lenguaje de programación. Un *idiom* describe cómo implementar los aspectos particulares de los componentes o las relaciones entre ellos utilizando las características que proporciona el lenguaje dado.

La diferencia entre estas tres clases de patrones está en su correspondiente nivel de abstracción y de detalle. Los patrones arquitectónicos son estrategias a alto nivel que tienen que ver con los componentes a gran escala y con las propiedades y mecanismos globales del sistema. Ellos tienen implicaciones ampliamente generalizadas que afectan a la organización y estructura del sistema en conjunto. Los patrones de diseño definen micro-arquitecturas de subsistemas y componentes. Los lenguajes son técnicas de programación específicas del lenguaje y del paradigma que completan los detalles a bajo nivel internos o externos de la estructura o comportamiento de los componentes.

Existen otras clasificaciones, aunque están muy relacionadas y enfocadas hacia la distinción entre niveles de abstracción.

Aunque es normal encontrar la palabra patrón cuando se habla de arquitectura, por el contexto se puede llegar a diferenciar el propósito de cada uno.

3.2.2 ¿Cómo se describen los patrones de diseño?

Inicialmente se describían los patrones mediante notaciones gráficas ayudadas por párrafos explicativos en lenguaje natural sin ninguna estructura. Sin embargo, las notaciones gráficas no son suficientes (aunque sí útiles e importantes). Ellas sólo capturan el producto final del proceso de diseño: las relaciones entre clases y objetos. Para reutilizar el diseño, debemos también registrar las decisiones, alternativas y consecuencias que nos llevaron a él. Los ejemplos concretos son también importantes porque nos ayudan a ver el diseño en acción.

Coad [Coad95], expone un esquema general para describir un determinado patrón que se compone de los siguientes puntos:

- Una breve introducción y discusión del problema típico que el patrón ayuda a resolver, incluyendo una analogía si es posible.

- Una descripción textual informal del patrón acompañada por una representación gráfica.

- Unas normas de cuándo usar (o no) el patrón, y qué patrones son convenientes para combinarlos con un patrón particular.

Y también proporciona un conjunto de patrones de diseño ya identificados y descritos completamente, de forma que han sido utilizados por mucha gente para sus diseños.

Por otra parte, Gamma et. al (llamados brevemente GoF – *Gang of Four*) [Gamma94] presentan un formato de descripción más elaborado y formal. Cada patrón se divide en secciones de acuerdo a la siguiente plantilla. La plantilla presenta una estructura uniforme de la información, haciendo que los patrones sean más fáciles de aprender, comparar y utilizar.

1. *Nombre del patrón y clasificación*: El nombre del patrón transmite la esencia del patrón. Un buen nombre es vital porque será parte de nuestro vocabulario de diseño. La clasificación del patrón refleja el esquema que se introducirá más adelante (de creación, estructural, de comportamiento).

2. *Propósito*: Una frase corta que responda a las siguientes preguntas: ¿Qué hace el patrón? ¿Cuál es su propósito y razón? ¿Hacia qué característica de diseño o problema se orienta?
3. *También conocido como*: Otros nombres bien conocidos para el patrón, si hay.
4. *Motivación*: Un escenario que ilustre un problema de diseño y cómo lo resuelven las estructuras de clases y objetos del patrón. Nos ayudará a comprender la descripción abstracta del patrón que se da a continuación.
5. *Aplicación*: ¿Cuales son las situaciones en las que se puede aplicar el patrón? ¿Qué ejemplos de diseños pobres puede arreglar? ¿ Cómo se pueden reconocer dichas situaciones?
6. *Estructura*: Una representación gráfica de las clases en el patrón utilizando UML. También utilizan diagramas de interacción para ilustrar secuencias de peticiones y colaboraciones entre objetos.
7. *Participantes*: Las clases y/o los objetos participantes en el patrón de diseño y sus responsabilidades.
8. *Colaboraciones*: Descripción de cómo colaboran los participantes para llevar a cabo sus responsabilidades.
9. *Consecuencias*: ¿Cómo apoya el patrón sus objetivos? ¿Cuales son las consecuencias y resultados de usar el patrón? ¿Qué aspectos de la estructura del sistema pueden variarse independientemente?
10. *Implementación* : ¿Qué sugerencias o técnicas debería saber cuando se implemente el patrón? ¿Hay características dependientes del lenguaje?
11. *Código de ejemplo*: Fragmentos de código que ilustran como se debería implementar el patrón.
12. *Utilidades conocidas*: Ejemplos del patrón encontrados en sistemas reales. Podrían incluirse varios ejemplos de diferentes dominios.
13. *Patrones relacionados*: ¿Qué patrones están fuertemente relacionados con éste? ¿Cuales son las diferencias más importantes? ¿Con qué otros patrones debería utilizarse este patrón?

Si nos fijamos en ambas formas de descripción, la de Coad y la de Gamma, la única diferencia es que el formato de la segunda es, quizás, más detallado y estructurado.

Como hemos dicho, una simple notación gráfica no sirve para describir un patrón, pero actualmente se están desarrollando métodos visuales que pueden mejorar el método de descripción anterior. A. Lauder y S. Kent [Lauder98] ponen de manifiesto en su investigación que las descripciones comunes de los patrones de diseño tienen los siguientes inconvenientes:

- Capturan instancias específicas del patrón: el espíritu del patrón se pierde en los detalles superfluos de las instancias específicas descritas.
- Las descripciones de los patrones existentes cuentan con notaciones de diagramas informales complementadas con notaciones del lenguaje natural. Esto puede originar imprecisión y ambigüedad.

Para abordar dichos problemas separa la especificación de los patrones en tres modelos (*role*, *type* y *class*). El modelo más abstracto (*role-centric*) presenta a los patrones en su forma más pura, capturando su espíritu esencial sin detalles perjudiciales. Un modelo *role* se refina mediante el modelo *type* (añadiendo restricciones específicas del dominio), el cual es también refinado mediante un modelo *class* (concretando más). Utilizan los recientes avances en notación de modelización visual para conseguir una mayor precisión. Lo importante es que esta especificación de patrones visual permite aclarar la comunicación entre los expertos del dominio y los escritores de patrones (y usuarios finales de patrones) y permite soportar una herramienta CASE para los patrones de diseño, permitiendo al diseñador (usuario del patrón) operar a un alto nivel de abstracción sin ambigüedad.

Nosotros, en la definición del patrón de diseño PDN [Paderewski99b] que veremos más adelante, utilizamos la notación de Gamma y su colaboradores, ya que nos parece bastante completa. Además, los modelos propuestos por Lauder y Kent, aunque nos parecen muy atractivos, no se utilizan realmente. Es necesario que los investigadores usen dichos modelos en la definición de sus patrones de diseño y que éstos estén recogidos en un catálogo público y ampliamente conocido como el de los GoF.

3.2.3 El catálogo de patrones de diseño.

Gamma y sus colaboradores [Gamma94] recopilaron un conjunto de 23 patrones de diseño aplicables a la programación básica orientada a objetos y se generó un incipiente catálogo de patrones de diseño. Como dicen sus creadores, este catálogo es útil si se utiliza y se va extendiendo, de tal forma que se cree entre los diseñadores de software, inicialmente orientado a objetos, un vocabulario común. En los últimos años, se han ido añadiendo patrones nuevos y creando nuevos catálogos de patrones aplicables a campos distintos de la programación orientada a objetos. Esto se puede comprobar observando las numerosas aportaciones relacionadas con los patrones de diseño en los congresos de ingeniería del software tanto nacionales como internacionales de los últimos años. En concreto, Kurt Schelfthout y

sus colaboradores [Schelfthout02] proponen un conjunto de patrones de implementación de agentes que describen problemas y soluciones genéricas para implementar los agentes y los sistemas multi-agentes. Definen los "patrones de implementación de agentes" como patrones de diseño orientados a objetos creados para solucionar cómo se representan las características y conceptos de los agentes y los sistemas multi-agentes en los sistemas reales, posiblemente distribuidos. No pretenden diseñar una arquitectura o modelo de agentes, sin embargo, estos patrones ayudan a implementarla. En este artículo también se presenta una importante revisión de una variedad de trabajos de investigadores de este campo donde se presentan más patrones de diseño de agentes y distintas clasificaciones de éstos.

Lógicamente, es necesaria una organización de todo este conjunto de patrones que, presumiblemente, irá creciendo. Los patrones de diseño varían en su granularidad y nivel de abstracción. El grupo GoF clasifica los patrones para facilitar su aprendizaje y orientar la búsqueda de nuevos patrones. La clasificación se realiza bajo dos criterios:

1. *Propósito*: referido a lo que hace un patrón. Los patrones pueden tener distintos propósitos: creación, estructuración y comportamiento. Los patrones de creación se preocupan del proceso de creación de objetos. Los patrones estructurales se preocupan de la composición de clases u objetos. Los patrones de comportamiento caracterizan la forma en la que las clases o los objetos interactúan y se distribuyen las responsabilidades para las cuales fueron creados.

2. *Ambito*: especifica si el patrón se aplica principalmente a clases o a objetos. Los patrones de clases tratan con las relaciones entre clases y sus subclases. Esas relaciones están establecidas a través de la herencia, de tal forma que son estáticas, fijadas en tiempo de compilación (salvo excepciones). Los patrones de objetos tratan con las relaciones entre objetos, lo que puede cambiarse en tiempo de ejecución y son más dinámicas. La mayoría de los patrones utilizan herencia para alguna extensión. Así que sólo los patrones etiquetados como "patrones de clase" son los que se enfocan hacia las relaciones entre clases.

3.2.4 Cualidades de un patrón. Elección y utilización de patrones.

Además de contener los componentes que se han mencionado, un patrón debería tener las siguientes cualidades [Lea]:

- Encapsulación y abstracción: cada patrón encapsula un problema bien definido y su solución en un dominio particular. Debería estar claro el espacio del problema y de la solución.

- Ser abiertos y variables: todo patrón debería poder extenderse o parametrizarse. El patrón debería poder implementarse de diversas y variadas formas (no depende de la implementación).
- Ser general y poder componerse con otros patrones para resolver un problema mayor.
- Equilibrado: cada patrón debe realizar alguna clase de balanceo entre sus fuerzas y sus restricciones (concepto de invariante).

Pero, ¿cómo se selecciona un patrón determinado? Es difícil encontrar un patrón en un catálogo para un problema de diseño determinado, especialmente, si no se conocen los patrones. Aquí se presentan distintos métodos para buscar el patrón que concuerde con un problema determinado:

- Considerar cómo resuelven, los patrones, los problemas de diseño.
- Mirar la sección *Propósitos*.
- Estudiar cómo se interrelacionan los patrones.
- Estudiar los patrones de propósito parecido.
- Examinar la causa del rediseño.
- Considerar qué debería ser variable en su diseño (ver qué es posible cambiar sin rediseñar).

Una vez que se ha elegido un patrón, ¿cómo se utiliza? El siguiente método explica paso a paso como se puede aplicar un patrón con éxito:

1. Leer el patrón. Poner una atención particular a las secciones de *Aplicación* y *Consecuencias* para asegurar que el patrón se adecua al problema.
2. Volver atrás y estudiar las secciones *Estructura*, *Participantes* y *Colaboraciones*. Se tiene que estar seguro de comprender las clases y objetos del patrón y cómo se relacionan entre sí.
3. Mirar la sección *Código de ejemplo* para ver un ejemplo concreto y su implementación.
4. Elegir nombres para los participantes del patrón que sean significativos en el contexto de la aplicación.
5. Definir las clases. Declarar sus interfaces, establecer sus relaciones de herencia y definir las variables de instancia que representan referencias a objetos y datos. Identificar las clases existentes en la aplicación y modificarlas.
6. Definir nombres específicos para las operaciones del patrón. Para ello se utilizan como guía las responsabilidades y colaboraciones asociadas con cada operación. Se debe ser consistente en las convenciones al asignar nombres.
7. Implementar las operaciones que llevan a cabo las responsabilidades y colaboraciones del patrón. Los ejemplos de la sección *código de ejemplo* ayudarán.

Los patrones no deberían aplicarse indiscriminadamente. Frecuentemente, se consigue flexibilidad y variabilidad introduciendo niveles adicionales de composición, y esto puede complicar un diseño y/o costar parte del rendimiento. Un patrón sólo debería aplicarse cuando la flexibilidad que proporciona es realmente necesaria. La sección *Consecuencias* es más útil cuando evaluamos los beneficios y responsabilidades del patrón.

Como veremos más adelante, nosotros, después de estudiar los patrones de diseño de Gamma y sus colaboradores, y otros presentados en congresos nacionales e internacionales, optamos por crear un nuevo patrón de diseño llamado PDN (*Pre-condition Dynamic Notifier*) que se ajustara a nuestras necesidades. Este patrón no nace de la nada, está relacionado con otros patrones de diseño como son: *Facade*, *Mediator* y *Observer* [Gamma94], *Notifier* [Mederos98] y *Event Notifier* [Riehle96].

3.3 Arquitecturas dinámicas.

Puesto que uno de nuestros objetivos o planteamientos iniciales es centrarnos en el desarrollo de software evolutivo, nos interesa estudiar no sólo las arquitecturas del software sino aquellas que sean dinámicas, que permitan crear sistemas que cambien, que se modifiquen a través del tiempo.

Cada vez tiene más importancia esta área dentro de la ingeniería del software ya que la creación de sistemas estáticos no se corresponde con la realidad de los sistemas actuales que queremos modelar. Los cambios en las tecnologías existentes, en los requisitos de los usuarios de los sistemas, en el entorno donde se crean inicialmente los sistemas, etc. hacen que, cada vez sea más necesario pensar en construir sistemas que puedan cambiar de forma dinámica.

3.3.1 Definición de arquitectura del software dinámica.

La arquitectura del software dinámica (ASD) [Anderson00] es un área de investigación bastante activa dentro de la investigación en arquitecturas del software. Todavía hay poco consenso sobre lo que es realmente una arquitectura dinámica y cuándo se debe utilizar para modelar los sistemas del mundo real.

Existe una gran diversidad en cuanto a técnicas de construcción, especificación y análisis de sistemas con arquitecturas dinámicas. Sin

embargo, la falta de consenso en cuanto a lo que es o no una ASD es un problema. J. Anderson opina que, basándonos en una definición unificada de arquitecturas dinámicas, se pueden comparar distintos enfoques, identificar mejoras y finalmente, encontrar nuevas combinaciones interesantes.

A la hora de hablar de ASD, se utilizan frecuentemente los términos "dinamismo" o "dinámica". Luckham y Vera [Luckham95] definen la "dinámica" en el lenguaje *Rapide* [Rapide] como la capacidad de modelar arquitecturas en las que el número de componentes, conectores y enlaces puede variar cuando el sistema software se está ejecutando. Medvidovic y Taylor [Medvidovic97] describen el "dinamismo" como un aspecto de las configuraciones. Las configuraciones que permiten replicación, inserción y borrado y re-conexión de elementos arquitectónicos, estática y dinámicamente, demuestran la propiedad de dinamismo.

Anderson identifica tres tipos de dinamismo:

1. Dinamismo "fácil de construir" (*constructible dynamism*)

Consiste en combinar un lenguaje de descripción, un lenguaje de modificación y un sistema de actualización dinámico. El lenguaje de descripción describe la configuración inicial de la arquitectura de la aplicación. Cuando deba cambiar la arquitectura, los cambios se describen con un lenguaje de modificación. Este lenguaje, normalmente, soporta la adición y la eliminación de elementos arquitectónicos, la activación y desactivación y, en algunos sistemas, la migración de elementos. Un requisito clave en un sistema de actualización dinámico es que debe mantener a la aplicación en el mismo estado en el que estaba antes de un cambio.

En la siguiente figura podemos ver este tipo de dinamismo, donde el cambio del sistema se inicia por algún evento. En la figura podemos ver dos tipos de vistas del sistema, una vista estática, donde tenemos una visión de la estructura del sistema (antes y después del cambio). Y una vista dinámica, donde se tiene la representación de la estructura del sistema (no en todos los sistemas) y el sistema funcionando. También en este último se puede ver el estado antes de cambiar la configuración del sistema y después.

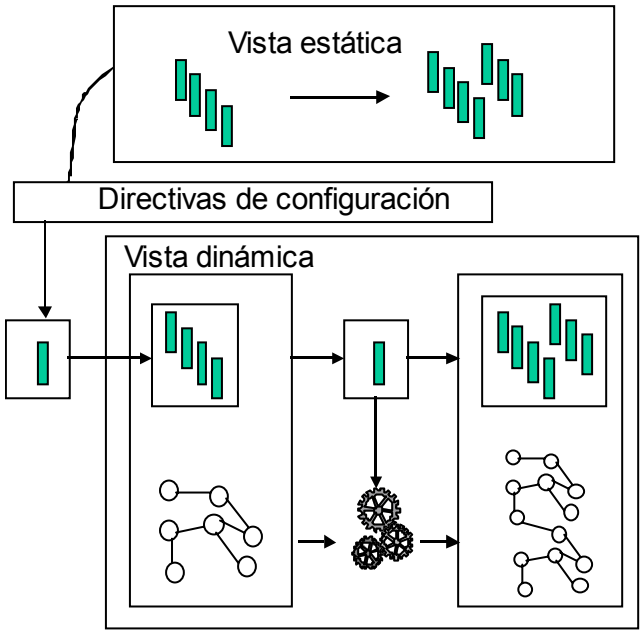


Figura 3.4 Dinamismo "fácil de construir"

2. Dinamismo adaptable (*adaptive dynamism*).

Algunas aplicaciones deben tener la capacidad de reaccionar inmediatamente a ciertos eventos. Aquí, el sistema no puede esperar y la inicialización, selección e implementación deben estar integradas en el marco arquitectónico.

El dinamismo adaptable (ver la siguiente figura) está basado en un conjunto de configuraciones predefinidas que han sido evaluadas durante el desarrollo. Los cambios dinámicos durante la ejecución se inician por algún conjunto de eventos predefinidos, la arquitectura selecciona una configuración alternativa e implementa la re-configuración. Las configuraciones se mantienen internamente en la arquitectura utilizando alguna representación arquitectónica. Para cada configuración existente el sistema necesita conocer cómo proceder para realizar un cambio en ella.

El evento que provoca un cambio en el sistema puede proceder de una herramienta de monitorización. El personal de mantenimiento desarrolla una configuración alternativa que resuelve el problema. En algunas situaciones esto puede ser difícil y el personal de mantenimiento tiene que evaluar distintas alternativas hasta encontrar la configuración adecuada y enviar las directivas de la configuración a la aplicación. Estas directivas pueden actuar sobre la representación de la arquitectura en la aplicación. Cuando ya se han llevado a cabo los cambios, el personal de mantenimiento evalúa y verifica los cambios realizados.

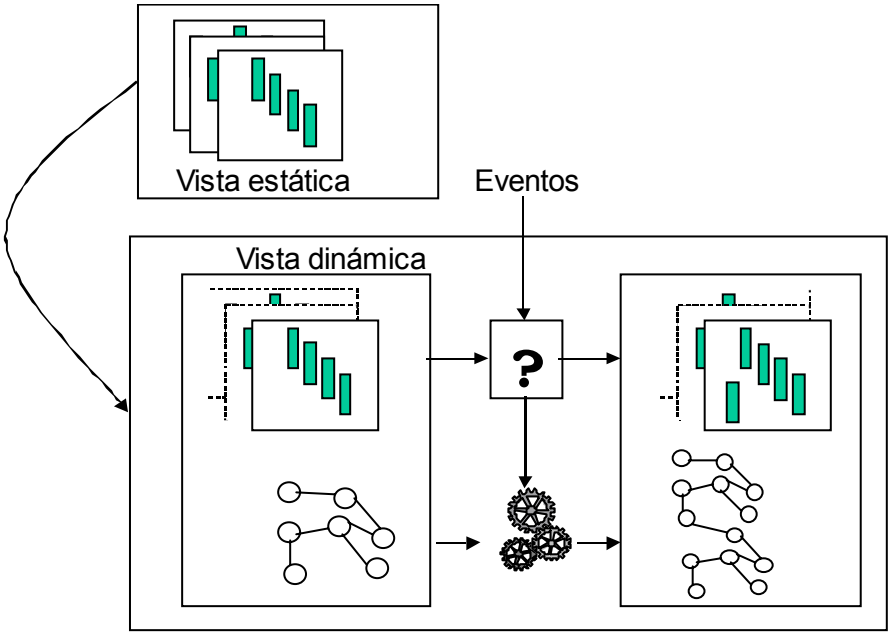


Figura 3.5 Dinamismo Adaptable

3. Dinamismo inteligente (*intelligent dynamism*)

El siguiente paso, después del dinamismo adaptable es eliminar la restricción de tener un conjunto limitado de configuraciones predefinidas. Con respecto a la anterior, una arquitectura con dinamismo inteligente mejora la funcionalidad de las transformaciones de selección. Las arquitecturas adaptables eligen a partir de un conjunto fijo de configuraciones, sin embargo, una arquitectura inteligente incluye la funcionalidad necesaria para permitir la construcción dinámica de las configuraciones candidatas ante una modificación del sistema. La arquitectura puede también evaluar las cualidades que tendrá la configuración. Esto es complejo y, en la práctica, no existen muchos sistemas que utilicen este enfoque.

En la siguiente figura se puede apreciar este tipo de dinamismo.

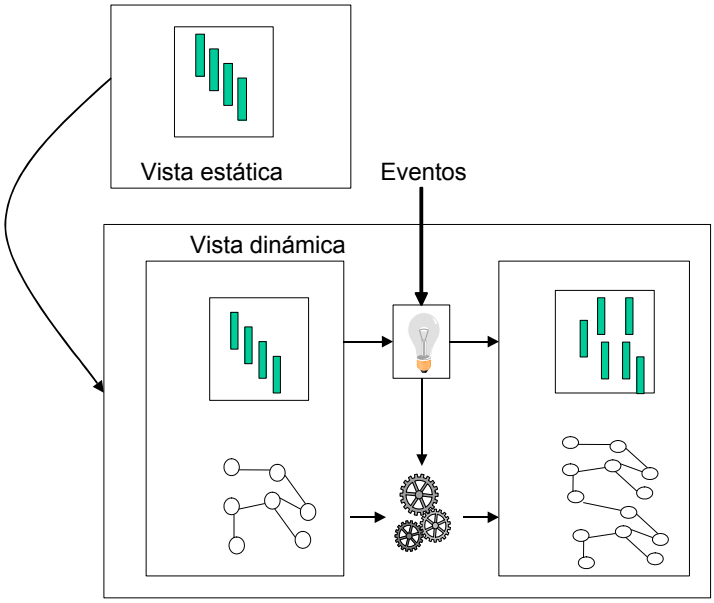


Figura 3.6 Dinamismo Inteligente

Finalmente, podemos dar una definición de arquitectura del software dinámica que se suele repetir por numerosos autores que investigan en este campo. Una arquitectura del software dinámica es aquella que describe cómo se crean, se modifican y se eliminan tanto los componentes como los conectores de un sistema software durante su ejecución de tal forma que el sistema siga siendo íntegro, consistente.

Nosotros, respecto al tipo de dinamismo, utilizamos tanto el dinamismo "fácil de construir" como el dinamismo adaptable.

3.3.2 Modelos y lenguajes de descripción para arquitecturas dinámicas.

Existen numerosos trabajos relacionados con el desarrollo de ADLs y modelos que permiten que un sistema evolucione durante su funcionamiento. Como ya hemos comentado, la evolución de sistemas software es nuestro principal objetivo.

Peschanski y sus colaboradores [Peschanski01] proponen un modelo y un lenguaje específico del dominio, llamado *Scope*, que permite la composición dinámica de arquitecturas del software basadas en componentes. *Scope* se basa en un descendiente del lenguaje *Lisp*, llamado *Scheme*. *Scheme* se ha utilizado en el prototipado de sistemas software.

El lenguaje *Scope* permite la composición incremental y la modificación de arquitecturas del software dinámicas. Los componentes y las conexiones entre éstos se añaden y eliminan en tiempo de ejecución de una forma controlada. También evoluciona la forma en la que los componentes se relacionan entre sí. La operación básica disponible es el establecimiento de una conexión entre dos componentes dados que pueden intercambiar información mediante las emisiones anónimas de eventos. Los tipos de eventos utilizados en este trabajo son para capturar las intenciones del programador a nivel conceptual y garantizar la seguridad en el nivel operacional. Se usa un tipo de algoritmo de inferencia para establecer la correspondencia entre los dos niveles.

Los autores destacan las siguientes contribuciones:

- El modelo propuesto como base es una taxonomía de los datos intercambiados entre componentes (los eventos y sus tipos) sin que tengan una relación directa con la correspondiente representación a nivel de ordenador. Cuando se conectan dos componentes, el desarrollador describe la naturaleza de los datos intercambiados: temperaturas, precios, peticiones a la BD, etc. Dichas informaciones se corresponden a las intenciones del desarrollador, por ello se designan como *tipos intencionales*. Después se utilizará un algoritmo de inferencia que deduce los *correspondientes tipos efectivos*, los cuales describen las categorías de datos que los componentes realmente quieren intercambiarse. Se adopta el problema composicional y su punto de vista conceptual mientras se preservan las necesidades operacionales.

- *Scope* considera entidades de primera clase a los eventos, los tipos de eventos, los componentes, las definiciones de componentes así como a los conectores intencionales.

- La contribución fundamental desde su punto de vista es que una arquitectura puede ser una entidad de primera clase, un componente complejo (*composite*) que puede ser reutilizado para componer nuevas arquitecturas. A esto le denominan “modelo de composición recursivo”.

La formalización se basa en las semánticas de transición de estados debidas a la naturaleza dinámica del modelo propuesto. El problema desde nuestro punto de vista es que lo único dinámico es la composición, pero no la modificación interna de cada componente.

Por otro lado, Canal y sus colaboradores [Canal99a][Canal99b] presentan un lenguaje formal de descripción de arquitecturas del software dinámicas llamado *LEDA*. *LEDA* es un ADL basado en el

cálculo π, un álgebra de procesos que puede expresar de forma natural la *movilidad*, permitiendo la especificación de arquitecturas dinámicas.

LEDA permite la descripción y verificación de propiedades estructurales y de comportamiento de sistemas software. El lenguaje se estructura en dos niveles:

- Los componentes: representan módulos o unidades software, cada uno con una cierta funcionalidad. Pueden ser simples o compuestos (por otros componentes). Se traducen en clases pasivas, no invocan métodos de otros componentes sino que sus métodos son invocados por los roles.

- Los roles: describen el comportamiento de los componentes y los protocolos de interacción que se establecen entre ellos. Se utilizan para la validación, prototipado y ejecución de la arquitectura. Un rol hace la función de conector de un componente con el resto del sistema. Se especifican mediante máquinas de estados y siguen el patrón de diseño Estado [Gamma94].

La especificación de un componente consta de hasta cuatro secciones:

- Interfaz: formada por varias instancias de roles
- Nombres: conjunto de variables locales
- Estructura o composición: formada por varias instancias de componentes
- Conexiones: lista de interconexiones entre roles que indican cómo se construye un compuesto a partir de sus componentes integrantes

LEDA dispone también de mecanismos para la creación e interconexión dinámica de componentes, así como para la derivación de componentes mediante herencia y la instanciación de arquitecturas [Canal99b]. Con este ADL se pueden crear prototipos ejecutables del sistema, ya que a partir de la especificación se genera una implementación en *Java* del sistema.

A pesar de que este trabajo nos parece muy interesante y completo, el formalismo utilizado para describir los roles, el cálculo π, es difícil de utilizar, como cualquier notación formal. Además, no se puede modificar en tiempo de ejecución el contenido de un componente o de un rol, sólo se pueden modificar las conexiones existentes.

Otro trabajo relacionado es el de Ferrer y sus colaboradores [Ferrer02] que presentan el modelo *OASIS* [Letelier98] como un ADL para especificar arquitecturas del software dinámicas en un dominio específico: los sistemas multiagentes [Lorenzo01]. En *OASIS* se distingue entre el modelo de coordinación, el de distribución y el

conjunto de componentes que constituyen un sistema multiagente y permite tratar la problemática asociada a la dinámica de un sistema software. El problema de la evolución se aborda en *OASIS* de una forma reflexiva, incorporando al modelo el metanivel.

OASIS se puede utilizar como un ADL basado en un modelo constituido por las siguiente entidades:

- Componente: pueden ser de dos tipos: Los *Componentes Elementales* que representan a los roles de los participantes y los *Componentes Actividad* que emergen por la agrupación de componentes elementales y un contrato.
- Contrato: conector software que relaciona componentes y define las propiedades de interacción entre ellos.

La arquitectura del software asociada a un sistema vendrá definida por un componente actividad (configuración) que ofrezca la funcionalidad global del sistema.

En *OASIS* se sigue la filosofia de la programación orientada a aspectos (AOP). Cada aspecto de un agente se representa en *OASIS* mediante dos objetos del lenguaje: uno que implemente la funcionalidad completa (nivel base) y otro que reifique en su estado las propiedades del anterior que pueden cambiar (metanivel).

Este modelo posee grandes propiedades, sin embargo, la interacción mediante contratos no nos permite solucionar fácilmente el siguiente problema: la realización de una o más acciones activa la realización de otras, sin importar qué agente las llevó a cabo. Si utilizamos como modelo de coordinación/comunicación los contratos, en éstos se especifica qué agentes o roles de agentes realizan las acciones que interesan a otros agentes. Si, en algún momento, el agente que realiza una determinada acción es eliminado o dicha acción la realizara otro, hay que sustituir el contrato por uno nuevo donde aparezca el nuevo agente, rol, involucrado.

Otro ADL importante es el definido en el proyecto *Rapide* [Rapide] de la Universidad de *Standford*. Realmente *Rapide* es un marco de trabajo que consta de:

- Un lenguaje de tipos (*type lang*uage)
- Un ADL ejecutable
- Un lenguaje de especificación
- Un lenguaje de programación reactivo concurrente

Estos lenguajes permiten compartir un sub-lenguaje de patrones que proporciona la forma de expresar la reacción y la restricción de las computaciones basadas en eventos.

Rapide también se define como un lenguaje orientado a objetos concurrente basado en eventos que se ha diseñado para el prototipado de arquitecturas de sistemas concurrentes y distribuidos. Este proyecto tenía dos objetivos de diseño principales:

(1) Proporcionar construcciones para definir prototipos ejecutables de arquitecturas.
(2) Adoptar un modelo de ejecución en el cual se representen explícitamente la concurrencia, la sincronización, el flujo de datos y las propiedades temporales.

Las arquitecturas se describen mediante: patrones de eventos, interfaces, composición de arquitecturas y emparejamientos (*mapping*) de patrones de eventos. También existen reglas de comportamiento y reglas de conexión. Esto nos sirve para construir modelos de eventos tanto para arquitecturas estáticas como dinámicas. Se utiliza el emparejamiento de patrones de eventos para definir la relación entre dos arquitecturas que se encuentran en diferentes niveles de abstracción. Sin embargo, *Rapide* no soporta composición dinámica aunque permita crear nuevos componentes y no se preocupa de la modificación dinámica del comportamiento de los componentes.

Existen otros modelos y lenguajes de definición de arquitecturas dinámicas como *Dynamic ACME*, *Dynamic Wrigth* y*Darwin*.

ACME [Garlan97] [ACME] posee siete tipos de entidades para representar la arquitectura de un sistema: componentes, conectores, sistemas, puertos, roles, representaciones y *rep-maps*. De ellos, los elementos más básicos de la descripción arquitectónica son los componentes, los conectores y los sistemas.

Los componentes representan elementos computacionales principales y almacenan datos del sistema. Los conectores constituyen las interacciones entre los componentes, son los mediadores entre las actividades de comunicación y coordinación entre componentes. Los sistemas representan las configuraciones dinámicas entre componentes y conectores.

Las interfaces entre componentes se definen mediante un conjunto de puertos. Cada puerto identifica un punto de interacción entre el componente y su entorno. Un componente puede proporcionar múltiples interfaces utilizando diferentes tipos de puertos. Los conectores también tienen interfaces definidas mediante un conjunto de

roles. Cada rol de un conector define a un participante de la interacción representado por el conector. Soporta la descripción jerárquica de arquitecturas. Cualquier componente o conector puede representarse mediante una descripción más detallada, a bajo nivel. El *rep_map* indica la correspondencia entre un componente o conector y su representación interna en el sistema.

Sin embargo, aunque es un lenguaje que proporciona una buena introducción al modelado de arquitecturas no es apropiado para describir cualquier tipo de aplicación, sólo permite describir arquitecturas relativamente simples.

Dynamic Wright es un ADL que extiende *Wright* [Allen98]. *Wright* es un ADL que permite describir y analizar arquitecturas software y estilos arquitectónicos. Con este fin, proporciona elementos en el lenguaje para describir el comportamiento abstracto de los componentes y conectores arquitectónicos y sus relaciones. *Dynamic Wright* incorpora la simulación o notación del dinamismo que no estaban presentes en *Wright*. El uso de *Wright* como lenguaje de descripción y su extensión con algunas características adicionales permite describir fácilmente un entorno dinámico. Entre las notaciones nuevas para describir cambios en la arquitectura, introduce un elemento que denomina "*Configuror*". Añadir un *Configuror* permite expresar el control que se realiza y facilita al diseñador la creación de los sub-sistemas. El *Configuror* se debe encargar de distintas cosas pero las más notables son:

- ¿Cuándo debería ser reconfigurada la arquitectura?
- ¿Qué causaría que la arquitectura se reconfigurara?
- ¿Cómo debería hacerse la reconfiguración?

Allen y sus colaboradores [Allen98] entienden por *sistema dinámico aquel que puede cambiar la composición de los componentes que interactúan en el curso de una computación*. Es decir, es dinámico si puede reconfigurarse, sin embargo no presta atención a las modificaciones que sufren los componentes internamente, sólo a los cambios en las interconexiones. Además, las posibles configuraciones que puede tener un sistema durante su ejecución, deben estar previamente definidas por los desarrolladores.

Para nosotros, un objetivo deseable, es permitir que se varíe la estructura del sistema software mientras funciona y sin limitaciones en cuanto a las posibles configuraciones de los componentes y los conectores. Además de que los propios componentes, los agentes, puedan cambiar también.

El modelo *Darwin* y su lenguaje *LAVA* [Darwin] ha sido desarrollado en el entorno de la "*configuration programming*". *Darwin* integra la

delegación dinámica (herencia basada en el objeto) en los lenguajes orientados a objetos tradicionales. Describe un programa como una configuración jerárquica de componentes. Tiene una representación gráfica y otra textual. Lo que le diferencia con otros ADLs es que siempre se especifica tanto lo que un componente necesita como lo que proporciona.

En general, un componente puede proporcionar y necesitar diferentes servicios. Cada servicio se especifica localmente y esto significa que cada componente puede llevarse de un sistema a otro y puede ser comprobado independientemente. Esto se conoce como independencia del contexto.

Sin embargo, *Darwin* sólo se preocupa de los aspectos estructurales de una arquitectura, no de los funcionales.

3.3.3 Problemas abiertos.

En [ISR] se puede encontrar una recopilación importante de información sobre la investigación en arquitecturas del software. En concreto, existe un punto donde se habla de los problemas que hay que abordar en las arquitecturas del software dinámicas. A continuación listamos un conjunto de estos problemas:

- Respecto al tema del mantenimiento de la integridad del sistema después de realizar un cambio en él, las cuestiones a plantearse son:

- ¿Cómo impacta un cambio en el sistema mientras éste funciona?
- ¿Se puede predecir el impacto de un cambio? ¿en qué grado?
- ¿Cómo se mantiene la integridad del sistema?

- Respecto a la descripción de los cambios en tiempo de ejecución:

- ¿Qué aspectos del sistema cambian? Por ejemplo, la implementación de los componentes, la interfaz externa de éstos o la topología de la arquitectura.
- ¿Qué operaciones de modificación en tiempo de ejecución son apropiadas? Las posibles operaciones incluyen la creación y destrucción de componentes y conectores, el establecimiento de los enlaces de comunicación, la sustitución de un componente por otro, el anidamiento jerárquico y los componentes no anidados, etc.
- ¿Cómo se deberían describir los cambios arquitectónicos? ¿son operaciones implícitas o explícitas?

- Relacionado con la aplicación de los cambios en tiempo de ejecución:

- ¿Cómo se deberían aplicar en tiempo de ejecución los cambios al sistema? secuencialmente (de una a una) o todas las operaciones al mismo tiempo (transacciones), dependiendo de la operación, etc.
- ¿Cómo afectan los cambios al sistema? ¿se crean nuevos procesos y hebras para cada nuevo componente? ¿hay problemas de gestión de memoria?
- ¿Cómo puede preservarse el estado interno de un componente y transferirse a su sustituto?
- ¿Qué aspectos del modelo arquitectónico deberían desplegarse para soportar el cambio y el análisis en tiempo de ejecución?
- ¿Qué dispara un cambio? ¿las peticiones de cambio son explícitas, procedentes de los usuarios finales o, implícitas, mediante un sistema de monitorización (sistemas auto-adaptativos)?

- Correspondencia entre la implementación y el modelo arquitectónico:

- Muchos sistemas establecen una correspondencia uno a uno entre los elementos del modelo y los de la implementación ¿Es adecuada esta correspondencia? ¿es útil soportar correspondencias uno a muchos, muchos a uno y muchos a muchos? Las técnicas de refinamiento arquitectónico puede ser un buen inicio a este estudio.
- ¿Cómo se especifica la correspondencia entre el modelo arquitectónico y la implementación?

3.4 Conclusiones.

La arquitectura del software y, concretamente, la arquitectura del software dinámico es un campo activo dentro de la ingeniería del software. Actualmente, debido a que los sistemas software evolucionan, cambian en el tiempo, se está tomando más interés en poder representar el cambio que puede sufrir la estructura de un sistema software y permitir que dicho cambio se pueda realizar en tiempo de ejecución. Esto nos lleva a la necesidad de nuevos métodos, notaciones y herramientas que permitan modificar dinámicamente un sistema preservando sus propiedades y obteniendo un sistema consistente, íntegro.

Como hemos visto, reservamos el término arquitectura del software para referirnos a la estructura de un sistema complejo, formado de componentes, conectores y donde existen una serie de interconexiones entre los distintos componentes. En este caso el sistema es una entidad global, que tiene sus propias características y propiedades independientemente de las de sus componentes y conectores.

Hablamos de arquitectura del software a un nivel de abstracción más alto: un sistema completo tiene una determinada arquitectura (simple o no) aunque dicha arquitectura pueda describir sus partes utilizando patrones de diseño. Los patrones de diseño en sí no describen un sistema completo sino una parte de él. Sin embargo, si podemos utilizar patrones de diseño ya validados para describir los componentes y esto nos permitirá optimizar el tiempo de creación.

Una vez estudiados un amplio conjunto de estilos arquitectónicos, hemos podido comprobar que necesitamos una arquitectura heterogénea formada por el estilo *blackboard* y el estilo de invocación implícita basada en eventos.

Uno de nuestros principales objetivos es la construcción de sistemas software evolutivos, por tanto necesitamos que la arquitectura utilizada sea dinámica. Esto nos ha llevado a estudiar distintos lenguajes de descripción de arquitecturas dinámicas, sin embargo, ninguno de ellos comparten el concepto de dinamismo o evolución que nosotros adoptamos. Como mucho, se plantean modificaciones dinámicas de las interconexiones existentes en una arquitectura pero no reparan en las modificaciones de los propios componentes.

CAPITULO 4

Agentes y Modelos de Comunicación y Coordinación

4.1 Agentes.

El concepto de agente nace inicialmente dentro del campo de la Inteligencia Artificial y, concretamente, de la Inteligencia Artificial Distribuida (DAI – *Distributed Artificial Intelligence)* donde se estudia el comportamiento de grupos de sistemas inteligentes desde distintos ángulos. Sin embargo su uso se ha extendido rápidamente a otros campos como la Ingeniería del Software.

Durante las últimas tres décadas, los ingenieros del software han estado estudiando las características de los sistema complejos. Una de las características más importantes de tales sistemas es la interacción. Las arquitecturas del software contienen muchos componentes que interactuan dinámicamente, cada uno con su propio flujo de control e involucrado en protocolos de coordinación complejos que son difíciles de construir y comprobar. Las aplicaciones del mundo real se van acercando más a este tipo de sistemas. Por ello, un importante campo de investigación en ciencias de la computación en las últimas dos

décadas ha sido desarrollar herramientas y técnicas para modelar, comprender e implementar sistemas donde la interacción tiene una gran importancia.

Desde la década de 1980, ha ido creciendo el interés en los agentes software [AgentWeb] y los sistemas multi-agentes [Multiagent] y hoy en día es un área de investigación y desarrollo muy activa. Una de las razones más importantes de este interés es que para un diseñador de software es natural el concepto de agente como sistema autónomo capaz de interactuar con otros agentes para satisfacer los objetivos de diseño. Actualmente ya se habla de un nuevo paradigma de programación llamado *Programación Orientada a Agentes* (AOP-*Agent-Oriented Programming*) [Shoham93] e incluso de una incipiente ingeniería del software basada en agentes (ABSE – *Agent-Based Software Engineering*) [Jennings99] [Petrie00] [Wooldridge01].

Muchos sistemas del mundo real constan de un conjunto de entidades que realizan acciones dentro del sistema. Estas entidades suelen ser autónomas, independientes, activas y reactivas. Por ello, como uno de nuestros objetivos es elaborar un modelo que permita describir este tipo de sistemas y las entidades que hemos descrito tienen muchas similitudes con los agentes, hemos visto la necesidad de estudiar lo que nos ofrece este nuevo paradigma. A continuación definimos el concepto de agente, sus diferencias y similitudes con los objetos y los componentes y el concepto de sistema multi-agente.

4.1.1 Definición de agente.

Como muchos autores indican [Jennings96] [Franklin96] [Gómez00] [Schelfthout02], realmente no existe una definición totalmente aceptada de agente. Cada investigador está influenciado por el trabajo que realiza y por las necesidades en cuanto a las características que tiene que tener un agente en su campo, en sus ejemplos o en las aplicaciones que desarrolla.

En lo que la mayoría de los investigadores coinciden es que un agente es una entidad (hardware, software o biológica, una persona) que :

- Percibe su entorno y realiza acciones sobre él
- Es autónomo
- Está integrado en su entorno de tal forma que es capaz de reaccionar dinámicamente a los cambios que se producen en él
- Tiene un estado

Sobre esta definición inicial se van añadiendo nuevas características que permiten diferenciar a unos agentes de otros. Esta diferenciación

da lugar a diferentes clasificaciones de agentes. Por un lado distinguimos entre:

- Agentes hardware
- Agentes humanos
- Agentes software

Por otro lado, si nos fijamos en las tareas que realizan o en su arquitectura de control, los podemos clasificar en tres tipos [Brustoloni91]:

- *Regulation agents*: reaccionan ante las entradas por sensores y siempre saben qué hacer. No planifican ni aprenden.
- *Planning agents*: planifican sus acciones en el sentido de la Inteligencia Artificial (toman sus propias decisiones), pero no aprenden.
- *Adaptative agents*: no sólo planifican sino que aprenden.

Sin embargo, como dice Stan Franklin [Franklin96], partiendo de una definición básica de agente software, podemos crear una subclasificación de acuerdo al conjunto de propiedades que se le añadan o que sea necesario añadirle a los agentes en un determinado dominio de aplicación.

Las propiedades que la mayoría de los investigadores están de acuerdo que definen a un agente software básico [Jennings96] [Wooldridge01] son:

1. *Autonomía* (*autonomy*): los agentes encapsulan algún estado, que no es accesible a otros agentes, y toman decisiones basándose en este estado, sin intervención de otros agentes o elementos externos como el usuario.

2. *Reactividad* (*reactivity*): los agentes se encuentran en un entorno (el mundo físico, dentro de una interfaz de usuario, entre una colección de agentes, en Internet o en una combinación de ellos). Son capaces de percibir los cambios que se produzcan en su entorno y de responder a ellos.

3. *Pro-actividad* (*pro-activeness*): los agentes no sólo actúan en respuesta a su entorno, son capaces de tener un comportamiento dirigido por objetivos (*goal-directed*) tomando la iniciativa cuando sea necesario.

4. *Sociabilidad* (*social hability*): los agentes interactúan con otros agentes (y posiblemente con los humanos) a través de alguna clase

de lenguaje de comunicación entre agentes con el fin de lograr sus objetivos y ayudar a que otros agentes lleven a cabo los suyos.

Si a la definición de agente anterior, se le añade una o más propiedades, obtendremos distintos tipos de agentes software:

- *Agentes software cooperativos*: aquellos que tienen la capacidad de cooperar con otros agentes de su entorno. Puede ser que los agentes necesiten cooperar con el fin de lograr unos objetivos comunes. Esta cooperación no es un simple problema de intercambio de información entre agentes sino un problema de coordinación y colaboración. A veces, se mezcla esta capacidad con la propiedad de sociabilidad pero nosotros la hemos distinguido porque podemos crear un sistema de agentes que no necesiten cooperar sólo comunicarse entre sí.

- *Agentes software inteligentes*: tienen la capacidad de aprender (en mayor o menor grado) [Wooldridge95] [FIPA]. Esta capacidad está muy relacionada con la capacidad de adaptación.

- *Agentes software adaptativos*: tienen la capacidad de adaptarse a los cambios que sufre su entorno y actuar de acuerdo al nuevo entorno.

- *Agentes software móviles*: aquel que tiene la capacidad de moverse por las distintas estructuras de su entorno (nodos de una red, procesos, etc.) para lograr sus objetivos. Las motivaciones de su movilidad pueden ser muy variadas, necesidad de utilizar ciertos recursos, requerir unos servicios remotos, etc.

Podríamos mencionar otros tipos de agentes según la definición del autor o autores y las características complementarias que se añadan a las que ya tiene un agente en su definición básica. También es posible que necesitemos diseñar un *agente híbrido*, es decir, un agente que posea más de una de las características mencionadas anteriormente. Por ejemplo, podríamos necesitar un agente software inteligente cooperativo o un agente software adaptativo y colaborativo. Este último tipo de agente híbrido es el que cubre nuestras necesidades y nos permite modelar los componentes activos de un sistema software.

Por tanto, después de estudiar las distintas clasificaciones y términos utilizados por distintos autores, presentamos, en la siguiente figura, una posible clasificación de los tipos de agentes existentes que sigue la propuesta de Brenner [Brenner98]:

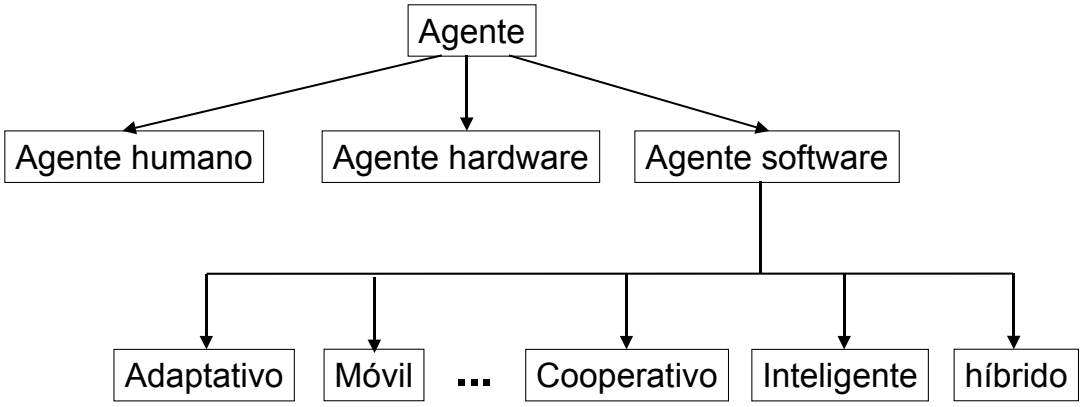


Figura 4.1 Una posible clasificación de tipos de agentes

Hablamos de *sistemas basados en agentes* [Wooldridge01] para referirnos a aquellos sistemas donde la principal abstracción utilizada es la de agente. Los sistemas basados en agentes pueden contener un único agente, aunque su potencial reside en la aplicación de los sistemas multi-agentes. Por tanto, nuestro trabajo se basa en esta definición. Nosotros vamos a centrarnos en definir un modelo que permita definir sistemas software basados en agentes.

4.1.2 Agentes versus Objetos y Componentes.

El paradigma de agentes comparte con el paradigma de la orientación a objetos un conjunto básico de conceptos como son: la abstracción, la modularidad y la encapsulación [Diagne97]. Podemos ver a los agentes como entidades que tienen capacidades tales como el razonamiento y la colaboración/cooperación que no son representadas en el paradigma de la orientación a objetos clásico.

Por otro lado, el software basado en componentes está muy relacionado con el software orientado a objetos, y por tanto, también nos interesa diferenciar entre agentes y componentes.

4.1.2.1 Agentes versus Objetos.

Los objetos se definen como entidades computacionales que encapsulan algún estado, que son capaces de realizar acciones (métodos) sobre dicho estado y que se comunican mediante el mecanismo de paso de mensajes. Aunque existen similitudes entre un objeto y un agente, se pueden distinguir las siguientes diferencias [Wooldridge01]:

- *Grado de autonomía de los agentes y de los objetos*. Un objeto tiene autonomía sobre su estado, tiene control sobre él. Sin embargo, un objeto no tiene control sobre su comportamiento, es decir, otro objeto puede invocar a uno de sus métodos (si es público) cuando lo desee, por tanto no tiene control sobre el inicio o no de la ejecución de un método. Cuando hablamos de agentes, no pensamos que un agente invoca métodos de otros, sino que solicita que se realicen las acciones y son los agentes los que deciden si las realizan o no. En resumen, en la POO, la decisión de invocar a un método se encuentra en el objeto invocante. En el caso de los agentes, la decisión la toma el agente que recibe la petición.

- *Noción de comportamiento flexible*. Otra diferencia es con respecto a la noción de comportamiento autónomo flexible (reactivo, pro-activo y social). El modelo estándar de objetos no dice nada acerca de cómo construir sistemas que integren estos tipos de comportamiento.

- *Ejecución infinita.* La tercera diferencia es que se considera que cada agente tiene su propio flujo de control. Los agentes siempre están activos y típicamente se encuentran en un ciclo infinito donde observan su entorno, actualizan su estado interno y seleccionan y ejecutan una acción a realizar. Sin embargo, los objetos están inmóviles durante más tiempo, se activan sólo cuando otro objeto necesita sus servicios y lo solicita mediante la invocación explícita del método en cuestión.

Esto es verdad para los objetos pasivos. Sin embargo, últimamente se ha trabajado mucho en la concurrencia dentro de la POO. Por ejemplo, el lenguaje *Java* proporciona construcciones para la programación multi-hilos (*multi-thread*). Existen muchos lenguajes de programación disponibles que fueron específicamente diseñados para permitir la programación basada en objetos concurrentes. Pero dichos lenguajes no capturan la idea que tenemos de los agentes como entidades autónomas. Quizás, a lo más cerca de los agentes que llega la comunidad de investigadores centrados en la POO, es a la idea de *objetos activos*:

> *Un objeto activo es aquel que comprende su propio flujo de control [...]. Los objetos activos son generalmente autónomos, lo cual significa que pueden exhibir un comportamiento sin ser iniciados por otros objetos. Los objetos pasivos sólo pueden sufrir un cambio de estado cuando explícitamente se actúe sobre ellos [Booch94].*

Por tanto, los objetos activos son esencialmente agentes que no necesariamente tienen la habilidad de exhibir un comportamiento autónomo flexible. De hecho, no tienen porqué ser sociables ni pro-activos.

En la siguiente tabla se resumen las diferencias esenciales entre los agentes, los objetos pasivos y los objetos activos.

	Agente	Objeto Pasivo	Objeto Activo
Decisión de invocar un método	Del agente que recibe la petición	Del objeto que invoca el método	Del objeto que invoca el método Del propio objeto que realiza la acción
Comportamiento flexible (reactivo, pro-activo, social)	Integrado en el modelo de agentes	No integrado en el modelo de objetos	Integra sólo el comportamiento reactivo
Ejecución	Entidad siempre activa	Sólo se activa cuando se invoca uno de sus métodos	Entidad siempre activa

Tabla 4.1 Diferencias entre agentes, objetos pasivos y objetos activos

No obstante, la POO puede ser una valiosa base para construir sistemas basados en agentes. Los conceptos como modularidad, encapsulación y abstracción pueden aplicarse a los agentes de forma satisfactoria como un primer nivel de estructuración y después se extenderán dichos agentes para que completen el conjunto básico de sus capacidades. De hecho Guessoum y Briot [Guessoum99] estudian cómo extender el concepto de objeto activo para llegar a implementar el concepto de agente. Extiende una plataforma de objetos activos llamada *Actalk* y crea una plataforma multi-agente genérica llamada *DIMA*.

4.1.2.2 Agentes versus Componentes.

El software basado en componentes es una tecnología que está muy relacionada con los sistemas orientados a objetos [Jennings99]. Existen muchos motivos que avalan su existencia pero el principal de ellos es la reutilización de software. Las arquitecturas de componentes software, de las cuales *JavaBeans* [JavaBeans] es el mejor ejemplo, permiten construir software a partir de componentes pre-construidos. Los ejemplos más simples de componentes software proceden del diseño de interfaces de usuario. Cuando se construye una interfaz de usuario utilizando un lenguaje como *Java*, se utiliza un rango de clases pre-definidas para implementar una determinada interfaz.

Como los componentes descienden de la tecnología orientada a objetos, heredan todas las propiedades de los objetos. Por tanto, la mayoría de las diferencias entre los objetos y los agentes comentadas anteriormente también se cumplen entre agentes y componentes. Sin embargo, los agentes comparten con los componentes el concepto de "entidad computacional autocontenida que no necesita de otros para realizar los servicios que proporciona".

Los componentes no tienen por qué ser activos en el sentido en el que se entiende que los agentes lo son. Además, como los objetos, no existe una correspondencia con las nociones de comportamiento reactivo, pro-activo, o social en el componente software.

Los sistemas orientados a objetos, los sistemas basados en componentes o cualquier otro paradigma de programación puede ser utilizado para construir los mecanismos necesarios para soportar a los agentes. Sin embargo, hay que tener en cuenta que el valor de los diferentes paradigmas del software reside en el conjunto de conceptos y las técnicas que proporcionan para la construcción de un software con ciertas características. Es necesario disponer de modelos y herramientas que faciliten la construcción de sistemas basados en agentes y que contemplen los conceptos propios asociados a los agentes. Un ejemplo de esto lo tenemos en Wooldridge y sus colaboradores [Wooldridge01] que presentan una amplia variedad de metodologías de desarrollo existentes, tanto formales como informales, para el análisis y el diseño de sistemas basados en agentes. Después de analizarlas se observa que ninguna de ellas se preocupa del tema de la evolución de los agentes, por lo que nos hemos propuesto construir un modelo que nos permita construir agentes y sistemas multi-agentes capaces de evolucionar.

4.1.3 Sistemas Multi-agentes.

Un sistema multi-agente [Müller96] [Ferber99] (MAS – *Multi-Agent System*) consiste en un grupo de agentes que pueden jugar roles específicos dentro de una estructura organizacional. Los sistemas multi-agente ofrecen una nueva cualidad, el *comportamiento emergente*, es decir, el grupo de agentes es más que la suma de las capacidades de sus miembros. Los investigadores utilizan esta propiedad para construir sistemas para aplicaciones complejas como la gestión de aeropuertos, control de transportes, sistemas de control de tráfico, robótica avanzada, etc. De hecho en *"The German Research Center for Artificial Intelligence GmbH"*, instituto alemán de investigación en el campo de la tecnología de software innovador [DFKI] y declarado centro de excelencia, se están aplicando técnicas de sistemas multi-agentes a tres áreas principalmente:

- Planificación dinámica para el transporte
- Comercio electrónico
- Agentes animados en mundos virtuales

Esto nos viene a confirmar la gran importancia que están tomando estos sistemas. Una propiedad que deben tener los agentes de un sistema multi-agente es la capacidad de coordinación con otros agentes.

Esto se debe a que no tiene sentido un sistema multi-agente donde los agentes actúan de forma totalmente aislada, ellos están dentro del sistema para colaborar en la realización de una o más tareas donde es necesario que se involucren dos o más agentes.

Un sistema formado por agentes que no necesitan colaborar no es un sistema multi-agente. Simplemente es un conjunto de agentes que realizan acciones para su entorno pero no proporciona una visión global de sistema con unas ciertas propiedades. Cada agente persigue y lleva a cabo sus propios objetivos locales. En este caso decimos que los diseñadores tienen una *visión centrada en el agente*, donde sólo se tiene en cuenta el agente y su entorno. Por el contrario, decimos que el diseñador utiliza una *perspectiva global* cuando utiliza un sistema multi-agente, es decir, define al sistema como un conjunto de agentes y sus interacciones.

Un término relacionado con los sistemas de más de un agente es el de *sociedad de agentes* [Jennings96]. Una sociedad de agentes se constituye cuando, por ejemplo, se reúne una colección de agentes organizados para planificar un encuentro entre sus usuarios. Estos agentes persiguen una meta común y, en un momento dado, emerge un comportamiento de grupo inteligente. Sin embargo, cuando se completa la planificación, los agentes se dispersan y quizás, nunca se volverán a reunir en el mimo grupo.

4.1.4 Problemas en el desarrollo de software basado en agentes.

Aunque el desarrollo de software basado en agentes tiene, potencialmente, muchos beneficios, existen un conjunto de problemas intrínsecos a este enfoque. Naturalmente se han construido sistemas de agentes robustos y fiables ya que los diseñadores han encontrado formas de salvar dichos problemas. Jennings y Wooldridge [Jennings99] detectan el siguiente conjunto de problemas:

- Es difícil mantener un balance entre el comportamiento pro-activo y reactivo ya que por un lado, el agente debe perseguir la realización de sus objetivos y por otro, debe mantener una interacción continuada con su entorno. Mantener un balance requiere una toma de decisiones sensible al contexto.

- Como los agentes son autónomos, los patrones y los efectos de sus interacciones son inciertos. Primero, un agente decide por sí mismo, en tiempo de ejecución, qué objetivos requieren una interacción en un contexto dado, la inicia (una o varias interacciones) y cuando se realiza, lleva a cabo los objetivos. Esto implica que no podemos

predecir ni el número de interacciones que realizará, ni el patrón de interacción que usará, ni el tiempo de realización de las interacciones. Segundo, existe un desacoplamiento y un considerable grado de variabilidad, entre la petición de un agente de una interacción y el cómo responde finalmente el agente destino de la interacción. La petición puede ser inmediatamente aceptada, rechazada o modificada gracias a algún intercambio social (negociación, por ejemplo).

- El último problema está relacionado con la noción de comportamiento emergente. Se sabe que la composición interactiva nos lleva a un fenómeno de comportamiento que no puede comprenderse generalmente de forma aislada como la suma de los comportamientos de los componentes individuales. Este comportamiento emergente es una consecuencia de la interacción entre agentes. Dada la sofisticación y flexibilidad de las interacciones entre agentes, está claro que el ámbito de los comportamientos individuales y de grupo inesperados es considerable.

También se describen una serie de problemas no tecnológicos que aparecen repetidamente en proyectos de desarrollo de sistemas basados en agentes. Los más significativos son:

- Sobrestimación del potencial de los sistemas basados en agentes. No se tiene claro en qué tipos de problemas es conveniente utilizar agentes y en cuales no. Es importante conocer tanto los beneficios de la programación basada en agentes como sus limitaciones.
- No saber porqué se utilizan los agentes. Este es un problema común, se utiliza la tecnología de agentes sin saber lo que nos proporcionan.
- Olvidar que se está desarrollando software. No se deben olvidar los principios básicos de la ingeniería del software.
- Olvidar que se desarrolla software multi-hilo. Por naturaleza, los sistemas multi-agentes son sistemas multi-hilos, por tanto no se deben olvidar los principios aprendidos de los sistemas concurrentes y distribuidos y sus problemas inherentes.
- El diseño no explota la concurrencia.
- Decidir que se quiere una arquitectura de agentes propia. Las arquitecturas de agentes son esencialmente diseños para construir agentes. Primero hay que estudiar las arquitecturas de agentes existentes y, en caso de no poder utilizar ninguna, diseñar una propia.
- Los agentes usan “demasiada” IA (inteligencia artificial). En general, tiene más éxito construir agentes con un mínimo de técnicas de IA.
- Ver agentes en cualquier lugar. Se tiene la tendencia de ver cualquier cosa como un agente. Sin embargo esto no es cierto y lleva a unas soluciones extremadamente ineficientes: la sobrecarga de gestionar los agentes y de las comunicaciones entre agentes paliarán

rápidamente los beneficios de una solución basada en agentes. Además, no es útil referirse a los agentes como entidades computacionales de granularidad fina.
- Tener muy pocos agentes. Mientras que algunos diseñadores tienen un agente para cada tarea, otros no reconocen el valor de un enfoque multi-agente. Soluciones con muy pocos agentes fallan porque no explotan el potencial del paradigma de agentes.
- Los agentes interactúan demasiado libremente o de forma desorganizada. Las dinámica de los sistemas multi-agentes es compleja y, puede ser caótica. Si un sistema contiene muchos agentes, entonces puede ser complejo manejarlo eficientemente. En muchos sistemas es necesaria una organización que permita disminuir esta complejidad.

4.2 Mecanismos de comunicación y sincronización.

Como hemos visto en los puntos anteriores, un aspecto importante de los sistemas basados en agentes es su comunicación. Por ello hemos realizado un estudio sobre los distintos mecanismos de comunicación y sincronización existentes.

Este estudio se presenta en el apéndice III, aunque en la siguiente tabla mostramos un resumen de los distintos mecanismos de comunicación y sincronización que se han dividido en dos categorías:

- Mecanismos basados en la existencia de una memoria compartida, común.

- Mecanismos basados en el paso de mensajes.

Mecanismos basados en variables compartidas	Espera ocupada	El proceso se mete en un ciclo en el que continuamente está comprobando que se cumpla una condición.
	Semáforos	Proporcionan las operaciones *wait*, *signal* e *init*. El proceso se bloquea (*wait* sobre un semáforo a 0). Se desbloquea con una operación *signal*.
	Regiones críticas y Regiones críticas condicionales	Definen trozos de código de un proceso que serán protegidos en exclusión mutua. Con RCC permite a un proceso esperar hasta que se cumpla una condición
	Monitores	Un monitor encapsula la definición de un recurso y las operaciones que lo manipulan. Los procedimientos del monitor garantizan la exclusión mutua.
	Expresiones de camino	Permite especificar las restricciones de ejecución de las operaciones definidas dentro de un módulo.
	Sleep-Wakeup	*Sleep* provoca el bloqueo del proceso que la ejecuta y *wakeup* desbloquea al proceso dado como argumento
	Cerrojos o Locks	Es un tipo de dato que sólo puede tomar dos valores: abierto y cerrado. Permite resolver los problemas de exclusión mutua.
Mecanismos basados en paso de mensajes	Paso de mensajes	Permiten resolver los problemas de sincronización y los de comunicación Proporcionan dos primitivas: *send* (enviar mensaje) y *receive* (recibir mensaje). La comunicación puede ser síncrona o asíncrona.
	RPC - Llamadas a procedimientos remotos	Utiliza mensajes para su implementación pero aporta algo más: la transparencia de la comunicación para el usuario el cual utiliza una llamada a procedimiento normal.

Tabla 4.2 Mecanismos de comunicación y sincronización

Nosotros, puesto que nos vamos a basar en la programación orientada a objetos para implementar los sistemas software basados en agentes, utilizamos el mecanismo de paso de mensajes para comunicar a los agentes con su *Controller*. Además, usamos semáforos internamente para resolver los problemas de exclusión mutua que se presentan en el acceso simultáneo a la historia de los agentes.

4.3 Concurrencia en la programación orientada a objetos.

Ya que nuestra herramienta y modelo han nacido dentro del paradigma de la programación orientada a objetos (POO), puede ser útil presentar el análisis de algunos conceptos sobre los cuales nos basamos. Veremos

qué mecanismos de los vistos en el punto anterior nos encontramos en los lenguajes orientados a objetos concurrentes. En el artículo de Briot y sus colaboradores [Briot98], se presenta un estudio muy interesante de los distintos esquemas que han surgido al intentar integrar el paradigma de la programación orientada a objetos con el paradigma de la programación concurrente. Si lo pensamos, las nociones de objeto, dirigidas por el principio de abstracción de datos y el paso de mensajes, son lo bastante fuertes para estructurar y encapsular módulos de computación y lo bastante flexibles para que funcionen en distintas arquitecturas hardware y software.

En dicho artículo se presenta una clasificación de las distintas formas en las que el paradigma de objetos se utiliza en sistemas concurrentes y distribuidos. En esta clasificación se distinguen tres enfoques:

1. Enfoque de biblioteca (*library*): aplica los conceptos orientados a objetos tal y como son para estructurar los sistemas concurrentes y distribuidos con una biblioteca de clases. Cada uno de los distintos componentes, como los procesos o los archivos, tiene asociada una clase. La idea básica es extender la biblioteca de clases más que el lenguaje.

2. Enfoque de integración (*integrative*): consiste en la unificación de los conceptos de los sistemas concurrentes y distribuidos con los de orientación a objetos. Por ejemplo, mezclar las nociones de proceso y de objeto nos da la noción de *objeto activo*, y mezclar las nociones de transacción y de invocación de objeto nos lleva a la noción de *invocación atómica*. Naturalmente, la integración no se hace siempre de esta forma. Además, algunos conceptos entran en conflicto, como la herencia con la sincronización (anomalía de la herencia) y la replicación con la comunicación.

3. Enfoque reflexivo (*reflective*): integra las bibliotecas dentro de un lenguaje de programación basado en objetos. La idea es separar el programa de aplicación de los distintos aspectos de su implementación y el contexto de computación (modelos de computación, comunicación, distribución, etc.), los cuales se describen en forma de *metaprogramas*. Esto hace posible caracterizar sistemas (dinámicos) con un impacto mínimo sobre los programas de aplicación.

Aunque parece que los distintos enfoques están en conflicto, en realidad no lo están. Los tres enfoques no se corresponden con categorías disjuntas de lenguajes y sistemas. Algunos lenguajes y sistemas pueden construirse bajo más de un enfoque, veamos cómo:

- El enfoque de biblioteca está orientado a sistemas construidos y basados en la identificación de abstracciones básicas de concurrencia y distribución. Este enfoque proporciona servicios y construcciones para crear un modelo concurrente o distribuido basado en objetos.

- El enfoque de integración se orienta a la construcción de aplicaciones y se basa en la definición de un lenguaje de alto nivel con unos pocos conceptos unificados. Este enfoque asume un modelo distribuido o concurrente basado en objetos que describe cómo interactúan los servicios.

- El enfoque reflexivo está orientado a las dos cosas, a la construcción de aplicaciones y de sistemas. Esto integrará distintas bibliotecas dentro de un lenguaje de programación o de un sistema. Ayuda a integrar los otros dos enfoques haciendo una separación explícita y construyendo una interfaz entre ambos niveles.

A continuación, veremos con más detalle cada uno de estos tres enfoques.

4.3.1 El enfoque de biblioteca.

La idea básica de este enfoque es aplicar la encapsulación y la abstracción, y posiblemente, también los mecanismos de herencia y clase como herramientas de estructuración para diseñar y construir sistemas de computación concurrentes y distribuidos. Es decir, usar la metodología orientada a objetos y un lenguaje orientado a objetos para construir tales sistemas. La principal motivación es incrementar la modularidad mediante la descomposición de sistemas en distintos componentes con interfaces claras.
Aplicado a los sistemas operativos distribuidos (SOD), este enfoque ha permitido una nueva generación de sistemas como *Chorus* y *Choices* [Choices], basados en el concepto de *microkernel*. La arquitectura de un SOD genérico se organiza como un conjunto de componentes abstractos que se especializarán cuando se instancie el sistema. Tales sistemas son más fáciles de comprender, mantener y extender, y podrían ser más eficientes.

Como ejemplo de este enfoque tenemos el lenguaje *Smalltalk*. Este lenguaje es muy simple y su riqueza proviene de su biblioteca de clases que describe e implementa distintas construcciones, estructuras de datos y entornos de programación sofisticados con herramientas integradas. Otro ejemplo es el lenguaje C++ que al ser ampliamente usado, ha provocado una proliferación de las bibliotecas de concurrencia.

La meta principal de este enfoque es diseñar e implementar abstracciones adecuadas sobre las cuales se pueden construir abstracciones a alto nivel. Uno de los ejemplos más significativos para la programación concurrente es la abstracción *semáforo*, que tiene una interfaz y un comportamiento bien conocidos. Tal abstracción se puede utilizar como base para construir otros mecanismos de concurrencia a más alto nivel. Los mecanismos de clasificación y especialización de la POO son apropiados para organizar abstracciones jerárquicamente.

4.3.2 El enfoque de integración.

Una de las mayores dificultades de la programación concurrente y distribuida es la cantidad de características y conceptos necesarios con los que se trabaja, ya que se introducen conceptos nuevos como el de proceso, semáforo, monitor y transacción. El enfoque de biblioteca es útil para la estructuración de dichos conceptos pero el programador se encuentra con la dificultad adicional de separar los objetos del programa de aplicación de aquellos necesarios para resolver los problemas de sincronización y de comunicación. Este enfoque pretende mezclar ambos aspectos integrando los conceptos y ofreciendo al programador un modelo de objetos unificado.

Distinguimos tres niveles de integración posibles entre los conceptos de la POO [Briot98] y los conceptos de concurrencia y distribución. Estos niveles son relativamente independientes unos de otros y los comentamos a continuación:

1) Un primer nivel de integración entre el concepto de objeto y el concepto de proceso (o de una actividad autónoma) nos lleva al concepto de *objeto activo*. Los lenguajes de actores son ejemplos de lenguajes que están basados en objetos activos. Los objetos que no son activos, se denominan *objetos pasivos*.

2) Un segundo nivel de integración asocia la sincronización con la activación de objetos, llevándonos a la noción de *objeto sincronizado*. El paso de mensajes permite realizar una sincronización implícita entre el emisor y el receptor. Además, frecuentemente asocia mecanismos para el control de la activación de invocaciones a nivel del objeto, por ejemplo, asociando una guarda a cada método. Nótese que el concepto de objeto activo ya implica alguna de forma el de objeto sincronizado, ya que la existencia de una actividad (simple) privada al objeto fuerza realmente la secuencialización de las invocaciones. Algunos lenguajes o sistemas como *Guide* o *Arjuna* asocian la sincronización con los objetos aunque distingan las nociones de objeto y de actividad autónoma. Otro ejemplo más

reciente es *Java*, donde existe un cerrojo (*lock*) privado que está asociado implícitamente con cada objeto nuevamente creado.

3) Un tercer nivel de integración considera al objeto como una unidad de distribución, llevándonos a la noción de *objeto distribuido*. Los objetos son entidades que pueden estar distribuidas y replicadas sobre distintos procesadores. El paso de mensajes se realiza de forma transparente en la invocación de objetos local o remota. *Esmerald* es un ejemplo de lenguaje de programación distribuido basado en la noción de objeto distribuido. También CORBA (*Common Object Request Broker Architecture*) [CORBA] se utiliza en la implementación de sistemas de objetos distribuidos. Se puede integrar el mecanismo de paso de mensajes y el concepto de transacción para soportar la sincronización inter-objeto y la tolerancia a fallos.

Vamos a comentar con más detalle cada uno de los tipos de objetos que se han diferenciado anteriormente: los objetos activos, los objetos sincronizados y los objetos distribuidos.

4.3.2.1 Objetos activos.

La idea básica que nos lleva al concepto de objeto activo es considerar que un objeto tiene su propio recurso de ejecución, es decir, posee su propia actividad. Este enfoque se ve influenciado por los lenguajes de actores. Agha y sus colaboradores [Agha93] realizan un estudio de la concurrencia dentro del paradigma de la programación orientada a objetos basándose en el *modelo de Actores*.

Podemos distinguir distintos niveles de concurrencia del objeto:

- La independencia de las actividades de un objeto se denomina concurrencia *inter-objeto*.

- En distintos modelos de computación, a un objeto activo se le permite procesar varias respuestas simultáneamente, por lo que realiza varias actividades, esto se denomina concurrencia *intra-objeto*. Permitir concurrencia interna, aumenta el poder de expresividad así como la concurrencia global, pero necesita controles de concurrencia adicionales para asegurar la consistencia del estado del objeto y una gestión cuidadosa de los recursos para mantener implementaciones eficientes.

Según Wegner [Wegner90], un objeto activo puede ser:

- *Serial o atómico*: Sólo se procesa un mensaje a la vez (POOL, *Eiffel*).

- *Cuasi-concurrente*: Pueden coexistir varias activaciones de métodos pero, como mucho, uno de ellos no está suspendido (esto es similar a un monitor utilizando variables para suspender a los procesos). Ejemplos: *ABCL/ 1* y *ConcurrentSmalltalk*.

- *Concurrente*: Hay concurrencia intra-objeto pero parte del control aplicado está especificado por los programadores (*ACT++, CEiffel*).

- *Totalmente concurrente*: no se restringe la concurrencia dentro del objeto. Esto significa que tales objetos son funcionales (no tienen estado o al menos no cambian de estado). Los lenguajes de actores soportan objetos totalmente concurrentes, los llaman actores *no-serializados*.

Por otro lado, tenemos el concepto de *objeto reactivo o autónomo*. Un objeto se dice que es *reactivo* si reacciona ante un evento (cuando se recibe un mensaje). Por tanto, la única forma de activar un objeto es mandándole un mensaje. Esto se opone a la idea de proceso, que comienza su actividad en cuanto es creado. La integración entre objeto y proceso (concepto de objeto activo) obliga a que el objeto pueda ser reactivado mediante el comportamiento autónomo del proceso. Esto hace que podamos distinguir también dos familias de objetos activos:

- *Objetos activos reactivos*: Sólo pueden ser activados a través del mecanismo de paso de mensajes (*ACT++, CEiffel*).

- *Objetos activos autónomos*: Pueden activarse antes de que reciba un mensaje (*POOL, Eiffel*).

Aunque ambos modelos son opuestos, cada uno puede utilizarse con facilidad para simular el comportamiento del otro. Un objeto activo reactivo con un método que sea un bucle infinito, se convierte en un objeto activo autónomo. Y un objeto activo autónomo cuya actividad sea aceptar un evento, se convierte en un objeto activo reactivo.

Otro aspecto a tener en cuenta es el tipo de aceptación de mensajes: implícita o explícita. La *aceptación implícita* significa que un mensaje es aceptado automáticamente después de que se reciba (el procesamiento actual puede retrasarse después de la recepción del mensaje debido a los requisitos de sincronización). La *aceptación explícita* significa que el objeto explícitamente indica los tipos de mensajes que va a aceptar (similar a la sentencia *entry* de las tareas de *Ada*).

Relacionando los dos aspectos anteriores, frecuentemente (no siempre), la reactividad implica aceptación implícita y la autonomía implica aceptación explícita.

Por último, es importante el *concepto de cuerpo*. La mayoría de los lenguajes que siguen el modelo de objetos activos autónomos están basados en el concepto de cuerpo. El cuerpo es un conjunto de operaciones centralizadas que explícitamente describe el tipo y la secuencia de las peticiones que el objeto puede aceptar durante su actividad (concepto de *Simula 67* que incluye soporte para las corrutinas). Aunque muchos modelos de objetos activos autónomos están basados en el concepto de cuerpo con aceptación explícita, hay otras alternativas. En *CEiffel*, las operaciones pueden especificarse como autónomas utilizando algún tipo de anotación. Una operación autónoma se ejecuta repetidamente sin ser invocada.

4.3.2.2 Objetos sincronizados.

La presencia de actividades concurrentes requiere cierto grado de sincronización. La sincronización puede estar asociada con los objetos y con la comunicación entre ellos a través de varios (sub) niveles de identificación.

1) *Sincronización a nivel del paso de mensajes*: En el caso de objetos activos, emisor y receptor tienen actividades independientes, y sería útil algún tipo de transmisión de mensajes asíncrona. Esto implica asociar a los objetos activos una cola de mensajes que deberá almacenar los mensajes que llegan (usualmente en el orden de llegada) antes de que el objeto activo esté listo para servirlos.

2) *Sincronización a nivel del objeto(s):* En el caso de objetos activos secuencializados, las peticiones se procesan según el orden de llegada. Distinguimos tres niveles diferentes de sincronización a nivel de objeto que se corresponden respectivamente con el procesamiento interno del objeto, su interfaz y la coordinación entre distintos objetos:

- *Sincronización intra-objeto*: Es necesario incluir algunos controles de concurrencia para asegurar la consistencia del estado del objeto (cuando el objeto realiza simultáneamente varias peticiones). Normalmente, el control se expresa en términos de exclusión entre operaciones. La sincronización intra-objeto es el equivalente en el nivel de objetos a lo que se llama sincronización por exclusión mutua a nivel de datos.

- *Sincronización de comportamiento*: Es posible que un objeto pueda temporalmente ser incapaz de procesar ciertos tipos de peticiones. En vez de devolver un error, puede retrasar la aceptación de la petición hasta que pueda procesarse. Esto hace que la

sincronización de servicios entre objetos sea totalmente transparente. La sincronización de comportamiento es el equivalente a nivel de objetos de lo que se llama sincronización basada en condiciones a nivel de los datos.

- *Sincronización inter-objeto*: Puede ser necesaria para asegurar algún tipo de consistencia, no sólo la individual sino la global (coordinación) entre objetos que interactuan mutuamente. La sincronización intra-objeto o de comportamiento, no son suficientes y se necesita una notación como las transacciones atómicas para coordinar las diferentes invocaciones.

3) *Esquemas de sincronización*: Se han propuesto diferentes esquemas de sincronización derivados de la programación concurrente. Se distinguen los esquemas de sincronización dependiendo de si ésta es o no centralizada:

- *Esquemas centralizados*: tales como expresiones de camino (*path expressions*) especifican de forma abstracta y centralizada la sincronización de un objeto. Como son centralizados, tienden a estar asociados e integrados con la clase.

- *Esquemas descentralizados*: tales como las guardas que especifican a nivel de programa la sincronización de los objetos. Tienden a estar asociados e integrados con los métodos.

En cada uno de los dos tipos de esquemas se debería estudiar la expresividad, reutilización, probabilidad y eficiencia. Vamos a ver distintas variaciones de estos esquemas centrándonos en cómo se integran con el concepto de objeto:

- *Expresiones de camino*: Esquema centralizado. Se especifica en una notación compacta los posibles entrelazados de invocaciones (lenguaje *Procol*).

- *El cuerpo revisado*: Esquema centralizado. En casos complejos, el cuerpo puede describir tanto el comportamiento específico de la aplicación como la lógica de la aceptación de las invocaciones. Esto es un problema cuando se quieren especializar los objetos (reutilizar la sincronización).

- *Sustitución del comportamiento*: El modelo de computación de actores de *Agha* [Agha93] se basa en tres conceptos principales: objetos activos, paso de mensajes asíncrono y sustitución del comportamiento. Cuando se crea un actor, éste se compone de una dirección y un comportamiento inicial (con la dirección se asocia una cola de mensajes de llegada). El comportamiento se puede describir

como un conjunto de variables y un conjunto de métodos, es decir, se describe como un objeto estándar. El comportamiento sirve el primer mensaje de llegada y especifica el comportamiento de sustitución, es decir, el comportamiento que debe tener el siguiente mensaje. Una vez aceptado un mensaje, el actor lo sirve y modifica su comportamiento para aceptar el siguiente mensaje y así repetidamente. Una vez que se especifique el comportamiento de sustitución, puede comenzar la ejecución del siguiente mensaje. Esto implica una concurrencia intra-objeto. Si no se especifica el comportamiento de sustitución, el siguiente mensaje no se procesará. Esto implica sincronización inter-objeto.

- *Estados abstractos*: Construimos un objeto con un estado abstracto que representa el conjunto de métodos habilitados. Después de completar el procesamiento de una invocación, se calcula el siguiente estado abstracto para posibilitar los cambios de estado y habilitar los servicios del objeto actuales.

- *Guardas*: Una guarda es, básicamente, una condición *booleana* de activación que se asocia con un procedimiento. La integración con los objetos es sencilla y natural, cada método tiene asociado una guarda. Las guardas alcanzan una buena integración porque no requieren ninguna sentencia de sincronización en la implementación de las operaciones del objeto. Las actividades se bloquean o se liberan implícitamente. El precio a pagar es el rendimiento. Los contadores de sincronización son contadores que almacenan el estado de invocación para cada método, es decir, el número de invocaciones completadas, comenzadas y recibidas. Si asociamos esto con las guardas tenemos un buen control de sincronización intra-objeto de fina granularidad.

- *Locks* (cerrojos): Es natural asociar un cerrojo con cada objeto o con cada dos (para distinguir entre lectores y escritores) para hacer del objeto, un objeto sincronizado. *Java* es un modelo de objetos sincronizado pero no activo porque mantiene los objetos y las hebras separadamente. A cada objeto *Java* se le asocia un cerrojo cuando se crea. Se puede especificar un método para que se ejecute en exclusión mutua con otros mediante el modificador *syncronize*. La sincronización por condición se gestiona utilizando eventos.

- *Anotaciones*: Se usan anotaciones sobre las operaciones para describir una relación de compatibilidad simétrica entre operaciones. Si la operación *op1* es declarada compatible con la operación *op2*, ambas pueden ejecutarse concurrentemente. Las operaciones incompatibles están en exclusión mutua. La declaración de compatibilidades es más segura que la declaración de requisitos de exclusión.

4.3.2.3 Limitaciones del enfoque de integración.

- *Anomalía de la herencia*: Es normal utilizar la herencia para especializar la sincronización asociada con una clase de objetos. Desafortunadamente la experiencia nos muestra que:

    - la sincronización es difícil de especificar y reutilizar debido a la alta interdependencia entre las condiciones de sincronización para los diferentes métodos.
    - distintos usos de la herencia entran en conflicto (heredar variables, métodos o sincronización).

- Las especificaciones basadas en esquemas centralizados son muy difíciles de reutilizar y, en algunos casos, deben ser completamente redefinidos. Los esquemas descentralizados, que son modulares en esencia, son mejores candidatos para una especialización selectiva.

Otras limitaciones que sólo citaremos son la compatibilidad de los protocolos de transacción, las replicaciones de objetos y comunicaciones en sistemas distribuidos.

4.3.3 El enfoque reflexivo.

En este enfoque se intentan mantener las ventajas de unificación y simplificación (para usuarios finales de programas de aplicación) del enfoque de integración mientras se retiene la flexibilidad del enfoque de bibliotecas (para los usuarios expertos).

La reflexión es una técnica general para describir, controlar y adaptar el comportamiento de un sistema de computación. La idea básica es proporcionar una representación de las características y parámetros importantes del sistema en términos del propio sistema. Las características de representación estáticas así como las características de ejecución dinámicas de los programas de aplicación se hacen concretas en uno o más programas que representan el comportamiento computacional por defecto. Así, un programa de descripción/control se denomina *metaprograma*. La especialización de dichos programas nos permite optimizar la ejecución de los programas de aplicación mediante posibles cambios de la representación de los datos, estrategias de ejecución, mecanismos y protocolos. Se utiliza el mismo lenguaje para escribir los programas y los metaprogramas.

Las principales características de cada enfoque podemos resumirlas en la siguiente tabla:

Biblioteca	- Aplica los conceptos de orientación a objetos para crear las herramientas necesarias de concurrencia y distribución - Extiende la biblioteca de clases, no el lenguaje - Orientado a la construcción de sistemas
Integración	- Unifica los conceptos de orientación a objetos con los de concurrencia y distribución: • Objeto activo → objeto + proceso • Objeto sincronizado → objeto + sincronización • Objeto distribuido → objeto + distribución y replicación - Orientado a la construcción de aplicaciones - Anomalía de la herencia
Reflexivo	- Integra los dos anteriores - Uso de bibliotecas dentro de los lenguajes orientados a objetos - Separación del programa de aplicación de los aspectos de implementación y contexto de composición → metaprogramas (reflexión) - Orientado a la construcción de sistemas y aplicaciones

Tabla 4.3 Resumen de los distintos enfoques para combinar la concurrencia y distribución con la POO

4.4 Modelos y lenguajes de coordinación.

Los modelos y lenguajes de coordinación ofrecen una forma para mitigar los problemas y orientar algunos de los temas relacionados con el desarrollo de sistemas complejos paralelos y distribuidos. Papadopulos y Arbab [Papadopulos98] presentan un estudio muy detallado de los modelos y lenguajes de coordinación, así como de los motivos que llevaron a su desarrollo. También en [Omici01] se presentan un conjunto de artículos muy interesantes que exponen los conceptos de coordinación así como los modelos y tecnologías existentes dentro de este campo y, concretamente, en el dominio de aplicación de los agentes en Internet.

4.4.1 Definición.

Existen muchas definiciones del término coordinación, algunas de ellas son:

- Coordinación significa gestionar las dependencias entre actividades.
- La coordinación es el procesamiento de información adicional realizado cuando múltiples actores persiguen metas que no podría realizar un único actor.
- La coordinación es el proceso de construcción de programas mediante la unión de elementos activos.

Podemos observar que en todas las definiciones aparece una entidad activa (procesos, hebras, objetos, procedimiento, sentencias, etc.) y que es necesario que exista una coordinación entre varias de ellas para poder realizar con éxito una tarea común. La coordinación no sólo implica comunicación. Incluso algunos autores, como Stan Franklin [Franklin], defienden que puede existir coordinación sin comunicación.

Por tanto, se define un *modelo de coordinación* como el *pegamento* que une actividades independientes para crear una entidad completa. Se puede describir un modelo de coordinación [Ciancarini96a] mediante un conjunto formado por tres elementos *(E,L,M)*, donde:

- ***E*** : representa a las entidades que van a ser coordinadas (procesos, hebras, objetos o incluso, usuarios)
- ***L*** : es el medio usado para coordinar las entidades (semáforos, monitores, canales, espacios de tuplas, *blackboards*, *pipelines*, etc.)
- ***M*** : es el marco de trabajo semántico del modelo, conjunto de leyes que describen cómo se coordinan los agentes y qué primitivas de coordinación existen. Un ejemplo de leyes son: comportamiento síncrono o asíncrono y la denominación de las entidades de coordinación implícita o explícita.

Además, asociado con un modelo de coordinación existe un *lenguaje de coordinación* que ofrece la forma en la que se define la sincronización, la comunicación y la creación y finalización de las entidades. El modelo y el lenguaje de coordinación pueden o no estar integrados con el modelo y el lenguaje de computación. El modelo de computación se utiliza para describir el comportamiento (funcionamiento) secuencial de las entidades. Gelernter y Carriero [Gelernter92] defiende la separación argumentando que se proporciona una mayor portabilidad (reutilización) y un soporte para la heterogeneidad de las entidades que deben ser coordinadas.

La *configuración* y la *descripción arquitectónica* están muy relacionadas con el concepto de coordinación. También ven un sistema como un conjunto de componentes e interconexiones y separan la descripción estructural de los componentes del comportamiento de éstos. Soportan la composición de componentes para crear otros más complejos. De hecho, están tan relacionados que algunos autores, como Papadopoulos y Arbab, los clasifican dentro de la categoría de lenguajes de coordinación.

4.4.2 Clasificación de los modelos y lenguajes de coordinación.

Existen muchos factores por los que podríamos clasificar los modelos y los lenguajes de coordinación, por las clases de entidades a coordinar, por la arquitectura utilizada, por la semántica del modelo, etc. Sin embargo, Papadopoulos y Arbab distinguen dos grandes categorías:

- *Orientados a datos* (*data-driven*): la evolución de la computación está dirigida por los tipos y propiedades de los datos involucrados en las actividades de coordinación. El estado de la computación en cada momento está definido por los valores de los datos recibidos y enviados y la configuración actual de los componentes coordinados.

- *Orientados a control o a procesos* (*control-driven, process-oriented*): los cambios en los procesos de coordinación están provocados por el disparo de eventos (entre otras cosas) que cambian los estados de sus procesos coordinados.

Las principales diferencias entre ambas categorías se puede ver en la siguiente tabla:

Orientados a datos	**Orientados a control**
Se mezclan en código los aspectos de coordinación y de funcionalidad	Se separan completamente los conceptos de coordinación y de funcionamiento
Ofrecen un conjunto de primitivas de coordinación	Definen un lenguaje de coordinación donde se trata la parte de computación como cajas negras con una interfaz de entrada/salida bien definida
Se tiende a coordinar datos	Se tiende a coordinar entidades
Dominios de aplicación: paralelización de problemas computacionales	Dominios de aplicación: Modelado de sistemas
No existe una separación de procesos	Separa los procesos en dos grupos: - Los computacionales puros (funcionamento) - Los de coordinación puros

Tabla 4.4 Diferencias entre los lenguajes y modelos de coordinación orientados a datos y los orientados a control

De todas formas, no existe una separación clara entre, por ejemplo, el dominio de aplicación en el que se utilizan los distintos modelos y los lenguajes de coordinación. Por ejemplo, el lenguaje de coordinación orientado a datos, *Laura* [Tolksdorf96], se ha utilizado para modelar sistemas distribuidos mientras que los lenguajes de coordinación orientados a control, *Manifold* [Arbab93] y *ConCoord* [Holzbacher96], pueden usarse para paralelizar programas que realizan un uso intensivo de datos. También existen excepciones en cuanto al grado de

desacoplamiento entre componentes computacionales y de coordinación. Por ejemplo, *Ariadne* [Florijn96] tiene un componente de coordinación independiente y pertenece a la categoría de lenguajes de coordinación orientados a datos.

Vamos a ver cada uno de los modelos con más detalle.

4.4.2.1 Modelos de coordinación orientados a datos.

Muchos de estos modelos parten de la noción de *Espacio de Datos Compartido* (*Shared Dataspace*). Un espacio de datos compartido se basa en mantener una estructura de datos común a través de la cual se comunican indirectamente un conjunto de procesos que están involucrados en alguna computación. Los procesos envían datos a dicha estructura y obtienen datos de ella. Los procesos no necesitan conocer la identidad de los demás.

Los diferentes modelos de esta categoría difieren respecto a ciertos parámetros, por ejemplo:

- La estructura de los datos: conjunto de registros o tuplas a un nivel o a varios.
- Los mecanismos utilizados para recuperar los datos.
- La eficiencia en el uso de los datos y la seguridad.

A continuación vamos a comentar algunos de los modelos y lenguajes de coordinación pertenecientes a esta categoría. Primero hablaremos del lenguaje *Linda*, ya que a partir de él se han desarrollado otros muchos lenguajes de coordinación. Después describiremos brevemente otros lenguajes descendientes en mayor o menor grado de *Linda*, intentando enfatizar las mejoras que proporcionan.

4.4.2.1.1 Linda.

Linda [Carriero89] es, históricamente, el primer miembro dentro de la familia de los modelos y lenguajes de coordinación. Proporciona una forma simple y elegante de separar los aspectos propios de la computación de los aspectos de comunicación.

Este modelo se basa en el paradigma de la *comunicación generativa* (*generative communication*): si dos procesos quieren intercambiarse algunos datos, no se intercambian mensajes ni comparten variables, el proceso que produce los datos genera un nuevo objeto de datos, llamado *tupla* (*tuple*), y lo deposita en una región llamada *espacio de tuplas* (*tuple space*). El proceso receptor puede ahora acceder a la tupla. Esto permite un desacoplamiento de los procesos en el espacio y en el tiempo; ningún proceso necesita conocer la identidad de otro ni es

necesario que todos los procesos involucrados en una determinada tarea existan al mismo tiempo.

Además de *tuplas pasivas*, aquellas que sólo contienen datos, el espacio de tuplas puede contener *tuplas activas*, es decir aquellas que representan procesos que, una vez finalizada su ejecución, se convierten en tuplas pasivas. Esto implica que:

1) La comunicación y la creación de procesos son dos facetas de la misma operación. El resultado es el mismo, añadir un nuevo objeto al espacio de tuplas donde cualquiera puede acceder. El crear un nuevo proceso significa crear una tupla activa y dejarla en el espacio de tuplas.
2) Los datos son intercambiables mediante objetos persistentes, no mediante mensajes transitorios. El receptor puede borrar la tupla generada para el intercambio de datos o dejarla en el espacio de tuplas para otros procesos. Se pueden organizar colecciones de tuplas en estructuras de datos distribuidas, estructuras accesibles a muchos procesos simultáneamente.

Linda propone cuatro operaciones básicas (primitivas de coordinación):

- *in* y *rd* para leer tuplas pasivas y borrarlas o no, respectivamente. Son primitivas bloqueantes ya que suspenden la ejecución del proceso hasta que obtengan la tupla deseada.
- *eval* y *out* para crear nuevos objetos de datos (activos o pasivos, respectivamente). Son primitivas no bloqueantes, una vez que se crean los objetos y se añaden al espacio de tuplas, continúan con su ejecución.

Si un proceso emisor *S* tiene datos para otro proceso receptor *R*, entonces *S* utiliza la operación *out* para generar una nueva tupla (los datos) y *R* utiliza la operación *in* para obtener dichos datos y borrar la tupla o *rd* para obtenerlos sin borrar la tupla. En el último caso, cualquier número de procesos puede leer la tupla y el emisor no necesita conocer cuántos receptores hay (y viceversa). Para crear los procesos *S* y *R*, el proceso inicial utiliza *eval*.

Una tupla existe independientemente del proceso que la creó y se pueden crear estructuras de datos en el espacio de tuplas. Existen dos clases de tuplas, activas y pasivas. Una tupla activa se corresponde con la definición de un proceso. Un proceso puede crear otro mediante la creación de una tupla activa que deja en el espacio de tuplas y que se ejecutará concurrentemente. Una vez finalizado el proceso correspondiente a una tupla activa, se crea una tupla pasiva con el resultado de su ejecución y se deja en el espacio de tuplas. Una tupla pasiva es una serie de campos con tipo, por ejemplo:

("una cadena", 15.01, 17, "otra cadena") ó (0, 1)

La ejecución de una operación *out* provoca la generación de la tupla y la añade al espacio de tuplas. Por ejemplo, *out (0,1).* La recuperación de datos del espacio de tuplas se realiza mediante un *emparejamiento de patrones asociativo* (*associative pattern matching*), es decir, el número y tipo de los parámetros debe coincidir con el de la tupla buscada (argumento de las primitivas *rd* e *in*). En una operación *in* o *rd* se especifica una plantilla a emparejar (*template*). Por ejemplo:

in ("una cadena", ?f, ?i, "otra cadena")

su ejecución origina una búsqueda, por el espacio de tuplas, de tuplas de cuatro elementos donde el primero y el cuarto son fijos y el segundo y el tercero deben ser de los tipos *f (float)* e *i (integer)* respectivamente. Si no se encuentra ninguna tupla que empareje, entonces *in* bloquea hasta que aparezca una. Si hay varias, se elige una de forma no determinista.

Se han construido otras primitivas adicionales como *rdp* e *inp* que son variantes no bloqueantes de *rd* e *in* respectivamente, es decir, que devuelven un valor *false* cuando la tupla que se busca no se encuentra en el espacio de tuplas.

Linda es un lenguaje de creación de procesos y de coordinación que es ortogonal al lenguaje base en el que está empotrado. Este modelo no se preocupa de lo que hacen las distintas entidades activas, de los aspectos computacionales, sino que se preocupa de cómo se crean y se organizan dentro de un programa. Las primitivas de *Linda* son verdaderamente independientes del lenguaje anfitrión. Linda ha sido implementado en un amplio rango de entornos. Se han añadido las operaciones de *Linda* a *C, C++, Fortran, Pascal, Lisp, Eiffel, Java* y *Scheme*, por nombrar algunos. Además, Linda se ejecuta sobre una amplia variedad de máquinas paralelas, multi-computadores de memoria compartida, multi-computadores de memoria distribuida y redes de área local.

Sin embargo, *Linda* tiene algunas limitaciones:

- Existe un solo espacio de tuplas y además no tiene nombre. Este espacio de tuplas es global, estático y persistente.
- No existe soporte para las transacciones.
- No hay forma de identificar y autentificar a los componentes activos que trabajan sobre el espacio de tuplas.

4.4.2.1.2 Modelos y lenguajes de coordinación basados en Linda.

Linda ha inspirado la creación de muchos lenguajes similares, algunos de ellos son extensiones del modelo básico de Linda pero otros difieren significativamente de él. Podemos dividir este conjunto de lenguajes en tres categorías [Omici01]:

- Aquellos que *extienden* el conjunto de primitivas de coordinación: se introducen nuevas primitivas para resolver problemas específicos o enriquecer la expresividad del lenguaje de coordinación. En esta categoría tenemos: *Bonita*, *WCL, KLAIM, Jada, T Spaces* y *JavaSpaces*.
- Aquellos que *modifican la semántica* del lenguaje o que utilizan reglas de control para el uso de las primitivas de coordinación. En este caso tenemos *Law-Governed Linda, MARS, LuCe* y *TuCSoN*.
- Aquellos que *modifican el modelo*: se modifican las primitivas o el acceso asociativo al espacio de datos compartido (o ambos). Algunos ejemplos son: *Bauhaus Linda, LAURA, LIME* y *Ariadne.*

A continuación enfatizamos lo más importante de cada uno de ellos

Bonita [Rowstron97] intenta optimizar y ampliar el funcionamiento de las primitivas de coordinación. Permite trabajar con múltiples espacios de tuplas y añade nuevas operaciones de manipulación de tuplas. Proporciona operaciones de recuperación de tuplas por parte de los procesos más eficientes. El proceso de búsqueda sobre el espacio de tuplas se trata independientemente de la acción de entrega de la tupla al proceso. Por tanto se pueden iniciar varias acciones de búsqueda sobre el espacio de tuplas que se realizan en paralelo. De esta forma, se puede disminuir el tiempo necesario para atender peticiones de varios procesos.

WCL [Rowstron98] fue diseñado para soportar la coordinación de agentes sobre Internet. Utiliza multiples espacios de tuplas y permite accesos síncrono, asíncrono, y mediante *streams*. Utiliza el mismo mecanismo de emparejamiento que *Linda* y tiene 19 primitivas no extensibles por el programador. Algunas de éstas se han heredado de *Bonita.* La distribución de las tuplas y de los espacios de tuplas es transparente a los agentes. Permite la movilidad de los agentes. No soporta transacciones.

KLAIM [De Nicola98] *(Kernel Language for Agent Interaction and Mobility*) soporta un paradigma de programación donde los procesos pueden moverse de un entorno de computación a otro. Permite multiples espacios de tuplas y proporciona un conjunto de operaciones heredadas de CCS. Los principales conceptos que utiliza son los procesos (entidades activas), los nodos (lugares dentro de un entorno) y las redes

(conjunto de nodos). Un concepto fundamental es el de localidad (*locality*), lugares en la red donde se encuentran tanto las operaciones como las tuplas. Debido a la movilidad, los aspectos claves de este lenguaje son la privacidad y la integridad de los datos. Existe un implementación en Java de este modelo llamado *KLAVA*.

Jada [Ciancarini96b] es un lenguaje de coordinación para Java que puede usarse para coordinar componentes paralelos y distribuidos. Sustituye el espacio de tuplas por un espacio de objetos y permite la creación de múltiples espacios. La entidad de coordinación básica es el espacio (*space*). Además de las primitivas de coordinación básicas, proporciona otras como *readAll*, *inAll*, *getAll* y *getAny* que resuelven el problema de leer una única tupla en cada operación de lectura sobre el espacio de datos compartido. Además, todas las primitivas tienen asociado un tiempo durante el cual tienen que ejecutarse. Las primitivas de coordinación son todas no bloqueantes. Permite movilidad de código.

T Spaces [Wyckoff98] es un producto de la sección de investigación de IBM propuesto como un intermediario (*middleware*) basado en la coordinación para un amplio rango de arquitecturas software. Utiliza servidores monolíticos de espacios de tuplas basados en Java y protocolos de comunicación apropiados para que los clientes puedan comunicarse con ellos. Se permiten multiples espacios de tuplas. Tiene operaciones de lectura múltiple e introduce una operación nueva, *rhonda*, que implementa una "cita" (*rendezvous*) entre dos clientes, el que introduce la tupla en el espacio y el que quiere leerla. La API de *T Space* permite al programador definir un nuevo conjunto de primitivas de coordinación. El mecanismo de acceso asociativo al espacio de tuplas puede realizarse también mediante consultas (*queries*). El soporte de seguridad incluye la identificación (mediante nombre de usuario y contraseña) y el uso de listas de control de acceso por cada espacio de tuplas. Además, permite que un agente solicite la notificación de la inserción o borrado de una tupla en el espacio de tuplas (notificación de eventos) y se permite el acceso directo entre los clientes del espacio de tuplas y el servidor si se encuentran en la misma JVM (máquina virtual Java).

JavaSpaces [Freeman99] ha sido desarrollado por SUN Microsystems. Su objetivo es facilitar el diseño y la implementación de las aplicaciones distribuidas mientras que se proporciona una interfaz uniforme para diferentes clases de servicios de información. Utiliza espacios de objetos. La interfaz define las operaciones básicas de *Linda* aunque con diferentes nombres. Soportan transacciones, notificaciones de eventos y asocian un tiempo de vida a las tuplas después del cual serán eliminadas del espacio de tuplas.

Law-Governed Linda [Minsky94] impone un conjunto de "leyes" que tienen que cumplir los procesos que desean realizar algún intercambio. Existe un controlador para cada proceso del sistema y todos los controladores tienen una copia de las leyes. Un controlador es responsable de interceptar cualquier intento de comunicación entre el proceso que controla y el resto. La comunicación se permite sólo si se cumplen las leyes. También permite la existencia de múltiples espacios de tuplas y tiene un conjunto de primitivas propias. Las leyes que hemos comentado pueden ser definidas en cualquier lenguaje, aunque los creadores de este lenguaje usaron *Prolog*.

MARS [Cabri00] (*Mobile Agent Reactive Spaces*) es una arquitectura de coordinación desarrollada en la Universidad de Modena y Reggio Emilia que implementa *espacios de tuplas reactivas programables* (*programmable reactive tuple spaces*) para aplicaciones de agentes móviles basados en Java. Parte de que una red puede modelarse como un conjunto de lugares. Existe un espacio de tuplas local a cada entorno de ejecución. Cuando un agente llega a un nodo, se le comunica la referencia del espacio de tuplas que pueden usar para coordinarse con las demás entidades del lugar. El espacio de tuplas no es un simple repositorio sino que permite especificar reglas de coordinación entre agentes en términos de reacciones, de esta forma se consigue una separación clara entre los temas algorítmicos y los de coordinación. Las reacciones representan reglas de coordinación específicas de la aplicación y se codifican como *meta-tuplas* almacenadas en un espacio de meta-tuplas.

LuCe [Denti99] (*Logic Tuple Centres*) sirve para construir sistemas multi-agentes constituidos por agentes autónomos, pro-activos y posiblemente hetereogéneos. Su principal contribución es la introducción del concepto de *centro de tuplas* (*tuple centre*). Es un espacio de tuplas mejorado que puede trabajar como un medio de coordinación programable. Las tuplas no se usan sólo para representar datos de la aplicación sino que también describen el comportamiento de un centro de tuplas, a estas tuplas se les llama *tuplas de especificación* (*specification tuples*). Estas definen las reacciones del centro de tuplas a los eventos de comunicación que entran o salen de él. Esto permite desacoplar la representación actual del conocimiento en un centro de tuplas de la percepción de los agentes de dicho conocimiento.

TuCSoN [Omici99] está inspirado en el modelo LuCe del cual hereda el concepto de centro de tuplas y el lenguaje de especificación ReSpecT. Cada centro de tuplas está en un nodo de la red y tiene asociado un nombre que lo identifica. Permite a los agentes realizar operaciones sobre centros de tuplas tanto de forma transparente como no transparente respecto a su posición dentro de la red.

Bauhaus Linda [Carreiro94] implementa múltiples espacios de tuplas (*multisets*). No diferencia entre tuplas y espacios de tuplas, tuplas y anti-tuplas (es decir, plantillas de tuplas) y tuplas activas y pasivas. Las primitivas no trabajan sobre tuplas sino sobre espacios de tuplas e introduce nuevas primitivas que permiten mover tuplas por los distintos niveles de espacios de tuplas.

Objetive Linda [Kielmann96] es otra variante directa del modelo básico de Linda e influenciado por *Bauhaus Linda*. Introduce un *modelo de objetos* apropiado para los sistemas abiertos y la independencia del lenguaje de programación anfitrión. Los objetos utilizan el espacio de objetos y se soportan jerarquías de múltiples espacios de objetos. Las operaciones son variantes de las primitivas de coordinación de Linda pero donde los elementos insertados o borrados son objetos.

LAURA [Tolksdorf96] es un enfoque derivado del modelo de coordinación tipo Linda para sistemas distribuidos abiertos. En éste, los *agentes* ofrecen *servicios* según sus funciones. Los agentes se comunican a través de un *espacio de servicios (service-space)* compartido por todos los agentes y mediante *formularios (forms)* de intercambio. Un formulario puede contener una descripción de un *servicio ofrecido* (*service-offer*), un *servicio requerido* (*service-request*) con argumentos o un *resultado de servicio* (*service-result*) con los resultados. Para cada uno de ellos existen primitivas que permiten introducirlos y recuperarlos en el espacio de servicios que es monolítico y tolerante a fallos. Se utiliza un lenguaje de descripción de interfaces para describir las operaciones que constituyen la interfaz de un proceso con el fin de identificar los servicios.

LIME [Picco99] (*LInda in a Mobile Environment*) tiene como principal objetivo la movilidad. Introduce el concepto de *espacio de tuplas compartido temporalmente* (*transiently shared tuple space*). Cada entidad (agente o dispositivo físico) está asociada a un ITS (*Interface Tupla Space*) que puede definirse como un espacio de tuplas personal. Cuando varios agentes móviles se encuentran en el mimo lugar físico, sus ITSs se fusionan automáticamente. Esto permite la coordinación entre entidades a través de espacios de tuplas compartidos temporalmente. Como consecuencia de esto, los agentes tienen espacios de tuplas variables dependiendo de su posición física. Los agentes pueden estar asociados a múltiples ITS que tienen un nombre que los identifica. Se pueden definir ITS privados, es decir, aquellos que no se van a compartir. Implementa las operaciones *read*, *in* y *out*. Otra característica interesante de LIME es su capacidad de reacción a los eventos. Para ellos tiene una sentencia *T.reactsTo(s,p)* donde *s* es un fragmento de código a ejecutar cuando se empareje la tupla *p* en el espacio de tuplas *T*.

Ariadne [Florijn96] y su lenguaje de modelización *HOPLa* son un intento de utilizar una coordinación tipo *Linda* para gestionar procesos colaborativos. Al igual que los anteriores, *Ariadne* utiliza un espacio de trabajo compartido que mantiene los datos en una estructura en árbol y que es auto-descriptivo, en el sentido de que la operación de añadir un dato tiene asociadas unas restricciones (es decir, definiciones de tipo) que gobiernan su estructura. Los procesos se definen mediante el lenguaje *HOPLa* (*Hybrid Office Process Language*) y utiliza el concepto de *registros flexibles* (*flexible records*). Las tareas se definen utilizando el término *Action* y existen los siguientes operadores de coordinación:

- *Serie* para la ejecución secuencial
- *Parl* para la ejecución paralela
- *Unord* para la ejecución en un orden aleatorio

4.4.2.1.3 Otros modelos y lenguajes de coordinación orientados a datos.

LO (*Linear Objects*) [Andreoli96] es un lenguaje orientado a objetos basado en el modelo computacional "*Interaction Abstract Machines*". Relaciona el concepto de interacción con la teoría de la Lógica Lineal. Utiliza reescritura de multiconjuntos (*multiset rewriting*) y comunicación asíncrona a través de *broadcasting*. Los multiconjuntos representan los estados de los agentes. Se pueden crear o destruir agentes. Un programa LO es un conjunto de reglas de reescritura .

MESSENGERS [Fukuda96] es un paradigma de coordinación para sistemas distribuidos, particularmente para la computación móvil. Está basado en el concepto de *Messenger* que es un mensaje autónomo que no sólo contiene datos sino también procesos (es decir, un programa junto con su estado actual de ejecución). Un sistema (distribuido) es una colección de funciones que son coordinadas por un conjunto de *Messengers* que navegan libre y autónomamente por la red. Soportan coordinación inter e intra objeto.

Synchronizers [FrØlund93] están basados en el modelo de Actor [Agha93] y constituyen un conjunto de herramientas para expresar patrones de coordinación en lenguajes multi-objetos basados en restricciones de especificación que limitan las invocaciones de un conjunto de objetos. Las restricciones se definen de forma abstracta y a alto nivel y son independientes del tipo de paso de mensajes que utilice el lenguaje de computación. Los *Synchronizers* deciden si una acción se puede llevar a cabo o no dependiendo de si se cumplen o no las restricciones.

4.4.2.2 Modelos de coordinación orientados a control.

Los lenguajes de coordinación orientados a control se basan en la observación de los cambios de estado en los procesos y, posiblemente, en la difusión de eventos. Los procesos son tratados como "cajas negras" y la gestión de datos dentro de un proceso no le concierne a su entorno. Los procesos se comunican con su entorno mediante interfaces bien definidas a las cuales normalmente se les llama *puertos de entrada* y *salida*. A través de los puertos los procesos envían mensajes de control o eventos con el propósito de que otros procesos conozcan su estado o para informar de los cambios de estado. Muchos de los lenguajes de coordinación que se van a comentar recogen este tipo de comunicación que sigue un formalismo tipo *Occam* (tipo CSP) [CSP]. Sin embargo pueden diferir en aspectos como:

- Si los eventos son unidades simples que significan cambios de estado o pueden tener parámetros (valores de datos y tipos)
- El mecanismo de difusión de los eventos utilizado (uno o más *streams*, comunicación síncrona o asíncrona, etc.)
- Soportan o no la creación dinámica de puertos y la exportación de su identificador para su uso por otros procesos.
- Existe o no un entorno de programación gráfico para el lenguaje de coordinación.

A continuación comentamos brevemente algunos de ellos.

PCL [Sommerville96] (*Proteus Configuration Language*) está diseñado para modelar arquitecturas de versiones múltiples de sistemas. Es orientado a objetos y polimórfico. Se ha utilizado para modelar configuraciones tanto estáticas como dinámicas. La coordinación se entiende como una *configuración*, *entidad familiar* que representa una o más versiones de un componente o sistema lógico. Una entidad familiar se puede relacionar con otras a través de la herencia, la composición y su participación en una relación. Una entidad familiar tiene asociados los siguientes datos: un nombre, una clasificación (especificando su tipo), una lista de atributos, información sobre su composición en función de otras entidades y una serie de descriptores de versiones. Los componentes de una configuración encapsulan un estado y proporcionan o requieren servicios a y desde otros componentes. Existen componentes simples y compuestos y sólo los primeros se corresponden con procesos. Los componentes simples pueden clasificarse en activos (proporcionan servicios a los demás pero son capaces de ejecutarse aunque no lleguen estímulos externos) y pasivos. Otro elemento de la configuración son los *puertos*, que permiten la transmisión de mensajes, síncrona o asíncrona, entre componentes.

Conic [Kramer90] es otro lenguaje donde la coordinación se ve como una configuración. Realmente son dos lenguajes: un lenguaje de programación que es una variante de *Pascal* con primitivas de paso de mensajes y un lenguaje de configuración similar a PCL. Un *nodo lógico* es una unidad de configuración del sistema constituido por conjuntos de tareas que se ejecutan concurrentemente dentro de un espacio de direcciones compartido. Los sistemas de configuración se construyen interconectando nodos lógicos. Al conjunto de nodos lógicos interconectados se les llama *grupos*. Otros lenguajes de configuración similares a este son *Darwin*, *Durra* y *Rapide* aunque mejoran este modelo, por ejemplo, permitiendo que las interconexiones entre componentes del sistema puedan modificarse dinámicamente.

Un programa en *ConCoord* [Holzbacher96] es una colección dinámica de procesos de computación y procesos de coordinación. Un proceso de computación ejecuta un algoritmo secuencial y puede escribirse en cualquier lenguaje de programación que posea algunas primitivas de comunicación. Los procesos de coordinación o coordinadores se escriben en CCL (*ConCoord's Coordination Language*). La comunicación se realiza enviando datos a los puertos de salida y recibiéndolos por los puertos de entrada. Los estados son comunicados mediante paso de mensajes. Permite la existencia de jerarquías de coordinadores pero éstas son demasiado rígidas. Utilizan una comunicación síncrona desde el punto de vista de los procesos coordinados. Este lenguaje es similar al lenguaje de coordinación *Manifold*.

Otro modelo de coordinación que tiene bastante éxito últimamente se basa en el concepto de *contratos* [Andrade01] [Andrade02]. Los contratos son primitivas de modelado construidas sobre una semántica formal basada en la *Teoría de Categorías*. Su propósito es proporcionar mecanismos para modelar e implementar una capa de coordinación, independiente de las entidades a coordinar, de forma composicional. De esta forma separan los aspectos propios del sistema de los aspectos de coordinación con el fin de poder modificarlos independientemente. Un contrato de coordinación es, en general, una conexión que se establece entre un grupo de objetos donde las reglas y restricciones se superponen al comportamiento de los participantes. Cuando un objeto se va a comunicar con otro, el contrato intercepta la llamada y permite la comunicación sólo si se cumplen las restricciones asociadas. Los objetos (participantes clientes y servidores) no conocen la clase de coordinación que se utiliza. Los autores utilizan un lenguaje de especificación de contratos llamado *Oblog*. La desventaja que tienen es que los objetos siguen utilizando una comunicación explícita. Otros modelos han adoptado el modelo de contratos para implementar los aspectos de coordinación, como por ejemplo, el modelo arquitectónico PRISMA (PlatafoRma OASIS para Modelos Arquitectónicos [Pérez02]. En este modelo se pone de manifiesto que una de las ventajas de los

contratos es que, al ser independiente de los componentes que los usan, es fácil cambiarlos si el sistema evoluciona.

Coordinated Roles [Murillo99][Murillo01] es un modelo de coordinación de objetos activos implementado en *ActiveJava*, un lenguaje de programación concurrente orientado a objetos basado en Java. Separa la actividad de computación de los objetos de su aspecto de coordinación. Una aplicación se organiza como una serie de componentes funcionales controlados por una jerarquía de componentes *coordinadores* que dirigen la aplicación. Ambas clases de componentes son objetos activos del modelo de objetos con el que se esté programando la aplicación. Cada coordinador implementa una parte del patrón de coordinación y la estructura como un conjunto de *Roles*. Un rol simboliza cada una de las posturas que pueden adoptarse en un patrón de coordinación (por ejemplo, escritor o lector). El coordinador determina, para cada rol, las restricciones que se imponen sobre los componentes funcionales o de coordinación que lo adoptan. Para imponer las restricciones se utiliza el mecanismo de *protocolos de notificación de eventos.* En este modelo existen cuatro clases de eventos:

- *RM* (recepción de un mensaje): A través de la notificación de este evento, un objeto puede conocer cuándo recibe otro un mensaje.
- *BoP* (Inicio de procesamiento): Permite conocer al que solicita cuándo un objeto va a procesar un mensaje.
- *EoP* (Fin de procesamiento): Permite que un objeto sepa cuándo otro objeto ha terminado de procesar un mensaje.
- *SR* (Estado alcanzado): Sirve para que un objeto sepa cuándo ha cambiado otro objeto a cierto estado abstracto.

Las ventajas de este modelo de coordinación frente a otros son: la transparencia (los objetos que están siendo coordinados no tienen que hacer un uso explícito del mecanismo de coordinación), la reutilización y extensibilidad de patrones de coordinación, el conseguir un alto poder de expresividad (se incluyen los cambios de estado internos que ocurren en un objeto sin violar la encapsulación) y el permitir una reconfiguración dinámica.

Ninguno de los modelos y lenguajes de coordinación que hemos comentado se adapta totalmente a nuestras necesidades. Necesitamos un mecanismo que nos permita comunicar de forma implícita a los agentes pero sin que ellos deban conocerse explícitamente. Por ello, podemos pensar que los modelos derivados de *Linda* son una buena elección gracias a la utilización de los espacios de tuplas. Pero, por otro lado, la forma de activación de los agentes es mediante la notificación de eventos al producirse un determinado cambio de estado en el sistema. Para este problema, nos servirían los modelos de coordinación orientados a control. Además, los sistemas basados en agentes se

encuentran dentro de un entorno y pueden recibir de él estímulos que han de ser tratados adecuadamente. Por tanto, necesitamos un modelo híbrido. Por último, lo que la mayoría de los modelos y lenguajes vistos no tratan es el tema de evolución.

4.5 Conclusiones.

A la hora de diseñar un sistema software complejo donde intervengan entidades activas y autónomas, agentes, no nos basta utilizar las abstracciones que nos proporciona la programación orientada a objetos. De hecho, la visión tradicional de un objeto y la visión actual de un agente tienen, al menos, tres diferencias:

- Los agentes incluyen una noción de autonomía más fuerte que la de los objetos, y en particular, deciden por sí mismos si realizan o no una acción solicitada por otro agente.
- Los agentes son capaces de tener un comportamiento flexible (reactivos, pro-activos, sociales) y el modelo estándar de objetos no dice nada acerca de tales tipos de comportamientos.
- Un sistema multi-agente es inherentemente multi-hilo, se asume que cada agente tiene al menos un flujo de control.

Sin embargo, es útil utilizar el paradigma de la programación dirigida a objetos para implementar los agentes y sistemas multi-agentes. Sobre todo, adoptando el concepto de objeto activo.

Un punto importante en cualquier sistema es que, por lo general, no se compone de entidades totalmente independientes sino que éstas necesitan comunicarse y cooperar. Hemos estudiado diferentes mecanismos de comunicación y sincronización y concretamente, los mecanismos usados en los lenguajes orientados a objetos concurrentes.

Por último hemos estudiado diferentes modelos y lenguajes de coordinación que se dividen en dos grandes categorías: orientados a datos y orientados a control.

La mayoría de los modelos y lenguajes de coordinación orientados a datos descienden de *Linda* y tienen las siguientes características:

- Desacoplamiento: la coordinación se realiza a través del espacio de datos compartido (espacio de tuplas). Esto permite que los componentes activos de un sistema no necesiten conocerse y que existan de forma independiente unos de otros (no tienen que estar juntos para poder interactuar). Por tanto, el espacio de tuplas permite abstraer los aspectos de localidad.

- La plantilla utilizada para recuperar una tupla especifica qué clase de tupla se necesita y no permite elegir una en concreto. Existen diferencias entre las primitivas de coordinación proporcionadas por cada lenguaje. Algunas difieren bastante de las presentadas en Linda.
- Las nociones de asincronía y concurrencia son propias del modelo de espacio de tuplas.

Los modelos y lenguajes de coordinación orientados a control se caracterizan porque las entidades coordinadas emiten y reciben eventos a través de puertos de entrada y salida. Los eventos recibidos indican a una entidad la acción a realizar. Los eventos pueden tener o no atributos, la comunicación puede ser síncrona o asíncrona y uno a uno o uno a muchos (*broadcasting*).

A nosotros nos interesa un modelo de coordinación híbrido, que utilice un espacio de datos compartido donde se almacene el estado de un agente pero que, a la vez, proporcione un sistema de notificación de eventos que permita la activación o no de las acciones de un agente. Para nosotros un sistema es una colección de agentes que están relacionados de forma jerárquica. Por tanto, necesitamos almacenar el estado de cada agente en una estructura de datos que será compartida por sus agentes hijos. Es decir, nuestro espacio de datos compartido es un sistema de múltiples espacios de datos compartidos, uno por cada agente existente en el sistema. La accesibilidad a dicho espacio de datos compartido está restringida sólo a los agentes que estén en el mismo nivel jerárquico. Aparte, también necesitamos un elemento que lleve a cabo todo el trabajo de la notificación de eventos y la gestión de un espacio de datos compartido con el fin de aislar los aspectos de coordinación de la funcionalidad propia de los agentes. Este aislamiento facilitará el tratamiento de la evolución de los sistemas software y de sus componentes. Así mismo, también nos permitirá cambiar el propio mecanismo de coordinación y activación de forma trasparente para los agentes de un sistema software.

CAPITULO 5

Definición del Modelo

Antes de definir el modelo, es preciso situar nuestro trabajo dentro del entorno donde empezó y desde donde se han tomado numerosas ideas y conceptos. Nos referimos al modelo MEDES (Método de Especificación, Diseño y Evolución de Software) basado en la Teoría del Sistema General [Le Moigne90] y a su herramienta asociada, HEDES (Herramienta de Especificación, Desarrollo y Evolución de Software), presentada en numerosos trabajos [Parets95] [Anaya96] [Anaya97] [Rodríguez99].

En MEDES se especifica tanto la estructura como el funcionamiento de un sistema software y proporciona los mecanismos necesarios para que éste evolucione. Este modelo se basa en la idea de que el funcionamiento y la estructura de un sistema están íntimamente ligadas (Teoría del Sistema General). Por tanto, cuando un sistema evoluciona, esta evolución provocará un cambio en su estructura, además de en su funcionamiento.

El modelo presentado en esta tesis se basa en definir la arquitectura de un sistema software partiendo de los conceptos introducidos en MEDES y HEDES. Sobre esta base se incorporan aspectos nuevos que nos permiten modelar una amplia variedad de sistemas, como pueden ser los sistemas interactivos, los sistemas de gestión de información, los sistemas de gestión de procesos de producción y flujos de trabajo,

control de aeropuertos, comercio electrónico o recuperación de la información en Internet.

Las entidades activas dentro de nuestro modelo son los agentes y, puesto que no siempre actúan de forma aislada, presentamos un mecanismo que les permita comunicarse, colaborar y coordinarse cuando sea necesario. Un sistema se modela según una estructura jerárquica de agentes que definen su comportamiento, esto simplifica el proceso de construcción.

Se añade un método de activación de agentes que se encarga de notificar los cambios de estado del sistema software a los agentes interesados en dichos cambios. Esta notificación puede provocar que un agente inicie una o más acciones.

Además, en los sistemas pueden existir tareas complejas que se realizan más eficientemente dividiéndolas en subtareas que se puedan ejecutar concurrentemente. Sin embargo, estas tareas deben realizarse completas, como si fueran acciones atómicas. Este concepto se corresponde con el de las transacciones. Por tanto, se incorpora el concepto de transacción y se expone el procedimiento a seguir para asegurarse de que, si la transacción no se lleva a cabo con éxito, su ejecución parcial no afecte al estado del sistema y, por tanto, a su buen funcionamiento.

Contemplamos dentro del modelo la evolución de los agentes, y por tanto, del sistema software garantizando la integridad y consistencia de éste después de los cambios realizados. Es importante tener una arquitectura de agentes dinámica que define un SS con el fin de que dicho SS pueda adaptarse a los cambios que se producen en su entorno y que afectan a los objetivos que se definieron inicialmente y que están totalmente relacionados con su funcionamiento.

En este capítulo seguiremos el siguiente esquema: primero comentaremos los principales conceptos que aporta el modelo MEDES. A continuación definiremos lo que, en nuestro modelo, es un sistema software y después describiremos cada uno de los elementos que van a estar presentes en su estructura. Una vez presentada la estructura, nos centraremos en los problemas de funcionamiento que se presentan entre los agentes que componen un sistema, los identificaremos y presentaremos una solución para ellos. Estos problemas originan la necesidad de definir un modelo de coordinación y comunicación entre agentes que sea eficiente y que permita que el sistema evolucione fácilmente. Finalmente, introducimos el concepto de transacción y los mecanismos necesarios para su gestión.

5.1 El modelo MEDES.

En este punto vamos a realizar una breve descripción del modelo. MEDES nació como resultado de un estudio exhaustivo de la problemática relacionada con la modelización del software y de su evolución en el proceso de elaboración del software [Parets95]. Su objetivo era proporcionar un método que permitiera la concepción y evolución de sistemas software integrando el proceso de desarrollo en el propio método. El desarrollo de MEDES se complementa mediante la implementación de una herramienta, HEDES [Parets99a][Rodríguez99], que permite llevar a la práctica los conceptos vistos en MEDES. Tanto el modelo como su herramienta asociada son, posteriormente, ampliados y completados en [Rodríguez00a], donde se estudia y formaliza el subsistema de decisión, parte muy importante en el proceso de evolución de un sistema.

En MEDES todo sistema software tiene un funcionamiento y una estructura y ambos están relacionados entre sí. El funcionamiento de un sistema viene determinado por su estructura, por tanto cualquier cambio en su estructura produce un cambio en su funcionamiento. El equipo de desarrollo puede cambiar un sistema mediante un sistema software especial denominado Metasistema.

Dentro de un sistema software se pueden distinguir dos partes, una *funcional* y otra *estructural* cuya composición es similar. La parte estructural es importante en el proceso de evolución del sistema. Cada parte puede tener los siguientes elementos:

- *Procesadores*: son las entidades activas que realizan acciones para el sistema. Los procesadores de la parte funcional realizan acciones de funcionamiento y los de la parte estructural realizan acciones de evolución. Un sistema está compuesto por un conjunto de procesadores.

- *Acciones*: actividades realizadas por los procesadores. Podemos distinguir dos tipos de acciones: acciones funcionales y acciones estructurales o de evolución. El conjunto de las acciones funcionales determinan el comportamiento del sistema. Las acciones estructurales son necesarias para cambiar la estructura de un sistema, es decir, nos permiten hacer evolucionar a un sistema. Además, las acciones tienen asociadas unas condiciones que determinan cuándo se pueden realizar y cuándo no. Existen tres tipos de condiciones: pre-condiciones (deben cumplirse antes de poder realizarse la acción), post-condiciones (deben cumplirse después de realizarse la acción) y durante-condiciones (deben ser ciertas durante la ejecución de la acción).

- *Historia*: tenemos dos historias. La historia funcional del sistema (SFH-*System Functional History*) almacena la información de lo que ha ocurrido hasta el instante actual en el sistema (las acciones funcionales realizadas en él). La historia estructural del sistema (SSH-*System Structural History*) almacena la información sobre los cambios realizados en el funcionamiento del sistema (cambios en su estructura, por ejemplo, eliminar un procesador) hasta el instante actual.

- *Eventos*: constancias de la realización de las acciones de los procesadores. Son los elementos almacenados en las historias y contienen la información necesaria por el sistema de la realización de una determinada acción.

- *Interfaz*: permite al sistema comunicarse con el exterior, con su entorno.

- *Subsistema de Decisión (SD)*: se encarga de evaluar las condiciones asociadas a las acciones para decidir qué acciones pueden realizarse. Para ello, consulta la información almacenada en la historia adecuada.

En la siguiente figura se puede ver la estructura básica de un sistema software con los componentes que hemos comentado anteriormente:

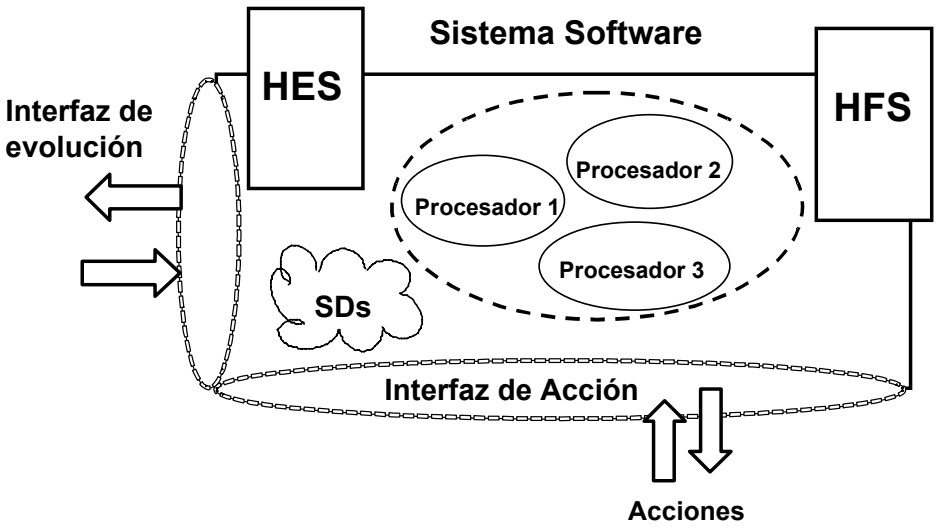


Figura 5.1 Estructura básica de un sistema software en MEDES

Cuando el equipo de desarrollo comienza a modelar un determinado sistema software sigue un ciclo de vida iterativo (por prototipos) [Sommerville01], ya que se concibe la evolución como un proceso de maduración en el que se van pasando por diferentes estadíos en el

modelo de un sistema software. Se van determinando los procesadores, las acciones, etc. y se comprueba cómo funciona dicho sistema. Si todavía no se comporta como se espera, se pueden ir realizando cambios sobre él e ir valorando los resultados. HEDES es la herramienta que nos permite realizar este proceso de desarrollo iterativo. Se puede decir que es una herramienta CASE que nos ayuda a realizar esta evolución y obtener, al final, un producto (sistema software) lo bastante maduro como para poder utilizarlo.

Este proceso se representa en la siguiente figura. Como se puede observar se comienza con una etapa inicial de recogida de requisitos y se entra en un proceso iterativo compuesto de tres etapas:

- Una etapa de selección funcional donde se identifican los elementos dentro del sistema software que se quiere construir.

- Una etapa de construcción de un prototipo que se utilizará en la siguiente etapa para realizar comprobaciones sobre la construcción del sistema software.

- Una etapa de uso y evaluación del prototipo donde el usuario o usuarios prueban el prototipo y detectan las posibles deficiencias.

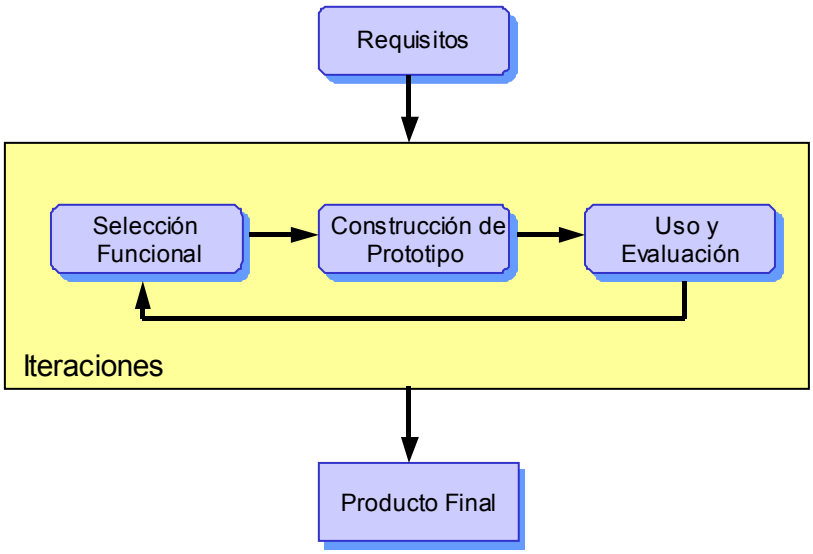


Figura 5.2 Ciclo de vida del software por prototipos

Cuando finalmente se llega a la conclusión de que el sistema software desarrollado cumple todos los requisitos para los cuales es modelado, entonces se genera el producto final.

5.2 Definición de Sistema Software.

En el modelo que presentamos en este trabajo [Paderewski02b] [Paderewski03a], un sistema software (SS) es una colección de agentes, entidades activas que son responsables de la realización de una o más acciones. Estos agentes pueden trabajar de forma individual e independiente o pueden cooperar con otros agentes para realizar una tarea compleja. Los agentes trabajan de forma simultánea, una característica innata de casi cualquier sistema que queramos modelar. Por ejemplo, supongamos un sistema de alquiler de coches. En él podremos tener distintos agentes, unos encargados de realizar los alquileres de coches y otros, por ejemplo, llevarán la contabilidad de la empresa o cualquier otra tarea burocrática. Todos estos agentes trabajan para que el sistema funcione y todos lo hacen de forma simultánea.

En nuestro modelo adoptamos la definición de agente software presentada por numerosos autores como [Franklin96] y [Jennings96]. Un agente se caracteriza porque es *autónomo* (actúa de forma independiente), *social* (se comunica con el usuario del sistema o el resto de agentes si es necesario), *reactivo* (actúa en respuesta a los cambios producidos en su entorno) y *pro-activo* (no sólo actúa en respuesta a su entorno, sino que tiene iniciativa y decisión propia). Inicialmente nos da igual cómo se implemente un agente, siempre y cuando posea las características mencionadas. A nivel de implementación puede ser una hebra, un proceso o un objeto activo.

Necesitamos mantener una historia de las acciones realizadas en el SS hasta el momento actual y de los estímulos recibidos tanto desde el exterior (equipo de desarrollo) como de los propios agentes porque ésta determina el estado actual del SS y nos interesa establecer condiciones en base a estados anteriores del sistema. Un agente podrá realizar una acción (o no) si el SS está en un determinado estado (es decir, si se han realizado o no ciertas acciones en él). Además, esta historia nos servirá como medio de comunicación entre los agentes que lo necesiten, entre los agentes cooperantes.

Para mantener esta historia en el sistema se utiliza una solución basada en la arquitectura *blackboard* [Shaw96]. Se mantiene una historia, llamada historia funcional del sistema (SFH-*System Functional History*), donde se almacenan todas las ocurrencias de las acciones realizadas en el sistema hasta el instante actual y los estímulos recibidos. El uso de esta historia permitirá que varios agentes se comuniquen sin que ninguno de ellos deba conocer la identidad del otro ni tenga que solicitarle explícitamente la información que necesita. Toda la información necesaria se mantiene en la historia y los agentes leen y

escriben en dicha historia, como se puede apreciar en la figura siguiente. Además, con esta arquitectura se almacena el estado del sistema en la historia y ésta actúa como disparador para que se sigan realizando (o no) acciones en el sistema.

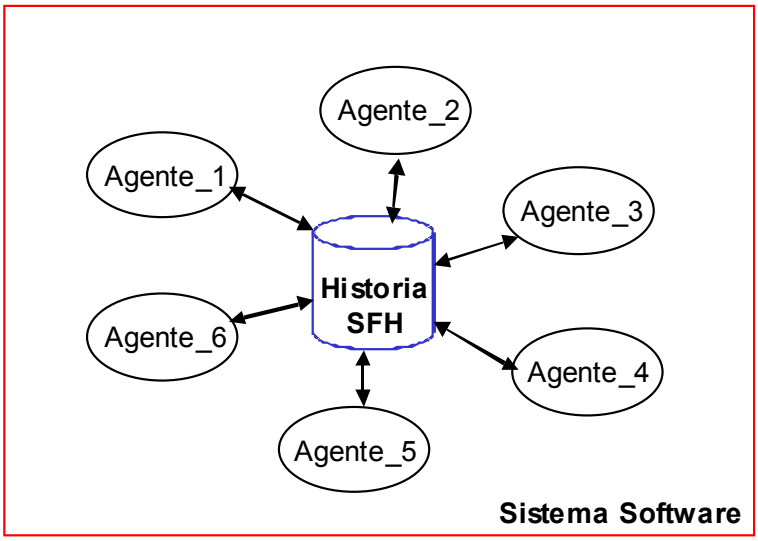


Figura 5.3 Sistema Software

Por otro lado y debido a que, normalmente, un sistema software evoluciona a lo largo de su vida, es decir, cambia, también nos interesa guardar una historia de los cambios estructurales que ha sufrido desde que es concebido hasta el instante actual. Esta información nos servirá para poder decidir si una determinada modificación de la estructura del sistema se debe llevar a cabo o no. Es decir, si una determinada modificación del sistema lo puede llevar a un estado no íntegro, inconsistente, no se permitirá dicha modificación. Además, puede que se intenten realizar cambios sobre la estructura que no se puedan llevar a cabo porque, por ejemplo, no existe el agente que deseamos eliminar (nunca se ha creado). También es factible, usando esta historia, reproducir un estado anterior del sistema, ya que cualquier cambio que haya llevado al sistema a su estado actual está recogido y podría "deshacerse". A esta nueva historia la denominamos historia estructural del sistema (SSH-*System Structural History*). Seguimos utilizando, para ella, una arquitectura *blackboard*, al igual que hicimos para la historia funcional. Aunque su contenido sea distinto, el tratamiento es análogo.

El sistema software se comunica con el "exterior", con su entorno, mediante *una interfaz de acción* y una *interfaz de evolución*. El entorno lo constituye el o los usuarios del SS y/o el equipo de desarrollo que se encarga de desarrollar el SS. A través de la interfaz de acción, el SS recibe los estímulos necesarios para que inicie una o más acciones. A través de la interfaz de evolución, recibe los estímulos que hacen que el SS cambie, evolucione.

En la figura siguiente se representa la estructura básica de un sistema software mostrando los distintos elementos que lo componen y que definiremos en el siguiente punto. Como hemos dicho anteriormente, partimos de la definición de SS dada en MEDES. Estos conceptos son la base de nuestro modelo.

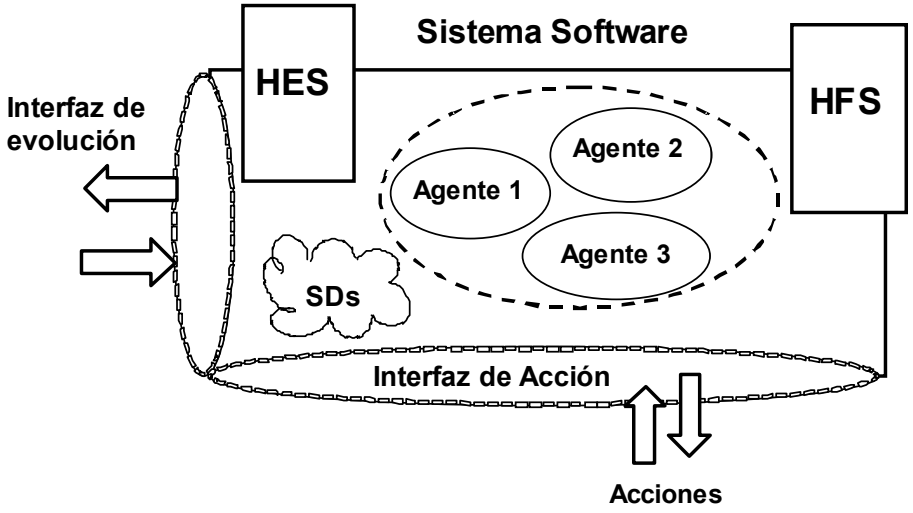


Figura 5.4 Sistema Software básico y su relación con el exterior

5.3 Elementos constitutivos de un Sistema Software.

A continuación describiremos detalladamente cada uno de los elementos que constituyen la estructura de un sistema software.

5.3.1 Los Agentes.

Los *agentes* trabajan para el sistema realizando acciones. Son los responsables del funcionamiento del sistema. Los agentes intentan continuamente realizar las acciones que definen su comportamiento en base al estado actual del sistema software (a su historia) en el que se encuentran. Cada agente puede ser, a su vez, un sistema software. Esto nos permite tener *anidamiento de sistemas o agentes*, cada uno con sus propias responsabilidades, y crear SS más complejos a partir de otros más simples. Además, como son independientes, un mismo diseño de un agente se puede utilizar en mas de un SS sin necesidad de tener que realizar cambios en su estructura. Es decir, podemos reutilizar un SS completo o algunos de sus componentes para construir y diseñar nuevos sistemas software con un menor coste que si se partiera de cero.

Un sistema software y un agente tienen idéntica estructura. La única diferencia entre ellos es que llamamos *sistema* al agente "raíz" de la estructura de agentes que define, a alto nivel, al sistema software que estamos modelando. El agente sistema presenta en su interfaz las acciones que puede realizar y que son visibles desde el exterior, desde su entorno. Por tanto, a partir de este punto, utilizaremos siempre al término agente y sólo cuando queramos destacar que citamos al sistema completo, usaremos el término sistema o SS.

5.3.1.1 Relaciones entre agentes: agregación y colaboración.

El anidamiento de agentes es importante porque permite modelar dos clases de relaciones entre agentes: la relación de agregación y la relación de colaboración.

La **relación de agregación** (padre-hijo) permite modelar agentes complejos a partir de agentes más simples. Un agente puede cambiar desde el punto de vista de la funcionalidad que proporciona. Este cambio puede provocar que el agente crezca, y este crecimiento debe poder realizarse sin tener que modificar el agente entero. Con una simple agregación de uno o más agentes podemos extender fácilmente la funcionalidad de un agente. Además, en muchos casos es necesario representar que existe una comunicación entre padre e hijo, por ejemplo, para que el hijo indique al padre que ha finalizado una determinada acción.

Usando esta relación podemos ver un SS como un árbol jerárquico de agentes. En la raíz de dicho árbol se encuentra el nodo que hemos llamado sistema y en las hojas tenemos los agentes que realizan acciones simples. Hablaremos más sobre esto cuando expongamos la definición de acción y los tipos de acciones que pueden realizar los agentes, ya que existe una estrecha relación entre la jerarquía de agentes y las acciones complejas definidas en ellos.

La **relación de colaboración** es necesaria para poder establecer una sincronización y una comunicación entre distintos agentes (agentes hermanos) con el fin de que realicen una tarea más compleja. Esta relación se establece entre agentes hermanos ya que comparten una historia funcional, SFH, a través de la cual se pueden comunicar para colaborar en la realización de una determinada tarea. Es decir, se puede establecer una cierta colaboración entre distintos agentes fácilmente y sin que haya que definirla desde la generación del sistema, sino en cualquier momento de su funcionamiento. Esto se consigue gracias a la estructura que tienen los agentes y a la forma en la que se realiza su evolución.

En la siguiente figura se representan ambas clases de relaciones. La relación de agregación está representada por el *Agente 1*. Este agente tiene dos agentes hijos, el *Agente 1.1* y el *Agente 1.2*. La relación entre *Agente 1* y sus hijos es de agregación. Sin embargo, la relación entre los agentes hermanos, *Agente 1.1* y *Agente 1.2*, es de colaboración.

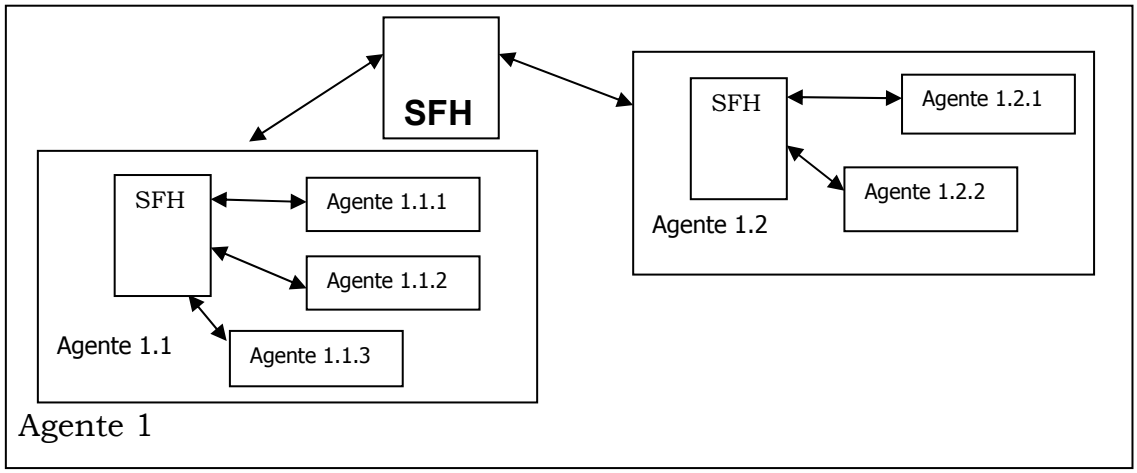


Figura 5.5 Relaciones entre Agentes

Obsérvese que cada agente tiene su propia historia y que la historia que comparten los agentes hermanos es la de su padre. Esto representa una historia del sistema que está distribuida, es decir, si juntamos todas las historias de todos los agentes, tendríamos la historia completa del sistema. El hecho de distribuir la historia nos proporciona dos grandes ventajas:

- Se disminuyen los problemas de eficiencia producidos por la existencia de cuellos de botella en el acceso a una historia única. Cada vez que se produzca una acción, este hecho se registrará en la historia del agente padre del agente que ha realizado dicha acción. En el ejemplo anterior, si *Agente 1.1* realiza una acción, éste hecho es almacenado en la SFH del *Agente 1*, pero cualquier ocurrencia de acción realizada por el *Agente 1.1.1* se recogería en la SFH del *Agente 1.1*.

- Permite que cada agente pueda estar físicamente en un lugar distinto dentro de un Sistema Distribuido. Es decir, este modelo permite definir SS que se ejecuten sobre una arquitectura distribuida. Cada agente puede estar ejecutándose sobre un nodo de una red. La comunicación con su padre o con sus hijos se puede hacer mediante el mecanismo de paso de mensajes.

El concepto de historia distribuida está muy relacionado con el de *multiespacios de tuplas*, que pueden o no estar anidados, utilizados en

los lenguajes y modelos de coordinación derivados de *Linda* [Gelernter92], como son *WCL*, *KLAIM*, *Jada*, *T Spaces*, etc. [Omici01].

5.3.1.2 Concurrencia inter-agente e intra-agente.

En nuestro modelo contemplamos las siguientes dos formas de ejecución concurrente relativas a los agentes que constituyen un sistema:

- *Concurrencia inter-agente*: nos referimos a la situación en la que dos acciones de distintos agentes pueden estar realizándose simultáneamente dentro del sistema, es decir, su ejecución se solapa en el tiempo.

- *Concurrencia intra-agente*: como un agente puede realizar más de una acción, este tipo de concurrencia se refiere a la realización simultánea de dos acciones del mismo agente o incluso de dos ejecuciones distintas de la misma acción. Permitir este tipo de concurrencia interna aumenta el poder de expresividad así como la concurrencia global del sistema. Nos basamos en la definición presente en distintos modelos de computación donde se denomina concurrencia *intra-objeto* a una entidad activa a la que se le permite procesar varias respuestas simultáneamente, por lo que puede realizar varias actividades concurrentemente [Briot98].

Cuando se construye un sistema basándose en nuestro modelo, se supone que, por defecto, las acciones de un agente se realizan o pueden realizarse concurrentemente con cualquier otra acción que se active en dicho sistema. Por tanto, el especificar que una acción debe llevarse a cabo después de finalizar otra u otras (es decir, las restricciones de la secuencialidad de acciones) se consigue mediante el sistema de pre, durante y post-condiciones que hemos comentado y que seguiremos tratando en este punto. Además, si se trata de las acciones que intervienen en una transacción o acción compleja, el orden adecuado lo especificamos usando un *Lenguaje de Definición de Transaciones* (TDL).

5.3.2 Las acciones, las ocurrencias de acciones, los estímulos y las condiciones de las acciones.

A continuación vamos a describir cada uno de estos elementos. Todos ellos están relacionados con las actividades que se realizan dentro del agente y con el proceso de activación de las mismas.

5.3.2.1 Las acciones.

Los agentes realizan acciones y se permite que existan varios agentes que realicen la misma acción. El conjunto de las acciones que se pueden llevar a cabo en el contexto de un agente determina su funcionalidad. Como hemos mencionando anteriormente, para mantener la información de lo que ha pasado en el agente hasta el instante actual mantenemos una historia denominada *Historia Funcional del Sistema (SFH).* En ella se guardan las *ocurrencias de acciones* realizadas hasta el instante actual y los estímulos recibidos.

Podemos distinguir dos clases de acciones que puede realizar un agente:

- *Acciones simples*: son acciones realizadas por un único agente.

- *Acciones complejas o transacciones*: conjunto de acciones (simples o complejas) que son realizadas en un orden determinado y por un conjunto de agentes para llevar a cabo una tarea compleja. Los agentes involucrados deben coordinarse y cooperar. Hablaremos de esta clase de acciones más detalladamente en un punto específico, cuando introduzcamos al resto de elementos del modelo.

5.3.2.2. Las ocurrencias de acciones.

Una *ocurrencia de acción* puede definirse como la constancia de la realización de una acción en el SS. Toda acción conoce la ocurrencia de acción que debe generar cuando se realice y la genera con toda la información necesaria (atributos de la acción que tomarán valor en la ocurrencia) para el SS. Cada ocurrencia de acción posee, al menos, un atributo que será el tiempo de realización de la acción. Por ejemplo, en un sistema de alquiler de coches, podemos tener la acción *alquilar*. Cuando se ejecute, se producirá una ocurrencia de dicha acción que guarda cierta información, como es el tiempo de realización, los datos del coche alquilado: persona o entidad a quien se le alquila, número de días, etc.

Normalmente, y por convención, las ocurrencias de acción tendrán el mismo nombre que la acción que la genera.

5.3.2.3 Los estímulos.

Podemos definir un *estímulo* como una señal procedente del entorno (usuario o desarrollador) o de otro agente y que pretende desencadenar la realización de una o más acciones. Al igual que las acciones, pueden

tener atributos que tomarán valor cuando sean enviados y tendrán un atributo que indicará el instante de tiempo en el que fue recibido dicho estímulo.

Necesitamos los estímulos porque los usuarios de un SS pueden necesitar activar una determinada acción. Por ejemplo, continuando con el sistema de alquiler de coches, la llegada de un cliente puede ser un estímulo para que se active el agente encargado de atenderle. Cuando se recibe un estímulo, éste se almacena, igual que ocurre con las ocurrencias de acciones, en la SFH. Se hace así porque el que se produzca el estímulo no significa que se pueda realizar una determinada acción, depende del estado actual del sistema, como veremos a continuación. El concepto de estímulo equivale al concepto de evento utilizado en otros modelos.

5.3.2.4 Las condiciones de las acciones.

Cada acción tiene asociada una serie de condiciones. En nuestro modelo, estas condiciones son de tres clases: *pre-condiciones, post-condiciones* y *durante-condiciones*. En cada agente tendremos un *Sistema de Decisión (SD)* que contendrá la lógica necesaria para poder tomar decisiones sobre si un agente puede o no realizar una acción (activación de una acción) según se verifiquen o no sus pre-condiciones, así como interrumpirla mientras que se realiza (durante-condición) o anularla una vez realizada si no se verifica su post-condición. Por ejemplo, para la acción *alquilar*, la pre-condición puede ser que exista un coche libre o, lo que es lo mismo, que en el instante actual no esté alquilado y esté disponible. Si la acción es *devolver*, y mientras se está revisando el buen estado del coche se llega a la conclusión de que no debe ser retirado (durante-condición), se interrumpe la acción devolver.

Las condiciones están basadas en las acciones que el agente haya realizado previamente y en los estímulos recibidos, es decir, el funcionamiento de un agente viene determinado por su estado en cada momento. Por ello, necesitamos conocer lo que ha ocurrido en él para comprobar si una acción se puede o no realizar en un determinado instante de tiempo. Por tanto, verificar si una condición se satisface equivale a ver si se ha realizado antes una o más acciones concretas (o no) y/o se ha recibido un estímulo determinado. Esta verificación se realiza fácilmente consultando el contenido de la historia del agente. Por ejemplo, para la acción *alquilar*, el estímulo que representa el hecho de que llegue un cliente que quiere alquilar un coche podría ser parte de su pre-condición. Si este estímulo no existe en la historia del agente padre donde se encuentra el agente encargado de realizarla, no tiene sentido que se intente llevar a cabo la acción.

La construcción de estas condiciones se realiza utilizando las ocurrencias de acciones y/o los estímulos involucrados más un conjunto de operadores que describan correctamente lo que debe determinar la activación, interrupción o invalidación de cierta acción de un agente. Para la definición de las condiciones se utiliza un subconjunto de la Lógica Temporal de Predicados de primer orden (LTP) [Gabbay95] [Rodríguez00a]. Esta lógica la utilizamos porque nos interesa conocer el tiempo en el que se realizan la acciones y establecer condiciones en base a estados anteriores del sistema [Anaya96] [Anaya97]. En el apéndice I se puede ver la descripción del lenguaje utilizado.

Cuando se construye un sistema software, el equipo de desarrollo será el encargado de crear los agentes necesarios, establecer sus relaciones, crear las acciones que van a realizar y definir las distintas condiciones (pre, durante y post-condiciones) que determinarán en cada momento el funcionamiento del sistema. La mayoría de las acciones de funcionamiento de un agente sólo tienen pre-condiciones, ya que la post-condición de una acción será la pre-condición de otra. Sin embargo, si nos referimos a acciones que modifican la estructura del agente, éstas si poseen su post-condición ya que se tiene especial cuidado en que un agente no quede en un estado inconsistente después del cambio efectuado. Además, sólo dentro de una acción compleja o transacción se contemplan las durante-condiciones, porque si la acción es simple se supone instantánea, es decir, no tenemos ningún control sobre ella. Por tanto, cuando hablamos del funcionamiento de un sistema, sólo contemplamos las pre y durante-condiciones. Cuando nos centramos en el aspecto evolutivo, nos centramos en las pre y post-condiciones y no en las durante-condiciones debido a que son acciones predefinidas y todas son simples.

5.3.3 Las historias de un Agente.

Como dijimos en el punto anterior, en todo instante de tiempo debemos saber lo que ha ocurrido en un agente hasta el momento actual. Esta información se guarda en su SFH y se consultará para saber cuándo se debe activar una acción de un agente hijo y cuándo no. Debido a su contenido y a que es importante conocer el orden en el que se han realizado las acciones o se han recibido los estímulos, es interesante mantener algún dato que nos indique dicha información. Para ello, almacenamos como un atributo de cada ocurrencia de acción o estímulo el instante de realización, al cual llamamos *tiempo de realización*, y utilizamos esta información para mantener un orden dentro de la SFH. El mantener la SFH ordenada por tiempo de realización facilita la búsqueda que luego se haga de dicha información y la labor de evaluación de las condiciones como veremos más adelante.

Se permite que existan dos o más ocurrencias y/o estímulos con el mismo orden temporal. A las acciones que se han realizado concurrentemente, se les asigna el mismo tiempo de realización. Esto no representa un inconveniente aunque hay que tenerlo en cuenta en el proceso de evaluación de las condiciones asociadas a las acciones.

Nos interesa saber el orden temporal de la realización de las acciones en el agente porque para evaluar una pre-condición, miramos las ocurrencias de las acciones y el orden temporal de éstas puede hacer que la evaluación sea cierta o no (no nos basta con la existencia de una o más ocurrencias en la historia). Por ejemplo, no sólo necesitamos saber que se devolvió un coche sino que después de dicha devolución no se ha vuelto a alquilar. Recordemos que en la historia se encuentran todas las ocurrencias de acciones y que es muy probable que una acción se realice más de una vez en el agente.

Además de esta historia, y ya que los agentes evolucionan, necesitamos también la información de cualquier cambio que haya sufrido el agente desde que se creó. Estos cambios se almacenan en la *Historia Estructural del Sistema* (SSH – *System Structural History)*. Por ejemplo, añadir un nuevo agente sería una acción estructural cuya ocurrencia se almacenaría en la historia estructural del agente padre. Igual que en la SFH, la información sobre las acciones estructurales y los estímulos de las acciones estructurales se almacenan según el orden temporal establecido por los tiempos de realización de cada uno de ellos.

Las acciones que hacen evolucionar un sistema se denominan *acciones estructurales*, y sus ocurrencias asociadas, *ocurrencias* (de acciones) *estructurales*. Por analogía, las acciones que realizan los agentes que hacen funcionar al SS las llamamos *acciones funcionales* y las ocurrencias asociadas, *ocurrencias funcionales*.

Existe una diferencia en cuanto al comportamiento de la parte estructural y la parte funcional de un agente relacionada con las distintas historias. Cuando un agente realiza una acción funcional, la ocurrencia de dicha acción se almacena en la SFH de su agente padre. Sin embargo, cuando un agente realiza una acción estructural, su ocurrencia se almacena en su propia y privada SSH. La razón es simple, la SFH sirve como medio de comunicación y coordinación entre agentes hermanos, por tanto, las acciones realizadas por éstos deben ser conocidas por sus agentes hermanos, por tanto, las ocurrencias funcionales se almacenan en una estructura compartida por todos, la SFH de su agente padre. Por otro lado, la SSH almacena información de las modificaciones estructurales de un agente, por tanto, sólo son necesarias para éste ya que esta representa su evolución. A los agentes

hermanos no les importa si a uno de ellos se le ha añadido una nueva acción, cambiando así su funcionalidad.

5.3.4 Partes estructural y funcional de un agente.

Todo lo dicho hasta ahora respecto a la historia y a las acciones se centró en el aspecto funcional de un agente pero nos sirve como referencia para tratar de forma análoga el aspecto relacionado con la estructura de un agente y su evolución.

Podemos observar en la figura que un agente se compone de dos partes bien diferenciadas: la *parte funcional* y la *parte estructural*. Ambas partes son isomorfas, los elementos que las constituyen son los mismos, una serie de agentes y una historia. Cada parte mantiene una historia de lo ocurrido hasta el momento actual pero sobre distintos aspectos, una sobre el funcional y la otra sobre el estructural.

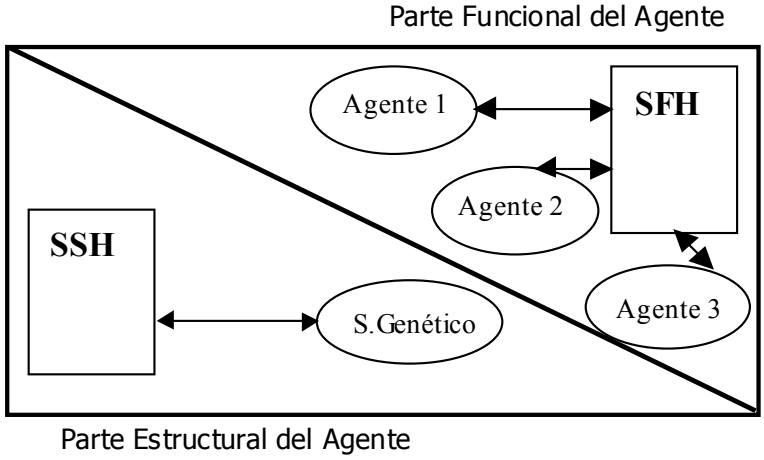


Figura 5.6 Partes de un agente: Funcional y Estructural

La parte funcional recoge el estado actual y pasado del agente. Las acciones realizadas por sus agentes hijos son las actividades que definen su comportamiento.

La parte estructural almacena la información de los sucesivos cambios estructurales realizados sobre el agente. Esta información nos sirve tanto para verificar si un determinado cambio estructural se puede o no realizar en un momento dado como para poder reconstruir el sistema con un comportamiento y estructura anterior al instante actual. Los cambios estructurales, como se comentó anteriormente, afectan al funcionamiento del agente. Si se elimina un agente hijo, es posible que no se pueda realizar alguna acción concreta.

5.3.5 La relación de cooperación entre agentes.

Al diseñar un sistema, si éste es complejo, es más fácil utilizar un diseño descendente de las distintas funciones que lo componen. Primero se intenta identificar las distintas funciones que se pueden realizar de forma independiente, a un nivel alto de abstracción. Después, repetimos el proceso con cada una de estas funciones hasta llegar a definir funciones básicas, fáciles de implementar y probar. Una vez que tenemos el conjunto de funciones básicas, podemos crear otras más complejas y así podemos ir construyendo la jerarquía de funciones que, finalmente, componen el sistema software que se desea. Cada una de las funciones definidas pueden llevarse a cabo por un agente determinado, por tanto, la jerarquía subyacente en las funciones implica una jerarquía en los agentes.

Es probable que no exista una independencia total dentro de cada nivel de la jerarquía que hemos comentado. Puede que distintos agentes hermanos tengan que cooperar para que, uniendo sus esfuerzos, lleguen a realizar una tarea más compleja que es lo que se pretende conseguir en el agente padre donde se encuentran. La relación de cooperación es, por tanto, muy importante, y además debe poderse establecer la relación entre dos o más agentes incluso durante el funcionamiento del sistema, ya que éste puede evolucionar y cambiar su comportamiento.

La coordinación entre los agentes hermanos que cooperan en un sistema se consigue a través de la información que se mantiene en la historia de su agente padre y las condiciones asociadas a las acciones. Si fuera necesario, los agentes podrían comunicarse explícitamente pero la forma usual de hacerlo que van a tener en este modelo es a través de la historia (implícitamente) y, por defecto, asíncronamente. Con esto conseguimos que cada agente actúe independientemente, en el sentido de que no tiene que conocer la existencia de los otros agentes y los agentes son menos dependientes unos de otros. Mientras la ocurrencia de una acción (u ocurrencias) y/o el estímulo (o estímulos) que necesita un agente para realizar una acción haya ocurrido, no importa quién realice dicha acción o si el agente que la realiza ha cambiado o ha sido sustituido por otro. Esto nos proporciona una buena base para conseguir que los cambios que sufre un agente debido a su evolución sólo afecten a los agentes implicados y no al resto de agentes que componen el sistema software.

Según la estructura expuesta anteriormente, cada agente tiene una SFH. En ésta se almacenan las ocurrencias de las acciones “internas” realizadas por los agentes que lo componen. Las ocurrencias de las acciones que tiene un agente y que presenta en su interfaz se

almacenan en la SFH de su agente padre que es donde se encuentra definido (ver figura 5.5).

El árbol jerárquico que constituye un sistema software tiene al menos dos niveles, el agente "raíz" que es el agente sistema y un conjunto de agentes que realizan acciones simples. A estos agentes que constituyen las hojas del árbol los llamaremos *agentes simples* porque sólo realizan acciones simples y, aunque por definición tienen su propia historia funcional, ésta se encuentra vacía. Las ocurrencias de las acciones simples que realizan se almacenan en la historia de su agente padre.

El interfaz del agente "raíz" esta constituido por las acciones de alto nivel que se realizan en él. Realmente es lo que un usuario ve desde el exterior de las funciones que realiza el SS. A estas acciones no se les asociará unas pre-condiciones válidas (por defecto son *true*) ya que no existe un agente padre y una historia donde comprobarlas. Sin embargo, si en cualquier momento agregamos este sistema a otro, asociaríamos los valores adecuados a las pre-condiciones para que comience a trabajar integrado en el nuevo entorno.

5.3.6 La interfaz de acción, la interfaz de evolución y el Metasistema.

La *interfaz de acción* permite comunicar al sistema software con el sistema de información en el que está incluido. A través de esta interfaz, el sistema software recibe estímulos que indican que se deben realizar determinadas acciones. Los estímulos suelen formar parte de la pre-condición de una acción y se almacenan en la SFH cuando son recibidos en el sistema software. Normalmente son los usuarios del sistema software los que introducen estímulos a través de esta interfaz, aunque existen estímulos que son producidos por agentes dentro del sistema software. Un agente puede enviar un estímulo a otro agente que lo recibirá a través de su interfaz de acción. Por ejemplo, un usuario puede generar un estímulo al pulsar un botón del ratón cuando el cursor está posicionado sobre una determinada ventana. Esto puede provocar que se genere un segundo estímulo originado por el gestor de la ventana indicando que el cursor está dentro de dicha ventana.

La *interfaz de evolución* es más compleja y permite que el equipo de desarrollo haga evolucionar estructuralmente al sistema por medio de un sistema software especial llamado *Metasistema*. El Metasistema realiza acciones para cambiar la estructura del sistema/s software del que es responsable.

Las acciones funcionales del Metasistema son acciones genéricas del tipo: añadir agente, eliminar agente, añadir acción, etc. Conociendo la

estructura de un sistema en cuanto a la clase de elementos que lo componen, es fácil definir la lista de acciones funcionales del Metasistema. Las acciones que hacen "funcionar" al Metasistema hacen "evolucionar" al sistema software. El Metasistema es un SS más y tiene las mismas características que cualquier SS. Posee por tanto agentes, una historia funcional (*MFH-Metasystem Functional History*) y una historia estructural (*MSH-Metasystem Structural History).* Su historia funcional se corresponde con la historia estructural del sistema software con el que posee una interfaz. Nótese que una acción funcional del Metasistema corresponderá a una acción estructural en el sistema que cambia. Su historia estructural es estática y no cambia nunca, contiene la definición de las acciones estructurales del Metasistema y todas las pre-condiciones de las acciones estructurales necesarias para que funcione, creando y haciendo evolucionar al sistema.

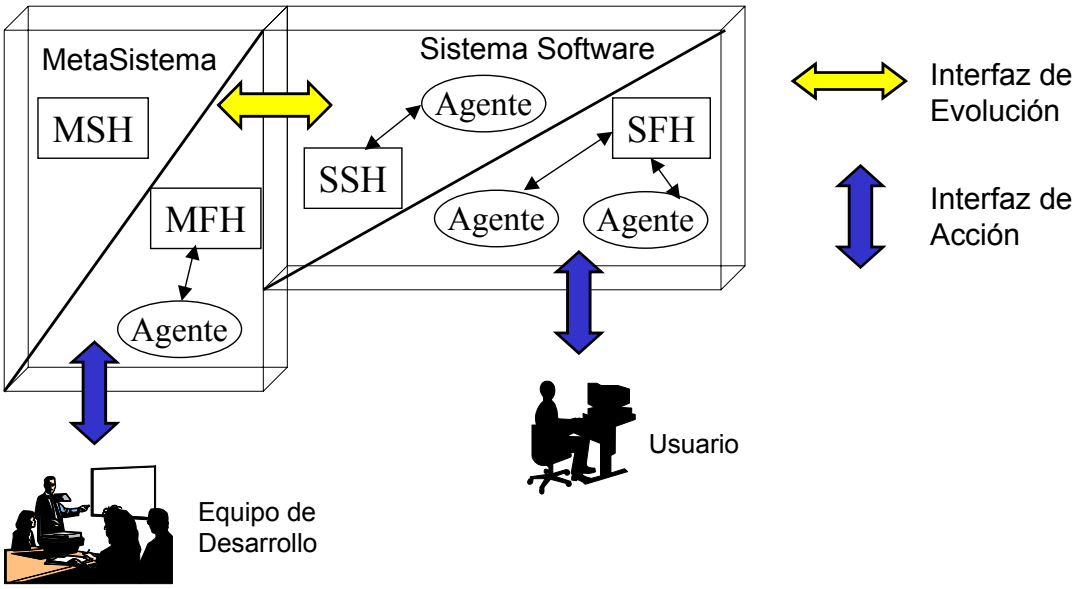


Figura 5.7 Interfaz de acción y de evolución

En la figura anterior podemos observar que existen dos clases de usuarios, los usuarios que utilizarán el sistema software y que se comunican con él a través de la interfaz de acción del sistema software y el equipo de desarrollo. El equipo de desarrollo es el encargado de trabajar con el Metasistema y comunicarle a través de la interfaz de acción de éste los cambios estructurales que se quieren realizar sobre el sistema software. Esta comunicación se realiza mediante estímulos, es decir, cada acción estructural definida en el Metasistema tiene un estímulo asociado que, una vez introducido, activará el funcionamiento del Metasistema.

Cuando llega un estímulo asociado a una acción estructural, el Metasistema comprueba si ésta se puede realizar en el sistema software. La comprobación se realiza verificando la pre-condición de la

acción estructural consultando, para ello, las historias pertinentes (SSH y SFH).

5.3.7 Un agente especial, el Sistema Genético.

En la figura siguiente se puede observar cómo es la relación entre un Metasistema y el sistema software que hace evolucionar. La flecha que los une representa la comunicación entre ambos a través de la interfaz de evolución. Es el Metasistema quién comprueba si una determinada acción estructural se puede realizar pero es un agente especial, llamado *Sistema Genético* (*GS - Genetic System*), quien realmente la lleva a cabo en el agente donde se encuentra. Cada agente tiene agregado un Sistema Genético en su parte estructural. Las acciones que tiene que realizar le llegan a través de la interfaz de evolución por parte del Metasistema.

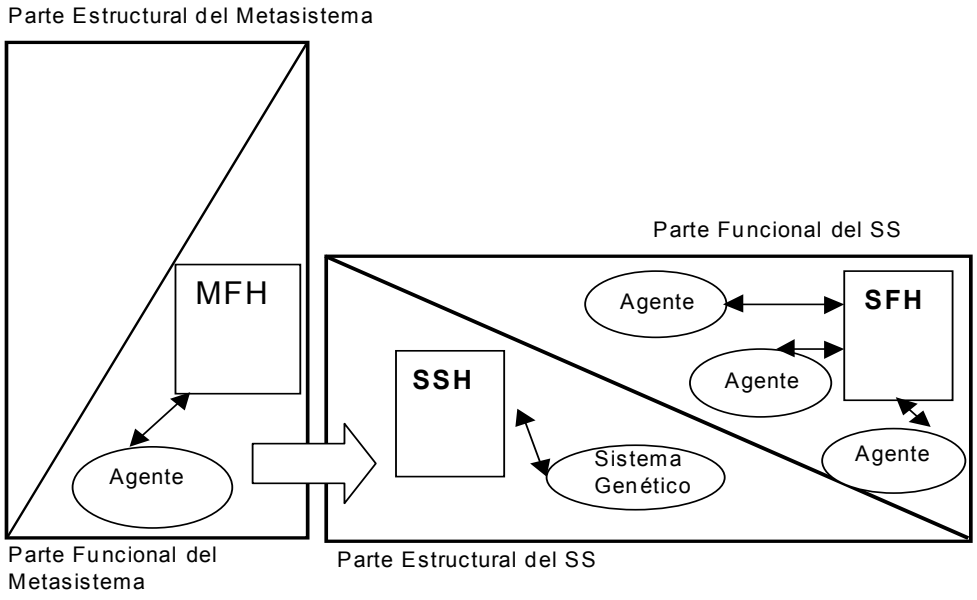


Figura 5.8 Relación Metasistema-SS a través de la interfaz de evolución

El Sistema Genético almacena en la historia estructural del agente las ocurrencias de las acciones estructurales realizadas en él. Esta información permite saber qué cambios ha sufrido un agente durante su vida, de tal forma que podríamos reconstruir el sistema en un instante anterior. Cuando el Metasistema determina que una acción estructural se puede llevar a cabo, el Sistema Genético realiza los cambios en la estructura del agente.

5.3.8 El reloj del Sistema Software.

Cada sistema posee un reloj interno que avanza periódicamente. Cada vez que se realiza una acción y cada vez que se recibe un estímulo, se toma el valor que marca el reloj y se almacena como valor del atributo *tiempo de realización* de la ocurrencia de acción o estímulo. Cuando dicha ocurrencia o estímulo se almacenen en la historia (funcional o estructural, según el caso), se hará en orden, según el valor de su atributo tiempo de realización. Esto es necesario porque nos interesa mantener la información sobre el orden temporal en el que se producen las acciones y/o llegan los estímulos al sistema para poder comprobar en cada instante si las condiciones asociadas a las acciones se cumplen o no.

Este reloj puede ser simplemente un temporizador creado e inicializado al generarse el sistema. El valor inicial será el 0 y, por tanto, el rango de valores que puede tomar el tiempo de realización asociado a las ocurrencias de acciones y estímulos producidos en el sistema será el de los números naturales [0,1, ...]. Este temporizador va incrementando su valor en períodos constantes de tiempo. La longitud de dicho periodo se puede determinar en la generación e inicialización del sistema. Como se puede observar el tiempo se trata como discreto, no como continuo ya que nos interesa sólo el tiempo en el que se finalizó la acción y no cuánto ha durado su realización.

El funcionamiento del SS se detiene mientras que se realizan acciones estructurales (acciones del Metasistema encargadas de cambiar la estructura del sistema) para que no se produzcan simultáneamente ocurrencias de acciones funcionales. Si así fuera, podrían darse situaciones no deseables, por ejemplo, el Metasistema podría estar borrando un agente y, al mismo tiempo, el agente padre desea que dicho agente realice una acción. El Metasistema y el SS comparten un mismo reloj temporal.

5.4 Problemas en el funcionamiento de los agentes.

Una vez visto el entorno en el que trabajamos y las definiciones básicas del modelo, podemos identificar varios problemas respecto al funcionamiento de los agentes.

5.4.1 Problema de concurrencia.

Como hemos dicho, las acciones tienen asociadas unas pre-condiciones que deben ser ciertas antes de iniciarse. Para saber si se cumplen o no, el agente tendrá que comprobar si las ocurrencias de acciones y/o los estímulos que definen la pre-condición están o no en la SFH de su agente padre, evaluar la pre-condición y, si ésta se cumple, comenzar a realizar la acción asociada. Además, cuando un agente realiza una acción, la ocurrencia de dicha acción debe almacenarse en la SFH de su agente padre para que sea conocida por el resto de sus agentes hermanos.

En un agente pueden existir distintos agentes hijos trabajando de forma concurrente, por tanto, puede darse la situación en la que existen dos o más agentes intentando comprobar las pre-condiciones de sus acciones, es decir, buscando la información necesaria en la SFH o intentando almacenar una ocurrencia de una acción finalizada. Por tanto, en un momento dado pueden realizarse distintos accesos de lectura/escritura en una estructura de datos común como es la historia. Esto nos lleva a pensar en el típico problema de Lectores/Escritores tan conocido en el ámbito de la Programación Concurrente [Bacon98].

Los problemas que se pueden originar son:

- *Una historia con información inconsistente*: si dos agentes intentan, a la vez, escribir una ocurrencia de acción en la historia, puede que dichas ocurrencias de acciones o alguna de ellas no llegue a almacenarse realmente en la SFH y por tanto, su estado será inconsistente. Este problema, según los términos utilizados en la programación concurrente, se produce por las *condiciones de competencia (o de carrera)* [Tanenbaum98] que se pueden producir ante dos accesos de escritura a una estructura de datos común. Recordemos que la acción de escribir en la historia no es una operación atómica, por tanto, dos escrituras pueden entremezclarse y finalmente, puede que el resultado no sea el adecuado.

- *Un mal funcionamiento de los agentes*: sería lógico pensar que el acceso concurrente a la historia para lectura no presenta ningún problema (típico problema de lectores/escritores) pero puede ser que las pre-condiciones de dos acciones no deban evaluarse concurrentemente. En este caso decimos que existe un conflicto entre las evaluaciones de dichas acciones. Un conflicto entre dos o más acciones se origina porque en sus pre-condiciones existen ocurrencias de acciones y/o estímulos que sólo son válidos para una de ellas, como si fueran recursos consumibles. Un ejemplo sencillo es el siguiente: supongamos dos acciones *alquilar* y *pintar*. Ambas

tienen asociada la misma pre-condición compuesta por la acción *devolver*. En el sistema de alquiler de coches sólo existe un coche. Si se da *devolver*, sólo una de dichas acciones puede realizarse (la misma acción no sirve para activar a las dos ya que, o bien se alquila de nuevo el único coche que existe, o se pinta). Si ambas se evalúan concurrentemente (se lee en la historia), ambas puede encontrar su pre-condición cierta y ambas se pueden iniciar cuando esto va en contra de la especificación dada (el coche no puede estar a la vez pintándose y estar alquilado).

- *Incompatibilidad entre acciones*: pueden existir acciones que no se deban ejecutar simultáneamente en un agente. Con el sistema de pre-condiciones descrito sólo podemos establecer el estado en el que tiene que estar un agente para que se inicie una acción pero no la incompatibilidad de la ejecución simultánea de dos o más acciones.

Por último, como vimos en puntos anteriores, no sólo existe una historia, cada agente tiene su propia historia y los agentes hermanos comparten la historia del agente en el cuál están definidos. Por tanto, para cada historia tenemos distintos agentes que están involucrados en el acceso concurrente a dicha historia. El acceso de agentes a distintas historias no presenta un inconveniente puesto que están trabajando sobre estructuras de datos distintas. Debemos proteger cada historia de forma individual y sólo involucrando a los agentes relacionados. No se debe permitir que un agente que va a almacenar una ocurrencia de acción en una historia *H1* tenga que bloquearse porque otro agente del sistema está leyendo de una historia *H2*.

5.4.2 Problema de la activación de los agentes.

Los agentes no se conocen entre sí y no saben lo que otros están haciendo. Los agentes sólo saben que deben realizar acciones y que para que las realicen, su agente padre tiene que estar en un determinado estado, es decir, debe haber ocurrido (o no) ciertas acciones en él y/o haber recibido ciertos estímulos externos a él. Por tanto, todo se rige por la historia, pero:

- ¿Cómo saben los agentes cuándo deben comprobar si pueden o no realizar una determinada acción? Es decir, ¿cómo saben cuándo deben activarse?

- ¿Debe un agente estar continuamente comprobando las pre-condiciones de sus acciones para activarse?

No sería eficiente para un SS que los agentes realizaran comprobaciones inútiles y con ninguna probabilidad de éxito. Como

hemos dicho, un agente no sabe si el estado de su agente padre ha cambiado o no, es decir, no sabe si se ha realizado una determinada acción. Por tanto, la única forma que tendría de enterarse sería comprobar periódicamente (es decir, leer) el contenido de la historia por si ha ocurrido algo que haga que una de sus acciones se active. Esto ralentizaría el funcionamiento del SS porque aparecerían cuellos de botella en el acceso a cada historia. Interesa que un agente acceda a la historia sólo cuanto realmente exista algún indicio de que la pre-condición de la acción que va a evaluar puede ser cierta.

5.4.3 Problema de evolutividad.

Un sistema software puede cambiar, evolucionar a lo largo del tiempo; pueden aparecer nuevos componentes (agentes, acciones) y desaparecer otros, o puede cambiar el comportamiento de un agente (las acciones que realiza, la pre-condición de alguna acción). Este cambio debe estar controlado y debe actualizarse lo antes posible en el SS. Es importante hacer notar que la arquitectura de agentes de un SS debe ser dinámica para que éste pueda adaptarse a los cambios que se producen en su entorno y que afectan a los objetivos que se definieron inicialmente y que están totalmente relacionados con su funcionamiento.

Deben controlarse los cambios porque también es importante poder establecer cuándo se debe o no permitir un cambio en un agente y si éste debe propagarse y hasta dónde. Por ejemplo, puede ocurrir que no queramos permitir borrar una acción de un agente si ésta se está realizando en el instante actual o si no hay ningún otro agente que la realice (dentro de un agente podemos tener más de un agente hijo que realice la misma acción). Otra alternativa es permitir la eliminación de dicha acción y propagar esta modificación de forma que cualquier pre-condición que contenga el nombre de la acción será modificada eliminándola de su definición.

En cuanto se realice un cambio en el SS, toda la información relacionada debe actualizarse, propagándose el cambio, para actuar en consonancia con la nueva estructura y funcionamiento del SS. Antes de que vuelva a funcionar el SS (recordemos que mientras evoluciona se para su funcionamiento) debe estar todo el SS en un estado consistente.

5.4.4 Problema de la ejecución de transacciones.

Recordemos (ver punto 5.3.2) que un aspecto importante dentro de un sistema es que proporcione la forma de especificar y realizar acciones complejas. Para nosotros, una acción compleja es una serie de acciones (simples o complejas) relacionadas entre sí según un orden temporal.

Estas acciones tienen que realizarse o todas o ninguna. Esta definición corresponde a la definición de transacciones [Bacon98] que ya nos es tan familiar (sobretodo en el ámbito de las Bases de Datos). Por tanto, en vez de hablar de acciones complejas hablaremos de transacciones.

Cuando se define un SS, puede ser necesario tener la capacidad de definir transacciones. Este problema de definición de transacciones involucra un trabajo adicional para el Metasistema, quién tendrá que realizar distintas comprobaciones antes de permitir la creación de una transacción. Estas comprobaciones pueden ser, por ejemplo, que las acciones que aparecen en la definición de la transacción existan, es decir, que exista al menos un agente que las realice o que no haya incompatibilidades entre las acciones que forman parte de la transacción. Para ello, tenemos un mecanismo que compara la definición de una acción compleja dada por el desarrollador en el lenguaje TDL con la especificación actual del SS y detecta las posibles incoherencias e incompatibilidades.

La definición de transacciones está muy ligada al concepto de agentes anidados. Como hemos visto, cuando definimos un sistema software como un conjunto de agentes, éstos no tienen porqué ser simples, pueden a su vez, componerse de otro conjunto de agentes que, mediante su colaboración, llevan a cabo una transacción. La colaboración entre estos agentes se realiza para llevar a cabo un acción compleja o transacción.

Las transacciones, al igual que las acciones simples, tienen asociadas unas pre-condiciones que determinan cuándo se pueden realizar y unas durante-condiciones que se usan para interrumpir su ejecución.

5.5 El agente Controller.

Con el fin de comenzar a solucionar gradualmente los problemas presentados, primero nos centraremos en los problemas de concurrencia y de activación. El problema de la evolución se encuentra muy ligado a los dos anteriores y según vayamos presentando una solución, veremos cómo ésta lo trata. El problema de la ejecución de transacciones lo abordaremos después, ya que es más específico y requiere un análisis independiente.

Supondremos inicialmente que la historia compartida por los agentes (*blackboard*) es un almacén de capacidad ilimitada, por tanto, no hay problemas de desbordamiento. Observamos que el primero, el problema de concurrencia, es un problema de exclusión mutua, similar al problema de los lectores y los escritores.

Puesto que los problemas comentados se refieren al uso apropiado de la historia que todos los agentes hermanos comparten, hemos creado un agente especial que llamamos *Controller* para gestionarla. El *Controller* es un agente más, ya que tiene la misma estructura que la descrita para todos los agentes, pero por otro lado es especial en el sentido de que no implementa funciones del SS (no realiza acciones funcionales) sino que desempeña una labor de control que hace que el sistema software funcione correctamente y resuelva los problemas anteriormente descritos. Los agentes pueden comunicarse con el agente *Controller* para realizar sus peticiones de lectura (comprobación de pre-condiciones y consultas) y escritura (almacenar una ocurrencia) relacionadas con la historia y éste las servirá. Además, como veremos a continuación, también el agente *Controller* se comunicará de forma explícita con agentes concretos para que logren activarse.

Lógicamente, la comunicación agente-*Controller* debe ser explícita aunque sigue siendo asincrona. Esto es así porque si necesitamos otra historia no solucionamos nada porque entraríamos en un ciclo ya que necesitaríamos otro *Controller* para dicha historia. Tengamos en cuenta que los agentes deberían acceder a la historia para escribir, por ejemplo, sus peticiones de servicio de donde las recogería el *Controller* y siguen existiendo los mismos problemas de concurrencia que describimos anteriormente.

Debido a sus responsabilidades anteriormente comentadas, dividimos al agente *Controller* en dos partes conceptualmente distintas. Ambas partes serán llevadas a cabo por dos agentes hermanos diferentes: el agente *Evaluator* y el agente *Ocurrence Receiver*. La estructura básica de este agente se puede ver en la siguiente figura.

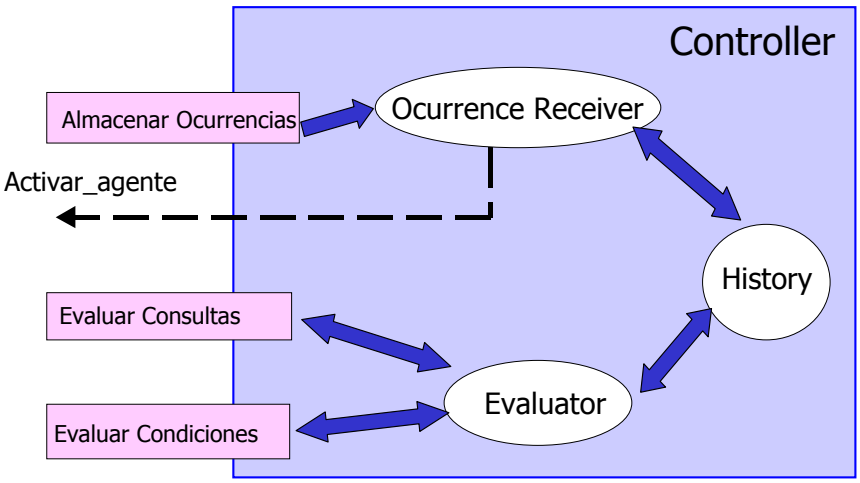


Figura 5.9 Estructura básica del Controller

5.5.1 El agente *Evaluator*.

Se encarga de recibir las peticiones (comunicación explícita) para evaluar las pre-condiciones procedentes de los agentes y, lógicamente, de realizar dicha evaluación y enviar una respuesta con el resultado de la evaluación a los agentes. Esta operación, la comprobación de una pre-condición, puede llegar a ser bastante compleja. No sólo hay que comprobar si ciertas ocurrencias de acciones y/o estímulos se encuentran en la historia, sino además, si otras no se encuentran y verificar las relaciones lógicas y temporales que se establecen entre éstas.

Debido a las continuas consultas a la historia que tendría que hacer un agente para evaluar una pre-condición, es más eficiente desligar esta comprobación de las responsabilidades propias de los agentes y ligarlas al agente *Evaluator*. Con esto conseguimos que los agentes, una vez que soliciten una evaluación determinada, puedan continuar realizando otra u otras acciones mientras esperan el resultado (que puede ser un valor *true* o *false*). Esto es especialmente útil si los agentes tienen una concurrencia intra-agente. Permitir concurrencia interna aumenta el poder de expresividad así como la concurrencia global, pero necesita controles de concurrencia adicionales para asegurar la consistencia del estado del objeto y una gestión cuidadosa de los recursos para mantener implementaciones eficientes.

Inicialmente existe un *Evaluator* por historia aunque la implementación de éste puede consistir en un conjunto de distintas entidades evaluadoras que se ejecutan concurrentemente con el fin de decrementar el tiempo necesario para realizar la evaluación. El *Evaluator* recibe una determinada petición de evaluación de una pre-condición a través del *Controller* y comprueba su validez o no basándose en el contenido de la historia. Si la pre-condición se cumple, la acción asociada podrá realizarse en el sistema; si no se cumple, la acción no se realizará por el momento. El funcionamiento del *Evaluator* está descrito en la tesis doctoral de M.J. Rodríguez [Rodríguez00a]. Un *Evaluator* sólo tiene acceso a la historia del *Controller* donde se encuentra anidado y no a distintas historias que existen distribuidas por la jerarquía de agentes existentes.

A veces, los agentes pueden necesitar cierta información de la historia del agente padre para realizar, por ejemplo, una acción que necesita tomar unos valores concretos en sus atributos. Los agentes solicitarán al *Evaluator* dicha información (los valores que tomarán los atributos) y el resultado de la consulta puede ser una lista de elementos. Por ejemplo, en un sistema de alquiler de coches, un agente quiere saber qué coches han sido alquilados en una cierta semana. Esta consulta la llevará a cabo el *Evaluator* quien, una vez consultada la historia, devolverá una lista con los identificadores de los coches que han sido alquilados en esa

semana. Nótese que esta solicitud difiere de la petición de evaluación de una pre-condición, donde la respuesta es sólo un *true* o un *false*. Por tanto, realizar estas consultas es otra de las funciones asociadas a este agente *Evaluator*.

Mientras el *Evaluator* realiza alguna de sus funciones, se bloqueará el acceso a la historia. La razón de hacerlo así es que no es deseable que cambie el contenido de la historia mientras se trabaja sobre ella y, por tanto, no se permitirá que se escriban nuevas ocurrencias y/o estímulos. Estas ocurrencias y/o estímulos, que son almacenadas temporalmente por el agente *Ocurrence Receiver*, serán introducidas en la historia cuando ésta se desbloquee.

5.5.2 El agente *Ocurrence Receiver*.

Se encarga de recibir los resultados de las realizaciones de las acciones de los agentes (ocurrencias de acciones) y los estímulos procedentes del exterior y registrarlos en la historia. Si la historia está bloqueada, los almacena temporalmente en un *buffer* interno de tal forma que para los agentes es como si dichas ocurrencias y/o estímulos se hubieran almacenado ya en la historia. Esto produce una mejora de la eficiencia global del sistema, ya que los agentes pueden continuar trabajando como si la historia hubiera sido realmente actualizada y no tienen que esperar a que ésta se desbloquee. Cuando la historia se desbloquee y llegue la siguiente ocurrencia o estímulo a almacenar o el *Evaluator* intente evaluar una nueva pre-condición se vaciará el contenido del *buffer* interno del *Ocurrence Receiver* a la historia para que contenga realmente el estado actual del agente.

Además, pensando en la activación de un agente y en cómo se debe realizar, este elemento mantiene información suficiente para ser él quien indique (active) a los agentes para que intenten evaluar una determinada pre-condición de una acción. Recordemos que este agente es quien recibe las ocurrencias de acciones y los estímulos y que, simplemente manteniendo información de a qué agentes afecta dicha ocurrencia, es el más idóneo para realizar este cometido.

La información necesaria para lograr este objetivo relacionaría a los agentes, a las acciones de éstos y a las ocurrencias de acciones y/o estímulos involucrados en las pre-condiciones asociadas a las acciones. Esta información se introduce al crear los agentes y la mantiene el agente *Ocurrence Receiver* en unas tablas internas. Si el sistema evoluciona y esta información varía, los datos almacenados en estas tablas deben ser consistentes y reflejar la nueva estructura y funcionamiento del sistema.

La activación de un agente sólo se realiza si existe alguna posibilidad de que pueda ser cierta alguna de sus pre-condiciones, no se activa indiscriminadamente a cualquier agente del sistema.

La función de este agente resuelve el problema de la activación presentado anteriormente, el hecho de que los agentes estén continuamente intentando comprobar las pre-condiciones de sus acciones. El *Ocurrence Receiver* supone que existe alguna posibilidad de realizar una acción cuando le llegue una ocurrencia de acción o un estímulo para almacenar en la historia que sea parte de la pre-condición de alguna o algunas acciones de ciertos agentes. Una vez determinados los agentes con posibilidades, el *Ocurrence Receiver* les envía un mensaje junto con el nombre de la acción (o acciones) para que inicien la evaluación de sus pre-condiciones correspondientes. El hecho de activar a un agente, no significa que éste puede comenzar directamente a realizar alguna de sus acciones, simplemente que tiene la oportunidad de solicitar la evaluación de una (unas) pre-condición (o sea, activar al *Evaluator*) y, si ésta se evalúa a *true*, entonces, y sólo entonces, podrá comenzar a realizarla.

Hay que tener en cuenta que aunque un agente tenga la posibilidad de realizar una acción, puede que en ese momento no quiera o pueda realizarla, por ello no se evalúa la pre-condición directamente, sino que se le notifica al agente la posibilidad y él decide si se evalúa o no. Recordemos que una de las características de un agente es la pro-actividad, es decir, la capacidad de decidir y tomar la iniciativa. Por tanto, si en un momento dado, es posible que pueda realizar una acción por las condiciones de su entorno, el agente puede decidir no hacerlo por otros motivos.

Podemos explicar mejor cómo trabaja este *Controller* y sus distintos componentes mediante un diagrama de secuencia de UML [Booch99] que ejemplifique las distintas actuaciones de los diferentes elementos respecto a cómo se activan y se llevan a cabo las acciones por los agentes de un SS. Vemos que el hecho de que se produzca una acción puede influenciar en la activación de otra acción.

La secuencia de la figura siguiente comienza con la intención de un agente, el *Agente_1*, que acaba de realizar una acción y solicita al *Controller* que almacene la ocurrencia correspondiente en la historia (paso 1). El elemento encargado de esta función del *Controller* es el *Ocurrence Receiver* quien, en cuanto pueda, almacena dicha ocurrencia (paso 2) en la historia del sistema. Además, y debido a la información que tiene del sistema y sus agentes, se da cuenta que dicha ocurrencia está en la pre-condición de la *acción-x* que puede realizar el *Agente_2*. Por tanto, dado que existe una posibilidad de que dicha acción se pueda realizar, el *Ocurrence Receiver* activa al *Agente_2* (paso 3) indicándole

que puede iniciar la comprobación de la pre-condición de la *acción-x*. El *Agente_2* solicita la evaluación de la pre-condición de la *acción-x* (paso 4). El *Evaluator* realiza tal evaluación consultando la historia (pasos 5 y 6) y finalmente emite un resultado (paso 7) que recibirá el *Agente_2* y que determinará si la *acción-x* se va a realizar (evaluación *true*) o no (evaluación *false*).

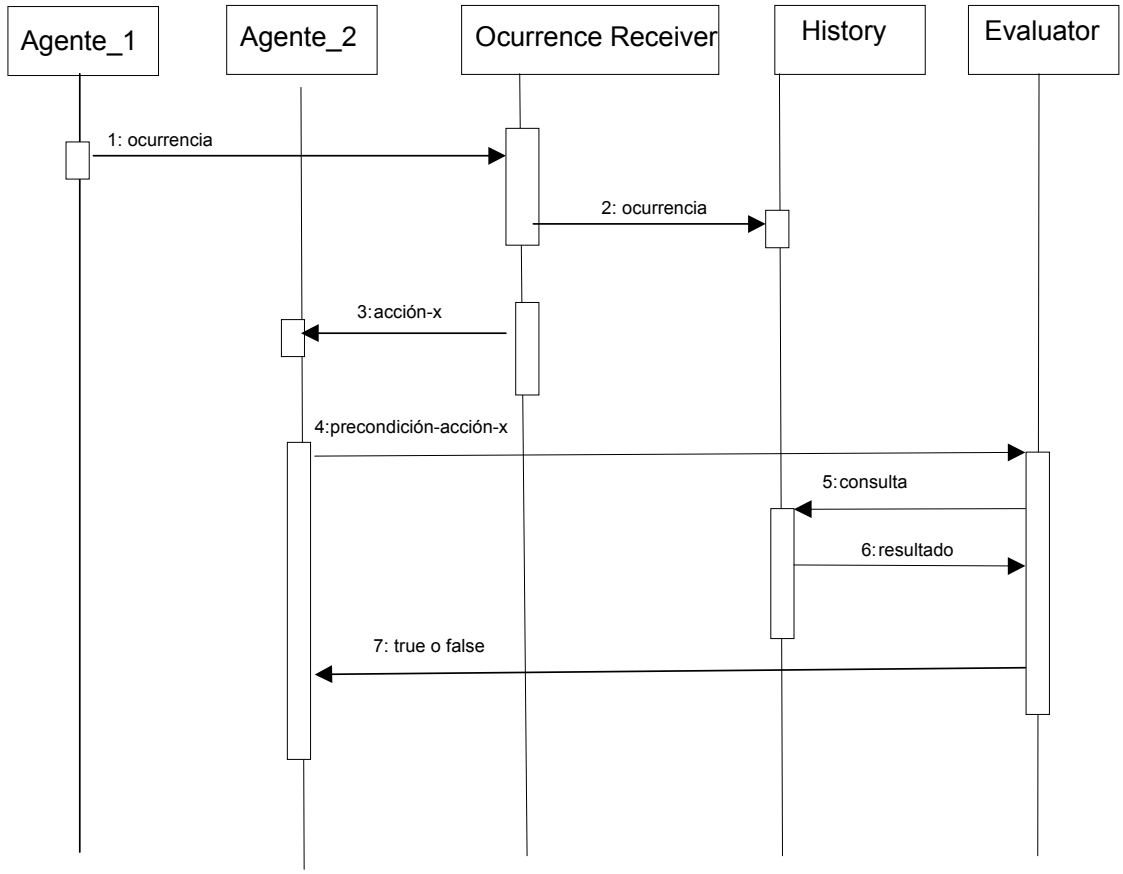


Figura 5.10 Diagrama de secuencia con el que vemos el funcionamiento de los agentes del SS con Ocurrence Receiver, Evaluator y History

Una vez comentado el diseño del *Controller* que hemos realizado, necesitamos abordar el problema la evolución de los SS. Este problema es importante ya que cuando un SS evoluciona, se producen cambios en él y estos cambios afectan a sus componentes, los agentes. Puede ser que sean eliminados, o, simplemente, que cambien su comportamiento, las acciones que realizan o las pre-condiciones de dichas acciones. Esto influye en el *Ocurrence Receiver*, ya que al encargarse de la activación, debe mantener la información de sus tablas actualizada en todo momento.

Al permitir que la arquitectura del sistema sea dinámica, cuando se produce cualquier cambio en él, el Sistema Genético le debe de comunicar dichos cambios al *Ocurrence Receiver* quien actualizará su

información y, una vez actualizada, comenzará a trabajar de forma consistente con la nueva estructura y funcionamiento del sistema.

5.5.3 Definición del patrón *PDN (Precondition Dynamic Notifier).*

El diseño que acabamos de presentar se generalizó y dio lugar a la definición de un patrón de diseño [Paderewski99b] que puede ser utilizado en la creación de sistemas basados en una arquitectura *blackboard* y que resuelve el problema de activación de las distintas entidades activas y concurrentes que llevan a cabo la funcionalidad de un sistema software. Las entidades activas pueden ser cualquier elemento que sea capaz de realizar acciones cuando su entorno se lo permite. En la definición del patrón hemos particularizado estas entidades en objetos activos pero podrían utilizarse para cualquier otro tipo que cumpla con las características citadas, por ejemplo, una hebra o un proceso.

La definición del patrón sigue la plantilla propuesta por Gamma [Gamma94]. Esto nos ha parecido muy conveniente porque aporta bastante información para poder hacer comprender los aspectos y decisiones de diseño que se han tenido en cuenta en la elaboración del patrón y de la solución anteriormente elaborada.

5.5.3.1 Nombre y Clasificación.

El patrón *PreconditionDynamicNotifier (PDN)* o Notificador Dinámico basado en Pre-condiciones se encuadra en el conjunto de patrones de comportamiento, según la clasificación de Gamma [Gamma94]. Su nombre deriva de las características más importantes:

- Notificador: notifica los eventos ocurridos en el sistema a los objetos activos que puedan estar interesados. En nuestro caso los eventos son las ocurrencias de acciones realizadas por los objetos activos o los estímulos procedentes del exterior del sistema o de otros objetos activos.
- Dinámico: se adapta a los cambios de estructura del sistema.
- Pre-condiciones: cada acción realizable en el sistema tiene asociada una pre-condición que debe cumplirse antes de su realización.

5.5.3.2 Propósito.

Mantener actualizada la historia de las acciones producidas por todos los objetos activos involucrados en un determinado sistema y notificar a los objetos activos interesados que se ha realizado una determinada acción (se almacena una ocurrencia de acción) en el sistema para que se activen. Los objetos activos se ejecutan concurrente e independientemente unos de otros. La comunicación entre ellos, si es necesaria, se realiza a través de la historia mediante el uso de pre-condiciones de activación.

5.5.3.3 Motivación.

Podemos identificar los dos problemas que resuelve, principalmente:

a) Por un lado, disponemos de una historia donde se almacenan las distintas ocurrencias de acciones/estímulos producidos por los objetos activos. Aparece un problema de compartición de recursos (condiciones de competencia) por los distintos objetos activos a la historia, ya que podrían intentar utilizarla a la vez para introducir una ocurrencia de acción. Es un problema de exclusión mutua.

b) Además, las ocurrencias de acción y estímulos guardados en la historia influyen en la activación de las acciones que realizan los objetos activos. La ocurrencia de acción generada por un objeto activo puede formar parte de una pre-condición para activar una acción de otro objeto activo. Los objetos activos consultan la historia del sistema para comprobar si se cumplen sus pre-condiciones.

5.5.3.4 Aplicabilidad.

Cuando se necesita modelar un sistema concurrente que sigue una arquitectura *blackboard*. El sistema está compuesto por objetos activos que realizan sus acciones independientemente de los demás. Los objetos activos comunican sus actividades al resto de los objetos activos del sistema a través de una estructura común que almacena la historia de las ocurrencias de acción generadas por cualquier objeto activo del sistema y cuya constancia es necesaria para el buen funcionamiento del resto de objetos activos. Con esto no se pierde la independencia de los objetos activos y, sin embargo, se oculta el hecho de que las acciones de uno dependen de las acciones realizadas por otro. Las acciones tienen asociadas unas pre-condiciones, que se deben cumplir y que se basan en el estado actual del sistema.

5.5.3.5 Estructura.

En la siguiente figura se puede ver el diseño de clases que se ha realizado según la metodología de *UML* [Booch99].

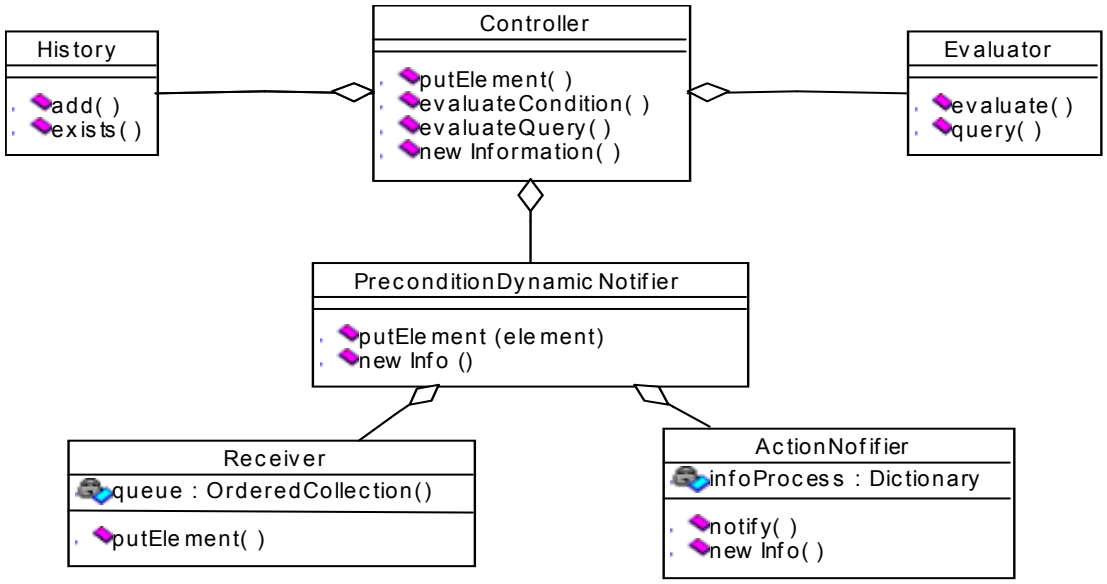


Figura 5.11 Diseño de clases del patrón PDN

5.5.3.6 Participantes.

History: Es la historia de los eventos generados por los objetos activos junto con la información temporal del instante en el que se han realizado. Proporciona operaciones de acceso: añadir un elemento (*add*) o comprobar si una determinada ocurrencia de acción o estímulo está en la historia (*exists*).

Controller: Encapsula el estado del sistema. Los objetos activos sólo ven al *Controller* del sistema donde están definidos. Las peticiones de guardar ocurrencias de acciones y estímulos y de consulta de pre-condiciones se envían al *Controller*. En su interfaz se ofrecen los servicios de:

- Añadir ocurrencias de acciones o estímulos a la historia: *putElement*.
- Comprobar si cierta pre-condición es o no verdadera, basándose en el contenido actual de la historia: *evaluateCondition*.
- Realizar una consulta: *evaluateQuery*.

Estos servicios son reconducidos a los elementos (*Evaluator* y PDN) que realmente son los que realizan dichos servicios.

PDN: Esta clase describe el patrón de comportamiento que resuelve los problemas expuestos y está formado por dos componentes:

1. *Receiver*: Recibe las ocurrencias de acciones y estímulos que hay que almacenar en la historia y, cuando ésta está libre, los introduce en ella. Cuando introduce un elemento en la historia se lo notifica a *ActionNotifier*.

2. *ActionNotifier*: Tiene principalmente dos funciones:

    1) Controla la activación de los objetos activos de un determinado sistema. Cuando *Receiver* le comunica la recepción de una ocurrencia de acción a *ActionNotifier*, éste avisa a los objetos activos interesados, indicándoles qué acciones deben comprobar.

    2) Mantiene la información que relaciona las ocurrencias de acción, los objetos activos y las acciones de los objetos activos en una tabla (*infoProcess*) creada y modificada dinámicamente. Esta tabla es consultada cuando recibe una ocurrencia de acción para determinar los *objetos activos-acciones* interesados y poder activarlos. La tabla *infoProcess* se puede crear automáticamente a partir de las pre-condiciones de las acciones de los objetos activos. Igualmente, cuando el sistema evoluciona, las modificaciones realizadas le serán comunicadas dinámicamente para que pueda actualizar la información y mantenerla consistente con los cambios estructurales realizados en el sistema.

Evaluator: Tiene dos funciones:

1) Evaluar las pre-condiciones que le llegan de los objetos activos del sistema y devolverles el resultado de dicha evaluación, puede ser un simple *true* o *false*.

2) Realizar consultas sobre la historia. Dependiendo de cómo se construyan las pre-condiciones asociadas a las acciones, se estudiará qué se almacena en la historia y cómo se realizará la búsqueda en ella por parte del evaluador.

5.5.3.7 Colaboraciones.

El esquema de colaboración dentro de *Controller* queda reflejado en la figura siguiente. Los objetos activos que realizan las acciones en el sistema se comunican única y exclusivamente con el *Controller* apropiado (y único para un determinado sistema). Cualquier petición relativa a la historia está canalizada por él.

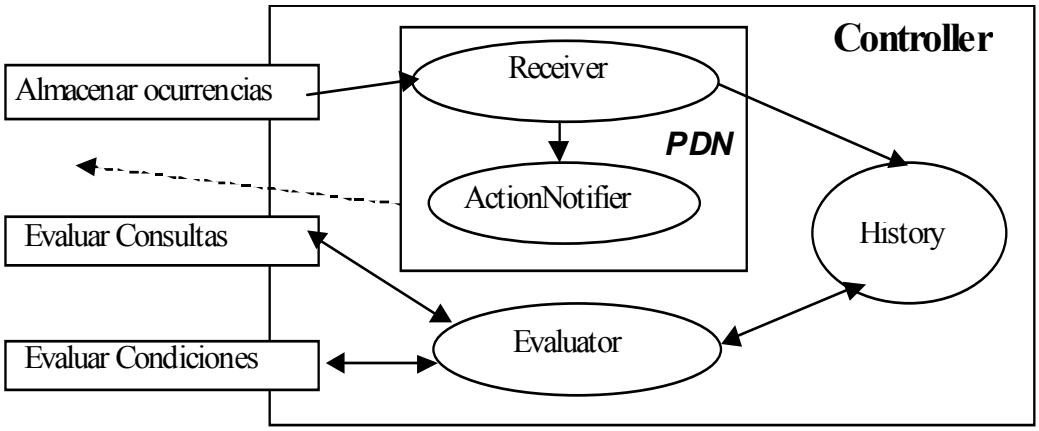


Figura 5.12. Relación entre los distintos elementos que componen Controller

Una vez que se ha producido una petición, dentro de la estructura de *Controller*, existen relaciones entre:

- *Receiver-History*: introducir un nuevo evento en la historia.
- *Receiver-ActionNotifier*: comunicación de la ocurrencia de acción o estímulo recientemente almacenado para provocar la activación del funcionamiento de algunos objetos activos. La activación se representa por una línea discontinua que parte de *ActionNotifier* hacia los objetos activos.
- *History-Evaluator*: para comprobar una pre-condición o realizar una consulta para un objeto activo determinado se ha de acceder a la historia del sistema. Mientras *Receiver* está trabajando con *History*, *Evaluator* no puede tener comunicación con *History* y a la inversa, para garantizar la consistencia de la información y por los problemas de acceso concurrente ya comentados.

El siguiente diagrama de secuencia (ver la siguiente figura) puede aclarar estas colaboraciones. Se produce una ocurrencia de acción como resultado de la realización de una acción de un objeto activo del sistema. *Receiver* almacena dicho evento en la historia (*History*) y después se lo comunica a *ActionNotifier* para que active a todos los objetos activos que estén interesados en dicha ocurrencia de acción

junto con información sobre la acción (o acciones) cuya pre-condición ha de comprobarse.

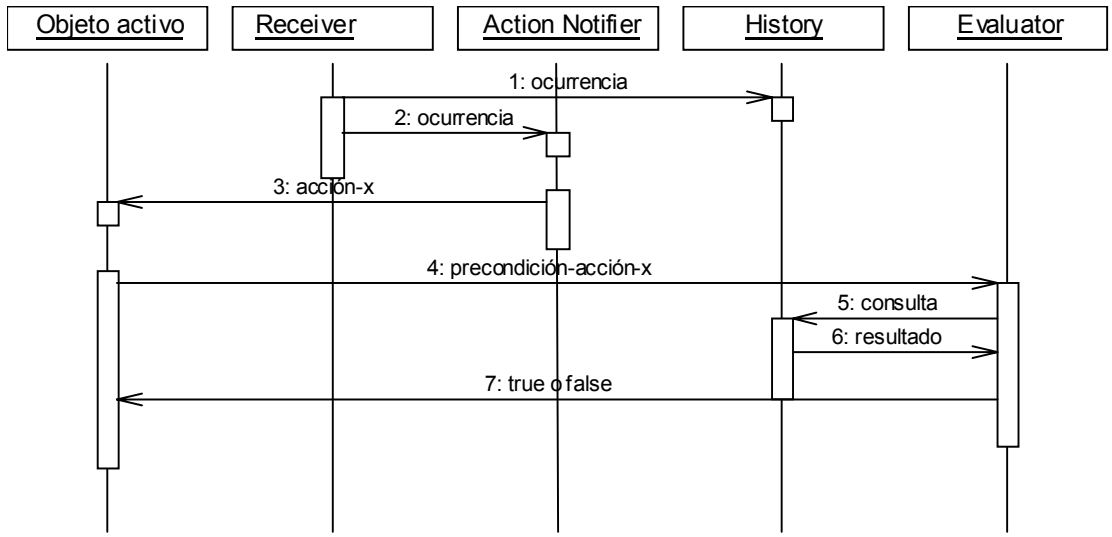


Figura 5.13 Diagrama de secuencia entre History, Evaluator y los componentes del patrón PDN

5.5.3.8 Consecuencias.

- Se separa lo que son funciones propias del sistema que se está modelando, es decir, las acciones que realizan los objetos activos, del estado del sistema y del control de activación de los objetos activos. Todo ello gracias a una comunicación a través de una estructura global, la historia.
- Protege el acceso a la historia común del sistema secuencializando los accesos a ella, pero permitiendo que los objetos activos envíen sus solicitudes de servicio y puedan continuar realizando otra acción sin tener que esperar a que esté libre el acceso a dicha estructura.
- Se puede modificar cada elemento independientemente sin que afecte a los demás. Permite, por tanto, la reutilización de componentes. Un sistema puede estar compuesto por subsistemas que tienen la misma estructura formando un sistema más complejo.
- La parte del *Evaluator* puede optimizarse y modificarse, no en cuanto a las relaciones con el resto de elementos, pero sí en cuanto a su funcionalidad. Depende mucho de cómo se decida implementar las pre-condiciones que se asocian a las acciones. Podríamos tener más de un *Evaluator* si optimizamos su trabajo haciendo que se evalúen concurrentemente varias pre-condiciones o las distintas partes de una pre-condición.
- Otro punto importante es la representación de la ocurrencia de acción. Esto puede hacer más compleja la historia y los algoritmos

de búsqueda y escritura, pero los métodos básicos siguen siendo necesarios. Hay que tener en cuenta la diferencia entre ocurrencias de acción y estímulos. La diferencia entre ellos puede variar un poco la actuación de *ActionNotifier*. Por ejemplo, si una pre-condición de una acción está formada sólo por un estímulo, entonces *ActionNotifier* podría activar directamente la acción del objeto activo sin que éste tuviera que comprobar si se cumple o no dicha pre-condición. Esta optimización nos lleva a una activación más eficiente.

- Un aspecto importante es el hecho de que un sistema *evoluciona*, se crean nuevos objetos activos, nuevas acciones o se cambian las pre-condiciones de otras dinámicamente. Es necesario que dicho cambio se propague hasta que *ActionNotifier* actualice la información que necesita para saber a qué objetos activos tiene que activar en un momento dado. Esto puede originar que, ante un cambio de este tipo, haya que evitar que siga trabajando el *Controller*. El objetivo de esta decisión es evitar que se presenten problemas de coherencia entre el funcionamiento actual del sistema y los cambios estructurales que se están realizando en él.

5.5.3.9 Implementación.

Existe una relación de agregación entre las clases *History*, *PDN* y *Evaluator* y la clase *Controller*. Esto sugiere que cada instancia de la clase *Controller* tiene sus propios elementos que serán variables de instancia de él. En su interfaz hay métodos para poder realizar las operaciones proporcionadas y realizadas por sus clases agregadas.

La clase *PDN* sería una clase con dos relaciones de agregación, una con la clase *Receiver* y otra con *ActionNotifier*.

No sólo puede ser necesario almacenar la ocurrencia resultante de realizar una acción, también puede ser útil almacenar la información de que una determinada acción ha comenzado a realizarse o que una acción ha sido cancelada. Lo único que varía es que los objetos activos que comienzan a hacer una acción, lo indican enviando un estímulo de inicio que se almacena en una estructura dependiente de *Receiver* y cuando la acción ha finalizado o se ha cancelado, se borra de dicha estructura y se procede convenientemente.

Para cada sistema solo es necesario una instancia de cada una de estas clases. Un único *Controller*, con su propio *PDN* y una sola instancia de *History*.

5.5.3.10 Código de ejemplo.

Mostramos sólo la parte de implementación más relevante de la clase *Controller* y de la clase *PDN* y sus clases agregadas. Hemos utilizado el lenguaje *Smalltalk* (*VisualWorks* 3.0)[1], y no otro lenguaje más comercial, porque es un lenguaje reflexivo que permite realizar creaciones, modificaciones y eliminaciones de la estructura de clases en tiempo de ejecución. El código puede verse a continuación. Presentamos parte del código de las clases *Controller*, *ActionNotifier* y *Receiver*.

```
Object subclass: #Controller
instanceVariableNames: 'history evaluator anPDN semaforo '
classVariableNames: ''
poolDictionaries: ''
category: 'HEDES-Controller'!

!Controller methodsFor: 'evaluation'!

evaluateCondition: aFormula

"Metodo para evaluar una pre-condicion. Bloquea a la historia al
inicio de la evaluacion y la desbloquea al final. Mecanismo de bloqueo
usado: semaforos"

|aux|
semaforo wait.
self acceptedElement.
aux:=(evaluator evaluate: (ConditionExp new: aFormula) withHistory:
history).
semaforo signal.
^aux.!

evaluateQuery: aFormula

"Metodo que realiza una consulta en la historia. Mientras se realiza,
la historia permanece bloqueada"

|aux|
semaforo wait.
self acceptedElement.
aux:=(evaluator query: (QueryExp new: aFormula) withHistory: history).
semaforo signal.
^aux.!

!Controller methodsFor: 'initialization'!

initialize

"Metodo de inicializacion de los elementos de un Controller"

history := History new.
evaluator := Evaluator new.
anPDN := PreconditionDynamicNotifier new: self.
```

[1] VisualWorks es marca registrada de Object Share, Inc.

```
semaforo:= Semaphore forMutualExclusion

!Controller methodsFor: 'elementReception'!

putElement: anElement

“metodo al que llaman los objetos activos para almacenar una
ocurrencia de accion. Se delega en el PDN”

anPDN putElement: anElement.!
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Controller class
      instanceVariableNames: ''!

!Controller class methodsFor: 'instance creation'!

new
^ super new initialize.! !

'From VisualWorksÂ®, Release 3.0 of February 5, 1998 on May 10, 1999
at 6:24:30 pm'!

Object subclass: #Receiver
instanceVariableNames:'elementQueue theController
lastOcurrenceOrStimulus anActionNotifier '
classVariableNames: ''
poolDictionaries: ''
category: 'HEDES-ElementHandler'!

!ElementR methodsFor: 'initialization'!

initialize: aController

“Metodo de inicializacion. Usa una cola donde almacena temporalmente
los
elementos que se deben guardarse en la historia”

elementQueue := Queue new.
anActionNotifier:= ActionNotifier  new.
theController:= aController.! !

!ElementR methodsFor: 'accesing'!

putElement: anElement

“Guarda el elemento recibido en la cola. Avisa a ActionNotifier de su
recepcion”

elementQueue put: anElement.
anActionNotifier done:anElement.! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ElementR class
      instanceVariableNames: ''!

!ElementR class methodsFor: 'creation'!

new: aController
```

```
^(super new ) initialize: aController.! !

'From VisualWorksÂ®, Release 3.0 of February 5, 1998 on May 10, 1999
at 6:24:26 pm'!
Object subclass: #ActionNotifier
instanceVariableNames: 'infoProcess elementInit '
classVariableNames: ''
poolDictionaries: ''
category: 'HEDES-ElementHandler'!

!ActionNotifier methodsFor: 'initialization'!

initialize: aController

"Metodo de inicializacion"

infoProcess := Dictionary new.
elementInit:= OrderedCollection new.! !

!ActionNotifier methodsFor: 'add information'!

newInfo: anOrderedCollection withElement: anElementName

"Metodo que actualiza la informacion que ha cambiado despues de una
accion de evolucion"

| proc |
(anOrderedCollection isEmpty)
      ifTrue:[self addElement:anElementName.]
      ifFalse:[proc:= anOrderedCollection removeFirst.
                  self addActions:anOrderedCollection to:proc
toElement:anElementName.].! !

!ActionNotifier methodsFor: 'accesing'!

notify:anOcurrenceOrStimulus

"Comprueba que objetos activos estan interesados en conocer que ha
llegado esta ocurrencia de accion o estimulo y les envia el nombre de
la accion o acciones para que inicien la evaluacion de su pre-
condicion"

|col|
col := OrderedCollection new.
(infoProcess keys) do:[:element| (element = anOcurrenceOrStimulus
name)
       ifTrue:[col add:(infoProcess at:element)].].
col do:[:dic|(dic keys) do:[:proc|(dic at:proc)do:[:act|self run:proc
with:act]]]! !

!ActionNotifier methodsFor: 'remove information'!

destroyInformation

infoProcess keys do: [:k | infoProcess removeKey:k].
elementInit := OrderedCollection new.!!
```

5.5.3.11 Patrones relacionados.

Comentaremos brevemente la descripción de otros patrones ya existentes y su relación con el patrón PDN o su entorno.

1) Patrón *Facade*:

Descripción: es un patrón de estructura [Gamma 94] que proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz a alto nivel que hace que el subsistema sea más fácil de usar.

Relación: se puede utilizar en la implementación de *Controller* porque básicamente se encarga de proporcionar las operaciones necesarias para los objetos activos. A los objetos activos les es indiferente que sean otros los que llevan a cabo realmente las operaciones solicitadas. Esto ayuda a minimizar la complejidad en el diseño del sistema.

2) Patrón *Mediator*:

Definición: es un patrón de funcionamiento [Gamma94]. Define un objeto que encapsula cómo interactúan un conjunto de objetos. Proporciona un acoplamiento débil impidiendo a los objetos referenciarse unos a otros explícitamente y permite variar su interacción independientemente.

Relación: puede servir también para diseñar *Controller*, desde el punto de vista del funcionamiento. *Controller* es responsable de coordinar un grupo de objetos mediante la historia y distribuye su comportamiento entre distintas clases (*PDN*, *History*, *Evaluator*). Sin embargo, este patrón no se utiliza para *PDN* porque no contempla una de sus principales funciones: informar a un objeto activo de que se ha realizado una determinada acción o de que se ha recibido un estímulo concreto.

3) Patrón *Observer* [Gamma94] (o *Publisher/Subscriber* [Coad95]):

Definición: es un patrón de funcionamiento. Su objetivo es definir una dependencia uno-muchos entre objetos, de tal forma que cuando un objeto cambie de estado se notifique y se actualicen todos los objetos que dependen de él.

Relación: Este patrón es parecido a *PDN* en cuanto a que realiza una propagación de los cambios de estado de un objeto a otros relacionados. Pero falta una parte muy importante, *PDN* informa a los objetos activos sobre cual (o cuales) de sus acciones podría intentar realizar como resultado del cambio de estado producido en el sistema. En el patrón

Observer, se notifica a los observadores que otro objeto ha cambiado su estado pero después los observadores deben solicitar información a los sujetos sobre dicho cambio de estado. En *PDN*, los observadores (los objetos activos) no conocen la existencia de los sujetos y cualquier información que necesiten la obtienen de la historia y se le notifica con la activación. Por tanto, el patrón *PDN* es más general.

4) Patrón *Notifier* [Mederos98]:

Definición: aísla y centraliza las dependencias muchos a muchos entre *n* objetos que pueden cambiar sus estados y *m* objetos dependientes de dichos cambios. Tiene similitudes con el patrón *Observer* y el patrón *Mediator*. Diseña un modelo de notificaciones de eventos a los objetos en aplicaciones con dependencias muchos a muchos.

Relación: Sirve para implementar parte de la funcionalidad asociada a *ActionNotifier* pero no completamente. El patrón *PDN* es más general en cuanto que aporta además información a los objetos activos (observadores en *Notifier*) y se encarga también de mantener la historia completa del sistema, su estado.

5) Patrón *Event Notifier* [Riehle96]:

Definición: Gestiona las dependencias de actualización entre objetos introduciendo un mecanismo de notificación de eventos basado en el mecanismo de invocación implícita.

Relación: Este patrón es conceptualmente similar al patrón *Observer* aunque proporciona una estructura distinta que resuelve algunas desventajas respecto a su implementación. Por tanto, la relación con *PDN* comentada para el patrón *Observer* es válida también para este patrón.

5.5.3.12 Ejemplos de uso del patrón.

Vamos a utilizar el patrón para modelar dos sistemas: la “simulación del funcionamiento de un vídeo” y la “simulación de una empresa de alquileres de coches”. Hemos elegido estos dos ejemplos porque en el primero queremos destacar que el patrón PDN es válido para modelar sistemas donde sólo existen estímulos. En el segundo ejemplo, además de los estímulos, tenemos acciones que influyen en la activación de otras acciones en el sistema.

a) Simulación del funcionamiento un vídeo

El primer ejemplo que vamos a modelar es el que apareció en el artículo [Mederos98]. Se trata de una aplicación de control y simulación de un vídeo junto con los objetos que definen la interfaz de usuario (objetos GUIs) necesaria.

Supongamos que:

- En la interfaz existe una ventana compuesta por varios elementos: barra de título, tres botones emulando los controles de un vídeo (*play, stop* y *pause*) y una lista de acciones realizadas anteriormente (histórico).

- En la barra de título, se muestra la acción que actualmente está realizando el vídeo.

- Cuando el usuario pulsa un botón, se dispara un estímulo y la aplicación debe reaccionar realizando la acción oportuna que previamente se ha asociado al estímulo.

Modelado del sistema software:

- Definimos un conjunto de estímulos producidos por los usuarios cuando éstos realizan una acción (por ejemplo, pulsar botón). Cada acción tiene asociado un estímulo distinto.

- La pre-condición de cada acción es el estímulo necesario para su activación (pulsar *stop*).

- Cuando el usuario realiza una acción a través de la interfaz, se genera un estímulo que se almacena en la historia (histórico).

- La historia puede visualizarse siempre actualizada en la lista de acciones.

- Cuando el *Receiver* almacena el estímulo en la historia se lo comunicará a *ActionNotifier* y éste comprobará a que objetos activos hay que comunicárselo. Por ejemplo, al objeto activo que cambia el título de la ventana. *ActionNotifier* le enviará un mensaje con el nombre de la acción y éste podrá enviar una solicitud al *Controller* para comprobar la pre-condición, determinar que es cierta y realizar el cambio del título en la ventana.

- La tabla con la información *objeto activo-accion-estímulo/ocurrencia* será muy sencilla y fácil de gestionar.

- En este caso el proceso de activación se puede optimizar. Como cada acción tiene asociada como pre-condición un solo estímulo, puede no ser necesario el *Evaluator*. Directamente se puede negociar la activación de las acciones en cuanto *ActionNotifier* notifique la realización del estímulo sin necesidad de la comprobación de la pre-condición. Sin embargo, hay que estudiar cada caso porque no siempre se cumple.

- Los objetos activos de la interfaz trabajan simultáneamente pero son independientes. Con el patrón de activación conseguimos también sincronizar las acciones de todos ellos controladas por la gestión del *Controller*.

b) Simulación de una empresa de alquileres de coches

En el segundo ejemplo, tenemos un tipo de objeto activo que se encarga de alquilar coches en una empresa de alquiler de coches.

Modelado del sistema software:

- En el sistema existirán distintos objetos activos que pueden, incluso, realizar las mismas acciones: *alquilar*, *devolver*, *comprar* y que se ejecutan de forma concurrente pero independientemente (en el sentido de que entre ellos no hay comunicación directa).

- Un objeto activo realiza la acción *alquilar*. La pre-condición de esta acción es: debe existir un coche para alquilar, pero esto hay que expresarlo en función de las acciones que se han producido hasta el instante actual en el sistema.

- Utilizando una sintaxis similar a la de la lógica temporal de predicados [Anaya96] [Anaya97] podemos expresar la pre-condición de la acción alquilar como:

  alquilar(x) <- (comprar(x) and (not alquilar(x) Since recoger(x))

  el significado es que se podrá alquilar un coche "x" siempre y cuando, previamente (está implícito en la semántica utilizada), se haya comprado y no se haya vuelto a alquilar desde que se recogió.

- Las acciones de unos objetos afectan a las de otros. Si uno realiza la acción *recoger*, el sistema debe notificar esta ocurrencia de acción a cualquier objeto activo que realice una acción en cuya pre-condición esté *recoger* para indicarle así que puede intentar realizar dicha acción. Después de esto, el objeto activo debe comprobar si la pre-condición completa es cierta antes de iniciar la ejecución de la acción.

- La tabla con la información *objeto activo-accion-ocurrencia/estímulo* puede ser más compleja ya que una ocurrencia y/o un estímulo puede estar involucrado en acciones del mismo objeto activo y en acciones de distintos objetos activos y en cada pre-condición, normalmente, hay más de un elemento. Esto dará lugar a una estructura anidada pero fácilmente construida en base a la información de las pre-condiciones. Por ejemplo, para el objeto activo con la acción *alquilar* y dada su pre-condición, se almacena la relación: *objeto activo->alquilar->ocurrencias comprar, alquilar y recoger*. Si uno de estos eventos se produce, se activará al objeto activo, se le comunicará que la activación está relacionada con la acción *alquilar* y el objeto activo intentará comprobar si la pre-condición de *alquilar* es cierta o no para comenzar a realizarla.

- Aparte de los objetos activos, en el sistema se crea una instancia de *Controller* que mantiene el estado global del sistema y activa a los objetos activos cuando se dé la más mínima probabilidad de que puedan realizar algunas de sus acciones (sin tener que estar continuamente comprobando si pueden o no realizar sus acciones).

- Puede sernos útil un diagrama de secuencia (figura siguiente) para saber qué ocurre en el sistema. Se puede observar que un determinado objeto activo (el Objeto_activo_1) ha realizado una recogida de un coche y esto ha generado una ocurrencia "recoger" que es enviada al *Controller* (paso 1). El *Controller* se lo envía a su vez al *PDN* (paso 2) para que lo almacene en la historia y para que le comunique a los objetos activos interesados con dicha ocurrencia de acción(por ejemplo, el Objeto_activo_N) junto con el nombre de la acción que ellos podrían intentar realizar (en el ejemplo, la acción *alquilar*, paso 4*).* Anteriormente se encuentra la definición de *alquilar* y se puede ver que *recoger* forma parte de su pre-condición). El Objeto_activo_N inicia una petición hacia el *PDN* (paso 5) para que se evalúe la pre-condición de dicha acción *alquilar*. Una vez realizada la evaluación, se le devolverá la respuesta que podrá ser un *true* o un *false* (paso 6). Si es *true*, podrá comenzar a realizar la acción. Se puede ver este ejemplo más desarrollado en el capítulo 8.

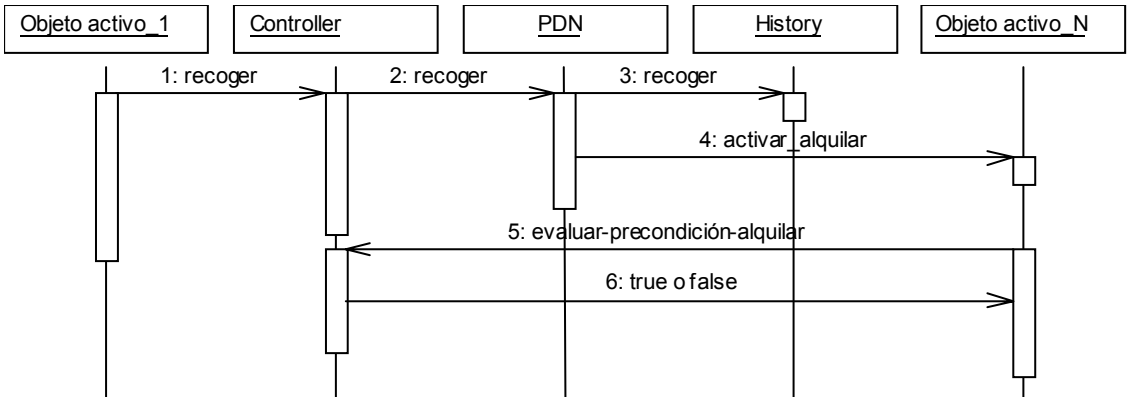


Figura 5.14. Diagrama de secuencia de un sistema de alquiler de coches

5.6 Un concepto adicional: Transacciones.

Como dijimos anteriormente, es necesario introducir el concepto de las transacciones o acciones complejas dentro de nuestro modelo. Una transacción se corresponde con una tarea, la cual se divide en otras más simples que se realizan en un determinado orden. Cada subtarea es realizada por un agente hijo del agente donde se encuentra definida la transacción [Paderewski02a]. Por tanto, las transacciones están estrechamente relacionadas con la jerarquía de agentes que define a un SS.

5.6.1 Concepto y propiedades de las transacciones

Una transacción es un conjunto de acciones que deben realizarse en su totalidad en el sistema, o bien, no debe realizarse ninguna. Este concepto está muy ligado al de agregación de agentes. Para que se realice una acción compleja, distintos agentes hermanos deben colaborar realizando las acciones que forman parte de ella. Si en la interfaz de un agente se encuentra una acción compleja, este agente no es simple sino que estará constituido por un conjunto de agentes que hacen posible que se realice la acción compleja colaborando entre ellos. Dicha acción compleja no es más que una transacción.

Las transacciones poseen cuatro propiedades [Tanenbaum96], llamadas propiedades *ACID (Atomicy Consistency Isolation Durability)*:

1. Atomicidad: esta propiedad garantiza que la transacción se realiza como si fuera una acción instantánea o indivisible.

2. Consistencia: Una transacción transforma el sistema de un estado consistente a otro. La transacción no viola los invariantes del sistema.

3. Aislamiento: Una transacción incompleta no puede revelar sus resultados a otras transacciones antes de que finalice correctamente. Las transacciones concurrentes no interfieren entre sí, es decir, para el sistema, una vez finalizadas, es como si estas transacciones se hubieran ejecutado de forma secuencial en un cierto orden.

4. Durabilidad o permanencia: Una vez finalizada una transacción, los cambios realizados en el sistema serán permanentes.

Para contemplar y manejar las transacciones tenemos que abordar los dos problemas que van ligados a ellas:

- El bloqueo de cualquier acción que forma parte de una transacción y que pueda invalidar el resultado de la transacción mientras ésta se está realizando.

- La recuperación (*backtraking*) en caso de que la transacción no pueda realizarse completamente. Es decir, el sistema tiene que quedar en el mismo estado que antes de que se iniciara dicha transacción. Este problema está relacionado con las propiedades de atomicidad y consistencia descritas anteriormente.

5.6.2 Las transacciones en nuestro modelo.

En el sistema definimos unas acciones especiales, transacciones, que se iniciarán con el mismo procedimiento que cualquier otra acción, es decir, por el cumplimiento de su pre-condición asociada. El agente *Controller* sabrá que es una transacción porque se crea con distinta acción estructural a la usada para las acciones simples.

Nos interesa distinguir las transacciones del resto de las acciones definidas (acciones que hemos llamado simples). Las transacciones se inician al cumplirse su pre-condición y el agente debe ser consciente de que se va a realizar una transacción y que todas las acciones que la constituyen no deben tener efecto en él hasta que la transacción finalice. Esto viene a contemplar la propiedad de aislamiento definida anteriormente. El resto de agentes que no están involucrados en la ejecución de una transacción no deben conocer que ésta se está llevando a cabo, ni tampoco conocerán los resultados parciales hasta que la transacción no finalice con éxito.

A los componentes de una transacción los denominamos *acciones transaccionales* para distinguirlos de las acciones simples o de otras

transacciones que son realizadas por los agentes que actúan fuera del ámbito de dicha transacción.

Necesitamos definir los componentes de una transacción y el orden en el cuál se han de ejecutar. Para ello proponemos un lenguaje que presentamos en el siguiente punto.

5.6.3 Lenguaje de Descripción de Transacciones (TDL).

En la definición de una transacción dentro de un agente debemos indicar qué acciones transaccionales y/o estímulos la constituyen. Es necesario un lenguaje que nos permita dicha definición. Nosotros hemos diseñado y utilizando un lenguaje al que llamamos Lenguaje de Descripción de Transacciones, TDL *(Transactions Description Language)*. Este lenguaje se construye utilizando un subconjunto de operadores de un lenguaje basado en CSP [Roscoe98]. Los operadores que nos interesan son:

- Operador de *secuencialidad*: *a* ; *b* siendo a y b acciones en el agente, indica que la acción b se debe ejecutar después de que se haya completado la acción a.

- Operador de *opcionalidad*: *a* | *b* siendo a y b acciones en el agente, indica que se puede ejecutar (de forma excluyente) a o b.

- Operador de *paralelismo*: *a* || *b* siendo a y b acciones en el agente, indica que a y b se deben ejecutar y que se pueden ejecutar simultáneamente y en cualquier orden.

Además, el orden de precedencia será de izquierda a derecha, pudiéndose modificar mediante la utilización de los paréntesis. En el cuadro 1 podemos ver la descripción en *BNF* del lenguaje *TDL*.

```
<Transaccion> ::= <Accion> <Operador> <Accion> |
                  <Accion> <Operador>  <Transaccion> |
                 '(' <Transaccion> ')'
<Accion> ::= letra { letra | numero }
<Operador> ::= ;  |  '|' |  '||'
```

Cuadro 5.1 Descripción BNF del lenguaje TDL

5.6.4 Pre-condiciones de una transacción.

Como decíamos anteriormente, las acciones de un agente se activan en base a pre-condiciones sobre la historia del agente padre (la historia funcional si hablamos de acciones funcionales o la historia estructural si hablamos de acciones estructurales). Para mantener una estructura homogénea en el manejo de transacciones se utilizan también pre-condiciones aunque son algo más complejas.

Como cualquier otra acción del sistema, la transacción tiene asociada una pre-condición, $\boldsymbol{p_t}$, que habitualmente será el estímulo necesario para su activación.

Cuando definimos una transacción indicamos mediante el lenguaje TDL las acciones transaccionales (simples o complejas) que la constituyen y el orden concreto en el que se deben realizar. Si dichas acciones transaccionales se corresponden a acciones que ya existen en el agente, éstas mantienen su pre-condición, ***p***. Si no existen, hay que definirlas. La pre-condición ***p*** se asocia a la acción en el momento en que ésta es creada aunque, posteriormente, se puede modificar en respuesta a un cambio en el agente en el cual está definida.

Además, por la definición de la transacción utilizando TDL, cada acción transaccional tiene asociada una *pre-condición de secuencia*, ***s***. Esta pre-condición de secuencia se deriva de la especificación del orden de ejecución de esta acción transaccional dentro de la transacción. Esta ***s*** se puede construir automáticamente pasando de la descripción de la transacción en TDL al lenguaje que se utiliza para describir una pre-condición en el sistema (LTP - Lógica Temporal de Predicados). Esto permite que los evaluadores de las pre-condiciones sean los mismos. La transformación de los operadores usados en la descripción de una transacción al lenguaje LTP según la semántica presentada en [Rodríguez00a] es la siguiente:

- Operador de secuencialidad: la traducción involucraría el operador *since*, ya que para hacer la acción b previamente se ha tenido que realizar a, "b desde a". Aunque este caso también puede reducirse a que se dé el último componente ya que por la secuencialidad las acciones anteriores han tenido que darse ya.

  (a ; b) ; c ------> pre-condición *s* de *c* es "*b since a*"

- Operador de opcionalidad: la traducción involucraría el operador *or* exclusivo, ya que sólo una de las dos acciones debe realizarse.

  (a | b) ; c -------> pre-condición *s* de *c* es "*a or b*"

- Operador de paralelismo: en este caso se utilizaría el operador *and*, ya que denota que ambas acciones tienen que realizarse.

    (a || b) ; c ------> pre-condición *s* de *c* es "*a and b*"

Realmente, el algoritmo que transforma una expresión en TDL en una expresión en lógica temporal de predicados con el fin de construir las pre-condiciones *s* de las acciones transaccionales, es complejo. Lo presentamos en un capítulo aparte donde profundizamos en el diseño del *Transaction Manager* (capítulo 6).

Por estar dentro de la transacción, la acción transaccional puede que requiera una pre-condición añadida que llamamos *pre-condición transaccional*, ***t*** (muchas veces puede ser simplemente *true*). Esta condición no se debe confundir con la condición de secuencia que se ha descrito anteriormente y surge para poder condicionar la realización de una acción transaccional a la realización de otras acciones transaccionales que hay definidas dentro de la misma transacción. Esto es necesario, por ejemplo, en el caso de que tengamos una transacción como:

$$T = a_{t1} ; a_{t2}$$

Imaginemos que no nos basta con que se ejecute a_{t1} sino que además debe devolver cierto valor, por ejecutarse dentro de la transacción, para poder realizar a_{t2}. Esta nueva condición es la pre-condición ***t*** de a_{t2}. Como explicaremos más adelante, esta condición se corresponde con una durante-condición ya que si no se cumple, se interrumpirá la transacción que finalizará sin que el agente donde se inició tenga conocimiento alguno de su ejecución parcial.

En el siguiente cuadro se pueden ver de forma resumida las distintas clases de pre-condiciones asociadas a una acción transaccional $\mathbf{a_{ti}}$.

Pre-condición de una transacción $\mathbf{p_t}$ = pre-condición asociada a la transacción por ser una acción más. Especifica las condiciones necesarias para que se inicie la transacción en el Sistema.
Pre-condiciones de una acción transaccional $\mathbf{a_{ti} = p_i + s_i + t_i}$
Pre-condición $\mathbf{p_i}$ = condiciones que han de cumplirse en el Sistema para activar esta acción independientemente de que se ejecute en el contexto de una transacción.
Pre-condición de secuencia $\mathbf{s_i}$ = condiciones derivadas del orden de $\mathbf{a_{ti}}$ en la transacción.
Pre-condición transaccional $\mathbf{t_i}$ = condiciones especiales que deben cumplirse para activar la acción transaccional y que se definen por el hecho de que la acción se ejecuta dentro de la transacción.

Cuadro 5.2 Pre-condiciones asociadas a una transacción y a una acción transaccional a_{ti}

Por ejemplo, la definición de una transacción **A** tendría un aspecto como:

$$\mathbf{A = a_{t1} \,||\, a_{t2} \;;\; a_{t3}}$$

y significa que estamos definiendo una transacción compuesta por tres acciones simples a_{t1}, donde cada a_{ti} tiene una $\boldsymbol{p_i}$, una $\boldsymbol{t_i}$ y una $\boldsymbol{s_i}$. Para aclarar un poco más el concepto de la pre-condición de secuencia y dado este ejemplo, en el caso de a_{t3}, su $\mathbf{s_3 = a_{t2}}$ **and** $\mathbf{a_{t1}}$. Las otras dos, a_{t1} y a_{t2}, tienen su correspondiente pre-condición de secuencia a *true*, ya que según esta descripción, previamente no ha tenido que realizarse ninguna acción transaccional antes que ellas.

Todas las pre-condiciones asociadas a una acción transaccional deben evaluarse y ser ciertas antes de poder iniciarse dicha acción.

5.7 Gestión de las transacciones: Transaction Manager.

Ya que una transacción es un caso especial de acción, hemos creado un agente nuevo, llamado ***Transaction Manager*** (*TM*), para que se encargue de la gestión relacionada con la ejecución de una transacción y para poder verificar propiedades de las transacciones y solucionar los problemas vistos anteriormente.

Mientras se está realizando una transacción, el resultado de las acciones transaccionales que la componen no debería registrarse en la

historia del agente siguiendo el procedimiento que hemos descrito al principio para cualquier acción. La razón es sencilla, si la transacción se interrumpe, el agente padre del que la realiza no debe tener conocimiento de ninguna ocurrencia de acción realizada dentro de la transacción. Esto garantiza las propiedades de aislamiento, atomicidad y consistencia de las transacciones.

5.7.1 Definición del agente Transaction Manager.

Para conseguir gestionar de una forma independiente las transacciones, hemos diseñado un agente especial que se crea cuando se inicia una transacción y se destruye cuando la transacción ha finalizado o bien se ha interrumpido. A este agente lo denominamos *Transaction Manager* (*TM*). Habrá un *Transaction Manager* por cada transacción iniciada dentro de un agente.

La estructura y funcionamiento de este agente, *Transaction Manager*, es equivalente al del agente *Controller* encargado de controlar el funcionamiento del agente donde se crea. Utiliza una arquitectura *blackboard* y almacena temporalmente las ocurrencias de las acciones transaccionales, de tal forma que se almacena un estado parcial del agente en una historia que es gestionada únicamente por el *Transaction Manager*. Se consigue así cumplir la propiedad de aislamiento. Si la transacción es interrumpida, esta "historia de la transacción" se pierde y conseguimos que el estado del sistema sea el mismo que antes de iniciar la transacción (realmente, todavía no se ha hecho ninguna modificación en su estado). De esta forma logramos que se cumplan las propiedades de atomicidad y consistencia. Si la transacción se completa con éxito, se guarda la ocurrencia de la transacción en la historia del agente padre y el agente *Transaction Manager* finaliza. Así incorporamos el cumplimiento de la propiedad de durabilidad o permanencia.

Cuando llegue al sistema un estímulo de inicio de transacción, el *Controller* se encargará de crear un agente *Transaction Manager* para llevar a cabo dicha transacción. Por cada transacción definida en el agente, el *Controller* almacena en una tabla independiente e interna la relación entre los estímulos de inicio de transacción y las transacciones correspondientes. De esta forma sabe cómo debe actuar cuando le llegue un estímulo correspondiente a una transacción y lo diferencia de un estímulo relacionado con otra acción (es una simple consulta en una tabla). Esta creación del TM involucra la selección de la información necesaria por el *Transaction Manager* para realizar su tarea, es decir, para poder activar a los agentes involucrados, almacenar las ocurrencias de las acciones transaccionales finalizadas y de los estímulos recibidos y realizar las comprobaciones de las pre-condiciones de las acciones transaccionales.

La información que necesita el *Transaction Manager* tanto para activar a los agentes como para evaluar las pre-condiciones es una lista que relaciona los agentes involucrados, las acciones transaccionales que realizan, las ocurrencias de las acciones y los estímulos que se encuentran en la definición de las distintas pre-condiciones asociadas a cada acción transaccional.

Cuando un agente ejecuta una acción transaccional (debido a que fue activado en el contexto de una transacción y por el *Transaction Manager* correspondiente) envía la ocurrencia de dicha acción al *Transaction Manager* (no al *Controller*) quien almacena dicha ocurrencia y comprueba si es parte de la pre-condición de otra acción transaccional (ver figura). En este caso, enviaría una señal de activación al agente adecuado.

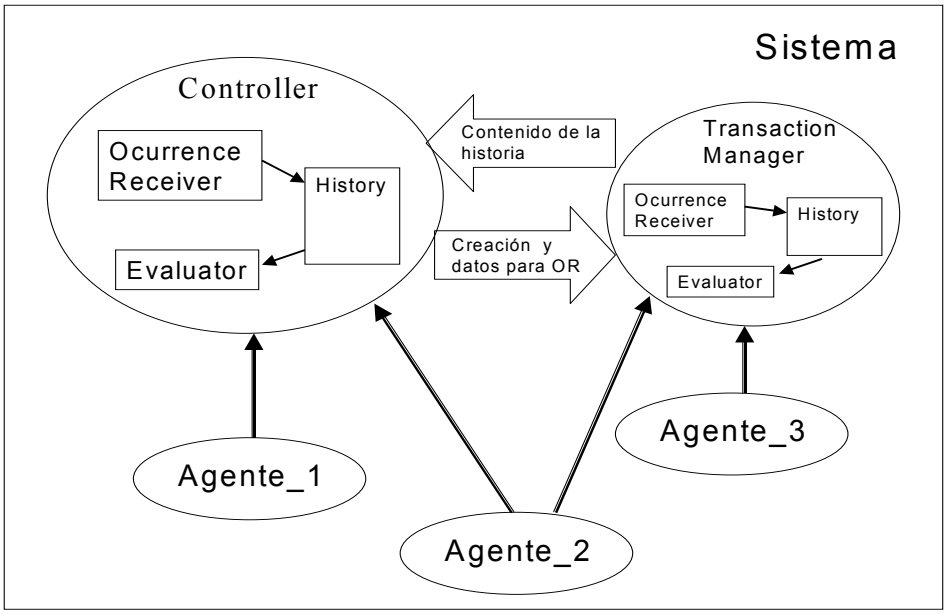


Figura 5.15 Relación entre Controller y Transacction Manager

Nótese que la estructura y funcionamiento de un *Transaction Manager* es exactamente igual que la de un *Controller*, salvo por la información que manejan y por el hecho de que un *Transaction Manager* desaparece del sistema cuando la transacción ya ha finalizado (con o sin éxito). Por tanto, un *Transaction Manager* tiene un *Ocurrence Receiver* y un *Evaluator* propios que hacen que su trabajo sea posible. Lógicamente entre ellos hay una comunicación. Primero, el *Controller* que es quien crea a un *Transaction Manager* debe darle toda la información necesaria para que pueda construirse sus tablas internas en base a las cuales se realizará la activación de los agentes que llevarán a cabo la transacción. Es decir, se selecciona parte de la información que gestiona el agente *Ocurrence Receiver* del *Controller* para que sea utilizada por el *Ocurrence*

Receiver del *Transaction Manager*. Segundo, el *Transaction Manager*, si la transacción ha tenido éxito (se ha podido completar) debe pedirle al *Controller* que almacene la ocurrencia de la transacción en su historia. Si no finaliza la transacción, simplemente *Transaction Manager* desaparece y con él todo lo que haya ocurrido durante la ejecución parcial de la transacción.

En la definición de una transacción pueden aparecer otras transacciones, por tanto se permite que existan transacciones anidadas. Dentro de un agente se inicia la ejecución de una transacción. Si esta transacción contiene en su definición otra transacción, la segunda se llevará a cabo por un agente hijo y se resolverá en el contexto de éste. Por tanto, será el *Controller* del agente hijo quien cree el TM adecuado para gestionarla. Para el agente padre lo único importante es que se realice la transacción que él ha iniciado y le es trasparente lo que ocurra dentro de ella. Este proceso de transacciones anidadas se corresponde a una activación en cascada (descendente) por la jerarquía de agentes.

5.7.2 Activación de las acciones transaccionales.

Las precondiciones asociadas a las acciones transaccionales están involucradas en dos historias distintas: la historia funcional y la historia gestionada por el TM generado para tratar dicha transacción. Primero veremos cómo resolver el proceso de activación de las acciones transaccionales de forma que se resuelvan los conflictos que aparecen. Después, veremos cómo contemplamos en el modelo propuesto la ejecución concurrente de transacciones.

5.7.2.1 Proceso de activación de las acciones transaccionales.

Como hemos dicho anteriormente, la pre-condición de una acción transaccional a_{ti} es la unión de $\mathbf{p_i + s_i + t_i}$. La comprobación de si se puede ejecutar a_{ti} o no podría generar un conflicto entre el *Controller* y el *Transaction Manager* ya que las $\mathbf{p_i}$ se deben evaluar observando el contenido de la historia del agente (gestionada por el *Controller*) y el resto se evalúan según la información almacenada en la historia de la transacción (a cargo del *Transaction Manager*). Como es el agente *Transaction Manager* quien se encarga de la evaluación de las pre-condiciones de una acción transaccional, tendríamos que permitir que accediera a la historia del agente, lo cual iría en contra de la filosofía de independencia y ocultación de la información que hemos estado aplicando, con el fin de conseguir agentes autónomos e independientes y facilitar la evolución.

Para resolver este problema, seguimos el siguiente proceso: antes de iniciar una transacción, el *Controller* comprueba si, además de la pre-condición $\mathbf{p_t}$ asociada a la transacción, las $\mathbf{p_i}$ de las acciones transaccionales que la componen se cumplen y solamente si se cumplen, inicia la transacción. Por ejemplo, si tenemos una transacción:

A = a ; b

se comprueba la pre-condición $\mathbf{p_t}$ de *A*, la $\mathbf{p_i}$ de la acción transaccional *a* y la $\mathbf{p_i}$ de la acción transaccional *b*. Sólo si todas estas pre-condiciones se cumplen, entonces se permitirá que comience la transacción. A partir de este momento, las pre-condiciones a evaluar se comprobarán sobre la historia transaccional únicamente. Lógicamente, no tienen que cumplirse todas las pre-condiciones **p** para que se pueda iniciar una transacción. Esto depende de los operadores utilizados en la definición de la transacción. Si en lugar de *A = a ; b* fuera *A = a | b*, sólo es necesario que se cumpla la $\mathbf{p_i}$ de *a* o la $\mathbf{p_i}$ de *b*. En caso del operador || sí deben cumplirse todas.

El propio *Controller*, dada la definición de la transacción, sabe que pre-condiciones tiene que evaluar:

- Si se encuentra *a ; b* o *a || b* deberán ser ciertas las pre-condiciones $\mathbf{p_i}$ de ambas.

- Si se encuentra *a | b* sólo tiene que ser cierta una pre-condición $\mathbf{p_i}$ de alguna de estas acciones. En este caso, puede ocurrir que la transacción se inice porque se había cumplido la $\mathbf{p_i}$ de *a* y después resulta que se ejecuta *b* (y no *a*). El *Evaluator* del TM debería conocer este hecho, es decir, debería saber si *b* en el momento de iniciar la transacción cumplía su $\mathbf{p_i}$ o no. En este caso, si sólo se cumple una de las $\mathbf{p_i}$ de los términos involucrados, almacenamos dicha información y la expresión se puede convertir en una más simple (quitamos la acción de la expresión *or* cuya $\mathbf{p_i}$ no haya sido cierta). Por ejemplo, supongamos la siguiente transacción:

  A = (a | b) ; c

  si la transacción se ha activado y sólo la $\mathbf{p_i}$ de *b* es cierta, para el TM la información que almacenará sobre la definición de dicha transacción será la correspondiente a:

  A = b ; c

- El orden de evaluación es de izquierda a derecha salvo si se utilizan paréntesis para modificarlo.

Por ejemplo, supongamos una transacción un poco más compleja:

$$A = (a \ ; \ b) \ || \ (c \ | \ d)$$

Supongamos que la transacción *A* sólo tiene en su pre-condición que se haya recibido un estímulo. Cada acción transaccional tiene asociada una pre-condición $\mathbf{p_i}$ que llamaremos $\boldsymbol{p_a}$, $\boldsymbol{p_b}$, $\boldsymbol{p_c}$ y $\boldsymbol{p_d}$ para las acciones transaccionales *a*, *b*, *c* y *d* respectivamente. Cuando el *Controller* recibe el estímulo que haría que la transacción se iniciara, no lo permite hasta que se cumpla lo siguiente:

$$(p_a \text{ and } p_b) \text{ and } (p_c \text{ or } p_d)$$

es decir, la pre-condición $\mathbf{p_i}$ de *a* y la de *b* deben cumplirse y también una de las pre-condiciones $\mathbf{p_i}$ de *c* o de *d*.

Hemos creado un algoritmo para pasar de una definición en TDL a una expresión en LPT que expresa los requisitos de evaluación de las distintas pre-condiciones $\mathbf{p_i}$ (ver capítulo 6).

Cuando se evalúan las pre-condiciones $\mathbf{p_i}$ a *true*, se crea el *Transaction Manager* que gestionará la transacción, se almacena un estímulo de inicio de la transacción en la historia del TM y ésta comienza a llevarse a cabo. El estímulo de inicio de transacción permite que las primeras acciones transaccionales se activen y el TM se encarga de la evaluación de las pre-condiciones ***s*** y ***t*** de cada acción transaccional. Cuando la transacción finalice, se almacena la ocurrencia de la realización de la transacción en la historia del agente donde se inició.

5.7.2.2 Transacciones concurrentes.

En un SS puede ser útil poder ejecutar varias transacciones concurrentemente, al igual que se puede ejecutar cualquier número de acciones simples concurrentemente (existe concurrencia inter-agente). Para cada una de ellas existirá un *Transaction Manager* encargado de gestionarla.

En la figura siguiente podemos ver un sistema donde existen cinco agentes, el *Controller* y dos *Transaction Manager*, TM_1 y TM_2, creados al iniciarse dos transacciones en el agente sistema. Se observa que en ambas transacciones están involucrados algunos de los agentes. Por ejemplo, en la transacción gestionada por TM_1 trabajan Agente 5 y Agente 4. Los agentes que realizan acciones no transaccionales, por ejemplo el Agente 3, se comunican con el *Controller* tanto para que almacene las ocurrencias de sus acciones como para comprobar si

pueden o no realizar otras acciones. Sin embargo, cuando se inicia la transacción, los agentes que realizan acciones transaccionales (involucradas en dicha transacción) enviarán al *Transaction Manager* correspondiente las ocurrencias de la realización de dichas acciones y preguntarán a éste si ciertas acciones transaccionales pueden ser realizadas según el estado (parcial) del sistema respecto a dicha transacción. Nótese que un determinado agente (en el ejemplo el Agente_2) puede realizar tanto acciones transaccionales como acciones no transaccionales y su comportamiento para un tipo de acciones y para otro será distinto en cuanto a quién le solicitará los distintos servicios.

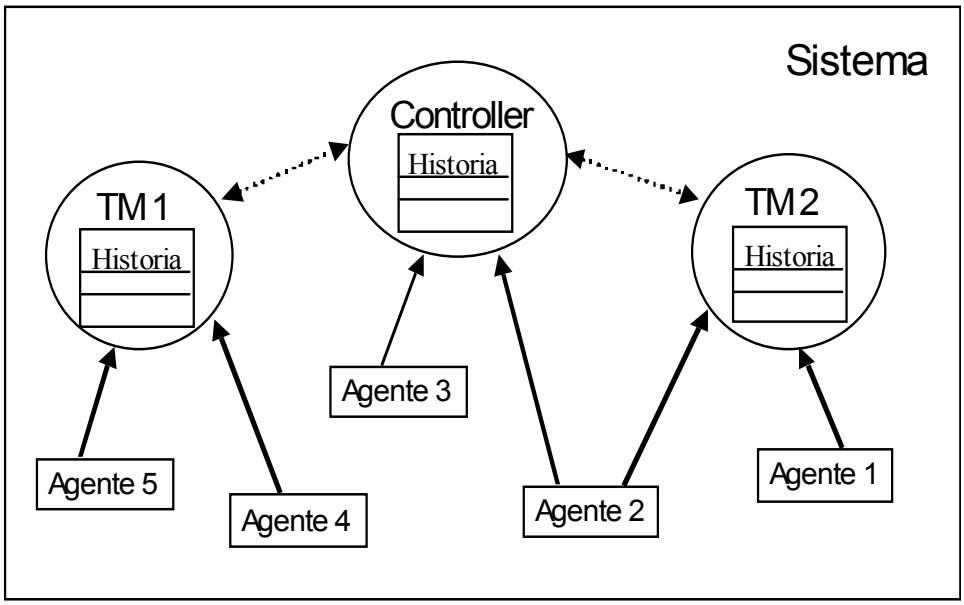


Figura 5.16 Sistema donde se han iniciado dos transacciones

Cada agente sabe quién lo ha activado, es decir, si solicitó la evaluación de la pre-condición de una acción al *Controller* porque fue éste quién le indicó que la evaluara, sabe que cuando la realice debe solicitar el almacenamiento de la ocurrencia asociada a dicho *Controller*. Pero si fue el *Transaction Manager* el responsable de su activación, será éste quien almacene dicha ocurrencia, es decir, el agente estará realizando la acción en el contexto de una determinada transacción.

La línea discontinua entre el *Controller* y cada *Transaction Manager* representa la comunicación establecida entre ellos cuando el *Controller* crea al *Transaction Manager* al iniciar una transacción. Esta comunicación, por un lado significa el envío de información del *Controller* al *Transaction Manager* para que pueda realizar su trabajo. Por otro lado y en el caso de que la transacción finalice, representa el

envío de la ocurrencia de la transacción en la historia general de este agente.

5.7.3 Interrupción de una transacción.

Hasta ahora no hemos abordado las causas que pueden provocar que una transacción se interrumpa y no pueda finalizar y cómo y cuándo se puede tomar la decisión de interrumpir una transacción. Dado que las pre-condiciones ***p*** deben cumplirse antes de poder iniciar la transacción, nos centraremos en el estudio de cómo pueden influir las pre-condiciones ***s*** y ***t*** en la terminación anormal de una transacción.

Recordemos que la pre-condición ***s*** especifica el orden de activación de una acción transaccional, por tanto, aunque en un momento dado no sea cierta, se espera que en un futuro se cumpla (salvo si la transacción se interrumpe). El hecho de que en el momento en el que se evalúe sea falsa no aporta ninguna información que determine que la transacción no debe continuar realizándose. Esto está influenciado por el modo en el que se activan los agentes, ha ocurrido algo que lleva al *Transaction Manager* a evaluar la pre-condición de una determinada acción transaccional pero el estado del agente no es el adecuado en ese momento y por tanto no se inicia dicha acción transaccional por el momento. Sin embargo, el agente todavía puede cambiar su estado (respecto a ***s***), y, en ese caso, la transacción puede continuar su ejecución. Esto nos lleva a la conclusión de que la evaluación de ***s*** no es determinante para decidir la interrupción de una transacción.

La pre-condición ***t*** se utiliza para imponer una condición especial que hace que se pueda realizar una determinada acción transaccional. Si fuera falsa, nos indica que la acción transaccional no se puede realizar, ni ahora ni en el futuro. Hay que tener en cuenta que lo que tengamos en ***t*** dependerá de lo que ocurra dentro de la transacción antes de que la acción transaccional asociada a dicha pre-condición ***t*** tenga la oportunidad de ejecutarse. Por tanto, si se va a evaluar ***t*** es porque ya ha tenido que ocurrir algo en el agente que implique definitivamente que ***t*** es cierta o es falsa en base al estado actual del agente, y esta evaluación no va a cambiar. Por tanto, esta condición hace las funciones de una durante-condición. Para que esto sea cierto, se debe imponer un orden a la evaluación de las pre-condiciones de las acciones transaccionales:

1. Primero se evalúa la pre-condición ***s*** asociada.

2. Si ésta es cierta, entonces se evalúa ***t***.

3. Si ***t*** ahora es falsa entonces debe interrumpirse la transacción. Es imposible que cambie el valor de ***t*** aunque el agente siga funcionando ya que depende de lo ocurrido hasta el instante en el que ***s*** se evalúa a *true*.

4. Si ***t*** es cierta, se inicia la realización de la acción transaccional (se devuelve *true* al agente que solicitó la evaluación y, por tanto, se comenzará a ejecutar la acción transaccional).

Dado este estudio, podemos concluir que el *Transaction Manager* interrumpirá la transacción cuando evalúe alguna pre-condición ***t*** y ésta sea falsa. En ese momento finalizará y el sistema quedará en el mismo estado que antes de iniciar la transacción ya que lo ocurrido durante la transacción desaparecerá junto con el *Transaction Manager*. Su ejecución parcial es totalmente transparente para el sistema y sus agentes.

5.7.4 Terminación normal de una transacción.

Hay que especificar cómo sabe el TM que la transacción ha finalizado. Cuando lo haga, generará una ocurrencia asociada a dicha transacción que se almacenará en la SFH del agente que la inició ya que todo TM conoce al *Controller* que lo creó.

Con el fin de que el TM sepa cuando ha finalizado una transacción con éxito, a la definición de la transacción dada por el desarrollador se le añade un ";" y una acción especial que llamamos *terminar*. Cuando se realicen la última o últimas acciones transaccionales y sus ocurrencias sean almacenadas por el OR del TM, se verá que se puede activar la acción *terminar*. En ese momento el TM puede generar la ocurrencia de la transacción y enviarla al *Controller*.

Además de esto, si una transacción finaliza con éxito, las ocurrencias de las acciones transaccionales que están almacenadas en la historia del TM podrían o no almacenarse en la SFH del agente que inició la transacción. Por tanto se puede proceder de dos formas:

- Hacer un volcado completo de la historia del TM en la SFH del agente responsable de la transacción. Puede ser interesante si en cualquier momento queremos reconstruir un estado anterior del sistema. En este caso, necesitamos toda la información, ya que el instante de tiempo elegido puede localizarse en medio de la realización de una transacción.

- Sólo almacenar la ocurrencia de la transacción porque no interesa saber qué ha ocurrido durante una transacción sino el resultado

final. Esta opción supone una mayor eficiencia en un doble sentido: por un lado ahorramos en espacio (por cada transacción sólo tenemos que almacenar una ocurrencia en la SFH). Por otro lado minimizamos el tiempo dedicado tanto a la búsqueda por la historia como en realizar el volcado de cada una de las ocurrencias y estímulos producidas durante una transacción. Además, puesto que una de las características propias de las transacciones es el hecho de que aparezcan como acciones atómicas, el resultado de sus acciones transaccionales no debe ser útil para el resto de sus agentes hermanos.

Para nuestro modelo hemos elegido la segunda opción. Por un lado pensamos que realmente al agente sólo le interesa conocer que se ha realizado la transacción, por ello se define así y no como un conjunto de acciones que se pueden o no realizar en el SS. Por otro lado, la eficiencia en el funcionamiento del SS también es un factor importante a tener en cuenta.

5.7.5 Evolución en la gestión de transacciones.

Un *Transaction Manager* solo existe mientras se realiza la transacción y por tanto, aunque podríamos pensar en la posibilidad de que éste pueda evolucionar, no tiene sentido, ya que se crea en el momento de iniciar una transacción y desaparece cuando ésta finaliza normal o anormalmente. Además, los cambios estructurales sólo se realizan cuando el SS está parado y esto sólo ocurre cuando todas las acciones que se iniciaron en el SS han finalizado. En el caso de realizar cualquier cambio en la definición de la transacción, éste no afecta a la transacción que ya ha sido realizada y cuando se inicie otra vez, el *Transaction Manager* nuevamente creado ya contemplará los cambios en la definición de la transacción.

Los que sí tienen sentido son los cambios que se pueden realizar sobre la transacción:

- Podemos eliminar o añadir una acción transaccional. Ambas operaciones involucran una redefinición de la pre-condición ***s*** de una o más acciones transaccionales que formen parte de la misma definición de transacción.

- Se pueden modificar las pre-condiciones ***p*** de la transacción y/o de las acciones transaccionales. Siempre se tendrá que garantizar que esta modificación es consistente con lo que hay en el SS actualmente. Por ejemplo, no puede ser parte de una pre-condición una acción que no exista en el SS.

- Se pueden redefinir las pre-condiciones ***t*** de las acciones transaccionales. Igual que en el caso anterior hay que tener cuidado con crear inconsistencias.

- También es viable añadir o borrar una transacción completa. La operación de modificación de una transacción es la sustitución completa de dicha definición, es decir, equivaldría a realizar secuencialmente una operación de borrado y después una operación de adición de la definición.

Todas estas operaciones se estudian detalladamente para que su ejecución cumpla siempre las restricciones de integridad del sistema y garantice la propiedad de consistencia definida para las transacciones. Además, todas estas modificaciones se propagan hasta el *Controller* quien actualiza sus tablas internas con la información que relaciona agentes/acciones/pre-condiciones y empieza a actuar según las nuevas modificaciones. En el capítulo 7 se profundiza sobre los aspectos de evolución.

Esta evolución también es llevada a cabo por el Metasistema y por el Sistema Genético del agente donde se encuentra la transacción que se pretende modificar, borrar o añadir.

5.7.6 Propiedades *ACID* en nuestro modelo.

Es interesante, para terminar este punto, resaltar cómo están presentes cada una de las cuatro propiedades que deben cumplirse en un SS donde existen y se ejecutan transacciones. Iremos explicando cada una de las propiedades y finalmente mostraremos una tabla explicativa, tabla 1, donde se resume la parte de nuestro modelo que garantiza cada propiedad.

La propiedad de *atomicidad* consiste en garantizar que la transacción se realice como una acción indivisible, atómica. Está presente en nuestro modelo gracias a la existencia del agente *TM*. Al iniciarse una transacción, se crea un *TM* para su gestión y éste actúa como un agente *Controller* aunque sólo para las acciones transaccionales. Existe un *TM* por cada transacción y ésta no será visible en el SS hasta que se haya realizado con éxito, por tanto, aparecerá ante el SS como una acción atómica.

La propiedad de *consistencia* especifica que una transacción transforma el SS de un estado consistente a otro. Para garantizarla, el *Controller* no almacena ninguna ocurrencia de acción transaccional si la transacción no ha finalizado y lo ha hecho con éxito. De esta forma, evitamos que el SS pueda quedar en un estado inconsistente si, por ejemplo, la

transacción se interrumpe pero se han realizado varias acciones transaccionales que hayan variado su estado.

La propiedad de *aislamiento* indica que los resultados parciales de una transacción no deben ser visibles hasta que la transacción finalice. Esto se consigue almacenando las ocurrencias de acciones transaccionales en la historia del *TM* a la cual no puede acceder ningún agente que no esté trabajando para la transacción.

La propiedad de *durabilidad* o *permanencia* se basa en la idea de que, una vez que la transacción ha finalizado, los cambios realizados en el sistema deben ser permanentes. Esta propiedad se garantiza en nuestro modelo ya que, una vez que la transacción ha finalizado, el *TM* genera y envía a almacenar la ocurrencia de la transacción en la SFH del *Controller* que lo creó.

En la siguiente tabla se resumen las propiedades ACID y su presencia en el modelo que presentamos.

PROPIEDAD	**Presencia en el modelo propuesto**
Atomicidad	Se crea un *TM* por cada transacción y el resultado de la transacción no se conoce hasta que ésta finalice con éxito
Consistencia	El estado del sistema queda inalterado (en la *SFH* no se almacena nada) mientras no se complete con éxito la transacción.
Aislamiento	Mientras se realiza la transacción, los agentes no involucrados en ella no saben que se está llevando a cabo
Durabilidad	Cuando la transacción finaliza con éxito, se almacena la ocurrencia asociada a la transacción en la *SFH*

Tabla 5.1 Propiedades ACID en el modelo propuesto

5.8 Conclusiones.

En este capítulo se ha expuesto el modelo que proponemos y que permite describir sistemas software basados en agentes capaces de evolucionar. Puesto que se parte del modelo MEDES, se han expuesto brevemente los conceptos que éste incorpora y que influyen decisivamente en nuestro trabajo.

Partimos del hecho de que la funcionalidad de un sistema software viene determinada por su estructura. Por ello, hemos explicado detalladamente los distintos elementos arquitectónicos presentes en la estructura de un sistema software y sus relaciones.

Se han presentado los distintos problemas que surgen en el funcionamiento de los agentes: concurrencia, activación, evolutividad y ejecución de transacciones. Se ha propuesto una solución consistente en la incorporación de un agente especial denominado *Controller*. Este agente, a su vez, consta de dos sub-agentes: el agente *Evaluator* y el *Ocurrence Receiver*. El primero de ellos se encarga de la evaluación de las pre-condiciones de las acciones de los agentes y determina si, según el estado actual del sistema, pueden o no realizarse. El segundo se encarga de mantener la información de la historia de las ocurrencias de las acciones realizadas y de comunicar a los agentes (mecanismo de notificación de eventos) que tienen probabilidad de realizar una o más acciones. Generalizando lo anterior se ha definido un patrón de diseño llamado PDN (*Precondition Dynamic Notifier*).

Respecto a las transacciones, se ha hecho un estudio de éstas y de cómo se incorpora este concepto en el modelo. Se ha establecido la relación entre las transacciones y la jerarquía de agentes que existen en el sistema software. Para la gestión de las transacciones se ha introducido un elemento nuevo, un agente, llamado *Transaction Manager*. Este agente se crea en el sistema cuando se inicia una transacción y desaparece cuando la transacción ha finalizado con o sin éxito. Su comportamiento y estructura es similar al de un agente *Controller*. Se garantiza que el estado de los sistemas software, antes y después de la ejecución de una transacción parcialmente ejecutada y que no ha finalizado normalmente, es el mismo. Es decir, los resultados parciales de las acciones que componen en una transacción no se guardan.

CAPITULO 6

Funcionamiento de los Agentes

En este capítulo vamos a centrarnos en el funcionamiento de los agentes. Puesto que este funcionamiento está dirigido por el agente *Controller* que se encuentra en cada agente del sistema software, vamos a profundizar en el comportamiento de sus elementos constitutivos: el agente *Evaluator* y el agente *Ocurrence Receiver,* además de una historia que representa el estado del agente en un instante de tiempo concreto.

Cuando un agente realiza una acción, la ocurrencia de realización de ésta se almacena en la historia de su agente padre. Por tanto, cada agente conoce al *Controller* de su agente padre, con el que tendrá que comunicarse y a quien tendrá que solicitarle los servicios necesarios. Esta comunicación es explícita. Si un agente no tiene agentes hijos, su *Controller*, aunque existe, no realiza ningún trabajo y su historia no tendrá ningún elemento. Sin embargo, si en un momento dado, este agente hasta ahora simple, evoluciona y se crean en él otros agentes, su *Controller* comenzará a funcionar y su historia dejará de estar vacía.

Cada *Controller* proporciona los siguientes servicios a los agentes hijos del agente donde se encuentra:

- Almacenar la ocurrencia de una acción realizada o un estímulo generado por un agente hijo en la SFH de su padre.

- Evaluar la pre-condición *p* asociada a una acción simple o compleja sobre la información mantenida en la SFH.

- Obtener los resultados de una consulta como una lista de valores que cumplen una determinada condición.

- Notificar a un agente que se ha recibido una ocurrencia o un estímulo que puede provocar la activación de una (o más) de sus acciones.

La comunicación entre un agente y el *Controller* que le responderá ante sus peticiones de servicio se realiza a través de un mecanismo de mensajes. Cada mensaje se construye con el nombre del servicio requerido y los parámetros o argumentos necesarios que dependen del tipo de servicio.

En los siguientes puntos, vamos a concretar cómo se realizan los distintos servicios centrándonos en el elemento que los lleva a cabo, ya que el *Controller* delega en sus dos sub-agentes la realización de éstos. Comenzaremos estudiando un poco más a fondo la actuación del *Ocurrence Receiver* y después del *Evaluator*. A continuación nos centramos en el agente *Transaction Manager*. Para finalizar, presentamos un lenguaje de descripción de sistemas basados en agentes.

6.1 Ocurrence Receiver (OcuR).

Este agente se encarga básicamente de la recepción y almacenamiento de ocurrencias de acciones y estímulos en la SFH e interviene en la activación de los agentes, ya que les notifica cuándo tienen posibilidades de realizar una acción y cuál es ésta. Además se encarga de iniciar las transacciones.

Con el fin de almacenar la información necesaria para la activación de los agentes hijos del agente donde se encuentra, necesita mantener los siguientes tipos de tablas:

- T*abla de Agentes:* contiene una entrada por cada uno de los agentes hijos del agente donde se encuentra el *OcuR*. Cada entrada mantiene un identificativo del agente hijo y una referencia a la tabla de acciones que realiza cada hijo. Cada elemento tiene dos partes:

  tabla_agentes = { nombre_agente + ref.Tabla_Acciones }

- *Tabla de Acciones*: existe una entrada por cada acción que es capaz de realizar un agente hijo. En cada entrada aparece el nombre de la acción y una referencia a la tabla de pre-condiciones correspondiente. Cada entrada es de la forma:

  tabla_acciones = { nombre_acción + ref.Tabla_Pre-condiciones }

  donde *nombre_acción* se corresponde con cualquier clase de acción, tanto simple como compleja o transacción.

- *Tabla de Pre-condiciones*: existe una entrada por cada nombre de acción y/o de estímulo que aparece en la pre-condición asociada a la acción que describe. Recordemos que una pre-condición es una expresión formada por operadores y nombres de acciones y/o estímulos. Cada acción de una tabla de acciones tiene asociada su propia tabla de pre-condiciones. Sin embargo, si existen agentes hermanos que realicen la misma acción (igual nombre), sólo tiene que existir una única tabla de pre-condiciones asociada a éstas ya que partimos de la premisa de que deben ser idénticas. Por ello, llevamos un contador de referencias que indica cuantas entradas de tablas de acciones apuntan a cada tabla de pre-condiciones.

  tabla_pre-condiciones = contador_referencias +
  { nombre_accion | nombre_estímulo }

  Cuando se crea la tabla de pre-condiciones, al contador de referencias se le asigna el valor 1. Si en algún momento se crea otra acción igual en algún agente hermano, se comprueba que ya está creada la tabla de pre-condiciones y se incrementa en 1 el contador de referencias de ésta. Al borrar una acción, sólo se borra la tabla de pre-condiciones a la cual referencia si el contador de referencias está a 1, sino, sólo se decrementa su valor.

  En la figura siguiente podemos ver estas tablas que después veremos en el ejemplo presentado en el capítulo 8. En ella se puede apreciar la estructura del agente donde se encuentra el *Controller*: sus agentes hijos y las acciones que éstos realizan.

  - En la tabla de agentes se mantiene la información de sus tres agentes hijos: *Empleado1*, *Empleado2* y *E_Taller.*
  - Cada agente hijo tiene su propia tabla de acciones.
  - *Empleado1* y *Empleado2* realizan las acciones *alquilar* y *Lavar.* Por ello, sus entradas de la tabla de acciones referencian a la misma tabla de pre-condiciones.
  - *E_Taller* realiza las acciones *Comprobar* y *pintar.* Cada una referencia su propia tabla de pre-condiciones.

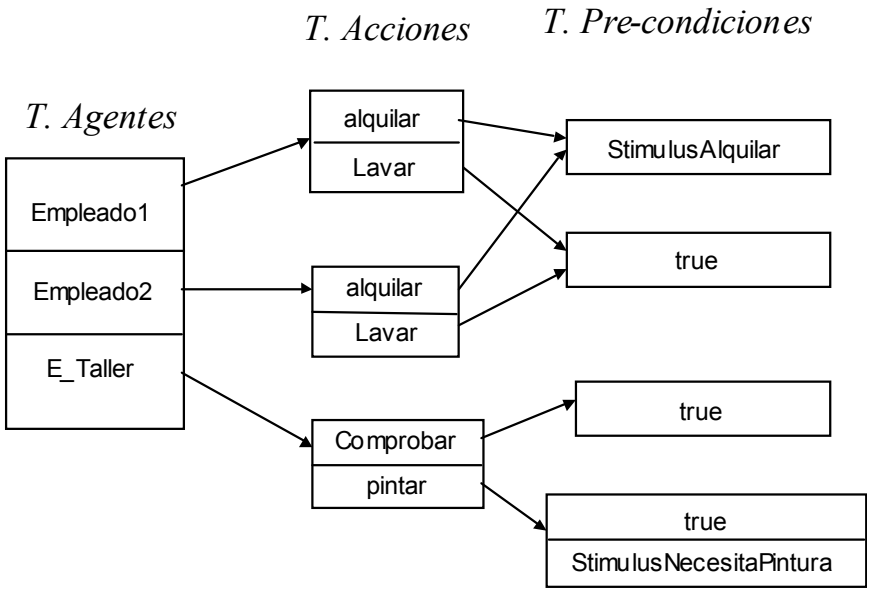


Figura 6.1 Tablas del OcuR para la activación de agentes

Relacionada con las transacciones, mantiene la siguiente tabla:

- *Tabla de Transacciones*: cada entrada describe una transacción o acción compleja llevada a cabo por un agente hijo. Los elementos de una entrada son: el nombre de la transacción, el nombre del estímulo que la inicia, la definición de la transacción (en el lenguaje TDL) y una lista con los nombres de sus acciones transaccionales junto con sus pre-condiciones ***t*** (recordemos que sus pre-condiciones s se obtienen de la definición). Cada entrada es de la forma:

  *tabla_transacciones = { nombre_transacción + nombre_estímulo + definición + { nombre_acción + **t** } }*

Además, comparte con el *Evaluator* el acceso a la siguiente tabla:

- *Tabla de Acciones Iniciadas*: en cada entrada se almacena la información de las acciones que han comenzado a realizar los agentes hijos pero que todavía no han finalizado.
  tabla_acciones_iniciadas = { nombre_accion }

A continuación vamos a centrarnos en las dos funciones principales que lleva a cabo el agente OcuR. La primera de ellas es la recepción y almacenamiento de las ocurrencias y estímulos que son enviados a la historia funcional del agente. La segunda es la activación de los agentes. Finalmente, comentamos su relación con la activación de las transacciones o acciones complejas.

6.1.1 Recepción y almacenamiento de ocurrencias y estímulos en la SFH.

La SFH es una estructura de datos donde se almacenan dos tipos de elementos:

- Ocurrencias de acciones: se generan al finalizar la realización de una acción y almacenan en sus atributos la información necesaria para tener constancia de ello.
- Estímulos: son generados y enviados por los usuarios y/o por los agentes con el objeto de iniciar una determinada acción. También tienen atributos.

Además, cada elemento introducido en la SFH se sitúa en una posición determinada según el orden establecido por el tiempo de realización de la acción o por el tiempo de envío del estímulo. Usamos ese valor para indexar la SFH y después poder realizar búsquedas eficientes.

Como puede darse el caso de que dos acciones y/o estímulos tengan el mismo tiempo asociado (debido a la concurrencia existente en la actividad de los agentes), podemos tener varias entradas que se correspondan al mismo valor de índice. Por este motivo se utiliza para su implementación una tabla *hash* con una función de acceso muy sencilla, que tomará como valor el tiempo de realización.

Para esta función, el OcuR proporciona en su interfaz la operación *almacenar(elemento)*. Cuando se activa esta acción se siguen los siguientes pasos:

1. Comprueba si la SFH está bloqueada: el agente *Evaluator* puede estar consultando la SFH y, para que no existan problemas de exclusión mutua, antes de acceder a su información, la ha bloqueado.

   1.1 Si lo está, almacena el elemento en una cola de espera que implementa internamente. Cuando el *Evaluator* va a iniciar una evaluación y, por tanto, a acceder a la SFH, antes de bloquearla pide al OcuR que vuelque los elementos contenidos en la cola de espera y después la bloquea. Esto se hace así para que la evaluación se realice correctamente, con una SFH que represente realmente el estado actual del agente.

   1.2 Si no lo está, bloquea la SFH y almacena el elemento recientemente recibido en su posición correcta siguiendo el orden temporal que hemos comentado. Hay que tener en cuenta que no es simplemente insertar al final de la historia sino que puede que una

ocurrencia de acción realizada antes que otra llegue después al OcuR.

2. Una vez introducido, desbloquea la SFH (si lo hizo).

El bloqueo/desbloqueo de la SFH se realiza mediante un mecanismo de sincronización como son los semáforos o el paso de mensajes.

Aparte de la función de almacenamiento, cada vez que llega una ocurrencia de acción, debe borrar la información que se mantiene indicando que esta acción está iniciada pero no terminada. Existe una tabla, la Tabla de Acciones Iniciadas, donde el *Evaluador* registra todas aquellas acciones que han comenzado a llevarse a cabo. El OcuR recorre esta tabla y elimina de ella la entrada correspondiente a la acción cuya ocurrencia se ha recibido. Como esta tabla es compartida con el *Evaluator*, su acceso también se realiza en exclusión mutua.

6.1.2 Activación de los agentes.

Esta función es un poco más compleja. No se activa de forma explícita, sino cada vez que reciba un elemento a almacenar en la SFH. Puesto que es éste a quien le llega la ocurrencia realizada o el estímulo recibido en un agente, es el más idóneo para comprobar a qué agentes les interesa que se les notifique este suceso.

Para ello, mantiene la relación de los agentes, las acciones y las pre-condiciones de éstas en las tablas de agentes, acciones y pre-condiciones que se comentaron anteriormente. Cuando se crea un agente, y antes de que éste comience a funcionar, se le proporciona esta información al *Controller* y éste al OcuR. Después, cada vez que se realice una modificación en la estructura del agente, se le envía la información necesaria para que la actualice. Siempre tiene que estar actualizada porque de ella depende el buen funcionamiento del agente.

La construcción de la estructura de tablas es simple. Al OcuR le llega la información de los agentes, sus acciones y las pre-condiciones de éstas. Crea una tabla de agentes donde existe una entrada para cada agente. Por cada agente hijo, construye una tabla de acciones con tantas entradas como acciones tenga en su interfaz. A su vez, cada acción tiene asociada una tabla de pre-condiciones.

El algoritmo a seguir para construir la tabla de pre-condiciones a partir de las pre-condiciones asociadas a cada acción se encuentra en la figura siguiente. Suponemos que la pre-condición consta de elementos, separados por caracteres en blanco, y que la operación *read(elemento)*

obtiene elementos completos (*tokens*) que son o bien operadores de la lógica, o paréntesis o nombres de ocurrencias y estímulos.

```
i:=0;
While (hay elementos sin leer en la pre-condición) {
   read(elemento);
   case (elemento) {
       '(': break;
       ')': break;
       operador: break;
       'not': { read (elemento);
                if (elemento == '(' )
                  then read (elemento) until (elemento == ')')
                 break; }
       default: { listaValidos[i]:= elemento;
                  i++; }
   } /* case */
}  /* while */
```

Figura 6.2 Algoritmo para construir una tabla de pre-condiciones

El algoritmo lee un elemento de la pre-condición. Si éste es un paréntesis abierto, un paréntesis cerrado o un operador distinto del *not*, continúa leyendo el siguiente elemento. Si es un operador *not* comprueba si el siguiente elemento es un paréntesis abierto, en tal caso sigue leyendo elementos hasta encontrar un paréntesis cerrado. Si no es ninguno de los operadores descritos, es un nombre de acción o de estímulo. En *listaValidos* vamos almacenando los nombres de las acciones (coinciden con los nombres de las ocurrencias) y de los estímulos que influyen en la activación de la acción asociada a la pre-condición.

Podemos observar que todas los nombres de acciones y estímulos que vayan precedidos por un operador *not* no se toman en cuenta, ya que si lo hiciéramos, al producirse las ocurrencias de dichas acciones o al recibir dichos estímulos podemos activar a un agente que realmente no tiene interés en ellas.

Por otro lado, es importante destacar que la construcción de estas tablas a partir de la información que se le suministra, se realiza y se mantiene automáticamente por el OcuR. Si, por ejemplo, se elimina un agente hijo, se busca en la tabla de agentes y se borra esa entrada (y con ella, las tablas a las que referencie). Si es una acción la que desaparece, buscamos el agente en cuestión en la tabla de agentes y accedemos a la tabla de acciones y borramos la entrada correspondiente a dicha acción. Estas acciones permiten que se realice el mantenimiento de la arquitectura dinámica de agentes que componen el sistema software.

Cuando al OcuR le llega una ocurrencia de acción o estímulo, consulta las tablas y determina qué agentes y con qué acciones pueden tener probabilidad de activarse. El OcuR le comunica a cada agente encontrado la acción o acciones que pueden tener éxito en la evaluación de sus pre-condiciones. Los agentes, ante esta noticia, solicitan la evaluación de las pre-condicies de cada una de esas acciones. Por ejemplo, si al OcuR le llega un *StimulusAlquilar*, consulta las tablas de la figura 6.1 y le comunica tanto al agente *Empleado1* como al agente *Empleado2* que pueden intentar activar su acción *alquilar*.

6.1.3 Transacciones.

Como hemos visto, en un agente pueden existir acciones complejas o transacciones. Toda transacción tiene asociado un estímulo de inicio cuyo nombre es la concatenación de la palabra *Stimulus* junto con el nombre de la acción (por convención).

El OcuR será responsable de conocer cuándo, al recibir un estímulo, se puede activar una transacción. Para ello, mantiene una tabla de transacciones donde se mantiene la información necesaria de cada transacción.

En la siguiente figura mostramos el proceso que se sigue cuando llega un estímulo de inicio de una transacción. Este estímulo se introduce a través de la interfaz de acción del agente padre, ya que este es responsable de realizar la transacción.

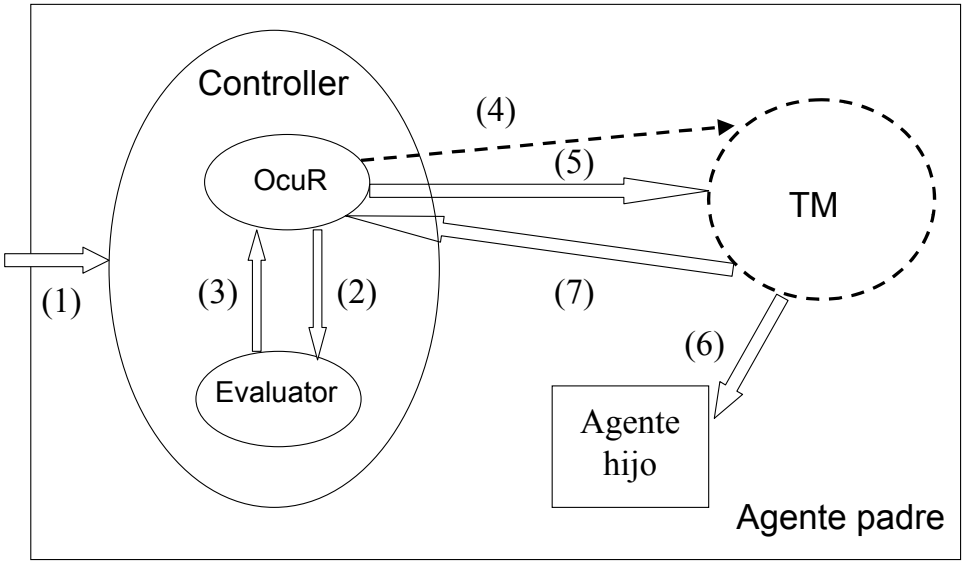


Figura 6.3 Proceso de inicio de una transacción

Cuando al *Controller* le llega un estímulo (paso 1 en la figura), este delega en su OcuR quien comprueba en la tabla de transacciones si es un estímulo de inicio de transacción. Si no lo es, procede como en los casos anteriores. Si lo es:

- Envía al *Evaluator* (2) la descripción de la transacción para que se encargue de evaluar las pre-condiciones *p* de las acciones transaccionales. El *Evaluator* le devolverá un *true* si puede iniciar la transacción o un *false* en caso contrario (3).
- Si es *false*, la transacción no se iniciará y será necesaria la llegada de un nuevo estímulo para que se realice.
- Si es *true*, se crea un *Transaction Manager* (TM) para gestionar la transacción (4).

El OcuR le suministra la suficiente información al TM (5) para que realice su trabajo. Este agente tiene igual estructura y funcionamiento que un *Controller,* aunque desaparece cuando la transacción finaliza normal o anormalmente.

La información que le pasa el OcuR al TM depende de la transacción, básicamente es:

- De sus tablas que relacionan los agentes/acciones/precondiciones, selecciona aquellos agentes y acciones involucrados en la transacción y se los envía al TM. El TM tendrá un OcuR que mantendrá unas tablas similares pero sólo de los agentes que realizan las acciones transaccionales que aparecen en la definición de la transacción.

- Le envía la información mantenida en la tabla de transacciones de la transacción a iniciar. En ella está la información propia la transacción y de las acciones transaccionales.

- Envía el estímulo de inicio de transacción al TM para que lo almacene en su historia. De esta forma, se activarán las primeras acciones transaccionales (6) siguiendo el mismo procedimiento que en un *Controller*.

Cuando termine (si lo hace con éxito) la transacción, el TM enviará la ocurrencia de dicha transacción (7) y desaparecerá. Si la transacción se interrumpe, el TM desaparece y el sistema queda en el mismo estado que antes de iniciarse. No hay nada en la SFH que pertenezca a ella.

6.2 Evaluator.

Tiene dos funciones principales: la evaluación de las pre-condiciones de las acciones y la realización de consultas, *queries*. Las peticiones de ambos tipos de servicios proceden de los agentes. Aparte, cuando se inicia una transacción, el OcuR le solicita la evaluación de las pre-condiciones *p* de las acciones transaccionales.

El proceso de evaluación que se sigue está basado en el propuesto en la tesis de Mª José Rodríguez [Rodríguez00a]. También se utiliza el lenguaje de descripción de consultas de dicho trabajo y su proceso asociado.

Por eficiencia, cada *Evaluator* tiene en una tabla, *Tabla de Funcionamiento*, almacenada la información de las acciones que se pueden realizar junto con la expresión en lógica temporal de predicados de su pre-condición *p*. Aunque, conceptualmente, los agentes, cuando inician una evaluación suministran al *Controller* el nombre de la acción y su pre-condición.

tabla_funcionamiento = { nombre_accion + pre-condición_p }

Esta Tabla de Funcionamiento es distinta a la Tabla de Acciones porque en la primera se almacena la pre-condición completa y en la segunda sólo se almacena la relación entre la acción y las acciones y estímulos que aparecen en la pre-condición de ésta, no los operadores que la relacionan.

Como dijimos en el punto anterior, comparte con el OcuR la Tabla de Acciones Iniciadas. El *Evaluator* inserta una entrada cuando el resultado de la evaluación de la pre-condición de una acción es *true*. Suponemos que en ése instante la acción ha comenzado a realizarse.

Además, con el fin de proporcionar a los usuarios una forma de especificar qué acciones no se deben ejecutar concurrentemente, este agente mantiene una *Tabla de Acciones Incompatibles*. En esta tabla se tiene una entrada por cada acción incompatible junto con la lista de acciones con las cuales es incompatible.

tabla_acciones_incompatibles = { nombre_acción + { nombre_acción } }

6.2.1 Evaluación de las pre-condiciones *p* al inicio de una transacción.

Cuando al OcuR le llega un estímulo de inicio de transacción, una de sus acciones es solicitar al *Evaluator* la evaluación no sólo de la *p* de la transacción sino de las *p* de las acciones transaccionales. Para ello, le envía tanto el nombre de la transacción como su definición. A partir de la definición de la transacción el *Evaluator* construye una expresión lógica donde los operandos son las distintas pre-condiciones *p* de las acciones transaccionales. En el capítulo 5 expusimos los motivos que fundamentan esta actuación.

Cuando llega una solicitud de este tipo, el *Evaluator*, primero evalúa sobre la SFH la pre-condición *p* de la transacción, y si esta es *true*, entonces comienza a evaluar las de las acciones transaccionales. Si es *false*, no continúa con el proceso, directamente envía una respuesta *false* al OcuR y la transacción no se lleva a cabo por el momento.

El *Evaluator*, dada la definición de la transacción, sabe que pre-condiciones *p* tiene que evaluar. En la siguiente tabla se puede ver la correspondencia de las expresiones en TDL más básicas con la evaluación que se realiza.

Transacción	Debe ser *true* la expresión
a ; b	p_a and p_b
a || b	p_a and p_b
a | b	p_a or p_b

Tabla 6.1 Relación entre la definición de una transacción y la evaluación de las precondiciones p de sus acciones transaccionales

Hemos creado un traductor que, dada una expresión en TDL, genera una expresión en Lógica Temporal de Predicados (LTP) donde los operandos son las pre-condiciones *p* de las acciones transaccionales correspondientes. Estas pre-condiciones son expresiones donde aparecen o pueden aparecer operadores temporales. La expresión generada es la que debe evaluarse junto con la pre-condición de la transacción y ambas deben ser *true* para que dicha transacción se inicie. Los operadores *and* y *or* no sólo comprueban la realización de una acción o la llegada de un estímulo sino el orden temporal de su realización.

6.2.2 Un problema adicional: la opcionalidad de acciones transaccionales.

Una vez evaluada una pre-condición y en caso de que ésta sea cierta, aparte de devolver un *true* al agente puede que se le deba enviar más información al OcuR. Esto sucede en el caso de que en la expresión lógica obtenida para la evaluación de las pre-condiciones *p* de las acciones transaccionales exista algún operador *or*. Por ejemplo, para la siguiente transacción, *A*:

$$A = (a \ ; \ b) \ || \ (c \mid d)$$

se obtiene que debe evaluarse:

$$(p_a \text{ and } p_b) \text{ and } (p_c \text{ or } p_d)$$

Imaginemos la siguiente situación: el *Evaluator* evalúa la expresión anterior (además de la pre-condición *p* propia de la transacción) y ésta es *true*. Esta evaluación fue *true* porque se cumplió la pre-condición *p* de la acción transaccional *c*, además de la de *a* y de la de *b*. Sin embargo, se puede dar el caso de que durante la ejecución de la transacción se active y se ejecute *d* (y no *c*) debido al mecanismo utilizado para la activación de las acciones. Por tanto, el funcionamiento no sería el correcto ya que la pre-condición de la acción transaccional *d* no se cumplía cuando se inició la transacción y, por tanto, su ejecución no puede hacer que la transacción finalice con éxito.

Estas situaciones deben controlarse. El *Evaluator* del agente TM debe conocer, en este caso concreto, si la acción transaccional *d* en el momento de iniciar la transacción cumplía su *p* o no. Nótese que esta situación sólo ocurre cuando en la definición de la transacción se encuentre uno o más símbolos de opcionalidad ('|') o en su defecto, que en la expresión en LTP generada para la evaluación aparezca uno o más operadores *or*.

La solución que proponemos es que el *Evaluator* devuelva al OcuR una nueva definición más simple de la transacción. Esta se usará exclusivamente para esta activación de la transacción en concreto y sólo en el caso de que la evaluación fuera *true*. La nueva definición la utiliza OcuR para suministrarle la información necesaria al TM creado para gestionar dicha transacción. Esta nueva definición sólo involucra a la transacción que está en curso, no cambia la definición de la transacción mantenida por el OcuR. En el ejemplo que exponíamos anteriormente, la nueva definición sería:

$$A = (a \ ; \ b) \ || \ c$$

En general, después de la evaluación de las pre-condiciones *p*, y una vez comprobado que la transacción puede iniciarse, se crea, si es necesario, una nueva definición de la transacción eliminando aquellos elementos involucrados en un *or* cuya evaluación de su pre-condición *p* hubiera sido falsa. Esta nueva definición es enviada al TM correspondiente para que cree sus tablas internas y realice la activación de acciones transaccionales de forma coherente al estado real del agente.

6.2.3 Acciones incompatibles.

Hasta ahora no hemos abordado ni resuelto el siguiente problema: en un sistema software puede que existan acciones que no pueden ejecutarse concurrentemente. Como indicamos en la definición del modelo, los agentes realizan acciones y éstas se llevan a cabo, por defecto, concurrentemente. Las pre-condiciones asociadas a las acciones nos permiten modelar cuándo una acción puede llevarse a cabo en el sistema software pero no que la ejecución de una acción es incompatible con la de otra. Nos interesa resolver este problema para construir un modelo más flexible, de forma que permita construir un amplio abanico de sistemas software.

Con el fin de solucionar este problema, cada vez que el *Evaluator* evalúe la pre-condición de una acción a *true*, suponemos que ésta se inicia en ese momento en el agente. Por tanto, en esas circunstancias, el *Evaluator*, además de devolver la respuesta al agente que lo solicitó, introduce en la tabla de acciones iniciadas una entrada con el nombre de dicha acción. Recordemos que, cuando la acción finalice y se le envíe la ocurrencia de dicha acción al OcuR, será éste quien elimine su entrada correspondiente de la tabla de acciones iniciadas.

La información de las acciones que no se pueden ejecutar concurrentemente procede del equipo de desarrollo que está creando el sistema software y que conoce, gracias a su contacto con los usuarios del sistema, cuales son las restricciones de incompatibilidad. Esta información se le envía al *Controller* quien, a su vez, se la da al *Evaluator*. El *Evaluator* almacena esta información en la tabla de acciones incompatibles y la mantiene actualizada, ya que durante la vida del sistema software, el conjunto de acciones incompatibles puede variar. Al crear una acción se introduce la lista de las acciones incompatibles con ella. Esta lista se puede modificar dinámicamente.

Cuando al *Evaluator* le llega una solicitud de evaluación de la pre-condición de una acción sigue la siguiente secuencia de pasos:

1. Comprobar si existe alguna acción iniciada que sea incompatible con ésta. Para ello, consulta la tabla de acciones iniciadas y la tabla de acciones incompatibles.

   1.1. Si existe alguna acción incompatible ya iniciada, el resultado de la evaluación es *false*.

   1.2. Si no existe, continúa con la evaluación normal.

2. Devolver el resultado de la evaluación al agente solicitante.

Muchas pueden ser las causas de que dos o más acciones sean incompatibles. La determinación de esto depende del sistema software que se está modelando. Por tanto, son los usuarios, a través del equipo de desarrollo, los que deben decidir qué acciones son o no incompatibles. La lista de acciones incompatibles con una dada se introduce al crear cada acción simple o compleja en el agente, aunque después se puede modificar a través de una acción de evolución.

Las acciones incompatibles dentro de un agente no entran en conflicto ni con el sistema de pre-condiciones ni con la definición de una acción compleja donde aparezcan dos o más acciones incompatibles. El único punto a tener en cuenta en la definición de una acción compleja es el caso en el que intervenga el operador de paralelismo. Por defecto, las acciones unidas con el operador de secuencialidad o con el operador de opcionalidad nunca se realizan simultáneamente. Por ejemplo, supongamos la siguiente definición de una transacción:

```
Transacción = acción_A || acción_B   / acción_A  y acción_B son incompatibles
```

en este caso estamos indicando que ambas acciones se deben realizar pero estamos influyendo en su orden ya que será: primero se lleva a cabo la *acción_A* y después la *acción_B* o al revés, pero nunca se solaparán sus ejecuciones. Es decir la transacción sería equivalente a la siguiente expresión:

```
Transacción = ( acción_A ; acción_B ) | ( acción_B ; acción_A )
```

6.2.4 Los queries.

El *Evaluator* se encarga de realizar consultas sobre la historia de un agente. Estas consultas pueden ser de dos tipos: *particulares*, aquellas que devuelven *true* o *false* y *generales*, aquellas que devuelven listas de valores. Las primeras se corresponden con la evaluación de pre-condiciones, las segundas, con el concepto de *query*.

Un *query* es una expresión que se evalúa sobre la historia del agente (funcional o estructural). Permite obtener un subconjunto de un dominio de valores de uno o más atributos cuando se consulta la ocurrencia de acciones relacionadas y/o estímulos en las que estos atributos intervienen.

El conjunto de valores obtenido en una consulta puede limitarse haciendo uso de los cuantificadores. En un *query*, los predicados del cuerpo también están relacionados con operadores lógicos y temporales. Se utilizan los lenguajes *L* para expresar los *queries*. En el apéndice II se puede ver cómo se llevan a cabo los *queries*. Por ejemplo, en un sistema de alquileres de coches, utilizaría un *query* para saber cuántas veces se ha alquilado determinado coche o la matrícula de los coches que hay alquilados actualmente.

6.3 Transaction Manager.

Su funcionamiento y estructura es muy similar al de un *Controller*. También tiene un OcuR que recibe las ocurrencias de acciones transaccionales y estímulos y un *Evaluator* que evalúa pre-condiciones, en este caso *t* y *s*, y realiza consultas. Sin embargo, sólo existe mientras se esté llevando a cabo una transacción. Cuando ésta finaliza o se interrumpe, desaparece del agente donde se inició dicha transacción.

El OcuR del *Controller* le suministra la información de los agentes que realizan las acciones transaccionales, las pre-condiciones *t* de éstas y la definición de la transacción. La información que relaciona agentes/acciones transaccionales/pre-condiciones se almacena en un conjunto de tablas iguales a las utilizadas por el OcuR. Estas son:

- Una *Tabla de Agentes*, donde existe una entrada por cada agente involucrado en la transacción.

- Una *Tabla de Acciones Transaccionales*, donde existe una entrada por cada acción transaccional realizada por el agente que referencia esta tabla.

- Una *Tabla de Pre-condiciones s* por cada acción transaccional. Esta tabla se construye automáticamente partiendo de la definición de la transacción.

- Una *Tabla de Pre-condiciones t* por cada acción transaccional. Ella se corresponde realmente con una durante-condición ya que determina cuándo se interrumpe una transacción.

6.3.1 Obtención de la pre-condición *s*.

Como describimos en el capítulo 5, cada acción transaccional tiene asociada una pre-condición de secuencia, *s*. Esta pre-condición de secuencia se construye basándonos en el orden de realización de dicha acción transaccional dentro de la definición de la transacción.

La *s* de una acción transaccional se puede construir automáticamente pasando de la definición de la transacción en TDL al lenguaje que es utilizado para describir una pre-condición (LTP). La transformación de los operadores usados en la descripción de una transacción al lenguaje LPT se puede ver en la tabla siguiente.

Transacción	**Pre-condición *s* de la acción *c***
a ; c	a
a | c	not a
a || c	Estímulo de inicio de transacción
(a ; b) ; c	b since a
(a || b) ; c	a and b
(a | b) ; c	a or b

Tabla 6.2 Relación entre la definición de una transacción y la pre-condición s de la acción transaccional c

Se ha diseñado una gramática con atributos para transformar una expresión en TDL y generar un conjunto de expresiones en lógica temporal de predicados que representan las pre-condiciones *s* de cada acción transaccional que aparece en la definición de una determinada transacción.

6.4 Optimizaciones del modelo.

Como hemos podido observar, el funcionamiento de los agentes viene determinado y controlado tanto por el *Evaluator* como por el OcuR. Es interesante intentar optimizar el funcionamiento de estos agentes, ya que esto mejoraría el funcionamiento general del sistema software.

Por un lado, el Evaluator de un agente es el único que comprueba las pre-condiciones de las acciones de sus agentes hijos. Esta evaluación se

realiza secuencialmente. Esta situación puede provocar un cuello de botella donde los agentes hijos están esperando a realizar sus acciones durante un tiempo significativamente grande. Por ello estudiamos cómo podemos optimizar el funcionamiento del *Evaluator* centrándonos en dos aspectos:

- La paralelización de la evaluación de una única pre-condición
- La paralelización de la evaluación de distintas pre-condiciones

Hasta ahora, el OcuR activa a los agentes en cuanto le llega una ocurrencia o un estímulo que se encuentra en la pre-condición de una de sus acciones. Sin embargo, esto es todavía ineficiente, estudiamos cómo asegurarse que la acción que vamos a activar de un agente tiene una alta probabilidad.

Es interesante que las acciones que los agentes realizan se activen lo antes posible. Por ello, consideramos un caso especial, aquel en el que las pre-condiciones de la acciones contienen un único elemento.

También aparece el problema de *selección justa* (*fairness*) asociado con la forma en la que el OcuR consulta sus tablas internas para activar a los agentes. Si siempre las consulta en el mismo orden, puede que algunos agentes sufran de inanición. Este problema puede producirse como consecuencia de la secuencialidad en la evaluación de pre-condiciones. Si, al llegar una ocurrencia o estímulo para que algún agente realice una acción determinada, se activan varios agentes, los primeros tienen mayor probabilidad de realizar la acción que los últimos, ya que solicitarán la evaluación de la pre-condición antes.

Además, también podemos optimizar el funcionamiento del OcuR y del *Evaluator* utilizando Redes de Petri Coloreadas.

6.4.1 Optimización del funcionamiento del agente *Evaluator*.

Como hemos visto en anteriores capítulos, este agente acepta peticiones para evaluar la pre-condición de una acción y devuelve una respuesta al agente que la solicitó. Además, puede también realizar consultas (*queries*) sobre la historia y devolver un conjunto de elementos.

La pregunta que nos hacemos es: ¿cómo podría realizar su trabajo lo más eficientemente posible? Vamos a estudiar las distintas formas en las que podría actuar el *Evaluator* para sacar el máximo rendimiento con el fin de obtener un funcionamiento del agente lo más eficiente posible.

Nos interesa estudiar si los accesos de lectura que realiza el *Evaluator* sobre la historia, pueden llevarse a cabo concurrentemente. En este caso se puede acelerar el proceso necesario para realizar los servicios solicitados.

Primero vamos a centrarnos en el estudio de la evaluación de pre-condiciones y luego abordaremos el problema de las consultas sobre la historia.

6.4.1.1 Paralelización de la evaluación de una única pre-condición.

Partimos del hecho de que una pre-condición expresada en el lenguaje L (ver apéndice I) es una fórmula bien formada compuesta por uno o más átomos unidos por algún operador lógico. Cada átomo se corresponde con un nombre de una acción (ocurrencia de acción) o de un estímulo. Para comprobar que una pre-condición se cumple, se necesitan realizar, normalmente, varios accesos a la historia. Hasta ahora hemos supuesto que la operación de evaluación bloquea a la historia cuando se inicia y la desbloquea al finalizar. Esto supone que la historia permanece bloqueada durante más tiempo del necesario (ya que parte del proceso de evaluación se puede hacer sin acceder a la historia).

Inicialmente no existen problemas, el *Evaluator* al recibir la pre-condición a evaluar, puede dividir la pre-condición en subfórmulas bien formadas independientes e iniciar la evaluación de cada una de ellas concurrentemente. Estas partes no tienen por qué ser un número fijo, dependen de la pre-condición a evaluar. Por ejemplo, supongamos que la pre-condición es:

a and b / a, b son nombres de acciones o estímulos

En este caso se pueden lanzar dos *Evaluator*s, hijos del *Evaluator* principal, que trabajarán sobre la misma historia. Uno de ellos comprobará si hay alguna ocurrencia de *a* en la historia y el otro si existe una ocurrencia de *b* en ella. El Evaluator principal se encargará de devolver el resultado evaluando la expresión según el operador *and*.

Un segundo ejemplo de pre-condición es el siguiente:

a and (b Since c) / a, b, c son nombres de acciones o estímulos

Igual que en el caso anterior, también se lanzarían dos *Evaluator*, uno para cada operando del *and*, uno para *a* y otro para *(b Since c)*. Aunque *(b Since c)* se compone de dos elementos, no se debe descomponer por la interpretación del operador *Since*. En nuestra interpretación de este operador, primero se busca en la historia la existencia de una ocurrencia

de *c* y después se comprueba que existe una ocurrencia de *b* almacenada con posterioridad. La búsqueda primero se hace desde el último elemento de la historia hacia atrás y una vez encontrado el componente de la derecha del *Since*, se realiza una búsqueda hacia adelante.

Este proceso de evaluación de las partes de una pre-condición en paralelo supone la creación de un árbol de agentes de tipo *Evaluator*, en el que el nodo raíz es el *Evaluator* principal. Este árbol de agentes se corresponde con el árbol resultante de la descomposición de la pre-condición a evaluar en subfórmulas. En éste, las hojas son formulas bien formadas indivisibles que se irán evaluando independientemente. La solución a la pre-condicón se construye desde las hojas, subiendo por las ramas y obteniendo los distintos resultados parciales hasta llegar al resultado final (*true* o *false*) obtenido por el Evaluator principal. Los distintos Evaluator trabajan concurrentemente, por tanto distintas partes de la pre-condición pueden estar evaluándose simultánemente.

La creación de los distintos *Evaluator* puede ser dinámica, y su número depende de las partes en las que se divida la pre-condición. Cada *Evaluator* creado seguirá los siguientes pasos:

1. Si es el *Evaluator* principal, bloquea la historia para que el OcuR no pueda acceder a ella (acceso de escritura) pero sí los agentes del tipo *Evaluator* (agentes hijos del agente *Evaluator* principal).

2. Divide la fórmula bien formada que se le ha encargado evaluar según unas reglas de descomposición. Si no se puede dividir, va al paso 5.

3. Crea un *Evaluator* por cada subfórmula.

4. Espera las soluciones parciales de la evaluación de las subfórmulas.

5. Evalúa la fórmula y envía a su *Evaluator* superior el resultado. Si es el nodo raíz, devuelve el resultado al agente solicitante y desbloquea la historia. Si no lo es y ha finalizado su trabajo, termina.

En la figura siguiente podemos ver un ejemplo del árbol de agentes *Evaluator* que se crea para la pre-condición:

(a and b) or (c and (d Since e))

En este árbol hemos supuesto los valores que devuelve cada *Evaluator* después de evaluar cada parte de la pre-condición.

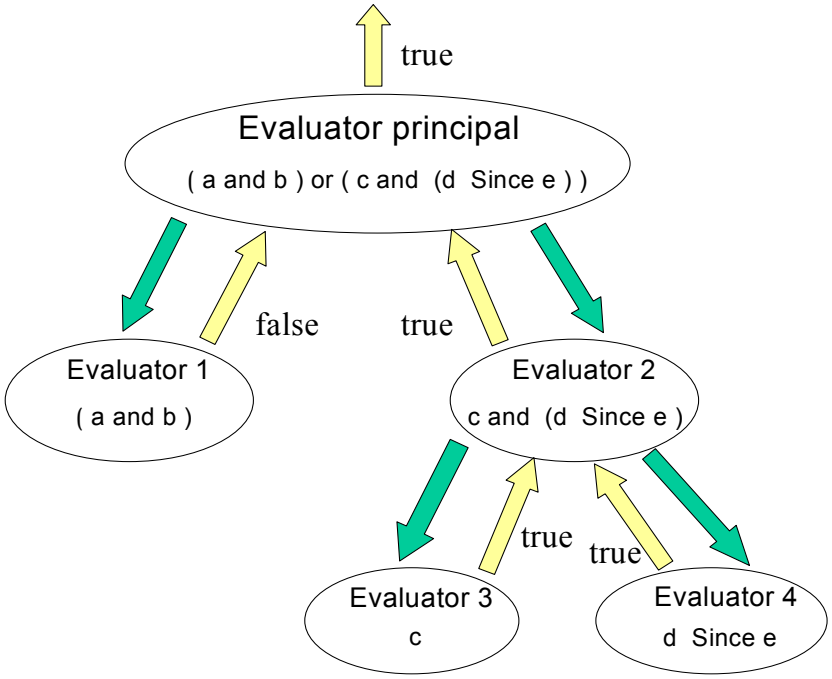


Figura 6.4 Arbol de agentes para la evaluación de una pre-condición

También es importante estudiar los conectores entre los átomos con el fin acelerar el proceso de evaluación.

- En el caso de que sea un *and*, si uno de los *Evaluator* encuentra falsa su parte de la pre-condición, como la pre-condición completa ya es falsa, se puede forzar la terminación del resto y dar su resultado. En otro caso, una vez terminadas de evaluar todas las pre-condiciones, se recogerían los resultados y se enviaría el resultado final de la evaluación al agente que lo solicitó.

- Si se trata de un *or*, distinguimos dos casos distintos. Si lo que llega de evaluar un operando del *or* es *true*, ya sabemos que el resultado final es *true* y no es necesario continuar la evaluación. En la figura anterior, si *Evaluator 2* devuelve *true*, el *Evaluator principal* devuelve *true* y detiene la ejecución de *Evaluator 1* ya que no tiene que esperar el resultado de su evaluación. Si el resultado de la evaluación parcial es *false*, se tiene que esperar a recibir todas las evaluaciones parciales para resolver la evaluación completa. Si a *Evaluator principal* le llega el resultado de *Evaluator 1* (*false*), debe esperar a que *Evaluator 2* le envíe su evaluación para resolver la pre-condición completa.

Con esto agilizamos la evaluación de una pre-condición sobre una historia, ya que ésta podría llegar a tener un tamaño grande en un cierto instante de la vida del sistema software y, normalmente, se debe recorrer parte de la historia para realizar una evaluación. Las búsquedas

simultáneas sobre dicha historia mejorarían el tiempo de evaluación necesario.

6.4.1.2 Paralelización de la evaluación de distintas pre-condiciones.

Otra cuestión que nos planteamos es: ¿podríamos tener varios *Evaluator* que se ejecutasen concurrentemente y que cada uno de ellos evaluara una pre-condición procedente de solicitudes distintas?

Suponemos que si, siempre que sus pre-condiciones (o parte de éstas) no entren en conflicto. Es decir, si el hecho de que se lea una ocurrencia o un estímulo en la historia hace que varias pre-condiciones sean verdaderas aún cuando sólo fuera válida para una de ellas, estaríamos provocando un mal funcionamiento en el agente. De alguna forma tendríamos que invalidar esa ocurrencia o estímulo para la evaluación de una de las pre-condiciones.

Un ejemplo sencillo de conflicto entre acciones es el siguiente: supongamos que en un sistema de alquileres de coches no existe más que un coche y hay dos empleados que intentan alquilarlo al mismo tiempo. Esto puede originar la evaluación de las pre-condiciones de dos acciones *alquilar* simultáneamente. La acción *alquilar* tiene la siguiente pre-condición:

alquilar(x) ← *devolver(x)*

Si se evaluaran simultáneamente, es probable que ocurra lo siguiente:

- Llega una ocurrencia de la acción *devolver*. Esto activa a los dos agentes hermanos que realizan la acción *alquilar*: *Empleado1* y *Empleado2*.
- Los dos agentes envían una solicitud de evaluación de la pre-condición de la acción *alquilar*. Ambas son iguales.
- Se inicia la evaluación simultánea de ambas pre-condiciones y se concluye con que ambas son ciertas, la de *Empleado1* y la de *Empleado2*.
- *Empleado1* inicia la acción *alquilar*.
- *Empleado2* también inicia su acción *alquilar*.

Consecuencia: ¡Estamos alquilando de forma simultánea dos veces el mismo coche! Podríamos haber definido una incompatibilidad de ejecución concurrente entre las dos acciones *alquilar*. Sin embargo, en este caso, no nos serviría de nada ya que ninguna de ellas ha comenzado a realizarse y, por tanto, ninguna aparece en la Tabla de Acciones Iniciadas que es la que se utiliza para comprobar la incompatibilidad de ejecución. Incluso si hubiese más de un coche para

alquilar, es posible que se inicie el alquiler de un mismo coche dos veces. Puede que en las dos evaluaciones de *alquilar* tomen como coche libre al que tiene matrícula 2000-BKT (el primero en una lista de coches disponibles). Por otra parte, si dos acciones *alquilar* sobre coches distintos se intentan realizar simultáneamente, se debe permitir.

Podemos resumir el problema en este sistema diciendo que la evaluación de *alquilar(x)* está en conflicto con la de *alquilar(y)* si y sólo si $x = y$. Este conflicto se deriva de la existencia de lo que llamamos ocurrencias y estímulos consumibles. Definimos *ocurrencia o estímulo consumible* a aquel que sólo puede ser utilizado en la evaluación de una única pre-condición si ésta ha dado como resultado *true*. La devolución de un coche, *devolver(x),* es una ocurrencia consumible, ya que sólo debe servir para iniciar una única acción de alquilar.

Para ello, tenemos que ser conscientes de qué acciones tienen pre-condiciones (o qué evaluaciones de pre-condiciones) que pueden entrar en conflicto en un momento dado. Una vez conocido esto, podemos insertar los conflictos en una tabla que llamamos *Tabla de Conflictos*. En esta tabla se relacionan las acciones en conflicto junto con las acciones que se han comenzado ya a evaluar. Mantenemos una *marca* asociada a cada acción de esta tabla para saber si la pre-condición de dicha acción está siendo evaluada. Si está siendo evaluada, la *marca* asociada a la acción en la tabla de conflictos tendrá un valor *true*. Por seguridad, todo acceso a esta tabla se realiza en exclusión mutua.

tabla_conflictos = { accion + marca + { accion_en_conflicto } }

Si no hubiera conflictos, varios *Evaluator* podrían ejecutarse concurrentemente. Para llevar a cabo esta paralización, necesitamos un elemento nuevo que debe encargarse de aceptar las solicitudes de evaluación de pre-condiciones y decidir si una pre-condición dada puede evaluarse concurrentemente con las pre-condiciones que se están evaluando actualmente. A este nuevo elemento lo llamamos *Coordinator* y en él distinguimos dos funciones:

- Comprueba y decide si se puede evaluar una determinada pre-condición en concurrencia con las que se están evaluando actualmente. Para ello consulta la tabla de conflictos. Si no es admisible, lo pospone hasta que finalice la evaluación de la pre-condición o pre-condiciones con las que entró en conflicto. En el ejemplo comentado anteriormente, se marcó la acción *alquilar* cuando se inició su evaluación solicitada por *Empleado1*. Al llegar la segunda solicitud de *alquilar*, por parte de *Empleado2*, se comprueba que esta acción entra en conflicto con la que se está evaluando y se pospone su evaluación. La acción *alquilar* en este caso se

encontraría entre el conjunto de acciones en conflicto con ella misma.

- Si es admisible la evaluación simultánea, crea y/o encarga a un agente *Evaluator* la evaluación de la pre-condición y el envío de la respuesta al agente solicitante. Este agente tiene la lógica necesaria para evaluar una pre-condición y comparte con los demás agentes *Evaluator* la tabla de acciones iniciadas y la tabla de acciones incompatibles.

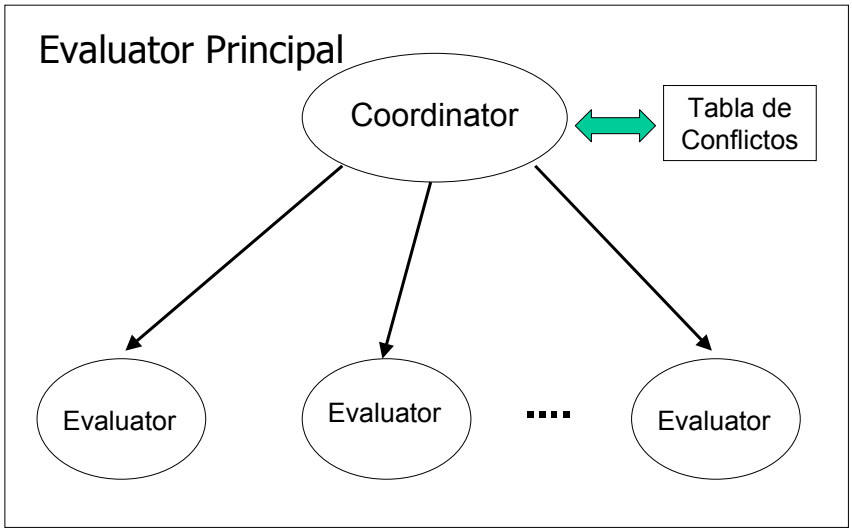


Figura 6.5 Nueva estructura del Evaluator que paraleliza la evaluación de pre-condiciones

Cuando un *Controller* comienza su tarea sigue los siguientes pasos:

1. Comprueba en la tabla de conflictos si existe una entrada para la acción cuya evaluación de pre-condición va a comenzar.

2. Si existe, mira la lista de acciones con las que entra conflicto y, para cada una de ellas, comprueba si está marcada:

   - No: marca en la tabla de conflictos la acción asociada a la pre-condición a evaluar. Con esto se evita que se comience a evaluar una pre-condición de alguna de las acciones que entran en conflicto con ella.

   - Si: almacena la petición de evaluación en un *buffer* interno del *Coordinator* hasta que pueda comenzar la evaluación, es decir, hasta que no existan acciones marcadas que estén en conflicto con la acción a evaluar.

3. Crea el *Evaluator* que evaluará la pre-condición. Existe un *Evaluator* por cada pre-condición que se esté evaluando en el sistema en un momento dado. Cuando éste termine su función, pueden haberse originado dos situaciones distintas:

   3.1 Que el resultado de la evaluación sea *true*:

   - El *Evaluator* habrá introducido en la tabla de acciones iniciadas la información de que ha comenzado una acción.

   - El *Evaluator* informa al agente solicitante del resultado de la evaluación y se retira la marca en la tabla de conflictos de esa acción (marca es *false*).

   3.2 Que el resultado sea *false*:

   - Se retira la marca de la tabla de conflictos (marca es *false*).

   - El *Evaluator* comunica al agente solicitante el resultado de la evaluación.

La creación de cada *Evaluator* puede ser dinámica o no. Tener un número fijo de *Evaluator* tiene sentido por el hecho de que el agente no está constantemente creando y destruyendo agentes (hebras, objetos activos, procesos, etc.) de este tipo. Estas operaciones consumen tiempo de CPU que no se dedicaría a realizar acciones de los demás agentes y por tanto, puede deteriorar el rendimiento del sistema software. Si fuera estática y todos estuvieran ocupados, dicha petición iría simplemente a una cola de espera. Cuando algún *Evaluator* se libere, cogerá más trabajo de dicha cola. Con esta solución se puede minimizar el paralelismo aunque esto depende del número de peticiones que lleguen simultáneamente y del número de *Evaluator* que tengamos creados.

Hemos dividido el *Evaluator principal* en dos partes conceptualmente distintas, la evaluación propiamente dicha y el problema de decidir qué pre-condiciones se pueden o no evaluar en paralelo. Por tanto, puesto que la evaluación no cambia, ahora debemos centrarnos en la segunda parte que es llevada a cabo por el *Coordinator*.

Nótese que esta alternativa de evaluación concurrente de pre-condiciones distintas no excluye a la anterior. Podemos combinarlas aunque esto complique aún mas las posibles soluciones y la evaluación de su eficiencia. Es decir, se puede paralizar a la vez la evaluación de una pre-condición y la evolución de varias pre-condiciones.

El conjunto de acciones que entran en conflicto con una acción determinada es una información fundamental para realizar

simultáneamente la evaluación de las pre-condiciones de varias acciones. Esta información depende del sistema software que estemos modelando y, por tanto, serán los usuarios los que informen al equipo de desarrollo de cuales son. Sin embargo, puede ser interesante que el propio *Coordinator* pueda crear automáticamente la tabla de conflictos a partir de las pre-condiciones asociadas a las acciones. En el siguiente punto estudiamos esta posibilidad y presentamos las conclusiones obtenidas.

6.4.1.3 Creación automática de la tabla de conflictos.

Vamos a utilizar d*iagramas de estados* para intentar obtener una regla general que permita al *Coordinator* saber, basándose en el contenido de las pre-condiciones de las acciones, si dos de ellas entran en conflicto y no se pueden evaluar concurrentemente. El objetivo de este estudio es determinar si el *Coordinator* puede crear la Tabla de Conflictos automáticamente a partir de la definición de las pre-condiciones.

Sabemos que si dos acciones están en conflicto es porque en sus pre-condiciones existen ocurrencias y/o estímulos consumibles. Puesto que inicialmente no sabemos qué ocurrencias y/o estímulos son consumibles, vamos a estimar que todas son potencialmente consumibles. De esta forma, aunque evitamos paralelizar la evaluación de algunas pre-condiciones de acciones que no entran en conflicto, resolvemos el problema de solicitarle al usuario que indique cuáles de ellas son o no consumibles.

Las acciones realizadas por los agentes tienen asociados atributos que indican, entre otras cosas, el objeto sobre el cual se realiza la acción. Dicho objeto es un elemento del sistema pasivo. Por ejemplo, en un sistema de alquileres de coches, la acción *alquilar* tiene como atributo el coche que se va a alquilar. Nos interesa estudiar qué pre-condiciones de acciones realizadas sobre objetos del sistema software pueden evaluarse concurrentemente. Por ello, se ha comenzado construyendo distintos diagramas de estado, representando con ellos las pre-condiciones que hacen que un objeto pase de un estado a otro y comprobando qué conflictos se localizan para saber si podemos llegar a unas conclusiones generales.

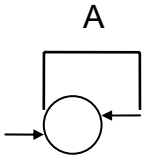
Un diagrama de estado representa los distintos estados en los que se puede encontrar un determinado elemento pasivo del sistema que estamos modelando. En los siguientes diagramas de estado, los nodos (círculos) representan los distintos estados del objeto, atributo de la acción que se realiza. Las flechas están etiquetadas con la acción que se realiza sobre dicho objeto y nos indican el paso de estado, si hay,

después de su realización. Siempre existe una flecha sin etiquetar que indica el estado inicial sobre el cual se parte.

Estudio de distintos diagramas de estados para un objeto *x*:

Hemos elegido un conjunto de diagramas de estado que representa a los casos más básicos. El razonamiento sobre diagramas de estados más complejos se construye tomando como base las conclusiones obtenidas de los diagramas de estado más simples.

a)

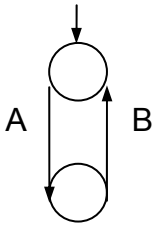


A(x) ← A(x) and not A(x) (pre-condición de la acción A)

Representa el caso en el que se pueden realizar tantas acciones *A* como se quiera, independientemente de que antes se hubiera realizado la acción *A* o no. Como no hay cambio de estado, es admisible evaluar paralelamente dos pre-condiciones de la acción *A*, aún sobre el mismo objeto. Se puede comprobar que la pre-condición siempre va a ser cierta.

Este caso es especial ya que este diagrama de estados no tiene sentido si la ocurrencia generada por la realización de la acción es consumible, ya que carecería de sentido la recursividad expresada en la pre-condición.

b)



A(x) ← not A(x) and (B(x) since A(x))
B(x) ← not (B(x) since A(x))

En este diagrama de estados podemos observar que se produce un cambio de estado del objeto. La pre-condición para *A* es que, o bien no se haya realizado nunca la acción *A*, o se realizó pero inmediatamente después se produjo una acción *B*. La pre-condición para *B* es que se haya producido la acción *A* y desde entonces no se hubiese producido una acción *B*.

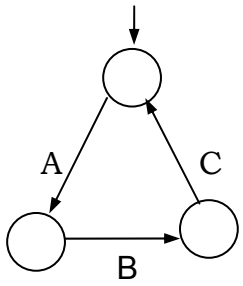
Observando este caso, se puede deducir que una evaluación de la pre-condición de *A* y una evaluación de la pre-condición de *B* sobre el mismo objeto se pueden realizar simultáneamente, ya que sólo una de

ellas será cierta: *(not B(x) since A(x))* será falsa si *(B(x) since A(x))* es verdadera y a la inversa.

Ahora, ¿ dos pre-condiciones de *A* o dos pre-condiciones de *B* se pueden evaluar en paralelo? Si, sólo si los objetos a los que afectan las dos acciones *A* o las dos acciones *B* son distintos (por ejemplo, dos acciones *alquilar* pero sobre diferentes coches) y no en caso contrario.

Por tanto, en la Tabla de Conflictos constará que la acción *A* entra en conflicto con ella misma e igual para la acción *B*.

c)



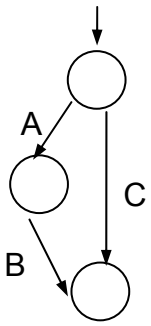
A(x) ← not A(x) and (C(x) since (B(x) since A(x)))

B(x) ← not B(x) since A(x)

C(x) ← not C(x) since B(x)

Igual que en el caso anterior, varias pre-condiciones de *A*, de *B* y de *C* se podrían realizar paralelamente si se refieren a objetos distintos. La cuestión es cuáles se podrían evaluar concurrentemente si se refieren al mismo objeto. A simple vista, si tres agentes solicitan evaluar la pre-condición de *A*, la de *B* y la de *C* respectivamente sobre el mismo objeto no habría problema porque, como puede observarse en el diagrama, se excluyen mutuamente ya que si una de ellas es cierta las otras serán falsas. Por tanto, en la Tabla de Conflictos no aparecería ninguna de ellas.

d)



A(x) ← not A(x) and not C(x)

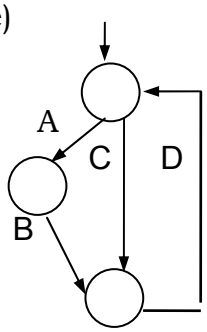
C(x) ← not A(x) and not C(x)

B(x) ← not B(x) since A(x)

En este caso podemos concluir que la pre-condición de *A* y la de *C* no se pueden evaluar concurrentemente si se trata del mismo objeto, ya que sus pre-condiciones son idénticas y, si suponemos que las ocurrencias son consumibles, sólo debe ser cierta la pre-condición de una de ellas. Sin embargo, la pre-condición de *B* sí se puede evaluar simultáneamente con las anteriores sin problema, aunque se refiera al mismo objeto, ya que su pre-condición no entra en conflicto con las de *A* y *C*.

En la Tabla de Conflictos almacenamos que *A* entra en conflicto con *C* y viceversa.

e)



A(x) ← (not A(x) and not C(x)) and (D(x) since
(B(x) since A(x))) and (D(x) since A(x))

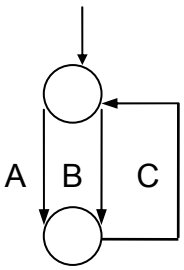
B(x) ← not B(x) since A(x)

C(x) ← (not A(x) and not C(x)) and (D(x) since
(B(x) since A(x))) and (D(x) since A(x))

D(x) ← not D(x) since (B(x) since A(x))

Las tres primeras pre-condiciones entran dentro del caso anterior. La pre-condición de *D* se puede evaluar concurrentemente con las otras tres incluso tratándose del mismo objeto ya que su pre-condición no entra en conflicto con ninguna de las pre-condiciones de las otras acciones. Se puede observar que cuando la pre-condición de *D* es cierta, las pre-condiciones de *A*, *B* y *C* son falsas. Luego, en la Tabla de Conflictos se almacena la misma información que en el caso anterior.

f)



A(x) ← (not A(x) or not B(x)) and (C(x) since A(x))
and (C(x) since B(x))

B(x) ← (not A(x) or not B(x)) and (C(x) since A(x))
and (C(x) since B(x))

C(x) ← not C(x) since (A(x) and B(x))

Como se puede observar en el diagrama, las pre-condiciones de *A* y de *B* no se pueden evaluar en paralelo si se trata del mismo objeto, aunque para objetos distintos no tendríamos problema. La pre-condición de *C* se puede evaluar concurrentemente con las otras dos incluso si se evalúa sobre el mismo objeto ya que si ésta es evaluada como cierta, la evaluación tanto de la pre-condición de *A* como la de *B* es falsa.

Por tanto, en la Tabla de Conflictos almacenamos el conflicto entre *A* y *B*.

Conclusiones:

Partimos de la suposición de que los componentes de las pre-condiciones son consumibles (aunque es posible que realmente no lo sean). Del estudio anterior podemos concluir lo siguiente:

1. Si las pre-condiciones de las acciones de los agentes se refieren a objetos distintos (distintos coches, por ejemplo), no existe problema en la evaluación concurrente de varias pre-condiciones de un mismo tipo. Esto se debe a que el cambio de estado que puede sufrir un objeto por la realización de una acción no influye en la evaluación de la pre-condición de dicha acción cuando tiene por atributo un objeto distinto.

2. Si las pre-condiciones de las acciones de los agentes se refieren a un mismo objeto pueden darse tres situaciones:
   2.1. Que las pre-condiciones sean idénticas (dos pre-condiciones de la acción *A,* por ejemplo) por lo que no se pueden evaluar en paralelo.
   2.2. Que las pre-condiciones sean excluyentes, por lo que se pueden evaluar en paralelo, por ejemplo:

$$A(x) = A(x) \text{ and not } B(x)$$
$$B(x) = A(x) \text{ and } B(x)$$

   Dos pre-condiciones son excluyentes cuando se cumpla que siempre que una es verdadera, la otra es falsa. En el ejemplo si la pre-condición de *A* es cierta, la de *B* no lo es.
   2.3. Que las pre-condiciones no sean idénticas ni excluyentes, por lo que supondremos que existen conflictos y, por tanto, no se pueden evaluar simultáneamente.

Por tanto, se puede construir de forma automática la Tabla de Conflictos en base a las pre-condiciones de las acciones que realizan los agentes. La Tabla de Conflictos creada siguiendo las conclusiones que hemos señalado anteriormente, recoge más conflictos de los que realmente puede haber en un sistema pero, como las evaluaciones terminan en un tiempo finito, lo único que hacemos es secuencializarlas y esto no afecta muy negativamente a la eficiencia del sistema de evaluación.

6.4.2 Optimización del funcionamiento del agente Ocurrence *Receiver*.

Ya hemos visto que una de las funciones del *Ocurrence Receiver* es recibir las ocurrencias y/o estímulos que hay que almacenar en la historia y controlar la activación de los agentes hijos del agente donde se encuentra. Cuando se solicita que se registre una determinada ocurrencia o estímulo en la historia, el *Ocurrence Receiver* se encarga de almacenarlo temporalmente en una cola y, cuando sea posible, se añade realmente a la historia (ésta puede estar bloqueada por el *Evaluator*). Por esto, es el más indicado para avisar a los agentes que podrían activar una o varias de sus acciones debido a la existencia de un nuevo elemento en la historia.

Para llevar a cabo la función de activación de los agentes, *Ocurrence Receiver* almacena la informacion necesaria en la tabla de agentes, tablas de acciones y tablas de pre-condiciones. Pero, ¿cómo obtiene *Ocurrence Receiver* dicha información? Tras cualquier cambio estructural, una vez comprobado por el Metasistema y llevado a cabo por el Sistema Genético del agente, se le suministrará al *Controller,* de la parte de funcionamiento del agente modificado, la información relativa a las ocurrencias y estímulos, agentes hijos y acciones y las relaciones entre éstos. El *Controller* enviará esta información a su *Ocurrence Receiver* quien la mantendrá consistente después de cualquier cambio realizado en el agente.

Veremos a continuación cómo puede optimizarse y hacerse más eficiente el funcionamiento del OcuR relativo a la activación de los agentes. Primero abordaremos las situaciones en las que las pre-condiciones de las acciones contienen un único elemento. En este caso nos planteamos si no será muy ineficiente que un agente tenga que iniciar la evaluación de la pre-condición de una de sus acciones si ya se sabe que ésta se cumple (ya que ha llegado la ocurrencia o el estímulo que provocó su activación). Seguimos ahondando en el tema de la activación intentando que, cuando se active a un agente, éste tenga el mayor grado de probabilidad de que la acción pueda realizarse. Finalmente abordamos el problema de selección justa que puede producirse en el proceso de activación de los agentes.

6.4.2.1 Pre-condición con un único elemento.

En el futuro, se puede perfeccionar el control de activación si se mantiene más información sobre las pre-condiciones de las acciones de los agentes. Por ejemplo, si una pre-condición consta de una única

ocurrencia o estímulo, la llegada de esta ocurrencia o estímulo podría originar la activación de la acción asociada de un agente sin que se tuviera que iniciar la evaluación de dicha pre-condición y esperar el resultado como se hace hasta ahora. Al darse el estímulo en el agente o llegar la ocurrencia, la pre-condición de la acción se cumple y no es necesaria la evaluación. Esto origina muchas cuestiones que se deben intentar contestar:

- Si una ocurrencia o estímulo permiten satisfacer simultáneamente las pre-condiciónes de varias acciones (probablemente de distintos agentes) ¿se activan todas las acciones o sólo una de ellas? En este caso deberíamos saber si dicho estímulo u ocurrencia es consumible o no, es decir, si sólo vale para activar a una acción o a varias.

- ¿Cómo puede distinguir *Ocurrence Receiver* los casos en los que es perfectamente factible la activación de todas las acciones de los casos en los que sólo es posible la activación de una, y solo una, de las acciones? Esto depende mucho del sistema a modelar. Se puede mantener una lista de ocurrencias y estímulos consumibles. Por tanto, si el estímulo u ocurrencia no es consumible se pueden activar todas las acciones que lo tengan como único componente en su pre-condición. Si es consumible, se activa sólo una de ellas. La elección de cual de ellas debe ser arbitraria, si no, se podría tener un problema de inanición.

- ¿Qué ocurre con los agentes que realizan acciones cuya pre-condición no se compone únicamente de un estímulo u ocurrencia? Siguen el procedimiento normal, será el *Evaluator* quien compruebe dicha pre-condición.

6.4.2.2 Activación con mayor probabilidad de las acciones de los agentes.

Otra optimización consiste en almacenar la información sobre todas las ocurrencias y estímulos que constituyen cada pre-condición de cada acción. El OcuR activará a los agentes responsables de realizar una acción sólo cuando se hayan registrado en la historia todos los componentes que aparecen en la pre-condición de ésta. En este caso los agentes solicitarán la evaluación de la pre-condición de dicha acción. No se evalúa nada, sólo se pretende que la probabilidad de que se pueda activar una acción sea alta y no se desperdicie tiempo en evaluar pre-condiciones que tiene poca probabilidad de ser ciertas. Esta aparentemente doble comprobación no es superflua, ya que pueden existir varios agentes iguales, que realicen las mismas acciones y se avisarían a todos aunque sólo para uno de ellos se resolverá su pre-condición como verdadera (no en todos los casos pero hay que

contemplarlo). Esto evita que los agentes realicen más peticiones de la cuenta al *Evaluator*, y, por tanto, éste trabajará centrándose en casos más seguros.

6.4.2.3 Problema de selección justa (*fairness*).

En la activación de los agentes, nos podemos encontrar con un serio problema de selección justa o inanición. Este problema aparece cuando se consultan las distintas tablas de agentes, acciones y pre-condiciones para conocer qué agentes están relacionados con la ocurrencia y estímulo recibido. Esta consulta, al comenzar consultando las tablas siempre desde el principio, podría favorecer a unos agentes más que a otros. Si los avisos de iniciar la evaluación de las pre-condiciones llegan antes a unos que a otros, es muy probable que soliciten el servicio de su evaluación antes, y, por tanto puedan comenzar a trabajar antes. Este problema se agrava cuando existen en las pre-condiciones ocurrencias o estímulos consumibles, ya que, probablemente el primero que consiga evaluar su pre-condición, será el que consiga obtener una evaluación igual a *true*. En este caso siempre es el mismo agente, favorecido por el orden en el que aparece en las tablas, el que consume el recurso e impide que el resto puedan realizar sus acciones.

Existen muchos trabajos relacionados que resuelven este problema [Corchuelo99], la idea es que el recorrido de dichas tablas no se realice siempre en el mismo orden y proporcione el menor grado de inanición de forma que todos los agentes involucrados tengan las mismas probabilidades de trabajar. Recordemos que el hecho de poder tener varios agentes que realicen la misma acción, puede provocar que unos estén siempre trabajando mientras que otros estén mucho tiempo ociosos debido al orden en el que se ha almacenado su información en las tablas del *Ocurrence Receiver* (o del *Transaction Manager*).

6.4.3 Optimización del *Controller* mediante RdPC.

Lo llamamos así porque esta optimización afecta a sus dos componentes, *Evaluator* y *Ocurrence Receiver*. La idea consiste en utilizar como mecanismo de activación de acciones las Redes de Petri Coloreadas (RdPC) [Jensen96] basándonos en el trabajo de Rodríguez Fórtiz [Rodríguez00a]. En este trabajo se demuestra que existe una forma de traducir una expresión en lógica temporal de predicados (LTP) a una Red de Petri Coloreada que tendrá una semántica equivalente. Otros autores utilizan las RdPC para bien para representar el funcionamiento de un sistema multi-agente [Ferber99] o bien para modelar el aspecto de sociabilidad dentro de los sistemas multi-agentes [Weyns02]. Sin embargo, ninguno de ellos utiliza sistemas compuestos

por una jerarquía de agentes, sino que trabajan con una estructura plana.

La utilización de una RdPC para representar un sistema software ofrece las siguientes ventajas:

- Se utiliza eficientemente en el proceso de activación de acciones.
- Permite almacenar información que puede ser consultada posteriormente. Hay un paralelismo entre las consultas que se realizan sobre la historia funcional y las consultas que se realizan sobre los lugares de la red de Petri y que permiten conocer las marcas que éstos contienen. Algunas consultas pueden resultar más simples en la red de Petri puesto que supondrán realizar búsquedas sólo en algunos lugares y no en toda la red.
- Nos facilita la realización de un análisis de propiedades del sistema software. La forma de construir la red obedece a la estructura del sistema y, cuando se ejecuta, vemos cómo se comporta éste. Podemos comprobar si un sistema tiene unas determinadas propiedades analizando tanto la estructura de la red, como la sucesión de marcados que ésta alcanza durante su ejecución.
- La definición de una acción compleja se muestra a través de una representación gráfica independiente. Esto nos va a permitir utilizar RdPC jerárquicas que se componen para formar la red del sistema.
- Se representan gráficamente las relaciones que se establecen entre los agentes, por medio de las acciones y las pre-condiciones de las acciones que realizan. La jerarquía de agentes del sistema está implícita en la red.
- Es fácil realizar una verificación de pre-condiciones consultando las marcas que hay en los lugares de la red.
- Permite una construcción modular de la red del sistema siguiendo la jerarquía de agentes que lo define.
- Tener asociadas las sub-redes a los agentes facilita el mantenimiento del dinamismo del sistema, que es uno de nuestros principales objetivos. Gracias a esto es fácil añadir, borrar o modificar las sub-redes que representan a las acciones que se realizan dentro de un agente. Por ejemplo, borrar una acción compleja implica la eliminación de la sub-red que representa su comportamiento.

6.4.3.1 Representación de un sistema software con Redes de Petri Coloreadas.

El funcionamiento y la estructura de un SS puede especificarse mediante una Red de Petri Coloreada. En ella podemos distinguir los siguientes elementos y su correspondencia con los componentes de un sistema software:

- Los nodos o lugares de la red representan a las acciones realizadas por los agentes del SS y a los estímulos asociados al SS.
- Las marcas o *token*s en cada lugar son los estímulos recibidos o las ocurrencias de las acciones realizadas.
- Como cada acción o estímulo tiene unos atributos, a cada lugar se le asocia un *conjunto de color* que representa los dominios de los atributos.
- La relación que hay entre una acción y su precondición se modela mediante arcos y transiciones.
- Una transacción se disparará cuando se vaya a realizar una acción porque su pre-condición es cierta.

Además, en el trabajo de Rodríguez Fórtiz [Rodríguez00a] se añaden dos elementos nuevos que son:

- Los *lugares depósito* que permiten guardar permanentemente todos los estímulos recibidos o las ocurrencias de una acción, por lo que su unión constituye la historia funcional. A cada lugar se le asocia un lugar depósito pero, por claridad en el dibujo de la red, no los pondremos aunque existen para almacenar los estímulos y/u ocurrencias recibidas en cada lugar con el fin de mantener así la historia del funcionamiento.

- Los *lugares copia* que sirven para expresar que un estímulo o una acción intervienen de forma diferente en más de una transición de una red. Las marcas de los lugares copia, en un momento concreto, dan información de qué transiciones pueden dispararse y, por tanto, de qué acciones pueden realizarse en ese momento. Cada lugar copia que se cree se nombrará concatenando la palabra *aux* con un número que se asignará de forma correlativa comenzando por el número 1. Así, si necesitamos dos lugares copias en una misma red, crearemos el lugar *aux1* y *aux2*.

A modo de ilustración, en la siguiente figura podemos ver una RdPC que representa la pre-condición de la acción *alquilar*. En ella podemos ver los elementos comentados anteriormente. Los nodos están etiquetados con el nombre de las acciones involucradas: *devolver*, *pintar* y *alquilar*. La transición *t* se dispara cuando se ha devuelto y pintado un coche, en este caso el coche con matrícula GR 223. El dominio de los conjuntos de color asociados a los nodos *devolver* y *pintar* es el mismo, el conjunto de todos los coches existentes en este sistema. En este caso suponemos que hay dos ocurrencias en la historia funcional: *devolver(GR 223)* y *pintar(GR 223),* por tanto la transición *t* se puede disparar y así alquilar el coche con matrícula GR 223.

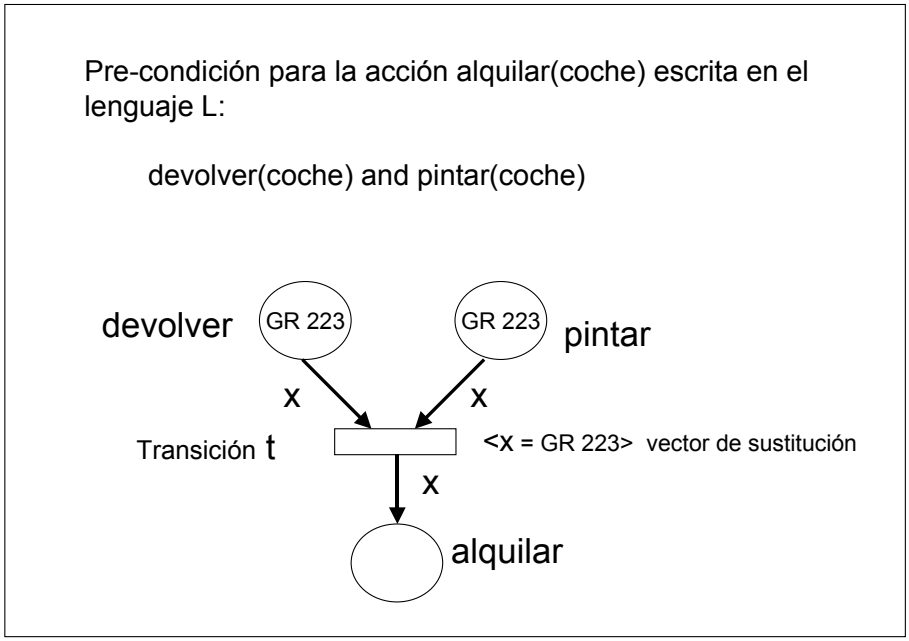


Figura 6.6 RdPC que modela la pre-condición de la acción alquilar

Asociado al sistema software, tendremos una Red de Petri Coloreada que especifica la arquitectura y funcionamiento del sistema software completo. Esta RdPC se construye tomando como base las pre-condiciones ligadas a cada acción definida en cada agente que forma parte del SS. Ahora, el agente *Evaluator* se usa para realizar las consultas generales sobre los lugares depósito, ya que el disparo de una transición implica que una pre-condición se cumple y que se va a realizar una acción. Esto implica que la evaluación de las pre-condiciones se realiza a la vez que se ejecuta la red, es decir, mientras el sistema está funcionando. La o las RdPC asociadas a un agente se encuentran gestionadas por su *Controller*. La historia de cada agente se mantiene en dichas RdPC y su OcuR se encargará de rellenar los lugares de la red para que se disparen las distintas transiciones que hacen que el agente funcione.

Básicamente, los estímulos y las acciones que forman parte de la precondición de una acción son los lugares de entrada a una transición, cuyo lugar de salida representa a la acción.

Los estímulos son lugares pero no tienen arcos ni transiciones de entrada, ya que sus marcas son introducidas por el usuario y no son el resultado de ejecuciones previas. Cuando el usuario desee realizar una acción, se añade una marca que habilite la transición, la cual, al dispararse, genera una ocurrencia para esa acción.

La realización de una acción, con unos valores concretos en los atributos de las acciones o estímulos referenciadas en su precondición, equivale al disparo de una transición habilitada con un vector de sustitución concreto.

En la siguiente tabla, basada en [Rodríguez00a], se presenta la correspondencia entre los elementos de un sistema software y los de una RdPC. Se ha complementado esta tabla distinguiendo entre acciones simples y complejas y añadiendo la descripción de un agente como una colección de sub-grafos de la red.

SISTEMA SOFTWARE/AGENTE	**RED DE PETRI COLOREADA**
Estructura del sistema que indica cómo debe ser su funcionamiento	Grafo de la red
Funcionamiento del sistema	Ejecución de la red
Estudio de las propiedades de un sistema	Análisis de la red
Acción simple	Lugar
Acción compleja	Sub-grafo de la red
Agente	Uno o más sub-grafos de la red
Estímulo	Lugar
Ocurrencia de acción	Marca o *token*
Estímulo recibido	Marca o *token*
Dominio de atributos de una acción o estímulo	Conjunto de color
Historia funcional	Conjunto de marcas de lugares depósito
Relación entre las acciones o estímulos de la precondición *p* y *s* de una acción	Arcos y transiciones
Acciones y estímulos que son parte de la precondición *p* y *s* de una acción	Lugares de entrada a una transición
Acción que se realiza obteniéndose nuevas ocurrencias de ella	Lugar de salida de una transición
Pre-condición *t* de una acción transaccional	Guarda de una transición
Realización de una acción	Disparo de una transición

Tabla 6.3 Correspondencia entre una red de Petri Coloreada y el sistema software

Para construir la RdPC de un sistema software se toma cada acción con su precondición y se van creando lugares, arcos y transiciones. Vamos componiendo la red, de tal forma que podamos especificar cuál es la

relación existente entre todas sus acciones y sus estímulos. Sin embargo, en el trabajo de Rodríguez Fórtiz [Rodríguez00a] no estaban contempladas las acciones complejas o transacciones. Recordemos que la introducción de este tipo de acciones hace que un SS se construya como una jerarquía de agentes. Por tanto, a nosotros nos interesa ampliar este trabajo y describir cómo se pueden introducir las acciones complejas en la construcción de la RdPC que representa a un SS fácilmente. Tampoco estaba contemplada la jerarquía de agentes y el hecho de que el funcionamiento de un agente se describe con una o más RdPC independientes y, para describir el comportamiento completo de un sistema se componen las RdPC de todos los agentes que intervienen. De todo esto hablamos a continuación.

6.4.3.2 Redes de Petri Coloreadas para un sistema software con transacciones.

La definición de una acción compleja establece un orden temporal entre varias acciones transaccionales realizadas dentro de un agente por sus agentes hijos. Podemos pensar en construir una RdPC que represente dicho orden. Esto nos hace reflexionar y ver que es equivalente al tratamiento que hemos dado a las acciones del sistema. El orden de realización de las acciones en el SS viene determinado por su pre-condición. Por tanto, lo que ocurre dentro de una acción compleja puede describirse también mediante una RdPC.

Necesitamos describir la RdPC equivalente a la definición de la acción compleja para cada operador del lenguaje de descripción de transacciones, TDL, que usa. Para conseguirlo usamos la relación que establecimos en el punto 6.4.1 de este capítulo entre las expresiones construidas con TDL y las expresiones construidas con LTP. Utilizamos siempre dos lugares adicionales en la red, uno etiquetado con el nombre del estimulo utilizado para iniciar la acción compleja y otro con el nombre de la acción compleja. No tomamos en cuenta la acción especial que llamamos *terminar* y que poníamos al final de la definición de cada transacción, ya que únicamente servían para indicar al TM que la transacción se había completado con éxito. Ahora no es necesario ya que si llegamos al lugar etiquetado con el nombre de la acción compleja es que ésta se ha completado. Estos nodos nos ayudarán a componer las sub-redes de Petri Coloreadas hasta llegar a construir la RdPC completa del SS. Recordemos que el nombre del estímulo de una acción compleja se suele construir concatenando la palabra *Stimulus* con el nombre de dicha acción comenzando en mayúscula.

Vemos a continuación la correspondencia existente entre la definición de las acciones complejas en TDL y sus RdPC.

1) Operador de secuencialidad:

Supongamos la siguiente acción compleja:

AC = *a ; b*

donde *a* y *b* son acciones (simples o complejas) o estímulos.

Esta expresión es equivalente a la siguiente construida con LTP:

b since a

En el apartado 1) de la siguiente figura se representa la RdPC equivalente a la última expresión. Hemos simplificado la RdPC de [Rodríguez00a] para el operador *since*, ya que una vez realizada la acción *b*, no se podrá realizar *AC* de nuevo hasta que se produzca otra ocurrencia de la acción *a*.

2) Operador de opcionalidad:

Supongamos la siguiente acción compleja:

AC = a | b

donde a y b son acciones (simples o complejas) o estímulos.

Esta expresión es equivalente a la siguiente construida con LTP:

a or b

El operador *or* al que nos referimos es un *or* exclusivo. En el apartado 2) de la figura siguiente se representa la RdPC equivalente a esta expresión.

3) Operador de paralelismo:

Supongamos la siguiente acción compleja:

AC = a || b

donde a y b son acciones (simples o complejas) o estímulos.

Esta expresión es equivalente a la siguiente construida con LTP:

a and b

El apartado 3) de figura siguiente contiene la RdPC equivalente.

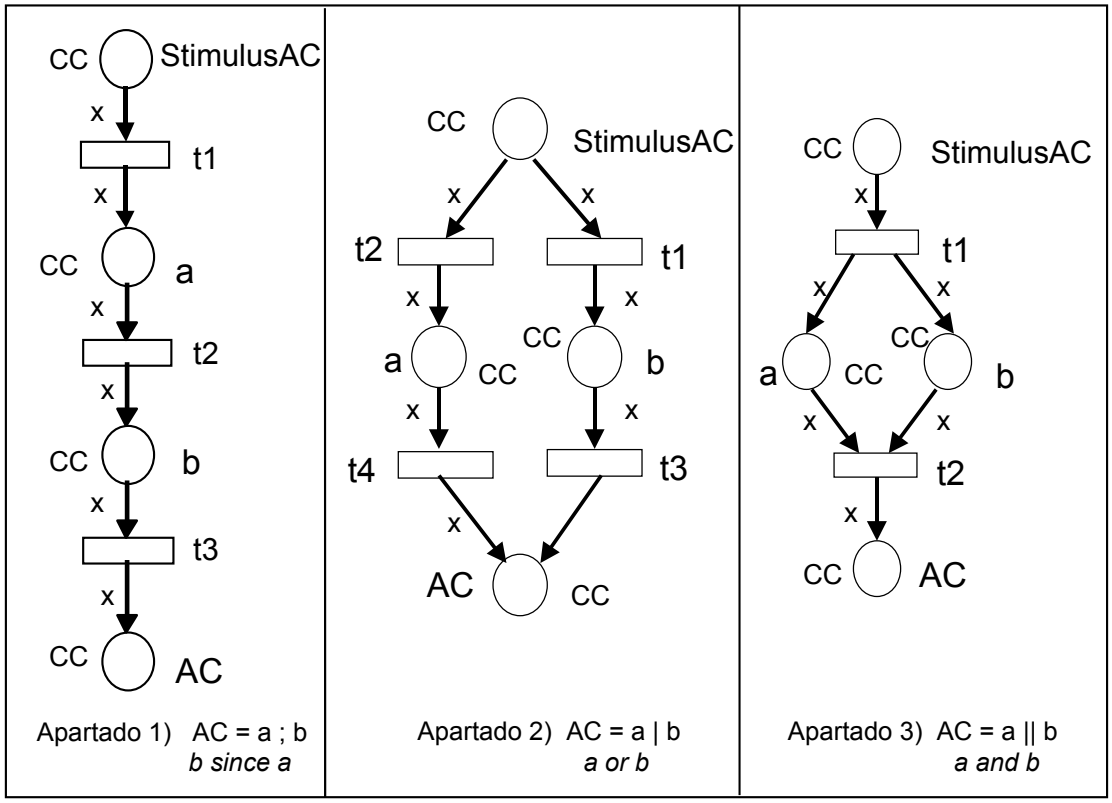


Figura 6.7 RdPC para acciones complejas

Lo que representa la sub-red de Petri Coloreada asociada a una acción compleja es el conjunto de las pre-condiciones *s* de sus acciones transaccionales, ya que las pre-condiciones *p* han sido previamente evaluadas. Pero ¿qué ocurre con las pre-condiciones *t* asociadas a cada acción transaccional? También deben estar representadas en la RdPC. Las pre-condiciones *t* aparecen como guardas asociadas a las transiciones. Si en una RdPC asociada a una acción compleja existe una acción transaccional con una pre-condición *t* distinta de *true*, dicha pre-condición sería la guarda asociada a la transición que la dispara.

En la figura anterior hemos representado mediante "CC" el conjunto de color asociado a cada lugar. En los siguientes ejemplos, por simplicidad en los grafos, asumiremos que existen pero no los representaremos si no es relevante. También representamos con "x" los valores de propagación de la red.

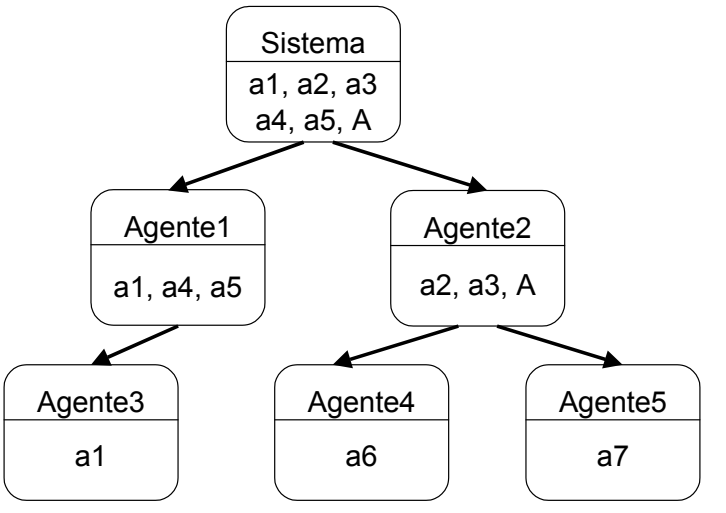
El proceso que nos permite construir la RdPC asociada a un sistema software se divide en dos fases: la *fase de construcción* y *la fase de composición*. A continuación vemos cada una de ellas.

6.4.3.2.1 Fase de construcción.

La fase de construcción de las RdPC asociadas a los agentes consiste en crear la o las RdPC para cada nodo de la jerarquía de agentes comenzando por el agente Sistema. Tomaremos en cuenta para ello las pre-condiciones *p* y las *s*. Usaremos unos lugares especiales, llamados *lugares desplegables*, que etiquetamos con el nombre de la acción junto con ":D". Estos lugares nos sirven para indicar que la acción en cuestión es una acción compleja o una acción simple que será iniciada por el agente padre. Luego usaremos esa información para sustituir cada lugar desplegable por su RdPC asociada.

Además de la optimización de los agentes *Evaluator* y OcuR, este proceso facilita la tarea tanto del diseño como de la implementación del SS. Es conveniente no tener una enorme RdPC que represente a todo el sistema software sino una en la que existan lugares que a su vez representen una RdPC (RdPC desplegables). Además, cada *Controller* gestionará la o las RdPC de su agente.

Antes de continuar indicando los pasos necesarios, vamos a mostrar en la siguiente figura un ejemplo de un sistema donde existe una acción compleja *A* y una acción simple, *a1*, que no es iniciada dentro del contexto de una acción compleja sino porque el *Agente1* delega la realización de su acción simple *a1* en su hijo, el *Agente3.* Veremos que esta acción simple a1 se trata como si fuera una acción compleja con un único elemento, es decir, tendrá asociado un estímulo de inicio de dicha acción.




```
a5 <-- (a4 since (a1 and a2)) and (A since (a2 and a3))
A = a6 || a7
```

Figura 6.8 Jerarquía de agentes y descripción del sistema de ejemplo

En general, cada acción simple delegada realizada por un agente hijo, tendrá asociado un estímulo de inicio que será enviado por su agente padre. De esta forma, consideramos a una acción delegada como si fuera una acción compleja compuesta por una única acción transaccional que será ella misma.

En la figura anterior podemos observar la jerarquía de agentes del ejemplo que hemos comentado, junto con las acciones que realiza cada uno de los agentes. Vamos a ir explicando los pasos a seguir para construir la RdPC del sistema basándonos en este ejemplo.

Paso 1: *Construcción de la red del sistema*. Se construye la RdPC del nodo Sistema, que representa al SS, y que llamamos *RdPC principal*. La RdPC principal correspondiente al ejemplo se presenta en la siguiente figura.

- Se crea un lugar en la red para cada una de las acciones realizadas por los agentes agregados.
- Se etiqueta como desplegable aquellos lugares correspondientes a una acción compleja y a una acción simple que sea realizada por algún agente hijo (acción simple delegada). Esta etiqueta se compone del nombre de la acción concatenado con ":D".
- Se conectan los distintos lugares utilizando la información que nos proporciona el conjunto de las precondiciones *p* de cada acción.

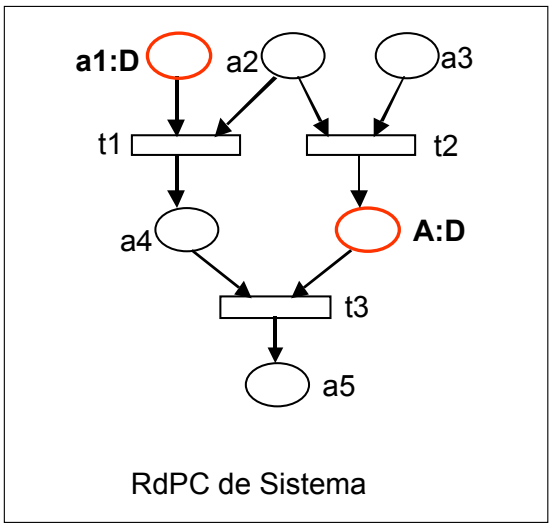


Figura 6.9 Paso 1 de la fase de construcción: RdPC principal

Paso 2: *Construcción de sub-redes*.

- Para cada agente hijo, construir su o sus RdPC, una por cada acción compleja o acción simple que sea delegación de su padre. Se siguen los mismos pasos que en el paso 1.
- Para las acciones complejas se tiene en cuenta su descripción en TDL y la traducción a RdPC que hemos mostrado en el punto anterior. Por tanto, se tiene en cuenta tanto las pre-condiciones *p* como las pre-condiciones *s*.
- Si las acciones transaccionales tienen asociada una pre-condición *t*, ésta se pone como guarda a la transición de entrada al lugar etiquetado con el nombre de la acción transaccional. Si existen varias transiciones de entrada, se asociará la guarda a cada una de ellas.
- Se repite este paso hasta llegar al final de la jerarquía. Los agentes que sean nodos hojas de la jerarquía de agentes no tienen RdPC, ya que sus acciones son simples. Se consideran nodos hoja porque no tienen acciones etiquetadas como delegadas (si simples ni complejas).

En la siguiente figura se pueden ver las distintas RdPC que se han creado en este paso. Se ha construido la RdPC para el *Agente1*. Puesto que la mayoría de sus acciones son simples, sólo se especifica la acción *a1* que se corresponde con una acción delegada. Para el *Agente2* se crea una única RdPC para su acción compleja *A*. Su acción compleja

A = a6 || a7 que equivale a la expresión en LTP siguiente: *a6 and a7*

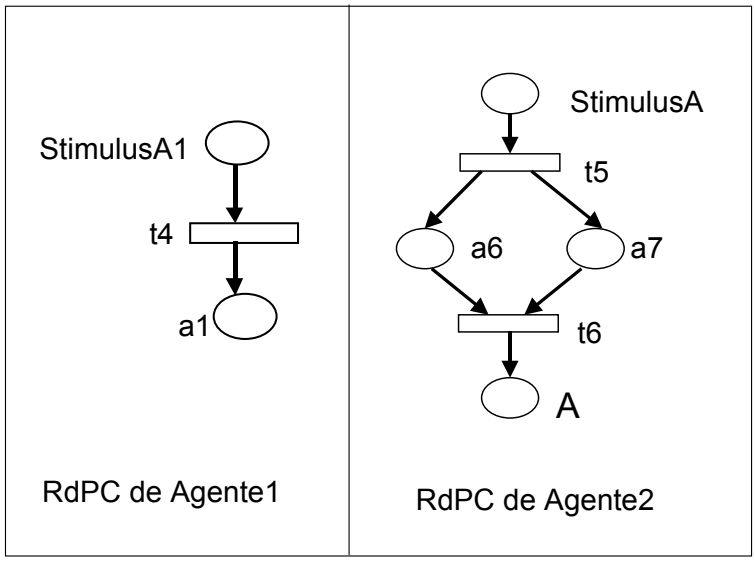


Figura 6.10 Paso 2 de la fase de construcción de RdPC

Paso 3: *Añadir lugares depósito.*

- Cuando ya se han creado todas las RdPC de todos los agentes que no son nodos hoja, pasar a la segunda fase.
- A todos los lugares de las RdPC construidas se les asocia un lugar depósito. El conjunto de todos estos lugares almacena la historia del funcionamiento del sistema. Por simplicidad en los gráficos, no los pondremos en los ejemplos.

6.4.3.2.2 Fase de composición.

La segunda fase es la *fase de composición*. Ahora se procederá desde los nodos de la jerarquía de agentes más bajos hasta llegar al nodo Sistema. En este recorrido ascendente se siguen los siguientes pasos:

1. Seleccionar un agente cuya RdPC posea lugares desplegables.

2. Sustituir cada lugar etiquetado con "**:D**" por su RdPC correspondiente definida en su agente hijo. Este paso conlleva varias comprobaciones:
   - Si existe más de un agente hijo que realice dicha acción, se realiza una composición de las RdPC de sus hijos correspondientes a dicha acción, como veremos a continuación.
   - Si la acción la realiza un agente hijo que tiene más de un padre, puede que su RdPC ya esté agregada y sólo hay que enlazarla con el lugar correspondiente. La forma de hacerlo se verá a continuación.

3. Si todos los lugares desplegables han sido sustituidos seleccionar otro agente del mismo nivel de la jerarquía. Si no existe un agente con RdPC con lugares desplegables subir al siguiente nivel de la jerarquía y repetir los pasos anteriores.

4. El proceso finaliza cuando se llegue al nodo Sistema y se hayan sustituido todos los lugares desplegables de su RdPC. En este momento tendremos la RdPC completa que representa el funcionamiento del sistema software.

En la siguiente figura y continuando con nuestro ejemplo, podemos ver el resultado de la realización de la fase de composición. En el lado izquierdo aparece la RdPC principal y en el derecho aparece la misma pero donde un lugar, correspondiente a la acción compleja realizada por el *Agente2*, se ha sustituido por la sub-red de Petri Coloreada que describe su funcionamiento. El mismo proceso ha sufrido la acción simple *a1* del *Agente2* que delega su funcionamiento en la acción simple *a1* del *Agente3*.

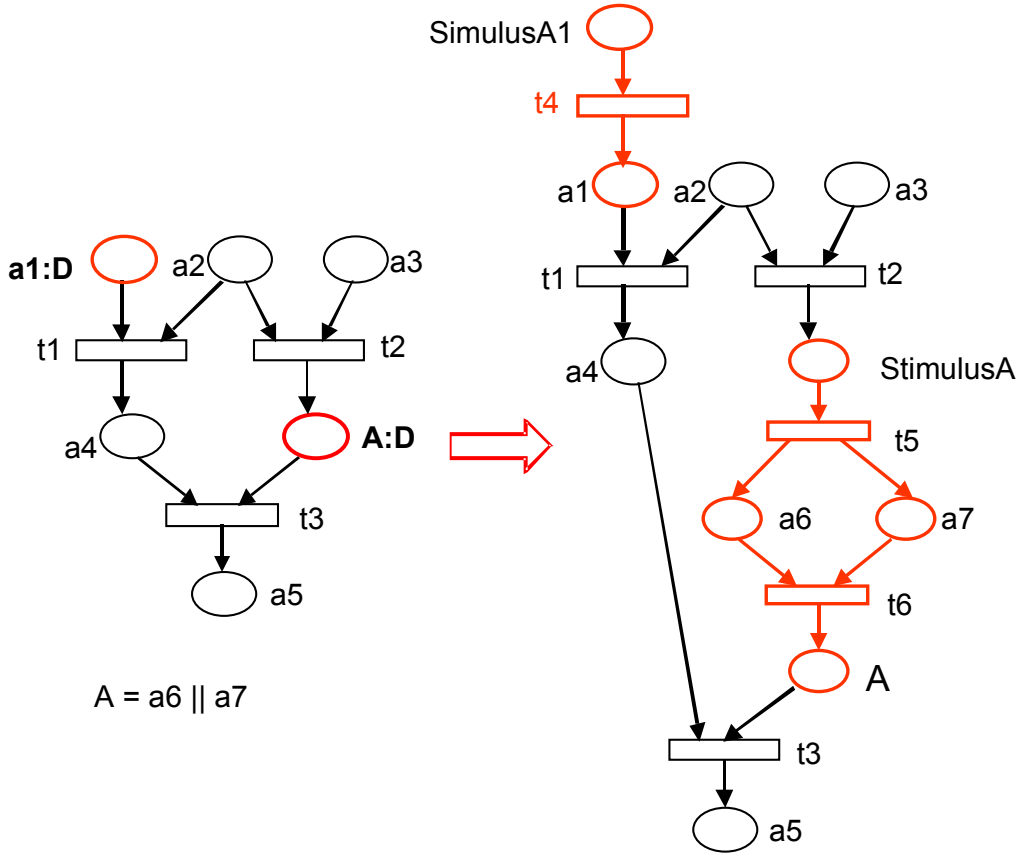


Figura 6.11 Fase de composición para crear la Red de Petri Coloreada que representa el funcionamiento del SS

Veamos el caso en el que existan varios agentes que realicen la misma acción. Imaginemos que el *Agente2* también realiza *a1*, en este caso tenemos que saber si el estímulo o la ocurrencia que le llega activa a las dos acciones *a1* o sólo a una de ellas, es decir, si el estímulo es consumible o no. Si es consumible, sólo ha de valer para una de ellas, si no lo es, valdrá para ambas.

Por tanto, por cada acción desplegable realizada por agentes hermanos tendremos tantas RdPC iguales como agentes la realicen. La forma de conexión será:

1) Si el estímulo u ocurrencia que la activa no es consumible, unir las sub-RdPC de tal forma que se puedan disparar las dos transiciones que representan la realización de las dos acciones iguales. Para ello se utilizan lugares copia como se verá en la siguiente figura (caso 1, *aux1* y *aux2* son lugares copia).

2) Si el estímulo u ocurrencia que la activa es consumible, unir las sub-RdPC de tal forma que el estímulo u ocurrencia sólo active una sola transición. En este caso duplicamos las transiciones, ya que el estímulo sólo sirve para iniciar una de las acciones (caso 2 de la figura).

En la siguiente figura se observan los dos casos para la acción *a1*.

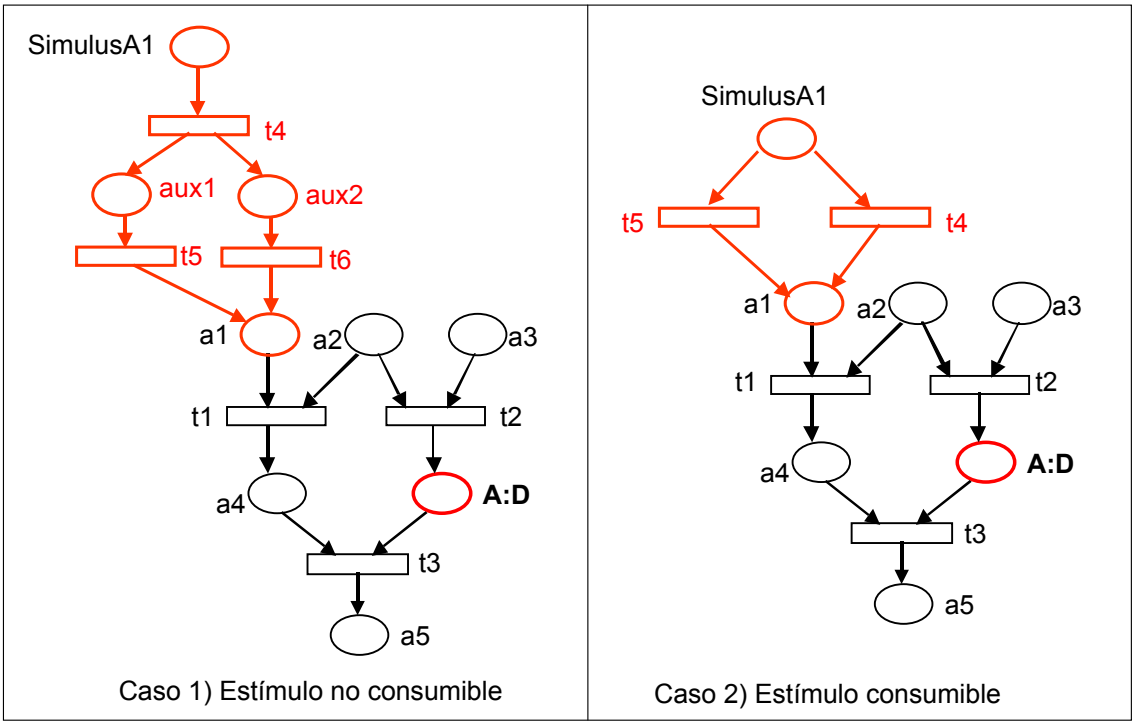


Figura 6.12 Composición de RdPC para el caso en que exista una acción realizada por más de un agente hermano

Nos queda por determinar el caso en el que tengamos un agente agregado a más de un padre. Al ser el proceso de composición ascendente, y conocerse cuáles son sus padres, las sub-redes de estos agentes se ligarán a cada una de las sub-redes de sus agentes padres. Se utilizan lugares copia para facilitar la composición. En la figura siguiente representamos una parte de un sistema donde existen dos agentes, *Agente2* y *Agente3* que tienen agregado al mismo agente, el *Agente4,* que realiza la acción delegada *b1*.

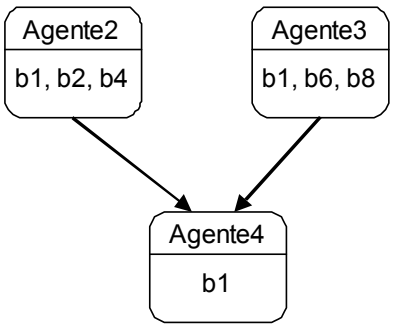


Figura 6.13 Representación de un agente, Agente4, con más de un padre

Las sub-RdPC de los agentes *Agente2*, *Agente3* y *Agente4* se representan en la figura siguiente.

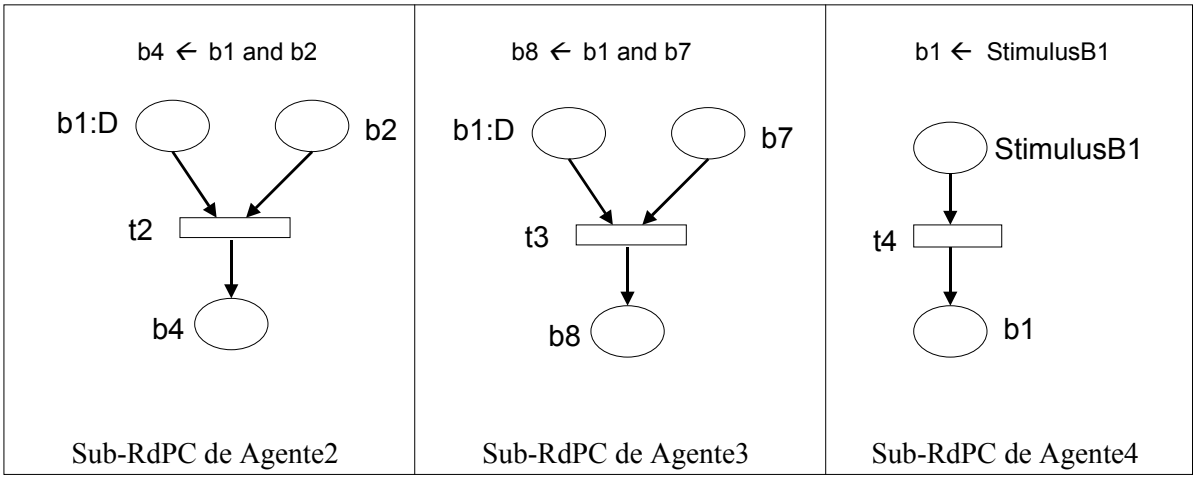


Figura 6.14 Sub-RdPC de Agente2, Agente3 y Agente4

En la fase de composición, al detectarse que un agente tiene más de un padre, se siguen los siguientes pasos:

- Para cada sub-red se crean tantos lugares copias para la acción como padres tenga.

- Cada agente padre enlazará su sub-red con la del hijo a través de un lugar copia.

En el ejemplo que estamos siguiendo, el *Agente4* realiza la acción *b1* para sus padres *Agente2* y *Agente3*. Cuando se está sustituyendo el lugar desplegable *b1* de *Agente2*, se crean los lugares copia *aux1* y *aux2* para la acción *b1* y uno de ellos se enlaza a la sub-red de *Agente2*. Cuando pasemos a sustituir el lugar desplegable de la sub-red de *Agente3*, ésta se enlazará al otro lugar copia. En la figura siguiente podemos ver la sub-RdPC obtenida después de la composición.

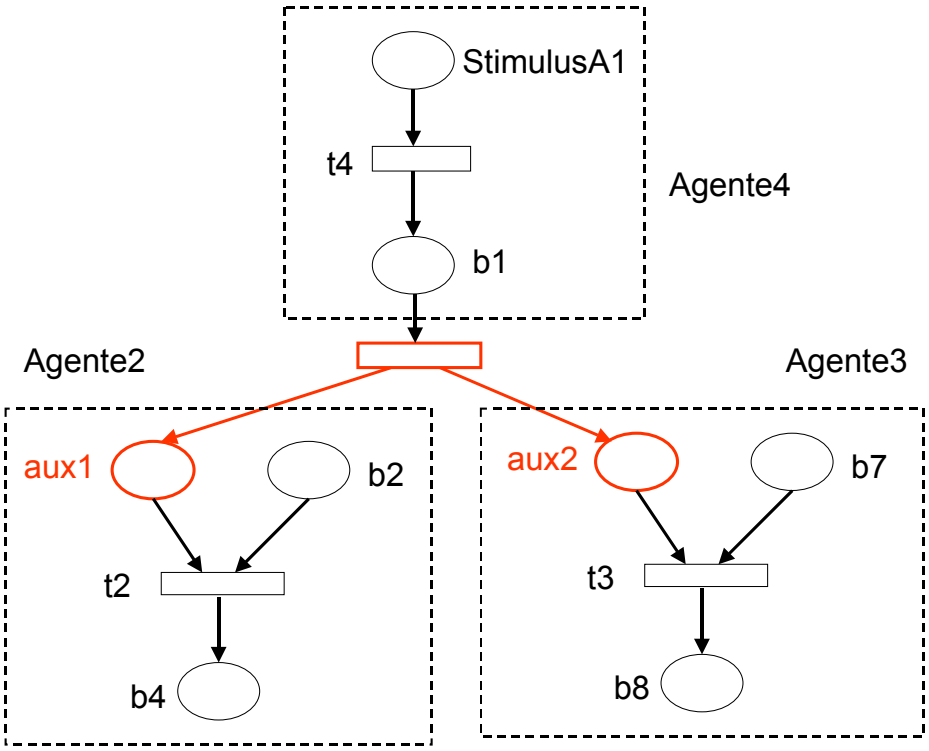


Figura 6.15 Composición de sub-RdPC cuando existe un agente con más de un agente padre.

6.5 Lenguaje de Descripción de Sistemas.

Hemos desarrollado un lenguaje para especificar la arquitectura de un sistema software siguiendo la filosofía descrita en este trabajo. Es conveniente tener este lenguaje porque:

- Pretende ser una especificación que ahorre tiempo en la generación inicial de un sistema software. Usamos este lenguaje que permite que tengamos descrito un sistema en un archivo ASCII y después, al pasar este archivo a la herramienta CASE, ésta creará el sistema software especificado. A partir de este momento, el SS puede comenzar a funcionar y también a evolucionar.

- Se puede almacenar independientemente la descripción de un sistema software o de un agente en un archivo y utilizarlo cuando sea necesario. Esto nos permite tener una batería de diseños de sistemas y agentes que podrán ser catalogados y reutilizados. Utilizaremos esta batería de diseños cuando tengamos que resolver un problema particular que se ajuste a alguno de los sistemas que ya tenemos diseñados. Además, esta coincidencia entre el problema y el sistema elegido no tiene por qué ser total, ya que se tiene la posibilidad de cambiarlo y modificarlo de acuerdo a las peculiaridades encontradas.

Este lenguaje tiene una palabra clave para casi todos los elementos que se pueden apreciar en la arquitectura de un agente.

La descripción de un sistema comienza por la palabra clave *System* seguida del nombre que le asignamos al sistema. A continuación, se describe la lista de acciones simples y otra de acciones complejas que realizarán sus agentes hijos. Por último, se encuentran los nombres de los agentes agregados al sistema. Estos agentes están descritos en archivos independientes.

El conjunto de acciones, simples o complejas, que realiza este sistema a un primer nivel. Aunque son sus agentes hijos los que realizan estas acciones, es necesario ya que si este sistema se puede agregar a otro. En este caso basta con sustituir la palabra *System* por *Agent* e introducirlo en la descripción del sistema donde se va a agregar como un agente más.

En el siguiente cuadro se muestran los distintos componentes de la definición de un sistema.

```
System nombre_sistema {


        acciones_simples = { accion, accion, ... }
        acciones_complejas = { accion, accion, ... }

        nombre_agente;
        nombre_agente;
                ...

}  // fin de la definición del sistema
```

Cuadro 6.1 Definición de Sistema

La definición de un agente comienza con la palabra clave *Agent*. A continuación se especifican las acciones simples o complejas que es

capaz de realizar este agente. Dividimos el conjunto de acciones en dos partes:

- *Simple actions*: conjunto de acciones simples.
- *Complex actions*: conjunto de acciones complejas.

y el nombre de sus agentes agregados, sus agentes hijos.

En el siguiente cuadro se puede ver las distintas partes en la definición de un agente. Como hemos visto con la definición de sistema, sólo aparece el nombre de los agentes ya que su descripción estará en un archivo independiente.

```
Agent nombre_agente {

        simple actions {
                ...
        }

        complex actions {
                ...
        }

        aggregated {

                nombre_agente;
                        ...
                nombre_agente;

        } // fin agentes hijos

 }  //  fin definicion de agente
```

Cuadro 6.2 Definición de Agente

La descripción de una acción simple es la siguiente. En ella se describe uno o más atributos. Por defecto, toda acción tiene el atributo *tr*, tiempo de realización. Después, se indica la lista de acciones incompatibles, estas acciones no se podrán realizar simultáneamente con la descrita.

Describimos la pre-condición *p* asociada a la acción utilizando el lenguaje L [ver apéndice I] basado en LPT. Por último, especificamos el código asociado a esta acción simple.

En el siguiente cuadro de texto se especifican los distintos elementos que intervienen en la descripción de una acción simple.

```
s_action nombre_accion {

        attribute {
                tr;             // tr es el tiempo de realización
                nombre_atributo;
                ...
                nombre_atributo;
        } //fin definición de atributos

        list of incompatibilities {
                nombre_accion;
                nombre_accion;
                  ...
                nombre_accion;
        } // fin de acciones incompatibles

        p_precondition = "expresión en LTP";

        action_code = "código de la acción";

}  // fin de la descripción de la acción simple
```

Cuadro 6.3 Definición de una acción simple

Una acción compleja se describe como veremos en el cuadro siguiente. La diferencia con una acción simple es que hay que introducir su definición en TDL y que no tiene *action_code*, ya que serán los agentes hijos quienes realicen las distintas partes en las que se divide una transacción. Se introducen las pre-condiciones *t* para las acciones transaccionales. Sin embargo se permite que no se especifiquen. Si no se describe la pre-condición *t* para una acción transaccional, ésta se crea por defecto con valor *true*.

```
c_action nombre_accion {

        attribute {
                tr;         // tr es el tiempo de realización
                nombre_atributo;
                        ...
                nombre_atributo;
        }

        list of incompatibilities {
                nombre_accion;
                nombre_accion;
                  ...
                nombre_accion;
        }

        p_precondition = “expresión en LTP”;

        definition = “descripción en TDL”;

        t_precondition of nombre_accion = “expresión en LPT”;

}  // fin de la descripción de acción compleja
```

Cuadro 6.4 Definición de una acción compleja

Cuando se genera un archivo con la descripción de un sistema y se le pasa a la herramienta que debe crearlo, se deben realizar todas las comprobaciones que hemos señalado en el tema de evolución de agentes. Además, las historias estructurales correspondientes no deben estar vacías sino con las ocurrencias de acciones realizadas.

Pongamos como ejemplo el conjunto de archivos relacionados con la descripción del sistema que modela la empresa de alquileres de coches que encontramos en el capítulo 8. No lo pondremos completo pero sí una parte suficientemente grande como para ver en qué consiste cada una de las secciones explicadas anteriormente.

El archivo con la definición del sistema de alquileres contiene lo siguiente:

```
System Alquileres {

        acciones_simples = { alquilar, cobrar, comprar }
        acciones_complejas = { Recoger }

        Trabajador;
        Cobrador;

} // fin de descripción del sistema Alquileres
```

La descripción del agente *Cobrador* es la siguiente:

```
Agent Cobrador {
        simple actions {
                s_action cobrar {
                        attribute {
                                tr ;        // tr es el tiempo de realización
                                coche ;
                                }
                        list of incompatibilities { }
                        p_precondition="StimulusRellenar_formulario(C) since  alquilar(C)";
                        action_code = "código de la acción";
                }  // fin de cobrar

                s_action comprar {
                        attribute {
                                tr;         // tr es el tiempo de realización
                                coche;
                                }
                        list of incompatibilities { }
                        p_precondition=" StimulusAumentarCoches(C)";
                        action_code = "código de la acción";
                }  // fin de comprar

        } // fin de acciones simples para Cobrador

        complex actions { } // no tiene acciones complejas
        aggregated { } // no tiene agentes hijos

}  //  fin de Cobrador
```

El agente *Trabajador* lo describimos como sigue:

```
Agent Trabajador {

        simple actions {
                s_action alquilar {
                        attribute {
                                tr;          // tr es el tiempo de realización
                                coche;
                                }
                        list of incompatibilities {  }
                        p_precondition=“ comprar(coche) and ( not alquilar(coche) since
                                        Recoger(coche)) and StimulusLlegaCliente(coche)”;
                        action_code = “código de la acción”;
                }  // fin de alquilar
        } // fin de acciones simples

        complex actions {
                c_action Recoger {
                        attribute {
                                tr;          // tr es el tiempo de realización
                                coche;
                        }
                        list of incompatibilities { }
                        p_precondition = “StimulusDevolver(coche)”;
                        definition  =  “Comprobar(coche)  ;  (Lavar(coche)  |  pintar(coche));
                                        terminar”;
                        t_precondition of Comprobar = “true”;
                        t_precondition of Lavar = “not StimulusFueraServicio (coche)”;
                        t_precondition of pintar = “not StimulusFueraServicio (coche)”;
                }  // fin de Recoger

        } // fin de acciones complejas

        aggregated {

                Empleado1;
                Empleado2;
                E_Taller;
        } fin de agentes hijos de Trabajador
} // fin Trabajador
```

Por último, vamos a ver a continuación la descripción de uno de los agentes hijos de *Trabajador*, que es la de *Empleado1*.

```
Agent Empleado1 {

        simple actions {
                s_action alquilar {
                        attribute {
                                tr;
                                coche;
                        }
                        list of incompatibilities {
                                pintar;
                         }
                        p_precondition=“ StimulusAlquilar(coche)” ;
                        action_code = “código de la acción”;
                }  // fin de alquilar
        } // fin de acciones simples

        complex actions {
                c_action Lavar {
                        attribute {
                                tr;
                                coche;
                        }
                        list of incompatibilities { }
                        p_precondition = “true”;
                        definition = “(enjabonar(coche)||enjuagar(coche));
                                        dar_brillo(coche) ; terminar”;
                }  // fin de Lavar
        } // fin de acciones complejas

        aggregated {

                L1;
                L2;

        } // fin de agentes agregados a Empleado1

}  //  fin Empleado1
```

6.6 Conclusiones.

En este capítulo hemos explicado detalladamente el comportamiento de los distintos agentes especiales que controlan el funcionamiento de los agentes, y por tanto, de un sistema software basado en agentes. Estos agentes especiales son el *OcuR*, *Evaluator* y el *Transaction Manager*. Para cada uno de ellos hemos presentado cómo se llevan a cabo las funciones para las cuales fueron diseñados. Hemos comentado las distintas relaciones existentes entre estos agentes.

El funcionamiento descrito de cada uno de los agentes especiales mencionados anteriormente puede optimizarse. Por ello, se han propuesto distintas soluciones para optimizar y mejorar el rendimiento

del sistema software una vez que éste esté funcionando. Estas pueden resumirse en:

- Mejorar el proceso de evaluación del agente Evaluator explotando el paralelismo potencial de éste. Se puede paralelizar tanto el proceso de evaluación de una única pre-condición como la evaluación varias, asegurando que no existan conflictos. En el primer caso no se utiliza un único *Evaluator* sino un conjunto de ellos relacionados jerárquicamente. En el segundo, hemos utilizado también un conjunto de *Evaluators* pero además un elemento adicional que llamamos *Coordinator*. Se ha añadido una Tabla de Conflictos al *Evaluator* para mantener la información de qué pre-condiciones no se pueden evaluar simultáneamente y se ha estudiado cómo ésta puede crearse de forma automática a partir de las pre-condiciones.
- Mejorar el proceso de activación de agentes eliminando dicha activación cuando no sea necesaria o asegurándose que cuando se activa a un agente la probabilidad de que pueda realizar una acción es muy alta. Además se optimiza este proceso resolviendo los problemas de selección justa que se pueden producir.
- Mejorar el tiempo de ejecución del sistema incorporando Redes de Petri Coloreadas (RdPC) como mecanismo de ejecución del sistema. Hemos descrito el proceso de creación de la RdPC que representa a un sistema software. Este proceso se ha dividido en dos fases: construcción y composición.

Por último, se ha presentado un lenguaje de descripción de sistemas donde se puede especificar textualmente la estructura de un determinado sistema. Este lenguaje proporciona una forma de almacenar descripciones modulares de sistemas y agentes que pueden luego ser reutilizados, facilitando de esta forma el dinamismo en la arquitectura especificada. Además, permite, a un desarrollador experimentado y familiarizado con el modelo propuesto, describir rápidamente un sistema en su etapa inicial, aunque luego lo haga evolucionar gracias a la utilización del Metasistema.

CAPITULO 7

Evolución de los Agentes

Un sistema software puede, y es muy probable, que cambie en el tiempo. Su evolución puede deberse a numerosas causas, cambios en los requisitos del sistema, cambios en los usuarios, cambios en el entorno donde se encuentra, etc. Esta evolución, modificación de su funcionalidad, se reflejará en los cambios que se producirán en su estructura: se añadirán nuevos agentes, desaparecerán otros o cambiará una acción, o más, realizada por un agente para el agente donde está agregado. A las acciones de evolución las llamamos acciones estructurales y las entidades encargadas de llevarlas a cabo son el Metasistema y los Sistemas Genéticos de los agentes.

En nuestro modelo, un sistema software es una estructura jerárquica de agentes, por tanto, tenemos acciones estructurales para agregar y desagregar agentes, lo cual implica un cambio en la topología de agentes existentes.

Las distintas acciones estructurales que proporcionamos para llevar a cabo una operación de creación de un agente, utilizan el mecanismo de clonación.

Cada acción estructural tiene asociada una pre y una post-condición. Además, existe una lista de invariantes asociada al sistema que se utiliza para construir tanto la pre-condición como la post-condición de cada acción estructural y para establecer el mecanismo de propagación

de cambios, si es necesario. De esta forma se garantiza que el sistema, después de realizar una acción estructural, sigue siendo íntegro y consistente.

Este capítulo está organizado en cinco secciones: la primera es una introducción al concepto de evolución. En la segunda sección nos centramos en el proceso de evolución donde presentamos las condiciones necesarias para poder realizar un cambio estructural y estudiamos con más detalle el funcionamiento del Metasistema y de los Sistemas Genéticos. La tercera sección describe distintos aspectos relacionados con las operaciones que se pueden realizar sobre los agentes. Una vez asentados estos conceptos básicos, en la cuarta sección describimos detalladamente cada una de las acciones estructurales. Finalmente, en la última sección, se exponen los aspectos a tener en cuenta para mantener consistente la información de un sistema cuya estructura es dinámica.

7.1 Introducción a la evolución.

Como vimos en el capítulo anterior, la estructura de un agente se divide en dos partes: la parte funcional y la parte estructural. Ambas partes están muy relacionadas ya que cualquier cambio en la parte funcional debe realizarse a través de acciones que se ejecutan sobre la parte estructural. Por ejemplo, si queremos que el agente realice una nueva acción podemos crear un agente hijo que lleve a cabo dicha acción. El agente habrá cambiado ampliando el conjunto de acciones que es capaz de realizar gracias a la incorporación de un agente hijo nuevo. Este cambio se consigue gracias a una modificación sobre la estructura del agente, en concreto, hemos extendido su conjunto de agentes hijos.

Un sistema software se define mediante una estructura jerárquica de agentes que realizan acciones para él. Hablamos, por tanto, de *agentes padres* y de *agentes hijos*. Un agente se dice que es hijo de otro si tiene una relación de agregación con él. Cada agente tiene conocimiento de sus agentes hijos. El nodo raíz de esta estructura lo compone un agente que llamamos Sistema y que es el único agente que no tiene un padre, el resto deben tener, al menos, un agente padre. Las acciones descritas en el nodo raíz, Sistema, conforman la interfaz de este sistema software hacia el exterior.

Para las labores de evolución existe un sistema especial que llamamos Metasistema. Este sistema es el que decide si una acción estructural se puede realizar sobre un agente o no, ya que sólo lo permitimos si el nuevo estado al que pasa el sistema software es consistente, es decir, si el sistema software ya evolucionado sigue siendo íntegro porque cumple

un conjunto de invariantes. Para ello, el Metasistema tiene un subsistema de decisión, construido utilizando el lenguaje *M2*, donde se incorpora toda la lógica necesaria para poder decidir si una acción estructural iniciada por el desarrollador puede llevarse a cabo o no. El lenguaje *M2* parte de la definición del lenguaje *M,* definido por Rodríguez-Fórtiz [Rodríguez00a]. Toda acción estructural consta de una pre y una post-condición que determinan las circunstancias en las que se puede realizar y que garantizan que se satisfacen los invariantes [Paderewski03b].

Sin embargo, quien realmente realiza los cambios sobre un agente es un agente especial que posee todo agente que llamamos *Sistema Genético*. Este agente se sitúa en la parte estructural y es quien, físicamente, realiza los cambios ya controlados y permitidos por el Metasistema. Trabaja con la *SSH* donde se introducen las ocurrencias de las acciones estructurales realizadas en el agente.

Puesto que el sistema software es una estructura jerárquica de agentes que tienen una relación padre-hijo, cuando el desarrollador inicia una acción estructural sobre ella, la realiza sobre los agentes que se encuentran en un determinado nivel de la jerarquía. El *nivel actual* se corresponde con el concepto de directorio de trabajo en un sistema de archivos jerárquico de un Sistema Operativo. De esta forma expresamos fácilmente y sin lugar a dudas a qué agente nos referimos, aunque exista más de un agente con igual nombre en toda la estructura de agentes.

El conjunto de acciones estructurales está previamente definido. Podemos distinguir tres grandes categorías:

1. *Acciones estructurales que actúan sobre las acciones del sistema/agente y sus pre-condiciones.* Es necesario poder asociar nuevas acciones, simples y/o complejas, a los agentes, eliminarlas, modificarlas, así como cambiar las pre-condiciones asociadas a las acciones.

2. *Acciones estructurales relacionadas con los atributos de las acciones*. Las acciones tanto simples como complejas tienen atributos, de hecho, deben tener, al menos, un atributo que contenga el tiempo de realización de la acción. Como el agente puede variar, es probable que un atributo asociado a una acción ya no sea necesario y, por tanto, se elimine de dicha acción. Igualmente puede ser útil poder añadir nuevos atributos o, incluso, cambiarle el nombre a alguno ya existente.

3. *Acciones estructurales relacionadas con la existencia de los agentes y la jerarquía de agentes*. Necesitamos crear agentes, para ello vamos a

tener un agente "plantilla" que llamamos *agente elemental* y que tiene los elementos básicos que hemos visto anteriormente y que constituyen la estructura de un agente. Todo agente debe estar agregado a otro, excepto el agente que llamamos *Sistema*. Con el fin de ahorrar tiempo y reutilizar agentes ya creados, se permite crear un agente como clon de otro. Aparte de las operaciones de creación, tendremos acciones estructurales que permiten variar la estructura de agentes que define a un sistema software: borrar, renombrar, agregar, desagregar o mover un agente.

Cada acción estructural (acción de evolución) tiene asociada una pre-condición y una post-condición. Ambas condiciones se construyen en base a la lista de invariantes asociada al sistema software que se desea evolucionar. De esta forma, hasta que la pre-condición de una acción estructural no se cumpla, no se podrá realizar dicha acción sobre el sistema software. Una vez realizada, se comprueba su post-condición. Si ésta no se cumple, se utilizará un mecanismo de recuperación que se encargará de deshacer la realización de dicha acción. Si se cumple, se garantiza que el sistema software ha llegado a un estado íntegro. Además, asociada a cada acción estructural, se establecen las acciones estructurales a realizar en el sistema software después de llevar a cabo una acción estructural con el fin de que se sigan cumpliendo los invariantes en él (mecanismo de propagación de cambios).

Las acciones estructurales se activan por dos causas distintas (en cuanto a sus orígenes):

- Alguien del equipo de desarrollo decide que hay que modificar la estructura y comportamiento del sistema software.
- La ejecución de una acción estructural provoca la activación de otras con el fin de mantener el sistema software consistente según la lista de invariantes asociada a él.

Todas las acciones estructurales tienen asociado un estímulo que procede del exterior (el desarrollador) o de un agente y que se introduce en la MFH provocando la activación de dicha acción estructural. Este estímulo se llamará *Stimulus* concatenado con el nombre de la acción estructural comenzada en mayúsculas, por ejemplo, asociado a la acción "añadir una acción compleja", *addCAction,* existe un estímulo llamado *StimulusAddCAction*. Este estímulo formará parte de la pre-condición de la acción unido al resto de acciones estructurales por un operador *and*.

7.2 Proceso de evolución.

Normalmente, cuando se crea un sistema software, se utiliza una acción estructural especial, *createSystem*. A partir de este momento y utilizando alguna de las acciones estructurales disponibles se crean agentes que se ligan al Sistema o a otros agentes. Se van definiendo las acciones que van a realizar, tanto simples como complejas. Es necesario comentar que para crear una acción compleja, los agentes que realizarán las acciones que componen su definición deben existir.

Cuando creamos un sistema con *createSystem* y le definimos las acciones que va a realizar, los valores de las pre-condiciones *p* de cada una son *true*. Esto es debido a que, al ser el nodo inicial de una jerarquía de agentes, no trabaja para ningún agente padre y, por tanto, no existe una historia de un agente padre donde comprobarlas. Sin embargo, si queremos que trabaje para otro sistema software (se agregaría a otro sistema software existente), su activación sí dependerá de las acciones realizadas en su, ahora, nuevo padre. Por tanto, en este caso redefinimos las pre-condiciones *p* de dichas acciones según las condiciones que queremos que se cumplan y que dependen del nuevo entorno donde empezará a funcionar.

7.2.1 Condiciones para realizar un cambio estructural.

A partir de su creación inicial, el sistema software comienza su funcionamiento normal. Ahora, en cualquier momento puede ser que el equipo de desarrollo quiera realizar modificaciones en él, es decir, quiera hacerlo evolucionar. Por tanto, la vida de un sistema software viene marcada por periodos estables en cuanto a que no cambian y periodos de inestabilidad por la necesidad de adaptarse a nuevos cambios, de evolucionar. Lógicamente, mientras el sistema software evoluciona (pasa de un estado estable a otro) debe parar su funcionamiento, pero en cuanto se hayan realizado las modificaciones necesarias, continúa trabajando con su nueva funcionalidad. Este comportamiento se ve representado en la siguiente figura.

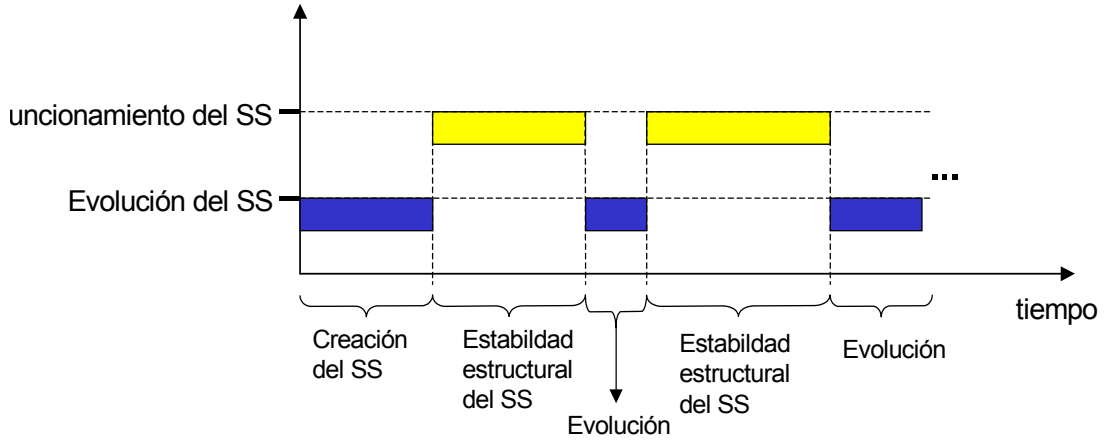


Figura 7.1 Periodos de funcionamiento y evolución de un SS

Se nos presenta el siguiente problema: ¿cómo se para el funcionamiento del sistema cuando se va a hacer un cambio estructural? y ¿cuándo se realiza realmente el cambio estructural? Puede ser peligroso realizar el cambio estructural en cualquier momento ya que pueden existir acciones funcionales que no han finalizado todavía y cuya terminación puede verse afectada por los cambios estructurales que se realicen. Por tanto, lo mejor es realizar el cambio estructural cuando todas las acciones funcionales comenzadas hayan finalizado. Para ello, primero evitamos que se inicien nuevas acciones funcionales. Esto lo conseguimos paralizando tanto el *Evaluator* como la parte de activación de agentes del *Ocurrence Receiver* de todos los agentes. De esta forma no se activarán más agentes y llegará un momento en el que el funcionamiento del sistema completo se pare. Ese es el momento de realizar cambios estructurales en el sistema software. Además, cuando el equipo de desarrollo tenga intención de realizar un cambio estructural en el SS, los usuarios de éste dejarán de interactuar con él.

Para saber si existen acciones pendientes se lleva la cuenta, en la tabla de acciones iniciadas, de las acciones que se están realizando en este momento y que todavía no han finalizado. Conforme van terminando las acciones, se van retirando de dicha tabla. Se puede realizar el cambio en el sistema software cuando las tablas de acciones iniciadas de todos los agentes estén vacías. Se paraliza el sistema software completo porque un cambio de estructura en un agente puede afectar a otros. Una vez realizada la modificación, el sistema software continúa su funcionamiento (se desbloquean el *Evaluador* y el *Ocurrence Receiver* de cada agente).

De esta forma, cada vez que evalúa el agente *Evaluator* una pre-condición y ésta es cierta, introduce en la tabla de acciones iniciadas que comparte con el agente *Ocurrence Receiver* el nombre de dicha acción. Cuando el agente que realiza la acción envíe su ocurrencia al *Ocurrence Receiver*, éste borrará la entrada que coincida con el nombre

de la acción de la tabla de acciones iniciadas. Lógicamente, el acceso a esta tabla compartida por ambos agentes se realizará en exclusión mutua.

7.2.2 Funcionamiento del Metasistema y de los Sistemas Genéticos.

Como hemos visto, la entidad que puede hacer evolucionar a un sistema software es el Metasistema. Antes de realizar modificaciones en la estructura de un sistema, el Metasistema debe comprobar que el sistema software quedará en un estado íntegro, consistente. Para ello, se asocia una lista de invariantes al sistema software, con el fin de establecer cuándo un sistema software se considera íntegro. Esos invariantes son propiedades que siempre se deben cumplir en el sistema software.

Además, todo agente tiene asociado un Sistema Genético que es quien realmente realiza los cambios estructurales sobre él. Por tanto, el Metasistema comprueba y decide si el cambio estructural se puede realizar y se lo comunica al sistema genético del agente sobre el cual se va a realizar dicho cambio que es quién, finalmente, lo lleva a cabo.

Con el fin de describir el comportamiento tanto del Metasistema como de los Sistemas Genéticos, vamos a ver la estructura que tiene el Metasistema. Como ya hemos comentado, el Metasistema es un sistema software y, por tanto, tiene la misma estructura que la de un agente. Tiene una parte estructural que no usa, ya que él no va a evolucionar, y una parte funcional con los siguientes componentes:

- Un agente *Controller* para la gestión de su historia funcional, MFH (*Metasystem Functional History*). Este agente almacena toda la información sobre las pre-condiciones de las acciones estructurales en la tabla de acciones y las tablas de pre-condiciones asociadas con cada acción estructural.

- Una interfaz de acción que utilizan los desarrolladores para introducir los estímulos de las acciones estructurales que desean realizar sobre un sistema software.

- Una interfaz de evolución que les permite comunicarse con el sistema software al que hace evolucionar. Realmente esta interfaz sirve para comunicar al Metasistema con los Sistemas Genéticos de los agentes que constituyen al sistema software cuando es necesario.

- Un agente que llamamos *Meta-Agent* y que realiza el conjunto de las acciones estructurales que se definen en su creación y que no va a cambiar mientras el Metasistema exista. Contiene una tabla de post-condiones donde existe una entrada por cada acción que posea una post-condición distinta de true. Con cada una almacenamos los nombres de las acciones que han de realizarse y unas condiciones que deben cumplirse y para los cuales este agente realizará unas consultas al *Controller*.

  tabla_post-condiciones = nombre_acción + { nombre_accion | condición y queries}

  Este agente no necesita un evaluador, ya que todos los componentes de las acciones estructurales que tienen post-condición se expresan como un conjunto de acciones o condiciones unidos por un operador *and*. Se utiliza únicamente este operador porque se deben realizar todas las acciones que forman parte de la post-condición. Por tanto, cuando se inicia una acción con post-condición, espera a que el Sistema Genético la realice, inicia las acciones de la post-condición y el *Meta-Agent* espera a tener respuesta de cada una. Si alguna parte de la post-condición no se puede llevar a cabo, se desharía la modificación realizada.

  El *Meta-Agent* mantiene una *tabla de acciones pendientes* donde en cada entrada se almacenan las acciones pertenecientes a la post-condición de cada acción estructural que se está realizando actualmente. Junto con el nombre de la acción estructural tenemos los nombres de las acciones estructurales que se han iniciado para que se cumpla su post-condición. Una acción que se está realizando y que tiene su post-condición a *true* no tendrá ninguna entrada asociada en esta tabla.

  tabla_acciones_pendientes = { nombre_accion + { nombre_accion } }

  Posee también una cola de mensajes procedentes del *Controller* que llamamos *cola de espera*. El *Meta-Agent* realiza las acciones estructurales procedentes del equipo de desarrollo secuencialmente. Hasta que no ha finalizado la acción estructural actual, no continúa con la siguiente. Sin embargo, durante la realización de una acción estructural, el Metasistema puede recibir más estímulos del equipo de desarrollo, por tanto, el *Controller* enviará sus mensajes de activación al *Meta-Agent*, quien los almacenará temporalmente en la cola de espera.

  El motivo de realizar secuencialmente las acciones estructurales se debe a que la ejecución de una acción estructural puede interferir en

la realización de otra. Nos interesa garantizar que una acción estructural se ha realizado completamente (con o sin éxito) antes de comenzar con otra. La única excepción la constituyen las acciones que intervienen en la realización de la acción estructural iniciada o las que aparecen en su post-condición. Estas acciones son iniciadas por él mismo, por tanto, puede distinguir entre las que son activadas desde el exterior, por el equipo de desarrollo, y las que él activa para que una acción se realice completamente y se asegure que el sistema software quedará en un estado íntegro.

Cada Sistema Genético de cada agente realiza las mismas acciones estructurales que las definidas para el *Meta-Agent* aunque las pre-condiciones de éstas son muy simples. La pre-condición de cada acción estructural realizada por el Sistema Genético consta de un único estímulo. Este estímulo será enviado por el Metasistema cuando decida que la acción estructural se puede llevar a cabo y en él irá toda la información necesaria por el Sistema Genético para realizar dicha acción. El Metasistema conoce a todos los agentes que componen la jerarquía de un SS y se comunica con los *Controller* de la parte estructural de todos ellos.

También es responsabilidad del Sistema Genético deshacer las modificaciones que realizó sobre el agente si la post-condición de alguna acción estructural no se ha cumplido. Por ello, almacena una copia del estado del agente o acción antes de realizar la última acción estructural. También tiene una acción especial, *deshacer*, que le indica que debe deshacer una acción estructural ya realizada. Será el Metasistema quien, después de que se complete la acción estructural y su post-condición, envíe un *StimulusDeshacer* en caso de que ésta última no se cumpla.

La acción *deshacer* se puede realizar de una forma sencilla ya que:

- Las acciones estructurales se realizan de forma secuencial, por tanto, cuando el Metasistema decide deshacer una acción estructural, el Sistema Genético se encarga de dejar la estructura del agente en el mismo estado que tenía antes de realizarla. De la SFH del agente desaparecerá la ocurrencia de la acción estructural que se deshace.
- Durante un cambio estructural, se para el funcionamiento del sistema software. Por tanto, ningún agente ha podido funcionar con la nueva estructura hasta que se completa con éxito el cambio deseado y se llega a un periodo de estabilidad estructural.

En la siguiente figura se muestra un esquema simplificado del proceso seguido para realizar un cambio de estructura. En ella aparecen numerados secuencialmente los pasos seguidos. Suponemos creado el Metasistema y un SS donde existen los agentes *Agente 1* y *Agente 2*. El Metasistema debe dar respuesta a cada estímulo enviado por un desarrollador.

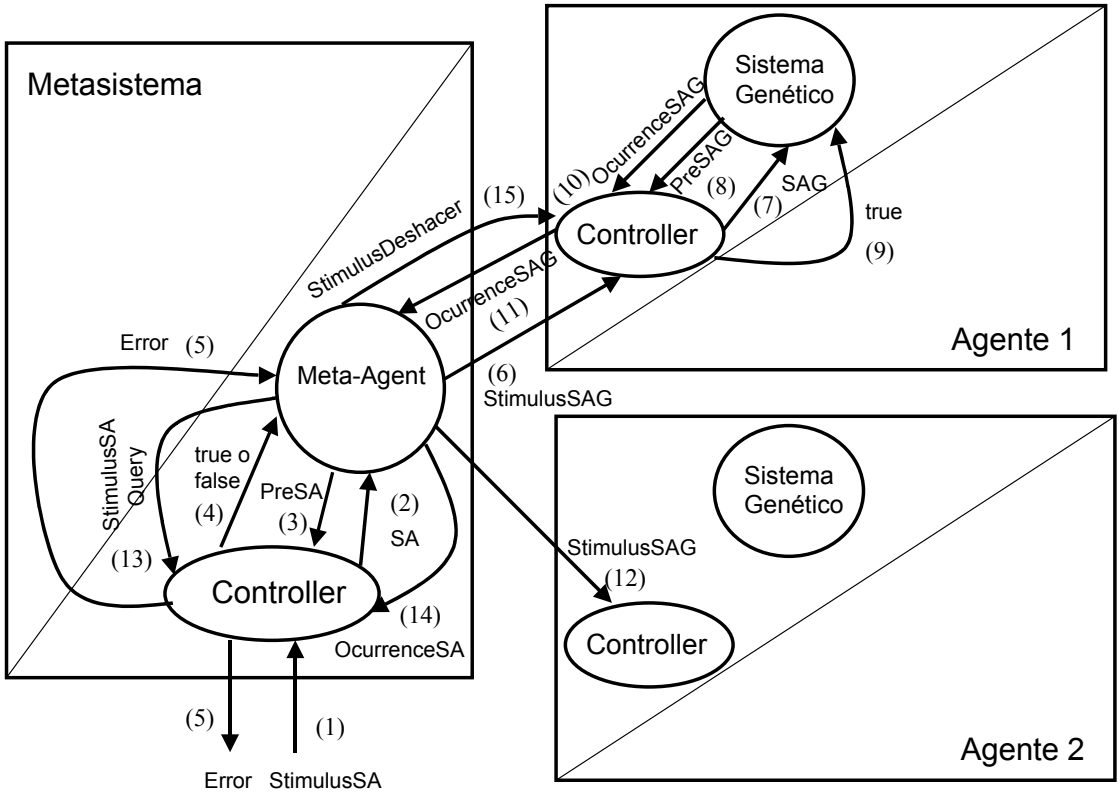


Figura 7.2 Proceso de evolución: Metasistema y Sistemas Genéticos

A continuación presentamos los pasos que realiza tanto el Metasistema como los Sistemas Genéticos de los agentes cuando un desarrollador tiene la intención de realizar un cambio estructural en el sistema software. Llamamos *SA* a cualquier acción estructural y *StimulusSA* a su estímulo asociado. Igualmente, *PreSA* denota a la pre-condición correspondiente.

(1) Llega un estímulo de una acción estructural, StimulusSA. El Controller lo almacena en la MFH.

(2) Buscar StimulusSA en Tablas de Pre-condiciones para determinar la acción SA asociada.

Enviar a Meta_Agent un mensaje con el nombre de la acción SA.

(3) Meta-Agent solicita al Controller la evaluación de la pre-condición, PreSA.

(4) El agente Evaluator consulta la MSH y evalúa la pre-condición, devolviendo una respuesta al agente solicitante:

¿ PreSA es true ?

(5) No: Informar a la entidad que solicitó la realización de SA de imposibilidad de cambio (al desarrollador o al Meta-Agent).

Si quien solicitó fue el desarrollador: Fin.

Si quien solicitó fue Meta-Agent: ir al paso 14.

Si: Continuar con paso siguiente

(6) Informar al Sistema Genético del agente apropiado del cambio físico que debe hacer:

Generar estímulo para el Sistema Genético, StimulusSAG

Utilizar interfaz de evolución para enviar al Controller el estímulo.

(7) Buscar StimulusSAG en Tablas de Pre-condiciones para determinar la acción SAG asociada.

Enviar al Sistema Genético un mensaje con el nombre de la acción SAG.

(8) El Sistema Genético solicita al Controller la evaluación de la pre-condición, PreSAG.

(9) El agente Evaluator consulta la SSH y evalúa la PreSAG: ¿es true?

No: Este caso no puede darse, ya que el único componente de PreSAG es StimulusSAG

Si: Continuar con paso siguiente

(10) El Sistema Genético modifica físicamente la estructura del agente, según la acción estructural solicitada

Guarda una copia del estado anterior del agente o acción que haya modificado, por si la post-condición de SA no se cumple y hay que deshacerla

Genera una ocurrencia de la acción estructural realizada, OcurrenceSAG

(11) Almacena OcurrenceSAG en SSH y se la envía al Meta-Agent, informando así al Metasistema del cambio

(12) Enviar estímulos de acciones estructurales a los Controller de la parte estructural de los agentes que estén involucrados en el mecanismo de propagación de cambios, si existen

Almacenar en la tabla de acciones pendientes las SAG iniciadas

(13) ¿ OcurrenceSAG se encuentra en alguna entrada de la tabla de acciones pendientes ?

Si: Eliminar esa SAG de la entrada de la tabla de acciones pendientes

Si la entrada está vacía, la post-condición de la acción estructural asociada se cumple. Ir al siguiente paso.

No: ¿ PostSA <> true ? Para ello mira la tabla de post-condiciones si hay una entrada etiquetada con en nombre SA (su post-condición no es true)

Si: Iniciar cada acción estructural para que se cumpla la post-condición: se envía un StimulusSAG para cada una al Controller. A veces, es necesario que el Meta-Agent realice una consulta (query) sobre la MFH para saber qué acciones estructurales hay que activar y sobre qué agentes(13)

Por cada acción iniciada, se almacena su nombre en la tabla de acciones pendientes y espera a que todas se realicen

No: ir al siguiente paso

(14) Enviar OcurrenceSA al Controller.

Si dicha acción tenía post-condición también se envían todas las ocurrencias de las acciones estructurales iniciadas por el Meta-Agent.

Si dicha acción tenía asociadas acciones estructurales adicionales, sus ocurrencias también son enviadas.

(15) La post-condición no puede cumplirse. Envía un estímulo StimulusDeshacer al Sistema Genético responsable de haber realizado

SA y a los Sistemas Genéticos que realizaron otras acciones estructurales por efecto de ésta.

Continúa con la siguiente acción estructural pendiente. Lee siguiente elemento de la cola de espera

No envía OcurrenceSA al Controller (por tanto, es como si nunca se hubiese iniciado) ni las ocurrencias de las acciones estructurales que se hayan realizado como consecuencia de ésta. Las borra después de haber iniciado el mecanismo para deshacer dichas acciones estructurales

Cuando se completa todo el proceso para una acción estructural iniciada por el equipo de desarrollo, se dice que el Metasistema ha hecho evolucionar al sistema software, ya que se ha realizado una acción de cambio o acción estructural. En este momento se consigue la estabilidad estructural.

El Sistema Genético limpia la memoria de las modificaciones realizadas cuando el sistema software comience a funcionar, ya que la historia de las modificaciones de los agentes de un sistema se almacenan en la MFH (historia funcional del Metasistema).

7.3 Operaciones sobre agentes.

En este punto tratamos de clarificar los conceptos que vimos en el capítulo anterior sobre la estructura de un sistema software como una colección de agentes que están relacionados de forma jerárquica entre sí. Para ello, vamos a comenzar con la operación de creación y de clonación de un agente y continuamos con la operación de agregación, que es la responsable de asociar jerárquicamente dos agentes. Además, y debido a la profunda relación existente, continuamos explicando la definición de acción compleja y la influencia que tiene en la estructura de agentes que finalmente tendrá el sistema software.

7.3.1 Notación utilizada.

Con la finalidad de formalizar las operaciones que vamos a ver y en concreto, la operación de agregación y facilitar la asociación de ésta con las acciones complejas, vamos a utilizar la siguiente notación:

- Citamos a un agente junto con sus operaciones anteponiendo el nombre del agente a unos corchetes donde aparecen los nombres de

las acciones simples o complejas que posee en su interfaz separados por comas.

Agente [a1, a2, ..., a3]

- Si el agente no es simple, los agentes que tenga agregados se muestran a continuación de la definición de agente y después de un símbolo = encerrados entre llaves y separados por comas.

Agente [a1, a2, a3] = { Agente_1 [a1,a2], Agente_2 [a1,a3] }

- El agente elemental (EA) que existe previamente y que se utiliza como plantilla para crear agentes nuevos se denota como:

EA [ ]

- Un agente que no tenga agentes agregados no tendrá ningún elemento entre las llaves. Denominamos a dicho tipo de agente, *agente simple*.

Agente_2 [a1,a3] = { }

- A las acciones simples se les asignarán nombres en minúsculas y las acciones complejas o transacciones se denotarán mediante nombres que comiencen en mayúscula.

- El nombre de los agentes comenzará con una letra mayúscula.

- Para expresar que un agente es hijo de otro concatenamos el nombre del agente padre con el del hijo separado por un punto. En el siguiente ejemplo estamos refiriéndonos al agente *Agente_1* que es hijo de *Agente*:

Agente.Agente_1 [...]

- La definición de las acciones complejas seguirá la notación del lenguajes *TDL* ya introducido anteriormente.

- Usaremos las siguientes funciones que nos serán de utilidad:

  - La función *def(A)* siendo A una acción compleja, devuelve el conjunto formado por los nombres de cada una de las acciones simples y complejas que componen la definición de dicha acción compleja.

  - La función *interfaz(Agente)* devuelve el conjunto de las acciones simples y complejas que puede realizar el agente *Agente*.

- La función *hijos(Agente)* devuelve el conjunto de los agentes agregados a Agente, es decir, el conjunto de sus agentes hijos.

- La función *hermanos(Agente)* devuelve el conjunto de los agentes que se encuentran en el mismo nivel jerárquico que el agente *Agente*, es decir, aquellos que tienen el mismo padre.

7.3.2 Creación de un Agente.

La creación del primer agente (es decir, de un sistema software) se realizará basándose en la definición de una clase Agente que consta de los elementos mínimos que se han descrito y que constituye la estructura de un agente. A este agente que sirve de "plantilla" lo denominamos *agente elemental* (EA).

Un *agente elemental* tiene las siguientes partes:

- Parte estructural:

    - Un sistema genético
    - Una historia llamada historia estructural (*SSH*)

- Parte funcional:

    - Un agente *Controller* compuesto por un agente *Evaluator*, un agente *Ocurrence Receiver* y una historia funcional (*SFH*).
    - Inicialmente el agente elemental no tiene en su parte funcional ninguna acción. Serán las operaciones de creación de agentes las que obliguen al desarrollador a asociar, al menos, una acción a cada agente.

La historia funcional del sistema para un agente elemental no tendrá contenido y sólo será útil si se agregan agentes a éste.

Igualmente, el *Controller* no tendrá trabajo si no existen otros agentes agregados ya que se encarga de que se lleven a cabo las acciones internas de dicho agente y si únicamente tiene acciones simples no es necesario, ya que no existirán problemas ni de comunicación ni de colaboración entre agentes. Las acciones simples se iniciarán gracias a la activación del *Controller* del agente padre y las ocurrencias de dichas acciones se almacenarán en la *SFH* del agente padre.

7.3.3 Clonación de agentes.

La clonación de agentes es otra forma de crear agentes. Es probable que, cuando tengamos definido un agente, queramos que existan, dentro del sistema software, otros agentes que realicen las mismas funciones. Es decir, queremos crear un agente idéntico a otro en cuanto a sus componentes y funcionalidad aunque con una identidad única y diferente. Para ello, se realiza una operación de clonación. El agente nuevamente creado no tiene por qué estar en la misma posición del grafo de agentes que el agente del cual se ha clonado.

El tener copias idénticas de un agente simplifica la creación de un sistema software donde se observan componentes con igual comportamiento pero que existen y trabajan simultáneamente. Una vez creado un agente, podemos reutilizar el diseño que hemos hecho de él para crear otros iguales. Sin embargo, a partir de la creación son autónomos y pueden evolucionar independientemente, aunque se controla este proceso para que no existan inconsistencias. También es útil en el siguiente caso: si en un agente queremos crear un agente hijo que realice un conjunto de acciones y existe otro agente en el sistema software que ya realiza un subconjunto de éstas. En este caso, creamos el nuevo agente mediante clonación y luego complementamos su creación añadiéndole las acciones que le faltan. Esto disminuirá el tiempo necesario para crear un agente nuevo.

Por ejemplo, en un sistema que modele una empresa de alquileres de coches podemos crear un agente *Empleado* que realice la acción *alquilar*, pero en la empresa puede existir más de un *Empleado* con las mismas funciones. Por tanto, primero creamos un agente *Empleado* basándonos en la definición de agente elemental, lo modificamos (le añadimos la acción *alquilar*) y cuando ya esté definido, lo clonamos tantas veces como agentes con esa misma funcionalidad queramos. De esta forma podemos crear *Empleado_2*, *Empleado_3*, etc. fácilmente. Con la notación dada anteriormente tendríamos un sistema con los siguientes agentes:

Sistema [alquilar] = { Empleado [alquilar], Empleado_2 [alquilar], Empleado_3 [alquilar], .. }

Empleado [alquilar] = { }

Empleado_2 [alquilar] = { }

Empleado_3 [alquilar = { }

Usamos la clonación porque no nos es útil tener una clase de agente tipo *Empleado* ya que, una vez en el sistema y mientras éste funciona,

cada agente puede evolucionar independientemente y pasar a realizar otras funciones que lo diferencien de los otros. Si tuviésemos una clase *Empleado*, cualquier modificación de los agentes de dicha clase sería común a todos y esto iría en contra de la independencia y autonomía que caracteriza a los agentes.

Además, la ventaja de la clonación es el ahorro de tener que definir varias veces el mismo tipo de agente y no sólo cuando se está construyendo el sistema sino en cualquier momento de su vida, durante su evolución.

7.3.4 Operación de agregación.

Como ya hemos visto, un sistema software se compone de una jerarquía de agentes. La estructura más sencilla de esta jerarquía es la de un árbol, aunque veremos más adelante que ésta puede ser más compleja. La raíz de este árbol la constituye el primer agente creado que es el que representa hacia el exterior el sistema software que se modela.

Inicialmente, sólo puede existir un nodo raíz, no varios, ya que esta estructura jerárquica representa a un único sistema software y no a varios. Esto no impide que si tenemos un sistema software ya constituido no pueda agregarse a otro pero como un agente más del conjunto de agentes que definen al sistema software.

Cuando se crea un agente, éste debe agregarse a la jerarquía de agentes. Si es un *agente simple*, aquel que no tiene agentes agregados y, por tanto, sólo realiza acciones simples, se agregará como una hoja del árbol. Si el agente realiza una acción compleja, será un nodo intermedio del árbol. La relación entre los agentes dentro de la jerarquía viene establecida por la definición de las acciones complejas que es capaz de realizar cada agente. El lugar que ocupe dependerá de a qué agentes se agregue según las acciones que realice. Cada acción transaccional que constituye la definición de una acción compleja puede ser realizada por un agente y este agente será hijo del agente que tiene en su interfaz a dicha acción compleja.

7.3.5 Agregación a más de un agente.

Un agente puede estar agregado a más de un agente, es decir, un agente puede tener más de un padre. Esto implica reutilización y flexibiliza la definición de un sistema software ya que permite que un agente trabaje para dos o más agentes sin tener que clonarlo o tener más agentes de los necesarios. Esta situación puede aparecer en el caso de que, basándonos en el ejemplo de alquiler de coches, queramos

representar un único agente, por ejemplo, un *Mecánico,* que puede trabajar en más de un departamento dentro de la empresa (así se ahorra tener que contratar más mecánicos, es decir, clonar).

El hecho de que un agente pueda tener más de un padre implica que, dependiendo de para quién trabaje en cada instante, se enviarán las ocurrencias de sus acciones a la *SFH* de un agente padre u otro. Incluso, como existe concurrencia intra-agente, en un determinado instante de tiempo puede estar realizando varias acciones para padres distintos. Por tanto, se plantea la siguiente cuestión: ¿cómo sabe un agente a qué padre enviar las ocurrencias de sus acciones? Dependerá del *Controller* que le envió la evaluación a *true* de la pre-condición de su acción (u acciones), ya que esta consulta se realizó sobre una historia determinada y basándose en su contenido (es similar al problema de las transacciones y de los *Transaction Manager*). Un agente conoce a los *Controller* de sus agentes padres y sabe a quien tiene que solicitar la evaluación de la pre-condición de una de sus acciones después de que el *Controller* lo active.

La agregación a más de un agente cambia la definición dada de la representación jerárquica del sistema software, ya no será un árbol sino un grafo dirigido. Se permite la ciclicidad pero existe una excepción: no se permite que un agente se agrege a sí mismo, es decir, que sea su propio padre. La ciclicidad se permite ya que un agente utiliza un conjunto de las acciones de su hijo y no necesariamente todas. Además, el agente hijo puede, a su vez, utilizar acciones que realiza su padre. Esto tiene sentido si pensamos que un agente puede tener distintos roles dentro del sistema y no sólo se crea para dar servicio a un único agente (se permite que un agente tenga varios padres).

En la siguiente figura se puede observar un sistema software con tres agentes agregados, *Agente_1*, *Agente_2* y *Agente_3*. Estos agentes tienen también agentes agregados. En particular el *Agente_5* está agregado a dos agentes, el *Agente_1* y el *Agente_2*. Las relaciones de agregación entre los distintos agentes que define al sistema se pueden expresar como:

Sistema[...] = { Agente_1[...], Agente_2[...], Agente_3[...] }

Agente_1[...] = { Agente_4[...], Agente_5[...] }

Agente_2[...] = { Agente_5[...], Agente_6[...] }

Agente_3[...] = { }

El resto de agentes, desde *Agente_4* al *Agente_7*, son también simples (no tienen agentes agregados), como el *Agente_3*.

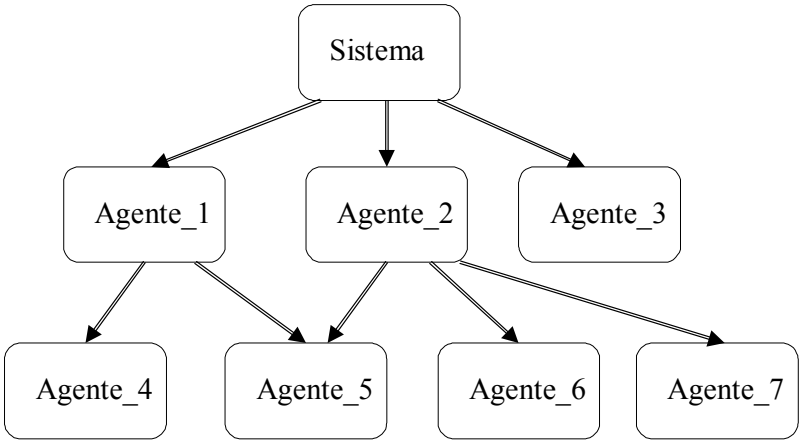


Figura 7.3 Grafo de agentes agregados

Esta operación de agregación múltiple no tiene restricciones en cuanto a quienes pueden ser los padres de un agente, puede ser cualquier agente de la jerarquía. Esto es así ya que cuando un agente se agrega a otro, conocerá quien es su padre y, por tanto, cuál es el agente *Controller* con quien tienen que comunicarse cuando trabaja para éste. Recordemos que es el agente *Controller* quien se encarga de la activación de los agentes, por tanto, un agente sabe desde qué *Controller* le llega la notificación de que evalúe la pre-condición de alguna de sus acciones. Después de la evaluación y si ésta es cierta y se realiza la acción. La ocurrencia de dicha acción se enviará de nuevo al *Controller* adecuado.

En la siguiente figura observamos, basándonos en la jerarquía de la figura anterior, la relación de comunicación entre los agentes y los *Controller* de sus agentes padres.

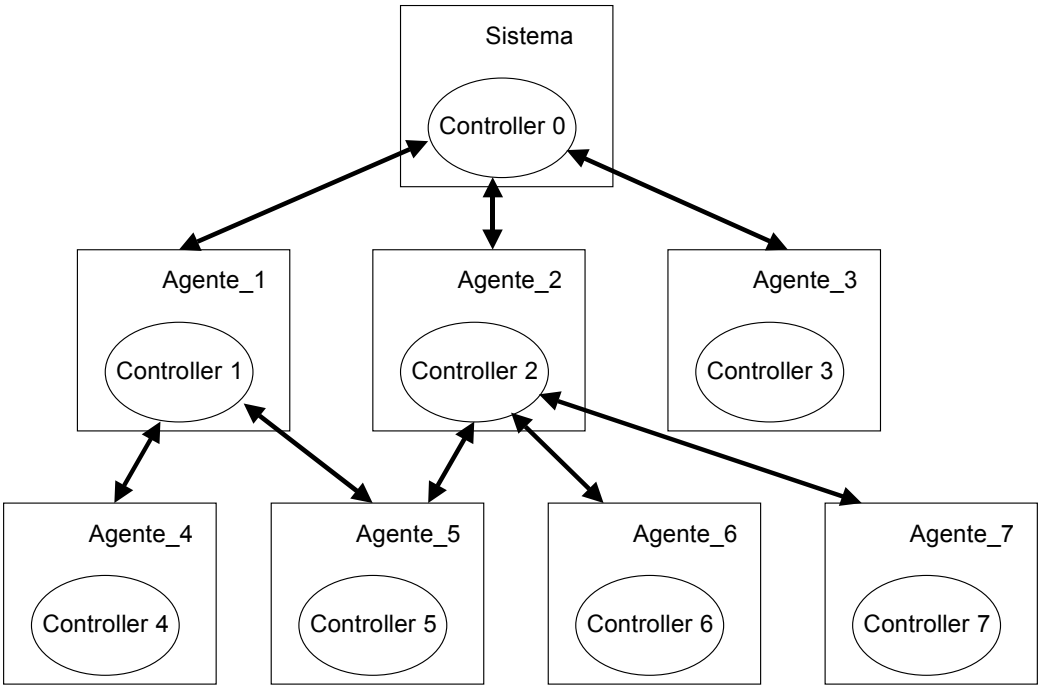


Figura 7.4 Relaciones de comunicación de los agentes con los Controller de sus agentes padres

7.3.6 Definición de acción compleja.

En la definición de una acción compleja o transacción se encuentran acciones (simples o complejas) que pueden ser realizadas por agentes distintos. No tendría sentido que todas las acciones transaccionales fueran acciones que realizara un único agente. Por tanto, la definición de una acción compleja dentro de un agente debe implicar que existan agentes agregados a él que realicen las acciones que la componen. Si alguna no existe, no se permitirá crear dicha acción compleja.

Una de las razones de tener acciones complejas es que queremos modelar la situación real en la que se pueden realizar distintas acciones concurrentemente, o no, con el fin de completar más eficientemente una tarea más compleja.

Un agente simple realiza una o más acciones simples. Un agente que realice una o más acciones complejas deberá tener uno o más agentes agregados. Estos agentes serán los que se encarguen de realizar las acciones transaccionales que constituyen la definición de la acción compleja. Si todas las acciones transaccionales son simples, ése será el

último nivel de su jerarquía, si no, se establecerá al menos otro nivel (para las acciones transaccionales complejas).

Vamos a intentar representar varios casos distintos que nos podemos encontrar en la definición de un agente:

Caso 1: supongamos que tenemos un agente, *Agente_1*, en cuya interfaz aparecen las acciones simples *a1*, *a2* y *a3*. Esto implica que el *Agente_1* puede tener hasta tres agentes agregados (vea la siguiente figura), de tal forma que cada uno de ellos realice respectivamente cada una de las acciones simples. Puede darse el caso de que una misma acción sea realizada por más de una agente agregado, como por ejemplo la acción *a1* que puede ser realizada por el *Agente_1.1* y por el *Agente_1.2*.

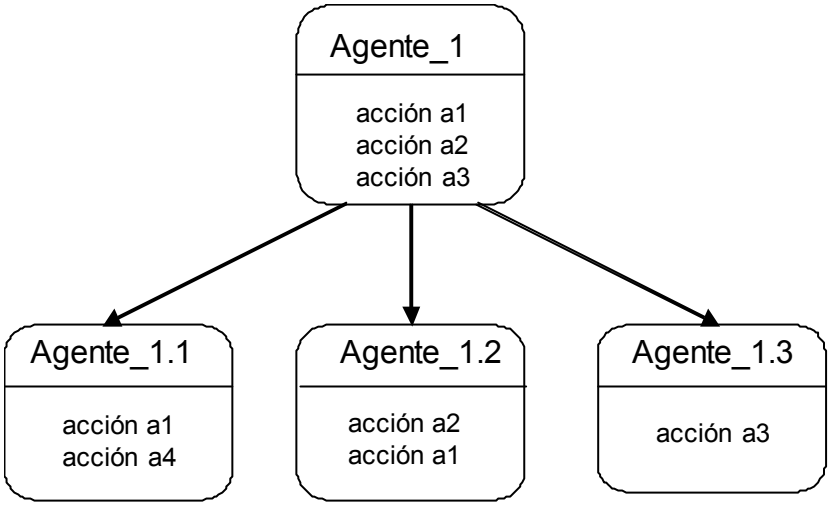


Figura 7.5 Relación jerárquica para el caso 1 a)

Aunque también podría tener sólo dos agentes, uno asociado a la acción *a1* y otro a las otras dos acciones o incluso podría no tener agentes agregados y ser él quien realice las tres acciones, ya que hemos supuesto que éstas son simples.

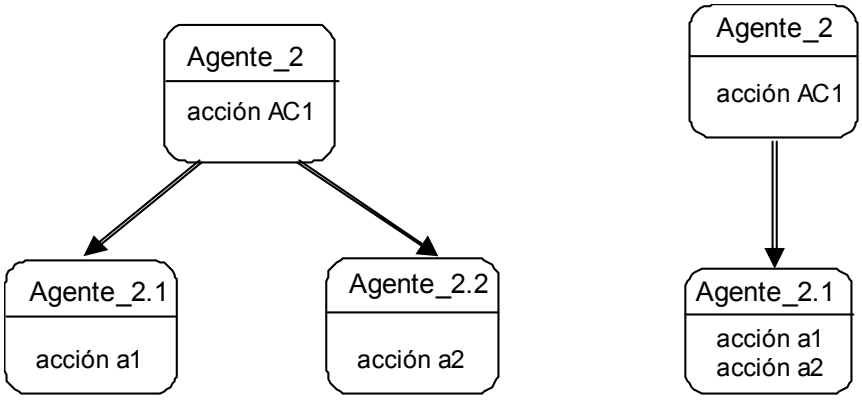
Es decir, podríamos tener tres definiciones de sistema distintas correspondiente a las tres configuraciones comentadas (aunque existen más posibilidades):

a) Agente_1[a1,a2,a3]={Agente_1.1[a1,a4],Agente_1.2[a1,a2], Agente_1.3[a3]}

b) Agente_1[a1,a2,a3] = {Agente_1.1[a1,a4], Agente_1.2[a2,a3] }

c) Agente_1[a1,a2,a3] = { }

Caso 2: suponemos un agente, *Agente _2* que tiene definida una acción compleja, *AC1*, que involucra a las acciones *a1* y *a2* (vea la siguiente figura). Esto obliga a tener dos agentes agregados que deben realizar las acciones simples *a1* y *a2* (grafo 1 en la figura) o a tener uno que realice ambas acciones simples (grafo 2 en la figura). Este último caso puede darse gracias a la existencia de concurrencia *intra-agente*, lo cual permite que varias acciones de un agente se puedan ejecutar al mismo tiempo, por tanto, se puede ejecutar una tarea compleja eficientemente (distintas subtareas se pueden llevar a cabo a la vez). Sin embargo, no tiene mucho sentido definir un agente con un único agente agregado, podría ser el agente padre quien se encargara de realizar las acciones simples y no es necesario definir una acción compleja.



Grafo 1 Grafo 2

Figura 7.6 Jerarquías de agentes para el caso 2

En la figura se representan dos grafos de agentes para el caso que hemos comentado. La especificación para cada uno de ellos es la siguiente:

Grafo 1: *Agente_2 [AC1] = { Agente_2.1 [a1], Agente_2.2 [a2] }*

Grafo 2: *Agente_2 [AC1] = { Agente_2.1 [a1,a2] }*

La acción compleja tendría, por ejemplo, la siguiente definición:

$$AC1 = a1 \;||\; a2$$

Como vemos, existe una relación entre la definición de la acción compleja y la jerarquía de agentes necesaria. Formalmente, podemos decir que debe cumplirse:

$$\forall\ A \in def(AC1)\ \ \exists\ P \in hijos(Agente\_2) \,/\, A \in interfaz(P)$$

Es decir, para toda acción que aparezca en la definición de una acción compleja, debe existir un agente hijo que la realice. Esto nos sirve para comprobar, cuando se define una acción compleja, si ésta es correcta según la jerarquía de agentes existentes.

Caso 3: suponemos ahora un agente que realiza una acción simple *a1* y una transacción *AC1*. Además dicha transacción involucra a otra transacción *AC2* definida como el conjunto de acciones simples *a2* y *a3*. Como podemos ver en la siguiente figura, esto ha dado lugar a un árbol de tres niveles al que podría corresponder la siguiente especificación:

Agente_3 [a1, AC1] = { Agente_3.1 [a1], Agente_3.2[AC2] }

Agente_3.2 [AC1] = { Agente_3.2.1 [a2], Agente_2.2[a3] }

Las definiciones de las acciones complejas son:

$AC1 = a1 \; ; \; AC2$

$AC2 = a2 \mid a3$

Para cada una de ellas debe cumplirse lo siguiente en relación con la especificación del agente que la realiza:

$\forall \; A \in def(AC1) \;\; \exists \; P \in hijos(Agente\_3) \; / \; A \in interfaz(P)$

$\forall \; A \in def(AC2) \;\; \exists \; P \in hijos(Agente\_3.2) \; / \; A \in interfaz(P)$

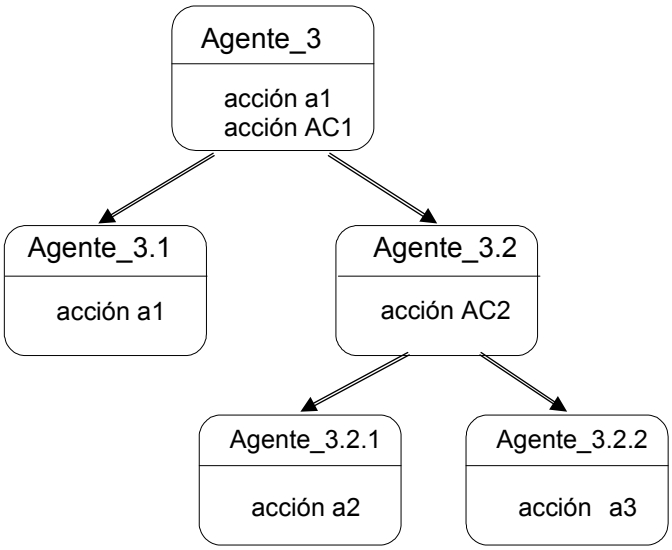


Figura 7.7 Jerarquía en el caso 3

La primera de las condiciones anteriores se cumple, ya que para la cada acción incluida en la definición de la acción compleja *AC1* existe un agente hijo que la realiza. La acción *a1* la realiza el agente *Agente_3.1* y la acción *AC2* la realiza *Agente_3.2*. Igualmente se puede comprobar que la segunda se cumple ya que las acciones *a2* y *a3* se realizan respectivamente por los agentes *Agente_3.2.1* y *Agente_3.2.2*.

7.3.7 Evolución en los agentes clonados.

Como hemos dicho, la operación de clonación de un agente supone la duplicación de éste. Lo único que cambia será el nombre del clon. Cuando el agente que queremos clonar tiene agentes agregados, se realiza una duplicación de todo el sub-árbol de agentes que definen al agente a clonar. Se hace una clonación en cascada, la única diferencia es que los descendientes no requieren un cambio de nombre debido a que su posición en la jerarquía no implica confusión.

Por ejemplo, si clonamos el *Agente_3* que hemos definido en el punto anterior, podemos crear un agente llamado *Agente_4* cuya definición será:

Agente_4 [a1, AC1] = { Agente_3.1 [a1], Agente_3.2[AC2] }

Agente_3.2 [AC1] = { Agente_3.2.1 [a2], Agente_2.2[a3]

AC1 = a1 ; AC2

AC2 = a2 | a3

Observamos que lo único que cambia de la copia realizada es el nombre del agente *Agente_3* por *Agente_4*, el resto de los nuevos agentes, permanece con igual nombre. Después de la clonación, *Agente_3* y *Agente_4* son idénticos. Sin embargo, a partir de este momento cada uno puede evolucionar independientemente aunque con ciertas restricciones. Veamos distintos casos que pueden surgir ante esta situación:

- *Agente_3* y *Agente_4* no son agentes hermanos, es decir, cada uno está situado en un lugar distinto del grafo de agentes. En este caso, cada uno puede evolucionar independientemente sin necesidad de ninguna comprobación, sin restricciones. El que *Agente_4* modifique su acción compleja *AC1*, no afectará al funcionamiento del *Agente_3*.

- *Agente_3* y *Agente_4* son hermanos. En este caso tenemos dos agentes que realizan las mismas acciones para su agente padre. Esta situación implica que las acciones con igual nombre de ambos deben

ser idénticas y, por tanto, no pueden modificarse independientemente. Por ejemplo, si *Agente_4* modifica la pre-condición de su acción compleja *AC1*, esta modificación tiene que realizarse también en la acción *AC1* de *Agente_3* o no realizarse en ninguno de ellos. Nótese que si no lo hacemos así, se podrían ejecutar en distintas circunstancias pero, después, la ocurrencia generada será la misma. Tendríamos a la vez dos pre-condiciones distintas para la misma acción. Tampoco sería factible que se modificara independientemente la definición o comportamiento de una de estas acciones o sus atributos. Esto se debe a que al evaluar una pre-condición lo que se comprueba es que exista una ocurrencia de dicha acción y no quién la realizó o qué atributos tiene dicha ocurrencia ya que ralentizaríamos mucho el funcionamiento del agente. Además, partimos de la premisa de que a un agente para activarse no le interesa conocer el agente que realiza una acción en el sistema software sino que dicha acción se ha realizado. Si dos agentes hermanos realizan una acción con igual nombre, es porque ambas son iguales, si no, deberían tener nombres distintos.

Por otro lado, con el fin de mantener el sistema software consistente y coherente en su diseño, cuando se va a añadir una acción a un agente con igual nombre que otra acción realizada por un agente hermano, se obliga a que sean idénticas. Realmente, se hace una copia de la del hermano.

Nótese que no importaría tener una acción con igual nombre en otro nivel de la jerarquía, ya que no afectaría al buen funcionamiento de los agentes.

7.4 Descripción de las acciones de evolución.

Basándonos en los trabajos de Rodríguez Fórtiz [Rodríguez00a] y partiendo de la definición de los lenguajes *L* y *M* utilizados para formalizar el subsistema de decisión, vamos a completar y modificar tanto la lista de invariantes que se deben cumplir cuando hacemos evolucionar un sistema software, como el conjunto de acciones estructurales permitidas. También es necesario incluir las acciones complejas o transacciones que no estaban contempladas en estos lenguajes. Adaptamos el trabajo realizado, a la definición de agente y a las consideraciones comentadas anteriormente.

En la figura siguiente recordamos que debe ser el equipo de desarrollo quien, a través del Metasistema, realiza modificaciones sobre la estructura del sistema software. En la figura podemos ver como un desarrollador quiere realizar la acción estructural “borrar un agente”,

delAgent. Esta decisión se lleva a cabo porque se introduce en la historia funcional del Metasistema un estímulo, *StimulusDelAgent*, que tiene información del agente que se desea borrar. Este hecho provoca la activación de la evaluación de la pre-condición de dicha acción de evolución y su ejecución en caso de que sea cierta. Recordemos que será el Sistema Genético del agente sobre el cual se va a realizar la acción de borrar un determinado agente hijo, el que realmente lleve a cabo dicha modificación.

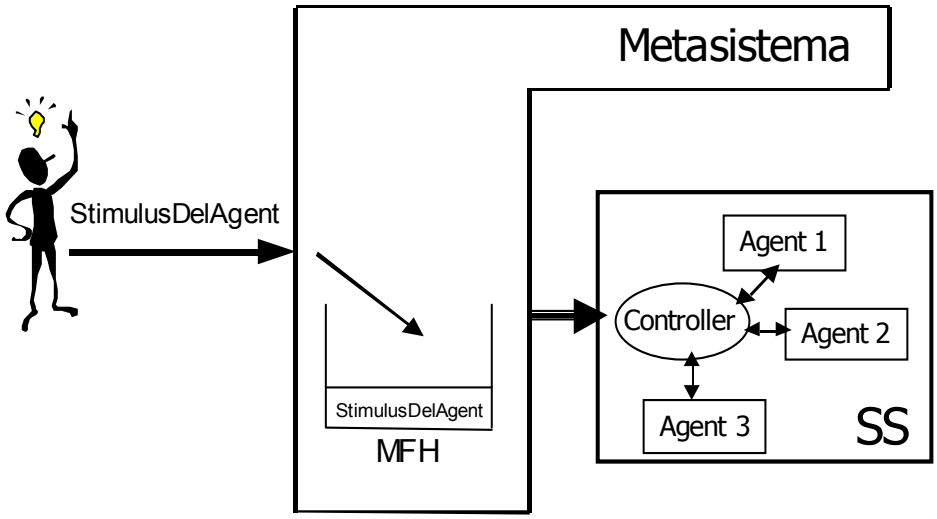


Figura 7.8 Activación de una acción estructural a través de la interfaz de acción del Metasistema

7.4.1 Lista de invariantes de un sistema.

A continuación vamos a presentar la lista de invariantes que definimos para un sistema software y especificamos el significado y justificación de cada uno de ellos.

1.- Nombres Únicos

Es necesario que se pueda referenciar cada elemento de un sistema software de forma única, sin ambigüedades. Como un sistema software está construido mediante una estructura jerárquica, este invariante debe cumplirse en el mismo nivel jerárquico en el cual se encuentre un determinado elemento.

- No pueden tener el mismo nombre dos agentes hermanos.
- No pueden tener el mismo nombre dos acciones (sean simples o complejas) asociadas a un mismo agente.

- Dos atributos de una misma acción no pueden tener el mismo nombre.

2.- Referencia

Cualquier elemento de un sistema software debe poder referenciarse. Si existe una referencia a un elemento del sistema software, éste debe existir. Si el sistema software ha evolucionado y un elemento es eliminado, éste no podrá referenciarse y cualquier referencia existente deberá eliminarse. Esto es coherente ya que como dejamos que toda acción iniciada termine antes de realizar una modificación en el sistema software, no puede ocurrir que una determinada acción referencie algo que ya no existe.

- Las acciones, atributos y agentes que no existan en un determinado instante de tiempo no deben ser referenciados, a no ser que hayan existido previamente y se quiera consultar el pasado.
- Todo atributo debe referenciarse indicando a qué acción pertenece.
- Un agente agregado a más de un agente es un único agente con varias referencias.
- Si se elimina una acción de un agente o un atributo de una acción, se deberán eliminar las referencias que se hagan de ellos en pre-condiciones, a no ser que exista otro agente hermano que realice también dicha acción.
- Hay que indicar en qué nivel de la jerarquía nos encontramos en cada momento.

3.- Caracterización por Atributos

Una acción simple o compleja debe tener al menos un atributo que la caracterice.

- Toda acción tiene como atributo el tiempo de realización de la acción según el reloj del sistema. Esto permitiría distinguir dos ocurrencias de la misma acción.
- Ninguna acción puede carecer de atributos. Debe tener, al menos, uno.

4.- Posibilidad de Realización de Acciones

Este invariante está relacionado con el hecho de que si existe una acción (simple o compleja) en el sistema software, debe garantizarse que, en algún momento, dicha acción podrá realizarse. El hecho de que la acción se realice porque su pre-condición fue cierta, no significa que

no se deshaga luego si su post-condición no se cumple. Si una acción no tiene post-condición, ésta se supondrá que es *true*.

- Cada acción debe tener asociada una única pre-condición.
- Cada acción puede tener asociada una post-condición, si no la tiene ésta se tomará como *true*. Si la tiene, será única.
- Una pre-condición debe referenciar acciones existentes en el sistema software o que hayan existido alguna vez y existan ocurrencias de ella en la historia. En otro caso, podría no realizarse la acción asociada ya que la pre-condición nunca será cierta.
- Las acciones que aparecen en la pre-condición de una acción deben existir en la interfaz del mismo agente que la realiza o en la de los agentes hermanos (en el mismo nivel de la jerarquía) ya que un sistema software está compuesto por un grafo de agentes. Si no fuera así nunca sabríamos que se ha realizado la acción ya que su ocurrencia se almacenaría en una historia distinta a la utilizada para comprobar si la pre-condición es cierta.
- Cada acción compleja debe tener asociada una definición no vacía.
- Toda acción transaccional (aquella que se encuentra dentro de la definición de una acción compleja) tiene que tener asociadas una pre-condición *t* y una pre-condición *s*.
- Si en alguna de las pre-condiciones *p*, *t* o *s* aparece la referencia de la realización de una acción, en las restantes pre-condiciones no puede aparecer la referencia a la no realización de la misma acción.

5.- Mantenimiento del Grafo de Agregación de Agentes

Este invariante está relacionado con el hecho de que la jerarquía de agentes que define a un sistema software debe ser consistente en todo momento.

- Todo agente se encuentra agregado, al menos, a otro agente, a no ser que se trate de un Sistema.
- Un agente no puede agregarse a sí mismo.
- La especificación de una acción compleja está relacionada con la jerarquía de agregación de agentes existentes. Al definir una acción compleja en un agente, deben existir agentes agregados (agentes hijos) que realicen cada una de las acciones (simples o complejas) que la componen. Si no fuese así, puede que nunca se pudiera llevar a cabo dicha acción compleja. Si cambiamos la jerarquía de agentes, podemos invalidar la definición de una acción compleja. Esto se puede especificar con la notación vista anteriormente como:

$$\forall AC \in interfaz(P)\ \forall acción \in def(AC), \exists Q \in hijos(P) / acción \in interfaz(Q)$$

donde *AC* representa una acción compleja, *P* y *Q* son agentes y *acción* denota una acción simple o compleja.

6.- Confluencia

Este invariante establece que si en un sistema software se realizaran las mismas acciones y en el mismo orden, siempre llega al mismo estado. Este invariante se asegura porque:

- La historia de cada agente contiene la traza lineal de acciones que han realizado sus agentes hijos.
- En cada instante de tiempo la pre-condición y la post-condición de una acción es única.

7.- Terminación

Debemos especificar las condiciones de terminación de un sistema software así como las condiciones de no terminación. Muchos sistemas a modelar se caracterizan porque no terminan nunca, y por tanto, su buen o mal funcionamiento no se puede medir por cómo finalicen sino por cómo trabajan. Por ejemplo, se debe evitar que un sistema entre en un estado de bloqueo indefinido ya que no realizaría ningún trabajo útil.

- Un sistema software inicialmente no terminará. Después de la creación, esperará a que se produzcan las condiciones oportunas para comenzar a trabajar. Una vez que ha empezado, por definición no terminará aunque habrá paradas funcionales para permitir su evolución. El hecho de que no se cumpla ninguna pre-condición de sus acciones en un instante dado no significa que nunca, en el futuro, se vayan a cumplir (ver el invariante 4).
- La terminación de un sistema software se puede provocar mediante el envío de un estímulo especial al agente Sistema a través de su interfaz de acción. El agente Sistema puede tener asociada una acción de terminación que sólo se activará al recibir dicho estímulo.
- Se deben evitar situaciones de bloqueo, recursividad y ciclos no controlados por el desarrollador, y por tanto, no deseados.

8.- Posibilidad de Consulta de Ocurrencias

Muchas pre-condiciones pueden implicar realizar consultas sobre la historia de un agente. Es necesario, por tanto, garantizar que cualquier pre-condición se pueda evaluar sobre la historia del agente.

- En la expresión de una consulta sobre la historia, sólo pueden referenciarse acciones que existan y que sean conocidas por el agente. Esto significa, que dentro de una consulta podemos citar acciones realizadas por los agentes que se encuentran dentro de un mismo nivel jerárquico.

  $\forall$ *acción in query of P,* ($\exists$ *Q* $\in$ *hermanos(P) /* *acción* $\in$ *interfaz(Q)*
  or acción $\in$ *interfaz(P)*)

- Si se hace referencia a atributos de acciones, éstos también tienen que existir.

9.- Unicidad de Acciones

Se puede dar el caso de que un agente tenga dos o más agentes hijos que realicen la misma acción (simple o compleja), es decir, que tengan una acción con igual nombre. Si dos o más agentes hermanos tienen una acción con igual nombre debe significar que son idénticas, tanto en funcionalidad, como en número y tipo de atributos. Cuando una de ellas se realiza, genera una ocurrencia y al comprobar el estado de la historia, sólo interesa conocer que dicha acción se realizó, no quién la realizó y si es o no diferente de la que se espera.

- Si dos acciones realizadas por agentes hermanos son distintas, deben tener nombres diferentes.
- Las pre-condiciones de dos acciones de agentes hermanos con igual nombre deben ser iguales.
- El número de atributos de las acciones con igual nombre realizadas por agentes hermanos debe coincidir.
- Las definiciones de las acciones complejas con igual nombre realizadas por agentes hermanos deben ser idénticas.
- La lista de las acciones incompatibles para una acción dada realizada por agentes hermanos debe ser la misma.

7.4.2 Acciones estructurales.

A continuación definiremos detalladamente cada una de las acciones estructurales que componen nuestro lenguaje *M2*.

En la descripción de cada acción estructural seguiremos el siguiente esquema:

- Primero definiremos brevemente la utilidad de la acción.

- Describiremos la pre-condición asociada a la acción utilizando el conjunto de predicados del lenguaje *M2* y de los predicados auxiliares (también ampliado) que facilitarán la tarea de descripción. La pre-condición determina las condiciones que deben existir en el sistema software para que la acción pueda llevarse a cabo. Comentaremos para cada una de ellas los invariantes que se pretenden preservar.

- Presentaremos la post-condición asociada, en caso de que exista. En ella se especifican las comprobaciones que se llevarán a cabo para verificar que la realización de la acción cumple todos los invariantes definidos en el sistema software y se garantice, por tanto, su integridad. Igual que en el caso anterior, citaremos los invariantes que se pretenden preservar.

- Por último, presentamos un conjunto de acciones que se realizan en la ejecución de la acción estructural, a esta sección la titulamos *acciones a realizar*. Este conjunto de acciones tiene dos objetivos:

    - Iniciar otras acciones estructurales que deben realizarse en el sistema software con el fin de hacer cumplir la post-condición asociada a la acción estructural que se está llevando a cabo, si es posible. En este caso, el agente *Meta-Agent* envía, por cada una de dichas acciones estructurales, el estímulo apropiado al *Controller* del Metasistema para que se evalúe también su pre-condición.

    - Ejecutar algunas acciones internas para preservar uno o más invariantes del sistema software y para simplificar el trabajo del Metasistema. Estas acciones implican que uno o más Sistemas Genéticos realicen modificaciones en sus respectivos agentes. El agente *Meta-Agent* envía al *Controller* de los agentes donde se tienen que realizar la o las modificaciones, el o los estímulos necesarios. La diferencia con el caso anterior es que no necesitan pasar por la evaluación del Metasistema.

  Este conjunto de acciones puede verse como un mecanismo de propagación de los cambios estructurales realizados por el Metasistema que garantiza que el sistema software quedará en un estado consistente. La acción estructural no se da por terminada hasta que se hayan realizado todas estas acciones y se compruebe que la post-condición se cumple.

7.4.2.1 Notación utilizada.

Con el objetivo de simplificar la exposición, y de clarificar las especificaciones de las pre y post-condiciones, presentaremos la notación que utilizaremos para referenciar agentes, acciones, atributos, pre-condiciones y post-condiciones.

- *Acciones*: Las primeras letras mayúsculas del alfabeto (*A, B, C*, ...) representarán a las acciones tanto simples como complejas. Si es necesario diferenciarlas, se añadirá *_s* para las simples y *_c* para las complejas. Por ejemplo, *A_s* es una acción simple y *A_c* se corresponde con una acción compleja. Si diera igual, sólo usaríamos *A*.

- *Definición de acciones complejas*:. Se podrá el nombre de la acción seguida de un símbolo igual y a continuación la definición de dicha acción utilizando el lenguaje TDL introducido en el capítulo 5. Por ejemplo:

  A = B ; (C || D)

  Esta expresión significa que la acción compleja *A* se constituye de tres acciones *B*, *C* y *D*. El orden de ejecución de éstas es: primero ha de realizarse *B* y después, paralelamente *C* y *D*.

- *Estímulos*: La palabra *Stímulus* seguida del nombre del estímulo nos da información de la acción que provoca. Si es necesario, se pondrá entre paréntesis el o los valores de los atributos que tiene dicho estímulo. Por ejemplo, *StimulusDevolver(C)* es un estímulo procedente de la devolución de un coche, el coche *C* (ver el ejemplo del sistema de alquiler de coches en el capítulo 8). Si existen varios estímulos asociados a una misma acción no tendremos problemas en distinguirlos ya que tendrán valores de atributos distintos.

- *Atributos de una acción o de un estímulo*: Utilizaremos letras minúsculas separadas por comas y entre paréntesis. Los situaremos a la derecha de la acción o del estímulo. Con *A(x,y,z)* indicamos que la acción *A* tiene tres atributos: *x,y,z.*

- *Valores de atributos*: Serán cadenas de caracteres entre comillas: *'a', '1'* son ejemplos de valores de atributos.

- *Ocurrencias de acciones*: Representan acciones instanciadas y, por tanto, las expresamos con valores de atributos entre paréntesis detrás de la acción. *A('1','a','bb')* y *A('2','b','d')* pueden ser ejemplos de dos ocurrencias diferentes de la acción *A*, almacenadas en la *SFH*. Un atributo fijo que tienen tanto las acciones como los atributos es el tiempo de realización (primer atributo en los ejemplos). Los estímulos

también se instancian de la misma forma. *StimulusA*(*('5','b','d')* es un estímulo que el agente ha recibido para realizar la acción *A* con esos valores en sus atributos.

- *Agentes*: Utilizaremos las últimas letras mayúsculas del alfabeto para representar a los agentes. Detrás del nombre del agente se ponen entre corchetes y separadas por comas las acciones que realiza. Por ejemplo, si los agentes *P* y *Q* tienen asignada una acción *A* y, además, *P* tiene asignada una acción *B* , lo representamos mediante: *P*[A,B] y *Q[A]*.

- *Pre-condiciones*: Las representaremos mediante expresiones del tipo

  $$A \leftarrow X$$

  a las que llamaremos *cláusulas*. Con ellas expresamos que el cuerpo, *X*, es la pre-condición de la acción *A*, la cabeza. Si *X* se evalúa a *true*, entonces *A* podrá ser realizada por el agente del sistema software que le corresponda.

  X se compone de un conjunto de nombres de acciones y/o estímulos separados mediante operadores de la LTP. De esta forma se representan las relaciones de orden temporales o lógicas que existen entre las acciones y/o estímulos que se realizan en un sistema software. Como ejemplo vemos que *X = B and C* es una pre-condición que referencia a la ocurrencia de dos acciones relacionadas:

  $$A \leftarrow B \text{ and } C$$

  Con esta cláusula hacemos referencia a que, para que la acción *A* se realice, las acciones *B* y *C* deben haberse realizado con anterioridad a ésta.

- *Post-condiciones*: Las representaremos mediante cláusulas del mismo tipo que las comentadas para las pre-condiciones. Aunque, en este caso, el nombre de la acción constituye ahora el cuerpo de la cláusula y la cabeza es la post-condición a cumplir. Esta condición se expresa utilizando LTP y un conjunto de nombres de acciones. Por ejemplo:

  $$Y \text{ and } Z \leftarrow D$$

  Con ellas expresamos que *Y and Z* es la post-condición de la acción *D*. Si *Y and Z* se evalúa a *true*, entonces podemos afirmar que la post-condición de la acción *D* se ha cumplido.

7.4.2.2 Lista de acciones estructurales.

Resumimos a continuación las operaciones de modificación de la estructura de un agente. Las hemos dividido en tres categorías dependiendo de los elementos de la estructura a los que afecte.

- Sobre las acciones del agente y sus pre-condiciones:

    - Añadir una acción simple *addSAction(A_s,P)*
    - Añadir una acción compleja *addCAction(A_c,P)*
    - Añadir o sustituir una pre-condición *p* a una acción (simple o compleja) *addPPrec(X,A,P)*
    - Añadir o sustituir una pre-condición *t* a una acción transaccional *addTPrec(X,A,B,P)*
    - Añadir o sustituir la lista de acciones incompatibles de una acción (simple o compleja) a*ddLAI(list,A,P)*
    - Borrar una acción simple o compleja *delAction(A,P)*
    - Modificar la definición de una acción compleja *defCAction(def,A_c,P)*
    - Cambiar de nombre una acción *renameAction(A,B,P)*

- Sobre los atributos de las acciones:

    - Añadir un atributo a una acción *addAtt(att,A,P)*
    - Borrar un atributo de una acción *delAtt(att,A,P)*
    - Cambiar de nombre a un atributo *renameAtt(att,att2,A,P)*

- Sobre la existencia de agentes y la jerarquía de agentes:

    - Clonar un agente ya existente *cloneAgent(P,Q)*
    - Crear un agente *createAgent(P,Q)*
    - Crear un sistema (un agente sin padre) *createSystem(P)*
    - Agregar un agente a otro *agrAgent(P,Q)*
    - Desagregar un agente de otro *disAgrAgent(P,Q)*
    - Borrar un agente *delAgent(P)*
    - Cambiar de nombre a un agente *renameAgent(P,Q)*
    - Mover una acción a otro agente *moveAction(A,P,Q)*
    - Mover un agente *moveAgent(P,Q)*

7.4.2.3 Predicados auxiliares.

Con el fin de simplificar la presentación de las pre-condiciones de las acciones estructurales, utilizamos un conjunto de predicados auxiliares. En la tabla 7.1 se puede ver un resumen de dichos predicados, así como su significado.

En resumen, la finalidad estos predicados auxiliares es informar del estado de existencia de determinadas acciones, agentes, pre-condiciones o atributos. Así, por ejemplo, la evaluación del predicado *exist* serviría para saber si un agente existe o no en un determinado momento, es decir, está informando sobre el estado. En realidad, para conocer el estado debemos consultar la historia y ver qué acciones estructurales se han hecho con anterioridad, observando cuál ha sido la relación temporal entre esas acciones. Para el predicado *exist* se vería si se ha creado un agente, si se ha borrado, si se ha renombrado desde que se creó,...

PREDICADOS AUXILIARES DEL LENGUAJE *M2*	
IsInSSHA(A,P)	Comprobar si la acción simple o compleja *A* del agente *P* existe
IsInSSHP(X,Q)	Comprobar si las acciones referenciadas en la pre-condición *X* de un agente *Q* son realizadas por agentes hermanos
isPrec(X,A,P)	Comprobar si *X* es la pre-condición de la acción *A* de un agente *P*
isTPrec(X,A,B,P)	Comprobar si *X* es la pre-condición de la acción transaccional *A* que forma parte de la transacción *B* de un agente *P*
exist(P)	Comprobar si existe el agente *P* en el nivel de la jerarquía de agentes actual
IsBrother(P,Q)	Comprobar si el agente *P* es hermano del agente *Q* (igual nivel en la jerarquía de agentes)
isFather(P,Q)	Comprobar si el agente *P* es padre del agente *Q*
hasAtt(att,A,P)	Comprobar si el atributo *att* existe en la acción *A* de un agente *P*
lastAtt(att,A,P)	Comprobar que *att* es el único atributo que tiene la acción *A* del agente *P*
isAgrAgent(P,Q)	Comprobar si el agente *P* está agregado al agente *Q*
component(A,X)	Comprobar si la acción *A* está referenciada en la pre-condición *X*
isIn(B,A,P)	Comprobar si la acción *B* se encuentra en la definición de la acción compleja *A* del agente *P*.
isInFH(A,P)	Comprueba si existe una ocurrencia de la acción *A* realizada por el agente *P* en la SFH
IsAncestor(P,Q)	Comprueba si el agente P es ascendiente del agente Q
IsInPrec(B,A,P)	Comprobar si *B* forma parte de la pre-condición de la acción *A* del agente *P*

Tabla 7.1 Lista de los predicados auxiliares

A continuación veremos la descripción más detallada de cada uno de estos predicados auxiliares.

***isInSSHA(A,P)* :**

Con este predicado se comprueba si existe una acción *A* en el agente *P* del sistema software. Esta acción puede ser simple o compleja.

- Comprueba si la acción *A* ha sido añadida previamente al agente *P* del sistema y no ha sido borrada desde entonces. Informará también de que, en el caso de que haya sido borrada, si se ha vuelto a crear después.

  addAction(A,P) and not (delAction(A,P) since addAction(A,P))

- También puede haber ocurrido que se haya movido esta acción *A* a otro agente o que se hay producido un cambio de nombre desde que se añadió.

  not ∃ Q (moveAction(A,P,Q) since addAction(A,P)) or
  (renameAction(A,B,P) since addAction(A,P))

- Por último se comprueba el caso de que la acción *A* fuera el nuevo nombre de una acción *B* anterior o el nombre de una acción antigua *C* que ahora pasa a llamarse *B*.

  not ∃ B ∃ C (delAction(A,P) since renameAction(B,A,P)) or
  (renameAction(A,C,P) since renameAction(B,A,P))

En resumen, este predicado auxiliar será *true* si se cumple lo siguiente:

(addAction(A,P) and not (delAction(A,P) since addAction(A,P)) or
(not ∃ Q (moveAction(A,P,Q) since addAction(A,P)) or
(renameAction(A,B,P)) since addAction(A,P)))) or
(not ∃ B ∃ C (delAction(A,P) since renameAction(B,A,P)) or
(renameAction(A,C,P) since renameAction(B,A,P)))

isInSSHP(X,Q):

Debe comprobar que todas las acciones que forman parte de la pre-condición *X* de un agente Q sean realizadas por agentes que están en el mismo nivel de la jerarquía que el agente al cual pertenece la acción X o por él mismo. Equivale a :

∀A isInPrec(A,X,Q) ((∃ P isBrother(P,Q) and isInSSHA(A,P)) or isInSSHA(A,Q))

isPrec(X,A,P):

Se comprueba que no se haya cambiado una pre-condición de una acción *A* para un agente *P*, verificando que su pre-condición vigente sea *X*.

not ∃ Y (addPrec(Y,A,P) since addPrec(X,A,P)

isPrecT(X,A,B,P):

Se evalúa a *true* si la pre-condición *X* de la acción transaccional *A* que se encuentra en la definición de la transacción *B* del agente *P* no ha cambiado.

not ∃ Y (addTPrec(Y,A,B,P) since addTPrec(X,A,B,P))

exist(P):

Este predicado sirve para comprobar si existe algún agente llamado *P* en el nivel de jerarquía de agentes actual. Cuando el desarrollador realiza una acción de evolución se encuentra situado en un determinado nivel de la jerarquía (el concepto es similar al de *camino de ruta* de un sistema de directorios en los Sistemas Operativos actuales). De esta forma, los nombres de los agentes son realmente únicos porque está implícito todo el recorrido del árbol de agentes. Así si tenemos el siguiente sistema:

Sistema[...] = { P[...], Q[...], R[...] }

P[...] = { S[...], T [...] }

Q[...] = { P[...], Q [...] }

Vemos que existen dos agentes *P* pero en distinto nivel de la jerarquía. Ambos son distintos porque realmente sus nombres serían:

Sistema.P

Sistema.Q.P

Aunque para simplificar, permitimos decir simplemente agente *P* y siempre mantenemos el nivel actual de la jerarquía de agentes sobre el cual está trabajando el desarrollador y, por tanto, el Metasistema.

- Hay que verificar que el agente no haya sido borrado ni desagregado desde que se creó o se agregó o se renombró al agente padre.

not ∃ Q (isFather (Q,P) and (delAgent(P) since (createAgent(P,Q) or agrAgent(P,Q) or (∃ R (renameAgent(R,P)) and (disAgrAgent(P) since (createAgent(P,Q) or agrAgent(P,Q) or (∃ R (renameAgent(R,P))

- En el caso de que el agente haya sido renombrado, hay que comprobar que después del cambio no se haya borrado ni vuelto a cambiar su nombre.

∃ S (renameAgent(P,S) since (createAgent(P,Q) or agrAgent(P,Q) or (∃ R (renameAgent(R,P))

En general:

(not ∃ Q (isFather (Q,P) and (delAgent(P) since (createAgent(P,Q) or agrAgent(P,Q) or (∃ R (renameAgent(R,P)) and (disAgrAgent(P) since (createAgent(P,Q) or agrAgent(P,Q) or (∃ R (renameAgent(R,P))) and (∃ S (renameAgent(P,S) since (createAgent(P,Q) or agrAgent(P,Q) or (∃ R (renameAgent(R,P)))

isBrother (P,Q):

Comprueba si dos agentes están en el mismo nivel del árbol de agentes que determina la estructura de un sistema software. Si esto es cierto se dice que ambos son agentes hermanos.

Devolverá *true* si ambos tienen el mismo padre.

isFather(R,P) and isFather(R,Q)

isFather (P,Q):

Comprueba si existe una relación padre-hijo entre ambos agentes según el árbol de agentes que determina la estructura de un sistema.

Devolverá *true* si *P* es un agente padre de *Q*. Para ello sólo tenemos que comprobar si en el agente P existe un agente llamado Q.

exist(Sistema. … P.Q)

hasAtt(att,A,P):

Informa de la existencia o no del atributo *att* de la acción *A* del agente *P*.

- Se verifica si no se borró el atributo *att* desde que se añadió.

not (delAtt(att,A,P) since addAtt(att,A,P))

- Si no ha cambiado de nombre a *att2* desde que se añadió a *A*.

  not ∃ att2 (renameAtt(att,att2,A, P) since addAtt(att,A,P))

- Y, por último si no se ha borrado desde que otro atributo, *att2,* se renombró a *att* ni se ha vuelto a renombrar con otro nombre, *att3*.

  not ∃ att3 att2 (delAtt(att,A,P) since renameAtt(att3,att,P))
  or (renameAtt(att,att3,A,P) since renameAtt(att2,att,P))

Todas las condiciones a evaluar para este predicado serán:

(not ((delAtt(att,A,P) since addAtt(att,A,P))
or ∃ att2 (renameAtt(att,att2,A, P)) since addAtt(att,A,P))) or
(not ∃ att3 ∃ att2 (delAtt(att,A,P) since renameAtt(att2,Att,P))
or (renameAtt(att,att3,A,P) since renameAtt(att2,Att,P)))

lastAtt(att,A,P):

Se comprueba que la acción *A* del agente P tiene un único atributo *att*.

hasAttAct(att,A,P) and not ∃ att2 hastAttAct(att2,A,P)

isAgrAgent(Q,P):

Se comprueba si un agente *Q* está agregado a otro *P*.

Se considerará agregado si no se ha desagregado o se ha borrado desde que se agregó, se creó, se renombró o se movió.

not ∃ R (disAgrAgent(Q,P) or delAgent(Q,P)) since (agrAgent(Q,P) or createAgent(Q,P) or renameAgent(Q,R) or moveAgent(Q,P,R))

component(A,X):

Con este predicado se pretende comprobar si hay una relación entre una acción y una pre-condición. Esta relación consiste en ver si aparece el predicado de una acción *A* en la sintaxis de una pre-condición *X*, evaluándose a *true* cuando así ocurre.

isIn (B,A,P):

Indica si la acción *B* forma parte de la definición de la acción compleja *A* del procesador *P*, evaluándose a *true* si se da esta circunstancia.

Debe existir la acción *A* en el agente actual, *P*, y se ha tenido que dar valor a su definición. Además, en esta definición se debe aparecer la acción *B*.

isInSSHA(A,P) and addCAction(A,P) and defCAction(def,A,P) and B∈def(A)

isInPrec(B,A,P):

Al evaluar este predicado obtenemos *true* si la acción *B* aparece refereciada en la pre-condición que se ha asociado a la acción *A* del agente *P*.

isInSH(A,P):

Al evaluar este predicado obtenemos *true* si existe una ocurrencia de la acción *A* en la historia funcional del sistema realizada por el agente *P*.

isAncestor(P,Q):

Este predicado devuelve *true* si el agente *P* es un ascendiente (padre, abuelo, etc.) respecto a la jerarquía de agentes del agente *Q*.

isFather(P,Q) or (∃ R isFather(P,R) and isFather(R,Q)) or
(∃ S isFather(P,S) and isAncestor(S,Q))

7.4.2.4 Añadir una acción simple *(addSAction)*.

Esta acción indica explícitamente que se desea añadir una acción simple a un agente. Cuando esta operación finalice, esta acción debe tener al menos un atributo y tiene que tener asociada una pre-condición *p*.

Pre-condición de *addSAction(A_s,P)*:

- Debe existir el agente, *P*, al cual se va a añadir la acción simple (invariante de Referencia).

  exist(P)

- El nombre de la acción a añadir no puede coincidir con el nombre de ninguna acción simple o compleja ya existente en el agente *P* (invariante de Nombres Únicos).

  not isInSSHA (A_s,P)

- Debe existir un estímulo para dicha acción llamado *StimulusAddSAction*.

La pre-condición completa es:

addSAction (A_s,P) ← *exist(P) and not isInSSHA (A_s,P) and*
StimulusAddSAction(A_s,P)

Post-condición de *addSAction(A_s,P)*:

- Toda acción simple tiene que tener asociada una pre-condición *p* (otra opción sería asociarle por defecto un valor en su creación, por ejemplo *false*, pero el desarrollador podría olvidarse de darle otro valor más significativo y dicha acción podría no realizarse nunca. Además si esta acción forma parte de la pre-condición de otra, ésta tampoco se realizaría). Esto garantiza que se cumpla el invariante de Posibilidad de Realización de Acciones.

  addPPrec(X,A_s,P)

- Si existe algún agente hermano que tenga en su interfaz una acción con igual nombre, éstas deben ser iguales ya que si no, y dado que las ocurrencias de acción se llamarán igual, puede originarse algún problema de inconsistencia. Por tanto, la acción añadida debe coincidir tanto en funcionalidad como en número y tipo de atributos con cualquier otra acción con igual nombre realizada por un agente hermano (invariante de Unicidad de Acciones).

  ∀ Q isBrother(P,Q) and isInSSHA(A_s,Q), P(A_s) = Q(A_s)

  donde *P(A_s)* denota la acción *A_s* del agente *P*.

La post-condición completa sería:

addPPrec(X,A_s,P) and (∀ Q isBrother(P,Q) and isInSSHA(A_s,Q), P(A_s)=Q(A_s)) ← *addSAction (A_s, P)*

Acciones a realizar en *addSAction*:

- Como toda acción debe tener como atributo el tiempo de realización, *tr*, esta acción estructural crea automáticamente una acción simple con dicho atributo. De esta forma se cumple el invariante de Caracterización por Atributos. Después se pueden añadir más atributos, si el desarrollador lo considera necesario, con la acción estructural *addAtt*.

- Se genera un estímulo, *StimulusAddPPrec(X,A_s,P)* para que se active la acción estructural *addPPrec* con el fin de asociarle una pre-condición *p* a la acción simple recientemente creada.
- En el caso en el que exista una acción con igual nombre realizada por un agente hermano, se realiza una copia de ésta y los pasos anteriores no se llevan a cabo. Lógicamente, se le da valor *true* a la post-condición de *addSAction*.

7.4.2.5 Añadir una acción compleja (*addCAction*).

Esta acción indica explícitamente que se desea añadir una acción compleja a un agente. Al igual que la acción anterior, ésta tendrá al menos un atributo y tiene que tener asociada una pre-condición.

Pre-condición de *addCAction(A_c,P)*:

- Debe existir el agente al cual se va a añadir la acción compleja (invariante de Referencia).

  exist(P)

- La acción a añadir no debe existir previamente (invariante de Nombres Únicos) ni puede existir una acción simple con igual nombre.

  not isInSSHA (A_c,P)

- Debe existir un estímulo para dicha acción: *StimulusAddCAction.*

La pre-condición completa es:

addCAction (A_c, P) ← *exist(P) and not isInSSHA (A_c,P) and*
StimulusAddCAction(A_c, P)

Post-condición de *addCAction (A_c,P)*:

- Cada acción compleja tiene asociada una pre-condición *p* por los mismos motivos que los expuestos en el caso de la acción estructural anterior. Esto garantiza que se cumpla el invariante Posibilidad de Realización de Acciones.

  addPPrec(X,A_c,P)

- Como vimos en la acción anterior, si existe algún agente hermano que tenga en su interfaz una acción con igual nombre, éstas deben ser iguales (invariante de Unicidad de Acciones). En este caso, la

acción añadida debe coincidir tanto en funcionalidad, es decir, igual definición, como en número y tipo de atributos con cualquier otra acción con igual nombre realizada por un agente hermano.

∀ Q isBrother(P,Q) and isInSSHA(A_c,Q), P(A_c)=Q(A_c)

La post-condición completa es:

addPPrec(X,A_c,P) and (∀ Q isBrother(P,Q) and isInSSHA(A_c,Q), P(A_c)=Q(A_c)) ← addCAction (A_c, P)

Acciones a realizar en *addCAction*:

- Igual que en la acción estructural anterior, se asocia automáticamente el atributo tiempo de realización, *tr* (invariante de Caracterización por Atributos).
- Toda acción compleja debe tener asociada una definición donde se numeran las acciones transaccionales que la componen y el orden de realización de éstas. Esto garantiza el invariante de Posibilidad de Realización de Acciones. En esta acción estructural se construye y asocia una definición por defecto a la acción compleja que se está creando. Esta definición consta de una única acción, la acción *terminar*. La acción *terminar* es especial, existe con el objeto de indicarle al *Transaction Manager* que se crea para la gestión de una acción compleja, que ésta ha terminado con éxito. Después, es responsabilidad del equipo de desarrollo modificar su definición cuando se creen los agentes que realizarán las acciones transaccionales que la componen. Se asocia a la acción terminar la pre-condición *t* con valor *true*.
- Al igual que comentamos para el caso de las acciones simples, se genera un estímulo *StimulusAddPPrec* que activará la acción estructural *addPPrec*, con el fin de asociarle a la acción compleja creada su pre-condición *p*.
- Si existe una acción compleja con igual nombre en un agente hermano, se realizará una copia de la acción compleja. Pero no sólo en cuanto a su definición, atributos y pre-condiciones sino que necesitamos hacer una copia de los sub-agentes que llevan a cabo las acciones transaccionales. En este caso los pasos anteriores no se realizarían. Lógicamente, se le da valor *true* a la post-condición de *addSAction*.

7.4.2.6 Añadir o sustituir una pre-condición *p* a una acción simple o compleja *(addPPrec).*

Una pre-condición *p* se representa con una fórmula de la lógica temporal compuesta por predicados (acciones simples o complejas) y operadores. Añadir una pre-condición *p* significa anular su pre-condición anterior (invariante de Confluencia). Puede ser que la acción se esté creando y no tenga asociada ninguna pre-condición, en este caso, simplemente significa añadirla.

Pre-condición de *addPPrec(X,A,P):*

Las siguientes comprobaciones necesarias antes de añadir una pre-condición *p* a una acción ya existente se unirán con un *and* pero por simplificar el razonamiento, las dividimos en distintos puntos:

a) *Existencia previa*: la acción debe existir y la pre-condición de una acción debe estar formada por otras acciones existentes en el agente al cual pertenece la acción o en sus agentes hermanos (invariante de Posibilidad de Realización de Acciones).

isInSSHA(A,P) and isInSSHP(X,P)

b) *Recursividad*: Para asegurar el invariante de Terminación hay que comprobar que el sistema software no entre en ciclos ni se bloquee.

La pre-condición de una acción puede contenerla a ella misma sólo si la acción ya se ha realizado alguna vez, ya que en otro caso la acción nunca se llevaría a cabo y se podría producir una situación de bloqueo (Invariante de Terminación). Esto significa que la recursividad está permitida si se aplica la regla con carácter retroactivo (*backward*) y en la historia funcional del sistema (*SFH*) hay ocurrencias de la acción que provocan la recursividad. En el caso en el que no se considere el pasado, la recursividad se permitirá sólo si la acción aparece ligada al resto de la pre-condición con el operador *or*. Así, al menos se garantiza que el resto de la pre-condición pueda ser *true*, con lo que la acción se podría realizar al menos una vez y a partir de ahí podría repetirse su realización.

Sólo se permite recursividad si la acción está en la *SFH:*

(component(A,X) and isInFH(A,P)) or not component(A,X)

Solamente el caso en que la acción no tenga ocurrencias y aparezca ligada al resto de la pre-condición con el operador *or* podría dar pie a que el resto de la pre-condición pudiera evaluarse a *true* y, por tanto, sí podría

realizarse la acción. El que la acción no tenga ocurrencias sobre las que se evalúe la pre-condición puede deberse a que se evalúe con carácter retroactivo pero no haya ocurrencias pasadas o a que se tenga en cuenta su aplicación a partir del momento en que se añade.

c) *Recusividad negada para garantizar unicidad de ejecución de una acción:*

Quizás nos interese que una acción se realice sólo una vez. Sólo en este caso la pre-condición de una acción la incluirá a ella misma de forma negada, garantizando que tras su ejecución su pre-condición pase a ser *false* (Invariante de Terminación).

Se permite recursividad negada cuando la acción no se haya realizado anteriormente, es decir, no aparezca en la *SFH*. Se aplica con carácter retroactivo pues hay que considerar las ocurrencias pasados.

(component(not A,X) and not isInFH(A,P)) or not component(not A, X)

Si la acción no tiene ocurrencias, o no consideramos el pasado, aseguramos que la acción, de realizarse, sólo se hará una vez. Si la acción ya tiene ocurrencias, al aplicarla con carácter retroactivo provocamos que la acción no se realice nunca más.

d) *Ciclos recursivos cortos:*

Sea *A* pre-condición de *B*, si a su vez *B* es pre-condición de *A*, entonces hablamos de *ciclo corto*. Sólo admitiremos esta situación en el caso de que alguna de las acciones ya se haya realizado alguna vez. En otro caso, ninguna de las acciones podría realizarse al no satisfacerse su pre-condición (Invariante de Terminación).

∃ B ∃ Q ∃ Y (((isPrec(Y,B,Q) and component(A,Y)) and component(B,X)) and (isInFH(A,P) or isInFH(B,Q))) or not ((isPrec(Y,B,Q) and component(A,Y)) and component(B,X))

La pre-condición vista se evalúa sólo si se va a aplicar la pre-condición X añadida en modo backward. En otro caso, no habría ocurrencias que considerar, luego al igual que en el caso 2, sólo se puede garantizar que la pre-condición pueda evaluarse a true cuando alguna de las acciones que provocan el ciclo esté unida al resto de su pre-condición con el operador or.

e) *Ciclos recursivos largos:*

Llamaremos *ciclo largo* a una situación de pre-condición en la que intervienen tres o más acciones de forma que, en el caso en el que haya tres acciones involucradas, la acción *A* es pre-condición de la acción *B, B*

es pre-condición de la acción C, y además, se desea que *C* sea pre-condición de *A*. Como extensión de la comprobación anterior para los ciclos cortos, sólo se admitirá un ciclo largo cuando algunas de las acciones que intervienen ya se han realizado (Invariante de Terminación).

∃C ∃R ∃B ∃Q ((isPrec(Z,C,R) and component(A,Z)) and
(((isPrec(Y,B,Q) and component(C,Y)) and component(B,X)) and
(isInFH(A,P) or (isInFH(B,Q))) or isInFH(C,R)))) or
not ((isPrec(Z,C,R) and component(A,Z)) and ((isPrec(Y,B,Q) and
component(C,Y)) and component(B,X)))

Por ser ésta una extensión del caso 4, se considera de igual forma el modo de evaluación *backward* de la pre-condición con ausencia de ocurrencias en el pasado.

f) *Creación de acciones:*

Como dijimos cuando vimos las acciones *addSAction* y *addCAction*, cuando se añade una acción a un agente debemos obligar a que la pre-condición *p* tome valor. Estas acciones generan un estímulo para activar esta acción. Además, el desarrollador puede querer modificar la pre-condición *p* y, por tanto, genera el estímulo necesario. Por ello, esta parte de la pre-condición se uniría al resto con el operador *and*:

StimulusAddPPrec(X, A, P)

La pre-condición completa es la siguiente:

addPPrec(X,A,P) ← (isInSSHA(A,P) and isInSSHP(X,P)) and
((component(A,X) and isInFH(A,P)) or not component(A,X)) and
((component(not A,X) and not isInFH(A,P)) or
not component(not A, X)) and *(∃B ∃Q ∃Y (((isPrec(Y,B,Q) and*
component(A,Y)) and component(B,X)) and (isInFH(A,P) or
isInFH(B,Q))) or not ((isPrec(Y,B,Q) and component(A,Y)) and
component(B,X))) and (∃C ∃R ∃B ∃Q ((isPrec(Z,C,R) and
component(A,Z)) and
(((isPrec(Y,B,Q) and component(C,Y)) and component(B,X)) and
(isInFH(A,P) or (isInFH(B,Q))) or isInFH(C,R)))) or not
((isPrec(Z,C,R) and component(A,Z)) and ((isPrec(Y,B,Q) and
component(C,Y)) and component(B,X)))) and
StimulusAddTPrec(X,A,B,P)

Post-condición *addPPrec(X,A,P)*:

Las acciones con igual nombre que sean realizadas por agentes hermanos deben ser idénticas en cuanto a su funcionamiento, número

y tipo de atributos, definición (si es compleja) y pre-condición p (invariante de Unicidad de Acciones).

∀ Q isBrother(P,Q) and isInSSHA(A,Q) and addPPrec(X,A,Q) ← addPPrec(X,A,P)

Acciones a realizar en *addPPrec*:

- Si existe otro u otros agentes hermanos que contienen la acción *A* en su interfaz, automáticamente se genera para cada uno de ellos un estímulo *StimulusAddPPrec(X,A,?)*; "?" denota cada uno de los agentes hermanos que cumplen esta condición. Estos estímulos se almacenarán en la MFH del Metasistema con el objetivo de que todos tengan la misma pre-condición *p* en su acción *A*.

7.4.2.7 Añadir o sustituir una pre-condición *t* a una acción transaccional *(addTPrec).*

Al igual que una pre-condición *p*, una pre-condición *t* es una fórmula de la lógica temporal. Añadir una nueva *t* a una acción transaccional supone la sustitución de la anterior (invariante de Confluencia). Hay que estudiar más a fondo si se podría controlar que no entrase en contradicción ni con la *s* ni con la *p* asociada a dicha acción transaccional. Con la *p* realmente no existe ningún problema porque la *t* sólo se comprobará cuando se haya iniciado la transacción, y, entonces, ya se habrá evaluado la *p* y ésta fue *true*.

Pre-condición de *addTPrec(X,A,B,P):*

Las siguientes comprobaciones son necesarias antes de añadir una pre-condición *t* a una acción transaccional *A* que se encuentra en la transacción *B* del agente *P*. Todas ellas se unirán con un *and*.

a) *Existencia previa*: la acción transaccional, *A*, debe existir y su pre-condición *t* debe estar formada por otras acciones existentes en el agente al cual pertenece la acción transaccional o en sus agentes hermanos (invariante de Posibilidad de Realización de Acciones).

isInSSHA(A,P) and isInSSHP(X,P)

b) *Recursividad*: Para asegurar el invariante de Terminación hay que comprobar que el sistema software no se bloquee. No se permite que una acción transaccional aparezca en su propia pre-condición porque podría no ejecutarse nunca. Recordemos que este tipo de pre-condición se evalúa en base a la historia del TM creado para gestionar a la transacción. Puesto que se para el funcionamiento del sistema software

y se asegura que ninguna acción se está realizando cuando se va a llevar a cabo una acción estructural, ni el TM ni su historia existirán.

not component(A,X)

c) *Ciclos recursivos cortos:*

No se admitirán ciclos cortos (invariante de Terminación). Se produce un ciclo corto si una acción transaccional *C* aparece en la pre-condición *t* de la acción transaccional *A* y, a su vez, *A* aparece en la pre-condición *t* de *C*.

∃ C ∃ Q ∃ Y not ((isTPrec(Y,C,B,Q) and component(A,Y)) and component(C,X))

d) Ciclos recursivos largos:

Llamaremos *ciclo largo* a una situación de pre-condición en la que intervienen tres o más acciones transaccionales de la misma transacción. Este es el caso en el que la acción transaccional *A* es parte de la pre-condición de la acción transaccional *C, C* de la pre-condición de la acción transaccional *D*, y además, se desea que *D* sea parte de la pre-condición de *A* (invariante de Terminación).

∃C ∃R ∃Z ∃D ∃Q not ((isTPrec(Z,C,B,R) and component(A,Z)) and (isTPrec(Y,D,B,Q) and component(C,Y)) and component(D,X))

e) Creación de acciones:

Debe existir un estímulo para que se active esta acción estructural. Este estímulo procederá del desarrollador cuando desee cambiar la pre-condición *t* de una acción transaccional.

StimulusAddTPrec(X,A,B,P)

La pre-condición completa es la siguiente:

addTPrec(X,A,B,P) ← (isInSSHA(A,P) and isInSSHP(X,P)) and not component(A,X) and (∃C ∃Q ∃Y not ((isTPrec(Y,C,B,Q) and component(A,Y)) and component(C,X))) and (∃C ∃R ∃Z ∃D ∃Q not ((isTPrec(Z,C,B,R) and component(A,Z)) and (isTPrec(Y,D,B,Q) and component(C,Y)) and component(D,X)) and StimulusAddTPrec(X,A,B,P)

Post-condición *addTPrec(X,A,B,P):*

- Si existen agentes hermanos que realicen la misma acción compleja, todas deben tener la misma pre-condición *t* para su acción transaccional *A* (invariante de Unicidad de Acciones).

∀ Q isBrother(P,Q) and isInSSHA(B,Q) and addTPrec(X,A,B,Q) ← addTPrec(X,A,B,P)

Acciones a realizar en *addTPrec*:

- Si existe otro u otros agentes hermanos que contienen la acción B en su interfaz, automáticamente se genera para cada uno de ellos un estímulo *StimulusAddTPrec(X,A,B,?);* "?" denota cada uno de los agentes hermanos que cumplen esta condición.

7.4.2.8 Añadir o sustituir la lista de acciones incompatibles de una acción simple o compleja *(addLAI).*

Podemos asociar a una acción simple o compleja una lista con los nombres de las acciones que no se pueden realizar simultáneamente con ella. Esta lista se denomina Lista de Acciones Incompatibles (LAI). Si la acción ya tenía asociada una LAI, se sustituye por la nueva.

Pre-condición de *addLAI(list,A,P):*

- La acción simple o compleja, *A*, debe existir (invariante de Referencia).

  isInSSHA(A,P)

- Existencia del estímulo asociado a esta acción estructural.

  StimulusAddLAI(list,A,P)

La pre-condición completa es:

addLAI(list,A,P) ← isInSSHA(A,P) and StimulusAddLAI(list,A,P)

Post-condición de *addLAI(list,A,P):*

- Si existen agentes hermanos que realicen la misma acción, todas deben tener la misma LAI (invariante de Unicidad de Acciones).

  ∀ Q isBrother(P,Q) and isInSSHA(A,Q) and addLAI(list,A,Q) ← addLAI(list,A,P)

Acciones a realizar en *addLAI*:

- Si existe otro u otros agentes hermanos que contienen la acción A en su interfaz, genera para cada uno de ellos un estímulo *StimulusAddLAI*.

7.4.2.9 Borrar una acción simple o compleja *(delAction).*

Esta acción permite borrar una acción simple o compleja de la interfaz de un agente.

Pre-condición de *delAction(A,P)*:

- La acción simple o compleja que se quiere borrar debe existir (invariante de Referencia).

  isInSSHA(A,P)

- Si la acción a borrar es realizada por otro agente hermano, no se realizará ninguna comprobación más. La acción puede borrarse ya que, al menos, otro agente la realiza. Con esto se preserva el invariante de Posibilidad de Realización de Acciones.

  $\exists$ *Q isBrother(P,Q) and isInSSHA(A,Q)*

- Si la acción a borrar sólo es realizada por el agente *P*, es decir, no existe ningún agente hermano que contenga en su interfaz otra acción con el mismo nombre:
  - Se comprueba si aparece en alguna pre-condición de otras acciones realizadas por él mismo o por agentes hermanos. Si lo hiciera, no se debería permitir que se borrara porque:
    - Si apareciera como único componente en alguna, no podría borrarse, ya que violaría el invariante de Posibilidad de Realización de Acciones. De esta forma obligamos al desarrollador a que modifique antes las pre-condiciones compuestas sólo por esta acción con el fin de tener un sistema consistente.
    - Si no, su borrado puede depender de su lugar dentro de la expresión de la pre-condición. Por ejemplo, si está involucrada en un *since*, no puede ser eliminada porque la expresión quedaría incompleta.

  $\forall$ *Q isBrother(P,Q) and* $\forall$ *B isInSSHA(B,Q) and not isInPrec(A,B,Q)*

  - Se comprueba si la acción a eliminar aparece en la definición de alguna acción compleja realizada por algún agente padre o agente hermano (invariante de Posibilidad de Realización de Acciones e invariante de Referencia). Si es así, no se puede borrar ya que, si modificamos una acción compleja, puede que ésta ya no se realice correctamente. De esta forma obligamos a que el desarrollador primero modifique la o las acciones complejas afectadas si quiere eliminar la acción.

∀ R (isFather(R,P) or isBrother(P,R)) and ∀B isInSSHA(B,R) and not isIn(A,B))

- Debe existir un estímulo *StimulusDelAction(A,P)* que indique al Metasistema que alguien, un agente o el desarrollador, quiere realizar dicha acción de evolución.

La pre-condición completa es:

delAction(A,P) ← isInSSHA(A,P) and ((∃ Q isBrother(P,Q) and isInSSHA(A,Q)) or (∀ Q isBrother(P,Q) and ∀ B isInSSHA(B,Q) and not isInPrec(A,B,Q)) and ((∀ R (isFather(R,P) or isBrother(P,R)) and ∀ B isInSSHA(B,R) and not isIn(A,B))) and StimulusDelAction(A,P)

Acciones a realizar en *delAction:*

- Si la acción se realiza porque la pre-condición se cumple y el agente contiene agentes anidados que realicen dicha acción (igual nombre), también hay que borrar la acción de sus subagentes (invariante de Posibilidad de Realización de Acciones). Este borrado se hace en cadena atravesando toda la jerarquía de agentes descendientes del agente donde se ha iniciado esta acción estructural ya que no tiene sentido que realicen una acción que nunca se iniciará.

7.4.2.10 Modificar la definición de una acción compleja *(defCAction).*

Cuando se crea una acción compleja, se introduce su definición con una composición por defecto, la acción *terminar*. Sin embargo, ésta debe modificarse de acuerdo al conjunto de acciones transaccionales que la deben constituir para conseguir el funcionamiento deseado. Para ello, utilizamos un lenguaje apropiado (*TDL*) que nos permite definir el conjunto de acciones transaccionales que deben llevarse a cabo dentro de la acción compleja junto con el orden temporal de su realización. Cada vez que modificamos la definición de una acción compleja, cambiamos su funcionamiento.

Pre-condición de *defCAction(def,A_c,P):*

- Debe existir la acción compleja cuya definición queremos modificar (invariante de Referencia).

  isInSSHA(A_c,P)

- Hay que comprobar que las acciones que se encuentran en su definición existen y son realizadas por agentes agregados al agente al

que añadimos la acción compleja (invariante de Posibilidad de Realización de Acciones).

∀ B isIn(B,def) ∃ Q isAgrAgent(Q,P) and isInSSHA (B,Q)

- Debido a la actuación del desarrollador, debe recibirse un estímulo para que se realice esta acción.

StimulusDefCAction(def,A_c,P)

La pre-condición completa es:

defCAction(def,A_c,P) ← isInSSHA(A_c,P) and
(∀ B isIn(B, def) ∃ Q isAgrAgent(Q,P) and isInSSHA (B, Q))
and StimulusDefCAction(def,A_c,P)

Post-condición de *defCAction(def,A_c,P)*:

- Las acciones complejas con igual nombre que sean realizadas por agentes hermanos deben ser idénticas en cuanto a su definición (invariante de Unicidad de Acciones).

∀ Q isBrother(P,Q) and isInSSHA(A_c,Q) and defCAction(def,A_c,Q) ← defCAction(def,A_c,P)

Acciones a realizar de *defCAction*:

- A cada acción transaccional de la nueva definición se asocia una pre-condición *t* que, por defecto, es *true*.
- Una vez descrita la definición de la acción compleja dada por el desarrollador, se concatena a la expresión introducida con el operador ";" seguido de la acción *terminar*.
- Se genera la pre-condición *s* asociada a cada acción transaccional según su orden en la definición.
- En el caso de que exista otro u otros agentes hermanos que contienen esta acción en su interfaz, automáticamente se introduce en la MFH del Metasistema, por cada uno de ellos, un estímulo *StimulusDefCAction* con la misma definición. En el caso de que no se pueda llevar a cabo en alguno de los agentes no se permitirá realizar esta acción en ninguno de ellos, avisando de ello al desarrollador.

7.4.2.11 Cambiar de nombre una acción simple o compleja *(renameAction).*

Puede ser interesante tener la posibilidad de cambiarle el nombre a las acciones, tanto simples como complejas.

Pre-condición de *renameAction(A,B,P)*:

- La acción debe existir y el nuevo nombre de la acción no debe coincidir con ninguno existente (invariante de Nombres Únicos).

  isInSSHA(A,P) and not isInSSHA(B,P)

- Existencia del estímulo asociado a esta acción estructural.

  StimulusRenameAction(A,B,P)

La pre-condición completa es:

renameAction(A,B,P) ← *isInSSHA(A,P) and not isInSSHA(B,P) and StimulusRenameAction(A,B,P)*

Acciones a realizar en *renameAction*:

- Si no existe ningún agente hermano que realice una acción con el antiguo nombre:
  - Se modifican todas aquellas pre-condiciones en las que aparezca la acción cuyo nombre se va a cambiar, se sustituye el nombre antiguo por el nuevo (invariante de Referencia. Invariante de Posibilidad de Realización de Acciones).
  - Se sustituye en la definición de cualquier acción compleja donde aparezca el antiguo nombre de la acción por el nuevo (invariante de Posibilidad de Realización de Acciones).

7.4.2.12 Añadir un atributo a una acción *(addAtt).*

Se le añade un atributo nuevo a una acción de un agente.

Pre-condición de *addAtt(att,A,P):*

- No puede existir otro atributo con igual nombre (invariante de Nombres Unicos).

  not hasAtt(att,A,P)

- Debe existir el estímulo asociado a esta acción: *StimulusAddAtt(att,A,P)*

La pre-condición completa es:

addAtt(att,A,P) ← *not hasAtt(att,A,P) and StimulusAddAtt(att,A,P)*

Post-condición de *addAtt(att,A,P):*

- Si existe algún agente hermano que realice la misma acción, también se debe añadir el mismo atributo (invariante de Unicidad de Acciones).

  ∀ *Q isBrother(P,Q) and isInSSHA(A,Q) and addAtt(att,A,Q)* ← *addAtt(att,A,P)*

Acciones a realizar en *addAtt*:

- Genera un *StimulusAddAtt* para cada uno de los agentes hermanos que realicen la misma acción.

7.4.2.13 Borrar un atributo de una acción *(delAtt).*

Permite borrar un atributo de una acción de un agente siempre y cuando no sea el único atributo de dicha acción (invariante de Caracterización por Atributos).

Pre-condición de *delAtt(att,A,P)*:

- El atributo debe existir en la acción especificada (invariante de Referencia).

  hasAtt(att,A,P)

- No debe ser el único atributo de la acción (invariante de Caracterización por Atributos).

  not lastAtt(att,A,P)

- No debe ser referenciado en ninguna pre-condición *t* y/o *p* de ninguna otra acción del agente al cual pertenece o de sus agentes hermanos (invariante de Posibilidad de Realización de Acciones).

  not ∃ *B* ∃ *Q isBrother (Q,P) and isPrec(X,B,Q) and isInPrec(A,X,Q)*

- Debe existir un estímulo para dicha acción.

  StimulusDelAtt(att,A,P)

La pre-condición completa es:

delAtt(att,A,P) ← hasAtt(att,A,P) and not lastAtt(att,A,P) and
(not ∃ B ∃ Q isBrother (Q,P) and isPrec(X,B,Q) and isInPrec(A,X,Q))
and StimulusDelAtt(att,A,P)

Post-condición de *delAtt(att,A,P):*

- Al igual que en el caso anterior, si existen varias acciones con igual nombre realizadas por agentes hermanos, éstas deben ser idénticas (invariante de Unicidad de Acciones), por tanto deben tener los mismos atributos.

  ∀ Q isBrother(P,Q) and isInSSHA(A,Q) and delAtt(att,A,Q) ← delAtt(att,A,P)

Acciones a realizar en *delAtt*:

- Se elimina el atributo de la acción aún cuando existan ocurrencias de dicha acción que mantendrán ese componente. Esto se debe a que no nos interesa perder información histórica sobre el agente.
- Se debe eliminar ese atributo de cualquier acción con igual nombre realizada por agentes hermanos con el fin de mantener su igualdad, por ello genera un estímulo *StimulusDelAtt* para cada una de dichas acciones.

7.4.2.14 Cambiar de nombre un atributo *(renameAtt).*

Puede ser interesante, en un momento dado, cambiar el nombre de un atributo asociado a una acción de un agente.

Pre-condición de *renameAtt(att,att2,AP):*

- El atributo, *att*, cuyo nombre queremos cambiar debe existir en la acción especificada (invariante de Referencia).

  hasAtt(att,A,P)

- No debe existir otro atributo en la acción con el nuevo nombre, *att2* (invariante de Nombres Unicos).

  not hasAtt(att2,A,P)

- Debe existir el estímulo asociado a esta acción.

  StimulusRenameAtt(att,att2,A,P)

La pre-condición completa es:

renameAtt(att,att2,A,P) ← *hasAtt(att,A,P) and not hasAtt(att2,A,P) and StimulusRenameAtt(att,att2,A,P)*

Post-condición de *renameAtt(att,att2,A,P):*

- Al igual que en el caso anterior, si existen varias acciones con igual nombre realizadas por agentes hermanos, éstas deben ser idénticas (invariante de Unicidad de Acciones).

  ∀ Q isBrother(P,Q) and isInSSHA(A,Q) and rename(att,att2,A,Q) ← *renameAtt(att,att2,A,P)*

Acciones a realizar en *renameAtt*:

- Debe renombrarse el atributo en cualquier pre-condición *t* y/o *p* de cualquier otra acción del agente citado o de sus agentes hermanos donde sea referenciado (invariante de Posibilidad de Realización de Acciones).
- Se genera un *StimulusRenameAtt* para cada acción con igual nombre realizada por agentes hermanos.

7.4.2.15 Clonar un agente *(cloneAgent).*

La forma normal de crear un agente será mediante la clonación. Esta acción debe realizarse sobre un agente previamente definido (en cuanto a su estructura) y que llamaremos agente elemental, AE, o bien, sobre un agente ya existente en el sistema software.

La operación de clonar crea un agente con las mismas características y funcionamiento que el agente a partir del cuál se realiza la copia y, por tanto, tendrá el mismo agente padre. Después el desarrollador podrá moverlo, desagregarlo, agregarlo a otro, etc.

Si el agente a clonar tiene agentes agregados, se realiza una copia de todo el sub-árbol que define su estructura, y por tanto, su funcionamiento. Así, después de esta operación tenemos dos agentes con distinto nombre pero que realizan las mismas acciones simples y/o complejas. Después de su creación, el clon puede evolucionar y realizar más o menos acciones.

Pre-condición de *cloneAgent(P,Q)*:

- El agente sobre el cual se va a hacer la operación de clonación debe existir (invariante de Referencia).

  exist(P)

- El nombre del clon no debe coincidir con ninguno de los agentes existentes en el mismo nivel de jerarquía (invariante de Nombres Únicos).

  ∀ R isFather(R,P) and not isAgrAgent(Q,R)

- Debe existir un estímulo para esta acción.

  StimulusCloneAgent(P,Q)

La pre-condición completa es:

cloneAgent(P,Q) ← exist(P) and (∀ R isFather(R,P) and not isAgrAgent(Q, R)) and StimulusCloneAgent(P,Q)

Acciones a realizar en *cloneAgent(P,Q)*:

- Si el agente que se va a clonar no es un agente simple, se copia toda su estructura (sub-agentes).

7.4.2.16 Crear agente *(createAgent).*

Crear un agente como hijo de otro agente existente en la jerarquía de agentes.

Pre-condición de *createAgent(P,Q)*:

- El agente padre, Q, debe existir (invariante de Mantenimiento del Grafo de Agregación de Agentes).

  exist(Q)

- No debe existir otro agente con este mismo nombre en Q (invariante de Nombres Únicos).

  not isAgrAgent(P,Q)

- Se debe haber producido un estímulo para esta acción.

StimulusCreateAgent(P,Q)

La pre-condición completa es:

createAgent(P,Q) ← exist(Q) and not isAgrAgent(P,Q) and StimulusCreateAgent(P,Q)

Acciones a realizar en *createAgent*:

- Se crea el agente realizando una clonación del agente elemental, y después se agrega al agente padre, *Q*.
- Como no tiene sentido crear un agente que no realice ninguna acción, se introduce en la MFH el estímulo necesario para llamar a la acción *addSAction* con el fin de que el nuevo agente lleve a cabo, al menos, una acción simple.

7.4.2.17 Crear un Sistema *(createSystem).*

Esta acción permite crear un agente sin padre al cual se le asigna un nombre. Inicialmente sólo se permitirá que exista un único agente que sea Sistema. Esto obliga al desarrollador a establecer la jerarquía sin que se le olvide ningún agente inconexo dentro del sistema software.

Pre-condición de *createSystem(P):*

- El Metasistema no debe haber creado ningún sistema con ese nombre (invariante de Nombres Unicos).

  not exist(P)

- Debe existir un estímulo para esta acción.

  StimulusCreateSystem(P)

La pre-condición completa es:

createSystem (P) ← not exist(P) and StimulusCreateSystem(P)

Acciones a realizar en *createSystem*:

- Se crea un clon del agente elemental cuyo nombre será *P*.
- Al igual que la acción anterior se ha de definir al menos una acción, por tanto se genera el estímulo necesario, *StimulusAddSAction*, para que se realice la acción *addSAction*.

7.4.2.18 Agregar un agente a otro *(agrAgent).*

Se agrega un agente ya creado a otro. Es posible que un agente esté agregado a más de un agente, es decir, tenga más de un padre.

Pre-condición de *agrAgent(P,Q):*

- No debe existir un agente hijo del agente al cual se pretende agregar con igual nombre (invariante de Nombres Unicos).

  not isAgrAgent(P,Q)

- Deben existir el agente a agregar, *P*, y el padre, *Q* (invariante de Referencia).

  exist(P) and exist(Q)

- No puede agregarse a sí mismo (invariante de Mantenimiento del Grafo de Agregación de Agentes).

  $P \neq Q$

- Debe existir un estímulo para esta acción.

  StimulusAgrAgent(P,Q)

La precondición completa es:

agrAgent(P,Q) ← *not isAgrAgent(P,Q) and exist(P) and exist(Q) and not isAgrAgent(Q,P) and* $(P \neq Q)$ *and StimulusAgrAgent(P,Q)*

Post-condición de *agrAgent(P,Q):*

- Sólo se agregará el nuevo agente, si en el caso de que realice una acción con igual nombre que alguno de sus nuevos agentes hermanos, éstas son idénticas (invariante de Unicidad de Acciones).

  $\forall$ *R isBrother(P,R) and* $\forall$ *A isInSSHA(A,P) and isInSSHA(A,R) and P(A) = R(A)* ← *agrAgent(P,Q)*

7.4.2.19 Desagregar un agente *(disAgrAgent).*

Esta acción implica que un agente deja de estar agregado a otro. Como obligamos a que tenga al menos un padre, sólo se permitirá que se realice si el agente tiene otro padre.

Además, si el agente a desagregar es el único que realiza una o más acciones para el agente padre, no se podrá realizar esta acción. Si se quiere desagregar, *previamente* se deben borrar sus acciones de cualquier acción compleja y de cualquier pre-condición de las acciones realizadas por sus agentes hermanos y de la interfaz de este agente padre. También se deben cambiar las definiciones de las acciones complejas en las que intervenga.

Pre-condición de *disAgrAgent(P,Q):*

- El agente a desagregar debe estar agregado previamente (invariante de Mantenimiento del Grafo de Agregación de Agentes).

  isAgrAgent(P,Q)

- El agente debe tener más de un padre.

  $\exists$ *S isFather(S,P) and (S <> Q)*

- Si este agente realiza alguna acción para el agente padre que no realice ninguno de sus agentes hermanos, no podríamos desagregarlo (invariante de Posibilidad de Realización de Acciones).

  $\forall$ *A isInSSHA(A,P)* $\exists$ *R isAgrAgent(R,Q) and A isInSSHA(A,R)*

- Debe existir un estímulo para esta acción.

  StimulusDisAgrAgent(P,Q)

La pre-condición completa es:

disAgrAgent(P,Q) ← *isAgrAgent(P,Q) and (*$\exists$ *S isFather(S,P) and (S<>Q) and (*$\forall$ *A isInSSHA(A,P)* $\exists$ *R isAgrAgent(R,Q) and A isInSSHA(A, R)) and StimulusDisAgrAgent(P,Q)*

7.4.2.20 Borrar un agente *(delAgent).*

Borra un agente del sistema software. Teniendo en cuenta las acciones que realiza, si el agente a borrar tiene un único padre, desaparecerá del sistema software así como todos los agentes que tenga agregados (todo el subárbol de agentes que representa).

Pre-condición de *delAgent(P)*:

- El agente a borrar debe existir (invariante de Referencia).

  exist(P)

- Las acciones que realiza para su agente padre deben poder realizarse (invariante de Posibilidad de Realización de Acciones). Por tanto, si no existe algún o algunos agentes que realicen su misma acción no se podrá borrar dicho agente hasta que estas acciones no sean necesarias (desaparezcan del conjunto de acciones del padre, de las pre-condiciones de cualquier acción y de la definición de cualquier acción compleja realizada por sus agentes hermanos).

  ∃ Q isFather(Q,P) and ∀ A isInSSHA(A,P) and (∃ R isAgrAgent(R,Q) and isInSSHA(A,R))

- Debe existir un estímulo para esta acción.

  StimulusDelAgent(P)

La pre-condición completa es:

delAgent(P) ← exist(P) and (∃ Q isFather(Q,P) and ∀ A isInSSHA(A,P) (∃ R isAgrAgent(R,Q) and isInSSHA(A, R))) and StimulusDelAgent(P)

Acciones a realizar en *delAgent*:

- Se borran todos sus agentes hijos, es decir, debe desaparecer toda esa sub-rama de la estructura de agentes existente.

7.4.2.21 Cambiar de nombre a un agente *(renameAgent).*

Se le cambia el nombre a un agente por otro nuevo.

Pre-condición de *renameAgent(P,Q)*:

- El agente a renombrar debe existir.

  exist(P)

- Se permite siempre que no haya otro agente dentro del mismo nivel de la jerarquía con igual nombre. Si este agente tiene varios padres, comprobaremos que no exista otro agente con el nuevo nombre agregado a ninguno de sus padres.

  $\forall$ *R isFather(R,P) and not isAgrAgent(Q,R)*

- Debe existir un estímulo asociado a dicha acción.

  StimulusRenameAgent(P,Q)

La pre-condición completa es:

renameAgent(P,Q) ← *exist(P) and ($\forall$ R isFather(R,P) and not isAgrAgent(Q, R)) and StimulusRenameAgent(P,Q)*

7.4.2.22 Mover una acción a otro agente *(moveAction).*

Esta acción del Metasistema implica borrar una acción de un agente al que estaba asociada, y que era el responsable de realizarla, para asociarla a otro. Se realiza la copia de la acción simple o compleja a mover en el agente destino.

En realidad, esta acción *moveAction(A,P,Q)* equivale a la realización secuencial de las dos acciones siguientes:

(addSAction(A,Q) or addCAction(A,Q)) ← *moveAction(A,P,Q)* depende si la acción a añadir es simple o compleja

delAction(A,P) ← *moveAction(A,P,Q)*

Pre-condición de *moveAction(A,P,Q)*:

- La acción, *A*, que se desea mover debe existir en *P* (invariante de Referencia).

  isInSSHA(A,P)

- El agente a donde se va a mover la acción debe existir y no debe realizar una acción con igual nombre.

  exist(Q) and not isInSSHA(A,Q)

- Puesto que la acción *A* se va a borrar de *P*, comprobamos si alguno de sus agentes hermanos la realiza. Si no, sólo se podrá borrar si no aparece en ninguna pre-condición de acciones realizadas por agentes hermanos o en la definición de alguna acción compleja de sus hermanos (invariante de Posibilidad de Realización de Acciones).

  (∃ Q isBrother(P,Q) and isInSSHA(A,Q)) or
  (∀ Q isBrother(P,Q) and ∀ B isInSSHA(B,Q) and not isInPrec(A,B,Q))
  and ((∀ R (isFather(R,P) or isBrother(P,R)) and
  ∀ B isInSSHA(B,R) and not isIn(A,B))

- Debe existir un estímulo para esta acción: *StimulusMoveAction(A,P,Q)*

La pre-condición completa es:

moveAction(A,P,Q) ← isInSSHA(A,P) and (exist(Q) and not isInSSHA(A,Q))
and (∃ Q isBrother(P,Q) and isInSSHA(A,Q)) or
((∀ Q isBrother(P,Q) and ∀ B isInSSHA(B,Q) and
not isInPrec(A,B,Q)) and ((∀ R (isFather(R,P) or
isBrother(P,R)) and ∀ B isInSSHA(B,R) and not isIn(A,B)))

7.4.2.23 Mover un agente *(moveAgent).*

Implicaría desagregar el agente de su padre actual y agregarlo al nuevo padre pero sin la pre-condición asociada a la acción de desagregar que evita realizarla si sólo tiene un padre. Ahora hay que controlar que si ese agente está agregado a más de un agente, se debe especificar desde qué agente padre queremos moverlo para no provocar efectos indeseados.

Pre-condición de *moveAgent(P,Q,R)*:

- El agente a mover, *P*, y el agente que será su nuevo padre, *R*, deben existir (invariante de Referencia).

  isAgrAgent(P,Q) and exist(R)

- No debe existir en el futuro nuevo agente padre un agente con igual nombre que el que se va a agregar (invariante de Nombres Unicos).

  not isAgrAgent(P,R)

- Si el agente a mover realiza alguna acción para el agente padre que no realice ninguno de sus agentes hermanos, no podríamos moverlo (invariante de Posibilidad de Realización de Acciones).

  $\forall$ *A isInSSHA(A,P)* $\exists$ *T isAgrAgent(T,Q) and A isInSSHA(A,T)*

- Debe existir un estímulo para esta acción.

  StimulusMoveAgent(P,Q,R)

La precondición completa es:

moveAgent(P,Q,R) $\leftarrow$ *isAgrAgent(P,Q) and exist(R) and not isAgrAgent(P,R) and not isAgrAgent(R,P) and* ($\forall$ *A isInSSHA(A,P)* $\exists$ *T isAgrAgent(T,Q) and A isInSSHA(A,T)) and StimulusMoveAgent(P,Q,R)*

En la tabla 7.2 se recoge un resumen de todas las acciones estructurales vistas.

ACCIONES ESTRUCTURALES DEL LENGUAJE *M2*	
addSAction(A,P)	Añadir una acción simple *A* a un agente *P*
addCAction(A,P)	Añadir una acción compleja *A* a un agente *P*
addPPrec(X,A,P)	Asociar o sustituir una pre-condición del tipo *p*, *X,* a una acción *A* de un agente *P*
addTPrec(X,A,B,P)	Asociar o sustituir una pre-condición del tipo *t*, *X,* a una acción transaccional *A de la transacción B* del agente *P*
addLAI(list,A,P)	Asociar o sustituir una Lista de Acciones Incompatibles, *list*, a una acción simple o compleja *A* de un agente *P*
delAction(A,P)	Borrar una acción simple o compleja *A* de un agente *P*
defCAction(def,A,P)	Modificar la definición *def* de una acción compleja *A* de un agente *P*
renameAction(A,B,P)	Renombrar una acción simple o compleja *A* de un agente *P* y pasar a llamarla *B*
addAtt(att,A,P)	Añadir un atributo *att* a una acción *A* de un agente *P*
delAtt(att,A,P)	Borrar un atributo *att* de una acción *A* de un agente *P*
renameAtt(att,att2,A,P)	Renombrar un atributo *att y* de una acción simple o compleja *A* de un agente *P* y pasar a llamarlo *att2*
cloneAgent(P,Q)	Clonar el agente *P* y al clon llamarlo *Q*
createAgent(P,Q)	Crear un agente *P* que será hijo del agente *Q*
createSystem(P)	Crear un agente *P* que será el único agente sin padre, el sistema
agrAgent(P,Q)	Agregar un agente *P* a otro agente *Q*
disAgrAgent(P,Q)	Desagregar un agente *P* de otro agente llamado *Q*
delAgent(P)	Borrar un agente *P*
renameAgent(P,Q)	Renombrar un agente *P* y pasar a llamarlo *Q*
moveAction(A,P,Q)	Mover una acción simple o compleja *A* de un agente *P* a otro *Q*
moveAgent(P,Q,R)	Mover el agente *P* de *Q* a *R*

Tabla 7.2. Lista de la acciones estructurales de M2

7.5 Mantenimiento de la estructura dinámica.

Las acciones de evolución pueden implicar un cambio en la información mantenida por el *Controller* y sus subagentes, OcuR y *Evaluator*. Después de un periodo de cambio estructural, el Sistema Genético envía la nueva información sobre la parte de la estructura del sistema que ha sido modificada al *Controller* del agente, ya que en él se mantiene la información actual de la estructura del agente. El *Controller* pasará dicha información a los agentes OcuR y *Evaluator*.

En los siguientes apartados vamos a comentar qué información afecta especialmente al OcuR y al *Evaluator*.

7.5.1 Actualizaciones en el OcuR.

Como vimos en el capítulo 6, el OcuR mantiene las siguientes tablas:

◊ T*abla de Agentes:* contiene una entrada por cada uno de los agentes hijos del agente donde se encuentra el *OcuR*. Cada entrada mantiene un identificativo del agente hijo y una referencia a la tabla de acciones que realiza cada hijo.

tabla_agentes = { nombre_agente + ref.Tabla_Acciones }

◊ *Tabla de Acciones*: existe una entrada por cada acción que es capaz de realizar un agente hijo. En cada entrada aparece el nombre de la acción y una referencia a la tabla de pre-condiciones correspondiente.

tabla_acciones = { nombre_acción + ref.Tabla_Pre-condiciones }

◊ *Tabla de Pre-condiciones*: existe una entrada por cada nombre de acción y/o de estímulo que aparece en la pre-condición asociada a la acción que describe.

tabla_pre-condiciones = contador_referencias + { nombre_accion | nombre_estímulo }

◊ *Tabla de Transacciones*: cada entrada describe una transacción o acción compleja llevada a cabo por un agente hijo. Los elementos de una entrada son: el nombre de la transacción, el nombre del estímulo que la inicia, la definición de la transacción (en el lenguaje

TDL) y una lista con los nombres de sus acciones transaccionales junto con sus pre-condiciones ***t.***

*tabla_transacciones = { nombre_transacción + nombre_estímulo + definición + { nombre_acción + **t** } }*

A continuación, presentamos las acciones de evolución que provocan una actualización de la información del OcuR y comentamos las acciones realizadas por éste:

- *addSAction(A_s, P):* Añadir una acción simple (*A_s*) a un agente hijo (*P*). Al OcuR se le envía el nombre de la acción simple del agente hijo y la pre-condición *p* de ésta. Las acciones que realiza el OcuR son:

    - Crear una entrada más en la tabla de acciones asociada a dicho agente hijo para *A_s*.

    - Construir la tabla de pre-condiciones y asociarla a la entrada de la tabla de acciones correspondiente a dicha acción. Si existe otro agente hermano que realice la misma acción, su tabla de pre-condiciones se enlaza a la acción nuevamente añadida, se incrementa el contador de referencias y no hace falta crear otra.

- *addCAction(A_c, P):* Añadir una acción compleja (*A_c*) a un agente hijo (*P*). Al OcuR se le envía el nombre de la acción compleja, del agente, la pre-condición *p*, la definición de la acción compleja y el estímulo de inicio que se generará al comienzo de la transacción. Las acciones que lleva a cabo el OcuR son:

    - Crear una nueva entrada en la tabla de acciones asociada al agente hijo.

    - Construir la tabla de pre-condiciones de la misma forma que en el caso anterior y con las mismas consideraciones.

    - Crear una nueva entrada en la tabla de transacciones donde se almacena el nombre de la acción compleja, el estímulo de inicio, su descripción (en TDL) y una lista de sus acciones transaccionales junto con sus pre-condiciones *t*.

- *addPPrec(X, A, P):* Añadir una pre-condición *p* (*X*) a una acción (simple o compleja, *A*) de un agente hijo (*P*). Al OcuR se le manda la siguiente información: nombre de la acción (debe existir), nombre del agente, nueva pre-condición *p*. Las acciones que realiza el OcuR son:

    - Borrar la tabla de pre-condiciones asociada a la acción.

- Crear una nueva tabla de pre-condiciones y enlazarla con la acción.

- *addTPrec(X,A,B,P):* Añadir una pre-condición *t* (*X*) a una acción transaccional (*A*). Al OcuR se le envía el nombre de la transacción (*B*), del agente (*P*), de la acción transaccional y la *t* de ésta. Las acciones que realiza el OcuR son:

    - Añadir la información de esta acción transaccional en la tabla de transacciones.

- *delAction(A, P):* Borrar una acción simple o compleja (*A*) de un agente hijo (*P*). Al OcuR le llega la información siguiente: nombre del agente y nombre de la acción borrada. Las acciones que se realizan son:

    - Borrar la entrada correspondiente a dicha acción de la tabla de acciones asociada a ése agente hijo.

    - Si es una acción compleja, borrar la entrada correspondiente de la tabla de transacciones.

- *delTPrec(A, B, P):* Borrar la pre-condición *t* de una acción transaccional (*A*) y asignarle *true*. Al OcuR se le envía el nombre de la transacción (*B*), de la acción transaccional y del agente hijo (*P*) que la realiza. Las acciones que realiza el OcuR son:

    - Sustituir la información de esta acción transaccional en la tabla de transacciones.

- *defCAction(def,A_c, P):* Modificar la definición (*def*) de una acción compleja (*A_c*) de un agente hijo (*P*). La información que le llega al OcuR es en nombre de la transacción, del agente y su nueva definición. Las acciones que realiza son:

    - Sustituir en la tabla de transacciones la definición antigua por la nueva.

- *renameAction(A,B,P):* Cambiar de nombre una acción. Al OcuR se le envía el nombre del agente (*P*), de la acción antigua (*A*) y el nuevo nombre de la acción (*B*). El OcuR realiza las siguientes acciones:

    - Sustituye en la entrada de la antigua acción de dicho agente, el nombre de la acción por el nuevo nombre recibido.

- Se busca en la tabla de transacciones alguna que tenga en su definición la acción con el nombre antiguo y se sustituye por el nuevo nombre.

- *cloneAgent(P, Q):* Clonar un agente ya existente. Al OcuR le llega el nombre del nuevo agente (*Q*), y el nombre del agente (*P*) del cual se clonó. Las acciones que realiza OcuR son:

  - Crear una nueva entrada en la tabla de agentes.

  - Buscar la entrada correspondiente del agente desde el cual se clonó.

  - Copiar la tabla de acciones y enlazarla a la entrada del nuevo agente.

- *createAgent(P, Q):* Crear un agente. Al OcuR le llega el nombre del agente hijo creado (*P*). Lo único que hace es crear una entrada en la tabla de agentes (ya que es un clon del agente elemental que no realiza acciones) etiquetada con el nombre de dicho agente.

- *createSystem(P):* Crear un sistema (un agente sin padre). Puesto que no tiene padre, su información se almacena en su OcuR con el fin de que si alguna vez se agrega a otro sistema, se pueda obtener la información de las acciones que realiza.

- *agrAgent(P, Q):* Agregar un agente (*P*) a otro (*Q*). Al OcuR le llega la siguiente información: nombre del nuevo agente hijo, acciones que realiza, pre-condiciones *p* de éstas y si lleva a cabo transacciones: nombre de la transacción, definición y lista de acciones transaccionales con sus pre-condiciones *t*. Las acciones que realiza OcuR sobre sus estructuras son:

  - Crear una nueva entrada en la tabla de agentes.

  - Crear una tabla de acciones con una entrada por cada acción de dicho agente.

  - Para cada acción, crear o duplicar (en caso de que un agente hermano realice la misma acción) la tabla de pre-condiciones.

  - Si tiene acciones transaccionales, crear una entrada por cada una de ellas en la tabla de transacciones (si no existe ya).

- *disAgrAgent(P, Q):* Desagregar un agente (*P*) de otro (*Q*). Al OcuR le llega el nombre del agente a desagregar. Se borra la entrada correspondiente al agente desagregado y sus tablas de acciones y de

pre-condiciones asociadas. Las tablas de pre-condiciones compartidas por otros no las borra, sólo la referencias a éstas (decrementa el contador de referencias).

- *delAgent(P):* Borrar un agente hijo (*P*). Para cuestiones de actualización es igual a la acción anterior.

- *renameAgent(P, Q):* Cambiar de nombre a un agente. Al OcuR se le informa del nombre antiguo del agente (*P*) y de su nuevo nombre (*Q*). Se realizan las siguientes acciones:

  - Localizar la entrada de la tabla de agentes del agente renombrado.

  - Sustituir su antiguo nombre por el nuevo.

- *moveAction(A, P, Q):* Mueve una acción simple o compleja, *A*, desde un agente *P* a un agente *Q*. Por tanto, al OcuR de *P* se le debe comunicar que se le elimina la acción *A*. Sigue las mismas acciones que para la acción estructural *delAction*. Al OcuR de *Q* se le comunica que se le añade una nueva acción. Las acciones que éste realiza son las mismas que las descritas para las acciones *addSAction* o *addCAction* (dependiendo de si *A* es una acción simple o compleja).

- *moveAgent(P, Q, R):* Mover un agente desde un agente padre, *Q*, a otro, *R*. Al OcuR del agente *Q* se le informa de que se desagrega de él su agente *P*. Los pasos que sigue son los mismos que para la acción *disAgrAgent.* Al OcuR del agente *R* se le informa de que tienen un nuevo agente llamado *P*. Las acciones que realiza son las mismas que para la acción *createAgent*.

7.5.2 Actualizaciones en el *Evaluator*.

El agente *Evaluator* almacena información necesaria para realizar correctamente los servicios de consulta y evaluación de pre-condiciones. Para ello necesita las siguientes tablas:

◊ *Tabla de funcionamiento*: almacena las acciones que se pueden realizar por los agentes hijos de un agente junto con la expresión en lógica temporal de predicados de su pre-condición *p*.

tabla_funcionamiento = { nombre_acción + pre-condición_p }

◊ *Tabla de Acciones Incompatibles*: se tiene un entrada por cada acción que posea una lista de acciones incompatibles.

tabla_acciones_incompatibles = { nombre_acción + { nombre_acción } }

Las siguientes acciones estructurales provocan una actualización en la información del *Evaluator*:

- *addSAction(A_s, P):* Añadir una acción simple (*A_s*) a un agente hijo (*P*). Al *Evaluator* se le envía el nombre de la acción y su pre-condición *p*.

    - Crear una entrada más en la tabla de funcionamiento si no existe ya una con dicho nombre de acción.

- *addCAction(A_c, P):* Añadir una acción compleja (*A_c*) a un agente hijo (*P*). Igual que en el caso anterior.

- *addPPrec(X, A, P):* Añadir una pre-condición *p* (*X*) a una acción (simple o compleja, *A*). Al *Evaluator* se le manda el nombre de la acción y su nueva pre-condición *p*.

    - Buscar en la tabla de funcionamiento una entrada que coincida con el nombre de la acción y sustituir su pre-condición antigua por la nueva.

- *addLAI(list,A,P)*: Añadir o modificar la lista de acciones incompatibles a una acción, *A*, simple o compleja de un agente, *P*. Al Evaluator se le envía el nombre de la acción y la lista de acciones incompatibles.

    - Buscar en la tabla de acciones incompatibles una entrada para dicha acción, si no existe, crearla y si no, sustituir su lista de acciones incompatibles antigua por la nueva.

- *delAction(A, P):* Borrar una acción simple o compleja (*A*) de un agente hijo (*P*). Si esta acción sólo es realizada por un único agente, se envía al *Evaluator* en nombre de la acción a borrar.

    - Borrar la entrada correspondiente a dicha acción de la tabla de funcionamiento.

- *renameAction(A,B,P):* Cambiar de nombre una acción. Al *Evaluator* se le envía el nombre de la acción antigua (*A*) y el nuevo nombre de la acción (*B*).

    - Buscar una entrada en la tabla de funcionamiento donde aparezca el nombre antiguo de la acción y sustitirlo por el nuevo.

- *agrAgent(P, Q):* Agregar un agente (*P*) a otro (*Q*). Al *Evaluator* le llega una lista de sus acciones junto con las pre-condiciones *p* de éstas.

    - Para cada acción, comprobar si ésta existe en la tabla de funcionamiento, si no crear una nueva entrada con nombre de acción y pre-condición *p*.

- *disAgrAgent(P, Q):* Desagregar un agente (*P*) de otro (*Q*). Al *Evaluator* le llega una lista de las acciones que ya no se realizan.

    - Para cada acción, buscar la entrada en la tabla de funcionamiento y eliminarla.

- *delAgent(P):* Borrar un agente hijo (*P*). Para cuestiones de actualización es igual a la acción anterior.

- *moveAction(A, P, Q):* Mueve una acción simple o compleja, *A*, desde un agente *P* a un agente *Q*.

    - Si el agente Q no es hermano de P, buscar la entrada correspondiente a dicha acción en la tabla de funcionamiento y eliminarla.

7.6 Conclusiones.

La evolución es un objetivo importante en este trabajo. Se ha explicado el proceso de evolución entrando en detalle en el funcionamiento del Metasistema y de los Sistemas Genéticos. Se ha abordado la relación entre éstos y la forma en la que colaboran para conseguir que se lleven a cabo las modificaciones en los agentes.

Se han explicado primero las operaciones de creación de agentes y de agregación, por estar muy relacionadas con la estructura jerárquica de agentes que se puede contemplar en un sistema software.

Después, nos hemos centrado en el subsistema de decisión. Hemos descrito la lista de invariantes que se asocian con los sistemas software y que son propiedades que siempre se deben cumplir. Se ha descrito cada una de las acciones estructurales o de evolución, sus pre- y post-condiciones y el mecanismo de propagación de cambios requerido, si es necesario.

Finalmente hemos visto qué actualizaciones se realizarían en la información mantenida por los agentes *Ocurrence Receiver* y *Evaluator* después de que se lleve a cabo una acción estructural en el sistema.

CAPITULO 8

Caso práctico: Empresa de alquileres de coches

8.1 Descripción del problema.

Queremos crear un sistema software que modele un sistema real como es una empresa de alquileres de coches.

Primero tenemos que determinar cuáles son los elementos constitutivos del sistema basándonos en el modelo presentado: los agentes, las acciones, las pre-condiciones de éstas, etc.

El sistema ofrece cuatro acciones al exterior:

- alquilar un coche
- recoger el coche alquilado cuando el cliente lo devuelve
- cobrar el servicio de alquiler
- comprar más coches para la empresa

En este sistema, en un primer nivel, existen dos clases de empleados:

- Aquel que llamaremos *Trabajador* y que su función en la empresa es la de alquilar coches y recoger los coches después de que los clientes los devuelvan.

- El *Cobrador* que se encarga de cobrar a los clientes cuando alquilan un coche en la empresa y, además, se encarga de comprar nuevos coches cuando el dueño de la empresa lo disponga o en el momento de creación de la empresa (debe comprarse al menos un coche para iniciar el negocio).

Por convención, los nombres de los agentes comienzan en mayúscula, las acciones complejas o transacciones comienzan también en mayúscula y las acciones simples se escriben sólo en minúscula. De esta forma podemos distinguir una acción simple de una compleja fácilmente.

El diseño del sistema software que vamos a hacer está representado en la siguiente figura. Primero vamos a poner el árbol de agentes que especifica dicha empresa de alquiler de coches y después iremos comentando cada uno de los agentes junto con las acciones que realizan.

Como vemos en la siguiente figura tenemos realmente trece agentes. El primer agente está etiquetado como Sistema y, como puede observarse, es el agente raíz de la jerarquía y, por tanto, no tiene un agente padre.

En el primer nivel tenemos dos agentes con el comportamiento del *Trabajador* descrito anteriormente: realiza las acciones *alquilar* y *Recoger*.

El agente *Trabajador* tiene tres subagentes que se encargarán de realizar las acciones que tiene en su interfaz:

- *Empleado1*: realiza las acciones *alquilar* y *Lavar.*

- *Empleado2*: realiza las acciones *alquilar* y *Lavar.*

- *E_Taller* (empleado de taller) ya que la empresa se encarga también de reparar sus propios coches y mantenerlos en buen estado. Las acciones que lleva a cabo son *Comprobar* y *pintar*. Realiza chequeos de los coches una vez devueltos, los pinta si es necesario, los repara si tienen alguna pieza mal y tiene un pequeño almacén de piezas para realizar estos cometidos.

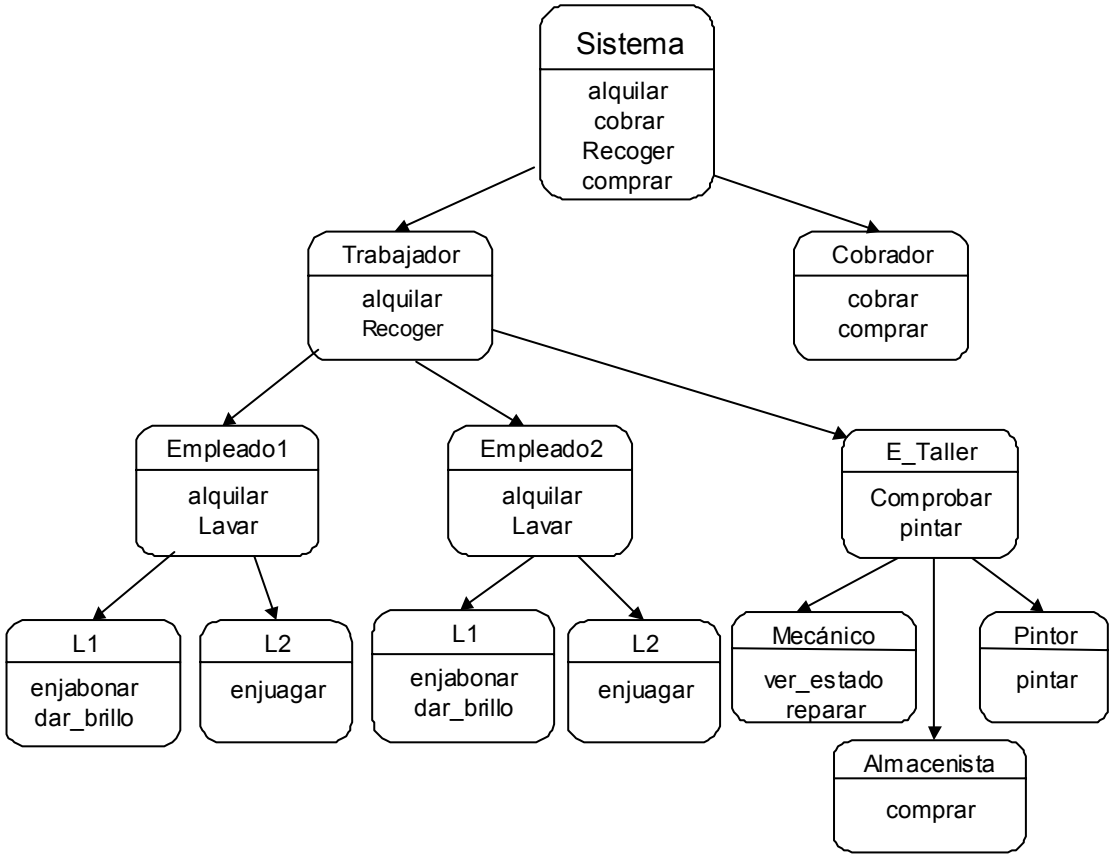


Figura 8.1 Sistema de alquileres de coches: árbol de agregación de agentes

Los agentes *Empleado1* y *Empleado2* son idénticos en cuanto a funcionalidad, ambos tienen dos subagentes que van a realizar las acciones siguientes:

- Los agentes *L1* (aunque tengan igual nombre como tienen distinto padre son diferentes) realizan las acciones *enjabonar* y *dar_brillo*.

- Los agentes *L2* llevan a cabo una única acción, *enjuagar*.

Nótese que cada agente hijo trabaja para su agente padre. Cada agente padre puede verse como el jefe que delega la realización de sus acciones, cuando le son encomendadas, en sus empleados.

El agente encargado del taller al que llamamos *E_Taller* también tiene tres agentes anidados:

- *Mecánico*: realiza las acciones *ver_estado* y *reparar.*

- *Almacenista*: se encarga de *comprar* los repuestos necesarios para reparar un coche. Nótese que esta acción es distinta a la acción

comprar del agente *Cobrador* que se encarga de adquirir nuevos coches.

- *Pintor*: lleva a cabo la acción *pintar.*

Por último, la empresa tiene otro agente en el primer nivel que es el agente:

- *Cobrador*: realiza las acciones *cobrar* y *comprar.*

8.2 Especificación del sistema.

Utilizando la notación introducida en el capítulo 7, podemos especificar el sistema de la siguiente forma:

Sistema[alquilar, cobrar, Recoger, comprar] = {Trabajador[alquilar, Recoger], Cobrador[cobrar, comprar] }

Trabajador[alquilar, Recoger] = {Empleado1[alquilar, Lavar], Empleado2[alquilar, Lavar], E_Taller[Comprobar, pintar] }

Empleado1[alquilar, Lavar] = { L1[enjabonar, dar_brillo], L2[enjuagar] }

Empleado1.L1[enjabonar, dar_brillo] = { }

Empleado1.L2[enjuagar] = { }

Nótese que si existe ambigüedad en los nombres de los agentes como es el caso de *L1* y *L2* que existen tanto en *Empleado1* como en *Empleado2* anteponemos el nombre de su agente padre para diferenciarlos. Esto se debe a que hay que recalcar que, aunque a simple vista se llamen igual, no son el mismo agente. Inicialmente coinciden en su comportamiento (acciones que pueden realizar) pero, después, cada uno puede evolucionar independientemente y podrían tener comportamientos muy diferentes.

Empleado2[alquilar, Lavar] = { L1[enjabonar, dar_brillo], L2[enjuagar] }

Empleado2.L1[enjabonar, dar_brillo] = { }

Empleado2.L2[enjuagar] = { }

E_Taller[Comprobar, pintar] = { Mecánico[ver_estado, reparar], Almacenista[comprar], Pintor[pintar] }

Mecánico[ver_estado, reparar] = { }

Almacenista[comprar] = { }

Pintor[pintar] = { }

Cobrador[cobrar, comprar] = { }

8.3 Descripción de las acciones y estímulos.

En este punto vamos a describir las acciones, simples o complejas, que pueden ser realizadas por cada agente del sistema utilizando el lenguaje L para especificar sus pre-condiciones. Nótese que no vamos a describir las acciones del agente Sistema ya que se puede entender que, como no tiene padre, no existe una historia funcional SFH donde consultar las pre-condiciones de sus acciones. Simplemente, se considera una interfaz útil si lo agregamos a otro sistema como un agente más, en ese caso, será cuando definamos las pre-condiciones de sus acciones que dependerán del nuevo entorno donde comenzará a operar.

Vamos a describir las pre-condiciones *p* de cada una de las acciones simples incluyendo un breve comentario sobre éstas. De las acciones complejas o transacciones, además de su pre-condición *p*, comentaremos su descripción utilizando el lenguaje TDL y las pre-condiciones *s* y *t* de cada una de las acciones transaccionales que intervienen en ella.

Abordaremos cada agente en particular porque, por ejemplo, la acción *alquilar* de *Trabajador* y de su subagente *Empleado1* tienen una pre-condición *p* distinta ya que realmente no son la misma acción. *Empleado1* lleva a cabo la acción *alquilar* porque, por ejemplo, se comprobó que la *p* de la acción *alquilar* de *Trabajador* era *true* y éste delegó en *Empleado1* la ejecución de dicha acción introduciendo un estímulo en su SFH. Dicho estímulo constituye la pre-condición *p* de la acción *alquilar* de *Empleado1*.

Paralelamente, comentaremos cada uno de los estímulos que utilizamos en el ejemplo, lo que significan y su procedencia. Los estímulos se tratan como acciones, tienen atributos que pueden ser consultados pero que provienen o bien del exterior (el entorno en el que se encuentra el sistema) por la acción directa de un usuario o un miembro del equipo

de desarrollo, o bien como consecuencia de la ejecución de alguna acción realizada por un agente del sistema.

En general, toda acción compleja o transacción, cuando se inicia, genera un estímulo de inicio de transacción que se insertará en la SFH del *Transaction Manager*, TM, creado para llevar a cabo dicha transacción de forma independiente. Será el primer elemento de la SFH del TM y servirá para que puedan activarse la o las primeras acciones transaccionales. Por seguir una misma notación, este estímulo se llamará *Stimulus* concatenado con el nombre de la acción compleja, por ejemplo, *StimulusRecoger* para la acción compleja *Recoger*.

Además, toda acción compleja tiene asociada una acción especial que siempre se pondrá al final de la definición de la transacción de forma secuencial, es decir, después de un operador ";". Así el TM sabrá que todas las acciones transaccionales han finalizado con éxito y, por tanto, se puede generar la ocurrencia de la acción compleja que será insertada en la SFH del *Controller* que inició dicha transacción. Inicialmente, a esta acción especial vamos a llamarla *terminar*. Al igual que cualquier acción transaccional, la acción *terminar* tiene asociadas las pre-condiciones siguientes:

- pre-condición *p* : tendrá valor *true* ya que como sólo tiene sentido dentro de la transacción, su ejecución no depende de ninguna acción realizada fuera de ella. El hecho de tener valor *true* no significa que siempre se pueda ejecutar, ya que tal y como se realiza la activación de las acciones de los agentes por el *Ocurrence Receiver*, ésta no se evaluará si no se ha iniciado la transacción donde se encuentra y han finalizado todas las acciones transaccionales anteriores.

- pre-condición *s*: su valor dependerá de las acciones transaccionales y los operadores que se encuentren en la definición de la transacción.

- pre-condición *t*: tendrá valor *true*, ya que esta acción sólo existe para indicar al TM que la transacción ha finalizado con éxito y para que genere la ocurrencia de la transacción oportuna.

8.3.1 Agente Trabajador.

Este agente realiza las siguientes dos acciones:

8.3.1.1 Acción alquilar.

Es una acción simple.

Pre-condición *p*: para alquilar un coche, previamente (operador que se supone implícito) se ha tenido que comprar y, si alguna vez se ha alquilado, éste debe haber sido devuelto a la empresa. Además, debe existir un cliente interesado en alquilarlo.

alquilar(C) ← *comprar(C) and (not alquilar(C) since Recoger(C)) and StimulusLlegaCliente(C)*

StimulusLlegaCliente(C) : este estímulo procede del exterior del sistema, se produce cuando llega un cliente que necesita alquilar un coche (puede ser la pulsación de un botón en una ventana de solicitud de alquiler).

8.3.1.2 Acción Recoger.

Es una acción compleja o transacción. Cuando se devuelve un coche, el sistema tiene que comprobar si se encuentra en buen estado y, secuencialmente, puede ocurrir que tenga que ser o bien lavado o bien pintado para poder volver a alquilarlo. Su descripción con el lenguaje TDL es la siguiente:

Recoger(C) = Comprobar(C) ; (Lavar(C) | pintar(C)) ; terminar

Pre-condición *p* de la acción compleja: cuando un cliente desea devolver un coche, introduce a través de la interfaz de acción un estímulo de devolución indicando que el coche debe ser recogido por algún empleado de la empresa. Este estímulo es su pre-condición *p*.

Recoger(C) ← *StimulusDevolver(C)*

StimulusDevolver(C) : procede de la interfaz externa cuando un cliente llega a la empresa y realiza la devolución del coche *C*.

Pre-condiciones *s* de cada acción transaccional:

Recordemos que las pre-condiciones *s* de cada acción transaccional se obtienen de la definición de la transacción.

De *Comprobar* es *StimulusRecoger* ya que como se comentó anteriormente, en la historia del *TM* creado para gestionar la transacción debe existir un estímulo que provoque la ejecución de la primera o primeras acciones transaccionales.

$$Comprobar(C) \leftarrow^{s}- \ StimulusRecoger(C)$$

StimulusRecoger(C) : es un estímulo de inicio de transacción. Indica que debe comenzar el proceso de comprobación del buen estado del coche antes de ponerlo como disponible para alquilar.

De *Lavar* y de *pintar* es que exista una ocurrencia de *Comprobar* en la historia del TM.

$$Lavar(C) \leftarrow^{s}- \ Comprobar(C)$$
$$pintar(C) \leftarrow^{s}- \ Comprobar(C)$$

De *terminar* es:

$$terminar \leftarrow^{s}- \ pintar(C) \ or \ Lavar(C)$$

Pre-condiciones *t* de cada acción transaccional:

La pre-condición *t* de *Comprobar* es *true*, lo cual es lógico porque debe ser la primera acción a realizar dentro de la transacción y no ha podido suceder nada para que dicha transacción deba interrumpirse.

$$Comprobar(C) \leftarrow^{t}- \ true$$

La *t* de *Lavar* y *pintar* es que no se haya recibido un estímulo indicando que el coche está fuera de servicio.

$$Lavar(C) \leftarrow^{t}- \ not \ StimulusFueraServicio(C)$$
$$pintar(C) \leftarrow^{t}- \ not \ StimulusFueraServicio(C)$$

StimulusFueraServicio(C) : es generado por la acción *Comprobar* en el caso, por ejemplo, que el coche se haya dado de baja debido a su robo, a una avería o a que se produjo un accidente que lo haya dejado totalmente inservible.

La *t* de *terminar* es *true*.

$$terminar \leftarrow^{t} true$$

8.3.2 Agente Cobrador.

Las dos acciones que es capaz de realizar son:

8.3.2.1 Acción comprar.

Es una acción simple.

Pre-condición *p*: Cuando el propietario de la empresa decide aumentar el parque automovilístico de ésta o cuando se crea la empresa, envía un estímulo mediante la interfaz de acción para que se inicie la compra de un coche.

comprar(C) ← *StimulusAumentarCoches(C)* /**D*el agente *Cobrador*/*

StimulusAumentarCoches(C) : procede del dueño de la empresa cuando decide aumentar el número de coches disponibles para alquilar. El dueño de la empresa será un usuario del sistema software e introduce el estímulo desde el exterior, a través de la interfaz de acción.

8.3.2.2 Acción cobrar.

Es una acción simple.

Pre-condición *p*: para cobrar un alquiler de un coche a un cliente, éste debe rellenar un formulario una vez decidida y realizada la operación de alquiler.

cobrar(C) ← *StimulusRellenar_formulario(C) since alquilar(C)*

StimulusRellenar_formulario(C) : cuando se le alquila un coche a un cliente, antes de pagar dicho cliente debe rellenar un formulario asociado al coche C.

8.3.3 Agentes Empleado1 y Empleado2.

Ambos agentes son idénticos en cuanto a funcionalidad, es decir, realizan las mismas acciones, y estructura, son clones. Vamos a comentar sus acciones al mismo tiempo. Estos agentes realizan las siguientes dos acciones:

8.3.3.1 Acción alquilar.

Es una acción simple.

Pre-condición *p*:

alquilar(C) ← *StimulusAlquilar(C)*

StimulusAlquilar(C): procede del agente padre *Trabajador* que, a través de la interfaz de acción introduce este estímulo en el agente *Empleado* de tal forma que delega ese trabajo en él.

8.3.3.2 Acción Lavar.

Es una acción compleja. Paralelamente se enjabona y enjuaga el coche y cuando ya se ha terminado, entonces se le da brillo al coche y ya está listo para poder ser alquilado de nuevo. Su descripción con TDL es la siguiente:

Lavar(C) = (enjabonar(C) || enjuagar(C)) ; dar_brillo(C) ; terminar

Pre-condición *p* de la acción compleja:

La acción de lavar un coche es el resultado de haber inspeccionado el estado del coche y determinar, generando un estímulo, que debe lavarse antes de poder alquilarse de nuevo. Esta es una acción compleja ya que implica enjabonar, enjuagar y dar brillo a la carrocería del coche. Al estar únicamente dentro de la transacción *Recoger*, no depende de las acciones realizadas fuera de dicha transacción, por tanto su pre-condición *p* es:

Lavar(C) ← *true*

Pre-condiciones *s* de cada acción transaccional:

La *s* tanto de *enjabonar* como de *enjuagar* es que se haya recibido *StimulusLavar* generado cuando se crea el TM para gestionar la transacción y con el fin de que se inicien las acciones transaccionales.

$$enjabonar(C) \leftarrow^{s}_{-} \; StimulusLavar(C)$$
$$enjuagar(C) \leftarrow^{s}_{-} \; StimulusLavar(C)$$

La *s* de *dar_brillo* es *enjabonar and enjuagar*, es decir antes de dar brillo se han tenido que terminar estos dos procesos.

$$dar\_brillo(C) \leftarrow^{s}_{-} \; enjabonar(C) \; and \; enjuagar(C)$$

La *s* de *terminar* es:

$$terminar \leftarrow^{s}_{-} \; dar\_brillo(C)$$

Pre-condiciones *t* de cada acción transaccional:

Las cuatro tendrían *t = true.* Podríamos pensar que si mientras lo lavan detectan alguna irregularidad (está roto el faro o una puerta) se puede generar un estímulo que aborte la operación de dar_brillo. Entonces, la *t* de ésta sería *StimulusNoTerminar*.

$$enjuagar(C) \leftarrow^{t}_{-} \; true$$
$$enjabonar(C) \leftarrow^{t}_{-} \; true$$
$$dar\_brillo(C) \leftarrow^{t}_{-} \; true$$
$$terminar \leftarrow^{t}_{-} \; true$$

8.3.4 Agente E_Taller.

Sus acciones son las siguientes.

8.3.4.1 Acción Comprobar.

Es una acción compleja. La acción *Comprobar* realiza un chequeo del coche recogido e indica si es necesario comprar alguna pieza porque no se encuentre en el almacén del taller. Si el *ver_estado* determina que no es necesaria la reparación, las acciones siguientes simplemente no realizarán tarea alguna. Descripción en TDL:

Comprobar(C) = ver_estado(C) ; comprar(pieza) ; reparar(C) ; terminar

Pre-condición *p* de la acción compleja:

Antes de volver a alquilar un coche, los empleados del taller deben comprobar su buen estado, tanto desde el punto de vista mecánico como estético (limpio y buen estado de la pintura). Recordemos esta acción tiene lugar dentro de otra transacción (y sólo se ejecuta en dicha transacción), *Recoger*, y, por tanto, la pre-condición p es *true*.

Comprobar(C) ← *true*

Pre-condiciones *s* de cada acción transaccional:

De la acción *ver_estado:*

ver_estado(C) $\leftarrow^{S-}$ *StimulusComprobar(C)*

De la acción *comprar* debe existir una ocurrencia de la acción *ver_estado* y un estímulo *StimulusComprar* que proporciona información de qué pieza hay que comprar (o incluso si no hay que comprar ninguna). Este estímulo es generado por la acción *ver_estado* después de realizar las comprobaciones oportunas.

comprar(pieza) $\leftarrow^{S-}$ *ver_estado(C) and StimulusComprar(pieza)*

De la acción *reparar:*

reparar(C) $\leftarrow^{S-}$ *comprar(pieza)*

La *s* de *terminar* es:

terminar $\leftarrow^{S-}$ *reparar(C)*

Pre-condiciones *t* de cada acción transaccional:

Las *t* de las cuatro acciones transaccionales es *true*.

8.3.4.2 Acción pintar.

Es una acción simple.

Pre-condición *p*: Cualquier trabajador puede determinar que el coche necesita una mano de pintura. Por ejemplo, *Empleado1*, cuando realiza la acción *Lavar* y como uno de los resultados de ésta puede generar un estímulo *StimulusNecesitaPintura(C)* que haría que se deba activar la acción *pintar* de *E_Taller*. O incluso, *E_Taller*, después de ejecutar la acción *Comprobar* puede generar dicho estímulo por estimar que el coche necesita un repaso. Por tanto, su pre-condición p es:

pintar(C) ← *StimulusNecesitaPintura(C)*

Sin embargo, como esta acción puede realizarse dentro de una transacción, en este caso su *p* debe ser *true* ya que no es necesario que se dé ningún estímulo realizado por un agente fuera del contexto de la transacción para que se active. Por tanto, también tenemos que:

pintar(C) ← *true*

Como una misma acción no puede tener dos pre-condiciones *p* distintas, según está establecido en el modelo y definido por el invariante de Posibilidad de Realización de Acciones (invariate 4), podemos definir esta pre-condición de la siguiente forma:

pintar(C) ← *true or StimulusNecesitaPintura(C)*

Recordemos que, aunque esta expresión no tiene mucho sentido en lógica ya que siempre es *true*, es coherente con la forma de activar las acciones que existe en nuestro modelo. Se abarcan de esta forma las dos situaciones siguientes:

- Si la acción se activa fuera de una transacción es porque el *Ocurrence Receiver* ha recibido un estímulo *StimulusNecesitaPintura*. El valor *true* nunca es enviado al OcuR para que sea almacenado en la historia del agente, por tanto la única forma de evaluar la pre-condición de *pintar* fuera de la transacción es que se haya recibido dicho estímulo. Una vez informado al agente *E_Taller* de que tiene probabilidad de realizar esta acción, inicia la evaluación de su pre-condición dando como resultado *true*. Por tanto, el agente *E_Taller* lleva a cabo esta acción.

- Si la activación se encuentra dentro de la transacción, no es necesario dicho estímulo y, por tanto, cuando se evalúe la *p* de esta acción transaccional, ésta debe ser *true* para que se inicie la transacción. Si no se hiciera así, estamos obligando a que deba existir un estímulo en la historia funcional que no siempre debe producirse porque existen dos situaciones distintas que llevan a la realización de esta acción en el sistema software. Una porque se

inicia la transacción y otra originada por la realización de la acción *Lavar*.

8.3.5 Agentes Empleado1.L1 y Empleado2.L1.

Como hemos dicho, ambos realizan las mismas acciones y por tanto, éstas son idénticas en cuanto a su comportamiento y pre-condiciones.

8.3.5.1 Acción enjabonar.

Es una acción simple.

Pre-condición *p*: al realizarse únicamente dentro de una transacción (*Lavar*) es *true*.

enjabonar(C) ← *true*

8.3.5.2 Acción dar_brillo.

Es una acción simple.

Pre-condición p: por las mismas razones que en la acción anterior es *true*.

dar_brillo(C) ← *true*

8.3.6 Agentes Empleado1.L2 y Empleado2.L2.

La única acción que realiza es:

8.3.6.1 Acción enjuagar.

Es una acción simple.

Pre-condición *p*: al ejecutarse únicamente si la transacción Lavar se inicia, su *p* es *true*.

enjuagar(C) ← *true*

8.3.7 Agente Mecánico.

Sus dos acciones son las siguientes.

8.3.7.1 Acción ver_estado.

Es una acción simple.

Pre-condición *p*: esta acción se activa cuando, al recoger un coche, se inicia el proceso de comprobación de su estado. Al ser la primera acción a realizar dentro de la transacción *Comprobar*, su pre-condición se construye en base al estímulo, *StimulusComprobar*, que se depositará en la historia del TM creado para gestionarla.

ver_estado(C) ← *StimulusComprobar(C)*

8.3.7.2 Acción reparar.

Es una acción simple.

Pre-condición *p*: Tras el proceso de estudio del estado del coche el *Mecánico* determina si debe ser reparado o no (si no, esta acción no haría nada, sólo generaría la ocurrencia donde en uno de sus atributos se almacenaría la información de que el coche se encontraba en perfecto estado). Como sólo tiene sentido dentro de la transacción, su pre-condición es *true*.

reparar(C) ← *true*

8.3.8 Agente Almacenista.

Su única acción es *comprar* y es una acción simple.

Pre-condición *p*: cuando en el chequeo del coche se comprueba que se necesita una pieza, el Almacenista debe comprarla o proporcionarsela al *Mecánico* si la tuviera en el almacén. Sin embargo, como esta acción sólo tiene sentido dentro de la transacción *Comprobar*, su pre-condición *p* será *true* ya que no depende de ninguna acción realizada fuera de dicha transacción.

comprar(pieza) ← *true* /**comprar* del agente *Almacenista*/*

8.3.9 Agente Pintor.

Su única acción es pintar y ésta es simple.

Pre-condición *p*: como esta acción sólo tiene lugar si se ha activado el *pintar* de *E_Taller*, sólo necesita un estímulo de inicio.

pintar(C) ← *StimulusPintar(C)*

8.3.10 Resumen.

En las siguientes tablas se resumen las actividades de cada uno de los agentes que forman el sistema, indicando (en este orden):

- El nombre de las acciones que realiza
- Su pre-condición *p*
- Si son acciones complejas (transacciones) o no (lo denotamos como C). Si lo son indicamos la descripción de dicha transacción.
- Si se encuentran en la definición de una transacción, el nombre de ésta y sus pre-condiciones *s* y *t*.

Se puede observar que las acciones *terminar* que aparecen en todas las descripciones de acciones complejas no aparecen como acciones que realice algún agente. Esto se debe a que es una forma de indicar al TM que se crea para gestionar dicha transacción que ésta ha finalizado con éxito y que debe generar su ocurrencia y comunicársela al *Controller* quien la almacenará en la SFH. Después de esto, el TM finalizará y el sistema ya conocerá que dicha transacción se ha realizado.

Acciones	**pre-condición *p***	**C**	**En compleja Nombre | *s* | *t***
alquilar(C)	*comprar(C) and (not alquilar(C) since Recoger(C)) and StimulusLlegaCliente(C)*	No	No
Recoger(C)	*StimulusDevolver(C)*	*Comprobar(C) ; (Lavar(C) | pintar(C)); terminar*	No

Tabla 8.1 Descripción del agente Trabajador

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
cobrar(C)	*StimulusRellenar_formulario(C) since alquilar(C)*	No	No
comprar(C)	*StimulusAumentarCoches(C)*	No	No

Tabla 8.2 Descripción del agente Cobrador

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
alquilar(C)	*StimulusAlquilar(C)*	No	No
Lavar(C)	*true*	*(enjabonar(C) || enjuagar(C)) ; dar_brillo(C) ; terminar*	Recoger *s = Comprobar(C)* *t = not StimulusFueraServicio (C)*

Tabla 8.3 Descripción de los agentes Empleado1 y Empleado2

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
Comprobar(C)	*true*	*ver_estado(C); comprar(pieza); reparar(C); terminar*	Recoger *s = StimulusRecoger(C)* *t = true*
pintar(C)	*true or StimulusNecesitaPintura(C)*	No	Recoger *s = Comprobar(C)* t = not *StimulusFueraServicio(C)*

Tabla 8.4 Descripción del agente E_Taller

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
enjabonar(C)	*true*	No	Lavar *s = StimulusLavar(C)* *t = true*
dar_brillo(C)	*true*	No	Lavar *s = enjabonar(C) and enjuagar(C)* *t = true*

Tabla 8.5 Descripción de los agentes Empleado1.L1 y Empleado2.L1

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
enjuagar(C)	*true*	No	Lavar *s = StimulusLavar(C)* *t = true*

Tabla 8.6 Descripción de los agentes Empleado1.L2 y Empleado2.L2

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
ver_estado(C)	*StimulusComprobar(C)*	No	Comprobar *s = StimulusComprobar(C)* *t = true*
reparar(C)	*true*	No	Comprobar *s = comprar(pieza)* *t = true*

Tabla 8.7 Descripción del agente Mecánico

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
comprar(pieza)	*true*	No	Comprobar *s = ver_estado(C) and* *StimulusComprar(pieza)* *t = true*

Tabla 8.8 Descripción el agente Almacenista

Acciones	pre-condición *p*	C	En compleja Nombre | *s* | *t*
pintar(C)	*StimulusPintar(C)*	No	No

Tabla 8.9 Descripción del agente Pintor

8.4 Representación del sistema software.

En la siguiente figura representamos el sistema software con sus distintos componentes y la jerarquía de agentes que lo define.

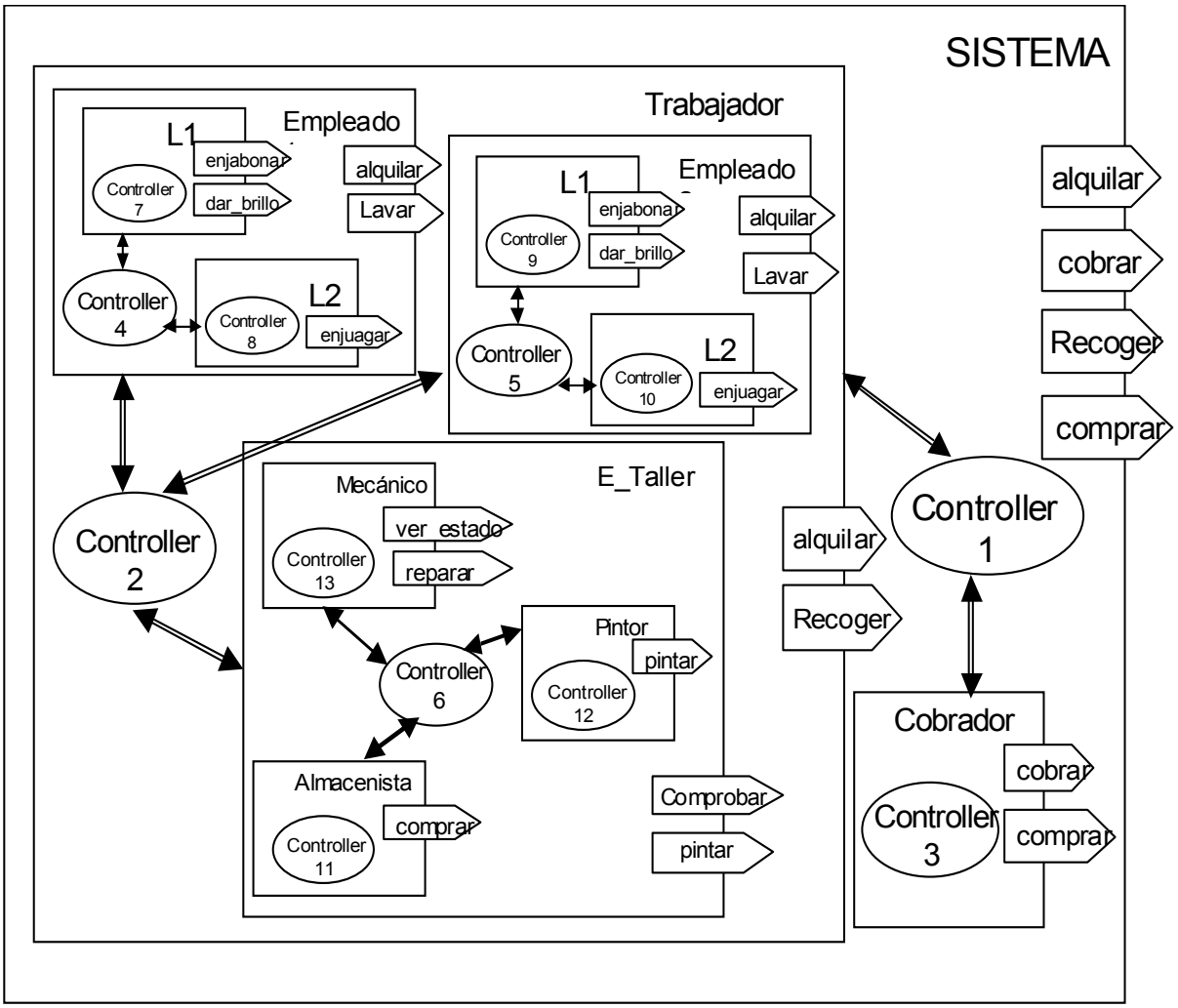


Figura 8.2 Sistema Software para modelar la empresa de alquileres de coches

Nótese que cada agente tiene su propio *Controller*, incluso el agente *Cobrador* que no tiene agentes anidados. Esto se corresponde con la definición básica de agente. Aunque el *Controller* de *Cobrador* no hace nada por ahora, si éste en cualquier momento evoluciona y se le agrega uno o más agentes, su *Controller* comenzará a funcionar. Los agentes que comparten el mismo *Controller*, son agentes hermanos.

8.5 Funcionamiento del sistema software.

Para comprender mejor el funcionamiento del sistema software, en las siguientes figuras vamos a representar la información que tiene cada uno de los agentes *Controller* que relaciona los agentes, sus acciones y las ocurrencias y/o estímulos y cómo se utiliza para conseguir que se activen los agentes. En realidad, dicha información es mantenida y utilizada por el agente *Ocurrence Receiver* de cada *Controller.*

La siguiente figura corresponde a la información que mantiene el *Controller 1*:

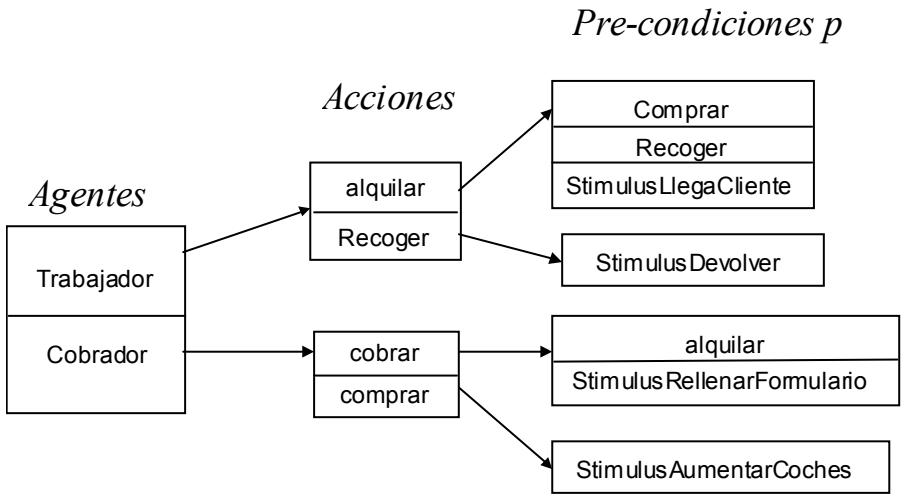


Figura 8.3 Información sobre agentes-acciones-ocurrencias y/o estímulos en el Controller 1 (agente Sistema)

En el *Controller 2*, correspondiente al agente *Trabajador*, se puede observar que algunas acciones tienen asociada como pre-condición *true*. Esto es para enfatizar el hecho de que, realmente, ninguna acción ocurrida en el sistema fuera de la transacción donde se halla esta acción, puede activarla. Quizás, lo mejor es no poner nada pero, entonces, podríamos pensar que no tiene pre-condición *p*.

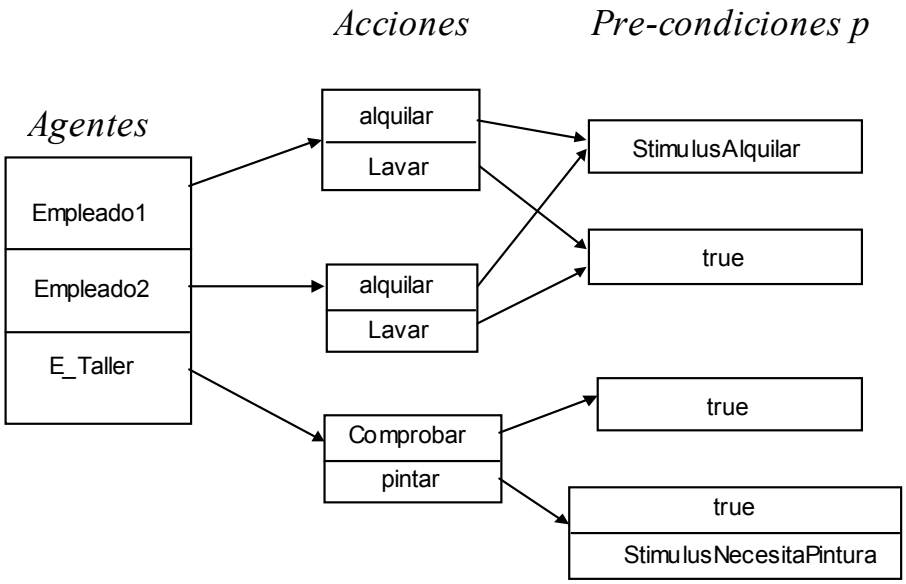


Figura 8.4 Información mantenida por el Controller 2 (agente Trabajador)

En las siguientes figuras, representamos la información que mantienen los *Controller 4*, *5* y *6*. En el resto de agentes, al no tener agentes anidados, aunque existe el *Controller* correspondiente, éste no mantiene ninguna información ni realiza tarea alguna:

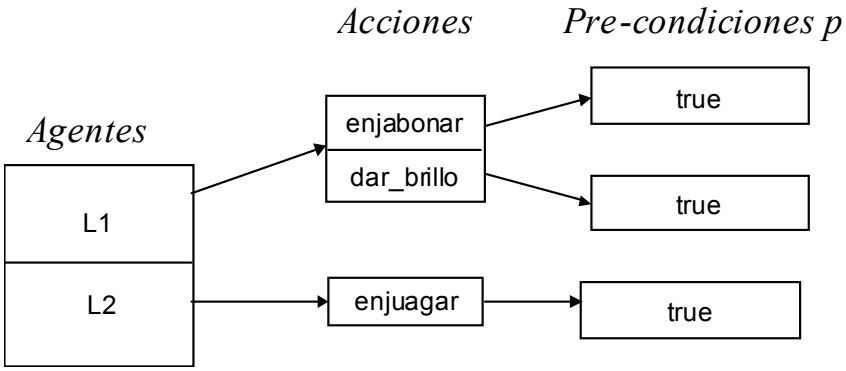


Figura 8.5 Controller 4 y 5 (agentes Empleado1 y Empleado2)

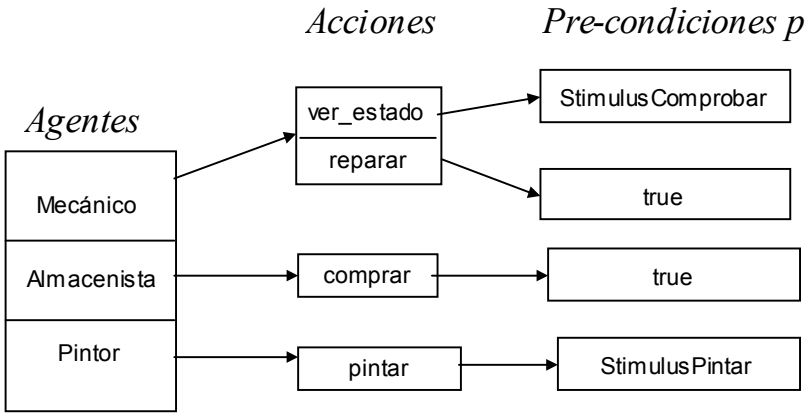


Figura 8.6 Controller 6 (agente E_Taller)

La generación de un estímulo de inicio de transacción, hace que el *Controller* cree un TM apropiado para gestionarla. La única información que el TM necesita conocer para crear las tablas que relacionan los agentes, sus acciones (ahora transaccionales) y las ocurrencias y/o estímulos es la definición de la transacción (de donde se obtiene la pre-condición s de cada acción transaccional) y las pre-condiciones *t* de cada uno de sus componentes.

En las siguientes figuras vemos la información de los TM de cada una de las posibles transacciones existen en nuestro sistema. La primera figura representa la información mantenida por el TM para la transacción Recoger.

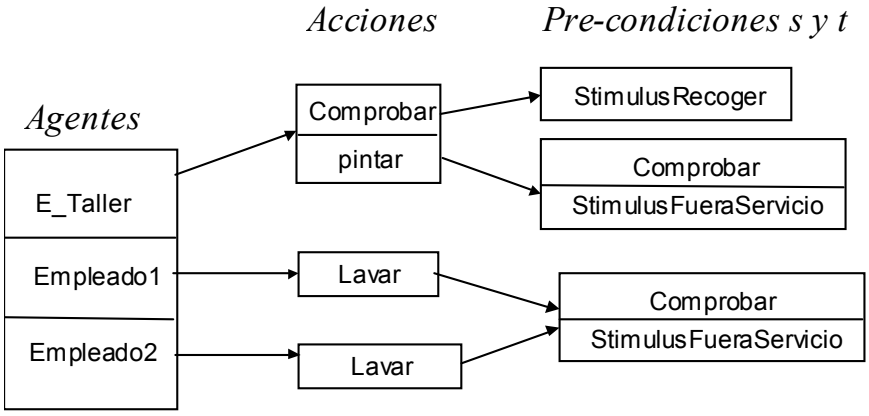


Figura 8.7 TM para la transacción Recoger

La transacción *Lavar* que realizan tanto *Empleado1* como *Empleado2* mantiene la siguiente información:

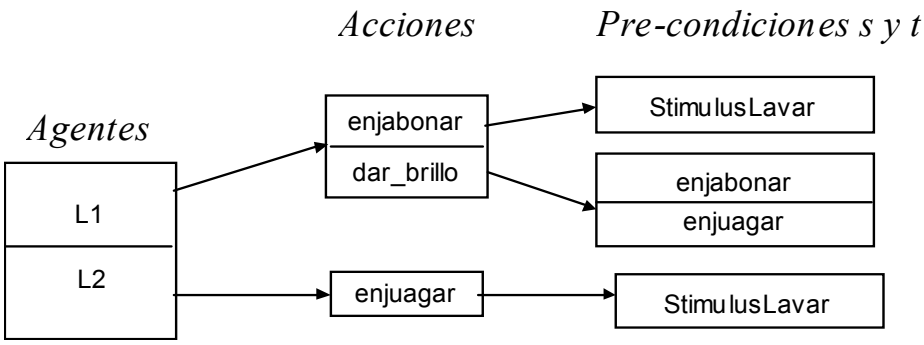


Figura 8.8 TM para Lavar

La última transacción, la acción *Comprobar*, realizada por el agente *E_Taller* y que, a su vez, se encuentra dentro de la transacción *Recoger* tiene un TM con la siguiente información:

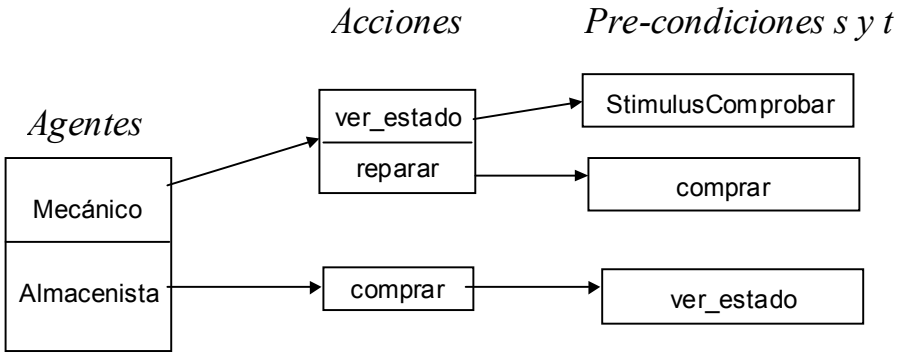


Figura 8.9 TM para Comprobar

8.6 Ejemplo de funcionamiento.

Este sistema software comenzará a funcionar cuando exista algún coche en la empresa que se pueda alquilar. Con el fin de que se vea más claro, iremos explicando paso a paso dos posibles trazas en el funcionamiento del sistema software.

8.6.1 Inicio de una acción simple.

Paso 1: Un usuario, probablemente el dueño de la empresa, introduce un estímulo, *StimulusAumentarCoches(C),* a través de la interfaz de acción del agente Sistema. Dicho estímulo lo recibe el *Ocurrence Receiver* del *Controller 1* quien lo almacena en la SFH del agente Sistema (véase etiqueta A en la figura siguiente).

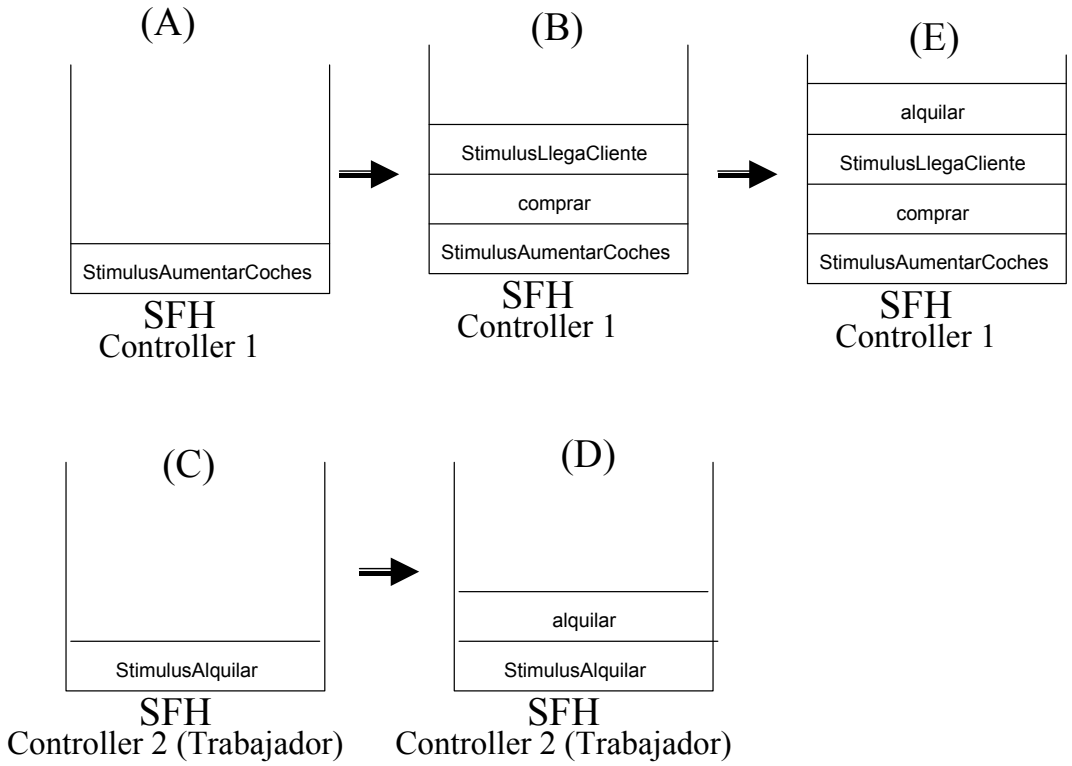


Figura 8.10 Ejemplo de funcionamiento del Sistema

Paso 2: Al llegar este estímulo, el *Ocurrence Receiver* del *Controller 1* consulta la información que relaciona los agentes, sus acciones y las ocurrencias y/o estímulos que forman parte de la pre-condición de cada una (figura 8.3) y determina que existe una posibilidad de que el agente *Cobrador* active su acción *comprar*, ya que este estímulo forma parte de la pre-condición de dicha acción. Avisa al agente para que intente realizar su acción *comprar*.

Paso 3: El agente *Cobrador* solicita que se evalúe la pre-condición de la acción *comprar.* El *Evaluator* la evalúa a *true* al encontrarse una ocurrencia de este estímulo en la historia.

Paso 4: El agente *Cobrador* inicia la realización de la acción *comprar*. Cuando finalice, se enviará al *Controller 1* la ocurrencia de dicha acción

que se almacenará en la correspondiente SFH (etiqueta B en la figura 8.10). En este instante, la empresa ya cuenta con un coche listo para alquilar.

Paso 5: Cuando el *Ocurrence Receiver* recibe la ocurrencia de la acción *comprar*, comprueba qué agentes y acciones pueden verse afectados. En este caso, es la acción *alquilar* del agente *Trabajador* la que tiene posibilidades de éxito. Se avisa a dicho agente.

Paso 6: Llega un cliente que desea alquilar un coche. Se almacena un estímulo *StimulusLlegaCliente* al *Controller 1* y se almacena en su SFH (etiqueta B).

Paso 7: El agente *Trabajador* solicita la evaluación de la pre-condición de su acción *alquilar* y el *Evaluator* del *Controller 1* comprueba que ésta es cierta y se devuelve esta respuesta al agente solicitante.

Paso 8: El agente *Trabajador* inicia la acción *alquilar*. Sin embargo, como el agente *Trabajador* delega la realización de dicha acción en los agentes *Empleado1* y *Empleado2*, introduce en su SFH (a través de su *Controller*, el *Controller 2*) un *StimulusAlquilar* (etiqueta C de la figura 8.10) que constituirá la pre-condición *p* de la acción *alquilar* de sus dos subagentes (véase la figura 8.4 anterior).

Paso 9: Puesto que sólo existe un coche para alquilar, sólo uno de los subagentes podrá realizar la acción *alquilar*. Una vez llevada a cabo, se almacena la ocurrencia de acción en la SFH de *Trabajador* (etiqueta D de la figura 8.10) y en la SFH de *Sistema* (etiqueta E de la figura 8.10).

8.6.2 Inicio de una acción compleja o transacción.

En cuanto a lo que ocurre en el sistema software en el caso de que se inicie una transacción, veremos un ejemplo partiendo del estado (etiqueta E en la figura 8.11) en el que se quedó el sistema software en el ejemplo anterior. Como antes, seguiremos una traza de su funcionamiento.

Paso 1: Un cliente quiere realizar la devolución de un coche, para ello introduce a través de la interfaz de acción un estímulo de devolución, *StimulusDevolver*, que se almacena en la SFH del *Controller 1* (etiqueta F en la figura 8.11).

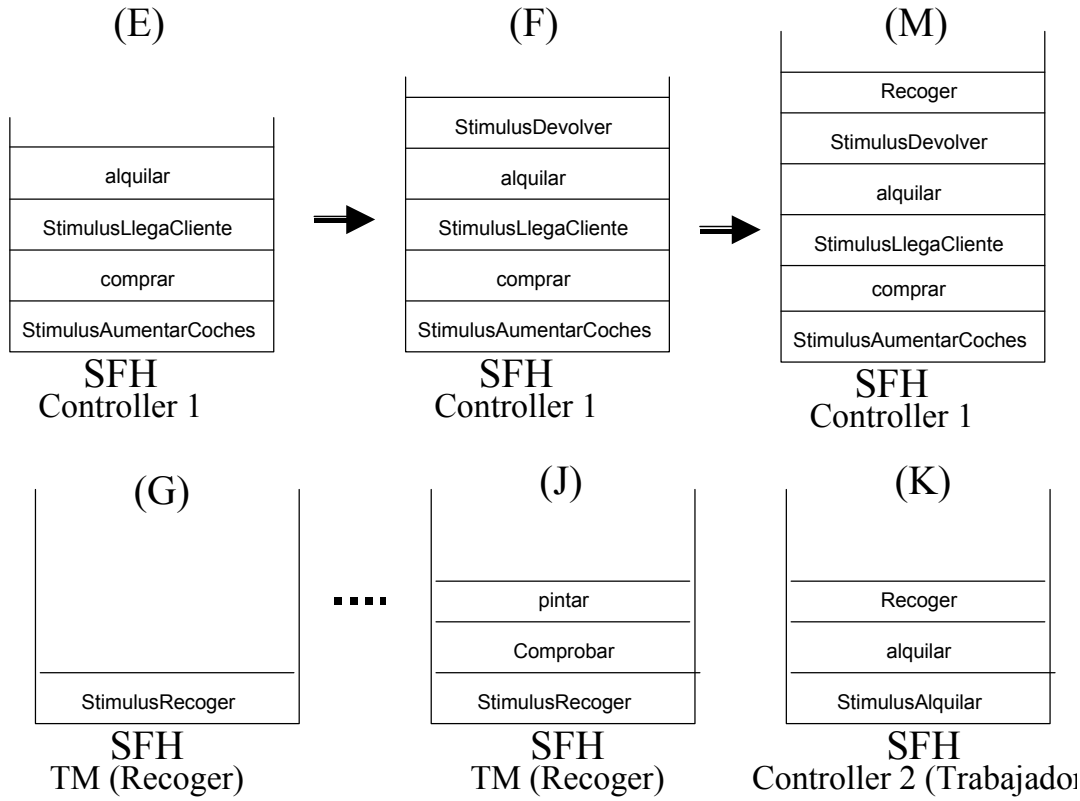


Figura 8.11 Ejemplo de funcionamiento de una transacción

Paso 2: El *Ocurrence Receiver* del *Controller 1* avisa al agente *Trabajador* que tiene probabilidades de activar su acción *Recoger*. Este agente envía la solicitud de evaluación de la pre-condición *p* de dicha acción. El *Evaluator*, según el estado de la historia, determina que ésta es cierta ya que existe un *StimulusDevolver* en ella.

Paso 3: La acción *Recoger* es una acción compleja, por tanto, *Trabajador* manda un estímulo, *StimulusRecoger,* al *Controller 2* que es quien gestiona su SFH.

Paso 4: El *Ocurrence Receiver* de *Controller 2* gracias a la Tabla de Transacciones, detecta que dicho estímulo se corresponde con la activación de una transacción, por tanto, tiene que avisar al *Evaluator* para que complete la evaluación de las pre-condiciones *p* de las acciones transaccionales sobre la SFH de *Controller 2*.

Paso 5: El *Evaluator* evalúa sobre la SFH, en este caso y por la descripción de la transacción, la *p* de *Comprobar*, de *Lavar* y de *pintar*. Como las acciones *Lavar* y *pintar* están unidas por un operador lógico (*or*), para poder iniciar la transacción deben ser ciertas:

- La *p* de *Recoger* : lo es ya que el estímulo necesario está en la SFH
- La *p* de *Comprobar* : lo es ya que ésta es *true* por defecto

- La *p* de *Lavar* o de *pintar* : en este caso ambas son *true* por definición

Si la evaluación hubiera dado *false*, la transacción no se podría haber iniciado y debe llegar un nuevo estímulo al *Controller 1* para que se vuelva a intentar.

Paso 6: Como la evaluación fue cierta, el *Ocurrence Receiver* de *Controller 2* crea el TM apropiado para que se inicie la transacción y le envía el estímulo que inicia dicha transacción, en este caso, *StimulusRecoger*. En este proceso de creación le suministra la información que relaciona los agentes, las acciones y sus pre-condiciones pero sólo de aquellos que están involucrados en la realización de la transacción. En la figura 8.7 anterior se puede ver la información mantenida por el TM que gestionará esta transacción.

Paso 7: El estímulo se almacena en la SFH del TM creado (etiqueta G de la figura 8.11). El funcionamiento del TM es idéntico al de un *Controller*, por tanto, la recepción de este estímulo hace que el TM avise al agente *E_Taller* porque tiene posibilidad de realizar su acción *Comprobar*.

Paso 8: El proceso es el mismo que hemos visto anteriormente, una vez realizada la acción compleja *Comprobar* (para la cual también se habrá creado su TM), se lleva a cabo, por ejemplo, la acción *pintar* y llegamos al punto etiquetado como J de la figura 8.11.

Paso 9: Todas las acciones transaccionales se han realizado, sólo falta la acción *terminar*. Esta acción le indica al TM que la transacción ha llegado a su fin con éxito y, por tanto, el TM puede generar una ocurrencia de la acción *Recoger*. El TM envía esta ocurrencia al *Ocurrence Receiver* que lo creó quien lo almacena en la SFH de *Trabajador* (etiqueta K de la figura 8.11). A continuación el *Controller 5* (de *Trabajador*) envía dicha ocurrencia al *Controller 1* para que la almacene en la SFH del *Sistema* (etiqueta M).

8.7 Evolución del sistema software.

A continuación vamos a explicar cómo se llevarían a cabo distintas acciones estructurales sobre el sistema que ha constituido nuestro ejemplo. Primero vamos a comentar un poco el contenido de la historia funcional del Metasistema, MFH. La unión de todas las historias estructurales, SSH, asociadas a los agentes del sistema constituye la MFH donde el Metasistema comprueba las pre-condiciones asociadas a las acciones estructurales del sistema software. Por claridad, presentaremos las SSH de los agentes en vez de la MFH que podría confundir a los lectores y no la de todos los agentes, sino las que

necesitamos para aclarar cómo se realiza el proceso de evolución. Después, partiendo del estado en el que ya tenemos construida la jerarquía de agentes que modelan nuestro sistema software y que se representó en la figura 8.1, presentaremos tres ejemplos relacionados con distintas etapas de su evolución.

8.7.1 Construcción del Sistema de alquileres de coches.

En la siguiente figura, mostramos la SSH del agente Sistema. Se encuentran todas las acciones estructurales relacionadas con la creación del sistema software, junto con sus acciones y con la de los agentes a un nivel (*Trabajador* y *Cobrador*).

Con el fin de facilitar las referencias al contenido de cada historia estructural, hemos asociado un número a cada elemento introducido en cada una de ellas que, aunque éste no es el tiempo de realización, si nos indica el orden en el que se han llevado a cabo las acciones estructurales. Utilizaremos éstos valores numéricos para referirnos a elementos concretos.

Como se puede observar en la figura, existe un estímulo asociado a cada acción estructural llevada a cabo en el sistema.

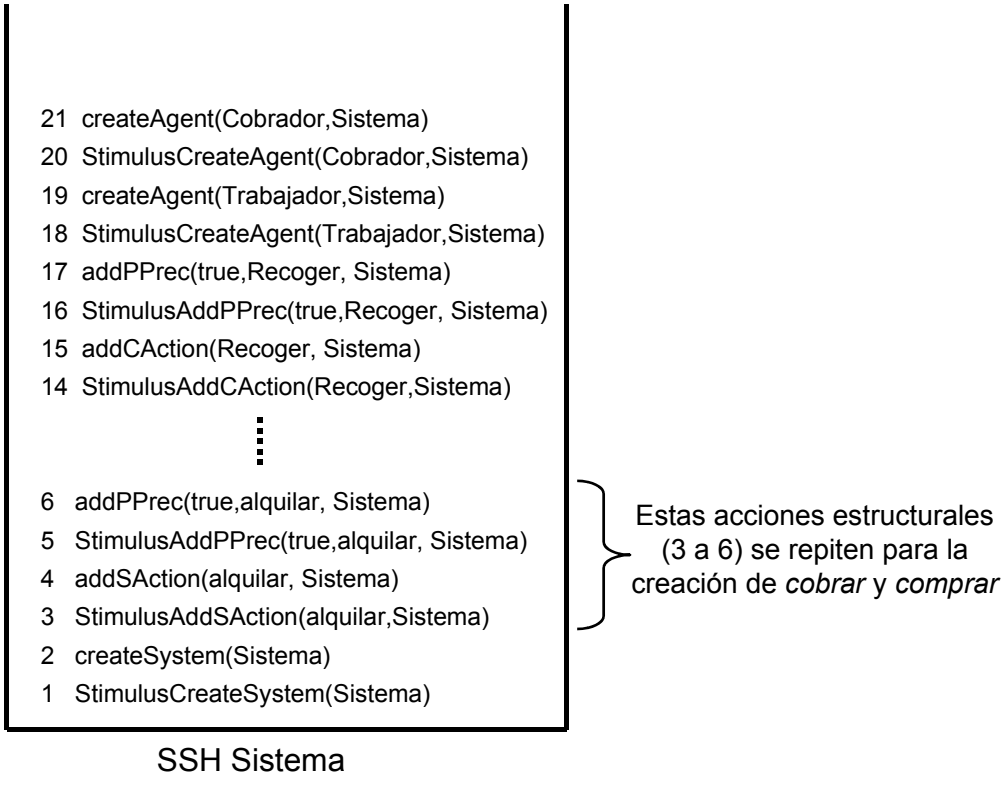


Figura 8.12 Historia estructural de Sistema

Comenzamos introduciendo un estímulo (1) para que se inicie la acción *createSystem,* una vez realizada (2), se crean cada una de las acciones que va a llevar a cabo nuestro sistema software. Primero la acción simple *alquilar* (3 – 6), después las acciones *cobrar* y *comprar* (como son simples, se repiten los pasos anteriores) y, por último, la acción compleja Recoger (14 – 17). Estas acciones representan el interfaz de este sistema con el exterior. Todas tienen su pre-condición a *true* y la definición de la acción *Recoger* sólo se compone de la acción especial *terminar*.

Los últimos elementos de la SSH son los relacionados con la creación de los dos subagentes *Trabajador* (18,19) y *Cobrador* (20,21).

En la siguiente figura podemos contemplar la SSH del agente *Trabajador*. Como se puede observar, las pre-condiciones *p* de sus acciones *alquilar* (3,4) y *Recoger* (7,8) no son *true* (como en el agente Sistema). Ambas son expresiones en lógica temporal de predicados y se corresponden con la descripción que hemos realizado anteriormente y que se puede consultar en la Tabla 8.1. Además, se ha cambiado la definición por defecto de la acción *Recoger* por una expresión en TDL (9,10). Cabe destacar, por último, que se ha realizado una acción estructural para crear un clon del agente *Empleado1*, al cual hemos

SSH Trabajador

Figura 8.13 Historia estructural del agente Trabajador

llamado *Empleado2* (15,16). Lógicamente, esta operación se ha realizado después de haber completado la definición del agente *Empleado1* y haber creado completamente todo su sub-árbol de agentes.

En la siguiente figura está representada la SSH del agente *Empleado1*. Se pueden observar los pasos seguidos en su creación, similares a los comentados para los dos agentes anteriores.

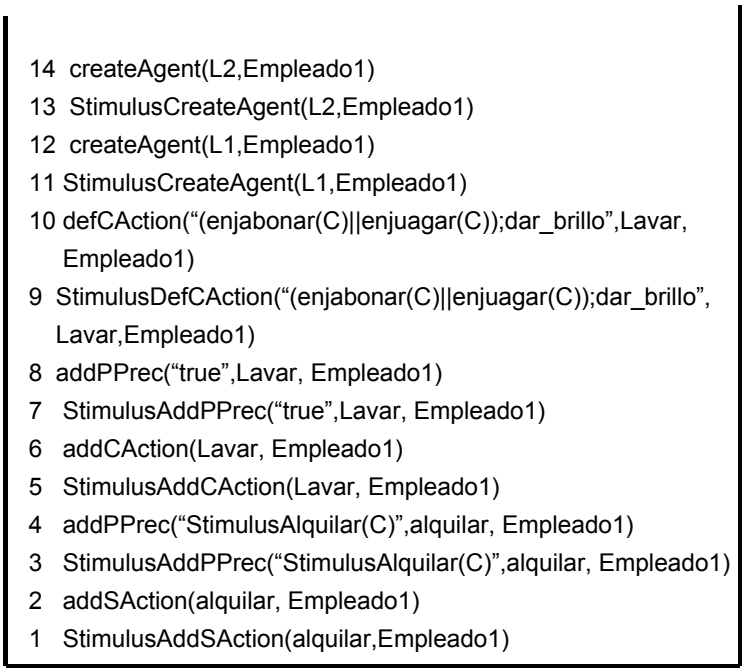


Figura 8.14 Historia estructural del agente Empleado1

8.7.2 Primer ejemplo de evolución: creación de un nuevo agente.

Vamos a comentar los pasos a seguir para crear un nuevo agente, *Empleado3,* que puede realizar las acciones *alquilar* y *pintar* y que será hijo del agente *Trabajador*. Partimos de la SSH de *Trabajador* presentada anteriormente.

Para realizar esta tarea se necesitan llevar a cabo varias acciones estructurales:

Paso 1: El desarrollador introduce a través de la interfaz de acción del Metasistema un estímulo, *StimulusCreateAgent(Empleado3, Trabajador)* para que se inicie la acción *createAgent* con los mismos atributos. No utilizamos la acción de clonación ya que no existe ningún agente cuya funcionalidad sea igual o esté incluida en la del nuevo agente que estamos creando. Se introduce dicho estímulo en la posición 17 de la SSH de *Trabajador* (ver la siguiente figura).

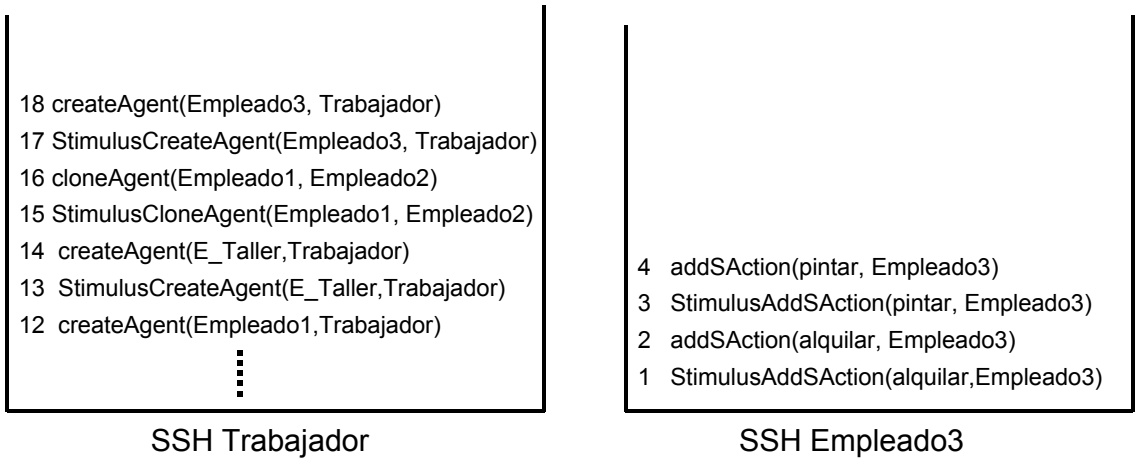


Figura 8.15 SSH de Trabajador y Empleado3 para el primer ejemplo

Paso 2: Al recibirse este estímulo, se activa la evaluación de la acción estructural *createAgent*. Recordemos que su pre-condición adaptada a este caso es:

createAgent(Empleado3,Trabajador) ← *exist(Trabajador) and*
not isAgrAgent(Empleado3,Trabajador) and
StimulusCreateAgent(Empleado3,Trabajador)

El Metasistema comprueba si existe el agente *Trabajador*. Después busca en la SSH de *Trabajador* (realmente lo hará en su MFH pero para no confundir, hablaremos de las SSH de los agentes) si existe alguna ocurrencia de la acción estructural *isAgrAgent* donde sus atributos coincidan con *Empleado3* y *Trabajador*. Si nos fijamos en la figura 8.13, no existe, por tanto se continúa con la evaluación, ya que tenemos un operador *and*. El último operando de esta expresión se comprueba mirando en la SSH si se ha recibido un *StimulusCreateAgent(Empleado3,Trabajador)* y, como se puede observar, existe (elemento 17).

Por tanto, la acción se realiza y se genera una la ocurrencia de acción que constituirá el elemento 18 en la SSH de *Trabajador*. Se crea un nuevo agente que, inicialmente, no tiene ninguna acción aunque, por efecto de ésta, se generó un nuevo estímulo, *StimulusAddSAction(...)*. Este estímulo se almacena en la SSH de *Empleado3* y, como resultado,

se añadirá a este agente una acción simple. La generación de este estímulo provocará una interacción del Metasistema con el desarrollador con el fin de que éste de valores a los atributos de dicho estímulo y se pueda realizar la acción *addSAction*.

Paso 3: Se activa la acción estructural *addSAction* para el agente *Empleado3* ya que se ha producido el estímulo necesario que constituye el primer elemento de su SSH. La pre-condición de esta acción es:

addSAction (alquilar, Empleado3) ← *exist(Empleado3) and*
not isInSSHA (alquilar, Empleado3) and
StimulusAddSAction(alquilar, Empleado3)

Vemos que la pre-condición se cumple, sin embargo, se comprueba que la acción *alquilar* la realiza ya un agente hermano y, por tanto, para que se cumpla la post-condición de esta acción en este caso, se realiza una copia de la acción. Recordemos que, por coherencia, no puede haber dos agentes que realicen la misma acción pero con funcionamiento diferente.

Por tanto, se genera una ocurrencia de la acción *addSAction* que se almacena en la SSH de *Empleado3.*

Paso 4: Se desea crear la acción pintar y, para ello, se introduce un estímulo en la SSH de *Empleado3*.

Paso 5: Se activa la acción *addSAction(pintar, Empleado3).* Al igual que en el caso anterior, su pre-condición es cierta pero existe un agente hermano, *E_Taller* que realiza dicha acción, por tanto, ambas deben ser idénticas o si no, la nueva acción debe tener un nombre distinto. Se crea (como copia de la existente) y se genera la ocurrencia de acción. En la figura anterior se puede contemplar el estado de las SSH de *Trabajador* y de *Empleado3* tras esta modificación del sistema software.

Lógicamente, tanto el *Controller 2* como el, nuevamente creado, *Controller 14* (del agente *Empleado 3*) tienen actualizadas sus tablas con la nueva información de tal forma que, por ejemplo, cuando se den las condiciones oportunas para poder alquilar un coche, cualquiera de los tres agentes puede llevarla a cabo.

8.7.3 Segundo ejemplo: agregación y eliminación de agentes.

En este ejemplo queremos ver cómo se resuelve la siguiente situación que modifica el sistema software. Llega una orden de despido del empleado *Empleado1.L2* que realiza la acción *enjuagar*. A partir de este

momento, el *Empleado2.L2,* que también realiza la acción *enjuagar,* va a trabajar tanto para *Empleado1* como para *Empleado2*. Comentaremos primero los problemas que nos encontramos.

El primer problema es que no podemos borrar el agente *L2* de *Empleado1* puesto que realiza una acción que se encuentra dentro de una transacción y, al ser el único agente de ese nivel que la puede realizar, la acción estructural *delAgent* no se llegará a completar aunque exista un estímulo para ella en la historia (su pre-condición no se cumple). Podemos pensar que, entonces, deberíamos agregar el agente *L2* de *Empleado2* a *Empleado1* pero nos encontramos un segundo problema: como ya existe un *L2* en *Empleado1* la acción *agrAgent* no se realizará ya que si no violaríamos el invariante de Nombres Unicos. El Metasistema informa al desarrollador si alguna acción estructural iniciada por éste falla. Sin embargo, es el desarrollador el que debe conocer el orden en el cual debe iniciar las acciones estructurales sobre el sistema software para conseguir que éste evolucione de acuerdo a sus necesidades.

Por tanto, para conseguir los cambios presentados anteriormente podemos seguir los siguientes pasos:

Paso 1: Vamos a iniciar un cambio de nombre del agente *L2* de *Empleado1* (15,16). Para ello se introduce un estímulo *StimulusRenameAgent*(*L2,Aux)* en la SSH de *Empleado1.*

Paso 2: Se activa la acción *renameAgent(L2,Aux)* cuya pre-condición es:

renameAgent(L2,Aux) ← *exist(L2) and (∀R isFather(R,L2) and not isAgrAgent(Aux,R)) and StimulusRenameAgent(L2,Aux)*

Como podemos observar en la SSH de *Empleado1* el agente *L2* existe (realmente el nombre del agente es *Sistema.Empleado1.L2* con lo que no se puede confundir con *L2* de *Empleado2*). También el estímulo necesario para llevar a cabo la acción (por el paso 1). Y, puesto, que no existen un agente *Aux* en *Empleado1*, la pre-condición se cumple.

Se actualiza la información del *Controller 4* y, puesto que no tiene ningún otro padre, se guarda la ocurrencia de esta acción en la SSH. Ya tenemos en *Empleado1* un agente llamado *Aux*.

Paso 3: El desarrollador introduce un estímulo en la SSH de *Empleado1, StimulusAgrAgent(Empleado2.L2, Empleado1),* para agregar *Empleado2.L2* a *Empleado1* (17,18).

Paso 4: Se activa la acción estructural a*grAgent(Empleado2.L2, Empleado1)* cuya pre-condición es:

agrAgent(Empleado2.L2,Empleado1) ← *not*
isAgrAgent(Empleado2.L2,Empleado1) and
(exist(Empleado2.L2) and exist(Empleado1) and
not isAgrAgent(Empleado1,Empleado2.L2)) and
(∀ R isAncestor(R,Empleado1) and
not isAgrAgent(R,Empleado2.L2)) and
StimulusAgrAgent(Empleado2.L2,Empleado1)

La pre-condición se cumple ya que:

- No existe un agente llamado *L2* en *Empleado1*.
- Tanto el agente a agregar como su futuro padre existen en el sistema software.
- No se producen ciclos ni cortos ni largos.
- Existe el estímulo asociado a esta acción.

Sin embargo, como existe otro agente en *Empleado1, Aux,* que realiza la misma acción, *enjuagar*, se comprueba si ambas son idénticas (post-condición de la acción estructural). Como lo son, la acción estructural se lleva a cabo.

Paso 5: El desarrollador introduce un estímulo (19) *StimulusDelAgent(Aux)* en la SSH de *Empleado1* (a través de la interfaz de acción).

Paso 6: Esta acción estructural (20) se puede realizar ya que ahora su pre-condición se cumple (existe otro agente que realiza su misma acción):

delAgent(Aux) ← *exist(Aux) and (∀ A isInSSHA(A,Aux) and*
∀ Q isFather(Q,Aux) (∃ R isAgrAgent(R,Q) and
A isInSSHA(A, R))) and StimulusDelAgent(Aux)

La pre-condición se cumple ya que:

- El agente *Aux* existe.
- El agente *Aux* realiza una acción, *enjuagar*, que se encuentra en la definición de la acción compleja *Lavar* de su único padre, pero no es un inconveniente ya que existe otro agente que la realiza (el que acabamos de agregar).
- El estímulo necesario se encuentra en la SSH (introducido en el paso 5).

Por tanto, la acción se lleva a cabo y se actualiza la información del *Controller 4* antes de que el sistema software continúe con su funcionamiento. La siguiente figura nos muestra el estado en el que ha quedado la SSH de *Empleado1* después de estas modificaciones.

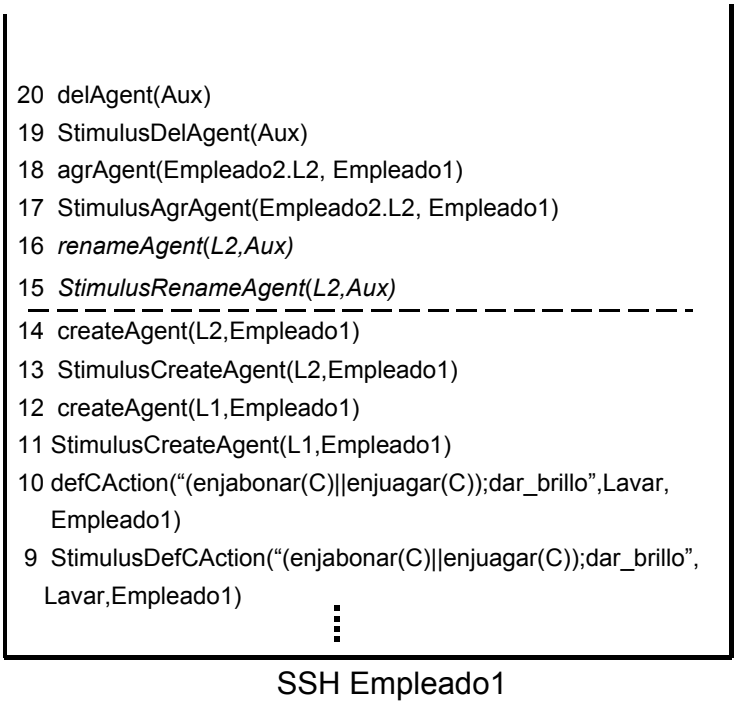


Figura 8.16 Historia estructural de Empleado1

8.7.4 Tercer ejemplo de evolución: modificación de una acción compleja.

Como último ejemplo ilustrativo de cómo se produce el proceso de evolución dentro del sistema software utilizando el modelo propuesto, vamos a cambiar la definición de una acción compleja. Planteamos dos variantes:

8.7.4.1 Modificación sencilla de la acción Lavar.

La acción *Lavar* de *Empleado1* tiene la siguiente definición:

Lavar(C) = (enjabonar(C) || enjuagar(C)) ; dar_brillo(C) ; terminar

Imaginemos que queremos cambiarla por:

Lavar(C) = enjabonar(C) ; enjuagar(C) ; terminar

Es decir, antes de enjuagar el coche queremos enjabonarlo entero y ya no se le sacará brillo a la carrocería. Para llevar a cabo esta modificación seguimos los siguientes pasos:

Paso 1: En la SSH de *Empleado1* se introduce un estímulo:

StimulusDefCAction("enjabonar; enjuagar", Lavar, Empleado1)

Paso 2: Se activa la acción estructural cuya pre-condición en este caso es:

defCAction("enjabonar;enjuagar",Lavar,Empleado1) ←
isInSSHA(Lavar,Empleado1) and ((∀ B isIn(B, "enjabonar;enjuagar") ∃ Q isAgrAgent(Q,Empleado1) and isInSSHA (B,Q)) and StimulusDefCAction("enjabonar;enjuagar", Lavar, Empleado1)

La pre-condición se cumple porque:

- La acción compleja que queremos modificar existe.
- Las acciones transaccionales que componen la nueva definición las realizan agentes hijos de *Empleado1*.
- El estímulo de realización de la acción se encuentra en la SSH.

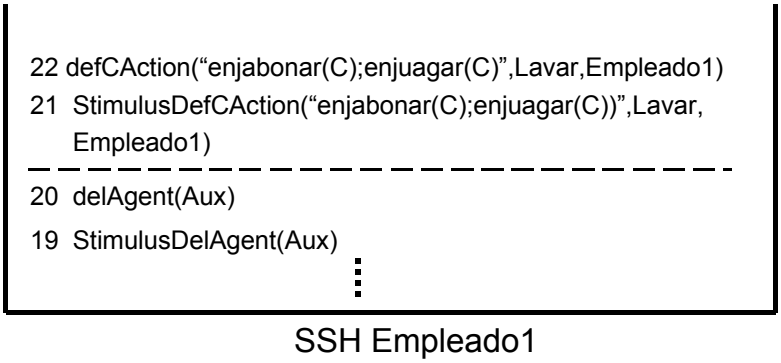


Figura 8.17 SSH de Empleado1 después de la modificación de Lavar

Por tanto, esta acción estructural se puede llevar a cabo pero vamos a comentar algunos aspectos que se valoran también (post-condiciones y acciones que se realizan). Se comprueba que existe un agente hermano, *Empleado2*, que también realiza la acción *Lavar*. El mecanismo de propagación de cambios realiza también la modificación de dicha acción ya que ambas acciones deben ser idénticas. Como acciones a realizar, se asocian las nuevas pre-condiciones *t* y *s* a cada acción transaccional. Así,

- La *t* de *enjuagar* y *enjabonar* es *true*.

- La *s* de *enjabonar* es *StimulusLavar* (igual que antes).
- La *s* de *enjuagar* es que exista una ocurrencia de *enjabonar*.
- Se concatena con la definición "*; teminar*", por lo que ahora la *s* de *terminar* es *enjuagar*.

8.7.4.2 Modificación de la acción Lavar sin éxito.

Partiendo del estado anterior de definición de *Lavar* de *Empleado1*:

Lavar(C) = enjabonar(C) ; enjuagar(C) ; terminar

Imaginemos que queremos cambiarla por:

Lavar(C) = enjabonar(C) ; enjuagar(C); libre(C) ; terminar

Además, queremos que se le ponga al coche un cartel de "Libre" con el fin de que los clientes puedan conocer qué coches están disponibles. Para llevar a cabo esta modificación seguimos los siguientes pasos:

Paso 1: Se introduce en la SSH de *Empleado1.L1* un estímulo (23) para añadir la acción simple *libre*:

StimulusAddSAction(libre, L1)

Recordemos que esto no afecta al agente *L1* de *Empleado2* porque ambos pueden evolucionar independientemente. Además *Empleado1.L1* y *Empleado2.L1* no son hermanos, luego no se hace ninguna comprobación.

Paso 2: Se activa la acción estructural cuya pre-condición en este caso es:

addSAction (libre, Empleado1.L1) ← *exist(Empleado1.L1) and*
not isInSSHA(libre,Empleado1.L1) and
StimulusAddSAction(libre(C),Empleado1.L1)

La pre-condición se cumple y la acción se realiza (24). Ahora el agente *L1* de *Empleado1* realiza una acción simple más.

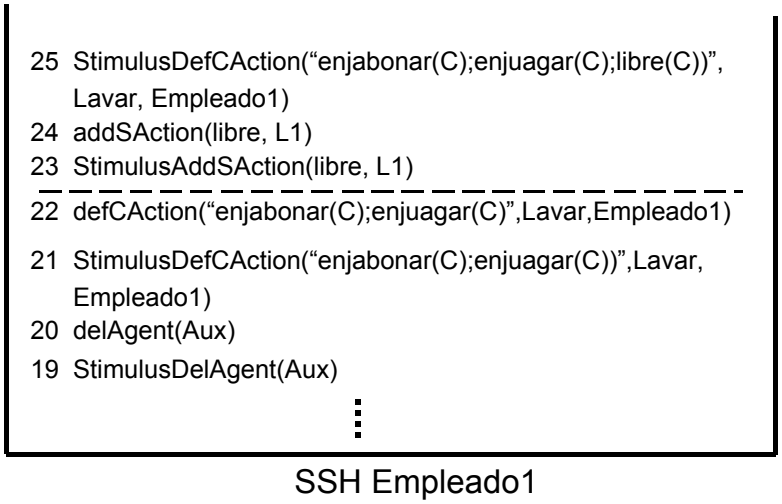


Figura 8.18 SSH de Empleado1 para la acción del punto 8.7.4.2

Paso 3: En la SSH de *Empleado1* se introduce el estímulo (25):

StimulusDefCAction("enjabonar(C); enjuagar(C); libre(C)", Lavar, Empleado1)

Paso 4: Se activa la acción estructural cuya pre-condición en este caso es:

defCAction("enjabonar;enjuagar;libre",Lavar,Empleado1) ←
isInSSHA(Lavar,Empleado1) and ((∀ B isIn(B, "enjabonar;enjuagar") ∃ Q isAgrAgent(Q,Empleado1) and isInSSHA (B,Q)) and StimulusDefCAction("enjabonar;enjuagar;libre", Lavar, Empleado1)

La pre-condición se cumple porque:

- La acción compleja que queremos modificar existe.
- Las acciones transaccionales que componen la nueva definición las realizan agentes hijos de *Empleado1*.
- El estímulo de realización de la acción se encuentra en la SSH.

El Metasistema guarda temporalmente la ocurrencia de la acción, *defCAction*, hasta que la post-condición de esta acción sea *true*. Pero la post-condición no puede cumplirse ya que existe un agente hermano, *Empleado2*, que realiza la misma acción compleja pero cuya modificación de su definición no se puede llevar a cabo porque ninguno de sus agentes realiza la acción *libre*. Por tanto, tampoco se llevaría a

cabo la modificación en el agente *Empleado1* ni se almacenará en la historia la ocurrencia de dicha acción. El desarrollador será informado de este problema que puede solucionar añadiendo la acción *libre* a alguno de los sub-agentes de *Empleado2* antes de volver a iniciar la acción *defCAction.* El Sistema Genético del agente *Empleado1* también será informado para que deshaga los cambios en la definición de *Lavar*.

En la siguiente figura mostramos cómo la nueva jerarquía de agentes que define al sistema que estamos modelando después de haber realizado las acciones de evolución de los puntos 8.7.2, 8.7.3 y 8.7.4. Como se puede observar, ya esta estructura tiene un nodo que la convierte en un grafo. Este nodo es el nodo etiquetado como *L2*.

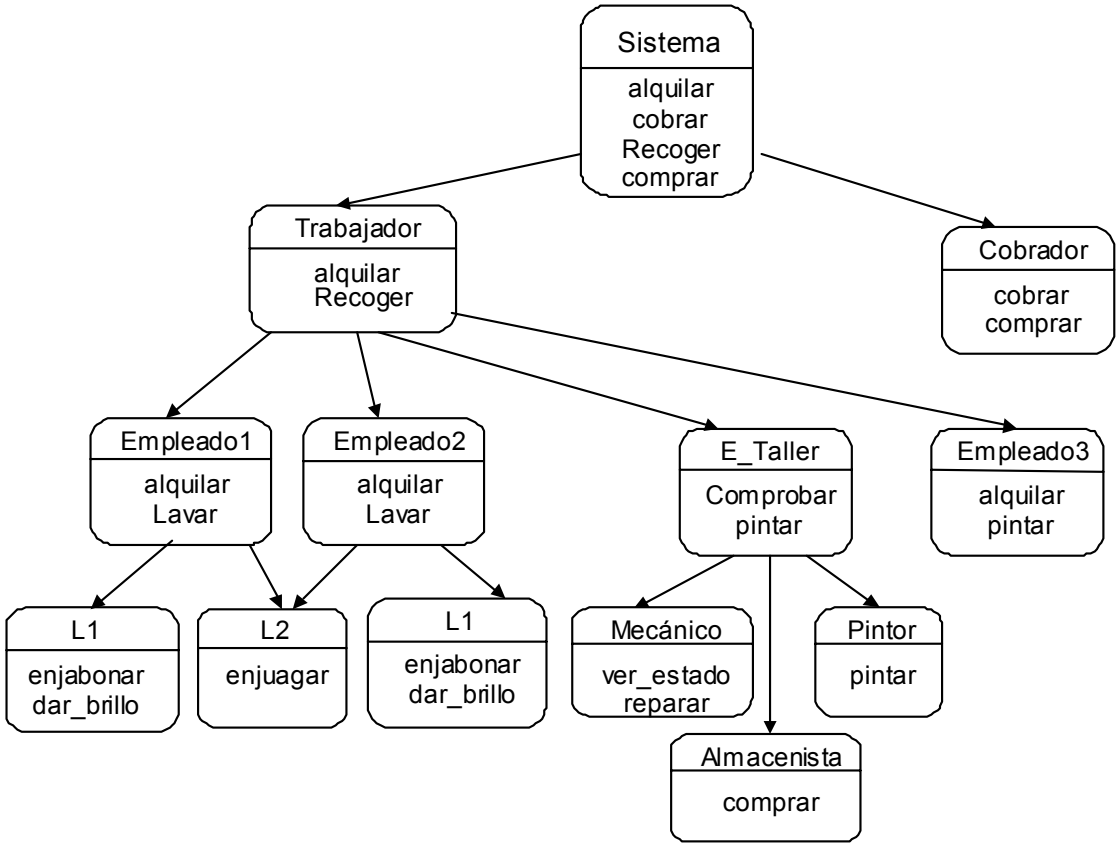


Figura 8.19 Jerarquía de agentes del sistema después de un proceso de evolución

8.8 Construcción de la RdPC del sistema.

Con el fin de comprobar cómo se lleva a cabo el proceso de creación de la RdPC que representa el funcionamiento del sistema, vamos a realizar paso a paso las dos fases de este proceso que se explicó en el capítulo 6.

Primero abordamos la fase de construcción de RdPC asociadas a cada nodo de la jerarquía de agentes y después pasamos a la fase de composición.

La fase de construcción empieza desde el nodo raíz, el nodo que hemos etiquetado con *Sistema*. Vamos a construir la RdPC asociada al sistema descrito en los primeros puntos de este capítulo, es decir, no tomamos en cuenta las acciones de evolución que hemos realizado posteriormente.

En la siguiente figura se muestra la RdPC asociada al nodo *Sistema*. En ella podemos observar que los lugares llamados *Recoger* y *alquilar* son lugares desplegables. El primero porque se corresponde con una acción compleja, el segundo porque está asociado a una acción simple delegada, es decir, una acción que será realizada por alguno de sus hijos.

Para construir esta red se utilizan las pre-condiciones *p* asociadas a las acciones de los agentes hijos de *Sistema*, *Trabajador* y *Cobrador* y gracias a ellas se conectan los distintos lugares y transiciones.

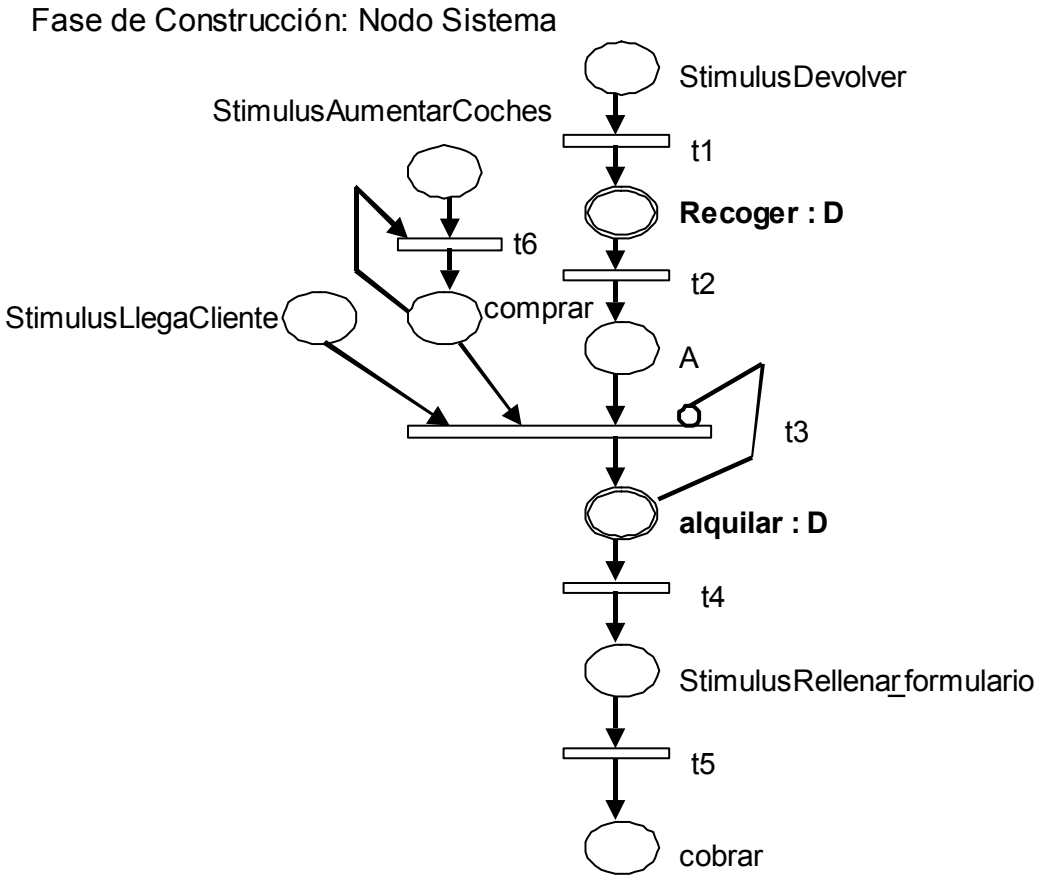


Figura 8.20 Fase de construcción: RdPC del nodo Sistema

Ahora continuamos con el agente *Trabajador*. Este agente tiene una sub-RdPC para su acción compleja *Recoger* y otra para la acción delegada *alquilar*. El agente *Cobrador* no tiene sub-RdPC ya que todas sus acciones son simples.

En la siguiente figura se muestran las sub-redes de Trabajador. Se puede observar que las transiciones *t8* y *t9* tienen asociada una guarda que se corresponde con las pre-condiciones *t* de las acciones *Lavar* y *pintar* respectivamente. Ambas pre-condiciones *t* son *(not StimulusFueraServicio)* que se ha abreviado por *(not StimulusFS)*.

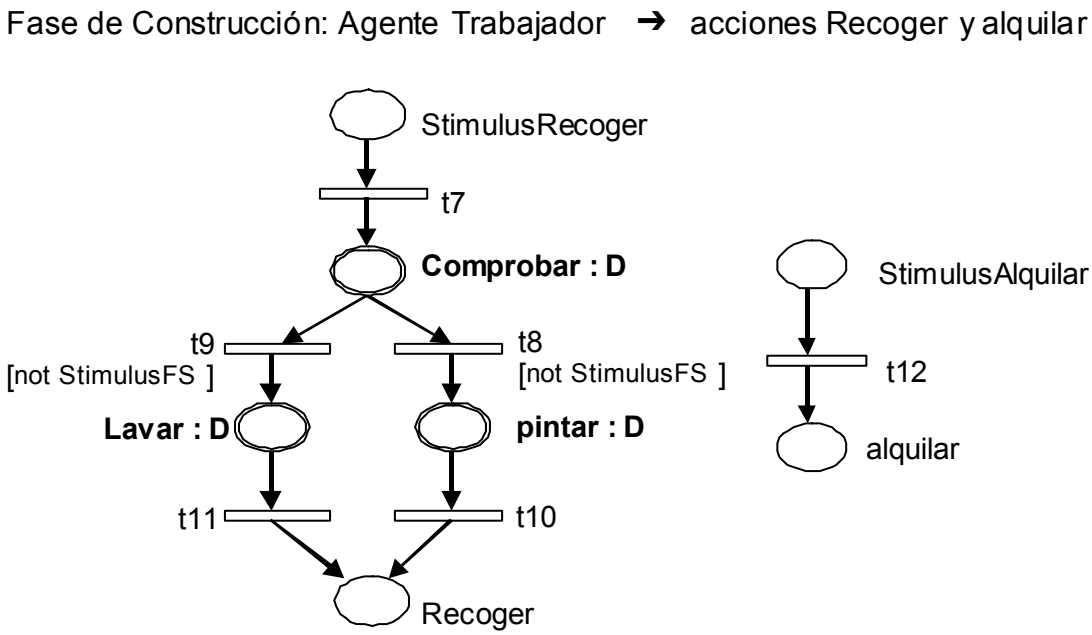


Figura 8.21 Fase de construcción-RdPC's de Trabajador

Ahora, bajamos de nivel en la jerarquía y construimos la sub-red de *Empleado1* y *Empleado2* (ya que son idénticos) para la acción compleja *Lavar* (ver siguiente figura).

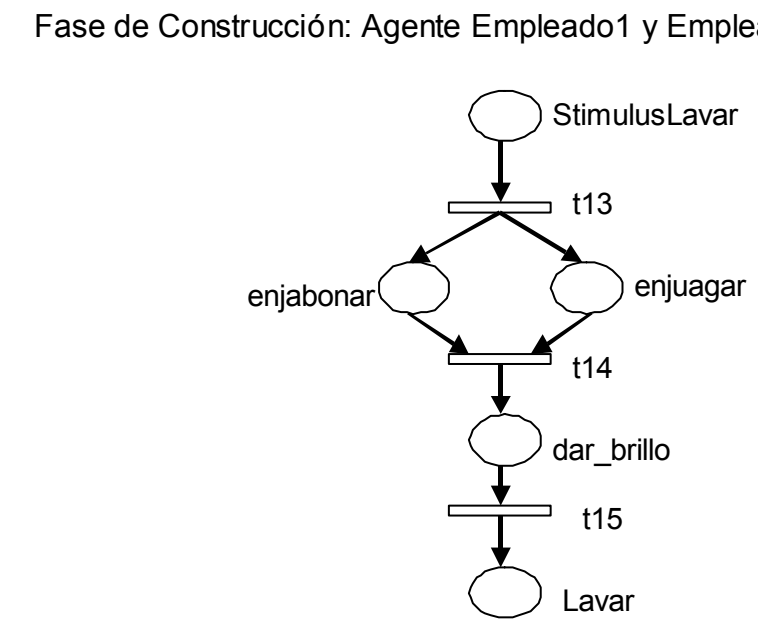


Figura 8.22 Fase de construcción: sub-RdPC de Empleado1 y Empleado2

También el agente *E_Taller* tiene asociadas dos sub-redes una para la acción compleja *Comprobar* y otra para la acción simple delegada *pintar* (ver la siguiente figura).

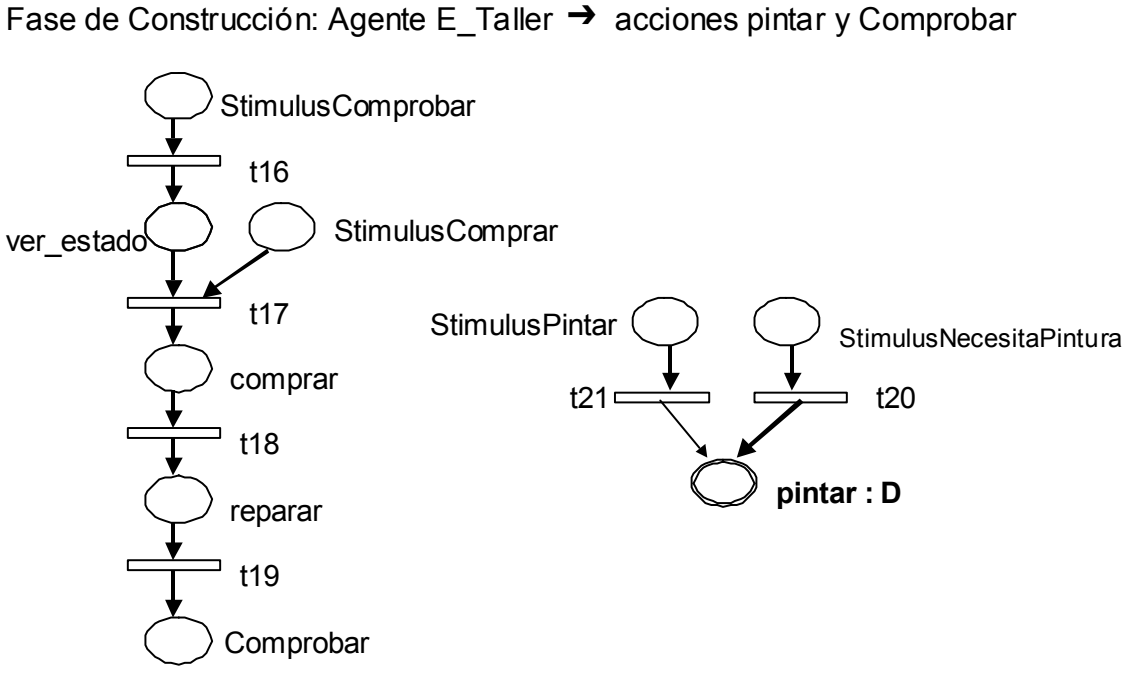


Figura 8.23 Fase de construcción : sub-redes de E_Taller

Por último, en la siguiente figura se representa la sub-red asociada al agente *Pintor* (ver figura siguiente). El resto de agentes hijos de *E_Taller* realizan sólo acciones simples. Con esto se termina la fase de construcción, ya hemos llegado a los nodos hojas de la jerarquía de agentes.

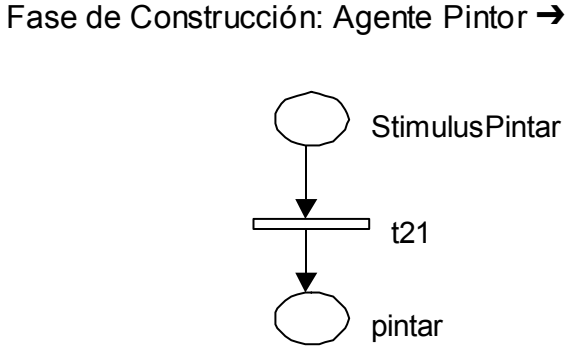


Figura 8.24 Fase de construcción: sub-red de Pintor

A continuación procedemos a realizar la fase de composición empezando a sustituir los lugares desplegables empezando desde las sub-RdPC de los agentes que se encuentran en los niveles más bajos de la jerarquía de agentes.

El primer agente que tiene una sub-RdPC con lugares desplegables es el agente *E_Taller*. En la siguiente figura podemos ver el resultado de sustituir el lugar desplegable *pintar* por la sub-red del agente *Pintor*.

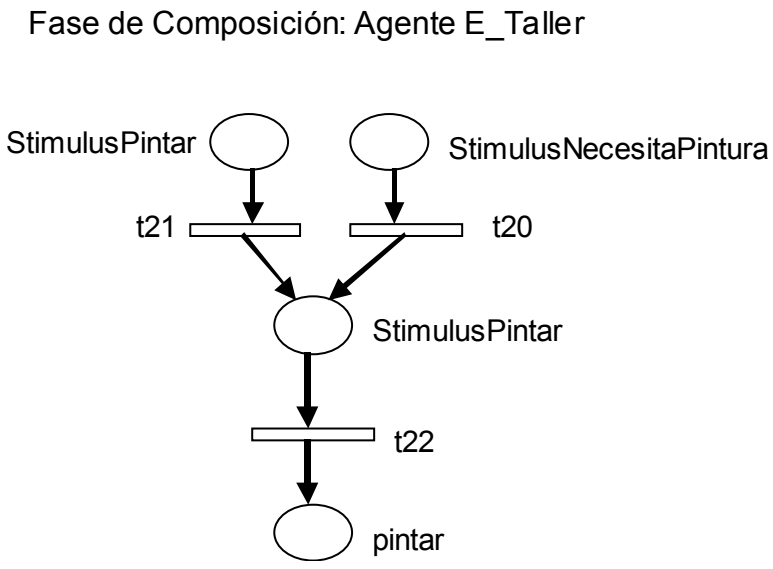


Figura 8.25 Fase de composición para E_Taller

La siguiente figura se corresponde con uno de las sub-redes de *Trabajador*. En este caso, como *Trabajador* tiene dos agentes hijos que realizan la acción *alquilar*, y como la ocurrencia de dicha acción es consumible se unen las dos sub-redes de *Empleado1* y de *Empleado2* de tal forma que el estímulo de alquilar sólo sirva para activar a uno de ellos (ver la siguiente figura).

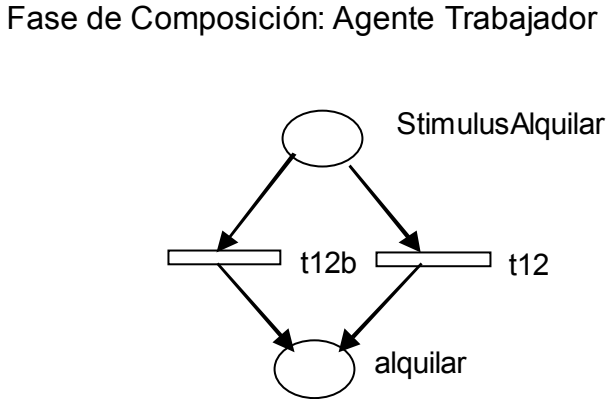


Figura 8.26 Fase de composición para la acción alquilar de Trabajador

En la figura siguiente se muestra la composición realizada en el caso de la sub-red asociada a la acción *Recoger* de trabajador. Igual que en el caso anterior, son dos los agentes que realizan la acción *Lavar* y, por tanto, sus sub-redes están duplicadas aunque sólo a una de ellas les llegará el estímulo que inicia la acción.

Fase de Composición: Agente Trabajador

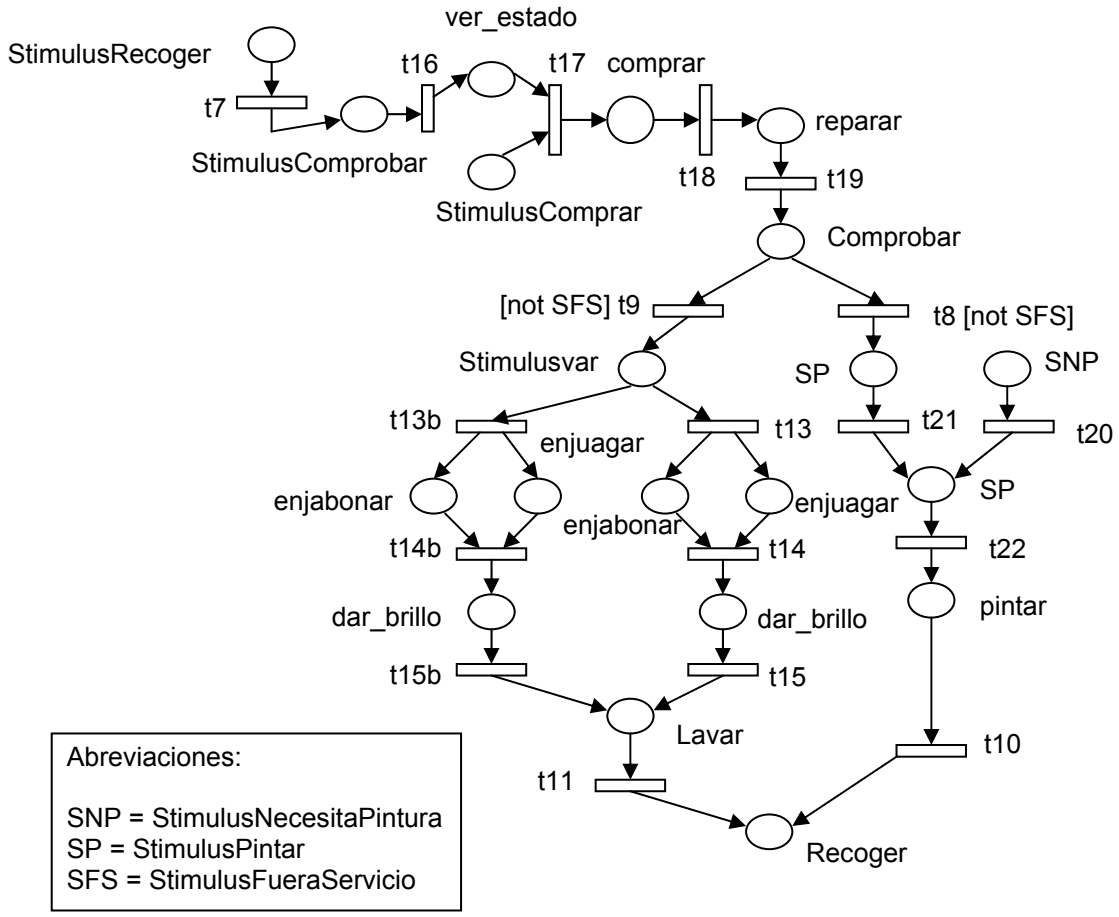


Figura 8.27 Fase de composición para la acción Recoger de Trabajador

A continuación, y finalizando con la fase de composición se sustituyen cada uno de los lugares desplegables de la RdPC del sistema por las sub-redes correspondientes que se han completado en los pasos anteriores. en la figura siguiente se puede observar la RdPC completa que representa al sistema de alquileres de coches.

Fase de Composición: Sistema

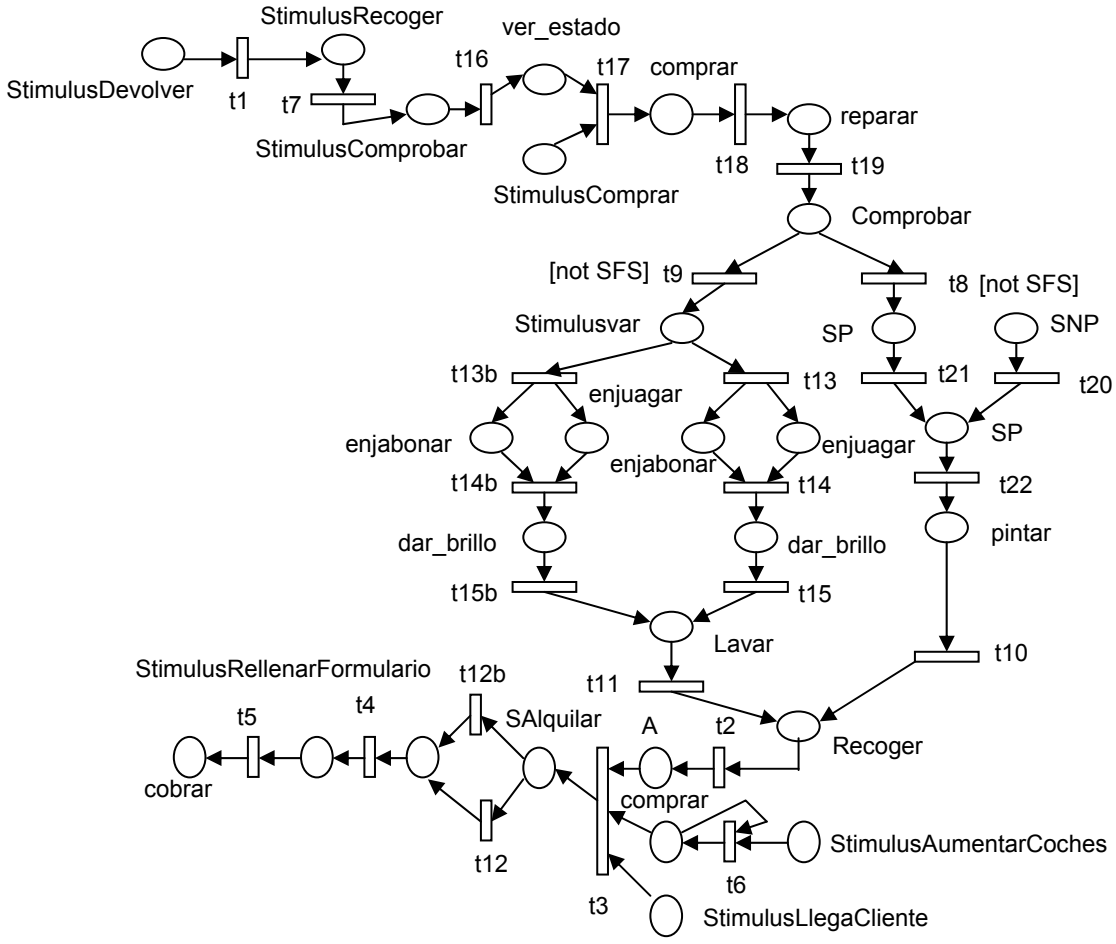


Figura 8.28 Fin de la fase de composición: RdPC del Sistema

8.9 Conclusiones.

En este capítulo se ha presentado un ejemplo de un sistema real: una empresa de alquileres de coches. Se ha realizado primero una descripción del problema y después, se ha especificado cada uno de los agentes que lo componen, sus acciones y las pre-condiciones de éstas, y los estímulos.

Hemos descrito la información que mantiene el agente *Controller* de cada uno de los agentes identificados.

A continuación, hemos descrito dos ejemplos significativos de funcionamiento del sistema software desarrollado: el inicio de una acción simple y el de una acción compleja, y tres ejemplos de evolución: creación de un nuevo agente, agregación y eliminación de agentes y modificación de una acción compleja (con y sin éxito).

Finalmente, se ha realizado el proceso de creación de la RdPC del sistema creando las sub-RdPCs de cada uno de los agentes. Esta es una de las optimizaciones que se realizan sobre el modelo y que comentamos en el capítulo 6.

En general, se ha puesto de manifiesto que los conceptos vistos en los capítulos anteriores y que forman la base del modelo propuesto son válidos.

CAPITULO 9

Prototipo

En este capítulo vamos a centrarnos en el prototipo que hemos desarrollado para crear sistemas software basados en agentes y permitir su evolución.

Presentamos también las interfaces, capturadas del prototipo, que permiten observar, de manera general, cómo funciona.

Mostramos el diseño de clases que se ha usado para la construcción de la herramienta. En este diseño están presentes todos los conceptos que hemos descrito en el modelo presentado, tanto la arquitectura del sistema, como los agentes necesarios para hacerlo funcionar y evolucionar.

9.1 Prototipo de la herramienta.

Como hemos indicado en otros capítulos partimos de un trabajo ya realizado dentro de nuestro grupo de investigación (GEDES), del modelo MEDES y su herramienta asociada HEDES [Parets98] [Parets99a] [Rodríguez99] [Paderewsk99a] [Paderewki00] [Rodríguez00a] [Rodríguez00b]. HEDES, y sus distintas versiones, fueron implementadas en *Smalltalk* y nos han servido para comprobar los conceptos fundamentales de los que partimos. Sin embargo, en HEDES, no existía una jerarquía de agentes (procesadores en dicho modelo) y tampoco se contemplaban las acciones complejas o transacciones. Esto ha provocado un cambio en la herramienta que nos ha decidido a realizar un nuevo prototipo con el fin de incorporar y probar los nuevos conceptos que hemos mencionado.

Para describir la jerarquía de agentes de un sistema software y las acciones complejas o transacciones, hemos utilizado una herramienta gráfica desarrollada dentro de nuestro grupo de investigación [Molina02]. Esta herramienta es un editor de estructuras conceptuales evolutivas que se implementó inicialmente como prototipo del modelo hipermedia SEM-HP [García01] aunque se ha usado para la edición de ontologías especiales utilizadas para integrar la información procedente de los Sistemas de Información y los Sistemas de Ayuda a la decisión [Hurtado02]. Gracias a su diseño, bastante versátil, actualmente estamos utilizando esta herramienta para realizar la descripción de los Sistemas Software basados en Agentes, aunque para ello lo hemos adaptado con el fin de que soporte las características propias que necesitamos. Este editor, llamado *CS Editor*, se ha implementado en *Java*, debido a su carácter multiplataforma, a su integración con la *Web* y a la disponibilidad de múltiples librerías.

CS Editor permite representar una estructura conceptual (EC), es decir, una red semántica con dos tipos de nodos: los conceptos y los itemes y hacerla evolucionar, gracias a un conjunto de acciones evolutivas. Por tanto, se permite editar la EC, añadir o borrar conceptos e itemes, así como crear o eliminar las relaciones entre ellos. Es una herramienta muy flexible, está diseñada e implementada de tal forma que se usan atributos genéricos para describir tanto a los conceptos como a los itemes. Así, se permite que los conceptos y los itemes tengan un número y tipo de atributos variable. Proporciona mecanismos para definir los atributos que tiene cada tipo de nodo y las propiedades de cada uno.

Para nosotros, los conceptos son de dos tipos: los agentes y las acciones complejas que pueden realizar éstos. Los itemes representan acciones simples y acciones transaccionales. Se diferencia cuándo un concepto es un agente o una transacción por el tipo de relación que la une a otros nodos de la red. Si un concepto tiene relaciones de agregación (*agregation*) con otros, significa que es un agente, ya que esta relación determina la relación padre-hijo entre agentes. Así mismo, si la relación entre conceptos es de realización (*performs*) significa que el concepto origen de la relación es un agente y el concepto destino es una acción compleja. A las relaciones que ligan conceptos con itemes no les hemos dado nombre ya que si un item está ligado a un concepto que es un agente significa que dicho agente realiza la acción simple que representa. Si, por el contrario, está relacionada con un concepto que es una acción compleja significa que es una acción transaccional de ésta. No existen relaciones entre itemes. Las relaciones comentadas se definen respectivamente mediante tripletas de la siguiente forma:

<agente-padre, agregation, agente-hijo>
<agente, performs, acción-compleja>
<agente, , acción-simple>
<acción-compleja, , acción-simple>

En la siguiente tabla se resumen las equivalencias que se acaban de comentar entre los elementos de una estructura conceptual y los elementos de un sistema software basado en agentes.

Estructura Conceptual	Sistema Software basado en Agentes
Concepto	agente
	transacción
Item	acción simple
	acción transaccional
Relaciones entre conceptos	agregation : agente padre → agente hijo
	performs: agente → transacción
Relaciones entre conceptos e itemes	(concepto = agente) → (item = acción simple)
	(concepto = transacción) → (item = acción transaccional)

Tabla 9.1 Equivalencia entre los elementos de una EC y los elementos de un sistema software basado en Agentes

Uno de los objetivos es que la EC y, por tanto, la arquitectura que representa, sea consistente. El editor controla algunos invariantes, como el de unicidad de nombres para los conceptos e ítemes, asociando restricciones o pre-condiciones a las acciones evolutivas. Aparte de éstos, se han incorporado otros invariantes propios de los sistemas software basados en agentes. Estos invariantes están relacionados con la definición de agentes y las relaciones entre ellos, las acciones

complejas y las acciones simples con los atributos necesarios. Por ejemplo, se comprueba que no se intente añadir una acción compleja sin su definición o un agente sin nombre.

En las figuras siguientes podemos ver la descripción del sistema de alquileres de coches que hemos utilizado en el capítulo 8.

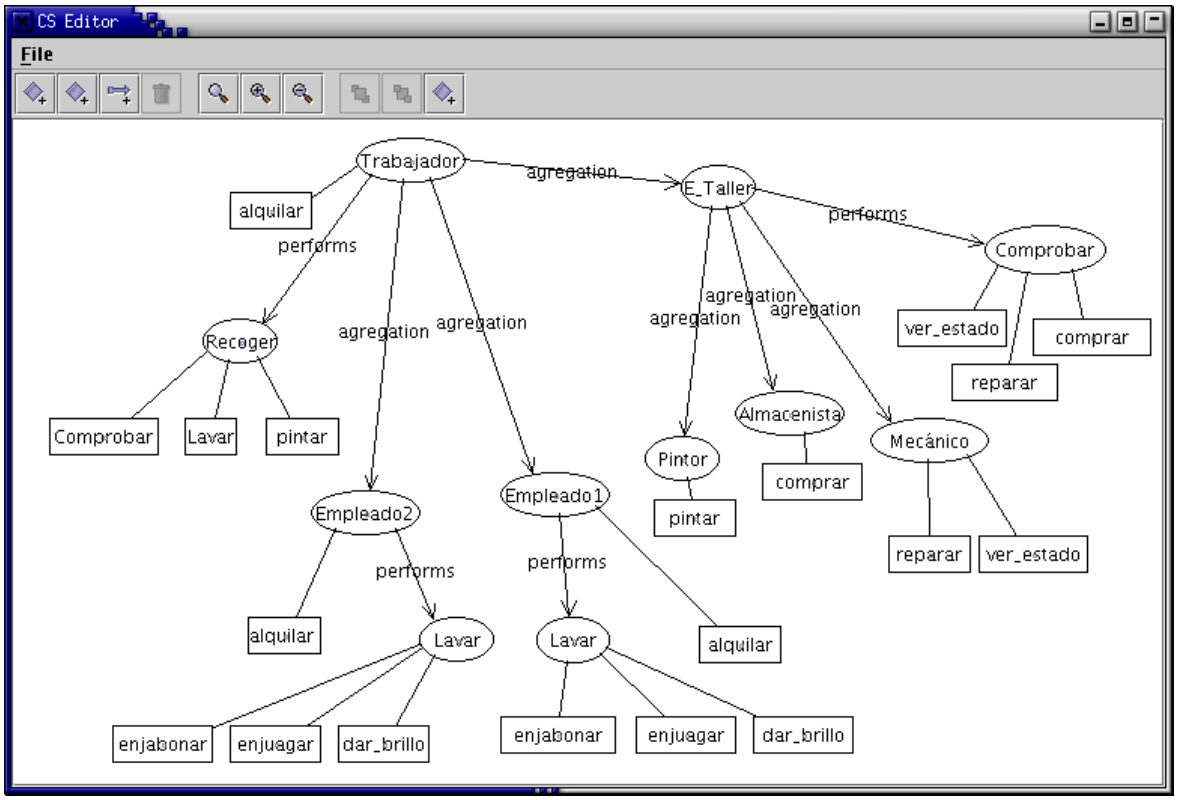


Figura 9.1 Pantalla de la herramienta CS Editor utilizada

En el grafo de la figura anterior pueden observarse elipses que representan los dos tipos de conceptos utilizados: agentes y acciones complejas. Los rectángulos corresponden a acciones simples o a acciones transaccionales. También vemos los dos tipos de relaciones: de agregación (agregation) y de realización (performs) de acciones. Por último, se puede observar cómo las relaciones entre acción compleja y acciones transaccionales y entre un agente y sus acciones simples no tienen nombre.

Cada concepto o item tiene definido un conjunto de atributos. Para la definición de un agente se pide el nombre de éste que debe ser distinto al de otro agente hermano. Se permite que existan nodos agentes con igual nombre siempre que estén en distinto nivel de la jerarquía. Para una acción compleja se solicita el nombre, su pre-condición *p* y la definición en TDL de dicha acción compleja, como se puede ver en la siguiente figura. La definición de acción simple tiene asociada el nombre

y su pre-condición *p*. En caso de definir una acción transaccional se pide el nombre y su pre-condición *t*. Recordemos que la misma acción simple puede formar parte de más de una acción compleja, por tanto nos interesa recoger el hecho de que la pre-condición *t* de una acción transaccional debe estar ligada a la transacción.

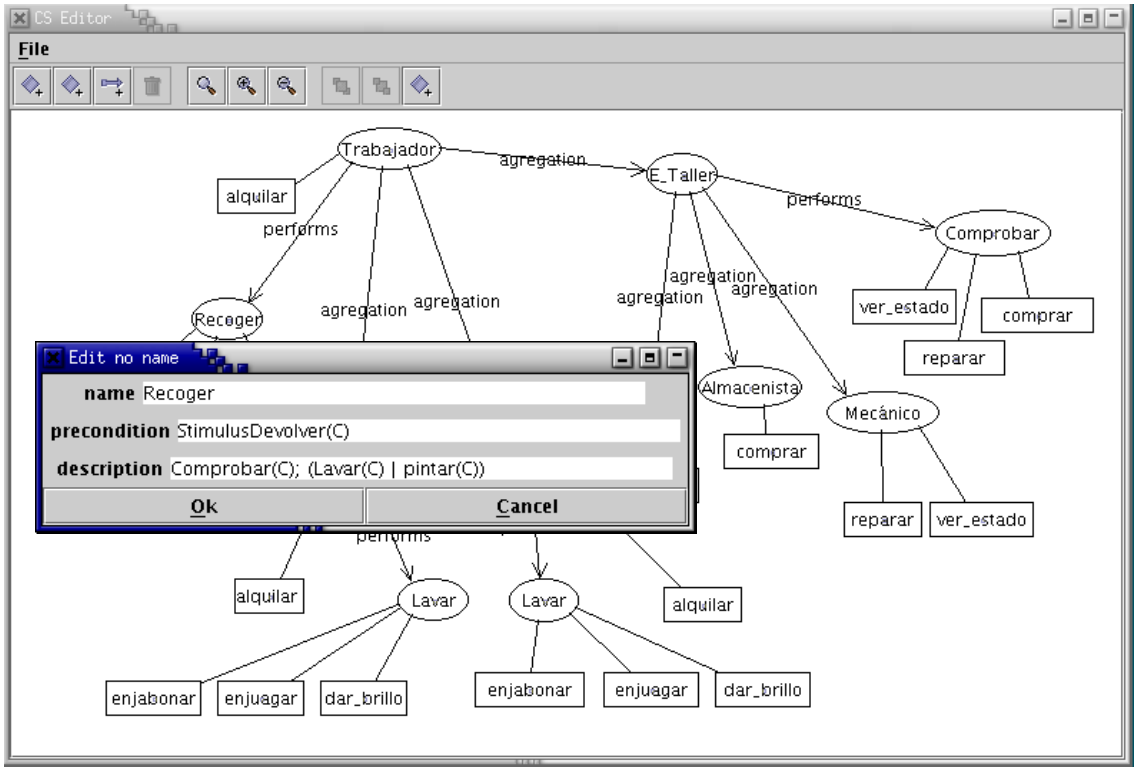


Figura 9.2 Creación de la acción compleja Recoger del agente Trabajador

9.2 Evolución en el prototipo.

En el diseño del prototipo se ha utilizado el patrón de diseño MVC (Modelo-Vista-Controlador) [Krasner88]. Nosotros hemos propuesto que el Metasistema modifique al sistema, es decir, el Metasistema realiza los cambios sobre el modelo. El desarrollador interacciona con el controlador, que se utiliza como interfaz de acción del Metasistema para indicar qué acciones evolutivas se desean hacer. El Metasistema realiza cambios sobre el sistema si se verifican los invariantes prefijados. Si los cambios pueden llevarse a cabo, las vistas recogen la nueva versión y se le muestra al desarrollador para que haga pruebas sobre ella y así pueda validar y verificar el sistema. Todo este proceso se es un ciclo iterativo. En concreto, en la herramienta la vista es la EC, el controlador

permite editar esta EC y hacerla evolucionar y el modelo es el sistema representado.

En la figura siguiente se puede ver el patrón de diseño utilizado.

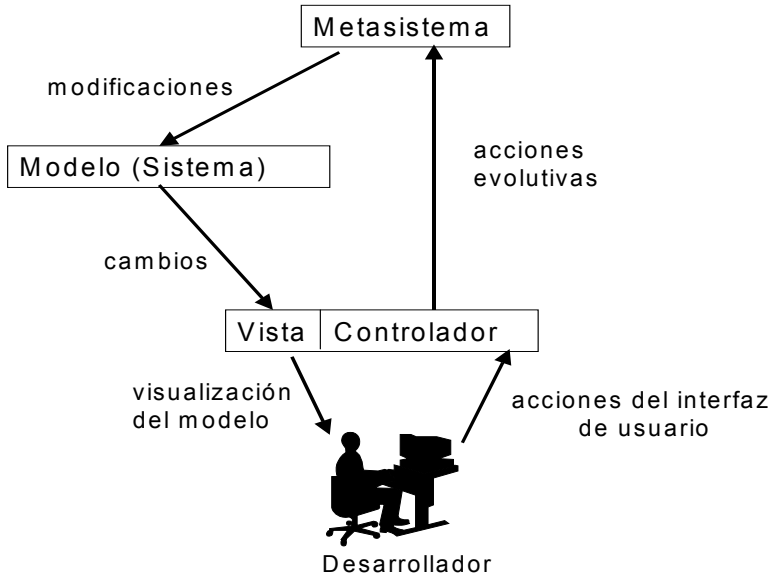


Figura 9.3 Modelo-Vista-Controlador Evolutivo

Se han descrito las acciones evolutivas que realiza el Metasistema de acuerdo al modelo presentado. En las siguientes figuras se puede observar que si el desarrollador, usuario de esta herramienta, intenta realizar una acción que modifica el sistema existente y que viola alguno de los invariantes que definimos para los sistemas software basados en agentes, se producirá su correspondiente mensaje de error y la acción no se llevará a cabo. Si la acción evolutiva requiere una propagación del cambio realizado, también se llevará a cabo.

En la siguiente figura se muestra la situación en la que se intenta realizar la acción evolutiva *createAgent* para crear un nuevo agente llamado *Empleado2* que será hijo del agente *Trabajador*. Esta acción de evolución origina un error ya que se viola el invariante de *Nombres Unicos* porque ya existe un agente hermano con igual nombre. Es decir, se va a introducir la relación:

<Trabajador, agregation, Empleado2>

Para comprobar el invariante de nombres únicos, la herramienta busca todas aquellas relaciones de agregación que tienen como concepto origen “Trabajador” y compara si en alguna de ellas el concepto destino coincide con “Empleado2”. Si encuentra algún triple con estas características, como es el caso de nuestro ejemplo, devuelve un error

indicando que la operación no puede realizarse, ya que existe un agente con dicho nombre que es hijo del agente *Trabajador*.

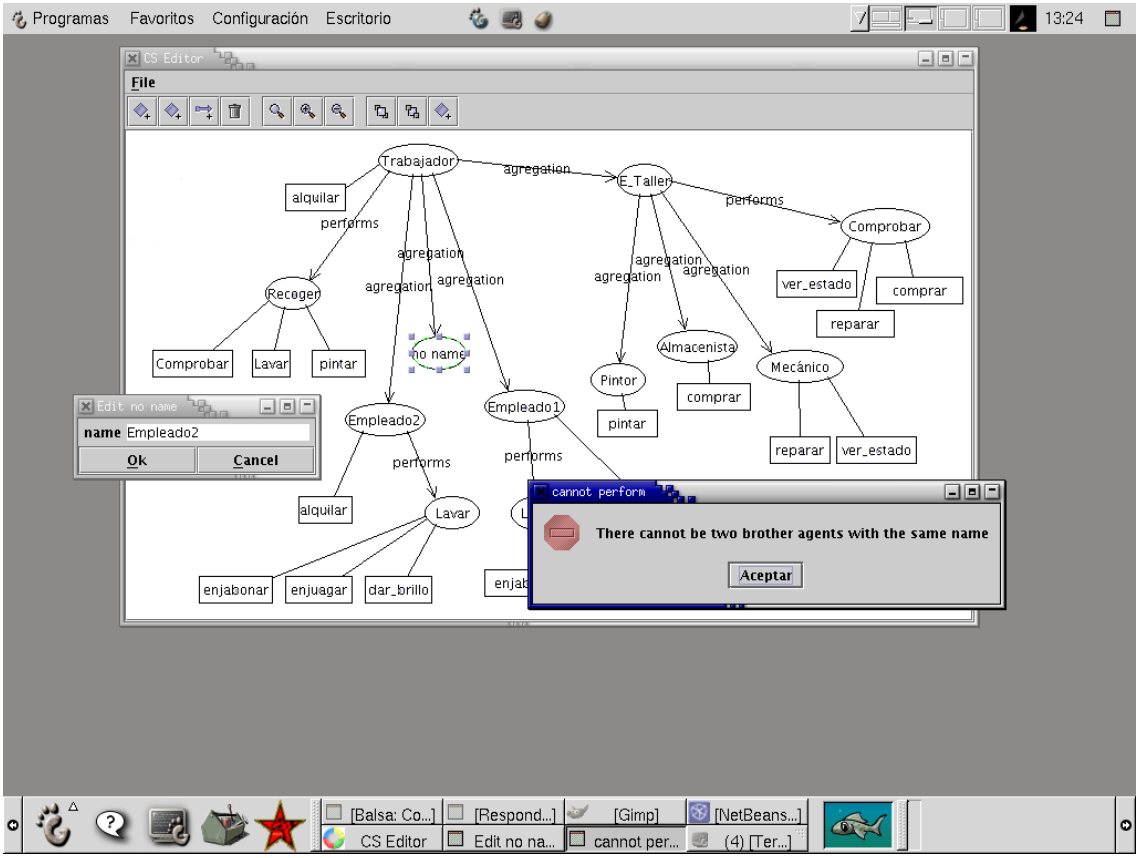


Figura 9.4 Error al intentar crear un agente, Empleado2, con igual nombre que un agente hermano ya existente

La figura siguiente muestra el caso en el que se intenta definir una acción compleja, *Recoger*, pero previamente no se han definido todas las acciones simples de los agentes hijos de *Trabajador* que aparecen en la descripción de dicha acción compleja. Por tanto, el intento de realizar esta acción evolutiva origina un error tal y como describimos en el capítulo 7 para la acción estructural *addCAction*. Se viola el invariante de *Referencia*. Las relaciones que se añadirían en este caso son:

<Trabajador, performs, Recoger>
<Recoger, , Comprobar>
<Recoger, , Lavar>
<Recoger, , pintar>

Para permitir esta acción evolutiva, la herramienta debe comprobar que existe algún agente hijo que realice cada una de las acciones transaccionales. Para ello busca todas las relaciones de agregación donde el concepto origen sea el agente Trabajador y para cada una de

ellas, busca si el concepto destino, un agente hijo, tiene definida alguna relación donde el item coincida con alguna de las acciones transaccionales. Si al finalizar esta búsqueda alguna de las acciones transaccionales no es realizada por algún agente hijo, la operación de añadir una acción compleja devuelve un error.

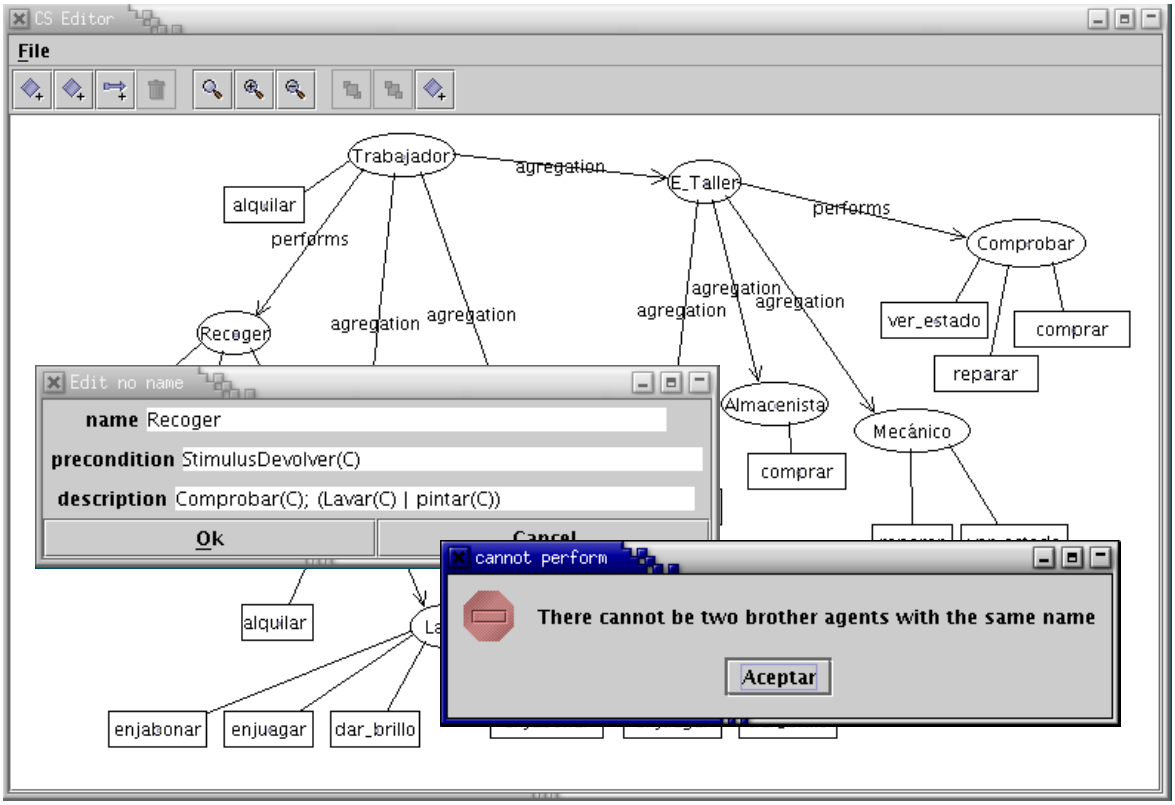


Figura 9.5 Intento de definir la acción compleja Recoger del agente Trabajador

9.3 AgentHEDES.

El prototipo descrito sólo nos permite mostrar la parte de evolución y el comportamiento del Metasistema como elemento que dirige y comprueba los cambios introducidos por el desarrollador en el sistema. Sin embargo, no trata los aspectos de funcionamiento del sistema software desarrollado. Uno de nuestros objetivos futuros es completar este prototipo, creando una herramienta que llamamos *AgentHEDES*.

En la siguiente figura mostramos el diseño completo del AgentHEDES utilizando UML [Booch99] y la herramienta *Rational Rose* [Rational].

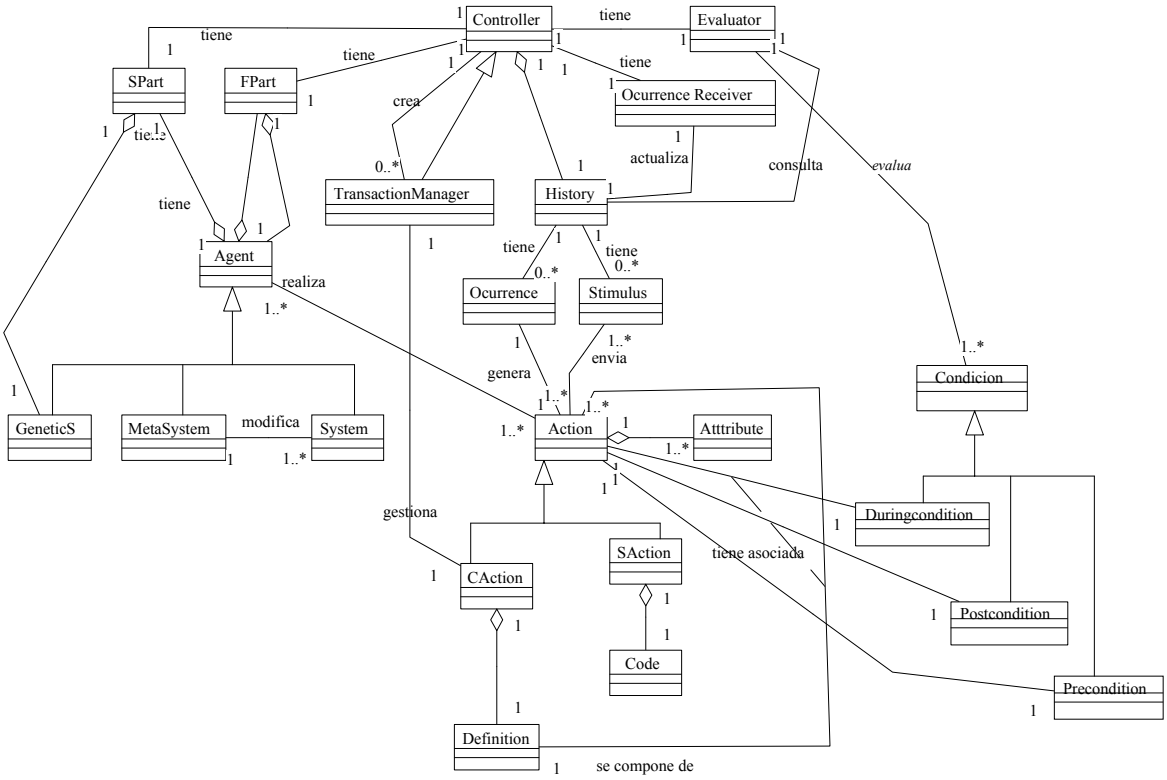


Figura 9.6 Diseño de clases para AgentHEDES

En la anterior figura se puede observar que:

- Un agente se compone de dos partes, una parte estructural (*SPart*) y otra funcional (*FPart*) que son equivalentes en el sentido de que se componen de los mismos elementos. Lo que las diferencia es el uso que se hace de ellas, la parte funcional se utiliza para el funcionamiento del agente y la estructural para su evolución.
- La parte estructural posee un único agente que es el Sistema Genético (*GeneticS*).
- La parte funcional puede componerse de uno o más agentes, que son los agentes que llamamos hijos.
- Aunque de la estructura de clases no se desprende (por no complicar en exceso el diagrama de clases), es interesante comentar que el Metasistema (*Metasystem*) es un agente cuya parte estructural está vacía y que el Sistema (*System*) es un agente que no tiene padre.
- Cada parte, funcional o estructural, tiene asociado un *Controller* que gestiona su historia y controla la actividad del agente.
- Cada *Controller* tiene una historia (*History*), un *Evaluator* que evalúa condiciones (*Condition*), y un *Ocurrence Receiver*. En la historia se almacenan ocurrencias de acciones (*Ocurrence*) y estímulos

(*Stimulus*). Las condiciones pueden ser de tres tipos, pre, post y durante-condiciones.

- La clase *Action* tiene una relación de especialización con las clases *CAction*, que corresponde a una acción compleja, y la clase *SAction* que representa a una acción simple. Ambas tienen atributos distintos, por ejemplo toda acción compleja tiene asociada una definición que no aparece en las acciones simples. Sin embargo, las acciones simples tienen asociado un código que no necesitan las acciones complejas.
- Toda acción funcional tiene asociada una pre-condición (*Precondition*). Si la acción además se encuentra en la definición de una transacción, también tiene asociada una pre-condición *t* (*Duringcondition*). Normalmente la postcondición (*Postcondition*) de una acción funcional es *true*.
- Las acciones estructurales tienen asociadas una pre-condición y una post-condición.
- Cada acción genera una ocurrencia de acción (*Ocurrence*) y debido a su ejecución puede enviar un estímulo (*Stimulus*).
- Para gestionar las acciones complejas tenemos un *Transaction Manager* que es una especialización de *Controller.* Recordemos que cada *Transaction Manager* es creado por un *Controller* cuando se inicia una transacción.

Como hemos indicado al principio de este capítulo, HEDES es una herramienta diseñada dentro de nuestro grupo de investigación que implementa parte de los conceptos de funcionamiento y de evolución que hemos descrito en nuestro modelo. Está implementado en *Smalltalk* y nuestra intención es reutilizar un subconjunto de las clases ya creadas en el último prototipo e implementar las clases restantes que necesitamos para incorporar los nuevos conceptos del modelo propuesto en esta tesis. De hecho, HEDES no contempla ni las transacciones ni la estructura jerárquica de los agentes, sin embargo nos ayudó a poner en práctica los conceptos introducidos por el modelo MEDES. Con HEDES se podía crear un sistema, ponerlo en funcionamiento y hacerlo evolucionar.

De HEDES podemos reutilizar la clase History, la clase Evaluator aunque con alguna modificación, la clase Processor como parte de la implementación de nuestra clase Agent y las clases GeneticS, MetaSystem y System. Concretamente, redefinimos la clase Action para incorporar la diferencia entre las acciones simples (SAction) y las acciones complejas (CAction) junto con las clases necesarias que definirán su componsición (sus atributos). Añadimos la clase Controller y la clase Transaction Manager. Introducimos las clases Ocurrence y Stimulus.

En la figura siguiente podemos ver la interfaz de este prototipo:

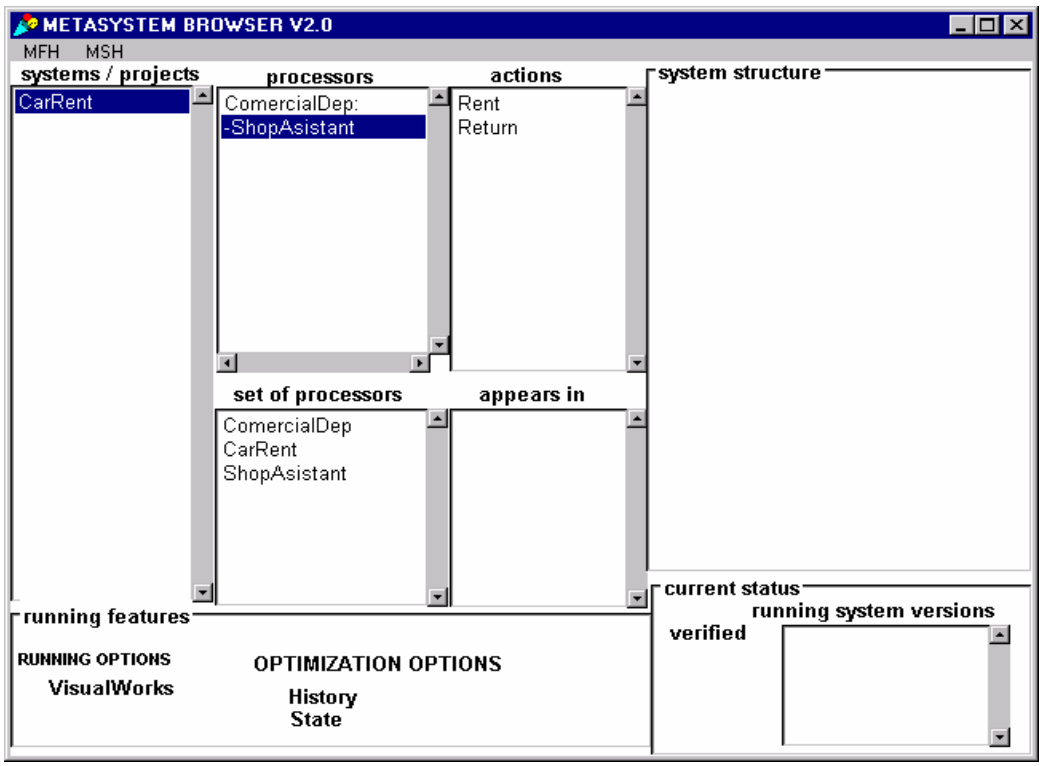


Figura 9.7 Prototipo HEDES

Una vez implementadas cada una de las clases, se realizaría un traductor que nos permita trasladar el diseño realizado con *CS Editor* del sistema software a los elementos especificados en el modelo. Es decir, por cada concepto que sea un agente se crea un objeto de la clase *Agent* con los componentes especificados de su estructura: un *Controller* con su historia y los agentes *Evaluator* y *OcuR*, un conjunto de acciones y un conjunto de agentes hijos (si los tiene). Las tablas que hemos definido en el tema 6 sobre el funcionamiento de los agentes se actualizan con la información necesaria. Se permitirá que un sistema empiece a funcionar gracias a la introducción de uno o más estímulos externos que introducirá el usuario a través de una interfaz gráfica que facilite su tarea. Se parará el sistema si el desarrollador debe realizar alguna modificación en el sistema pero después de esto, el sistema continuará funcionando, eso sí, con los cambios realizados y siempre disfrutando de un estado íntegro.

Por último, se añadirá a la herramienta la posibilidad de indicar que el diseño de un sistema o agente se encuentra en un conjunto de archivos, tal y como se describió en el capítulo 6. En este caso, se construirá un programa que lea el contenido de cada archivo y traduzca la especificación dada en el conjunto de acciones estructurales necesarias

para crear el sistema software e iniciar su funcionamiento. Cabe destacar que, si existiera algún problema en la especificación, el sistema o agente no será creado y se informará apropiadamente al usuario. Por ejemplo, si se ha definido una acción compleja pero no las acciones transaccionales que forman parte de su descripción en TDL, el Metasistema informará del error y no se continuará con la construcción del sistema, de forma similar al procedimiento seguido si se estuviera trabajando de forma interactiva.

9.4 Conclusiones.

Hemos adaptado una herramienta de edición de estructuras conceptuales evolutivas, llamada *CS Editor,* para permitir describir, fácilmente, la estructura de un sistema software basado en agentes. Gracias a la flexibilidad de esta herramienta, hemos diseñado los atributos necesarios para los agentes, las acciones simples y las acciones complejas. Se han modificado las acciones evolutivas para determinar el funcionamiento del Metasistema según lo especificado en el capítulo de evolución de los agentes.

Hemos diseñado el conjunto de clases necesarias para crear una nueva herramienta llamada AgentHEDES, que, partiendo de la experiencia obtenida con HEDES y con *CS Editor*, permitirá no sólo hacer evolucionar un sistema sino que éste funcione e interaccione con los usuarios del sistema software basado en agentes desarrollado. Además se propone un paso intermedio que permita diseñar la jerarquía de agentes de forma gráfica mediante el editor pero que dicho diseño se traduzca en los elementos que componen la estructura del sistema.

Por último, la herramienta contemplará la posibilidad de suministrar el diseño de un sistema software utilizando el lenguaje de descripción introducido en el capítulo 6. A partir de la descripción del sistema y de los agentes que lo componen, éste se construirá automáticamente siguiendo la filosofía descrita por el modelo que hemos propuesto en esta tesis.

CAPITULO 10

Conclusiones y Trabajos Futuros

En este último capítulo vamos resumir las principales contribuciones que aporta esta tesis junto con las conclusiones que resumen el trabajo. Finalmente se introducen distintos trabajos futuros que complementarían y enriquecerían el modelo y el prototipo de la herramienta propuesto.

10.1 Aportaciones.

Hemos presentado un modelo arquitectónico evolutivo para desarrollar sistemas software basados en agentes. Con el fin de destacar las principales aportaciones del modelo propuesto vamos a presentar las diferencias encontradas con las arquitecturas dinámicas comentadas en el capítulo 3.

Los puntos a destacar en esta comparación son los siguientes:

1.- El tipo de *modelo de interacción y comunicación* existente entre los componentes de la arquitectura. Esta interacción puede ser explícita o implícita, síncrona o asíncrona.
2.- El tipo de *modelo de coordinación* entre componentes. Aquí vamos a destacar si existe un modelo propio soportado por la arquitectura o son los propios componentes los que deben coordinarse entre sí.

3.- Se permite o no la *composición de componentes*. Es decir, existe la posibilidad de crear componentes compuestos.
4.- Se permite o no la *composición de conectores*.
5.- Posibilita o no la *reutilización* tanto de arquitecturas como de componentes.
6.- Todas las arquitecturas o modelos arquitectónicos que vamos a comparar son dinámicos pero se puede distinguir entre *dinamismo arquitectónico restringido o no restringido*. El dinamismo arquitectónico restringido implica que todos los cambios en la arquitectura deben conocerse a priori. En el dinamismo arquitectónico no restringido, en principio, se permiten todos los cambios (añadir, borrar, sustituir y reconectar). La validez de los cambios debe asegurarse en tiempo de ejecución.
7.- Se utiliza o no un *metanivel* para la evolución.
8.- Se usa la *reflexión* tanto para describir el funcionamiento del sistema software como su evolución.
9.- Se realizan o no *comprobaciones de integridad y consistencia* del sistema al realizar un modificación en su estructura.
10.- La arquitectura soporta explícitamente las *transacciones*.
11.- Genera un *prototipo ejecutable*.

En los siguientes cuadros podemos ver las características de las distintas arquitecturas con respecto a los puntos presentados anteriormente. Comentamos, simultáneamente, las diferencias encontradas con respecto al modelo arquitectónico propuesto.

El **modelo de interacción y comunicación** entre los componentes varía mucho en las distintas arquitecturas, aunque como podemos observar en el primer cuadro comparativo, la mayoría se basa en un modelo de interacción y comunicación explícito. Esta característica hace más difícil la evolución de un sistema. Existen algunas arquitecturas como LEDA y OASIS que, además de los componentes y conectores, tienen otro elemento adicional que son los Roles. Un componente puede tener asociado más de un Rol. En estos lenguajes, son los Roles los que realmente se comunican e interaccionan. Por tanto, aunque los componentes no necesitan conocerse entre sí, existe un modelo de comunicación explícito entre los roles que son los que, en este caso, necesitan conocerse para cooperar. En esta tesis se presenta un modelo de interacción implícito y asíncrono donde los componentes no tienen que conocerse entre sí para interaccionar cuando sea necesario. De esta forma, se respeta la autonomía de los componentes, los agentes, y se facilita su evolución ya que les basta con conocer la existencia de la realización de una determinada acción, no quién la realiza.

Respecto al **tipo de interacción**, todos proporcionan una forma de interacción asíncrona, aunque algunos de ellos también permiten definir interacciones síncronas como LEDA, *Dynamic Wright* y OASIS.

En nuestro caso, la interacción entre componentes es, por defecto asíncrona, aunque tenemos un sistema de pre-condiciones que influye en el orden en el que se realizan las acciones dentro del sistema software. También se puede especificar cuándo dos acciones no deben realizarse simultáneamente.

	Modelo de interacción y comunicación	**Modelo de coordinación**	**Composición componentes**	**Composición conectores**
Scope	Explícito Unidireccional Emisión de eventos Asíncrono	No propio	Si	Si
LEDA	Roles (canales entre roles) Síncrono y asíncrono	No propio	Si	No
OASIS	Roles Síncrono, asíncrono	Contratos	Si	No
RAPIDE	Interfaces Explícito Asíncrono	Eventos y restricciones	Si	No
ACME	Interfaces (conjunto de puertos) Explícito Asíncrono	No propio	Si	Si
Dynamic Wrigth	Puertos y roles. Explícito Síncrono y asíncrono	Restricciones asociadas al estilo arquitectónico	No	No
Darwin	Enlaces Explícito Asíncrono	No propio	Si	No
Nuestra Propuesta	Implícito Asíncrono	Variante de *blackboard Controller*	Si	No

Cuadro 10.1 Comparación con otras arquitecturas dinámicas (1-4)

Podemos observar que la mayoría de las arquitecturas dinámicas comentadas no presentan un **modelo de coordinación** propio. Sólo en OASIS se propone un modelo de coordinación mediante contratos. Sin embargo, como ya vimos en el capítulo 4, la coordinación mediante contratos limita la autonomía de los componentes que deben coordinarse. En este trabajo se propone un modelo de coordinación que es un híbrido entre los llamados modelos de coordinación orientados a datos y los orientados a control. En este caso, usamos una estructura

de datos compartida como medio de comunicación y coordinación y un mecanismo de activación de agentes y notificación de eventos cuando ocurra un cambio en el sistema. Para ello usamos un agente especial que llamamos *Controller*. Realmente no usamos una única estructura de datos compartida entre todos los componentes que necesitan coordinarse sino que tenemos varias (similar al concepto de multi-espacios de tuplas de los lenguajes de coordinación derivados de *Linda*). Además, lo que diferencia el modelo de coordinación propuesto de los vistos en el capítulo 4 es la capacidad de adaptarse a los cambios producidos en el sistema software. Las tablas internas que mantiene el *Controller* o alguno de sus elementos se modifican, si es necesario, después de realizar una acción evolutiva, de tal forma que cuando el sistema continúe con su funcionamiento lo haga correctamente.

Para finalizar con el primer cuadro, vemos que la mayoría de las arquitecturas soportan la **composición de componentes** aunque muy pocas la **composición de conectores**, incluido el modelo propuesto en esta tesis. En nuestro caso, no es necesaria la composición de conectores ya que los componentes tienen asociada su propia estructura de datos compartida que será utilizada sólo por los componentes que tenga agregados. Las únicas operaciones que necesitan realizar son agregar un nuevo elemento o consultar si existe un determinado elemento.

La **reutilización** de arquitecturas normalmente está relacionada con el hecho de que se permita o no composición de componentes. Como vemos en el siguiente cuadro, sólo *Dynamic Wright* no la soporta.

Respecto a la evolución, las arquitecturas que implementan un **dinamismo arquitectónico** no restringido, como el modelo que proponemos, sólo proveen operaciones para añadir, eliminar o sustituir componentes pero sin permitir su modificación mientras el sistema funciona. Nosotros, además de las anteriores, proporcionamos un amplio conjunto de operaciones que nos permiten modificar la estructura y comportamiento de un componente, como vimos en el capítulo 7. Además, en nuestro modelo no se necesitan operaciones propias para añadir o eliminar conectores ya que el mecanismo de comunicación está íntimamente ligado al componente. Sin embargo, sí se proporcionan acciones estructurales para modificar las pre-condiciones de las acciones de los agentes, lo que implica un cambio en su reacción ante lo que ocurra en el sistema.

Como se puede observar, salvo OASIS y nuestra propuesta, ninguna arquitectura utiliza el **metanivel** como mecanismo para modificar un sistema mientras evoluciona. Además, en ninguna, excepto en OASIS y en nuestra propuesta, se utiliza la **reflexividad** para expresar tanto el funcionamiento como la evolución de un sistema.

	Reutilización arquitecturas	Dinamismo arquitectónico	Metanivel	Reflexivo
Scope	Si	No restringido: Añadir y eliminar componentes Establecer y eliminar conexión	No	No
LEDA	Si	No restringido: Añadir y eliminar componentes Establecer y eliminar conexión	No	No
OASIS	Si	No restringido: Añadir y eliminar componentes Establecer y eliminar conexión	Si	Si
RAPIDE	Si	Restringido	No	No
ACME	Si	Restringido	No	No
Dynamic Wrigth	No	Restringido	No	No
Darwin	Si	Restringido	No	No
Nuestra Propuesta	Si	No restringido: Añadir, eliminar y **modificar** componentes	Si	Si

Cuadro 10.2 Comparación con otras arquitecturas dinámicas (5-8)

Al realizar una modificación en tiempo de ejecución de un sistema, hay que garantizar que el sistema quede en un estado consistente, íntegro. Esta característica la podemos observar en el cuadro siguiente. Sólo *Scope* y LEDA, además de nuestra propuesta, realizan algún tipo de **comprobación de integridad** pero está muy limitada. *Scope* sólo permite asociar invariantes a los tipos de eventos. De esta forma, sólo se permitirá cambiar los tipos de eventos que pueden comunicarse dos componentes si se cumplen sus invariantes. LEDA sólo comprueba que cuando se elimina un componente y se sustituye por otro, el nuevo componente sea compatible con el anterior.

A continuación se resalta si la arquitectura soporta la definición de **transacciones** o no. Como se puede observar, sólo OASIS y nuestra propuesta tratan específicamente el tema de las transacciones. Esto no quiere decir que no se puedan conseguir definir transacciones con las demás arquitecturas pero bajo la responsabilidad del desarrollador.

Finalmente, muchas de las arquitecturas vistas proporcionan un **prototipo ejecutable** que permita ver y comprobar el diseño realizado del sistema a distintos niveles de abstracción. Nuestro modelo también

proporciona un herramienta que nos permite generar un prototipo ejecutable del sistema. Además, también se proporciona un lenguaje para especificar textualmente el sistema y sus componentes.

	Comprobación integridad	**Transacciones**	**Prototipo ejecutable**
Scope	Sólo invariantes para comprobar los cambios realizados en los tipos de eventos	No	-
LEDA	Sólo comprueban la compatibilidad entre componentes	No	Si
OASIS	-	Si	Si
RAPIDE	No	No	Si
ACME	No	No	No
Dynamic Wrigth	No	No	No
Darwin	No	No	No
Nuestra Propuesta	Si	Si	Si

Cuadro 10.3 Comparación con otras arquitecturas dinámicas (9-11)

Podemos resumir, tras esta comparación, que las principales aportaciones de nuestro modelo están relacionadas con la evolución controlada y consistente de un sistema y con la incorporación de un modelo de coordinación entre los componentes que está incluido en la estructura de cada uno de ellos.

10.2 Conclusiones.

Partiendo de los conceptos del modelo MEDES y su herramienta asociada HEDES, se ha estudiado cómo modificar el modelo para desarrollar sistemas software basados en agentes evolutivos, es decir, sistemas que puedan cambiar su funcionamiento y estructura a través del tiempo. El modelo propuesto define la arquitectura que tiene un sistema software de este tipo. Se contempla un sistema como una jerarquía de agentes autónomos, reactivos, pro-activos y sociales. Los agentes son capaces de relacionarse y cooperar para llevar a cabo acciones complejas o transacciones. Esta **arquitectura** es **dinámica**, ya que permite modificar cada uno de los elementos que componen un sistema (agentes, acciones, pre-condiciones, ...).

Conseguimos separar la evolución de un agente/sistema de su funcionamiento gracias al mantenimiento de dos partes bien diferenciadas dentro de su estructura: la parte funcional y la parte estructural. Ambas partes se tratan de forma homogénea, cada una mantiene una historia de las acciones realizadas, la historia funcional almacena la información de las acciones funcionales realizadas en él y la historia estructural contiene las ocurrencias de las modificaciones, acciones estructurales, que ha sufrido el agente desde su creación.

Se han utilizado **dos lenguajes**, basados en la lógica temporal de predicados, para describir las condiciones asociadas a la realización de una acción. Si la acción es funcional, hablamos de pre-condiciones y durante-condiciones. A una acción estructural se le asocian pre y post-condiciones. Estos dos lenguajes son: el lenguaje L para describir las acciones funcionales y el lenguaje M2 para describir las acciones estructurales.

Se ha propuesto un **modelo de coordinación** entre agentes que ha dado lugar, posteriormente, a la definición de un patrón de diseño llamado PDN (*Precondition Dynamic Notifier*). Este modelo de coordinación combina el estilo arquitectónico mediante repositorios, concretamente su variante llamada *blackboard*, y el estilo de invocación implícita. El modelo de coordinación propuesto separa el aspecto de coordinación de los agentes del aspecto de computación o funcionamiento, ya que los agentes no tienen que realizar acciones específicas para comunicarse y cooperar con otros, todo se hace a través de una historia común. Las ocurrencias de las realizaciones de las acciones por los agentes se almacenan en una historia, así como los estímulos recibidos. Un agente sólo sabrá que se ha realizado una acción en el sistema o que se ha recibido un estímulo si está interesado en él, es decir, si este cambio de estado en el sistema puede hacer que él realice una acción. Cada acción tiene asociada una única pre-condición que define cuándo se pueden realizar y que depende de lo ocurrido en el sistema hasta el instante de su evaluación. No importa qué agente realice una acción en el sistema sino que ésta se haya realizado.

Hay que tener en cuenta que no tenemos una única historia del sistema sino que, al tener una jerarquía de agentes, tenemos una historia por cada agente que almacena la información de lo ocurrido dentro de dicho agente, por sus agentes hijos. **La gestión de dicha historia y el mecanismo de activación** se lleva a cabo por el *Controller* y sus agentes, *Evaluator* y *Ocurrence Receiver*. Debido a que la eficiencia del sistema depende en gran medida del comportamiento de cada *Controller*, hemos estudiado distintas formas de optimizarlo. Se ha visto cómo agilizar el tiempo dedicado a la evaluación de las pre-condiciones que realiza el agente *Evaluator*, mediante la paralelización de la

evaluación de una única condición o de varias. Por otro lado, sabemos que se pueden producir problemas de inanición que se resuelven utilizando métodos de selección mejorados. Además, también se ha estudiado cómo activar a un agente cuando se asegure que la acción tiene una alta probabilidad de poder realizarse y evitar la evaluación de una pre-condición cuando ésta se compone de un único estímulo u ocurrencia.

Por otro lado, con la idea de mejorar el tiempo de ejecución del sistema, hemos incorporando las **Redes de Petri Coloreadas** (RdPC) como mecanismo de ejecución del sistema software. El proceso que se sigue para la creación de la RdPC del sistema sigue dos fases, la fase de construcción y la de composición. Cada agente que no sea un agente hoja dentro de la jerarquía de agentes, tiene asociada una o más RdPC que definen su funcionalidad, es decir, que definen la secuencia de ejecución de las acciones de sus agentes hijos. El *Controller* de cada agente es el encargado de introducir las marcas en los lugares de la RdPC adecuada cuando reciba una ocurrencia o un estímulo. El *Evaluator* utilizará la o las RdPC para realizar las consultas, *query*, solicitadas por los agentes.

Se han incorporado las **acciones complejas o transacciones** a los agentes y los mecanismos necesarios para llevarlas a cabo. En concreto, hemos creado un agente especial, el agente *Transaction Manager*, para gestionar cada transacción cuando ésta se inicia en el sistema. Si la transacción no finaliza con éxito, en el sistema no quedará ninguna señal que indique que ésta se ejecutó parcialmente, es decir, el estado del sistema no habrá sufrido cambios. Se permite la ejecución simultánea de varias transacciones. Además se ha diseñado un lenguaje para describir las transacciones llamado TDL (*Transaction Description Language*) y se han presentado dos nuevas clases de pre-condiones asociadas a las acciones transaccionales, las pre-condiciones de secuencia (pre-condiciones *s*) y las pre-condiciones transaccionales (pre-condiciones *t*).

La **evolución** de un sistema software se logra gracias al funcionamiento del Metasistema y de los Sistemas Genéticos de los agentes. Para ello, el Metasistema tiene asociado un conjunto de acciones estructurales o de evolución previamente definido. Cada una de éstas acciones está cuidadosamente estudiada para lograr que, después de la modificación realizada en el sistema, éste quede en un estado consistente, íntegro. Para cada acción estructural se ha definido su pre y post-condición utilizando el lenguaje M2, un lenguaje basado en la lógica temporal de predicados y se ha descrito el conjunto de acciones que se han de realizar en el agente durante la ejecución de dicha acción estructural. Además, queda especificado el mecanismo de propagación de cambios para cada una de las acciones estructurales que lo requiera. También

se ha propuesto una lista de invariantes adaptada a los sistemas software basados en agentes.

Se ha desarrollado un **prototipo** de la herramienta que permite tanto hacer funcionar a un sistema software como hacerlo evolucionar. Esta herramienta posee una interfaz gráfica que facilita su uso al equipo de desarrollo y a los usuarios del sistema. que permita tanto crear un sistema software como hacerlo evolucionar.

Hemos definido un **lenguaje** que permite describir la estructura de un sistema software basado en agentes según el modelo y la arquitectura propuesta. En este lenguaje la descripción del sistema y de los agentes se almacena en archivos independientes, lo cual facilita la reutilización de éstos.

10.3 Trabajos futuros.

A continuación vamos a exponer las líneas de investigación futuras que seguirán completando este trabajo de investigación. No se puede decir que se ha terminado un trabajo de investigación, éste debe evolucionar, al igual que los sistemas software que hemos descrito.

Es muy interesante poder incorporar nuevas características a los agentes como son el aprendizaje y la capacidad de decisión que definen a los **agentes inteligentes**. Se puede estudiar cómo afectarían estas nuevas capacidades a la arquitectura propuesta y a la forma de evolución que actualmente se lleva a cabo.

Debido a la gran influencia que tiene Internet en nuestros días y a los sistemas distribuidos, otra línea de investigación futura puede ser la de adaptar la arquitectura descrita para un sistema software pensando en que su ejecución se realizará sobre un **sistema distribuido**. Con este fin, también intentaríamos diferenciar el aspecto de distribución de los demás aspectos asociados a un agente (coordinación, computación). Así mismo, y como se puede observar en las publicaciones recientes, existe un gran interés en la **capacidad de movilidad** de los agentes. Un agente móvil es aquel que es capaz de moverse por los nodos de una red con el fin de realizar las funciones para las cuales fue diseñado. Este tema es muy interesante y es otra de las líneas que complementarían este trabajo.

Puesto que nuestro modelo se basa en definir agentes independientes y autónomos que tienen sus acciones bien definidas, otro objetivo futuro es ver cómo se puede integrar la descripción de los sistemas en nuestro modelo con la **programación orientada a componentes**. Es decir,

cómo integrarlo para convertir a los sistemas en componentes que puedan ser reutilizados en sistemas heterogéneos. Esto, posiblemente, se puede llevar a cabo añadiendo una capa de abstracción superior y utilizando los lenguajes adecuados para conectarlos con otros componentes.

El **prototipo** presentado no está totalmente desarrollado, uno de los trabajos futuros es completarlo e incorporarle las distintas optimizaciones que hemos estudiado. Una de ellas es la utilización de las RdPC para hacer funcionar al sistema software. Para ello se puede añadir un editor gráfico que permita describir fácilmente la o las RdPC de cada agente o incluso que las genere automáticamente partiendo de la descripción de las pre-condiciones de las acciones tal y como hemos descrito en el proceso de construcción y composición descrito en el capítulo 6.

Incorporar a la herramienta mecanismos que nos permitan **validar el comportamiento** del sistema software desarrollado tomando como base los requisitos que tiene asociados. Esto se facilita si se han incorporado las RdPC como mecanismo de ejecución del sistema.

Relacionado con lo anterior, es interesante poder **estudiar** las **propiedades** que tiene el sistema y **verificarlas** en cada momento. Esto es posible gracias a que la especificación del sistema software basado en agentes ya se hace con un lenguaje formal.

I. Definición del lenguaje L.

El lenguaje L propuesto por Rodríguez Fórtiz [Rodríguez00a] se utiliza para especificar las acciones funcionales de los agentes y sus pre-condiciones de las acciones funcionales de los agentes. Este lenguaje está basado en la lógica temporal de predicados.

El lenguaje estará compuesto por:

Const: constantes *Cl= {a,b,c,...}* de diferentes dominios.

Vars: variables *Vl= {x, y, z, u, v, w, x_0, x_1,..., x_n }*. Se utilizan las últimas letras del alfabeto y la letra x con subíndices. Cada variable puede tomar valor en un dominio diferente. Con *Dom(x_i)* se representa el dominio de una variable x_i. Por otro lado, $\vec{x}$, $\vec{y}$, $\vec{z}$ son grupos de variables, cada una de las cuales puede pertenecer a diferente dominio.

Preds: predicados *Pl* con la forma: $p(\vec{x})$, donde $\vec{x}= \{x_1 ,x_2,..., x_j \}$ representa a las variables del predicado que pueden ser unificadas o instanciadas con valores del dominio de cada tipo de variable*:* $x_i = a$, $a \in Dom(xi)$. Si el predicado tiene sus variables instanciadas se le llama predicado instanciado.

Operadores lógicos: *and, or, not.*

Operadores temporales: *once (◊), since (S) y andT*. No incluimos ni los operadores *previo*, ni *siempre_en_el_pasado*.

Operadores relacionales: =, ≠, >, <, ≥ y ≤

Cuantificadores: Dos universales: $\forall$, $\forall_e$, y tres existenciales*:* $\exists$, $\exists_f$, $\exists_l$.

No consideramos funciones.

Fórmulas: una fórmula es una expresión del lenguaje que se obtiene de la aplicación de operadores a los predicados y cuantificadores a las variables:

1) *true* y *false* son fórmulas atómicas.

2) Así, dado $p(x^{\rightarrow})$ y $q(y^{\rightarrow})$, donde p y q son nombres de predicados del lenguaje L y $\{x^{\rightarrow}\}$ e $\{y^{\rightarrow}\}$ los conjuntos de las variables de estos predicados:

- *once* $p(x^{\rightarrow})$ es una fórmula atómica
- *not once* $p(x^{\rightarrow})$ es una fórmula atómica
- son fórmulas simples: *once* $p(x^{\rightarrow})$ *and once* $q(y^{\rightarrow})$; *once* $p(x^{\rightarrow})$ *or once* $q(y^{\rightarrow})$; $p(x^{\rightarrow})$ *andT* $q(y^{\rightarrow})$; $p(x^{\rightarrow})$ *since* $q(y^{\rightarrow})$; *not* $p(x^{\rightarrow})$ *since* $q(y^{\rightarrow})$
- también $\forall x$ *once* $p(x^{\rightarrow})$; $\forall_e x$ *once* $p(x^{\rightarrow})$; $\exists x$ *once* $p(x^{\rightarrow})$; $\exists_f x$ *once* $p(x^{\rightarrow})$; $\exists_l x$ *once* $p(x^{\rightarrow})$ son fórmulas simples.

3) *not g ; f and g ; f or g ;* $\forall x\ g$; $\forall_e x\ g$; $\exists x\ g$; $\exists_f x\ g$; $\exists_l x\ g$ son fórmulas complejas, donde f es una fórmula atómica, simple o compleja y g es una fórmula simple o compleja.

Llamaremos *Fl* al conjunto de todas las fórmulas que se pueden escribir con este lenguaje.

En una fórmula tienen mayor precedencia los operadores temporales que los lógicos, además, se establece un orden de precedencia de izquierda a derecha, y los paréntesis se utilizarán para cambiarlo.

Cláusulas: son expresiones con la forma*:*

Cabeza ← *Cuerpo*

Resultado del cierre universal de la fórmula *Cuerpo* → *Cabeza* que equivale a *not Cuerpo or Cabeza.* No deben ser necesariamente cláusulas de *Horn* ya que no aparece exclusivamente el operador *and* en el cuerpo.

Cabeza: es un único predicado de la lógica temporal de predicados y modela una acción de un procesador.

Cuerpo: es una fórmula que representa la precondición de la acción de la cabeza.

Programa Lógico: Conjunto de todas las cláusulas con las precondiciones de todas las acciones de un sistema en un momento dado.

En la siguiente tabla podemos la descripción de este lenguaje en notación BNF.

programa_ lógico::= {<cláusula>}
cláusula ::= <cabeza> ← <cuerpo>
cabeza ::= <predicado>
cuerpo ::= <fórmula>
*predicado ::= <nombre_prec> **(** <variable> [, <variable>] **)***
nombre_prec ::= una palabra
*variable::= **x** | **y** | **z** | **u** | **v** | **w** | x_0, | x_1 | ... | x_n*
*constante::= **a** | **b** | **c**| **...***
<cuantificador > ::= ∀, | $\forall_e$ | ∃ | $\exists_f$ | $\exists_!$.
<fórmula> ::= <f_atómica> | <f_simple> |<f_compleja>
*<f_atómica> ::= [**not**] **once** <predicado>*
*<f_simple> ::= **once** <predicado> **and once** <predicado> | **once** <predicado> **or once** <predicado> | <predicado> **andT** <predicado> | <predicado> **since** <predicado> | **not** <predicado> **since** <predicado> | <cuantificador> <variable> **once** <predicado>*
*<f_compleja>::= **not** (<f_simple> | <f_compleja>) | <formula> **and** (<f_simple> | <f_compleja>) | <formula> **or** (<f_simple> | <f_compleja>) | <cuantificador> <variable> (<f_simple> | <f_compleja>)*
*predicado_consultado::=<nombre_prec> **(**<variable_consul> [,<variable_consul>] **)***
variable_consul ::= <variable> [<op_rel> (<valores> | <fórmula_query>)]
*<valores> ::= **(** <constante> [{ **,** <constante> }] **)***
<fórmula_query>::=<cuantificador> <variable> (<f_simple> | <f_compleja>)

Tabla I.1 Notación BNF para el lenguaje L.

II. Las consultas sobre la historia.

Es necesario conocer el estado de un agente, es decir, qué acciones ha realizado hasta el instante actual, tanto funcionales como estructurales. Recordemos que, para que una acción pueda comenzar a realizarse, el *Evaluator* correspondiente debe evaluar la pre-condición de dicha acción en base a la historia del agente padre y determinar si ésta se cumple o no. También puede ser necesario evaluar las durante y post-condiciones asociadas a las acciones.

En este trabajo, nos hemos basado en la tesis de Rodríguez Fórtiz [Rodríguez00a] donde se elaboran los algoritmos necesarios para realizar las consultas sobre la historia. En este apéndice reproducimos los algoritmos utilizados para realizar dichas consultas.

Las consultas pueden dividirse en dos tipos:

1. *Particulares*: devuelven valores *true* o *false*. Se utilizan en la evaluación de las pre-condiciones de las acciones.
2. *Generales*: devuelven listas de valores. Se corresponden con el concepto de *query*. Un *query* es una expresión que se evalúa sobre la historia y que permite obtener un subconjunto de un dominio de valores de uno o más atributos, cuando se consulta la ocurrencia de acciones relacionadas y en las que estos atributos intervienen.

II.1 Algoritmo de evaluación de consultas particulares

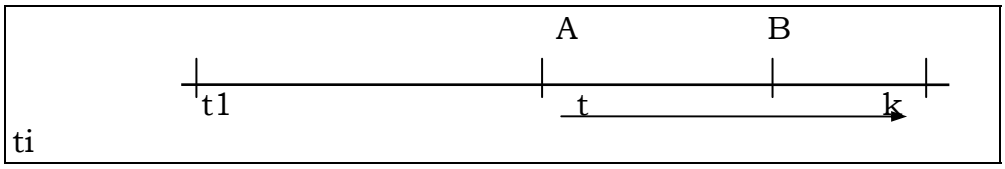
Para evaluar consultas particulares sobre la historia se toman los eventos de ésta como una traza y se determina si un evento se ha realizado antes o después de otro. Son el tipo de consulta más directa y para realizarla, se realiza sobre la historia la siguiente pregunta compuesta:

¿Qué evento de una acción dada, *A*, ha ocurrido en el instante *t* ?

Y

¿Ha ocurrido algún otro evento de otra acción *B* desde el instante *t*?

El grafo del tiempo es el siguiente:



La pregunta se corresponde con la evaluación e interpretación del operador *since* de la lógica temporal definida por Gabbay [Gabbay95]:

B since A es cierto si y sólo si *B* fue cierto desde que *A* fue cierto, o lo que es lo mismo, si B fue cierto y previamente lo había sido A:

B since A = true sí y solo sí:

(1) $\exists k, t_i > k > t_1$, $(\sigma, k) \not\models B$ *y* *(2)* $\exists j, t_i > k\text{-}j > t_1$, $(\sigma, k\text{-}j) \not\models A$,

donde σ es la secuencia de tiempo dada por el intervalo $[t_1,t_i]$.

En el primer bucle (1) se busca la existencia de un evento de *B,* y en el segundo, de un evento de *A*, cuando *B* ya ha sido encontrado. La búsqueda se hace hacia atrás.

```
k = ti
while k > t1 + 1                                                        (1)
      if  B|t=k
            repeat
                  if  A|t=k-1                                           (2)
                        then (B since A = true) end.
                        else k=k-1
            while k> t1+1
      else (B since A =false) end
end-while
B since A = undefined end
```

donde t_1 representa el instante inicial y *t* el instante de la evaluación, que suele coincidir con el instante actual. La consulta se hace comenzando por el último evento de la traza y se va hacia atrás, para lo cual se utiliza *k* para referenciar el instante en el que se está buscando en cada momento.

$A|_{t=k}$ representa la evaluación de *A* en un tiempo *t=k*. La comprobación sobre la historia se hace basándonos en el cálculo relacional, tratando la historia como una tabla y mediante operaciones de selección y proyección.

La evaluación será *true* o *false* dependiendo de si se encuentra un evento de esa acción en ese instante de tiempo o no. En caso de indeterminación, *undefined*, se considera que la consulta devuelve un valor *false*.

Gabbay demuestra que todos los operadores temporales pueden expresarse utilizando solamente el operador temporal *since*. Por ello, todas las consultas que incluyen el operador *once* pueden transformarse en consultas que incluyan solamente *since* y aplicarles el mismo algoritmo.

II.2 Algoritmo de evaluación de consultas generales

Ofrecen un subconjunto de la historia de eventos relacionados con uno o más procesadores y que satisfacen las mismas condiciones. Estas consultas se hacen cuando no basta como respuesta un *true* o un *false* , sino que se necesita conocer una lista de eventos concretos para:

- Obtener información sobre el pasado.
- Simular la ocurrencia de acciones del pasado.
- Simular la ocurrencia de acciones en el instante actual.
- Tomar decisiones de ejecución por parte del sistema o del Metasistema para poder hacer instanciaciones posteriores basadas en los valores obtenidos.

En una consulta general se especifica:

- El ámbito de consulta: qué eventos de cada acción se consideran en la consulta. Puede ser el primero de una acción, el último o todos los ocurridos.
- El atributo a consultar de cada acción.
- Las relaciones lógicas y temporales entre acciones que, al igual que para las condiciones, se transforman en una expresión que contiene el operador *since*.

Para evaluar una consulta general hay que responder a estas tres preguntas:

¿Qué evento de una acción *A* ha ocurrido en el instante *t* ?

Y

¿Qué otros eventos de otra acción *B* han ocurrido desde el instante *t*?

Y

¿Qué valor tiene el atributo *Att* en los eventos de la acción *B*?

Que equivalen a la evaluación del operador *since*, la cual, si es *true*, va seguida de la consulta de un atributo. Al igual que para las consultas particulares, una consulta en la que aparezca el operador *once* será transformada en una que sólo contenga *since* como operador temporal.

El algoritmo que proponemos para resolver las consultas generales está basado en el utilizado para las consultas particulares:

$k = t_i$
$RE = \varnothing$

while $k > t_1 + 1$
  if $B|_{t=k}$
    repeat
      if $A|_{t=k-1}$
        then (B since A = true)
          $RE = RE \cup \{$ *evento de B}* *at t*
        else $k=k-1$
    while $k > t_1+1$
end-while

R= {valores de Att en RE}

R es la respuesta a la consulta y contiene los valores del atributo *Att* para los eventos de *B* acumulados en *RE* cuando (*B since A*) es *true*.

Si en la expresión de una consulta hay varias subfórmulas, el algoritmo se utilizará para cada una de ellas, obteniéndose diferentes conjuntos de eventos *RE*. La aparición de operadores lógicos como *and*, *or* y *not* para conectar las subfórmulas influye a la hora de construir el conjunto *R* de valores de atributos a partir de los conjuntos de *RE*. Así, si el operador es *and*, se realizará una intersección de conjuntos, si es *or* una unión, y si es *not* una diferencia.

Puede darse el caso de que se esté consultando la posibilidad de la realización de una acción y se desee conocer el valor que tomarán sus atributos cuando se lleve a cabo. Antes de obtener estos valores debemos verificar su precondición para saber si puede o no realizarse.

III. Mecanismos de comunicación y sincronización.

En este apéndice presentamos un estudio sobre los diferentes mecanismos de comunicación y sincronización existentes. En él se diferencian los mecanismos de comunicación-sincronización de las formas de especificar la ejecución concurrente. Esta división es la que hacen G. R. Andrews y F. B. Schneider [Andrews83]. Ellos presentan una discusión sobre tres cuestiones que constituyen la base de todas las notaciones de la programación concurrente:

- cómo se expresa la ejecución concurrente
- cómo se comunican los procesos
- y cómo se sincronizan los procesos

III.1 Formas de especificar la ejecución concurrente.

A continuación comentamos brevemente diferentes formas que se han utilizado y se utilizan para especificar la ejecución concurrente.

III.1.1 Corrutinas.

Las *corrutinas* son construcciones de programa similares a las subrutinas, pero con una transferencia de control simétrica en lugar de jerárquica. La transferencia de control entre una corrutina y otra se realiza mediante una operación específica:

```
resume 'subrutina'
```

Cuando se ejecuta *resume,* se transfiere el control a la rutina especificada, salvando la información de estado necesaria para cuando se vuelva a la ejecución de la rutina que ahora se abandona.

El programa principal, en la primera invocación a una corrutina, utilizará la sentencia *call* (esta instrucción dependerá del lenguaje, igual que la instrucción *return*) y cuando una corrutina desee devolver el control al programa principal, usará la sentencia *return.*

Con *resume* se provoca la sincronización de procesos. Sin embargo las corrutinas no son adecuadas para el paralelismo real, debido a que sólo

permiten la ejecución de una única corrutina en cada momento. Cuando se ejecuta una instrucción *resume*, ésta ignora a qué punto pasa el control, lo cual nos proporciona cierta independencia.

III.1.2 Sentencias *fork* y *join* (bifurcar y reunir).

La sentencia *fork P*, siendo P un proceso, especifica que la ejecución de P debe comenzar en paralelo con la del proceso que efectúa la operación. La sentencia *join P* hace que el proceso que la ejecuta espere la terminación de P, es decir, permite la sincronización entre el proceso P y el proceso desde donde se ejecuta la instrucción *fork*.

Cuando se usan de una forma disciplinada, estas instrucciones son prácticas y potentes, por ejemplo, *fork* puede utilizarse como mecanismo para la creación dinámica de procesos.

III.1.3 Instrucción concurrente: *cobegin-coend* (*parbegin-parend*).

Este par de instrucciones causan la ejecución concurrente de los procesos o conjunto de instrucciones que engloban. En el siguiente ejemplo:

```
cobegin S1; S2; ... Sn; coend;
```

se ejecutarían concurrentemente S1, S2,... , Sn y su ejecución terminará cuando lo hagan todos los procesos concurrentes. El problema es que tiene cierta dificultad en su realización aunque esto no es un factor determinante.

III.2 Mecanismos de comunicación y sincronización.

A continuación se describen distintos mecanismos de comunicación y/o sincronización.

III.2.1 Primitivas de sincronización basadas en variables compartidas.

Espera ocupada (Busy-waiting)

Es una forma de implementar la sincronización usando variables compartidas, cuyos valores son actualizados y comprobados por los

procesos. Este mecanismo se utiliza adecuadamente en la implementación de la sincronización que depende de cierta condición. La condición será verdadera cuando un proceso coloque un valor determinado en una variable compartida. La sincronización se consigue porque se obliga a los procesos a que esperen hasta que la condición se cumpla. Esto se lleva a cabo haciendo que los procesos comprueben repetidamente el valor de la variable compartida hasta que ésta tenga el valor deseado.

Se dice que el proceso que realiza la espera ocupada está *'spinning'* (en órbita). Las variables usadas de esta forma se denominan algunas veces *'spin locks'*.

Para implementar la exclusión mutua usando instrucciones de espera ocupada, la actualización y la espera en las condiciones se combinan dentro de los protocolos (de entrada y de salida a las secciones críticas involucradas) cuidadosamente construidos.

Semáforos

Los semáforos son un mecanismo fácil de implementar y lo suficientemente potente para solucionar problemas de programación concurrente. Pueden usarse para definir o implementar primitivas estructuradas de más alto nivel. Un semáforo '*s*' es (Dijkstra) una variable entera que sólo puede tomar valores enteros no negativos. Una vez que se le ha asignado un valor inicial (operación de inicialización), las únicas operaciones que se pueden realizar sobre 's' son: Wait(s) (o P(s) o Down(s)) y Signal(s) (o V(s) o Up(s)). Estas operaciones se definen como:

- `Wait(s) -> si s > 0 entonces s = s - 1 si no el proceso que realizó esta llamada queda suspendido.`
- `Signal(s)-> si hay procesos suspendidos en la cola de este semáforo entonces desbloquear uno, si no s = s + 1.`

Lo descrito anteriormente es para *semáforos generales*, si el semáforo sólo puede tomar valores 1 y 0 entonces hablamos de *semáforos binarios*.

La operación de inicialización del semáforo a un valor entero no negativo sólo puede hacerse una vez, antes de que se use y la mayoría de las veces se realiza en el programa principal.

Los semáforos son una herramienta muy general para resolver problemas de sincronización (de condición y de exclusión mutua) pero la comunicación será responsabilidad del programador. Se basan en la existencia de memoria común ya que se implementan utilizando variables compartidas.

Regiones críticas y regiones críticas condicionales

Regiones Críticas (Hoare 1972):

Los semáforos no están relacionados sintácticamente con los recursos compartidos que protegen y el compilador no sabe si una estructura es compartida y debe estar controlada. Para ello, Brinch Hasen (1972) propuso una construcción del lenguaje, llamada *Región Crítica* (RC). Una RC [Milenkovic94] protege a una estructura de datos compartida haciéndola conocida al compilador, el cuál puede entonces generar código que garantice el acceso exclusivo a los datos afectados. La declaración de una variable compartida tiene el siguiente formato:

```
var mutex : shared T;
```

'shared' informa al compilador que la variable 'mutex', de tipo 'T', definido por el usuario, es compartida por varios procesos. Los procesos pueden acceder a una variable protegida por medio de la construcción *'region'* tal como sigue:

```
region mutex do
```

donde, la instrucción compuesta que sigue a 'do' es ejecutada como sección crítica. Cuando genera código para una región, el compilador inserta automáticamente un par de operaciones WAIT y SIGNAL, o sus equivalentes, alrededor de la sección crítica.

La desventaja es que las RC sólo solucionan los problemas de exclusión mutua y no los problemas normales de sincronización entre procesos.
Regiones Críticas Condicionales (RCC):

Una RCC es sintácticamente similar a una RC. La variable compartida se declara del mismo modo, se vuelve a utilizar la construcción *'region'* para controlar el acceso y la única palabra clave nueva es '*await*'.

La implementación de esta construcción permite a un proceso esperar en una condición dentro de una RC quedando suspendido en una cola especial, pendiente de la satisfacción de la condición correspondiente. A diferencia de un semáforo, una RCC puede admitir a otro proceso dentro de la sección crítica.

```
var v : shared T;
begin
...
region v do
      begin
        ...
```

```
await condicion;
 ...
end;
```

Un proceso que espera por una condición no impide que otros utilicen el recurso. Cuando la condición sea verdadera, se despertará al proceso suspendido. Cada vez que un proceso abandona la sección crítica, se evalúan todas las condiciones que han suspendido a procesos anteriores, y si se cumplen, se despierta a uno de ellos. Se concede precedencia a los procesos que esperan, asegurando así que no se retrasen los procesos en espera indefinidamente.

Debido a que su implementación es un poco compleja, las RCC son raramente soportadas directamente en sistemas comerciales.

Monitores

Un monitor se forma mediante la encapsulación de la definición de un recurso y las operaciones que lo manipulan. Esto permite que un recurso sea visto como un módulo. Así, un programador puede ignorar los detalles de implementación del recurso cuando lo usa.

Podemos definir un monitor como una colección de variables permanentes, usadas para almacenar el estado del monitor, y algunos procedimientos, los cuales implementan operaciones sobre el recurso. Un monitor también tiene código de inicialización para las variables permanentes, que se ejecuta una vez antes de ejecutarse cualquier procedimiento. Los valores de las variables permanentes se conservan entre las activaciones de los procedimientos del monitor y pueden ser accedidas sólo desde dentro del monitor.
Los procedimientos del monitor pueden tener parámetros y variables locales, cada uno de los cuales obtiene nuevos valores para cada activación del procedimiento.

La invocación es semánticamente parecida a la llamada a un procedimiento. Además, la ejecución de los procedimientos en un monitor dado garantiza la exclusión mutua, es decir, la entrada a un monitor por un proceso excluye la entrada a él por cualquier otro proceso. Esto asegura que las variables permanentes nunca son accedidas concurrentemente.

Expresiones de camino (Path expressions)

Otro enfoque para la definición de un módulo sujeto al acceso concurrente es suministrar un mecanismo con el cual un programador especifique, en una parte de cada módulo, todas las restricciones sobre la ejecución de las operaciones definidas por el módulo. La implementación de las operaciones está separada de la especificación de

las restricciones. Por otra parte, el código para hacer cumplir las restricciones es generado por el compilador. Las expresiones de camino (definidas por Campbell y Habermann, 1974) son un mecanismo de sincronización que adopta este enfoque.

Cuando se usan expresiones de camino, un módulo que implementa a un recurso tiene una estructura como la de un monitor. Contiene variables permanentes que almacenan el estado del recurso y procedimientos que realizan las operaciones sobre el recurso. Las expresiones de camino en la cabecera de cada recurso definen las restricciones sobre el orden en el que se ejecutan las operaciones. En los procedimientos no hay código de sincronización.

La sintaxis de una expresión de camino es:

```
path path-list end
```

donde, la lista *path-list* contiene los nombres de operación y los operadores *path*. Los operadores *path* incluyen ',' para la concurrencia, ';' para la secuencialidad, 'n:(path-list)' para especificar n activaciones concurrentes de path-list, y '[path-list]' para especificar un número no limitado de activaciones concurrentes de path-list.

Las expresiones de camino están basadas en el enfoque operacional. Una expresión de camino define todas las secuencias legales de las ejecuciones de las operaciones sobre un recurso. Este conjunto de secuencias puede verse como un lenguaje formal, en el cual cada sentencia es una secuencia de nombres de operaciones.

Sin embargo, si bien las expresiones de camino nos suministran una notación elegante para expresar las restricciones de sincronización, se adaptan mal en la especificación de la sincronización de condición (Bloom,1979). Si una operación puede ejecutarse dependiendo del estado del recurso, sería necesario poder acceder a los parámetros y/o a su información de estado cuando se estén tomando las decisiones de sincronización. Así, además de las expresiones de camino, seria necesario introducir otros mecanismos para solventar estos problemas.

Sleep-wakeup (dormir-despertar)

Son operaciones similares a las que utilizaría el sistema operativo, actúan sobre los procesos, no sobre partes de ellos. *sleep* provoca el bloqueo del proceso llamante y *wakeup 'proceso'* desbloquea al proceso que se pasa como argumento. Si el proceso aludido no está dormido, se pierde.

Locks (llaves o cerrojos)

Las llaves o cerrojos operan sobre algo, un ente, que se encuentra dentro de un proceso. Es un tipo de dato que puede tomar los valores: abierto y cerrado. Cada cerrojo tiene asociada una lista con los procesos bloqueados en el cerrojo. Se pueden realizar las siguientes operaciones sobre él:

```
- lock (v): si v = abierto -> v = cerrado
            si v = cerrado -> incluir proceso en la lista de
                              bloqueados de v

- unlock (v): si lista de procesos = vacía -> v = abierto
              si lista de procesos ≠ vacía -> desbloquear uno
```

III.2.2 Primitivas de sincronización basadas en el paso de mensajes.

Paso de mensajes

Los mensajes sirven tanto para resolver los problemas de sincronización como los de comunicación entre procesos concurrentes. Se definen dos operaciones en el mecanismo de paso de mensajes: *send* (enviar) y *receive* (recibir). La comunicación se consigue porque un proceso, después de recibir un mensaje, obtiene valores enviados por el proceso emisor. La sincronización se consigue porque un mensaje puede recibirse sólo después de que haya sido enviado, lo cuál define el orden en el que pueden ocurrir dichos eventos. Esto nos lleva a un tipo de sincronización, pero se pueden conseguir otros dependiendo del tipo de paso de mensajes usado.
La implementación de este mecanismo suele diferir de un sistema a otro, en una serie de detalles que afectan a la semántica y a la sintaxis de ambas operaciones. Dos aspectos importantes a tener en cuenta son:

1.- Designación del emisor y del receptor:

a) *Designación directa*: el emisor identifica siempre al proceso receptor y a la inversa. Suele ser una comunicación segura al ser 1 a 1.
b) *Designación indirecta*: los mensajes se envían y se reciben mediante buzones. Permite asignaciones entre emisor y receptor 1 a 1, 1 a muchos, muchos a 1 y muchos a muchos. El primer receptor que ejecute la operación recibir saca el mensaje del buzón.

2.- Forma de comunicación:

a) *Síncrona*: la comunicación sólo tiene lugar cuando ambas partes quieran el cambio activamente (el emisor queda suspendido hasta que el receptor realice la operación recibir y viceversa). Sólo se emite un mensaje a la vez por un par emisor-receptor. A esta forma de comunicación también se denomina *'cita'* o *'rendezvous'*.
b) *Asíncrona*: el proceso emisor no necesita suspenderse si el receptor no está preparado (forma de actuar 'envía y olvida'). El proceso receptor sí se bloquea si el emisor no ha enviado el mensaje.

Llamadas a procedimientos remotos (RPC)

Se basa en utilizar mensajes aunque aporta algo más. Su objetivo es conseguir que se ejecute un procedimiento remoto como si fuese local (transparencia para el usuario). Las primitivas de bajo nivel son suficientes para programar cualquier tipo de interacción entre procesos utilizando paso de mensajes. Para programar interacciones cliente/servidor, ambos procesos, cliente y servidor, ejecutan dos sentencias de paso de mensaje: el cliente un *send* (enviar) seguido de un *receive (recibir)*, y el servidor un *receive* seguido de un *send*. Como este tipo de interacción es muy frecuente, se han propuesto sentencias de más alto nivel que directamente la soportan. Éstas se llaman sentencias de *llamada a procedimiento remoto* debido al interfaz que presentan: un cliente "llama" a un procedimiento que es ejecutado por un servidor en una máquina potencialmente remota.

Cuando se utilizan llamadas a procedimiento remoto, un cliente interactúa con un servidor en términos de una sentencia de llamada. Ésta sentencia tiene una forma similar a la usada para una llamada a procedimiento en un lenguaje secuencial:

```
call servicio (argumentos_valor; argumentos_resultado).
```

El *servicio* es realmente el nombre de un canal. Si se utiliza designación directa, *servicio* designa al proceso servidor; si se utiliza designación indirecta (por puerto o por buzón), *servicio* podría designar la clase de servicio requerido. La llamada remota se ejecuta como sigue: los *argumentos_valor* son enviados al servidor apropiado, y el proceso llamador es retrasado hasta que el servicio ha sido realizado y los resultados han sido devueltos y asignados a los *argumentos_resultado*. Tal llamada podría ser traducida en un *send*, inmediatamente seguido de un *receive*. El cliente no puede olvidarse de esperar los resultados del servicio requerido.

IV. Glosario de términos.

Acción funcional

Acción realizada por un agente que se encuentra en la parte funcional del sistema.

Acción estructural (o acción de evolución)

Acción realizada por un agente que se encuentra en la parte estructural del sistema.

Acción compleja

Conjunto de acciones (simples o complejas) que son realizadas en un orden determinado y por un conjunto de agentes para llevar a cabo una tarea compleja. Su ejecución debe llevarse a cabo como si fuera una acción atómica.

Acción simple

Acción realizada por un único agente.

Agente

Componente activo de un sistema software. Realiza acciones para el sistema. Es autónomo, independiente y siempre está preparado para trabajar, si es que las condiciones de su entorno lo permiten.

Agente elemental (AE)

Agente con una estructura básica utilizado para la creación agentes mediante la operación de clonación.

Agente hijo

Agente que está agregado, al menos, a otro agente, su agente padre.

Agente padre

Agente que tiene, al menos, otro agente agregado a él. Representa un nodo intermedio de la jerarquía de agentes que describe a un sistema.

Agente simple

Agente que no tiene agentes agregados (agentes hijos). Representa un nodo hoja de la jerarquía de agentes que describe a un sistema.

Agentes hermanos

Aquellos agentes que se encuentran en el mismo nivel de la jerarquía de agentes que define un sistema. Son agentes que están agregados al mismo agente (padre).

Arquitectura *blackboard*

Estructura de datos central, llamada repositorio, donde se almacena una información que será compartida, leída o escrita, por una colección de componentes independientes.

Cola de espera

Estructura de datos donde el agente *Meta-Agent* almacena temporalmente los mensajes de activación de acciones estructurales enviados por el *Controller*.

Concurrencia inter-agente

Situación en la que dos acciones de distintos agentes pueden estar realizándose simultáneamente en el sistema, es decir, su ejecución se solapa en el tiempo.

Concurrencia intra-agente

Realización simultánea de dos acciones del mismo agente o incluso de dos ejecuciones distintas de la misma acción.

Controller

Agente que se encarga de controlar y gestionar la historia del sistema, activar a los agentes y evaluar las pre-condiciones de las acciones de los agentes cuando éstos lo soliciten.

Coordinator

Entidad que comprueba y decide si se puede evaluar en el instante actual una determinada pre-condición.

Estímulo

Señal procedente del entorno (usuario o desarrollador) o de otro agente y que pretende desencadenar la realización de una o más acciones.

Estímulo consumible

Un estímulo consumible es aquel que sólo puede ser utilizado en la evaluación de una pre-condición que ha dado como resultado *true*.

Evaluator

Forma parte del *Controller* y se encarga de realizar consultas en la historia y de evaluar las condiciones asociadas a las acciones por orden de los agentes.

Evolución

Capacidad de cambiar la estructura y funcionamiento de un sistema a lo largo del tiempo.

Fórmula

Expresión escrita en un lenguaje basado en la Lógica Temporal de Predicados y que se usa para modelar una pre-condición.

Funcionamiento

Comportamiento especificado por la secuencia de acciones que realiza un sistema o un agente debido a su estructura en cada momento.

Historia Funcional del Metasistema (MFH)

Almacena las ocurrencias de acciones funcionales realizadas en el Metasistema y los estímulos recibidos a través de su interfaz de acción.

Historia Estructural del Metasistema (MSH)

Almacena las ocurrencias de acciones estructurales realizadas en el Metasistema. Está vacía ya que el Metasistema no evoluciona.

Historia Funcional del Sistema (SFH)

Almacena las ocurrencias de las acciones funcionales realizadas por los agentes del sistema y los estímulos recibidos a través de la interfaz de acción o procedentes de otros agentes.

Historia Estructural del Sistema (SSH)

Conjunto de ocurrencias de acciones estructurales realizadas por el Sistema Genético y estímulos recibidos por la interfaz de evolución o por efecto de la realización de una acción estructural.

Interfaz de acción

Permite comunicar al sistema software con el sistema de información en el que está incluido. A través de esta interfaz, el sistema software recibe estímulos que indican que se deben realizar determinadas acciones.

Interfaz de evolución

Permite que el equipo de desarrollo haga evolucionar estructuralmente al sistema por medio de un sistema software especial llamado *Metasistema*.

Invariante

Declaración semántica sobre una propiedad del sistema que debe verificarse durante toda su vida. También llamado restricción de integridad.

Metasistema (MS)

Sistema software utilizado por el equipo de desarrollo con el fin de hacer evolucionar a un determinado sistema software.

Ocurrence Receiver (OcuR)

Forma parte del *Controller* y se encarga de recibir y almacenar ocurrencias y estímulos en la historia del sistema así como de activar a los agentes.

Ocurrencia de acción

Constancia de la realización de una acción en el sistema.

Ocurrencia (de acción) consumible

Es aquella ocurrencia que sólo puede ser utilizada en la evaluación de una pre-condición si ésta ha dado como resultado *true*.

Post-condición

Restricción que debe ser cierta cuando se finaliza la realización de una acción.

Pre-condición

Restricción que debe ser cierta para que se inicie la realización de una acción.

Pre-condición p

Pre-condición asociada a una acción simple o compleja que es capaz de realizar un agente. Se evalúa sobre la historia del *Controller* del agente padre asociado a dicho agente.

Pre-condición s

Pre-condición asociada a una acción transaccional. Se evalúa sobre la historia del *Controller* del TM creado para gestionar la transacción donde se encuentra la acción transaccional. Se construye tomando como base el orden de la acción transaccional dentro de la definición de la transacción.

Pre-condición t

Pre-condición asociada a una acción transaccional. Representa a las condiciones especiales que deben cumplirse para activar la acción transaccional y que se definen por el hecho de que la acción se ejecuta dentro de la transacción. Se evalúa sobre la historia del *Controller* del TM, al igual que en la pre-condición s.

Query

Consulta particular que se realiza sobre una historia con el objeto de conocer una lista de ocurrencias y/o estímulos cuyos atributos toman un valor especificado mediante una condición.

Relación de colaboración

Relación establecida entre un agente y sus agentes hermanos con el fin de realizar entre todos una tarea más compleja.

Relación de agregación

Relación establecida entre un agente y su agente padre, para el cual trabaja.

Sistema Genético (SG)

Agente de la parte estructural de un Sistema Software que lleva a cabo las acciones estructurales enviadas por el Metasistema a través de la interfaz de evolución del agente donde se encuentra.

Sistema

Agente que no está agregado a ningún otro agente (no tiene padre). Representa el nodo raíz de la jerarquía de agentes que define a un SS.

Sistema Software (SS)

Colección de agentes que son responsables de la realización de una o más acciones del sistema.

Subsistema (o sistema) de Decisión (SD)

Parte de un sistema software que contiene toda la lógica necesaria para realizar comprobaciones sobre la historia del sistema

Tabla de Acciones

Información mantenida por el OcuR que describe las acciones que es capaz de realizar un agente. Cada elemento lo constituye el nombre de una acción y una referencia a la tabla de pre-condiciones de dicha acción.

Tabla de Acciones Incompatibles

En cada entrada se mantiene una relación de acciones que no pueden realizarse a la vez en el agente/sistema. La construye el equipo de desarrollo. La utiliza el *Evaluator* para decidir si una acción puede o no

iniciarse en el sistema independientemente de que su pre-condición sea cierta.

Tabla de Acciones Iniciadas

Almacena la información de qué acciones han comenzado a realizarse en un sistema o agente y que aún no han finalizado. Esta tabla es gestionada por el *Evaluator* y por el *Ocurrence Receiver*. Cuando una pre-condición de una acción se evalúa a *true*, el *Evaluator* introduce la información de que dicha acción ha comenzado a realizarse. Cuando el *Ocurrence Receiver* recibe la ocurrencia de realización de dicha acción, borra esa información de la TAI.

Tabla de Acciones Pendientes

Almacena la información de qué acciones pertenecientes a la post-condición de la acción estructural que se está realizando por el Metasistema en este momento han comenzado a realizarse y se está esperando su resultado. Esta tabla la usa el *Meta-Agent*.

Tabla de Agentes

Información que mantiene el OcuR de un agente de sus agentes hijos. Cada elemento tiene dos partes: el nombre de un agente (hijo) y una referencia a la tabla de acciones correspondiente.

Tabla de Conflictos (TC)

Almacena una relación de las acciones que se encuentran en conflicto. Existe un conflicto entre acciones cuando sus pre-condiciones no se pueden evaluar concurrentemente.

Tabla de Funcionamiento

Almacena la lista de acciones, tanto simples como complejas, realizadas por los agentes hijos de un agente junto con su pre-condición *p*. La mantiene y utiliza el *Evaluator* por eficiencia a la hora de realizar la evaluación de pre-condiciones solicitadas por los agentes.

Tabla de Pre-condiciones

Información que mantiene el OcuR de un agente de las ocurrencias de acción y estímulos que intervienen en la pre-condición de una acción que puede llevar a cabo un agente hijo. Cada elemento representa el nombre de una acción o estímulo que aparece en la pre-condición de dicha acción.

Tabla de Post-condiciones

Información que mantiene el agente *Meta-Agent* de las post-condiciones asociadas a las acciones estructurales. Sólo almacena la información de aquellas acciones estructurales que tengan una post-condición distinta a *true*.

Tabla de Transacciones

Relación de transacciones. Para cada transacción, se especifica el nombre de ésta, el estímulo generado cuando se inicia , la descripción de la transacción y una lista de las acciones transaccionales junto con su pre-condición *t*.

Tiempo de realización

Atributo que tiene toda acción y estímulo y que determina el instante de tiempo en el que la acción fue realizada o el estímulo fue recibido.

Transacción

Véase acción compleja.

Transaction Manager (TM)

Entidad creada para gestionar la ejecución de una determinada transacción. Una vez que la transacción finalice con o sin éxito, desaparece del sistema.

Bibliografía

[ACME] Página web del proyecto ACME http://www-2.cs.cmu.edu/~acme/

[Addy98] E. A. Addy. “Report from the First Annual Work-shop on Software Architectures in Product Line Acquisitions”. ACM SIGSOFT Software Engineering Notes vol 23 no 3 May 1998

[AgentWeb] UMBC AgentWeb. http://www.csee.umbc.ed/aw/

[Agha93] G. Agha, P.Wegner, A. Yonezawa. “Research Directions in Concurrent Object-Oriented Programming”. The MIT Press, 1993.

[Agresti86] W.W.Agresti. Tutorial: “New Paradigms for Software Development”. Los Angeles. CA IEEE Computer Society Press, 1986.

[Allen98] R.Allen, R.Douence; D.Garland. “Specifing and Analyzing Dynamic Software Architectures”. Proceedings on Fundamental Approaches to Software Engineering, Lisbon, Portugal, 1998.

[Anaya96] A. Anaya, M.J. Rodríguez, P. Paderewski, J. Parets. “Time in the evolution and functionality of information systems”. Jornadas de trabajo en Ingeniería del Software (JIS’96). Sevilla, 1996.

[Anaya97] A. Anaya, M.J. Rodríguez, J. Parets. “Representation and management of memory and decision in evolving software systems”. In: F.Pichler, R. Moreno Díaz (eds.): Computer Aided Systems Theory- EUROCAST'97 Lecture notes in Computer Science 1333 Berlin: Springer-Verlag 1997.

[Anderson00] J. Andersson. “Issues in Dynamic Software Architectures”. Proceedings of the Fourth International Software Architecture Workshop (ISAW’4) en conjunción con ICSE’2000, Junio 2000.

[Andrade01] Andrade, L.; Gouveia, J.; Koutsoukos,G.; Fiaeiro, J.L. “Coordination Contracts, Evolution and Tools”. Proceedings of the Workshop on Formal Foundation of Software Evolution. Lisboa, Portugal 2001.

[Andrade02] L.Andrade, J.F.Fiadeiro, J.Gouveia, G.Koutsoukos. “Separating computacion, coordination and configuration”. Journal of Software Maintenance and Evolution: Research and Practice, 14:353-369, 2002.

[Andreoli96] J.M. Andreoli, H.Gallaire, R.Preschi. “Rule-Based Object Coordination”. First International Conference on Coordination Models, Languages and Applications (Coordination’96). LNCS 1061 pp.1-13, 1996.

[Andrews83] G.R. Andrews, F.B. Scheneider: “Concepts and Notations for Concurrent Programming”. Computing Surveys. ACM. 1983.

[AOSD] AOSD (Aspect Oriented Software Development) Steering Committee. http://aosd.net/

[Appleton] http://www.enteract.com/~bradapp/docs/patterns-intro.html – Patterns and Software: Essencial Concepts and Terminology. Brad Appleton.

[Arbab93] F.Arbab, I.Herman, P.Spilling. “An Overview of Manifold and its Implementation”. Concurrency: Practice and Experience vol.5, pp.23-70, 1993.

[Bacon98] J. Bacon. “Concurrent Systems”. Addison Wesley, 2ª edición, 1998.

[Banerjee87] Banerjee, J., Kim, W., Kim, H.K., Korth, H.F. “Semantics and implementation of schema evolution in object-oriented databases”, In Proc. Of ACM-SIGMOD International Conference on Management of Data, San Francisco, 1987.

[Belady76] L.A. Belady; M.M. Lehman. “A Model of Large Program Development”, IBM SYST.J. Vol. 15.3, 225-252. (1976).

[Berzins93] Berzins , V., Luqi, Yehudai, A:. “Using Transformations in Specification-Based Prototyping”. IEEE Trans.S.E. Vol. 19.5, 436-452, 1993.

[Boehm86] Boehm, B.W. “A Spiral Model of Software Development and Enhancement”. ACM SIGSOFT S.E.NOTES. AUGUST Vol. 11, 14-24, 1986.

[Boehm98] B. W. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, R. Madachy. “Using the WinWin Spiral Model: A Case Study”, IEEE Computer. pp: 34-44, 1998.

[Booch94] G. Booch. “Object-Oriented Analysis and Design with Applications”. Addison Wesley, 1994.

[Booch99] G. Booch, J. Rumbaugh, I. Jacobson. “The Unified Modeling Language User Guide”. Addison-Wesley. 1999.

[Brenner98] W.Brenner, R.Zarnekow, H.Wittig. “Intelligent Software Agents”. Sprinter, 1998.

[Briot98] J.P. Briot, R.Guerraoui, K.P. Lohr. “Concurrency and Distribution in Object-Oriented Programming”. ACM Computing Surveys, Vol. 30, No. 3, September, 1998.

[Brustoloni91] J.C. Brustoloni. “Autonomous Agents: Characterization and Requirements”. Carnegie Mellon Technical Report CMU-CS-91-204, Pittsburgh: Carnegie Mellon University.

[Cabri00] G. Cabri, L. Leonardi, F. Zambonelli. “MARS: a Programmable Coordination Architecture for Mobile Agents”. IEEE Internet Computing, 2000.

[Canal99a] C.Canal; L.Fuentes; E.Pimentel; J.M. Troya. “Coordinación de Componentes Distribuidos: un Enfoque Generativo Basado en Arquitectura del Software”. IV Jornadas de Ingeniería del Software y Bases de Datos (JISBD’99). 1999. Pp: 443-454.

[Canal99b] C.Canal, E. Pimentel, J.M. Troya. “Specification and Refinement of Dynamic Software Architectures”. En: P.Donohoe (Eds), Software Architecture. Kluwer Academic Publisher, San Antonio USA 1999 Pp. 107-126.

[Carsí99] J.A. Carsí. “OASIS como Marco Conceptual para la Evolución del Software”. Tesis Doctoral. Universidad Politécnica de Valencia. 1999.

[Carreiro89] N. Carreiro, D. Gelernter. “Linda in Context”. Communication of ACM, vol. 32, no.4, 1989.

[Carreiro94] N. Carreiro, D. Gelernter, L. Zuck. “Bauhaus Linda”. Object-Based Models and Languages for Concurrent System, Bologna, Italy, LNCS 924, Springer Verlag. Pp: 66-76, 1994.

[Casais90] Casais, E. “Managing Class Evolution in Object-Oriented Systems. In: Object Management”. Tsichritzis, D. GENEVE. Centre Universitaire d'Informatique. 133-195 (1990).

[Choices] Sistema Operativo Choices http://choices.cs.uiuc.edu/choices/

[Ciancarini96a] P.Ciancarini. “Coordination Models and Laguanges as Software Integrators”. ACM Computing Surveys, vol. 28 no. 2, 1996.

[Ciancarini96b] P.Ciancarini, D. Rossi. “Jada - Coordination and Communication for Java Agents”, Proceedings of MOS’96 (Second International Workshop on Mobile Object Systems: Towards the Programmable Internet. LNCS 1222, Springer Verlag, Pp. 213-228, 1996.

[CMU] Carnegie Mellon University. “How Do You Define Software Architecture?” http://www.sei.cmu.edu/architecture/definitions.html

[Coad95] P. Coad, D. North, M. Mayfield. “Object models: strategies, patterns, and applications”. Englewood Cliffs, New Jersey. Prentice-Hall, 1995

[COP] Component-Oriented Programming en http://www.math.tau.ac.il/~guy/COP/

[Coplien] http://www.research.att.com/orgs/ssr/people/cope – Jim Coplien’s Home Page.

[CORBA] Object Management Group. CORBA Specification. http://www.omg.org/

[Corchuelo99] R. Corchuelo Gil. “Prototipado de Especificaciones de Sistemas Distribuidos Basados en Restricciones. Aplicación al lenguaje TESORO”. Tesis Doctoral. Universidad de Sevilla. 1999.

[Corkill91] D.D.Corkill. “Blackboard Systems”. AI Expert 6 (9) : 40-47, september 1991.

[CSP] CSP y Occam. http://www.comlab.ox.ac.uk/archive/csp.html

[Darwin] Proyecto Darwin. http://javalab.cs.uni-bonn.de/research/darwin/project.html

[De Nicola98] R. De Nicola, G.L. Ferrari, R. Pugliese. “KLAIM: A Kernel Language for Agents Interaction and Mobility”. IEEE Transaction on Software Engineering, 24(5). Pp. 315-330, 1998.

[Denti99] E. Denti, A. Omici. “An Architecture for Tuple-based Coordination of Multi-Agent Systems”. Software – Practice & Experience, 29(12), pp.1103-1121, 1999.

[DFKI] The German Research Center for Artificial Intelligence GmbH http://www.dfki.de/dfki.html

[Diagne97] A. Diagne. “Architectural Concepts for Agent Paradigm: A Way to Separate Concerns in Open Distributed Systems” Proceeding of the 2nd Workshop on Formal Methods for Open Object based Distributed Systems (FMOODS’97). Chapman & Hall(Ed.), Canterbury, UK, 1997. Pp: 387-398

[Duran99] A. Durán, B. Bernárdez, M. Toro, A. Ruíz. “Elicitación de Requisitos de Usuario mediante plantillas y patrones de requisitos”. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos (JISBD’99). pp:183-194. 1999.

[Ferber99] J.Ferber. “Multi-Agent Systems, an introduction to distributed artificial intelligence”. Addison-Wesley, 1999.

[Ferrer02] J.Ferrer; A.Lorenzo, I.Ramos, J.A.Carsí. “Dinamismo en arquitecturas y sistemas multiagente”. IDEAS 2002 5º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software. La Habana, Cuba.

[FIPA] The Foundation for Intelligent Physical Agents. http://www.fipa.org

[Fisher91] A. S. Fisher. “CASE : using software development tools”. Wiley Professional Computing. John Wiley 1991.

[Florijn96] G.Florijn, T.Bessamusca, D.Greefhorst. “Ariadne and HOPLa: Flexible Coordination of Collaborative Process”. First International Conference on Coordination Models, Languages and Applications (Coordination’96). LNCS 1061 pp.197-214, 1996.

[Franklin] Stan Franklin. “Coordination without Communication” http://www.msci.memphis.edu/~franklin/coord.html

[Franklin96] S.Franklin, A.Graesser. “Is it an Agent, or just a Program?”. Proceedings of the Third International Workschop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[Freeman99] E. Freeman, S. Hupfer, K. Arnold. “JavaSpaces: Principles, Patterns, and Practice”. The Jini Technology Series. Addison-Wesley, 1999.

[FrØlund93] S. FrØlund, G. Agha. “Abstracting Interactions Based on Message Sets”. Seventh European Conference on Object-Oriented Programming (ECOOP’93). LNCS 707, Springer Verlag, pp. 346-360, 1993.

[Fukuda96] M. Fukuda, L.F. Bic, M.B. Dillencourt, F.Merchant. “Intra- and Inter-Object Coordintation with MESSENGERS”. First International Conference on Coordination Models, Languages and Applications (Coordination’96). LNCS 1061 pp.179-196, 1996.

[Gabbay95] D.M. Gabbay, M. Hodkinson, I., Reynolds. "Temporal logic. Mathematical Foundations and Computational Aspects, Volume 1. Oxford Science Publications. Oxford Logic Guides:28, 1995.

[Gelernter92] D. Gelernter, N. Carreiro. “Coordination Languages and their Significance”. Communication of ACM, vol. 35, no.2, 1992.

[Gamma94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. “Design Patterns. Elements of Reusable Object-Oriented Software”. Addison-Wesley. 1994.

[García01] L.García Cabrera. “SEM-HP: Un modelo sistémico evolutivo y semántico para el desarrollo de sistemas hipermedia”. Tesis doctoral. Universidad de Granada, 2001.

[Garlan97] D.Garlan; R. Monroe; D.Wile. “ACME: An Architecture Description Interchange Language”. Proceedings of CASCON 97, November 1997.

[Gomez00] J.J. Gómez Sanz. “Termostatos y Agentes”. Simposio Español de Informática Distribuida, SEID 2000.

[Guessoum99] Z. Guessoum; J.P. Briot. “From Active Objects to Automous Agents”. IEEE Concurrency 7(3): 68-76, November 1999.

[Heckel01] Heckel, R; Engels, G. Graph. “Transformation as a Meta Language for Dynamic Modelling and Model Evolution. Formal Foundations for the Evolution of Hypermedia Systems”. 5th European Conference on Software Maintenance and Reengineering, Workshop on FFSE. IEEE Press. Lisbon, Portugal, March 42-47, 2001.

[Henderson90] Henderson-Sellers, B., Edwards, J.L. “The Object-Oriented Systems Life Cycle”. CACM 33.9. p.143-159, 1990.

[Holzbacher96] A.A. Holzbacher. “A Software Environment for Concurrent Coordinated Programming”. First International Conference on Coordination Models, Languages and Applications (Coordination’96). LNCS 1061 pp.249-266, 1996.

[Hurtado02] M.V.Hurtado Torres. “Un modelo de integración evolutivo entre sistemas de información y sistemas de ayuda a la decisión”. Tesis doctoral. Universidad de Granada, 2002.

[ISBC] Ingeniería del Software Basada en Componentes, http://www.um.es/giisw/isbc/ página web en España. http://www.um.es/giisw/isbc/ página web en España

[ISR] Institute for Software Research. Software Architecture Research. http://www.isr.uci.edu/architecture/

[Jacobson00] I. Jacobson, G. Booch, J.Rumbaugh. “El Proceso Unificado de Desarrollo de Software”. Addison-Wesley, 2000.

[JavaBeans] JavaBeans http://java.sun.com/beans/

[Jennings96] N. Jennings, M. Wooldridge. “Software Agents”. IEE Review, january 1996, pp 17-20.

[Jennings99] N.R. Jennings; M. Wooldridge. “Agent-Oriented Software Engineering”. Proceedings of the 9th European Workshop on Modelling Autonomous Agents in the Multi-agents World (MAAMW99). Springer Verlag, vol 1647. 1999.

[Jensen96] K. Jensen, “Coloured Petri Nets. Basic concepts, analysis methods and practical use”. Volume 1, 2 y 3. Springer Verlag, Berlin, 1996.

[Kielmann96] T. Kielmann. “Designing a Coordination Model for Open Systems”. First International Conference on Coordination Models, Languages and Applications (Coordination’96). LNCS 1061 pp.267-284, 1996.

[Kozaczynski92] Kozaczynsky, W., Ning, J., Engberts, A. “Program Concept Recognition and Transformation”. IEEE Trans.S.E. 18,12. p.1065-1075, 1992.

[Kramer90] J. Kramer, J. Magee, A. Finkelstein. “A Constructive Approach to the Design of Distributed Systems”. 10th International Conference on Distributed Computing Sustems(ICDCS’90), IEEE Press, pp.580-587, 1990.

[Krasner88] G.E.Krasner, S.T.Pope. “A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1 (3). 1988.

[Lauder98] A. Lauder, S. Kent. “Precise Visual Specification of Design Patterns”. In Procs. ECOOP’98

[Le Moigne90] J..L. Le Moigne. “La Théorie du systéme général. Thórie de la modelisation”. Paris: Presses Universitaires de France 1977-1983-1990.

[Lea] http://gee.cs.oswego.edu/dl/ca/ca/ca.html – Doug Lea: Christopher Alexander: An Introduction for Object-Oriented Designers.

[Lehman80] M.M. Lehman. “Programs, Life Cycles and the Laws of Software Evolution”. PROCEED.IEEE 68.9, PP. 1060-1076, 1980.

[Lehman00] M.M. Lehman, J. Ramil. “Towards a Theory of Software Evolution – And its Practical Impact”. Princeiples of Software Evolution ISPSE 2000. IEEE Computer Society, pp. 2-13, 2000.

[Lehman01a] Lehman, M.M.; Ramil, J.F. “Evolution in Software and Related Areas”. 4th International Workshop on Principle4s of Software Evolution (IWPSE 2001). ACM, 2002. Pp: 1-16.

[Lehman01b] Lehman, M.M.; Ramil, J.F. “An Approach to a Theory of Software Evolution”. 4th International Workshop on Principle4s of Software Evolution (IWPSE 2001). ACM, 2002. Pp: 70-74.

[Lehman02] M.M.Lehman, G.Kahen, J.F.Ramil. “Behavioural modelling of long-lived evolution processes – some issues and an example”. Journal of Software Maintenance and Evolution: Research and Practice, 14:335-351, 2002.

[Letelier98] Letelier, P.; Ramos, I.; Sánchez, P.; Pastor, O. ”OASIS 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto”. Universidad Politécnica de Valencia (SPUPV-98.4011), 1998.

[Lieberherr93] Lieberherr, K.J., Xiao, C. “Object-Oriented Software Evolution”. IEEE Trans.S.E., Vol. 19.4, 313-343, 1993.

[Lieberherr94] Lieberherr, K.J., Silva-Lepe, I., Xiao, C. “Adaptative Object-Oriented Programming Using Graph-Based Customization”. CACM Vol. 37.5, 94-101, 1994.

[Lieberherr96] Lieberherr, K.J. “Adaptative Object-Oriented Software: The Demeter Method with Propagation Patterns”. PWS Publishing Company, Boston. 1996.

[Lorenzo01] A.Lorenzo, I.Ramos, J.A.Carsí, J.Ferrer. “Hacia un modelo de Arquitectura Software: Un caso de estudio”. I Jornadas Dolmen, pp. 1-10, Sevilla, 2001.

[Luckham95] D.C.Luckham; J.Vera. “An Event-Based Architecture Definition Language”. IEEE Transaction on Software Engineering. 12(9):717-734, 1995.

[Mederos98] J. Mederos, J. García, J. Galve-Francés. Patterns for Event Notification. The Notifier and the Interest Publisher. JIS’98 in Murcia, Spain. 1998.

[Medvidovic97] N.Medvidovic; R.N.Taylor. “A Framework for Classifying and Comparing Architecture Description Languages”. In Software Engineering, ESEC/FSE’97, vol. 1301 of Lectures Notes in Computer Science, pp. 60-76, 1997.

[Mens01] Mens, T. “Transformational Software Evolution by Assertions. Formal Foundations for the Evolution of Hypermedia Systems”. 5th European Conference on Software Maintenance and Reengineering, Workshop on FFSE. IEEE Press. Lisbon, Portugal, March (2001) 67-74.

[Meyer97] B.Meyer. “Object-Oriented software construction”. Prentice-Hall. 2d. ed. 1997.

[Milenkovic94] M.Milenkovic. “Sistemas Operativos. Conceptos y Diseño”. 2ª ed. Mc-Graw Hill, 1994.

[Minsky94] N.H. Minsky, J. Leichter. “Law-Governed Linda as a Coordintion Model”. Object-Based Models and Languages for Concurrent System – Proceedings of the ECOOP’94, Bologna, Italy, LNCS 924, Springer Verlag. Pp: 125-145, 1994.

[Molina02] F.Molina Ortiz, L.García Cabrera, N.Medina Medina, M.V.Hurtado Torres. “Editor de Estructuras Conceptuales Evolutivas: Consideraciones Prácticas”. Actas de las III Jornadas de trabajo Dolmen. Pp: 77-82, El Escorial, Madrid, 2002.

[Müller96] Jörg P. Müller. “The design of intelligent agents: a layered appproach”. Lectures Notes in Artificial Intelligent, vol. 1177 Springer-Verlag 1996.

[Multiagent] Página web de sistemas multi-agentes. http://www.multiagent.com

[Murillo99] J.M.Murillo, J.Hernández, F.Sánchez, L.A.Alvarez. “Coordinated Roles: Promoting Reusability of Coordinated Active Objects Using Event Notification Protocols”. Third International Conference COORDINATION’99. Springer Verlag, LNCS 1594, 1999

[Murillo01] J.M. Murillo Rodríguez. “Coordinated Roles: un modelo de coordinación de objetos activos”. Tesis Doctoral. Universidad de Extremadura. 2001.

[Nierstrasz95] O.Nierstrasz, T.Meijler. “Research directions in software composition”. ACM Computer Surveys, 27(2), pp.262-264, 1995.

[NIST] National Institute of Standars and Technology (NIST). Descripción de ADLs. http://www.itl.nist.gov/div897/ctg/adl/adl_info.html

[Omici99] A. Omici, F. Zambonelli. “Coordination for Internet Application Development”. Journal of Autonomous Agents and Multi-Agent Systems, 2(3), september 1999. Special Issue on Coordination Mechanism and Patterns for Web Agents.

[Omici01] A. Omici, F. Zambonelli, M. Klusch and R.Tolksdorf (editors) “Coordination of Internet Agents. Models, Technologies, and Applications”. Springer-Verlag, 2001.

[Paderewski99a] Paderewski, P.; Parets-Llorca, J.; Anaya, A.; Rodríguez, M.J.; Sánchez, G.; Torres, J.; Hurtado, M.V. “A Software Development Tool for Evolutionary Prototyping of Information Systems”. IMACS/IEEE CSCC'99. In: Computers and Computational Engineering in Control, Electric and Computer Engineering Series, pp.347-352. Ed. Word Scientific and Engineering Society Press, 1999.

[Paderewski99b] P.Paderewski; J.Parets. “Un patrón de activación de objetos activos”. JISBD’99. 1999.

[Paderewski00] P.Paderewski, M.J.Rodríguez, F.Molina, A.Anaya, M.V.Hurtado, J.Parets. “Subsistema de Acción y Subsistema de Evolución en HEDES”. V Jornadas de Trabajo MENHIR. Pp. 207-218, Marzo de 2000.

[Paderewski02a] P.Paderewski Rodríguez, M.J.Rodríguez Fórtiz, J.Parets Llorca. “Una arquitectura Dinámica y Evolutiva para Agentes Cooperativos: Gestión de Transacciones”. II Jornadas de Trabajo DOLMEN. Pp: 69-80, Marzo de 2002.

[Paderewski02b] P.Paderewski Rodríguez, M.J.Rodríguez Fórtiz, J.Parets Llorca. “Estructura y Evolución en los Sistemas Software basados en Agentes”. III Jornadas de Trabajo DOLMEN. Pp: 83-88, Noviembre de 2002.

[Paderewski03a] P.Paderewski-Rodríguez, M.J.Rodríguez-Fórtiz, J.Parets-Llorca. “An Architecture for Dynamic and Evolving Cooperative Software Agents”. Computer Standards & Interfaces. Vol 25/3 pp. 261-269. In Press (2003). Ed. Elsevier.

[Paderewski03b] P.Paderewski-Rodríguez, J.J.Torres-Carbonell, M.J.Rodríguez-Fórtiz, N.Medina-Medina, F.Molina-Ortiz. “A Software System Evolutionary and Adaptive Framework. Application to Agent Based Systems”. Aceptado en WASA’03. Pendiente de celebración en Las Vegas, USA. Junio 2003.

[Papadopoulos98] G.A.Papadopoulos, F.Arbab. “Coordination Models and Languages”. Advances in Computers, Academic Press, vol. 46: The Engineering of Large Sytems, pp. 329-400, 1998.

[Parets94] J.Parets, A.Anaya, M.J.Rodríguez, P.Paderewski. “A representacion of software systems evolution based on the Theory of the General System”. En: Pichler, F.; Moreno, R. (eds.): Computer Aided Systems Theory. Pp: 96-110. Springer-Verlag, 1994.

[Parets95] J. Parets Llorca. “Reflexiones sobre el proceso de concepción de sistemas complejos: MEDES: un método de especificación, desarrollo y evolución de sistemas software”. Tesis Doctoral. Universidad de Granada 1995.

[Parets96] J. Parets-Llorca. "The evolution of Software Systems: a framework for modelling organizational evolution". Actas de Computational Engineering in Systems Applications. pp: 846-851. 1996.

[Parets98] J.Parets, M.J.Rodríguez, P.Paderewski, A.Anaya, G.Sánchez. “Prototipado evolutivo de software: Diseño arquitectónico y prototipo de una herramienta CASE”. Repor interno, LSI-98-2.

[Parets99a] Parets, J.; Rodriguez, M.J.; Paderewski, P.; Anaya, A. “HEDES: A System Theory based tool to support evolutionary Software Systems”. EUROCAST'99.

[Parets99b] J. Parets, J. Carsí, J.H. Canós, A. Anaya, M.V. Hurtado, M.C. Penadés, P.Paderewski, I. Ramos, M. J. Rodríguez. “La evolución de modelos en el desarrollo de software: los enfoques OASIS y MEDES”. Actas de las IV Jornadas de Ingeniería del Software y Bases de Datos, JISBD’99. pp: 207-218.1999.

[Pérez02] J. Pérez, I. Ramos, A. Lorenzo, P. Letelier, J. Jaén. “PRISMA: PlatafoRma OASIS para Modelos Arquitectónicos”. VII Jornadas de Ingeniería del Software y Bases de Datos (JISBD’02]. Pp. 349-360, 2002.

[Peschanski01] F.Peschanski; C.Queinnec; J.P.Briot. “A Typeful Composition Model for Dynamic Software Architectures”. Technical Report , Univ. of Paris VI, 2001 http://www.lip6.fr/reports

[Petrie01] C.Petrie. “Agent-Based Software Engineering”. AOSE 2000, P.Ciancarini and M.J.Wooldridge(Eds.), LNCS 1957, pp.59-75, 2001.

[Picco99] J.P. Picco, A.L. Murphy, G.C. Roman. “LIME: Linda meets Mobility”. Proceedings of the 21th International Conference on Software engineering (ICSE’99). Pp. 368-377. ACM, 1999.

[Pree95] W. Pree. “Design Patterns for Object-Oriented Software Development”. Addison-Wesley. 1995.

[Pressman02] R.S. Pressman. “Ingeniería del Software. Un enfoque práctico”. 5ª edición. McGraw-Hill 2002.

[Rapide] The Stanford Rapide™ Project. http://pavg.stanford.edu/rapide/

[Rational] Rational Rose/C++ Demo Version 4.0.3 http://www.rational.com/

[Reiss90] S.P. Reiss. “Connecting tools using message passing in the field program development environment”. IEEE Software, july, 1990.

[Riehle96] D. Riehle. “The Event Notification Pattern. Integrating Implicit Invocation with Object-Orientation”. Theory and Practice of Object Systems 2, 1. 1996.

[Rising96] L. Rising. “Design Patterns: Elements of Reusable Architectures”. Annual Review of Communications, Vo.49, 1996.

[Rodríguez99] M.J. Rodríguez , P. Paderewski, Ana Anaya, José Parets. “HEDES: lessons learned from a first prototype”. IV Jornadas de Trabajo MENHIR. Sedano, Burgos. Mayo 1999.

[Rodríguez00a] M.J. Rodríguez. “Evolución del Software: Una Formalización Basada en Lógica Temporal de Predicados y Redes de Petri Coloreadas”. Tesis doctoral. Universidad de Granada. 2000.

[Rodríguez00b] M.J.Rodríguez, J.Parets, P.Paderewski, A.Anaya, M.V.Hurtado. “HEDES: A system theory based tool to support evolutionary software systems”. Lectures Notes in Computer Science, vol. 1798, pp. 450-464, Springer Verlag, 2000.

[Rodríguez01a] M.J. Rodríguez-Fórtiz, P. Paderewski-Rodríguez, L. García-Cabrera, J. Parets-Llorca. “Evolutionary Modelling of Software Systems: its Application to Agent-Based and Hypermedia Systems”. 4th International Workshop on Principles of Software Evolution (IWPSE 2001), pp. 62-69, 2001.

[Rodríguez01b] A.Rodríguez, A. Márquez, M.Toro. “Gestión de la evolución del software. El eterno problema de los *legacy systems*”. Actas del Taller de Evolución del Software. VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD01), pp. 13-25, Noviembre 2001.

[Roscoe98] A.W. Roscoe. “The Theory and Practice of Concurrency”. Prentice Hall, 1998.

[Rowstron97] A.Rowstron, A.Wood. “BONITA: A Set of Tuple Space Primitives for Distributed Coordination”. Proceedings of the 30th Hawaii International Conference on System Sciences. Vol. 1, pp. 379-388, IEEE Computer Society Press, 1997.

[Rowstron98] A.Rowstron. “WCL: a Web Coordination Language”. Word Wide Web Journal, 1(3), pp. 167-179, 1998.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. “Object Oriented Modelling and Design”. Prentice Hall. 1991.

[RUP] Rational Unified Process. http://www.rational.com

[Said01] Said, J; Steegmans, E. “Transformations of Binary relations into Associations and Nested Classes. Formal Foundations for the Evolution of Hypermedia Systems”. 5th European Conference on Software Maintenance and Reengineering, Workshop on FFSE. IEEE Press. Lisbon, Portugal, March (2001) 75-82.

[Sane] http://hillside.net/patterns/ – WWW Patterns Home Page. Aamod Sane.

[Sartipi97] K. Sartipi. “A survey on Software Architecture Domain”. Part of his phD. http://www.swen.uwaterloo.ca/~ksartipi

[Schelfthout02] K. Schelfthout, T.Coninx, A.Helleboogh, T.Holvoet, E.Steegmans, D.Weyns. “Agent Implementation Patterns”. OOPSLA 2002 Workshop on Agent-oriented Methodologies, Seattle, WA USA, Pp. 119-130, November, 2002.

[Shaw96] M. Shaw, D. Garlan: Software Architecture. Perspectives and emergencing discipline. Prentice Hall. 1996.

[Shoham93] Y. Shoham. “Agent-oriented programming”. Artificial Intelligence 60, pp.51-92, Elsevier, 1993.

[Sommerville96] I. Sommerville, G. Dean. “PCL: A Language for Modelling Evolving System Architectures”. Software Engineering Journal, IEEE, pp. 111-121, 1996.

[Sommerville01] I. Sommerville. “*Software Engineering*”, 6ª Edición, Addison-Wesley, 2001.

[Tanenbaum96] A.S.Tanenbaum. “Sistemas Operativos Distribuidos”. Prentice Hall, 1996.

[Tanenbaum98] A.S.Tanenbaum, A.S.Woodhull. “Sistemas Operativos. Diseño e implementación”. 2ª edición. Prentice Hall, 1998.

[Tardieu92]Tardieu, H. “Issues for Dynamic Modelling through Recent Developments in European Methods” In: Dynamic Modelling of Information Systems, II. Sol, H.G., Crosslin, R.L. (Eds.), North-Holland, Amsterdam, 3-23, 1992.

[Tolksdorf96] R. Tolksdorf. “Coordinating Services in open distributed systems with LAURA” First International Conference on Coordination Models, Languages and Applications (Coordination’96). LNCS 1061 pp.386-402, 1996.

[Torres00] Torres-Carbonell, J.J.; Parets-Llorca, J.. “A Formalisation of Evolution of Software Systems”, in Pichler, F.R.; Moreno-Díaz, R.; Kopacek, P. (Eds.) “Computer Aided System Theory – EUROCAST´99”, Springer-Verlag. LNCS 1798. ISSN 0302-9743. mpp 439-449, 2000.

[Torres02] J.J. Torres Carbonell. “Evolución de Sistemas Software. Aplicación de modelos biológicos a la concepción evolutiva de sistemas software”. Tesis Doctoral. Universidad de Granada. 2002.

[Wegner90] P. Wegner. “Concepts and paradigms of object-oriented programming”. ACM OOPS. 1990.

[Wermenlinger01] Wermelinger, M; Lopes, A; Fiadeiro, J. L. “A Graph Transformation Approach to Architectural Run-Time Reconfiguration. Formal Foundations for the Evolution of Hypermedia Systems”. 5th European Conference on Software Maintenance and Reengineering, Workshop on FFSE. IEEE Press. Lisbon, Portugal, 59-66, 2001.

[Weyns02] D.Weyns, T.Holvoet. “A Colored Petri Net for a Multi-Agent Application”. Modeling Components, Objects and Agents, MOCA’02, pp.121-140. Aarhus Denmark, august 26-27. 2002.

[Wooldridge95] M.J. Wooldridge, N.R. Jennings (1995) “Intelligent Agents: Theory and Practice”. The Knowledge Engineering Review 10 (2). 115-152.

[Wooldridge01] M.Wooldridge; P.Ciancarini. “Agent-Oriented Software Engineering: The State of the Art” In P. Ciancarini and M. Wooldridge, editors, Agent-Oriented Software Engineering. Springer-Verlag Lecture Notes in AI Volume 1957, January 2001.

[Wyckoff98] P. Wyckoff, S. McLaughry, T. Lemman, D. Ford. “T Spaces” IBM System Journal, 37(3) pp.454-474, 1998.