(1 181801) xx

CINVESTAV

Centro de Investigación y de Estudios Avanzados del IPN
Unidad Guadalajara

Coordinación Distribuida Basada en Agentes de Sistemas de Manufactura Flexible

Tesis que presenta
José Guadalupe Morales Montelongo

Para obtener el grado de
MAESTRO EN CIENCIAS

En la especialidad de
INGENIERÍA ELÉCTRICA

Guadalajara, Jal.; Noviembre de 2002.

Coordinación Distribuida Basada en Agentes de Sistemas de Manufactura Flexible

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por

José Guadalupe Morales Montelongo
Ingeniero en Computación
Universidad de Guadalajara

Becario de CONACYT
Expediente **143985**

Director de Tesis
Dr. Luis Ernesto López Mellado

CINVESTAV del IPN Unidad Guadalajara, Noviembre de 2002.

Agradecimientos

A Dios.

A Araceli.

A mis padres.

A mi asesor.

A mis compañeros.

Al CONACYT.

Indice General

Introducción

La fuerte competencia en el mercado de consumo requiere que los sistemas de manufactura sean flexibles y adaptables sin realizar grandes inversiones. Esto es debido a los continuos cambios en las líneas de producción, y por la gran diversidad y evolución de los productos. Para lograr esto, se han propuesto algunos tipos de sistemas tales como los Sistemas de Manufactura Flexible (FMS, Flexible Manufacturing System) y los Sistemas Holónicos de Manufactura (HMS, Holonic Manufacturing System) [Bussmann98a] [Bussmann98b] [Ouelhadj98].

Los FMS se componen de máquinas programables que manejan múltiples herramientas. Así, las líneas de producción pueden ser adaptadas reubicando las máquinas (si es necesario) y reprogramándolas.

Los esquemas de manufactura centralizados no pueden adaptarse fácilmente a los continuos cambios en la producción. En éstos sistemas centralizados existe un único sistema de software coordina el proceso de producción dando las instrucciones adecuadas a los dispositivos.

Se han propuesto sistemas que distribuyen el control del proceso de producción sobre varios entes autónomos denominados *agentes*. Cada *agente* es responsable de una parte del proceso, por lo que deben coordinarse para llevar a cabo la tarea asignada globalmente. Estos conjuntos de agentes se denominan *sistemas multiagentes*.

Los sistemas multiagentes distribuidos pueden controlar el proceso de producción además de adaptarse más naturalmente a los cambios demandados, y en caso de fallo no se detiene el sistema completo. El avance logrado en la investigación de los sistemas multiagente y el desarrollo de plataformas de programación orientada a agentes han permitido el desarrollo de sistemas robustos, confiables y tolerantes a fallos basados en esquemas distribuidos. Algunos investigadores han propuesto arquitecturas de sistemas de manufactura con enfoque multiagente [Ouelhadj98] [Bussmann98a] [Bussmann98b] [Bussmann97].

Algunas propuestas proponen estandarizar las plataformas para que los agentes se comuniquen independientemente de su implementación particular y de la plataforma usada en su desarrollo, tal como el estándar FIPA (Foundation for Intelligents Physical Agents) y algunas implementaciones como JADE (Java Agent DEvelopment framework).

Control de FMS

La programación de sistemas de coordinación de actividades se ha convertido en una labor compleja por la explosión combinatoria de estados que se deben tener en cuenta. Así, el problema es cómo programar de manera sencilla y eficiente sistemas de coordinación de actividades para sistemas de manufactura flexible complejos.

El presente trabajo presenta un método para crear sistemas multiagente para el control de tareas de manufactura partiendo de la especificación de funcionamiento del sistema.

Este trabajo propone el uso de una arquitectura basada en un estándar industrial para crear agentes capaces de realizar la coordinación distribuida de actividades de manufactura, así como el desarrollo de una metodología para crear estos agentes de manufactura basados en la arquitectura mencionada, partiendo de la especificación de la tarea global con el enfoque de los sistemas multiagente.

Objetivos

Los objetivos de este trabajo son: (1) proponer una metodología de programación de sistemas distribuidos para la coordinación de actividades usando el paradigma de los

sistemas multi-agente. (2) Crear una plataforma multi-agente para la programación sencilla de agentes de manufactura que tome en cuenta un estándar industrial.

El alcance del trabajo es demostrar la utilización de la metodología y el funcionamiento de la plataforma extendida a través de ejemplos implementados en un ambiente distribuido con varias computadoras.

Organización del trabajo

La tesis está organizada como sigue: en el capítulo 1 se hace una revisión del estado del arte de los sistemas multiagentes, las plataformas y las experiencias de la aplicación del enfoque multiagente a diversas áreas incluida la manufactura.

En el capítulo 2 se describe la primera parte de la metodología de modelado, donde se explica en detalle el proceso de modelar la tarea global y de obtener las actividades que realizará cada agente del sistema de coordinación.

El capítulo 3 presenta una revisión de las características y facilidades de la plataforma Jade, base de la propuesta que es utilizada para la programación de los agentes que coordinan las subtareas del sistema de manufactura.

El capítulo 4 describe la forma de programar los agentes partiendo del subgrafo/subtarea asignado al agente de manufactura.

Por último, en el capítulo 5 se aplica la metodología propuesta usando como ejemplo un sistema de ensamble de piezas. Al final, la metodología permite obtener un sistema multiagente para el control distribuido de la tarea de ensamble.

CAPÍTULO

1

Agentes y Sistemas de Manufactura

En este capítulo se hace una revisión de los trabajos que utilizan la tecnología de agentes para la coordinación de actividades en los sistemas de manufactura flexible.

1.1. Agentes: conceptos básicos y tecnología

1.1.1. Definición de agente

De las definiciones existentes, en este trabajo se adopta la definición de *agente* propuesta por Franklin [Franklin96] en la cual un agente es un sistema de hardware o más usualmente de software el cual es autónomo, social, reactivo y proactivo. Es autónomo en el sentido de que opera sin intervención de personas y que tiene cierto tipo de control sobre sus acciones y su estado interno; es social porque debe interactuar con otros agentes o incluso personas a través de un lenguaje de comunicación de agentes; es reactivo porque percibe su entorno además de que puede responder a los cambios que ocurren en el mismo ambiente; y finalmente, es proactivo porque puede tener un comportamiento basado en metas y por ello puede tomar la iniciativa.

1.1.2. Clasificación de agentes

Se han propuesto múltiples clasificaciones de los agentes según los criterios utilizados [Case01]. Se pueden distinguir distintos tipos de agentes en una taxonomía que

se basa en las capacidades de cooperación, proactividad y adaptabilidad de los agentes inteligentes, los cuales pueden tener algunas o todas de dichas capacidades. En la Figura 1-1 se muestra un diagrama con la taxonomía de agentes. Los *agentes cooperativos* se comunican con otros agentes y actúan según los resultados de la comunicación. Los *agentes proactivos* inician acciones sin intervención del usuario. Los *agentes adaptables* cambian su comportamiento en determinadas situaciones al aprender de experiencias pasadas. Los *agentes personales* son proactivos y atienden a usuarios individuales. Los *agentes colaborativos* son proactivos y cooperan con otros agentes.

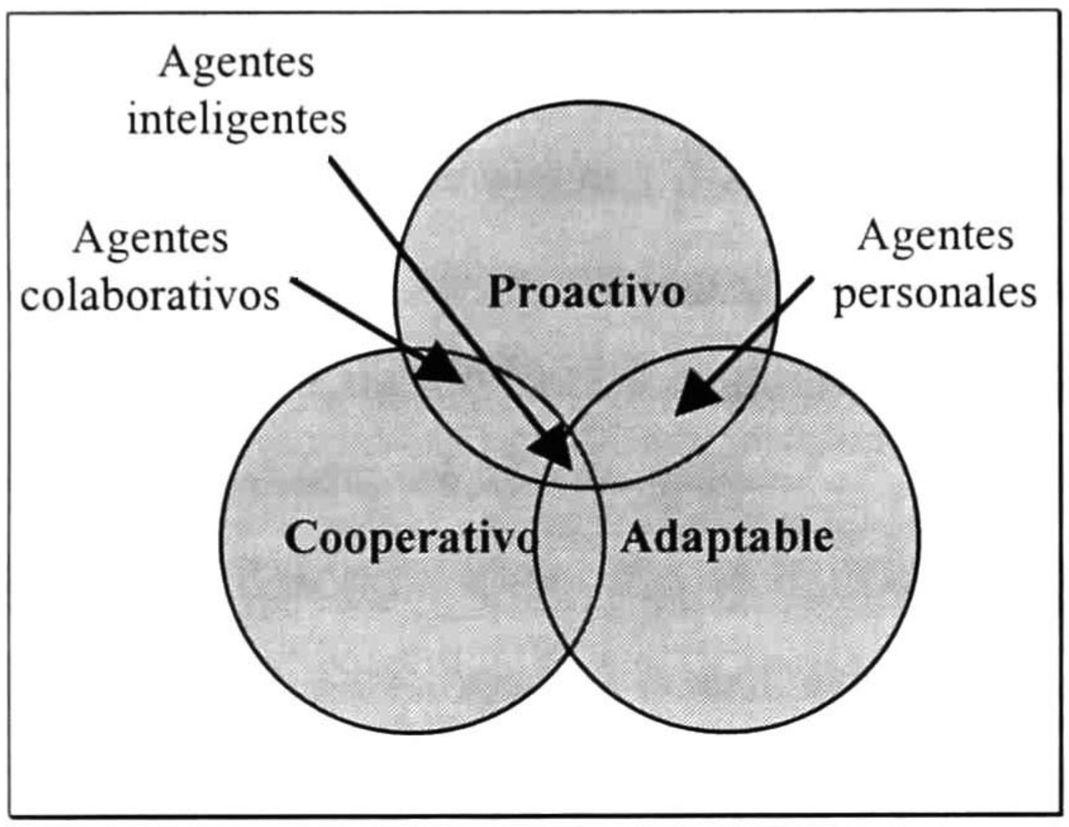

Figura 1-1. Taxonomía de agentes.

Los agentes personales adaptables son una tecnología ideal para buscar información personalizada para el usuario. Los investigadores han desarrollado agentes personales de software para ayudar en la administración de la creciente cantidad de información electrónica disponible. Como estos agentes pueden iniciar tareas sin intervención del usuario, pueden encargarse de tareas en segundo plano tales como búsqueda de información. Igual de importante, algunos agentes aprenden de la experiencia. En el contexto de las *e-comunidades*, aprender más sobre los miembros individuales facilita la actualización del perfil de miembro; con el tiempo, se mejora la exactitud de los datos de la comunidad, incluyendo información sobre documentos, personas y contactos. Los agentes personales producen y consumen información; al compartir su dominio de conocimiento con otros agentes, obviamente sujeto a las limitaciones de privacidad impuestas por el mismo, ellos contribuyen al conocimiento comunitario.

Los agentes colaborativos filtrantes se especializan en promover la interacción entre los miembros de la comunidad. Estos agentes benefician tanto a emisores y receptores porque los usuarios pueden difundir información a aquellos que están interesados en ella sin molestar a otros miembros. Los agentes para búsqueda de contactos pueden localizar miembros con intereses o competencias distintas de tal manera que los miembros pueden encontrar expertos de un subdominio dado u otros miembros con intereses similares a los propios. Los agentes también pueden trabajar en nombre de miembros individuales, protegiéndolos del exceso de información o protegiendo a los miembros expertos de peticiones excesivas.

Al ser los agentes entes sociales, surge el concepto de comunidades o sociedades de agentes. Por ejemplo, cuando un grupo de agentes planificadores intentan acordar una cita entre los usuarios que representan, ellos persiguen una meta común y surge un comportamiento grupal inteligente. Cuando la cita se acordó, los agentes se dispersan y es posible que nunca más vuelvan a formar el mismo grupo.

Ejemplos de los agentes descritos antes son el llamado Bugle, Jasper y ProSearch entre otros, que son parte de una e-comunidad desarrollada por una división de British Telecommunications. Dicha e-comunidad posee un agente administrador de perfiles de los usuarios [Case01].

Bugle es un agente personal que usa la información del perfil para generar un periódico personalizado en web que contiene noticias de interés para el usuario. Lo que se muestra son el encabezado de la noticia junto con un resumen y una liga al artículo completo. Al final de cada artículo completo aparecen una lista de palabras que el agente considera importantes. Los usuarios pueden agregar cualquiera de esas palabras en la categoría de interés apropiada.

Un ejemplo de agente colaborativo filtrante es Jasper. Éste agente ayuda a compartir información, distribuir una página web o un documento a la comunidad sin determinar quiénes pueden estar interesados en el material. Jasper colabora con el agente de perfiles para determinar quiénes deben recibir la información. Además, Jasper sugiere mejoras al perfil del usuario, tal como lo hace Bugle.

El agente filtrante ProSearch trabaja en segundo plano y usa los perfiles para encontrar páginas web que les interesan a los usuarios. El agente además elimina resultados duplicados, páginas que el usuario ya ha visto antes y ligas a páginas inexistentes.

1.1.3. Sistemas Multi-Agente

La investigación en Sistemas Multiagente (MAS, Multi-Agent Systems) se refiere al comportamiento de un conjunto de agentes autónomos posiblemente pre-existentes que ayudan a resolver un problema dado.

Un Sistema Multi-agente puede considerarse como una red de entidades poco acopladas capaces de solucionar problemas; las entidades trabajan conjuntamente para encontrar respuesta a problemas que están más allá de la capacidad y el conocimiento individual de cada entidad. Más recientemente se ha dado al termino sistema multi-agente un significado más general, y se usa para definir todos los tipos de sistemas compuestos por múltiples componentes autónomos que poseen las siguientes características [Jennings98]:

- Cada agente tiene capacidad para solucionar parcialmente el problema.
- No hay un sistema global de control.
- Los datos no están centralizados.
- La computación es asíncrona.

Uno de los factores actuales que promueven la investigación en MAS es la creciente popularidad de Internet, que proporciona la base para un entorno o ambiente abierto donde los agentes interactúan para conseguir sus objetivos individuales o colectivos. Para interactuar en este entorno, los agentes tienen que superar dos problemas: (a) deben ser capaces de encontrarse (los agentes pueden aparecer, desaparecer o moverse en cualquier momento) unos a otros; y (b) deben ser capaces de interactuar.

El avance de la Ingeniería de Software Orientada a Agentes [Durfee01] permite desarrollar sistemas distribuidos muy complejos, y los agentes que los componen deben ser capaces de actuar e interactuar de manera flexible. Un sistema distribuido es por sí mismo muy complejo y es posible obtener resultados similares con otras tecnologías o paradigmas arquitecturales, pero los agentes permiten una mayor flexibilidad al sistema.

1.1.4. Estándares y plataformas

El uso de la tecnología de agentes en la industria depende de que existan herramientas para desarrollo y plataformas que eviten a los desarrolladores tener que programar la funcionalidad básica para cada sistema [Bussmann98a]. Esto supone la existencia y uso de estándares donde se establezca la funcionalidad básica y cómo se presentan.

Actualmente existen intentos para crear estándares para el desarrollo de sistemas basados en agentes, pero no específicamente para desarrollar controladores de sistemas de manufactura basados en agentes. Hasta ahora se ha intentado establecer a KQML (Knowledge Query Manipulation Language, *http://www.cs.umbc.edu/kqml/*) como un lenguaje común de agentes usando KIF (Knowledge Interchange Format, *http://www.csee.umbc.edu/kse/kif*) como formato de contenido, aunque también se ha intentado lo mismo con ACL (Agent Communication Language, *http://www.fipa.org/repository/aclspecs.html*) por parte de FIPA (Foundation for Intelligents Physical Agents, *http://www.fipa.org*). Se han usado algunos estándares ya existentes para los sistemas basados en objetos, tales como CORBA (Common Object Request Broker Architecture, *http://www.corba.org*) para la comunicación entre agentes y STeP (Stanford Temporal Prover, *http://www-step.stanford.edu*) para proporcionar semánticas de mensajes en aplicaciones de manufactura.

Recientemente, FIPA, organización creada en 1996, produjo la versión 0.1 del conjunto de especificaciones FIPA 98. El objetivo de FIPA es promover el desarrollo de especificaciones de tecnologías genéricas de agentes que maximicen la interoperabilidad dentro del sistema y entre aplicaciones basadas en agentes. Las especificaciones generadas se pretenden ser vistas como un conjunto de tecnologías base, que puedan ser integradas por desarrolladores para crear sistemas complejos con un alto grado de interoperabilidad [Bellifemine99].

Otra organización, el NIIIP (National Industrial Information Infrastructure Protocol), organización que agrupa empresas estadounidenses, pretende desarrollar protocolos abiertos de software industrial que permitan a los fabricantes comunicarse con sus proveedores como si fueran parte de la misma empresa.

Este trabajo de tesis usa el estándar FIPA para la programación de los agentes porque pretende ser un estándar internacional, útil para gran número de aplicaciones y que permite aplicaciones con un alto grado de interoperabilidad.

Hay organizaciones comerciales y de investigación involucradas en la creación de aplicaciones de agentes y se han realizado herramientas para construcción de aplicaciones. Algunas son AgentBuilder, dMars, Mole, Open Agent Architecture, RETSINA, Zeus y Jade [Bellifemine00], de los cuales se presenta enseguida una breve descripción.

AgentBuilder es una herramienta para la construcción de sistemas de agentes basados en Java. Los agentes se comunican generalmente con mensajes KQML aunque el programador puede definir nuevos comandos para la comunicación según sus necesidades. AgentBuilder se basa en dos componentes:

- El Toolkit, que incluye herramientas para administrar el proceso de desarrollo de agentes, analizando el dominio del dominio de operaciones del agente, definiendo, implementando y probando los agentes.
- El Run-time System. Proporciona una máquina agente que es un intérprete usado como ambiente de ejecución de los agentes.

dMARS es un ambiente de implementación y desarrollo orientado a agentes para construir sistemas distribuidos basados en el modelo de agente BDI que ofrece soporte para la configuración del sistema, diseño, mantenimiento y reingeniería. Ha sido usado para aplicaciones de control de tráfico aéreo y administración de procesos de negocios y telecomunicaciones [Bellifemine00].

La plataforma Mole, desarrollada en la Universidad de Stuttgart, en Alemania, ofrece una de las mejores soluciones para permitir la movilidad de agentes. Los agentes Mole son entidades multi-hilo con un identificador global único de agente que usan RMI (Remote Method Invocation) en interacciones cliente/servidor y con intercambio de mensajes en caso de interacciones punto-a-punto. El RMI no es un estándar, sino un mecanismo para comunicación e intercambio de objetos entre varias JVM (Java Virtual Machine).

La Open Agent Architecture permite realizar sistemas de agentes distribuidos en C, Java, Prolog, Lisp, Visual Basic y Delphi. Posee un facilitador que coordina las tareas de todos los agentes. El facilitador puede recibir tareas de agentes, descomponerlas y

asignárselas a otros agentes. En esta arquitectura los agentes siempre se comunican entre ellos a través del facilitador, lo cual puede volverse un cuello de botella para la aplicación.

RETSINA (Reusable Environment for Task Structured Intelligent Networked Agents) permite desarrollar agentes reusables para realizar aplicaciones. Cada agente tiene cuatro módulos reutilizables para comunicación, planeación, planificación y monitoreo de la ejecución de tareas y peticiones de otros agentes. Se comunican a través de mensajes KQML. RETSINA puede interoperar con agentes desarrollados en otras plataformas y sistemas no basados en agentes. La biblioteca para creación de agentes de RETSINA, denominada AFC (Agent Foundation Classes), está desarrollada en Java y en C++.

Zeus permite el desarrollo de sistemas en Java al proporcionar una biblioteca de componentes de agentes, proporcionando un ambiente visual para capturar especificaciones de usuario, un ambiente de construcción de agentes que incluye un generador automático de código del agente y una colección de clases que son los bloques constructores de agentes individuales.

JADE (Java Agent DEvelopment Framework) es una plataforma desarrollada por CSELT (Telecom Italia Group Research Center) y la Universidad de Parma, en Italia. La plataforma, desarrollada en Java, se basa en el estándar industrial FIPA (Foundation for Intelligent Physical Agent). Proporciona uan infraestructura de servicios y comunicaciones para la programación de agentes. Propone un agente básico que puede adaptarse a las necesidades del sistema multiagente. La comunicación entre agentes es con paso de mensajes eligiendo el mecanismo de transporte de mensajes más adecuado entre RMI, CORBA y eventos de Java. La versión Jade v1.3 fue liberada bajo la licencia Open Source LGPL. El grupo de desarrollo está disponible via internet para responder dudas a quienes usan la plataforma para programar sus propias aplicaciones.

1.2. Sistemas de manufactura flexible y tecnología multiagente

Un Sistema de Manufactura Flexible (FMS, Flexible Manufacturing System) es un sistema compuesto de máquinas-herramienta, sistemas automáticos de transporte, robots y otros dispositivos, todos controlados por un sistema de cómputo. El objetivo del sistema es fabricar productos finales, obtenidos mediante trabajos realizados sobre las piezas que

llegan al sistema. Cada pieza tiene asociada una serie de tareas (operaciones) llamada plan de proceso, que debe ser realizado por los dispositivos del sistema [Rodriguez00].

La organización de los sistemas de manufactura ha pasado de un modelo centralizado con maquinaria especializada a una estructura compuesta de unidades más pequeñas llamadas *células* con máquinas fáciles de reconfigurar y que pueden manejar múltiples herramientas.

Un sistema centralizado coordina la tarea de manufactura dando órdenes a cada dispositivo del sistema. En sistemas de manufactura muy grandes, el control de la tarea se complica por el gran número de estados que deben tomarse en cuenta [López-Mellado97]. Un sistema de control distribuido no maneja un estado global y cada subsistema es responsable de una parte de la tarea de manufactura. Además, los subsistemas deben coordinarse para lograr la tarea global.

El núcleo de un sistema de coordinación es un software muy complejo que determina la flexibilidad y el rendimiento del sistema automatizado. Este software se encarga de la ejecución, el monitoreo, la toma de decisiones y la planificación de las actividades a realizarse en el sistema de manufactura [López-Mellado97].

1.2.1. Control jerárquico en los FMS

La coordinación de un FMS puede ser centralizada. Esto implica que un solo sistema controle la tarea de manufactura completa y ordene a cada dispositivo las operaciones que debe realizar.

Las funciones del sistema de control, previamente mencionadas, generalmente se distribuyen en una jerarquía de cuatro niveles propuesta por Gershwin [Gershwin89]. En dicha jerarquía el tiempo de respuesta es menor en los niveles más bajos. Un ejemplo se muestra en la Figura 1-2.

El nivel más bajo contiene los controladores locales de los dispositivos físicos en la célula de manufactura (robots, bandas, máquinas, sensores, etc.). El nivel de coordinación de tareas o nivel de célula controla y supervisa las actividades de los controladores locales involucrados en una célula a través de la generación de comandos según los eventos que ordene el nivel de control local. El planificador de tareas genera la estrategia del

controlador para las células de manufactura según las especificaciones del nivel de planeación de la producción.

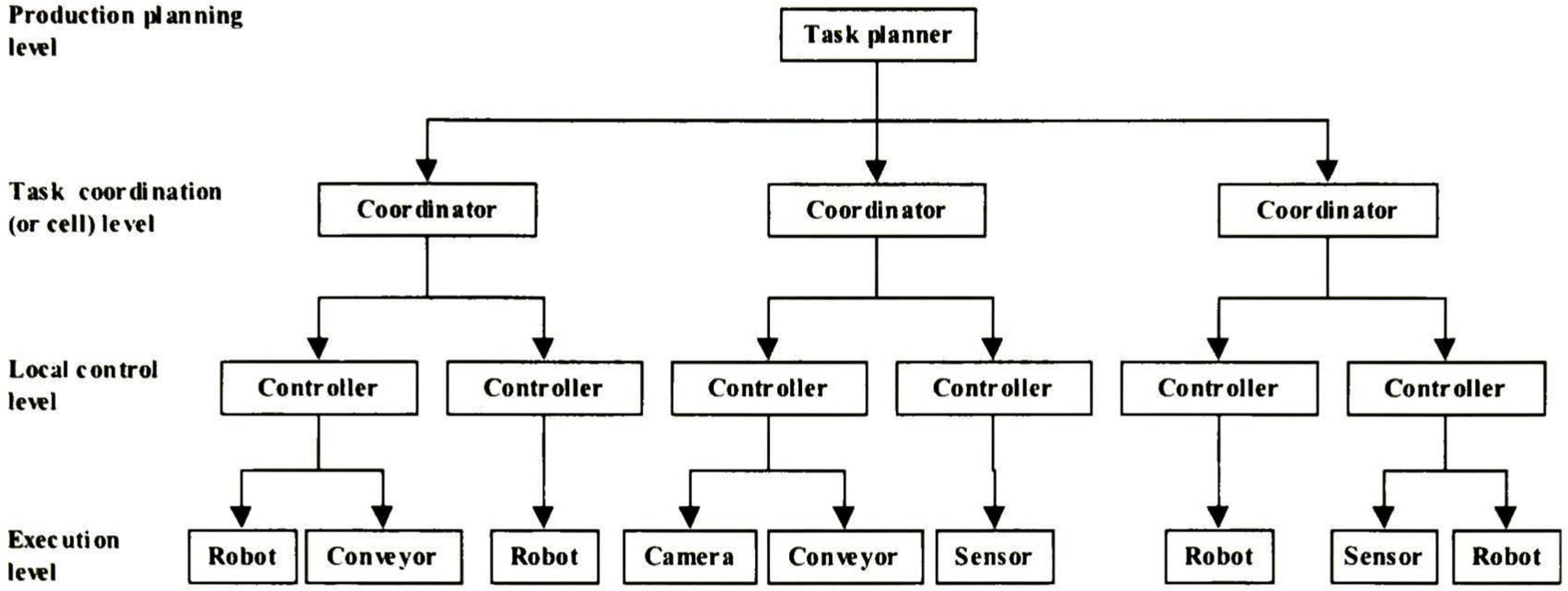

Figura 1-2. Jerarquía para el sistema de control de los FMS.

Las funciones del nivel de coordinación de tareas son (1) el secuenciamiento de operaciones para que la tarea de ensamble/manufactura funcione normalmente y (2) el manejo de excepciones, que representan fallos de operación. En el nivel de control local, un proceso de monitoreo indica los eventos esperados para el secuenciador de operaciones; el secuenciador envía a este proceso comandos para verificar la información de los sensores.

El trabajo realizado en esta tesis se ubica en el nivel de coordinación de tareas y propone un método basado en el paradigma de los sistemas multi-agente para descentralizar el control del sistema en varios subsistemas que trabajen de manera distribuida y coordinada. Cada subsistema controlará y será responsable de solo una parte del proceso de producción.

1.2.2. Sistemas holónicos de manufactura

Los sistemas holónicos de manufactura (HMS, Holonic Manufacturing Systems) son una propuesta de los investigadores y de la industria [Ouelhadj98]. Estos sistemas están formados por holones, que son unidades de manufactura autónomas y confiables que cooperan para lograr las metas de manufactura globales. Un sistema de holones se denomina holarquía. Este concepto es recursivo, pues una holarquía puede comportarse como un holón que coopere y actúe de manera autónoma en otra holarquía.

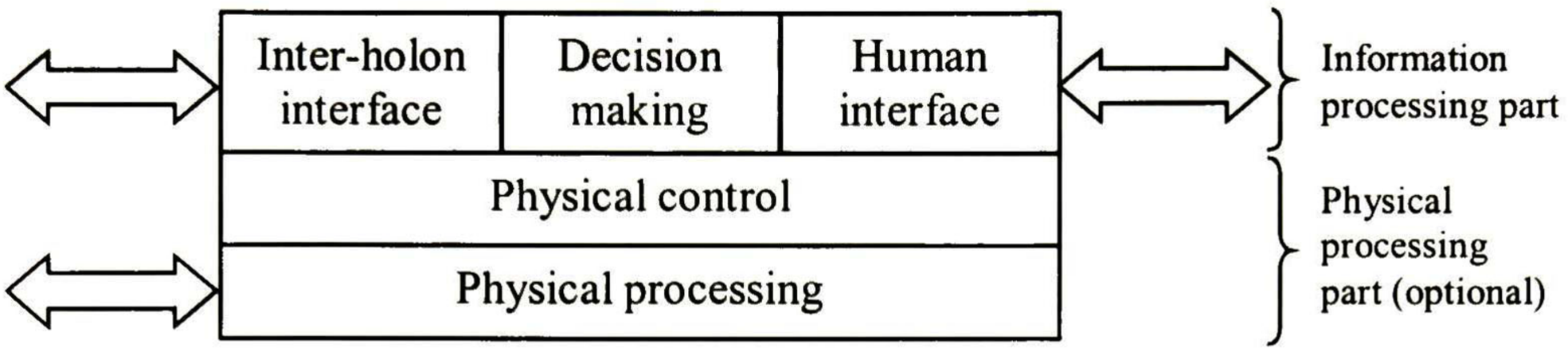

Figura 1-3. Arquitectura general de un holón.

En la Figura 1-3 se muestra la arquitectura de un holón [Bussmann98b]. El nivel de procesamiento físico es el hardware real que realiza la operación de manufactura. Esta operación es controlada por el nivel de control físico. La toma de decisiones representa el núcleo del holón y proporciona una interface para interactuar con otros holones y otra interface para interactuar con operadores humanos.

Los holones fueron propuestos en 1967 por Arthur Koestler y son considerados como una forma de estructurar y controlar procesos de producción [Bussmann98b]. En el mismo trabajo, Bussmann propone un diseño orientado a agentes de los holones, donde la parte de procesamiento de información de cada holón sea implementada usando la tecnología de agentes (MAS, Multi Agent Systems), tal como se muestra en la Figura 1-4.

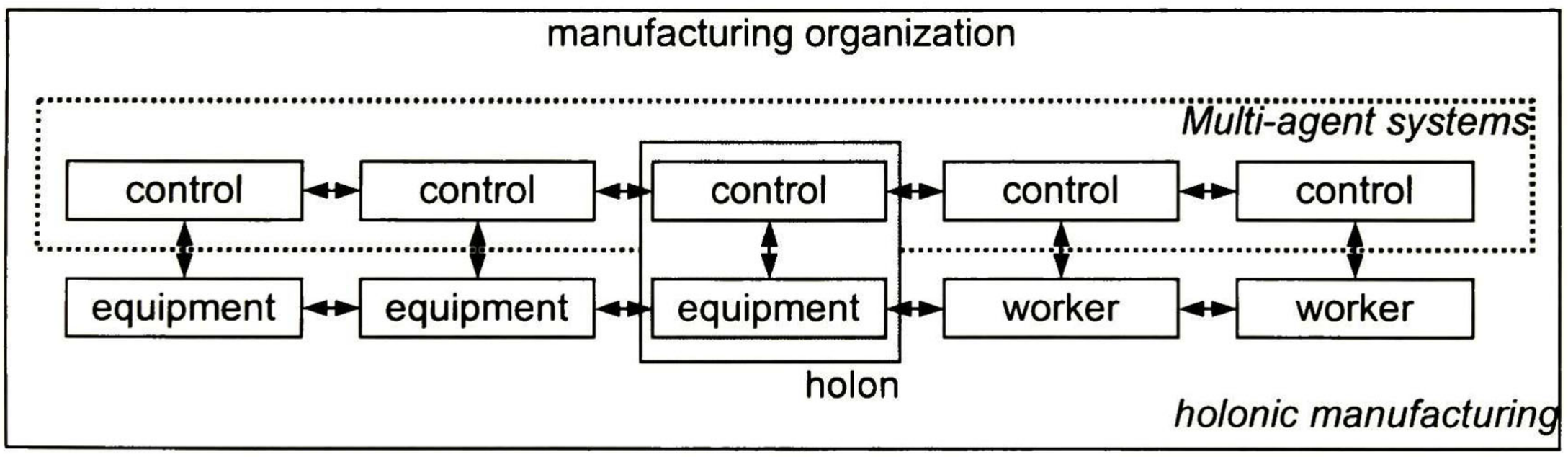

Figura 1-4. Vistas de los HMS y MAS en el proceso de manufactura.

1.2.3. Coordinación de sistemas de manufactura con enfoque multiagente

El éxito de los sistemas multi-agente en la manufactura crea nuevos retos para la tecnología. Además de la funcionalidad al resolver los problemas, un sistema de manufactura usado por la industria debe ser confiable, tolerante a fallos, de fácil mantenimiento, transparente, etc.

Para que la tecnología multi-agente sea ampliamente usada y aceptada en la industria es necesario que quienes no sean investigadores estén capacitados para aplicar técnicas orientadas a agentes como cualquier otro método de ingeniería. Esto implica, en particular, que los ingenieros sean capaces de diseñar y programar sistemas de coordinación orientados a agentes de una manera sencilla y eficiente. [Bussmann98a]

En los últimos años los sistemas de manufactura se han vuelto cada vez más complejos. La competencia ha obligado a los fabricantes a acortar el ciclo de producción, disminuir inventarios, optimizar el uso de la maquinaria y acortar los tiempos de procesamiento. Como resultado, el problema de controlar un FMS es muy importante y debe ser diseñado para que el sistema posea mayor inteligencia.

La función de un sistema de control de células puede variar dependiendo del tamaño de la celda además del tipo y grado de la toma de decisiones de la celda. Las principales funciones de un sistema de control de células incluye la necesidad de planificar y monitorear los recursos de la célula, y la habilidad para reaccionar ante condiciones anormales o excepciones.

Como los sistemas de manufactura generalmente son distribuidos por naturaleza, tanto geográfica como organizacionalmente, la toma de decisiones en los sistemas de coordinación de manufactura puede verse como un problema distribuido. Físicamente los sistemas de manufactura involucran varios recursos (robots, bandas de transporte, máquinas de control numérico, etc.). Desde el punto de vista lógico, varias tareas pueden ser realizadas en paralelo.

Por las razones antes expuestas, Ouelhadj propone que el área de la Inteligencia Artificial Distribuída (DAI, Distributed Artificial Intelligence) y en particular los sistemas multi-agente son adecuados para la coordinación de sistemas de manufactura [Ouelhadj98].

Los sistemas multi-agente permiten construir sistemas de producción descentralizados en vez de centralizados, con comportamiento emergente en vez de planeado, y con ejecución concurrente (asíncrona) en vez de secuencial (síncrona). Asimismo, en vez de que la información y el control estén centralizados, las arquitecturas de agentes autónomos están conscientes de que los datos y el control están distribuidos a lo largo del sistema.

La primera implementación reportada de sistemas multiagente en manufactura fue YAMS (Yet Another Manufacturing System) [Ouelhadj98] que asigna un agente a cada nodo de la jerarquía de control (fábrica, estación de trabajo, máquina). Un agente en un nivel usa un protocolo de negociación para identificar los agentes bajo su control en el siguiente nivel más bajo a los cuales asignar las tareas.

En [López-Mellado97] se propone un método para programar sistemas orientados a objetos que controlen FMS con toma de decisiones de acuerdo a la secuencia de operaciones del sistema de manufactura/ensamblado. Propone modelar el sistema con grafos de flujo de partes para definir rápidamente las clases y métodos acortando el ciclo de desarrollo del sistema. Sin embargo, no define cómo se puede llevar a cabo la ejecución de la tarea de manufactura en varios procesadores ni propone una formalización ni maneras de validar del modelo de la tarea para evitar interbloqueos e inconsistencia entre otros problemas.

En [Ouelhadj98] se propone una arquitectura para control dinámico de células de manufactura basada en una plataforma que usa actores. Esta consiste de Agentes de Recurso (RA, Resource Agent) y actores, donde cada Agente de Recurso está asociado con un recurso particular de la célula. Se distribuyen las funciones de control (planificación, ejecución, análisis de fallos y monitoreo) en todos los agentes del sistema. El sistema utiliza como mediador para llevar a cabo el plan de producción un Agente de Tarea (Task Agent), el cual que puede convertirse en un cuello de botella en caso de sobrecarga del sistema. Se propone el protocolo *contract-net* para la planificación dinámica de tareas. El sistema resultante es flexible, tolerante a fallos, adaptable, reconfigurable y aprovecha el paralelismo. Sin embargo, el sistema no es completamente distribuido. No presenta una forma de programar los agentes ni los actores involucrados. Tampoco propone un método para especificar las tareas de un sistema de manufactura y determinar el número de agentes/recursos necesarios y sus tareas.

Bussmann [Bussmann98a] ha propuesto una arquitectura de agente y un método de tres pasos para programar tareas de control en manufactura partiendo de la especificación de las tareas de control para el agente. La ejecución concurrente de tareas se programa para uno solo procesador. Sin embargo, no propone un método para partir del problema específico de manufactura y crear una especificación de tareas de los agentes, sino que para

ello remite a trabajos de Burmeister y Kenny entre otros, aunque con aplicación no específica a manufactura. Tampoco se refiere a algún método de partición de la tarea de manufactura y su asignación a los agentes correspondientes.

En otro trabajo, Bussmann [Bussmann97] también ha utilizado a los agentes como entes negociadores para la planificación del flujo de partes en una ensambladora de automóviles utilizando álgebra de restricciones. Cada agente recibe una entrada máxima de partes y produce una salida máxima, que puede ser la entrada para otro agente, realizándose un análisis de restricciones y generando una solución óptima con el uso de avance/retroceso en la planificación. Este sistema pretende hacer más entendible cómo afectan las decisiones locales en el proceso global. Sin embargo, el método no pretende controlar el proceso de producción sino solamente modelarlo con restricciones. No se propone una manera de programar los agentes ni de identificar sistemas que pueden tratarse con este tipo de enfoque.

En [Brazier95] se presenta el desarrollo de una plataforma de especificación formal para sistemas de razonamiento complejo que fue aplicado a la administración de redes. La plataforma permite especificar tareas primitivas y complejas para un componente y definir la interacción con ligas de información. Sin embargo, la representación de tareas está términos empleados en el área de ingeniería del conocimiento (KE, Knowledge Engineering), lo cual no es apropiado para tareas de manufactura que generan rutinas/procedimientos y estrategias operativas.

En [DeLoach98] se propone usar AgML (Agent Modeling Language) y AgDL (Agent Definition Language) para describir los tipos de agentes en un sistema y sus interfaces hacia otros agentes. Además define la metodología MaSE (Multiagent System Engineering) para la síntesis formal de sistemas de agentes. La metodología propone cuatro pasos: Diseño del nivel de dominio, diseño de nivel de agente, diseño de componentes y diseño del sistema. La ventaja de este enfoque es que permite una sintaxis y semántica formal para verificar propiedades. La desventaja es que no muestra la relación existente entre el modelado y la posible generación de código; tampoco propone una arquitectura de agente ni una metodología para programar el agente una vez obtenida la especificación formal.

Para este trabajo se ha elegido la plataforma JADE porque el agente básico propuesto cuenta con los servicios mínimos necesarios para coordinarse con otros agentes. Los agentes pueden fácilmente ser adaptados para controlar la tarea asignada usando reglas. Estos agentes pueden estar en equipos y plataformas distintos. La plataforma cuenta con herramientas para su administración.

Además, el software y material necesario para realizar aplicaciones con la plataforma están disponibles en internet (*http://sharon.cselt.it/projects/jade/*). El grupo de desarrollo está al pendiente para dar soporte en la implementación de sistemas multiagente usando JADE.

1.3. Características de JADE y FIPA

JADE (Java Agent DEvelopment framework) es una plataforma para facilitar el desarrollo de aplicaciones de agente cumpliendo con especificaciones para interoperabilidad de FIPA. JADE cumple con las especificaciones establecidas en [FIPA97] e incluye los agentes de sistema que administra la plataforma: el ACC (Agent Communication Channel), el AMS (Agent Management System) y el DF (Directory Facilitator) por default. La comunicación entre agentes se realiza con paso de mensajes cuya representación se basa en el lenguaje FIPA ACL. El modelo de comunicación es punto-a-punto. JADE usa tecnologías de transporte como RMI (Remote Method Invocation) de Java, CORBA y los eventos de Java. También proporciona herramientas para administrar la ejecución de la plataforma y monitorear y depurar sociedades de agentes.

1.3.1. Estándar de FIPA

FIPA es una organización creada en 1996 dedicada al desarrollo de estándares para plataformas de agentes y aplicaciones basadas en agentes. Está integrada por empresas como IBM, Toshiba, Boeing, Siemens, Fujitsu entre otras, y universidades como la University of London, University of Calgary, etc. El trabajo de estandarización de FIPA se orienta a permitir una facil interoperabilidad entre sistemas de agentes. Pero en vez de solo especificar un lenguaje de comunicación de agentes, especifica los agentes necesarios para

la administración de un sistema de agentes y la ontología necesaria para la interacción entre dos sistemas.

La organización FIPA hace dos suposiciones: (1) Que quizás es muy poco el tiempo para alcanzar consensos y terminar el estándar antes de que las industrias lleguen a acuerdos al respecto, lo cual debe ser una razón para continuar desarrollándolo. (2) Solamente se debe especificar el comportamiento externo de los componentes del sistema, dejando al programador los detalles de implementación y arquitectura interna.

En [FIPA97] se especifican las normas que permiten a una sociedad de agentes interoperar, existir, operar y ser administrada. Describe el modelo de referencia de una plataforma de agentes, que identifica los roles de algunos agentes básicos necesarios para la administración de la plataforma, y especifica la ontología (contexto) y lenguaje de contenido para la administración de los agentes. Los roles básicos necesarios identificados en la plataforma de agente son:

1. El Agente de Administración del Sistema (AMS, Agent Management System) se encarga de ejercer un control supervisor en el acceso y uso de la plataforma. Se encarga de verificar la autenticidad de agentes residentes y el control registros.
2. El Agente del Canal de Comunicación (ACC, Agent Communication Channel) proporciona la forma para el contacto básico entre agentes dentro y fuera de la plataforma. Es el método de comunicación por default que ofrece una rutina de servicio de mensajería confiable y ordenado. También permite el uso de IIOP (Internet Inter-ORB Protocol) para la interoperabilidad entre diferentes plataformas.
3. El Facilitador de Directorio (DF, Directory Facilitator) es el agente que proporciona a la plataforma del agente un mecanismo de consulta de información sobre agentes y servicios disponibles.

El estándar especifica también el Lenguaje de Comunicación de Agentes (ACL, Agent Communication Language). La comunicación de agentes está basada en el paso de mensajes, donde éstos se comunican formulando y enviando mensajes individuales a otros. No se establece un mecanismo para la transportación interna de mensajes. Los mensajes transportados entre plataformas deben ser transmitidos en forma textual.

El estándar permite formas comunes de conversaciones entre agentes con la especificación de protocolos de interacción, que son secuencias de mensajes intercambiados por dos o más agentes, como el protocolo de negociación *contract-net*. Este es un protocolo basado en la negociación diseñado para facilitar la solución de problemas en ambientes distribuidos (*http://web.mit.edu/fuencis/www/contract_net.htm*).

Implementaciones que cumplen con FIPA

Actualmente existen algunas implementaciones que FIPA reconoce públicamente que cumplen con sus estándares de plataforma de agentes.

- April Agent Platform, de Fujitsu Labs of America.
- FIPA-OS, de Emorphia.
- Grasshopper, de IKV++.
- JADE, de CSELT.
- Zeus, de British Telecommunications.

1.3.2. Plataforma JADE

JADE (Java Agent Development Framework) simplifica el desarrollo de aplicaciones de agentes asegurando el cumplimiento de las especificaciones de FIPA para la interoperabilidad de sistemas multiagente inteligentes mediante un conjunto comprensible de servicios y agentes de sistema. JADE proporciona al programador de agentes las siguientes facilidades:

- Una plataforma de agentes que cumple los estándares de FIPA, que incluye los agentes AMS, DF y ACC. Estos tres agentes son activados al inicio de la ejecución de la plataforma.
- Una plataforma para agentes distribuidos que puede utilizar varias computadoras. Los agentes se implementan con hilos y eventos de Java.
- Un mecanismo de transporte e interface para enviar/recibir mensajes desde/hacia otros agentes.
- Un protocolo IIOP que cumple con [FIPA97] para conectarse con otras plataformas de agentes.
- Una biblioteca de protocolos de interacción de FIPA listos para usarse.

- Un registro automático de agentes con el agente AMS.
- Un servicio de nombrado que cumple con FIPA que al comenzar la ejecución de los agentes, éstos obtienen su GUID (Globally Unique IDentifier) de la plataforma.
- Una interface gráfica de usuario para administrar varios agentes y plataformas de agentes desde un mismo agente. Se puede monitorear la actividad de cada plataforma.

El sistema de comunicación usado por JADE elige el sistema más económico para realizarlo. Así, cuando un agente envía mensajes pueden darse los siguientes casos:

- Que el agente receptor esté en el mismo contenedor del agente emisor. En este caso el mensaje se le envía al receptor a usando un objeto evento.
- Que el agente receptor se encuentre en la misma plataforma JADE pero en otro contenedor. El mensaje ACL se envía usando RMI.
- Que el agente receptor se encuentre en una diferente plataforma. Se usa el protocolo estándar IIOP y la interface OMG IDL (Object Management Group Interface Definition Language) para entregar el mensaje.

JADE está escrito en Java y consta de varios paquetes, proporcionando a los programadores objetos funcionales listos para usarse e interfaces abstractas para adaptarse a aplicaciones según sus tareas.

Este trabajo retoma el método propuesto en [López-Mellado97] extendiéndolo para la programación de sistemas de coordinación en un ambiente distribuido cuyos agentes tengan control de una parte de la tarea. La función de coordinación de los agentes estará basada en reglas. Para la implementación de los agentes se usará la plataforma JADE, que cuenta con la infraestructura necesaria para la comunicación y administración de los agentes.

CAPÍTULO

2

Metodología de Modelado

En este capítulo se describe la metodología de modelado, parte esencial de la estrategia propuesta para el desarrollo de software de coordinación de actividades distribuidas basados en sistemas multi-agente. Se explica en detalle el proceso de modelar la tarea global y de obtener las subtareas de las cuales se encargarán los agentes del sistema de coordinación.

2.1. Descripción general

La metodología propuesta para la programación de sistemas de coordinación de actividades distribuidas basados en sistemas multiagente ha sido dividida en dos partes: (1) la metodología de modelado, y (2) la programación del sistema multiagente. En el presente capítulo se describe la primera parte de la metodología.

La metodología propuesta es una extensión a la presentada en [López-Mellado97]; ésta permite obtener sistemáticamente las bases del conocimiento necesarias a partir de un modelo del sistema de manufactura. Este modelo incluye la discretización del espacio de trabajo, una clasificación de componentes y una descripción del flujo de partes.

En este capítulo se explica en detalle el proceso para obtener las subtareas de las cuales se encargarán sendos agentes del sistema. Partiendo de una descripción del sistema y una

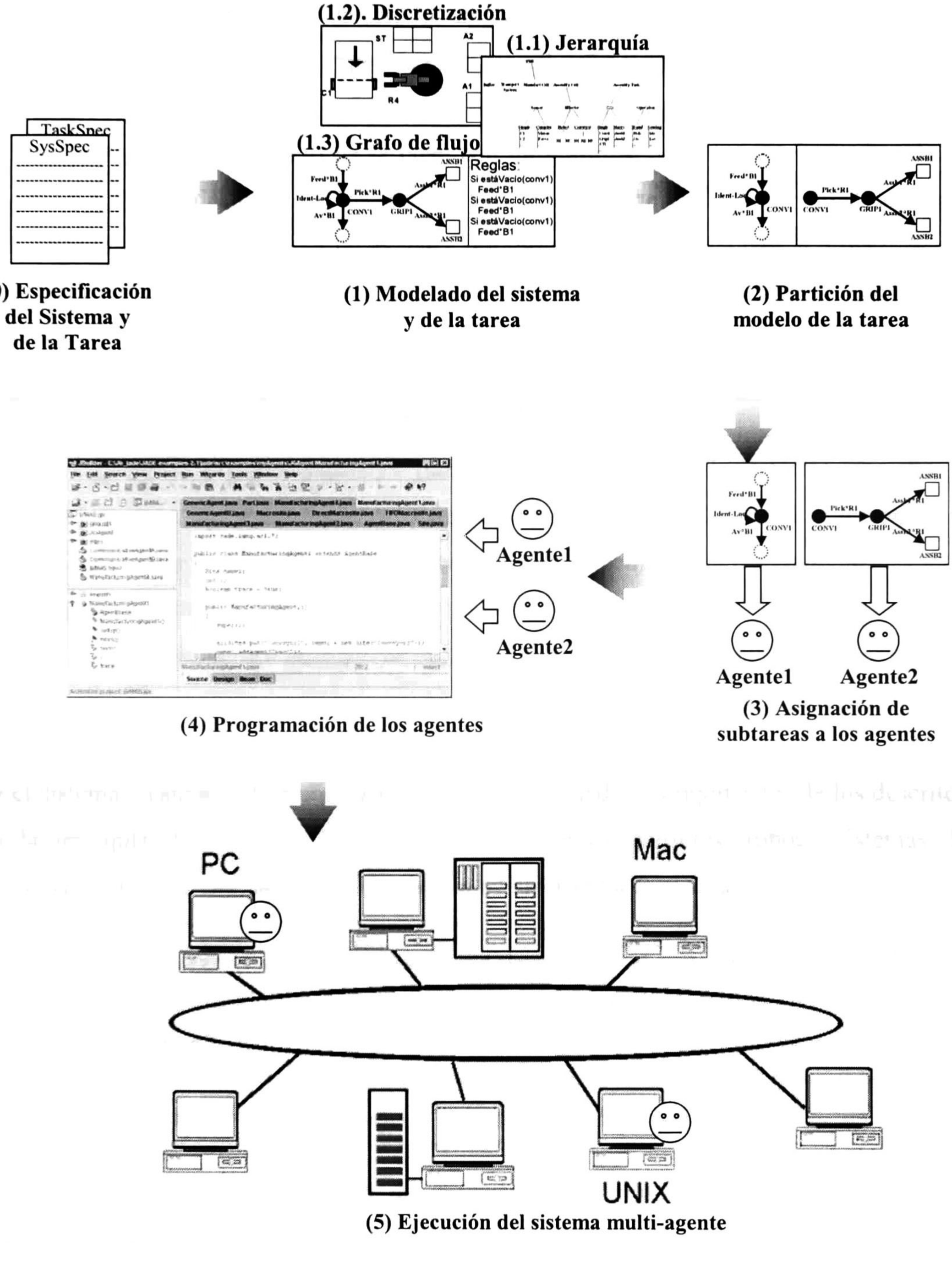

Figura 2-1. Pasos de la metodología propuesta.

descripción de la tarea de ensamble se obtiene de manera sistemática el software necesario para la coordinación de actividades en un sistema de manufactura. La Figura 2-1 muestra gráficamente las etapas del desarrollo de un sistema de coordinación usando la metodología. Se parte de (0) la especificación verbal del sistema y de la tarea para generar (1) una especificación formal que incluye la discretización del área de trabajo, la jerarquía de componentes y el modelado de la tarea con un grafo de flujo de partes. Enseguida se realiza la partición del grafo (2) de acuerdo a criterios que se verán posteriormente. Cada partición/subtarea es asignada a un agente (3); enseguida se continúa con la programación de cada uno de ellos (4) para finalmente llevar a ejecución el sistema multiagente obtenido (5).

2.2. Descripción del sistema y la tarea

El desarrollo de un controlador está basado en el modelo de la tarea; para ello es necesario partir de una *descripción del sistema de manufactura* y de una *descripción de la tarea. Esto es, una descripción textual acompañada con gráficos de los componentes y de los requerimientos del FMS.*

La *descripción del sistema de manufactura* debe proporcionar una descripción taxonómica de los componentes del sistema de manufactura, declarando sus características y capacidades. Esta jerarquía de componentes será útil para programar las clases necesarias en el sistema. También debe mencionar el tipo y cantidad de componentes de los descritos en la jerarquía, tales como las bandas transportadoras, sensores, robots, sistemas de reconocimiento. Un ejemplo de la descripción de un sistema es el siguiente:

> "La célula de ensamble consiste de una cinta transportadora B1, un robot R1 y dos mesas para ensamble A1 y A2. Cada mesa para ensamble tiene dos lugares para ensamblar piezas. Frente a R1 hay un sensor óptico C1 que detecta la llegada de partes sobre la banda B1. Encima de esta zona hay una cámara que forma parte de un sistema de reconocimiento y localización de piezas. Esto se ilustra en el diagrama de la Figura 2-3"

La *descripción de la tarea de ensamble* debe mostrar la lógica que sigue el ensamblado de partes, la secuencia de ensamblaje, las decisiones a tomar, etc. Por ejemplo, puede describirse la tarea de la siguiente manera:

"El flujo de entrada en B1 está formado por cuatro tipos de partes (A, B, C y D) que llegan en orden aleatorio. R1 toma las partes y realiza ensambles en A1 (con las partes A y B) o A2 (con las partes C y D) según un orden predefinido para cada producto. La detección de partes por C1 detiene la banda B1. La identidad de cada parte la determina el sistema de visión cuando llega a la posición de C1.'

2.3. Especificación formal

2.3.1. Modelado del sistema

Discretización del espacio de trabajo

La descripción de una tarea de ensamble/manufactura requiere expresar las operaciones en términos cualitativos y cuantitativos. La descripción cualitativa toma en cuenta la identidad de las piezas y su estado de procesamiento, así como de los lugares en que una determinada pieza visitará a lo largo de su procesamiento; para este fin se introduce la noción de sitio [Chochon87] la cual se define como "todo lugar donde puede residir la pieza en posición estable". Ejemplos de sitios se muestran en la Figura 2-2.

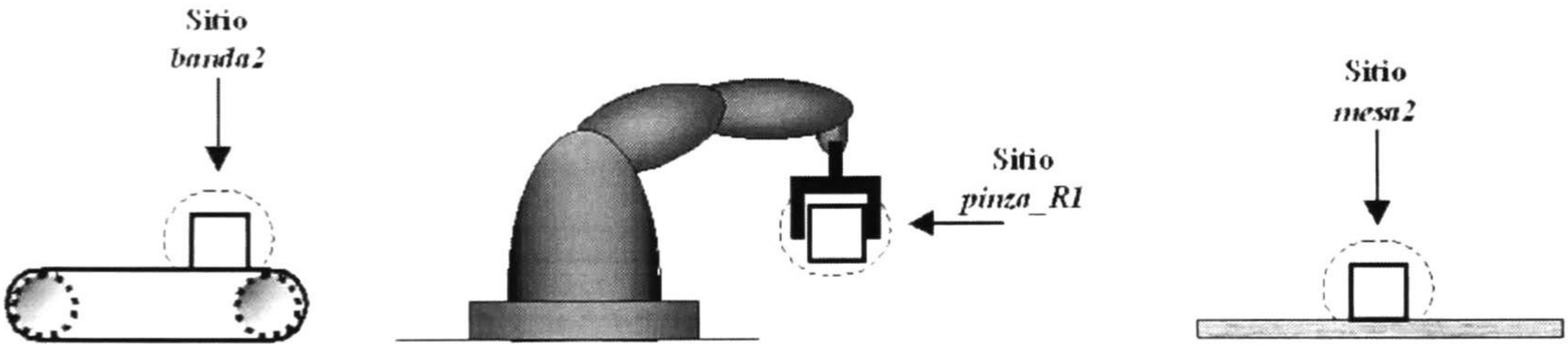

Figura 2-2. Ejemplos de sitios en dispositivos.

Los sitios pueden ser individuales o compuestos. Los sitios individuales poseen un solo lugar para colocar/ensamblar piezas. Los sitios compuestos tienen dos o más lugares. La organización del acceso a los lugares de los sitios compuestos puede ser realizado con acceso directo, acceso FIFO, acceso LIFO, etc. de acuerdo a la conveniencia del proceso. En general, los círculos modelan sitios individuales, mientras que los cuadrados modelan sitios compuestos.

Clasificación funcional

Los componentes del sistema de ensamble son clasificados desde un punto de vista funcional como sensores, actuadores, etc. En la Figura 2-3 se muestra la jerarquía de componentes. Esta clasificación ayuda a estructurar el conocimiento requerido para el sistema de ensamble como es el estado de la tarea, las capacidades y relaciones entre los componentes, etc. Los elementos del nivel más bajo de la jerarquía pueden ser objetos instanciados de los conceptos (clases) que están situados más arriba en la jerarquía [López-Mellado97].

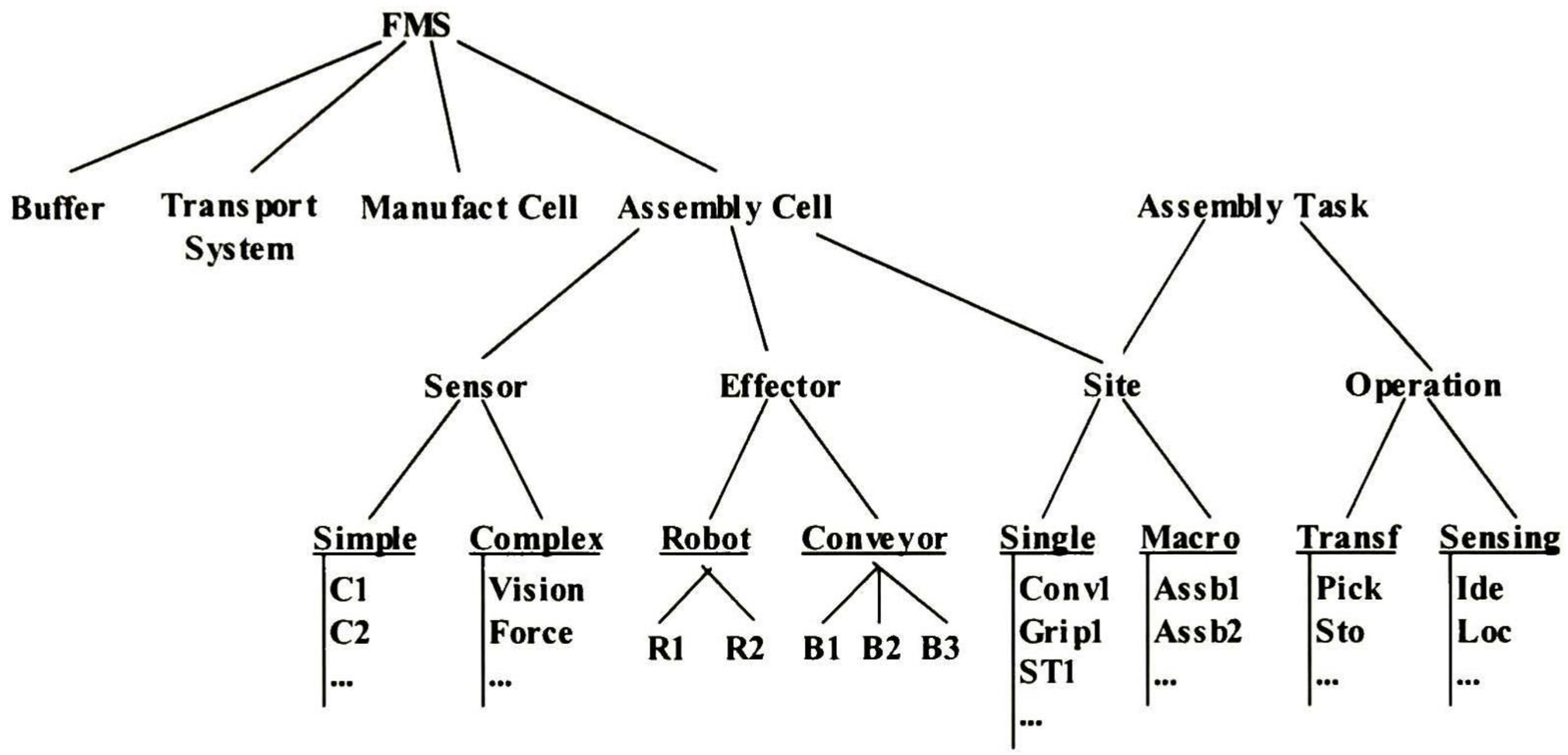

Figura 2-3. Clasificación de los componentes de FMS.

2.3.2. Modelado de la tarea

El *modelado de la tarea* consiste en la especificación del flujo de piezas en el sistema por medio de un grafo dirigido cuyos nodos son los sitios del sistema y los arcos son las operaciones que desplazan, transforman o cambian algunas propiedades de las piezas. Este grafo es llamado el *grafo de flujo de piezas* FP y se define a continuación.

Definición. Un grafo de flujo de piezas (FP) es la tupla

$$FP = (G, SITES, \varepsilon, OPER, \lambda, \phi) \text{ donde}$$

- G es un grafo dirigido conexo $G = (V, A)$ donde

  V es un conjunto finito de vértices

  $A = \{(v_i, v_j) \mid v_i, v_j \in V\}$ es un conjunto de arcos.

- SITES = {sitio1, ..., sitio***n***} es un conjunto finito de sitios que no son de E/S.
- ε = {input1,..., input***p***, output1, ..., output***q***} es un conjunto finito de sitios de entrada/salida por donde entran o salen piezas del sistema de manufactura.
- OPER = {oper1, ... , oper***z***} es un conjunto finito de las operaciones del sistema.
- $\lambda : V \rightarrow SITES \cup \varepsilon$ es una función de etiquetado que asigna nombres de sitios a los vértices de G.
- $\phi : A \rightarrow OPER$ es una función de etiquetado que asigna nombres de operaciones a los arcos de G.

Adicionalmente, para cada operación se describen las condiciones de ejecución y modificaciones provocados por la operación bajo la forma de regla *si-entonces*.

Así, tomando la descripción del sistema, es posible identificar los sitios definidos: CONV1, un sitio asociado al lugar que es accesible por el robot sobre la banda transportadora; GRIP1, sitio asociado al brazo del robot R1; finalmente, sabiendo que las mesas de ensamble tienen dos lugares, se identifican los sitios ASSB1j y ASSB2j (j=1,2), controlados por ASSB1 y ASSB2 respectivamente.

Los arcos que unen los nodos representan las operaciones necesarias para transferir las partes de un sitio a otro. Es posible que la operación sólo modifique las propiedades de la parte que está en el sitio, tal como lo ejemplifica la operación *Ident-Loc*, la cual no transfiere la pieza a otro sitio, sino que modifica los atributos de una parte hasta ese momento desconocida. Las operaciones también pueden ingresar partes al sistema o sacar productos del sistema. Esto es indicado en el modelo mediante los nodos del conjunto ε.

Tomando en cuenta los sitios identificados y la descripción de la tarea, se puede modelar la tarea con el grafo de flujo de material mostrado en la Figura 2-4. La regla que describe las condiciones de ejecución de la operación R1.pick() es la siguiente:

```
SI conv1=vacio ∧ grip1<>vacio ∧ piezaProcesable(conv1) ENTONCES
   R1.pick();
   Actualizar( grip1, contenido(conv1) );
   Actualizar( conv1, vacío );
FIN SI
```

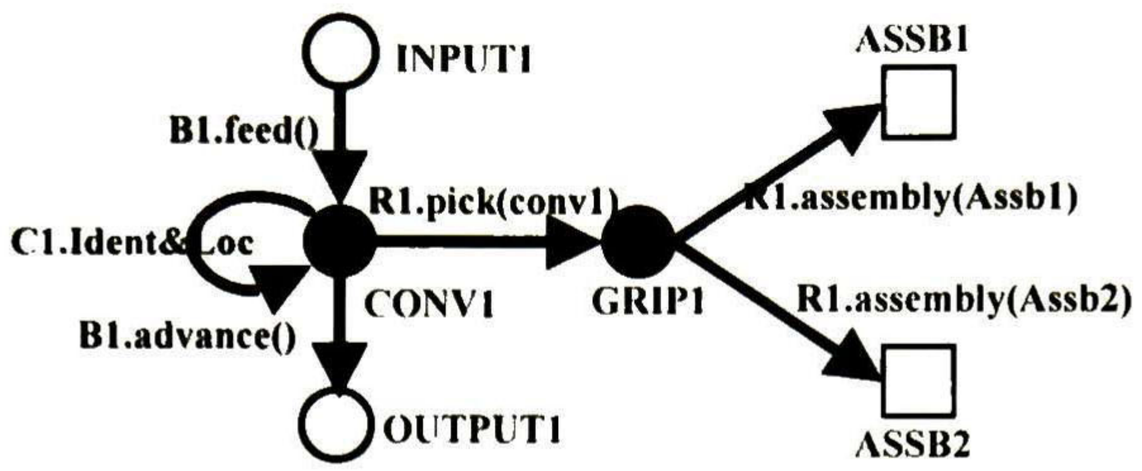

Figura 2-4. Grafo de flujo de partes.

2.4. Ejemplo de modelado

Enseguida se muestra un ejemplo paso a paso de obtención de un modelo partiendo de la descripción del sistema, de la tarea de ensamble y de la clasificación de componentes. La descripción textual del sistema de manufactura y de la tarea se dan a continuación.

Descripción de un sistema de manufactura.

"La célula de ensamble consiste de las cintas transportadoras B1, B2 y B3, los robots R1 y R2, dos mesas para ensamble ASSB1 y ASSB2 , una mesa para colocar piezas ST. Cada mesa para ensamble tiene dos lugares para ensamblar piezas. Frente a R1 hay un sensor óptico C1 que detecta la llegada de partes sobre la banda B1. Encima de esta zona hay una cámara que forma parte de un sistema de reconocimiento y localización de piezas"

Descripción de la tarea de ensamble.

"El flujo de entrada en B1 está formado por cuatro tipos de partes (A, B, C y D) que llegan en orden aleatorio. R1 toma las partes y realiza ensambles en A1 (con las partes A y B) o A2 (con las partes C y D) según un orden predefinido para cada producto. La detección de partes por C1 detiene la banda B1. La identidad de cada parte la determina el sistema de visión cuando llega a la posición de C1. Si la pieza puede ser ensamblada en A1 o A2 de acuerdo al patrón de ensamble, el robot R2 tomará la pieza de B1 y la ensamblará en A1 o A2. En caso de que la pieza no pueda ser ensamblada ahora pero pueda serlo posteriormente se guardará en ST1. Cuando ha sido terminado el ensamble de A1 el robot R2 toma la pieza y la coloca sobre la banda B2, que saca la pieza del sistema. De la misma forma, si el ensamble de A2 está terminado, R2 toma la pieza y la coloca sobre B3, que saca la pieza del sistema."

La clasificación de componentes del sistema para este ejemplo es el mostrado en la Figura 2-3.

De acuerdo con lo descrito antes, el grafo de flujo de partes FP que modela la tarea es el siguiente: FP = (G, SITES, OPER, λ, ϕ) donde

$$G = (V, A)$$

$V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$

$A = \{ (v_0, v_1), (v_1, v_2), (v_1, v_1), (v_1, v_3), (v_3, v_4), (v_4, v_3), (v_3, v_5), (v_3, v_6),$
$(v_5, v_7), (v_6, v_7), (v_7, v_8), (v_7, v_9), (v_8, v_{10}), (v_8, v_{11}) \}$

SITES = {conv1, conv2, conv3, grip1, grip2, assb1, assb2, st1}

ε = {input1, output1, output2, output3}

OPER = {`B1.feed()`, `B1.advance()`, `R1.pick(conv1)`,
`R1.store(st)`, `R1.recover(st)`, `R1.assembly(assb1)`,
`R1.assembly(assb2)`, `R2.pick(assb1)`, `R2.pick(assb2)`,
`R2.place(conv2)`, `R2.place2(conv3)`, `B2.advance()`,
`B3.advance()`, `C1.ident&Loc()`}

$\lambda = \{ (v_0, \text{input1}), (v_1, \text{conv1}), (v_2, \text{output1}), (v_3, \text{grip1}), (v_4, \text{st1}), (v_5, \text{assb1}),$
$(v_6, \text{assb2}), (v_7, \text{grip2}), (v_8, \text{conv2}), (v_9, \text{conv3}), (v_{10}, \text{output2}),$
$(v_{11}, \text{output3}) \}$

$\phi = \{ ((v_0, v_1), \texttt{B1.feed()}), ((v_1, v_2), \texttt{B1.advance()}),$
$((v_1, v_3), \texttt{R1.pick(conv1)}), ((v_3, v_4), \texttt{R1.store(st)}),$
$((v_4, v_3), \texttt{R1.recover(st)}), ((v_3, v_5), \texttt{R1.assembly(assb1)}),$
$((v_3, v_6), \texttt{R1.assembly(assb2)}), ((v_5, v_7), \texttt{R2.pick(assb1),}),$
$((v_6, v_7), \texttt{R2.pick(assb2)}), ((v_7, v_8), \texttt{R2.place(conv2)}),$
$((v_7, v_9), \texttt{R2.place(conv3)}), ((v_8, v_{10}), \texttt{B2.advance()}),$
$((v_8, v_{11}), \texttt{B3.advance()}), ((v1,v1), \texttt{C1.ident&Loc()})\}$

Este modelo de la tarea descrito puede representarse gráficamente tal como se muestra en la Figura 2-5.

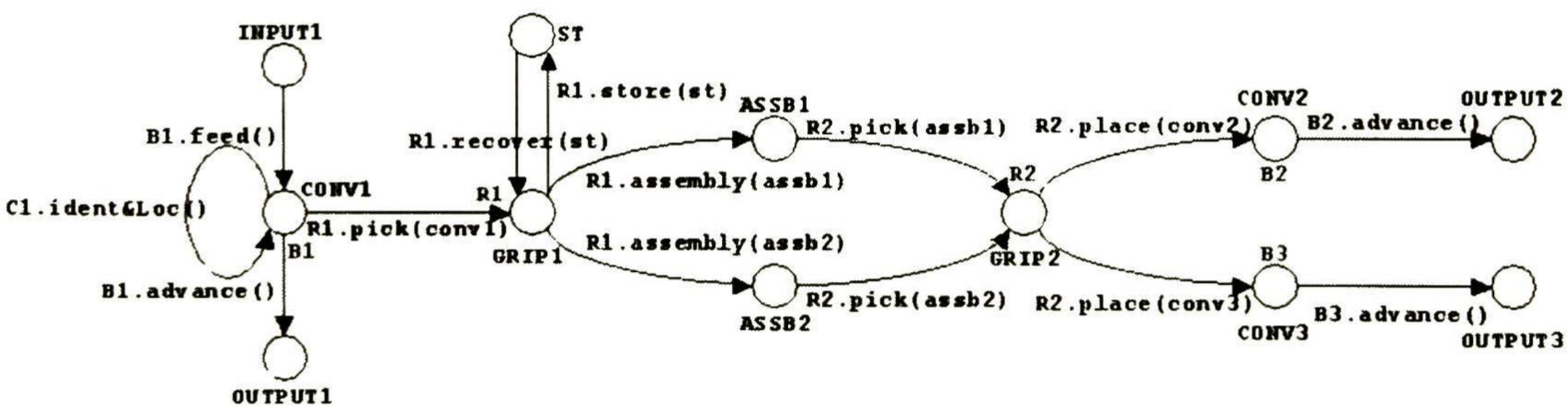

Figura 2-5. Modelo de la tarea.

Las reglas para esta tarea se describen enseguida de manera informal considerando que a cada arco corresponde una regla:

```
// B1.feed()
SI conv1=vacío ENTONCES
   B1.feed();
FIN SI
// B1.advance()
SI conv1<>vacio ∧ no piezaProcesable(conv1) ENTONCES
   B1.advance();
   Actualizar( conv1, vacío );
FIN SI
// C1.ident&Loc()
SI conv1<>vacío ∧ piezaAunNoIdentificada(Conv1) ENTONCES
   C1.ident&Loc();
   Actualizar( conv1, identificarPieza(conv1) );
FIN SI
// R1.pick()
SI conv1=vacio ∧ grip1<>vacio ∧ piezaProcesable(conv1) ENTONCES
   R1.pick();
   Actualizar( grip1, contenido(conv1) );
   Actualizar( conv1, vacío );
FIN SI
// R1.recover()
SI grip1=vacio ∧ st<>vacio ∧  piezaProcesable(ST)
   R1.recover(ST);
   Actualizar( Grip1, contenido(ST) );
   Actualizar( ST, vacío );
FIN SI
// R1.store()
SI st=vacio ∧ no piezaAdmisible(contenido(grip1),assb1) ∧ no
                        piezaAdmisible(contenido(grip1),assb2) ENTONCES
  R1.store(ST);
  Actualizar( ST, contenido(grip1)  );
  Actualizar( grip1, vacio );
FIN SI
// R1.assembly1()
SI grip1<>vacio ∧ piezaAdmisible(contenido(grip1),assb1) ENTONCES
   R1.assembly(assb1);
   Actualizar( assb1, contenido(assb1) );
   Actualizar( grip1, vacio );
FIN SI
// R1.assembly2()
SI grip1<>vacio ∧ piezaAdmisible(contenido(grip2),assb2) ENTONCES
   R1.assembly(assb2);
   Actualizar(assb2, ensamblar(contenido(assb2),contenido(grip1)));
   Actualizar( grip1, vacio );
FIN SI
// R2.pick1()
SI grip2=vacio ∧ assb1<>vacio ∧ ensambleTerminado(assb1) ENTONCES
   R2.pick(assb1);
   Actualizar( grip2, contenido(assb1) );
   Actualizar( assb1, vacio );
FIN SI
// R2.pick2()
SI grip2=vacio ∧ assb2<>vacio ∧ ensambleTerminado(assb2) ENTONCES
   R2.pick(assb2);
   Actualizar( grip2, contenido(assb2) );
   Actualizar( assb2, vacio );
FIN SI
// R2.place1()
SI grip2<>vacio ∧ conv2=vacio ∧ contenido(grip2)=ENSAMBLE1 ENTONCES
   R2.place(conv2);
  Actualizar( conv2, contenido(grip2) );
  Actualizar( grip2, vacío );
FIN SI
// R2.place2()
SI grip2<>vacio ∧ conv3=vacio ∧ contenido(grip2)=ENSAMBLE2 ENTONCES
```

```
   R2.place(conv3);
  Actualizar( conv3, contenido(grip2) );
   Actualizar( grip2, vacío );
FIN SI
// B2.advance()
SI conv2<>vacio ENTONCES
   B2.advance();
   Actualizar( conv2, vacío );
FIN SI
// B3.advance()
SI conv3<>vacio ENTONCES
   B3.advance();
   Actualizar( conv3, vacío );
FIN SI
```

2.5. Partición del modelo de la tarea

La partición del grafo de flujo es necesaria para la obtención de subgrafos que representan subtareas que pueden ser asignadas de manera independiente a los agentes de manufactura.

Los criterios para la partición pueden ser variados de acuerdo al enfoque empleado. Desde partir el grafo según la distribución natural de la tarea de manufactura hasta el máximo número de particiones permitidas para el grafo, pasando por una partición según la similaridad de subgrafos/subtareas en la tarea global.

Los sitios pueden ser *sitios activos* o *sitios pasivos*. Los *sitios activos* están asociados a actuadores como robots. Los *sitios pasivos* están asociados a componentes pasivos tales como mesas de ensamble de piezas.

2.5.1. Algoritmo de partición del modelo

Para realizar el máximo número de particiones válidas en el grafo el criterio es considerar los sitios asociados a los actuadores como pivotes de subtareas; los límites de una subtareas son los sitios predecesores y sucesores.

1. *Identificación de sitios activos y pasivos.*

   Sea *Sitios(G)* el conjunto de sitios del grafo *G*.

   Sea *Activos(G)* el conjunto de sitios asociados a los actuadores. Por lo tanto, $Activos(G) \subseteq Sitios(G)$.

   Sea *Pasivos(G)* el conjunto de sitios no asociados a actuadores. Por lo tanto, $Pasivos(G) \subseteq Sitios(G)$.

2. *Creación de los subgrafos.*

Para cada sitio $s \in Activos(G)$ crear un subgrafo g que incluya el sitio activo s llamado *pivote de la tarea*, los sitios con los que se relaciona el sitio s a través de los arcos, y los arcos que los relacionan. En este subgrafo g el sitio s se sigue considerando un sitio activo y el resto de sitios del subgrafo g ahora se consideran como sitios pasivos. Los subgrafos formarán un conjunto llamado $Subgrafos = \{g_1, g_2, \ldots g_r\}$. Dos subgrafos g_i, g_j son *subgrafos adyacentes* si $Sitios(g_i) \cap Sitios(g_j) \neq \varnothing$.

3. *Eliminación de sitios y arcos duplicados en el sistema.*

   Las operaciones de los arcos del subgrafo g_i se deben referir sólo a los actuadores/sensores controlados de manera exclusiva por el sitio. El sitio posee al menos un actuador, que es el asociado al sitio activo considerado como pivote de la tarea. Esto implica que se deben eliminar los arcos cuyas etiquetas se refieren a operaciones ajenas a dichos actuadores. Asimismo, se eliminan los sitios que resultaran aislados del grafo.

2.5.2. Ejemplo de partición de la tarea

Tomando como base el modelo de la Figura 2-5, se identifican los sitios activos y pasivos de tal forma que los conjuntos se definen como

Activos = {conv1, conv2, conv3, grip1, grip2}

Pasivos = {assb1, assb2, st1}

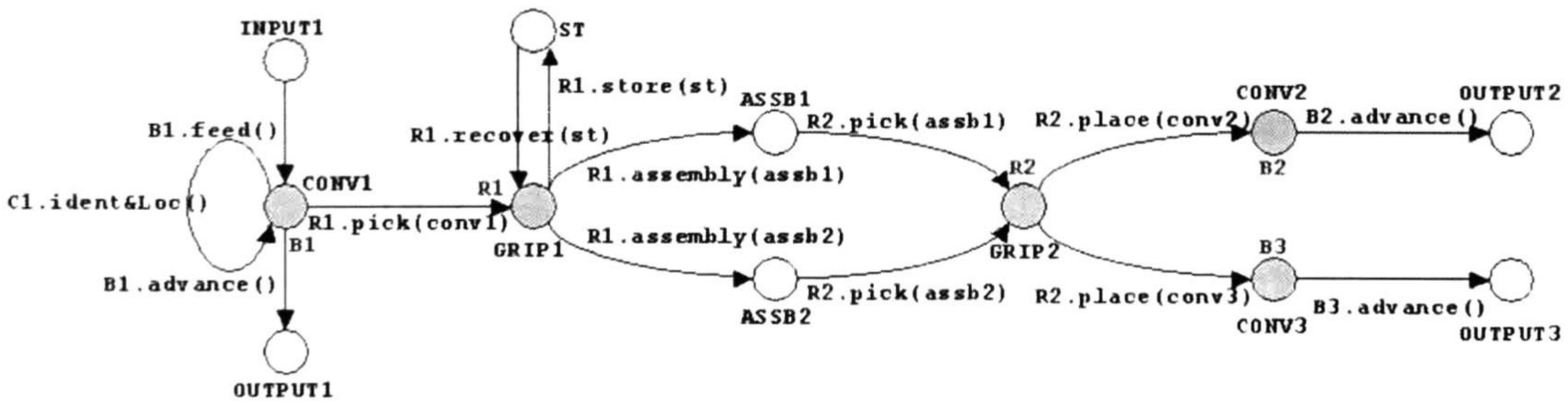

Figura 2-6. Identificación de sitios activos.

Cada sitio activo identificado es tomado como pivote de la tarea, por lo que se obtienen cinco subgrafos que incluye el sitio pivote, los sitios con los que se comunica y los arcos correspondientes.

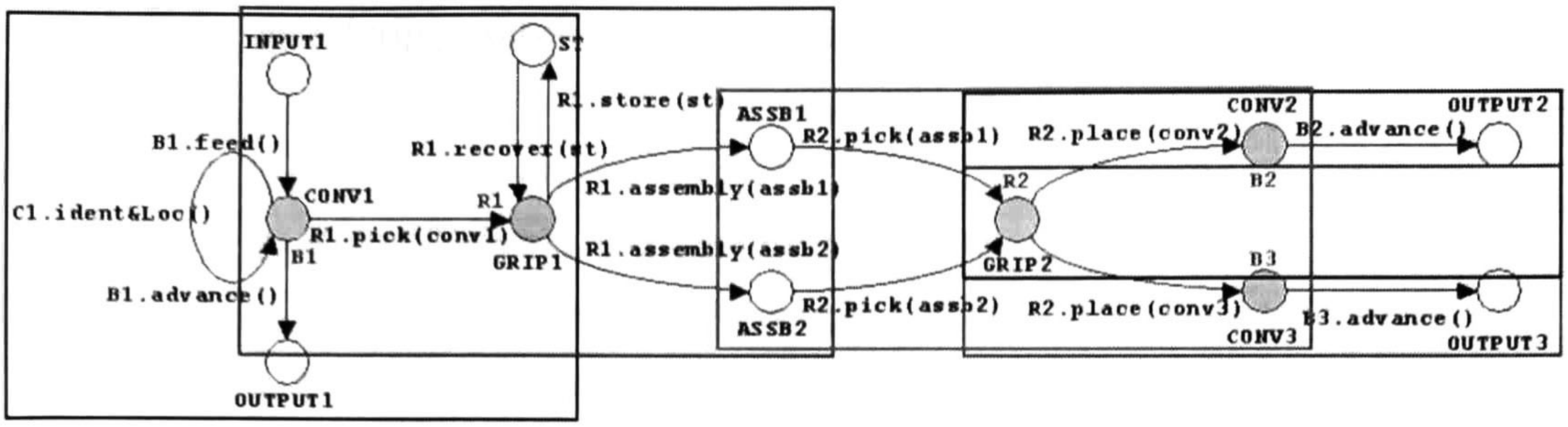

Figura 2-7. Creación de los subgrafos.

En la Figura 2-8 se muestran los subgrafos separados, donde se aprecian los nodos y arcos que componen cada subgrafo.

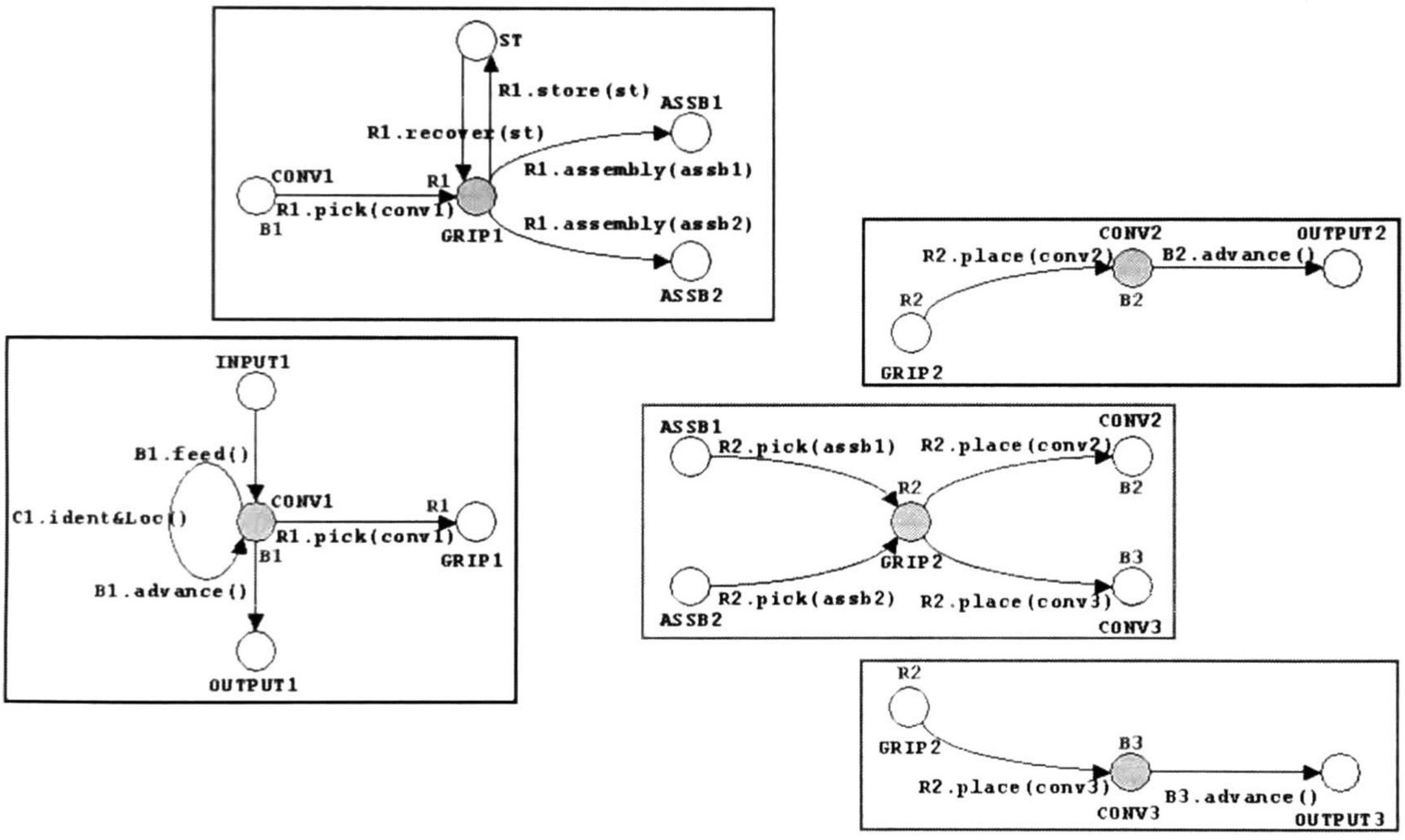

Figura 2-8. Subgrafos iniciales.

Por lo tanto el conjunto *Subgrafos* generado se define como

$$Subgrafos = \{g_1, g_2, g_3, g_4, g_5\}$$

donde

g_1 = {conv1, grip1, input1, output1, `B1.feed()`, `B1.advance()`, `R1.pick(conv1)`, `C1.ident&Loc()`}

g_2 = {grip1, conv1, st, assb1, assb2, `R1.pick(conv1),R1.store(st),`
`R1.recover(st),R1.assembly(assb1),`
`R1.assembly(assb2)}`

g_3 = {grip2, assb1, assb2, conv2, conv3, `R2.pick(assb1),`
`R2.pick(assb2),R2.place(conv2),R2.place(conv3)}`

g_4 = {conv2, grip2, output2, `R2.place(conv1),B2.advance()`}

g_5 = {conv3, grip2, output3, `R2.place(conv2),B3.advance()` }

y cada sitio activo o pivote se muestra subrayado.

Ahora, para cada subgrafo, se eliminan los arcos / operaciones que no se refieren a los actuadores/sensores asociados a los sitios del subgrafo. Al eliminar los arcos asociados a las operaciones algunos sitios pueden quedar aislados, tal como se muestra enseguida.

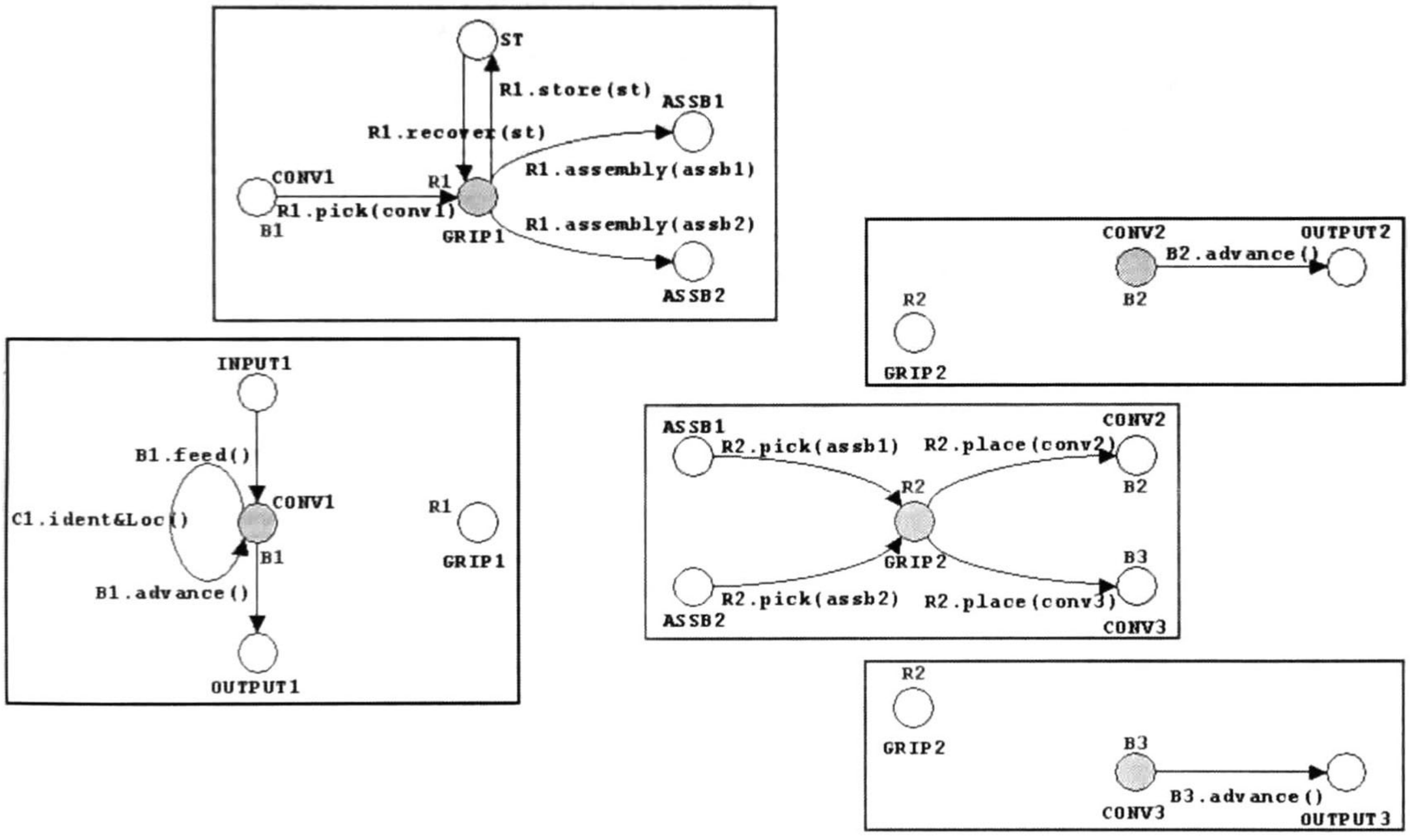

Figura 2-9. Eliminación de arcos.

y los subgrafos poseen los siguientes elementos

g_1 = {conv1, grip1, input1, output1, `B1.feed(),B1.advance(),`
`C1.ident&Loc()}`

g_2 = {grip1, conv1, st, assb1, assb2, `R1.pick(conv1),R1.store(st),`
`R1.recover(st),R1.assembly(assb1),`
`R1.assembly(assb2)}`

g_3 = {grip2, assb1, assb2, conv2, conv3, `R2.pick(assb1),`
`R2.pick(assb2),R2.place(conv2),R2.place(conv3)` }

g_4 = {conv2, grip2, output2, `B2.advance()`}

g_5 = {conv3, grip2, output3, `B3.advance()`}

donde cada sitio activo o pivote se muestra subrayado.

Finalmente se eliminan del subgrafo los sitios que han quedado aislados del subgrafo que contiene el sitio pivote luego del paso anterior. Como resultado se deben tener subgrafos conexos conteniendo el sitio pivote de la tarea.

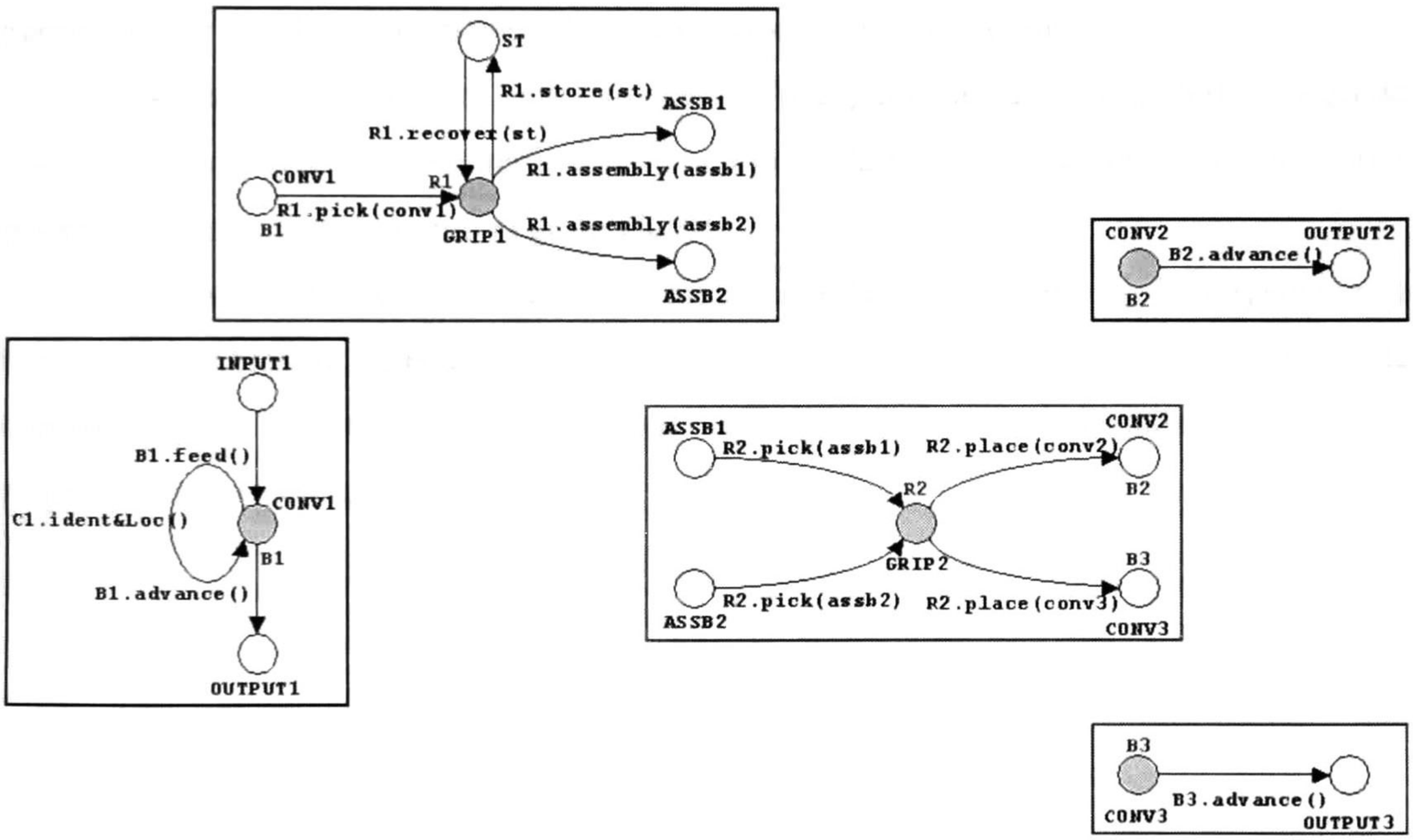

Figura 2-10. Eliminación de sitios aislados.

y los subgrafos resultantes son

g_1 = {conv1, input1, output1, `B1.feed(),B1.advance(),`
`C1.ident&Loc()`}

g_2 = {grip1, conv1, st, assb1, assb2, `R1.pick(conv1),R1.store(st),`
`R1.recover(st),R1.assembly(assb1),`
`R1.assembly(assb2)`}

g_3 = {grip2, assb1, assb2, conv2, conv3, `R2.pick(assb1),`
`R2.pick2(assb2),R2.place(conv2),R2.place(conv3)`}

g_4 = {<u>conv2</u>, output2, `B2.advance()`}

g_5 = {<u>conv3</u>, output3, `B3.advance()`}

donde cada sitio activo o pivote se muestra subrayado.

Una vez que la tarea ha sido partida en subgrafos/subtareas, éstos deben asignarse a los agentes para su programación y posterior ejecución.

Cada subtarea posee solamente un sitio activo que corresponde a un actuador, el cual debe ser administrado por el agente que le corresponda ejecutar la subtarea. Esto permite evitar conflictos de manejo de dispositivos entre varios agentes.

Cada subtarea posee el conocimiento de una parte de la tarea global, agregando como redundancia los sitios interface para conservar la consistencia entre las subtareas y poder actuar en conjunto como el grafo global.

Los sitios interface permiten a las subtareas relacionarse entre sí. Los agentes que sean responsables de ejecutar dichas subtareas deberán cooperar para mantener la consistencia de los sitios compartidos que les corresponden. Los agentes pueden tener asignadas uno o más subgrafos/subtareas.

CAPÍTULO

3

Plataforma de Agentes de Manufactura

En este capítulo se presenta una plataforma para la programación de los agentes que implementan la coordinación de las subtareas del sistema de manufactura. La plataforma está basada en JADE, por lo que primero se hace una breve revisión de sus características y facilidades. Posteriormente se presenta la metodología para la programación los agentes coordinadores.

3.1. Introducción a JADE

3.1.1. Componentes para desarrollo y administración de la plataforma

JADE (Java Agent DEvelopment framework) es una plataforma de desarrollo que permite programar sistemas multiagente y aplicaciones de acuerdo a los estándares de FIPA para agentes inteligentes. JADE ofrece dos grandes ventajas: es una plataforma de agentes que cumple con FIPA y es un paquete para desarrollar agentes en Java. En este trabajo fue usada la JADE versión 1.2.

3.1.1.1. Componentes para la programación de agentes en JADE

JADE está programado en Java. Está compuesto de varios paquetes de Java, que proporcionan al programador componentes de software funcionales ya desarrollados e interfaces para adaptar tareas que dependen de la aplicación.

La razón para usar Java en el desarrollo de la plataforma es por ser un lenguaje orientado a objetos y por su capacidad de estar en ambientes distribuidos y heterogéneos a través de la Serialización de Objetos, el Reflection API y el Remote Method Invocation (RMI).

3.1.1.2. Paquetes de clases para programar agentes en JADE

El lenguaje Java agrupa clases en bibliotecas de clases denominados paquetes. Cada paquete puede contener clases y subpaquetes, que a su vez pueden contener sus propias clases y subpaquetes.

Los paquetes de clases que JADE proporciona para la programación de agentes se describen a continuación.

El paquete `jade.core` implementa el núcleo de la plataforma. Contiene la clase *Agent* que es extendida por los programadores de agentes. En el subpaquete `jade.core.behaviours` está la jerarquía de clases *Behaviour*. Los *comportamientos* implementan las tareas del agente. Estos comportamientos son unidades de actividad lógica que pueden ser compuestas en varias maneras para lograr patrones complejos de ejecución y que pueden ser ejecutados de manera concurrente. Los programadores de la aplicación definen las operaciones del agente implementando comportamientos.

El paquete `jade.lang` Tiene un subpaquete para cada lenguaje usado en JADE. Así, el subpaquete `jade.lang.acl` es proporcionado para procesar el *Agent Communication Language* de acuerdo con las especificaciones del estándar de FIPA. El subpaquete `jade.lang.sl` Contiene el codec del lenguaje SL-0, tanto el parser como el encoder.

El paquete `jade.onto` contiene un conjunto de clases para que el usuario pueda definir sus propias ontologías. El subpaquete `jade.onto.basic` contiene los conceptos básicos (es decir *Action*, *TruePredicate*, *FalsePredicate*, etc.) que generalmente son parte de las ontologías, y una *BasicOntology* que puede usarse en las ontologías definidas por el usuario.

El paquete `jade.domain` contiene las clases de Java que representan las entidades para la administración de la plataforma definidas por el estándar de FIPA, en particular los agentes AMS y DF, que manejan el ciclo de vida y los servicios de páginas blancas y páginas amarillas. El subpaquete `jade.domain.FIPAAgentManagement` contiene la ontología de *FIPA-Agent-Management* y las clases que representan el concepto. El subpaquete `jade.domain.JADEAgentManagement`, por su parte, contiene las extensiones JADE para el *Agent-Management* (p.ej. para seguimiento de mensajes, control del ciclo de vida de los agentes, etc.), e incluye la Ontología y las clases que representan sus conceptos.

El paquete `jade.gui` contiene un conjunto de clases genéricas útiles para crear GUIs para mostrar y editar los *Agent-Identifiers*, *Agent Descriptions*, *ACLMessage*, etc.

El paquete `jade.mtp` contiene una interface de Java que debe implementar cada *Message Transport Protocol* para estar totalmente integrado a la plataforma JADE. Este paquete incluye la implementación de algunos protocolos.

El paquete `jade.proto` contiene clases para modelar protocolos estándar de interacción, como *fipa-request*, *fipa-query*, *fipa-contract-net* y otros definidos por FIPA. También incluye clases para que los programadores de aplicaciones definan sus protocolos propios.

El paquete `fipa` contiene el módulo IDL definido por FIPA para transporte de mensajes basado en IIOP.

3.1.1.3. Herramientas para la administración de la plataforma

JADE incluye algunas herramientas que simplifican la administración de la plataforma y el desarrollo de aplicaciones. Cada herramienta está en un subpaquete de *jade.tools*. Actualmente están disponibles las siguientes herramientas:

- *RMA (Remote Management Agent).* Actúa como una terminal gráfica para la administración y control de la plataforma. Para ejecutar la primera instancia de un RMA puede iniciarse con la opción *-gui* en línea de comandos, y después de pueden activar varias GUI. JADE mantiene la consistencia entre los distitos RMAs usando multicasting de eventos para todos ellos. Esta consola permite ejecutar las demás herramientas.

- *Dummy Agent.* Es una herramienta para monitoreo y depuración, creado con una interface gráfica y usando un agente de JADE. Con la GUI se pueden construir mensajes ACL y enviarlos a otros agentes; también se puede desplegar la lista de todos los mensajes ACL enviados o recibidos, incluyendo el instante para permitir al agente grabar conversaciones y simularlas de nuevo.
- *Sniffer.* Agente que puede interceptar mensajes ACL cuando están en camino, desplegándolos gráficamente usando una notación similar a los diagramas de secuencia de UML. Es útil para depurar las sociedades de agentes al observar cómo intercambian mensajes ACL.
- *SocketProxyAgent.* Es un agente que actúa como un *gateway* bidireccional entre una plataforma JADE y una conexión TCP/IP. Los mensajes ACL, cuando viajan por el servicio de transporte de JADE son convertidos a cadenas ASCII y son enviadas por la conexión de socket. A la inversa, los mensajes ACL pueden ser recibidos por la conexión TCP/IP en la plataforma JADE. Este agente es muy útil para manejar los *firewalls* de la red o para proporcionar las interacciones de la plataforma con los applets de Java de un navegador de internet.
- *DF GUI.* Es una interface gráfica usada por el DF (Directory Facilitator) default de JADE y que también puede ser usada por cualquier otro DF que el usuario pueda necesitar. En algunos casos el usuario necesita crear una red muy compleja con dominios y subdominios de páginas amarillas. Esta GUI permite controlar la base de conocimientos de un DF, para asociar un DF con otro DF, y para controlar de manera remota (registrar/desregistrar/modificar/buscar) la base de conocimientos del DF principal y de los DF hijos del principal (implementando la red de dominios y subdominios).

3.1.2. Creación de sistemas multiagentes con JADE

Las clases de JADE que permiten el desarrollo de sistemas multiagentes garantizan el cumplimiento sintáctico y, hasta donde es posible, el cumplimiento semántico de las especificaciones de FIPA.

3.1.2.1. Arquitectura de la plataforma de agentes

Las especificaciones de FIPA definen las características de la arquitectura de referencia de la plataforma de agentes que se muestra en la Figura 3-1.

En esta arquitectura, el *Agent Management System (AMS)* es el agente que ejerce un control supervisor en el acceso y uso de la Plataforma de Agentes. Hay un AMS por cada plataforma. El AMS proporciona servicios de páginas blancas y de ciclo de vida de los agentes, manteniendo un directorio de identificadores de los agentes (AID, Agent IDentifiers) y sus estados. Cada agente debe registrarse con un AMS para obtener un AID válido.

El *Directory Facilitator (DF)* es el agente que proporciona el servicio de páginas amarillas en la plataforma.

El *Message Transport System*, también llamado *Agent Communication Channel (ACC)*, es el componente de software que se encarga del intercambio de mensajes de la plataforma hacia/desde plataformas remotas.

JADE cumple con la arquitectura de referencia antes descrita. Cuando la plataforma JADE es ejecutada, los agentes AMS y DF son creados inmediatamente, y el módulo ACC es habilitado para el manejo de mensajes.

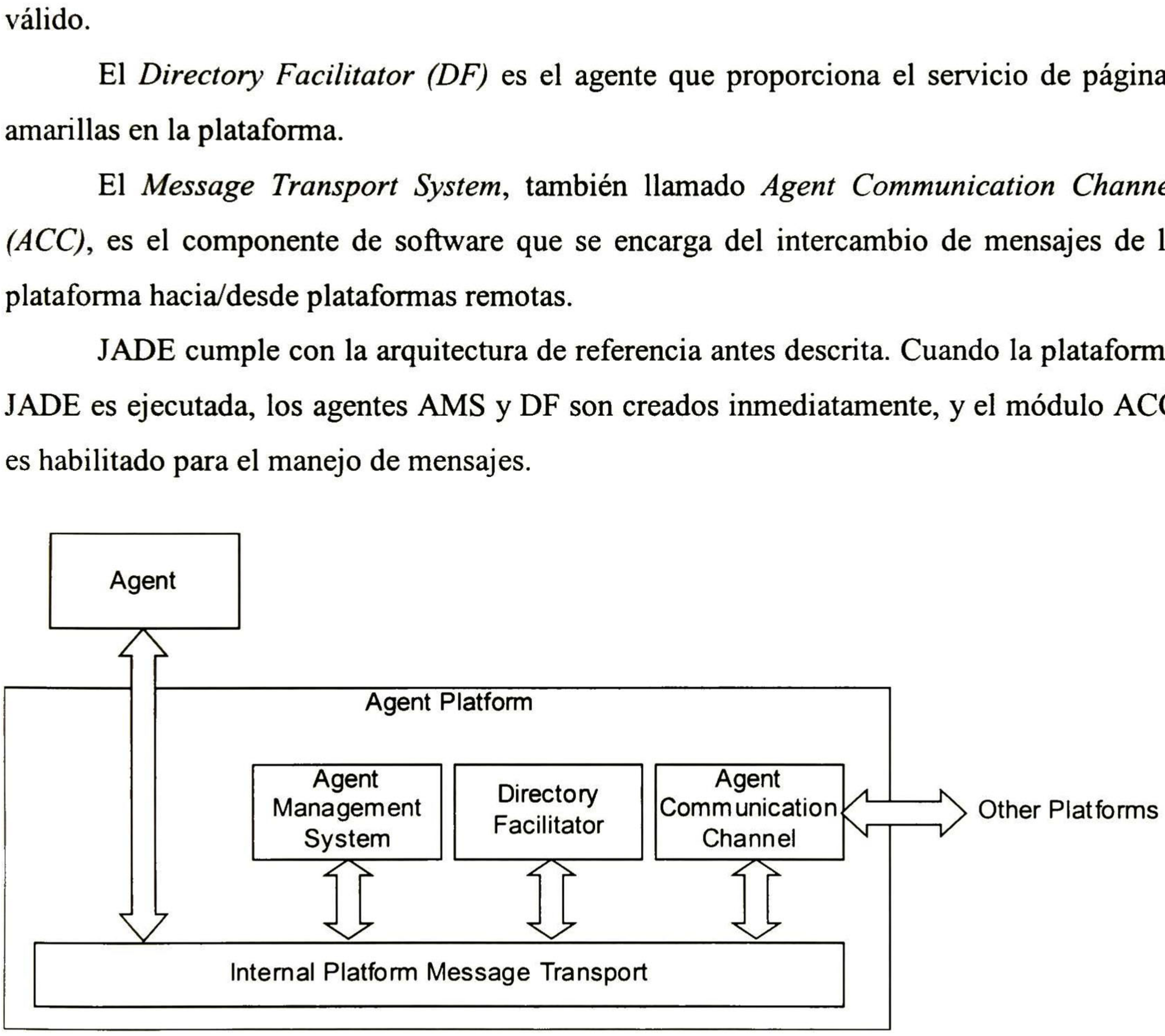

Figura 3-1. Arquitectura de referencia de una plataforma de agentes que cumple el estándar FIPA.

La plataforma de agentes puede estar distribuida en varias computadoras (*host*). En cada host puede ser ejecutada solamente una aplicación de Java usando la Java Virtual Machine (JVM). Cada JVM es un contenedor básico de agentes que proporciona un

ambiente de ejecución para los agentes y permite que varios sean ejecutados de manera concurrente en el mismo host.

Un ejemplo de la plataforma se muestra en la Figura 3-2. El *main-container* es el contenedor de agentes donde residen el AMS y el DF, y donde es creado el registro RMI que usa JADE internamente. Los otros contenedores de agentes se conectan al *main-container* y proporcionan un ambiente para ejecutar agentes de JADE.

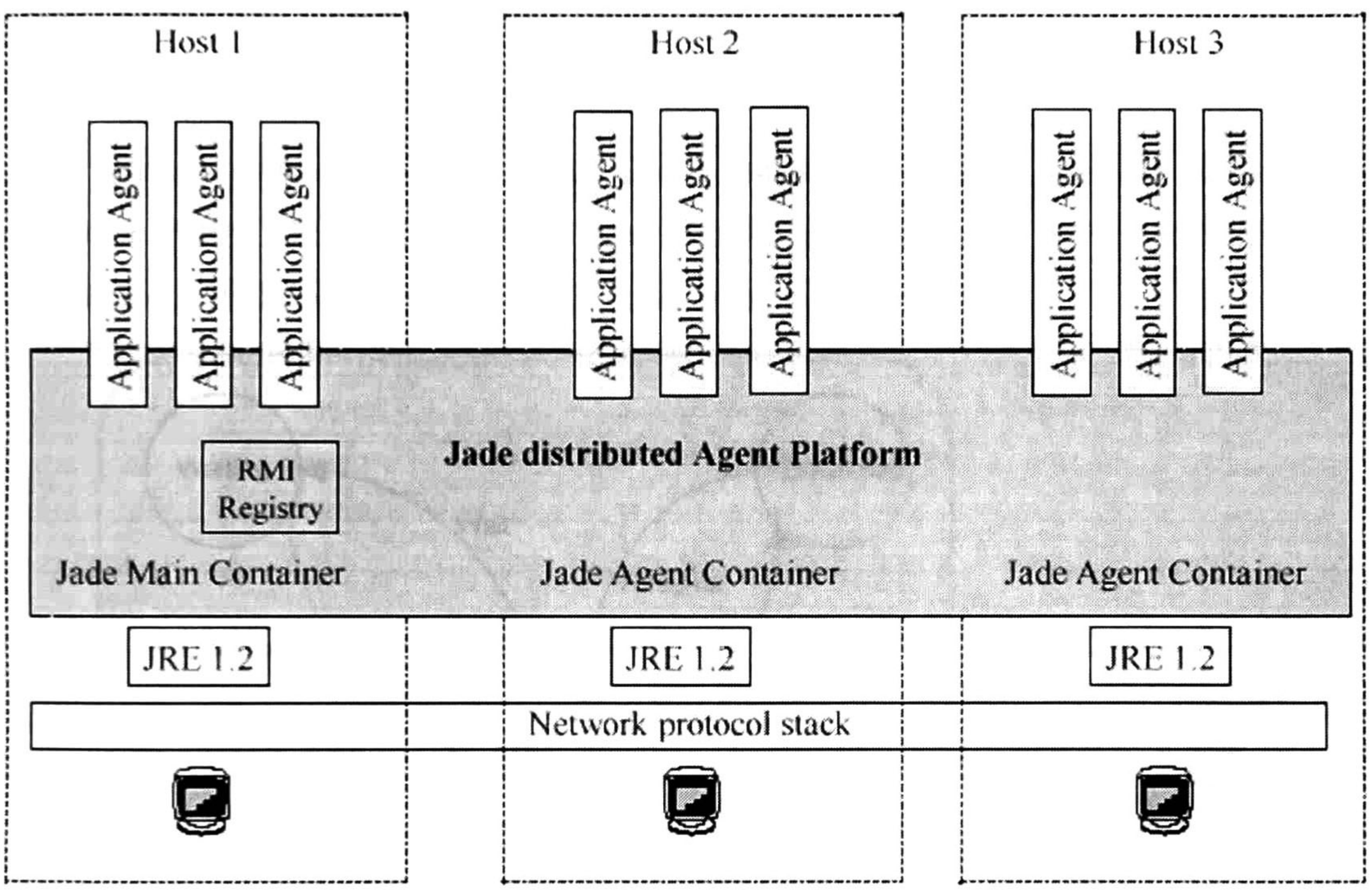

Figura 3-2. Plataforma de Agentes de JADE distribuida en varios contenedores.

3.1.3. Clases para la programación de agentes

3.1.3.1. Clase Agent

La clase *Agent* es la base común de los agentes programados por el usuario. Desde el punto de vista del programador, un agente de JADE es la instancia de una clase de Java definida por el usuario, la cual hereda de la clase base *Agent*. Esto significa que hereda las características para llevar a cabo las interacciones básicas con la plataforma de agentes (registro, configuración, administración remota, etc.) y un conjunto de métodos que pueden usarse para programar el comportamiento particular del agente (como el envío/recepción de

mensajes, el uso de protocolos de interacción estándar, registrarse con varios dominios, etc.).

Los agentes de JADE son multitarea, ya que las tareas/comportamientos son ejecutados de manera concurrente. Las funcionalidades o servicios proporcionados por un agente deben ser implementados como uno o más comportamientos. En la clase base *Agent* hay un planificador de tareas (*scheduler*) que se encarga de la planificación de los comportamientos. Este planificador no es visible para el programador.

3.1.3.2. Ciclo de vida de los agentes de JADE

Cada agente puede estar en un único estado entre varios posibles, tal como lo especifica FIPA. Los estados están representados por constantes en la clase *Agent*. El diagrama de estados y sus transiciones se muestran en la Figura 3-3.

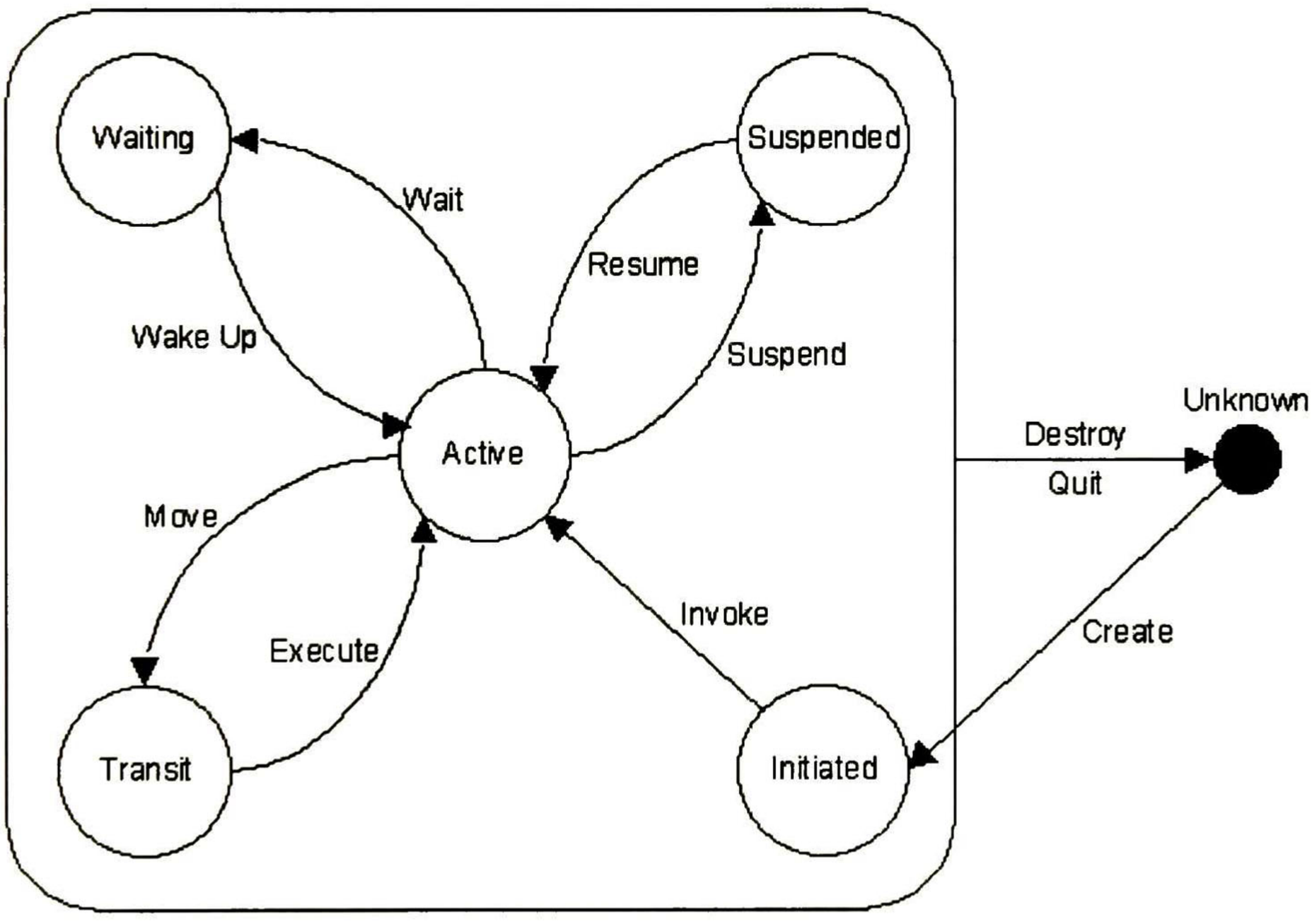

Figura 3-3. Ciclo de vida del agente definido por FIPA.

La clase *Agent* tiene métodos públicos para llevar a cabo las transiciones entre los diferentes estados. Algunos de estos métodos están sobre las transiciones de la Figura 3-3.

Mientras el agente se encuentre en el estado ***Active*** podrá ejecutar sus comportamientos/tareas. Si algún comportamiento llama el método *doWait( )*, el agente pasará al estado ***Waiting*** y sus actividades estarán bloqueadas. En cambio, el método

block(), al ser parte de la clase *Behaviour*, permite suspender un solo comportamiento del agente.

Estado	Descripción
`Initiated`	El objeto de *Agent* ha sido creado pero aún no se registra ante el AMS. No posee nombre ni dirección, y tampoco puede comunicarse con otros agentes.
`Active`	El objeto de *Agent* se registró con el AMS, tiene un nombre, una dirección, y puede usar los servicios de JADE.
`Suspended`	El objeto de *Agent* ha detenido su ejecución. Su hilo interno está suspendido y ningún comportamiento del agente está en ejecución.
`Waiting`	El objeto de *Agent* está bloqueado esperando algo. Su hilo interno está dormido en un monitor de Java y despertará cuando se cumpla la condición (generalmente cuando llega cierto mensaje).
`Deleted`	El objeto de *Agent* ha dejado de existir para el sistema. El hilo interno terminó su ejecución y el Agente ya no está registrado con el AMS.
`Transit`	Un agente móvil entra a este estado mientras está moviéndose a otro lugar. El sistema continúa guardando los mensajes que serán enviados a su nueva ubicación.
`Copy`	Este estado es usado internamente por JADE para agentes que están siendo clonados.
`Gone`	Este estado es usado internamente por JADE cuando un agente móvil se ha ido a una nueva ubicación y tiene un estado estable.

Tabla 3-1. Estados del ciclo de vida de los agentes JADE.

3.1.3.3. Ejecución de los agentes

Siempre que un agente es creado dentro de la plataforma, JADE lleva a cabo los pasos siguientes en ese orden:

1. Se ejecuta el constructor del agente.
2. Se le asigna un identificador al agente.
3. Se registra el agente con el AMS.
4. Se pone al agente en el estado *Active*.
5. Se ejecuta el método *setup()*.

De acuerdo con las especificaciones de FIPA, un identificador de agente AID contiene los atributos siguientes:

- Un *nombre global único*, que en JADE se forma con la concatenación del nombre local dado en la línea de comandos, el símbolo '**@**', y el identificador del host donde reside la plataforma de agentes de JADE, es decir la cadena que contiene el *<nombreDelHost>*:*<puertoDelRegistroRMIdeJADE>*/**JADE**. Por ejemplo:

  `MAgX@gmorales:1099/JADE`
- Un conjunto de direcciones de agentes, ya que cada agente hereda las direcciones del lugar donde reside su plataforma de agentes.

- Un conjunto de servicios de páginas amarillas con los que el agente es registrado.

El método *setup( )* es el punto de partida de las actividades que el agente tiene definidas. Por lo tanto el programador debe implementar el método *setup( )* para inicializar el agente. Cuando se ejecuta el método *setup( )*, el agente ya se registró con el AMS y su estado es *Active*. El programador debe usar éste método de inicialización para:

- Opcionalmente modificar los datos registrados en el AMS si es necesario.
- Opcionalmente asignar la descripción del agente y de los servicios que proporciona. También es opcional registrar el agente con uno o más dominios (*DF*s) en caso necesario.
- Es necesario que se agreguen comportamientos a la cola de tareas listas con el método *addBehaviour( )*. Estos comportamientos son planificados cuando termina de ejecutarse el método *setup( )*.

El método *setup( )* debe añadir al menos un comportamiento al agente. Al finalizar la ejecución del método, JADE ejecuta automáticamente el primer comportamiento de la cola de tareas listas y entonces continúa ejecutando el siguiente comportamiento de la cola usando una planificación Round Robin no apropiable. Para añadir o quitar comportamientos de la cola de tareas se pueden usar los métodos *addBehaviour(Behaviour)* y *removeBehaviour(Behaviour)* de la clase *Agent*.

3.1.3.4. Deteniendo la ejecución de los agentes

Cualquier comportamiento puede detener la ejecución del agente llamando el método *doDelete()* de la clase *Agent*.

Cuando el agente está siendo destruido, antes de que vaya al estado interno *Deleted*, se ejecuta el método *takeDown()* de *Agent*. Este método puede ser redefinido para implementar alguna tarea que se deba realizar el agente al finalizar. Cuando este método es ejecutado, el agente continúa registrado con el AMS y puede enviar mensajes a otros agentes. Antes de que el método termine de ejecutarse, el agente es *"desregistrado"* y su hilo es destruido. El propósito del método es realizar operaciones de limpieza tales como *desregistrarse* de los agentes DF.

3.1.3.5. Comunicación entre agentes

La clase *Agent* tiene métodos para la comunicación entre agentes. Según la especificación de FIPA, los agentes se comunican pasando mensajes asíncronos que intercambian instancias de la clase *ACLMessage*. También hay algunos protocolos de interacción definidos por FIPA que están disponibles como comportamientos listos para usarse, los cuales pueden ser planificados como actividades de los agentes. Estos protocolos están en el paquete *jade.proto*.

El método *send( )* de *Agent* permite enviar un *ACLMessage*. El valor del campo *receiver* contiene la lista de AIDs de los agentes receptores. La llamada del método es transparente respecto a donde se encuentra el agente, es decir, ya sea que el agente sea local o remoto, la plataforma se encarga de seleccionar la ruta y el mecanismo de transporte más apropiado.

3.1.3.6. Acceso a la lista de mensajes privada

Los mensajes que recibe un agente son colocados en su buzón privado por la plataforma de agentes. Hay varios modos de acceso para obtener los mensajes de esta cola privada:

- La cola de mensajes puede ser accesada con bloqueo usando el método *blockingReceive()*, o sin bloqueo usando el método *receive()*. El acceso con bloqueo debe usarse cuidadosamente, ya que suspende las actividades del agente, y en particular la ejecución de sus *Behaviours*. El acceso sin bloqueo inmediatamente devuelve *null* si el mensaje solicitado no está en la cola.
- Los dos métodos mencionados pueden aprovecharse para obtener mensajes de la cola que cumplan con un patrón que describe el mensaje solicitado. Este patrón es pasado como parámetro a los métodos.
- El acceso con bloqueo puede tener un parámetro para un tiempo de espera determinado. Este valor de tipo *long* indica el máximo número de milisegundos que la actividad del agente debe estar bloqueada esperando el mensaje solicitado. Si el tiempo máximo expira antes de que llegue el mensaje, el método devuelve *null*.

- Los comportamientos *ReceiverBehaviour* y *SenderBehaviour* pueden usarse para planificar tareas del agente que necesiten recibir o enviar mensajes.

3.1.3.7. Mensajes en ACL

La clase *ACLMessage* representa mensajes ACL que pueden ser intercambiados entre agentes. Estos mensajes contienen los atributos definidos por las especificaciones de FIPA.

Un agente que desea enviar un mensaje debe crear un objeto de tipo *ACLMessage*, llenar sus atributos con valores adecuados y finalmente llamar el método *send()* de la clase *Agent*. De la misma forma, un agente que está preparado para recibir un mensaje debe llamar el método *receive()* o *blockingReceive()*, ambos implementados por la clase *Agent*.

El envío o recepción de mensajes también puede ser planificado como una actividad independiente del agente añadiendo los comportamientos *ReceiverBehaviour* y *SenderBehaviour* a la cola de tareas del agente.

Los atributos de la instancia de *ACLMessage* pueden ser accesados con métodos set/get y enseguida el nombre del atributo, tal como lo establecen las especificaciones de FIPA. Los métodos *get()* devuelven *null* cuando un atributo aún no ha sido asignado.

`+ACLMessage(p : integer)`	Construye el mensaje con la performativa pasada como argumento.
`+getPerformative():integer`	Devuelve la performativa del mensaje.
`+addSender(idAg:AID)`	Asigna al mensaje el AID del agente que lo enviará. El mensaje tiene solo un solo emisor.
`+addReceiver(idAg:AID)`	Asigna el AID de un agente que recibirá el mensaje. Este método puede usarse varias veces para enviar a diferentes agentes el mismo mensaje.
`+setContent(content:String)`	Asigna el contenido del mensaje.
`+getContent():String`	Devuelve el contenido del mensaje.
`+getSender():AID`	Devuelve el AID del agente emisor del mensaje.

Tabla 3-2. Algunos métodos de la clase *ACLMessage*.

Esta clase también define constantes que deben usarse para referirse a performativas de FIPA, tales como REQUEST, INFORM, etc. Cuando se crea una instancia de *ACLMessage*, debe pasarse una de esas constantes al constructor para seleccionar la performativa del mensaje. El método *reset()* inicializa los valores de los campos del mensaje. El método *toString()* devuelve una cadena que representa el mensaje. En la Tabla 3-2 se describen algunos métodos de la clase *ACLMessage*.

3.1.3.8. Implementación de comportamientos del agente.

Un agente debe ser capaz de realizar varias tareas de manera concurrente respondiendo a los diferentes eventos. Para que la administración del agente sea eficiente, cada agente de JADE está compuesto de un solo hilo de ejecución y todas sus tareas deben ser implementadas como objetos tipo *Behaviour*.

El programador que desee implementar una tarea específica de un agente, debe definir una o más subclases de *Behaviour*, instanciarla y añadir los objetos de comportamiento a la lista de tareas del agente. La clase *Agent*, que debe ser heredada por los programadores del agente, tiene dos métodos: *addBehaviour(Behaviour)* y *removeBehaviour(Behaviour)*, que permiten administrar la cola de tareas listas de un agente específico. Los comportamientos y sub-comportamientos pueden añadirse cuando sea necesario, y no solamente en el método *setup()*. Añadir un comportamiento debe verse como una manera de generar un nuevo hilo de ejecución dentro del agente.

Un planificador, implementado por la clase base *Agent* y oculto para el programador, aplica una política round-robin no apropiativa con los comportamientos disponibles en la cola de tareas listas, ejecutando una clase derivada de *Behaviour* hasta que ésta entregue el control, lo que ocurre cuando retorna del método *action()*. Si la tarea devuelve voluntariamente el control y no ha sido terminada, ésta será replanificada en la siguiente vuelta. Un comportamiento también puede bloquear esperando que llegue un mensaje. En detalle, el agente planificador ejecuta el método *action()* de cada comportamiento presente en la cola de comportamientos listos. Cuando *action()* regresa, el método *done()* es llamado para verificar si el comportamiento terminó su tarea. Si la terminó, el objeto de comportamiento es eliminado de la cola.

Para evitar la espera activa de mensajes, que lleva a un desperdicio de tiempo de CPU, cada comportamiento *Behaviour* permite bloquear su propia ejecución. El método *block()* pone el comportamiento en una cola de comportamientos bloqueados y su efecto se lleva a cabo cuando el método *action()* termina.

Cuando llega un nuevo mensaje, todos los comportamientos bloqueados son despertados y replanificados. Además, un comportamiento puede autobloquearse por un tiempo determinado pasando el tiempo deseado como parámetro al método *block()*. Si el

comportamiento no está interesado en el mensaje que llegó, el programador puede bloquear de nuevo el comportamiento.

Al programar los comportamientos debe evitarse el uso de ciclos infinitos porque el método de planificación es no apropiativo: al ciclarse un agente en un bucle infinito, el comportamiento no puede regresar el control al planificador del agente para continuar con la ejecución de otra tarea. Por la misma razón, los programadores deben evitar operaciones largas dentro de los métodos *action()*. Obviamente, cuando se está ejecutando el método *action()* de un comportamiento, ningún otro comportamiento del mismo agente puede ejecutarse hasta que termine el método.

Además, como no hay pila para guardar información sobre la ejecución de comportamientos, cada que se ejecuta el método *action()* éste lo hace desde el principio. Esto implica que no hay forma de interrumpir un comportamiento a la mitad de su *action()*, ceder el CPU a otros comportamientos y entonces continuar la ejecución del comportamiento desde donde se dejó.

3.2. Plataforma para el desarrollo de agentes coordinadores

3.2.1. Requerimientos de la solución propuesta

Tomando en cuenta que JADE proporciona el entorno operativo de los agentes, ahora se presentan los componentes necesarios y sus requerimientos para programar la infraestructura de base de los agentes de manufactura basados en la metodología de modelado presentada en el capítulo 2.

3.2.1.1. Requerimientos del agente de manufactura

La coordinación distribuida de manufactura involucra equipos de cómputo que pueden ser de fabricantes, arquitecturas y plataformas distintas, pero es necesario que puedan comunicarse a través de mensajes.

Los requisitos que deben cumplir los agentes coordinadores propuestos como solución son los siguientes:

1. *Cada agente tiene un nombre único en el sistema multiagente.*
2. *Administrar el estado de la tarea que le corresponde.*

   Inicializar y actualizar el contenido de los sitios y macrositios del agente.

3. *Intercambiar mensajes con los demás agentes.*

   Enviar / recibir mensajes y mantener las conversaciones.

4. *Procesar los mensajes básicos recibidos de otros agentes.*

   Interpretar y llevar a cabo alguno de los servicios básicos solicitados por los demás agentes, tal como enviar información sobre el estado de un sitio.

5. *Procesar los mensajes específicos recibidos de otros agentes.*

   Interpretar y llevar a cabo alguno de los servicios específicos solicitados por los demás agentes, tal como informar al agente que cierta pieza no será transferida de un sitio a otro.

6. *Coordinarse con los demás agentes para lograr la exclusión mutua de sitios compartidos.* Debe mantener la conversación con los demás agentes que intervienen en la autorización de la exclusión mutua de un sitio.

7. *Controlar los actuadores y sensores que le corresponden.*

   Solicitar a los dispositivos asociados al agente las operaciones necesarias para llevar a cabo la tarea asignada.

3.2.1.2. Requerimientos de los sitios

La discretización del espacio de trabajo permite modelar como *sitios* los lugares donde las piezas son estables en el sistema. Así, se considera que el *estado de la tarea* es la distribución física que tienen en cada momento las piezas en los sitios del sistema de manufactura.

Enseguida se listan los requisitos que deben tener los sitios propuestos como solución para el modelado de la tarea:

1. *Cada sitio tiene un nombre único en el sistema multiagente.*

   Al planear el sistema deben asignarse nombres únicos a cada sitio del sistema.

2. *Administrar su propio estado.*

   Inicializar y actualizar su propio contenido de acuerdo con las necesidades del programador. Cada sitio contiene una pieza o un conjunto de ellas (ensamble). El contenido puede ser removido, sustituido por otra pieza o ensamblado con otras piezas. También debe guardar información sobre cuándo fue modificado su estado por última vez.

3. *Administrar las características del sitio.*

Inicializar y controlar si el sitio está en exclusión mutua o no, si ha solicitado entrar en exclusión mutua, si es un sitio interface cuáles son los agentes que lo comparten, si es un sitio de ensamble cuál es su patrón.

3.2.1.3. Requerimientos de los macrositios

Es útil agrupar sitios para realizar funciones que involucran múltiples sitios en una sola operación global. Un ejemplo es el modelado de los sitios que están sobre un conveyor, donde al mover la banda, las piezas se transfieren de un sitio a otro de manera simultánea. Estos conjuntos de sitios se denominan *macrositios*, los cuales están formados por sitios que cumplen los requisitos mencionados antes.

A continuación se listan los requerimientos de los macrositios propuestos como solución para el modelado de la tarea:

1. *Cada macrositio tiene un nombre único en el sistema multiagente.*

   Al planear el sistema deben asignarse nombres únicos a cada macrositio del sistema.

2. *Administrar su propio estado.*

   Debe inicializar y agregar sitios al macrositio, realizando las operaciones necesarias sobre los sitios cuando sea necesario un comportamiento colectivo.

3. *Aplicar políticas de acceso a los sitios.*

   Cada sitio puede ser accesado con las operaciones definidas de acuerdo a la política de acceso que necesite el programador. Por ejemplo, un macrositio de acceso directo puede obtener una referencia a cualquier sitio usando el nombre del agente o la posición en la lista al ser agregado. En este caso, los sitios pueden modificarse directamente sin involucrar al macrositio.

3.2.1.4. Requerimientos de los dispositivos

Los requerimientos de los dispositivos propuestos para la simulación con los agentes:

1. *Un dispositivo tiene un nombre único en el sistema.*

   Al planear el sistema deben asignarse nombres únicos a los dispositivos.

2. *Manejar su propio estado.*

Debe inicializar y mantener su estado respondiendo adecuadamente a las operaciones del agente.

3. *Cada dispositivo debe mostrar su información y su estado en todo momento.* Para efectos de simulación es necesario monitorear los dispositivos.

3.2.2. Programación de los componentes propuestos

3.2.2.1. Paquetes implementados para los agentes coordinadores

Los componentes implementados están en el paquete `jgpackage` agrupados en cuatro subpaquetes que se muestran en la Figura 3-4.

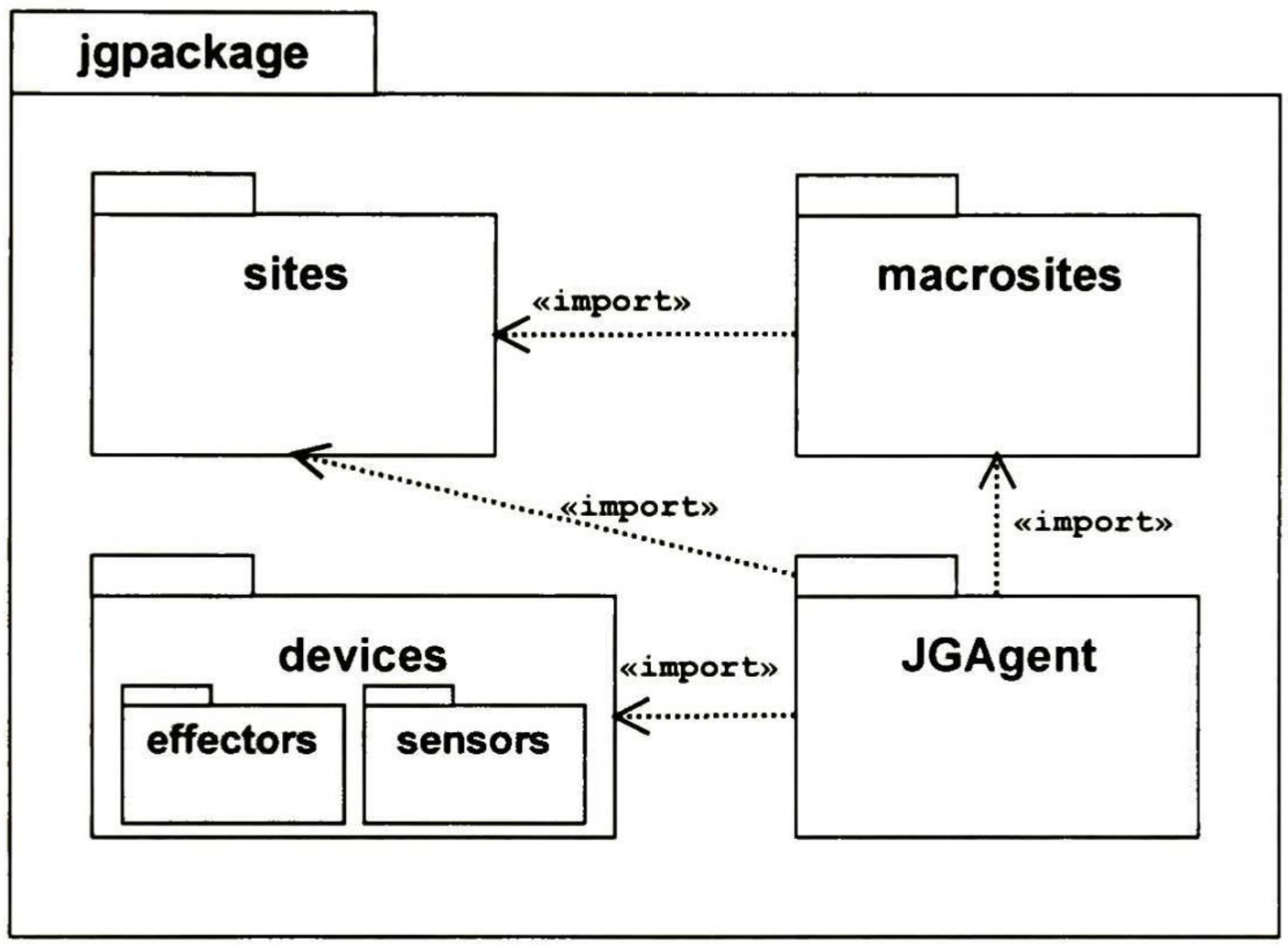

Figura 3-4. Subpaquetes del paquete *jgpackage*.

El subpaquete `jgpackage.sites` contiene las clases *Site* y *Part*, que modelan los sitios de manufactura. El subpaquete `jgpackage.macrosites` contiene las clases *Macrosite* y las relacionadas con el acceso de los sitios en el macrositio. En la Figura 3-5 se muestra la relación entre estos dos subpaquetes.

El subpaquete `jgpackage.devices` se describe en la Figura 3-6 y contiene las clase *Device* y otros subpaquetes. El subpaquete `jgpackage.devices.effectors` contiene la clase *Effector* y algunas especializaciones como *Robot* y *Conveyor*. Al mismo nivel se encuentra el subpaquete `jgpackage.devices.sensors` que contiene la clase *Sensor* y las especializaciones *Camera* y *Detector*.

El subpaquete `jgpackage.JGAgent` contiene la clase *AgentBase* que fue analizada previamente. Esta clase es útil para la programación de los agentes coordinadores. Esta clase importa los tres subpaquetes antes descritos.

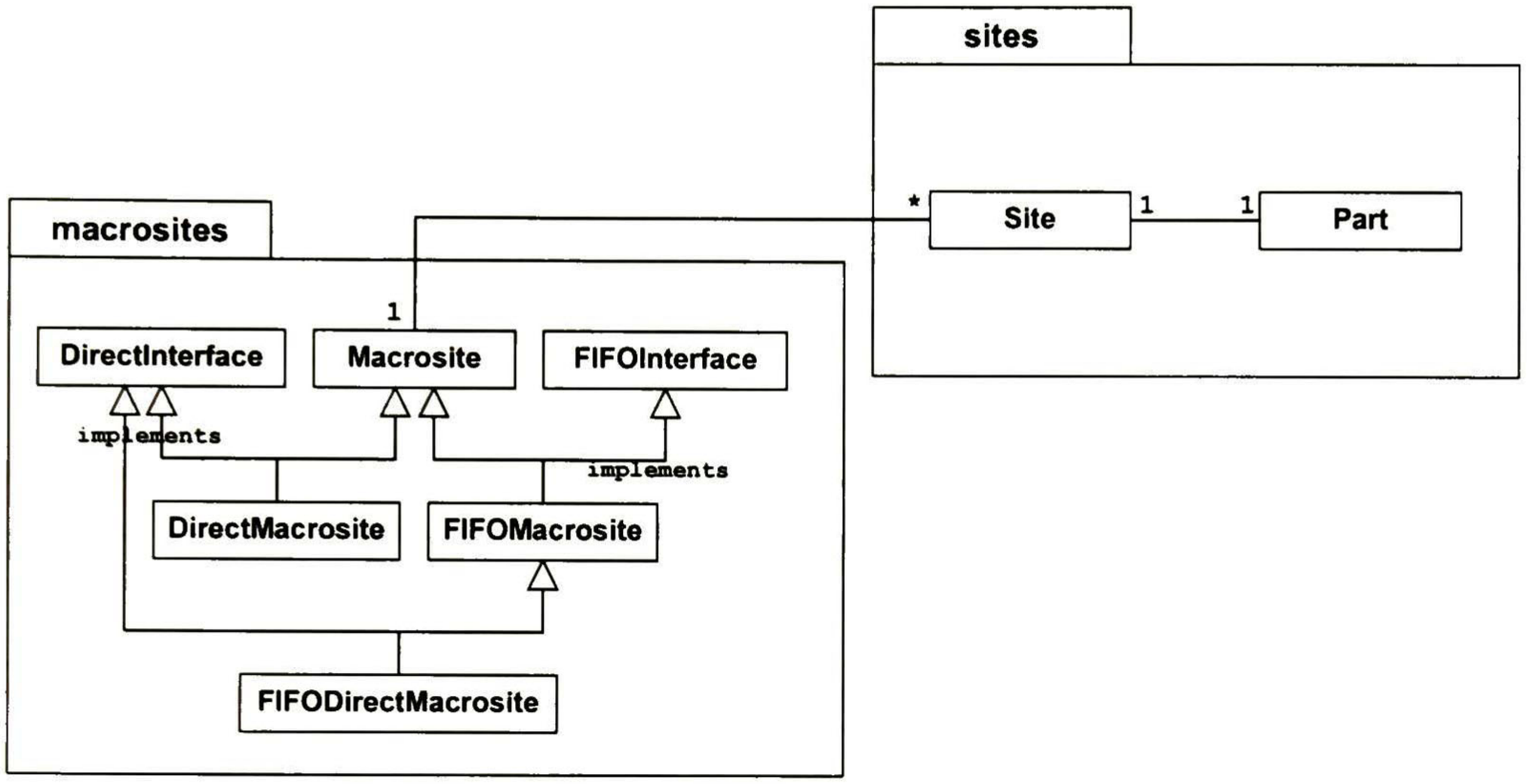

Figura 3-5. Relación de los subpaquetes *sites* y *macrosites*.

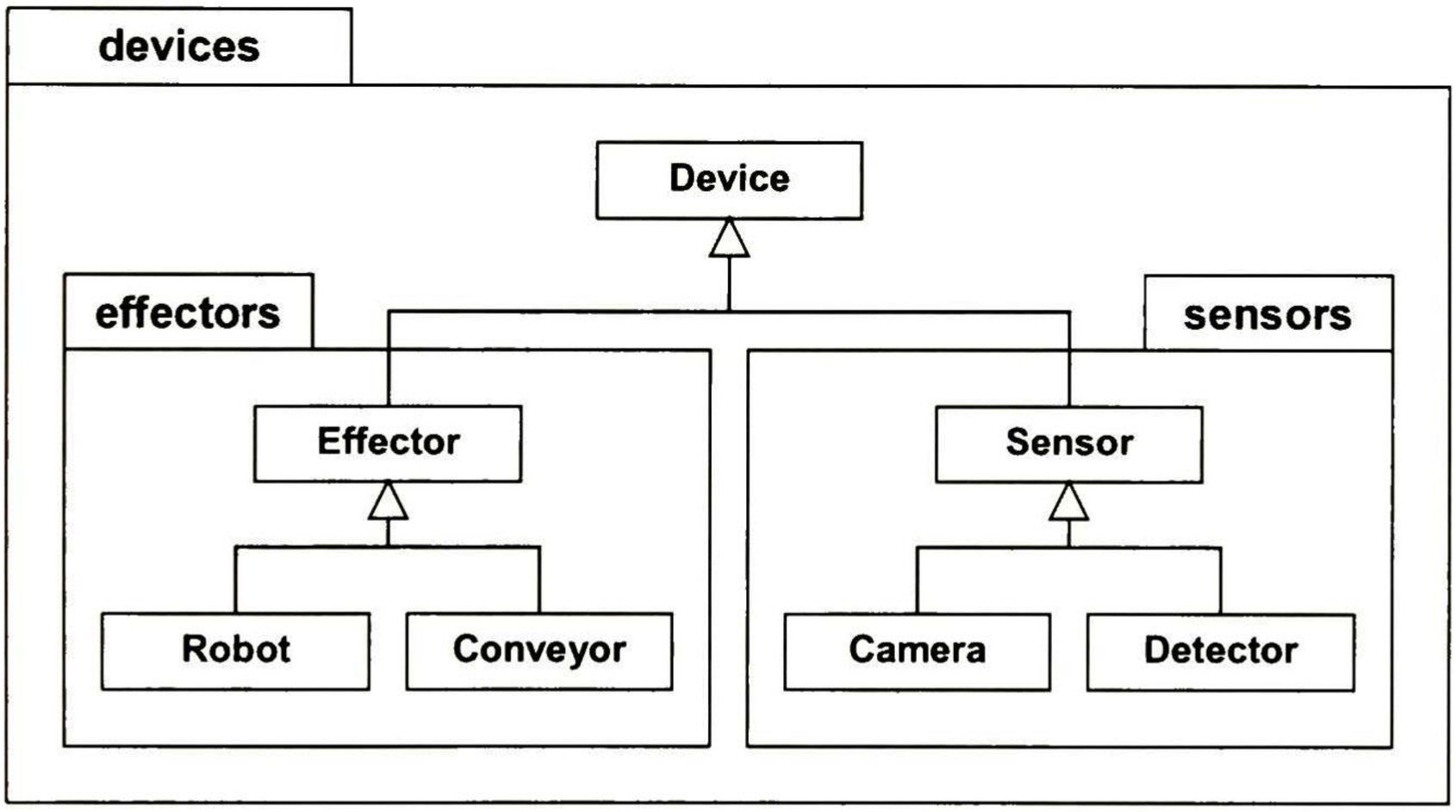

Figura 3-6. Subpaquete *devices*.

3.2.2.2. Clase base de los agentes coordinadores

La clase *AgentBase* es la clase base para los agentes del sistema de manufactura basado en sitios. Esta clase cumple los requerimientos establecidos antes.

AgentBase especializa a la clase *Agent* de la plataforma JADE y tiene programado el comportamiento mínimo necesario para formar parte del sistema de manufactura, por lo

que también se le denomina *agente genérico*. Al programar los agentes del sistema, las clases correspondientes deben extender la clase *AgentBase*, programar la tarea del agente y finalmente instanciar para obtener un agente funcional. La clase *AgentBase* fue creada solo para heredar, por lo que no puede tener instancias. Por eso se dice que *AgentBase* es una clase abstracta.

El agente genérico posee todo lo necesario para la administración de sitios y el procesamiento de mensajes básicos relacionados a la tarea de manufactura basada en sitios. La Tabla 3-3 describe algunos métodos de la clase *AgentBase*.

Método	Descripción
`+AgentBase()`	Llama el constructor de *Agent* e inicializa los componentes agregados al agente genérico para manufactura basada en sitios.
`#setup()`	Método abstracto en *Agent*. En las subclases este método debe programarse para inicializar los componentes del agente. En este método es donde comienza a ejecutarse el agente.
`#rules()`	Método abstracto en *AgentBase* que en las subclases se programa con las reglas que controlan la tarea asignada al agente.
`#processMessages()`	Procesa los mensajes que haya en el buzón del agente. Usa los métodos *processBasicMessage()* y *processSpecificMessage()*.
`#processBasicMessage (msg: ACLMessage):boolean`	Interpreta el mensaje *msg* y realiza la acción correspondiente. Si no reconoce el mensaje devuelve *false*.
`#processSpecificMessage (msg: ACLMessage):boolean`	Método abstracto en *AgentBase*. En las subclases debe interpretar el mensaje especial *msg* y realizar la acción correspondiente. Si no reconoce el mensaje devuelve *false*.
`#sendUpdateMessages (modifiedSite: Site)`	Envía a los agentes que tienen una representación del sitio *nameSite* la información de su estado.
`+addSite(s:Site)`	Registra el sitio *s* en la lista de sitios del agente.
`+getSite(nameSite:String): Site`	Devuelve una referencia al sitio de nombre *nameSite*.
`#updateSite(nameSite, contentSite: String, msgTimestamp:VectorClock)`	Asigna al sitio *nameSite* el contenido *contentSite* siempre que el *msgTimestamp* no sea anterior al registrado por la última modificación del sitio.
`#updateSite(nameSite, contentSite, patternSite:String, VectorClock msgTimestamp)`	Asigna al sitio de ensamble *nameSite* el contenido *contentSite* y el patrón de ensamble *patternSite* siempre que el *msgTimestamp* no sea anterior al registrado por la última modificación del sitio.
`#initializeSites()`	Indaga con los demás agentes del sistema el estado actual de los sitios que posee el nuevo agente.
`#initializeSite(s:Site)`	Indaga con los demás agentes el estado actual del sitio ***s***.
`#getMutex(nameSite:String) :boolean`	Solicita a los demás agentes el uso exclusivo del sitio *nameSite*..
`#freeMutex (nameSite:String)`	Libera el sitio *nameSite* de la condición de exclusión mutua.
`#isMutexAuthorized (siteName: String):boolean`	Devuelve *true* si el sitio *siteName* ha sido autorizado por los demás agentes para ser usado de manera exclusiva.

Tabla 3-3. Métodos de la clase *AgentBase*.

3.2.2.3. Mensajes básicos de los agentes coordinadores

Los agentes coordinadores deben mantener interacciones con los demás agentes para lograr la consistencia en el estado de los sitios en el sistema. La ausencia de conocimiento centralizado y el uso compartido de los sitios hace necesario el intercambio de mensajes para coordinar el uso exclusivo de los sitios.

Mensaje INFOSITE

Este mensaje es enviado por un agente que informa a otros agentes acerca del estado de un sitio o de un macrositio. Los agentes que modifican el estado de sitios interface deben enviar un mensaje *infosite* para informar a los demás agentes.

Mensaje REQUESTINFO

Es enviado cuando un agente solicita a otro, información sobre el estado de un sitio o de un macrositio. Es usado cuando un agente de manufactura comienza su ejecución, debe inicializar sus sitios interface de acuerdo al estado de la tarea del sistema. Para lograr esto, el agente solicita a los agentes con los que comparte sitios que le envíen información sobre el estado de los mismos. Este mensaje se contesta con un mensaje *infosite*.

Mensaje MUTEXREQUEST

El agente emisor de este mensaje solicita el uso exclusivo de un sitio que ambos comparten. Cuando el agente receptor no está interesado en usar el sitio o cede su uso, entonces debe enviar un mensaje *mutexconfirm*.

Mensaje MUTEXCONFIRM

El agente emisor envía la autorización de uso exclusivo de un sitio que comparten. Previamente el agente receptor de este mensaje solicitó el uso exclusivo del sitio con un mensaje *mutexrequest*. Es necesario que todos los agentes implicados en el sitio interface autoricen el uso exclusivo del sitio para que el agente solicitante pueda hacer uso del mismo.

3.2.2.4. Programación de los sitios

La clase *Site* es la clase que modela los sitios en el sistema de manufactura. Esta clase cumple los requerimientos establecidos anteriormente. Los sitios pueden ser:

- *Sitios interface*. Varios agentes tienen representación de este sitio que pueden modificar su estado de manera independiente. La coherencia del estado del sitio en el sistema se mantiene, ya que cuando un agente modifica el estado del sitio, comunica a los demás agentes el nuevo estado del sitio. El sitio tiene registrados a los agentes que lo comparten.

Método	Descripción
`+Site(siteName:String)`	Constructor que recibe el nombre asignado al sitio.
`+getName():String`	Devuelve el nombre del sitio.
`+getContent():Part`	Devuelve el contenido del sitio.
`+setContent(p:Part)`	Asigna el contenido ***p*** al sitio.
`+removeContent()`	Elimina el contenido del sitio.
`+assembly(p:Part)`	La pieza es ensamblada con el contenido del sitio siempre que corresponda con el patrón de ensamble del sitio.
`+getPattern():String`	Devuelve el patrón de ensamble del sitio. Si no es sitio de ensamble devuelve *null*.
`+isAssembled():boolean`	Devuelve *true* si la pieza que contiene el sitio es la que corresponde al patrón de ensamble.
`+isAdmissible(p:Part)`	Devuelve *true* si la pieza puede ensamblarse en el sitio de acuerdo al patrón de ensamble.
`+setVectorClock (vc:VectorClock)`	Guarda el valor que tiene actualmente el reloj del agente.
`+addAgent (agentName:String)`	Registra el nombre del agente que contiene representación de este mismo sitio.
`+setMutex()`	Pone el sitio en exclusión mutua local.
`+isMutex():boolean`	Devuelve *true* si el sitio está actualmente en exclusión mutua.
`+hasRequestedMutex(): boolean`	Devuelve *true* si el sitio ha solicitado la exclusión mutua y está esperando confirmaciones de los agentes.
`+hasAuthorized(int k)`	Devuelve *true* si ya ha confirmado la autorización para el uso exclusivo del sitio el k-ésimo agente registrado en la lista de agentes que comparten este sitio.
`+getAgent(int k):String`	Devuelve el nombre del k-ésimo agente registrado en la lista de agentes que comparten este sitio.
`+isInterface():boolean`	Devuelve *true* si el sitio es interface y *false* en caso contrario.
`+isNonInterface() :boolean`	Devuelve *true* si el sitio es interno y *false* en caso contrario.
`+isRemote():boolean`	Devuelve *true* si el sitio es remoto y *false* en caso contrario.
`+setInterface()`	Caracteríza al sitio como un sitio interface.
`+setNonInterface()`	Caracteríza al sitio como un sitio interno.
`+setRemote()`	Caracteríza al sitio como un sitio remoto.

Tabla 3-4. Algunos métodos de la clase *Site*.

Sitios internos. Son sitios que pertenecen a un solo agente, el cual puede modificar el estado del sitio en cualquier momento.

- *Sitios Remotos*. Es un sitio especial que esta relacionado con un sitio de otro agente. La modificación del sitio remoto no se refleja en el sitio original. Cualquier modificación en el sitio original no será comunicada al agente con el sitio remoto. Por tanto, cuando se necesite conocer el estado actual del sitio se solicita la información al agente con el sitio original.

La Tabla 3-4 describe algunos métodos de la clase *Site* que usan los agentes.

3.2.2.5. Programación de los macrositios

La clase *Macrosite* es la clase base que modela conjuntos de sitios en el sistema de manufactura. Esta clase cumple los requerimientos establecidos previamente. Esta clase es abstracta, por lo que no puede tener instancias. Se desarrollaron dos interfaces de Java:

- *DirectInterface*, que declara operaciones para acceder directamente a los sitios.
- *FIFOInterface*, con operaciones para el acceso tipo cola de los sitios.

Se pueden definir otras políticas de acceso a los sitios de un macrositio extendiendo la clase *Macrosite* o cualquiera de las subclases, programando las operaciones deseadas. En este trabajo se implementaron tres formas de acceso, las cuales se describen enseguida.

La clase *DirectMacrosite* implementa los métodos de *DirectInterface* y extiende la clase *Macrosite*. Los métodos de esta clase se describen en la Tabla 3-5.

Método	Descripción
`+DirectMacrosite(msname:String)`	Construye el macrositio de acceso directo con el nombre dado como argumento.
`+add(s:Site)`	Agrega un sitio al macrositio.
`+size():integer`	Devuelve la cantidad de sitios que están en el macrositio.
`+get(k:integer):Site`	Devuelve una referencia al k-ésimo sitio registrado en el macrositio.
`+get(sitename:String):Site`	Devuelve una referencia al sitio registrado en el macrositio cuyo nombre es el dado como argumento.
`+set(parts:String):boolean`	Descompone el argumento en tokens y asigna a los sitios el contenido correspondiente.

Tabla 3-5. Métodos de la clase *DirectMacrosite*.

La clase *FIFOMacrosite* implementa los métodos de *FIFOInterface* y extiende la clase *Macrosite*. Los métodos de esta clase se describen en la Tabla 3-6.

La clase *FIFODirectMacrosite* implementa los métodos de *DirectInterface* y extiende la clase *FIFOMacrosite*, y contiene los métodos de las dos clases descritas antes.

3.2.2.6. *Programación de los dispositivos*

La clase *Device* es la clase base que modela los dispositivos que controla el agente. Esta clase es extendida para programar dispositivos actuadores y sensores. Las clases abstractas *Effector* y *Sensor* modelan los dispositivos actuadores y sensores en el sistema respectivamente.

Método	Descripción
`+FIFOMacrosite(msname:String)`	Construye el macrositio de acceso FIFO con el nombre dado como argumento.
`+add(s:Site)`	Agrega un sitio al macrositio.
`+size():integer`	Devuelve la cantidad de sitios que están en el macrositio.
`#get(k:integer):Site`	Devuelve una referencia al k-ésimo sitio registrado en el macrositio.
`#get(sitename:String):Site`	Devuelve una referencia al sitio registrado en el macrositio cuyo nombre es el dado como argumento.
`+set(parts:String):boolean`	Descompone el argumento en tokens y asigna a los sitios el contenido correspondiente.
`+insert(p:Part)`	Inserta la pieza *p* recorriendo el contenido de los sitios al siguiente sitio. La pieza que está al frente de la cola de sitios será removida.
`+insert(p:String)`	Inserta la pieza *p* recorriendo el contenido de los sitios al siguiente sitio. La pieza que está al frente de la cola de sitios será removida.
`+remove():Part`	Elimina de la cola de sitios el contenido del último sitio, recorriendo el contenido de los sitios al siguiente sitio. Devuelve la referencia a la pieza removida.
`+lastRemoved():Part`	Devuelve la referencia a la última pieza removida de la cola de sitios.

Tabla 3-6. Métodos de la clase *FIFOMacrosite*.

Finalmente se extienden estas clases para proporcionar soporte a diversos componentes. Las clases *Robot* y *Conveyor* heredan de *Effector* y contienen los métodos básicos para simular estos actuadores. La clases *Camera* y *Detector* heredan de *Sensor* y contienen los métodos necesarios para simular este tipo de sensores.

Finalmente se deben extender las clases descritas para programar los dispositivos con las operaciones que serán usadas en las reglas del agente. En general, el dispositivo debe tener un método para cada regla que lo involucre en la transferencia o procesamiento de piezas.

3.2.3. Tecnologías de JADE para la cooperación de agentes

Los estándares de plataformas de agentes en la industria permiten la cooperación de agentes independientemente de su programación interna. Las implementaciones concretas

de plataformas basadas en estándares aceptados permite la creación de sistemas multiagentes sin limitaciones en la comunicación con otros sistemas.

Estas plataformas proporcionan un entorno para la ejecución de los agentes y proporcionan al programador un conjunto de servicios disponibles para crear sistemas multiagente. De esta manera el programador puede concentrarse en las tareas propias de los agentes y del sistema que desarrolla.

La plataforma JADE usa la tecnología RMI para la comunicación entre plataformas basadas en Java y aprovecha CORBA con el protocolo IIOP para la intercomunicación entre plataformas que cumplen con FIPA pero no están basadas en JADE.

CAPÍTULO

4

Programación del Sistema Multiagente

En este capítulo se presenta la segunda parte de la metodología de construcción de sistemas de coordinación de actividades de FMS.

Ahora se describirá la forma de programar los agentes partiendo del subgrafo/subtarea asignado al agente de manufactura.

4.1. Identificación de los componentes

Para llevar a cabo la programación de los agentes primero se deben identificar los componentes que debe contener cada agente en particular de acuerdo a la subtarea que le corresponde controlar en el sistema de manufactura.

Los componentes del agente se identifican a partir del subgrafo asignado al mismo agente. Se refiere a los sitios, las reglas, los agentes que comparten los sitios, los actuadores y sensores involucrados así como mensajes específicos para el agente.

Para identificar los sitios se debe (1) obtener los nombres de los sitios que manejará cada agente a partir del subgrafo asignado al mismo agente; enseguida (2) revisar para cada sitio del grafo, cuáles agentes comparten el mismo sitio y al configurar el sitio registrar los nombres de los agentes con el método `addAgent`; después (3) revisar cada sitio para determinar si es un sitio remoto para asociarlo con el sitio/agente remoto correspondiente;

finalmente (4) revisar cada sitio para verificar si es un sitio de ensamblado de partes para asignarle el patrón de ensamble que debe producir.

Los dispositivos asociados a los sitios son actuadores (effectors) o sensores. Un sitio puede carecer de dispositivos asociados mientras otros pueden tener varios dispositivos. Para identificarlos se debe analizar la descripción del sistema y cada partición del modelo de la tarea.

Cuando un sitio interface está asociado a dispositivos, solamente uno de los agentes que posee representación del sitio debe mantener el control de ellos para evitar condiciones de competencia con otros agentes. Se propone que el agente controle únicamente los dispositivos que maneja directamente la computadora que lo ejecuta.

Se debe obtener una tabla conteniendo, para cada agente, el conjunto de sitios y dispositivos asociados a cada uno de ellos. Esto debe basarse en la partición del modelo de la tarea y la descripción de la misma tarea para obtener los dispositivos asociados a cada sitio, además de las operaciones que se aplicarán a los dispositivos. La Tabla 4-1 muestra la tabla correspondiente a la tarea de ensamble del ejemplo.

Enseguida se deben obtener las reglas utilizando la partición del modelo de la tarea y la tabla de componentes de los agentes.

4.2. Redacción y obtención de las reglas

Las reglas son construcciones si-entonces, de tal manera que si se cumplen las *precondiciones* establecidas entonces se ejecutan las *acciones* asociadas a la regla.

Para identificar las reglas se debe (1) obtener el nombre de las reglas que tendrá cada agente a partir del subgrafo asignado y (2) determinar las precondiciones, acciones y poscondiciones asociadas a cada regla con base en la especificación del sistema y de la tarea.

Agent name	Site name	Site type	Number of places	Agent	Related devices	Device's operations
MAg1	Conv1	Container site.	1	MAg2	Conveyor1	Feed, Stop
					Camera	Identify
					Sensor	Detect
MAg2	Conv1	Container site.	1	MAg1		
	Grip1	Container site.	1		Robot1	Get, place, assembly.
	Assb1	Assembler site.	2	MAg3		

	Assb2	Assembler site.	2	MAg3		
	ST	Container site.	4			Get, place, assembly.
MAg3	Assb1	Assembler site.	2	MAg2		
	Assb2	Assembler site.	2	MAg2		
	Grip2	Container site.	1		Robot2	Get, place, assembly.
	Conv2	Container site.	1	MAg3		
	Conv3	Container site.	1	MAg4		
MAg4	Conv2	Container site.	1	MAg3	Conveyor2	Feed, Stop
MAg5	Conv3	Container site.	1	MAg4	Conveyor3	Feed, Stop

Tabla 4-1. Componentes de los agentes.

Las *precondiciones* son un conjunto de condiciones que al cumplirse permiten la ejecución de las acciones asociadas a la regla. Las precondiciones pueden referirse a estados de los sitios, de los actuadores y de los sensores.

Las *acciones* son un conjunto de instrucciones relacionadas con el envío de comandos a los actuadores y el cambio de estado en los sitios.

Las *poscondiciones* son los estados finales resultantes tanto de los sitios como de los dispositivos luego de la ejecución de la regla.

Para describirlas se empleará una plantilla/formato que contenga el nombre del agente al cual pertenece la regla, el nombre de la regla, una breve descripción, los sitios y dispositivos involucrados, y finalmente las precondiciones, acciones y postcondiciones de la regla.

Como antes se estableció, el estado del agente está dado por el estado de los dispositivos y sitios que posee. Así, las reglas hacen referencia a un conjunto de estados específico que en determinado momento permiten disparar acciones del agente.

En la Tabla 4-2 muestran las reglas obtenidas para el agente MAg1. Por conveniencia, el agente posee un estado interno que permite diferenciar dos estados: cuando se ha identificado una pieza y cuando se rechaza el uso de la pieza para ensamble. El estado de sitios y dispositivos para los dos estados es exactamente el mismo, por lo que se debió añadir un estado interno de la tarea del agente para diferenciar las dos etapas. Los estados y transiciones de la tarea se muestran en la Figura 4-1.

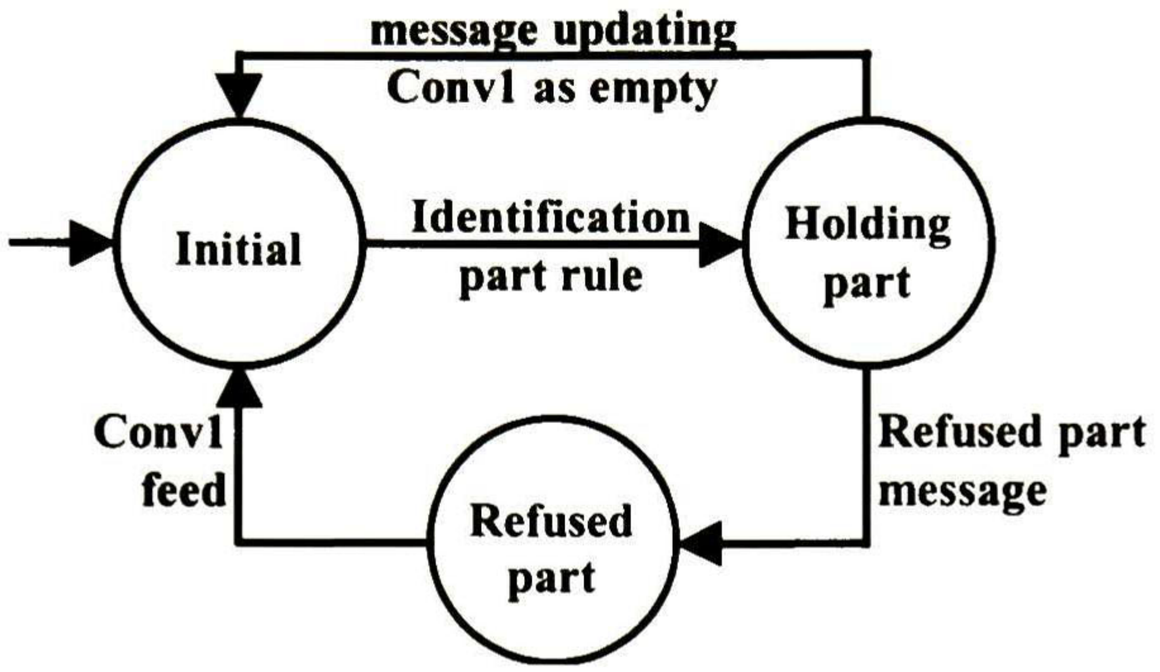

Figura 4-1. Estados y transiciones de la tarea del agente MAg1.

Operation	**Feed*B1**
Description	Si no hay piezas sobre la banda transportadora detenida, se pone en funcionamiento para alimentar el sistema.
Related sites	Conv1
Related devices	Conveyor1, Camera, Detector.
Preconditions	Conv1 está vacío, el estado de Conveyor1 es STOP, del Detector es IDLE y el estado de la tarea es STARTING.
Actions	Poner Conveyor1 en funcionamiento. Inicializar Camera y Detector.
Postconditions	El estado de Conveyor1 es FEED, de Camera es IDLE y del Detector es DETECT.
Operation	**Ident-Loc**
Description	Si durante la alimentación de la banda se detecta una pieza, se detiene la banda, se identifica la pieza e inicializa el detector.
Related sites	Conv1
Related devices	Conveyor1, Camera, Detector.
Preconditions	Conv1 está vacío, el estado de Conveyor1 es FEED, el Detector es DETECTED y el estado de la subtarea es INITIAL.
Actions	Detener Conveyor1. Identificar pieza con Camera. Inicializar Detector.
Postconditions	Conv1 contiene una pieza, el estado de Conveyor1 es STOP, de Camera es IDLE y del Detector es IDLE. El estado de la tarea es HOLDINGPART.
Operation	**Av*B1**
Description	En caso de que la pieza no sea tomada el sistema la desecha.
Related sites	Conv1
Related devices	Conveyor1, Camera, Detector.
Preconditions	Conv1 contiene pieza, el estado de Conveyor1 es STOP, y el Detector es IDLE y el estado de la tarea es REFUSEDPART.
Actions	Detener Conveyor1. Identificar pieza con Camera. Inicializar Detector.
Postconditions	Conv1 contiene una pieza, el estado de Conveyor1 es STOP, de Camera es IDLE y del Detector es IDLE. El estado de la tarea es INITIAL.

Tabla 4-2. Identificación de reglas del agente MAg1.

4.2.1. Programación de las reglas

La programación de las reglas debe basarse en los formatos ya descritos. Solamente resta traducir en código para Java las precondiciones, acciones y postcondiciones de cada

regla. Por ejemplo, el código mostrado en el Listado 4-1 es la programación de las reglas del agente ManufacturingAgent3.

```
// implementing rules for the agent named ManufacturingAgent3
void rules()
{
   // declaring reference to site for place the part
   Site destinationPart = null;

   // GRIP2.TAKE1: Grip2 takes the assembled part from ASSB3
   if(assb3.getSite(0).isAssembled() && grip2.isEmpty() &&
                        conv2.isEmpty() && destinationPart==null)
   {
      grip2.setContent( assb3.getContent(0) );
      assb3.removeContent(0);

      sendUpdateMessages(grip2);
      sendUpdateMessages(assb3);

      //
      destinationPart = conv2;
   }
   // GRIP2.TAKE2: Grip2 takes the assembled part from ASSB4
   if(assb4.getSite(0).isAssembled() && grip2.isEmpty() &&
                        conv3.isEmpty() && destinationPart==null)
   {
      grip2.setContent( assb4.getContent(0) );
      assb4.removeContent(0);

      sendUpdateMessages(grip2);
      sendUpdateMessages(assb4);

      destinationPart = conv3;
   }
   // GRIP2.TAKE: Grip2 places the took part in CONV3 takes the
   //             assembled part from ASSB4
   if(!grip2.isEmpty() && destinationPart.isEmpty() &&
                                          destinationPart!=null)
   {
      destinationPart.setContent(grip2.getContent());
      grip2.removeContent();

      sendUpdateMessages(destinationPart);
      sendUpdateMessages(grip2);

      destinationPart = null;
   }
}
```

Listado 4-1. Programación de las reglas del agente.

4.3. Definición de mensajes específicos

Los mensajes genéricos de los agentes de manufactura surgieron de las necesidades encontradas en los ejemplos implementados. El envío/recepción de estos se realiza de manera transparente sin gran complejidad para el programador.

Cuando es necesario que dos o más agentes intercambien mensajes no-genéricos se debe implementar un mensaje específico. Se pueden identificar las características del mensaje específico empleando un formato semejante al mostrado enseguida.

En los ejemplos fue necesario implementar un mensaje específico cuando no es posible diferenciar entre dos estados temporalmente adyacentes en la tarea de manufactura. Para diferenciar el estado de la tarea de manufactura del agente se incluyeron estados en la tarea cuyo diagrama de transiciones se muestra en la Figura 4-1. Las caracterísiticas del mensaje específico denominado *RefusedPart* se muestran en la Tabla 4-3.

Specific Message	**RefusedPart**
Description	MAg2 informa al agente MAg1 la imposibilidad de utilizar la pieza que se encuentra en el sitio Conv1.
Source agent	MAg2
Target agent	MAg1
Performative	`INFORM`
Content	`RefusePart Conv1 <refused-part>`
Preconditions in source agent	Conv1 no está vacío y la parte que contiene no es posible ensamblarla ni ponerla en la mesa que guarda piezas temporalmente.
Actions in target agent	Verifica que <refused part> sea el contenido de Conv1 y el estado del agente no sea REFUSEDPART para poner el estado del agente en REFUSEDPART.
Postconditions in target agent	El estado del agente pasa a REFUSEDPART.
Comments	Ninguno.

Tabla 4-3. Identificación del mensaje *RefusedPart*.

4.3.1. Programación de los mensajes específicos

La programación de los mensajes específicos debe realizarse en los agentes involucrados según el formato de características.

Todo agente emisor debe programar este mensaje definiendo un método para enviar el mensaje. El envío del mensaje deberá ser realizado al ejecutarse alguna regla que use el método definido. Todo agente receptor debe programar la recepción del mensaje redefiniendo el método `processSpecificMessage(ACLMessage)` de tal forma que procese el mensaje específico definido y ejecute el nuevo comportamiento.

4.4. Programación de las clases de los agentes

Para la programar las clases de cada agente resultante se llevan a cabo los siguientes pasos en ese orden:

1. Se crea la clase del agente especializando la clase base.
2. Se declaran los sitios identificados.
3. Se crean instancias y configuran los actuadores y sensores.
4. Finalmente se programan las reglas identificadas.

La programación de las clases de los agentes implica la especialización de la clase base propuesta en este trabajo. Dicha clase posee las funcionalidades básicas para la programación sencilla de agentes de coordinación de tareas basados en el grafo de la subtarea correspondiente.

Para programar las clases de los agentes se debe heredar de la clase base AgentBase propuesta. Por ejemplo, para programar la clase *ManufacturingAgent1* que hereda de la *AgentBase* se crea un nuevo archivo llamado ManufacturingAgent1.java y en la declaración se hereda como se muestra en el Listado 4-2

```
import JGAgent.*;
import devices.effector.*;
import devices.sensor.*;

public class ManufacturingAgent1 extends AgentBase {
   private Site conv1; // declaring site

   public ManufacturingAgent1() // constructor definition
   {
      // site's initialisation and configuration
      conv1 = new Site(Site.INTERFACE);
      ..
      // configuring site as interface with mag2 agent
      conv1 = new Site("Conveyor1"); // site's
      conv1.setInterface();
      conv1.addAgent("mag2@gmorales/JADE:1099");

      // adding new site to agent site list.
      allSites.put(conv1);
      ...
   }
}
```

Listado 4-2. Declaración de la clase del agente.

4.4.1. Declaración de sitios en el agente

La declaración de sitios implica que se tiene el nombre de los sitios, el número de lugares internamente, el tipo de sitio y, en su caso, los agentes que lo comparten como sitio interface o remoto.

Por ejemplo, la manera de declarar los sitios del agente *ManufacturingAgent1* mencionado antes, dentro de la declaración de la clase se muestra en el Listado 4-2.

4.4.2. Declaración de dispositivos

La programación de clases que modelen a los dispositivos actuadores y sensores ha sido realizada para fines académicos. En la realidad se espera que se posean los drivers necesarios para el control de los dispositivos y una interface de software adecuada para la programación de clases con los métodos que en los ejemplos se usan.

Cada dispositivo será manejado a través de una instancia de dispositivo, la cual tendrá el control del dispositivo físico. Cada dispositivo debe ser controlado por un agente. Por lo tanto solamente habrá una instancia de cada dispositivo en todo el grupo de agentes. El agente que posee la instancia es el responsable de aplicar las operaciones necesarias sobre el dispositivo.

La declaración de instancias de los dispositivos se hace en el cuerpo de la clase y su inicialización en el contructor. Por ejemplo, en el Listado 4-3 se muestra la declaración e inicialización de los dispositivos en el agente *ManufacturingAgent1*.

```
// declaring devices
private Conveyor conveyor1;
private Camera camera1;
private Detector detector1;
.
// constructor definition
public ManufacturingAgent1()
{

   // device's initialization
   conveyor1 = new Conveyor("CONV1");
   camera1   = new Camera("CAMERA1");
   detector1 = new Detector("DETECTOR1");

}
```

Listado 4-3. Inicialización de dispositivos del agente.

CAPÍTULO

5

Desarrollo de un sistema de coordinación distribuido

En este capítulo se aplica la metodología propuesta usando como ejemplo un sistema de ensamble de piezas. Al final, la metodología permitirá obtener un sistema multiagente para el control distribuido de la tarea de ensamble.

5.1. Descripción del sistema de manufactura y de la tarea

5.1.1. Descripción del sistema

Los componentes del sistema están basados en la taxonomía presentada en el capítulo 2. El sistema de manufactura usado para ejemplificar la aplicación de la metodología produce ensambles manejando las piezas que entran al sistema en orden aleatorio.

El sistema de manufactura está distribuido de acuerdo al esquema mostrado en la Figura 5-1. Los componentes del sistema han sido colocados de tal forma que lleven a cabo el ensamble de una línea de productos. Los componentes del sistema y el uso de cada uno de ellos en la tarea del sistema se detallan en la Tabla 5-1.

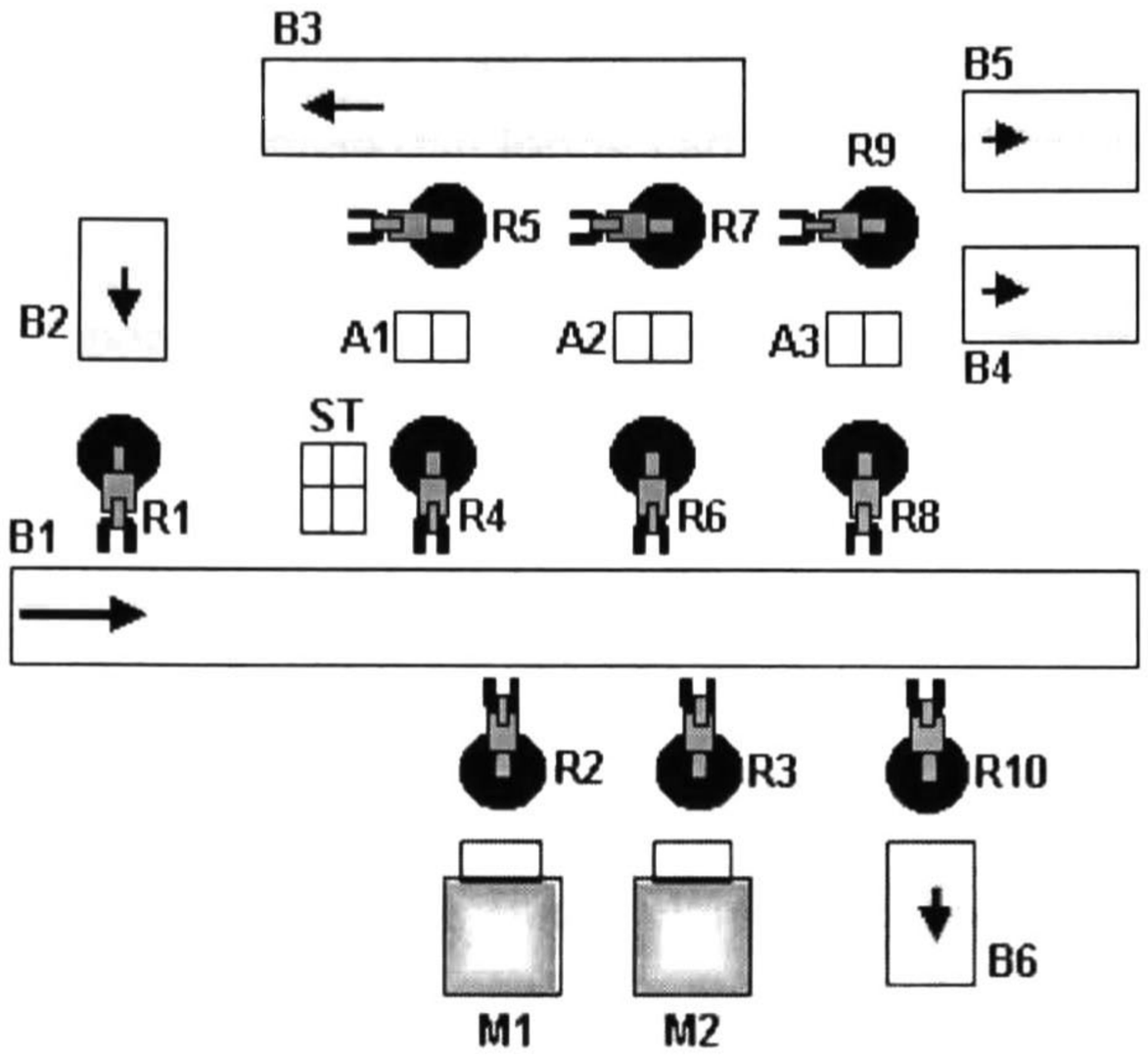

Figura 5-1. Sistema de ensamble.

Tipo de Comp.	Componente	Descripción
Conveyor	B1	Banda que transporta piezas a lo largo del sistema.
	B2	Banda donde entran las piezas al sistema.
	B3	Banda de salida de piezas del sistema.
	B4	Banda de salida de piezas del sistema.
	B5	Banda de salida de piezas del sistema.
	B6	Banda de salida de piezas del sistema.
Robot	R1	Robot que toma las piezas de B2 y las coloca en B1.
	R2	Robot toma piezas de B1 y las coloca en M1. Toma piezas de M1 y las coloca en B1.
	R3	Robot toma piezas de B1 y las coloca en M2. Toma las piezas de M2 y las coloca en B1.
	R4	Robot toma las piezas de B1 para ensamblar en A1. De ser necesario, usa ST para colocar piezas temporalmente.
	R5	Robot toma los ensambles de A1 y las coloca en B3.
	R6	Robot toma las piezas de B1 para ensamblar en A2.
	R7	Robot toma los ensambles de A2 y las coloca en B3.
	R8	Robot toma las piezas de B1 para ensamblar en A3.
	R9	Robot toma los ensambles de A3 y las coloca en B4 o B5.
	R10	Robot toma piezas de B1 y coloca en B6.
Mesa	ST	Mesa para colocar piezas a usar en los ensambles.
Mesa de ensamble	A1	Mesa para ensamblar piezas.
	A2	Mesa para ensamblar piezas.
	A3	Mesa para ensamblar piezas.
Máquina	M1	Máquina realiza un procesamiento sobre la pieza.
	M2	Máquina realiza un procesamiento sobre la pieza.

Tabla 5-1. Componentes del sistema.

Frente a R1 hay un sensor óptico C1 que detecta la llegada de partes sobre la banda B2. Encima de esta zona hay una cámara que forma parte de un sistema de reconocimiento y localización de piezas.

Cada uno de los componentes del sistema de manufactura tiene un controlador local el cual está conectado a una red y puede recibir instrucciones/comandos que son interpretados y ejecutados por el componente.

5.1.2. Tarea de manufactura

Las piezas llegan al sistema por la banda B2, donde el sensor óptico S1 detiene la banda cuando detecta un componente y se inicia el reconocimiento de la pieza con la cámara C1. Una vez identificada, el robot R1 coloca la pieza sobre la banda B1.

La banda B1 transporta las piezas a través del sistema. Cuando una pieza A pasa frente al robot R2, la banda se detiene y el R2 la toma y la coloca en la entrada de la máquina M1. La máquina procesa y cambia los atributos de la pieza a M, la cual pone en la entrada de la máquina. El robot R2 ahora toma la pieza y la coloca sobre la banda B1.

Sitio	Asociado a
Conv2	B1
Conv3	B1
Conv4	B1
Conv5	B1
Conv6	B1
Conv7	B1
Conv8	B1
Conv9	B1
Conv10	B1

Sitio	Asociado a
Conv11	B1
Conv12	B1
Grip2	R2
Machine1	M1
Grip3	R3
Machine2	M2
Grip4	R4
ST1	ST1

Sitio	Asociado a
Assb1	A1
Grip5	R5
Conv14	B3
Conv1	B2
Grip1	R1
Grip6	R6
Assb2	A2
Grip7	R7

Sitio	Asociado a
Conv15	B3
Grip8	R8
Assb3	A3
Grip9	R9
Conv16	B4
Conv17	B5
Grip10	R10
Conv13	B6

Tabla 5-2. Lista de sitios identificados en el modelo.

Cuando la pieza B pasa frente al robot R3, la banda se detiene y el R3 la toma y la coloca en la entrada de la máquina M2. La máquina procesa y cambia los atributos de la pieza a N, la cual pone en la entrada de la máquina. El robot R3 ahora toma la pieza y la coloca sobre la banda B1.

Cuando la pieza N pasa frente al robot R4, la banda se detiene y el R4 la toma y trata de ensamblarla en la mesa A1 de acuerdo al patrón predeterminado NNN. La mesa ST es usada para colocar temporalmente las piezas que no puede ensamblar pero utilizará posteriormente. Cuando el ensamble está terminado será tomado por el robot R5, que lo colocará en la banda de salida B3.

Cuando cualquiera de las piezas M ó C pasa frente al robot R6, la banda se detiene y el R6 la toma y trata de ensamblarla en la mesa A2 de acuerdo al patrón predeterminado MCCM. Cuando el ensamble está terminado será tomado por el robot R7, que lo colocará en la banda de salida B3.

Cuando cualquiera de las piezas M ó C pasa frente al robot R8, la banda se detiene y el R8 la toma y trata de ensamblarla en la mesa A3 de acuerdo al patrón predeterminado MMC. Cuando el ensamble está terminado será tomado por el robot R9, que lo colocará en la banda de salida B4, y en caso de estar ocupada lo colocará en la banda B5.

Cualquier pieza que pase frente al robot R10 será tomada por el mismo, colocándola en la banda de salida B6.

5.2. Modelado de la tarea de manufactura

Analizando la descripción del sistema, se extrae el conjunto de los sitios, actuadores y sensores que son necesarios en el sistema. De la tarea de ensamble se obtiene el conjunto de reglas necesarias para llevarla a cabo, además de los sitios, actuadores y sensores relacionados con cada una de ellas.

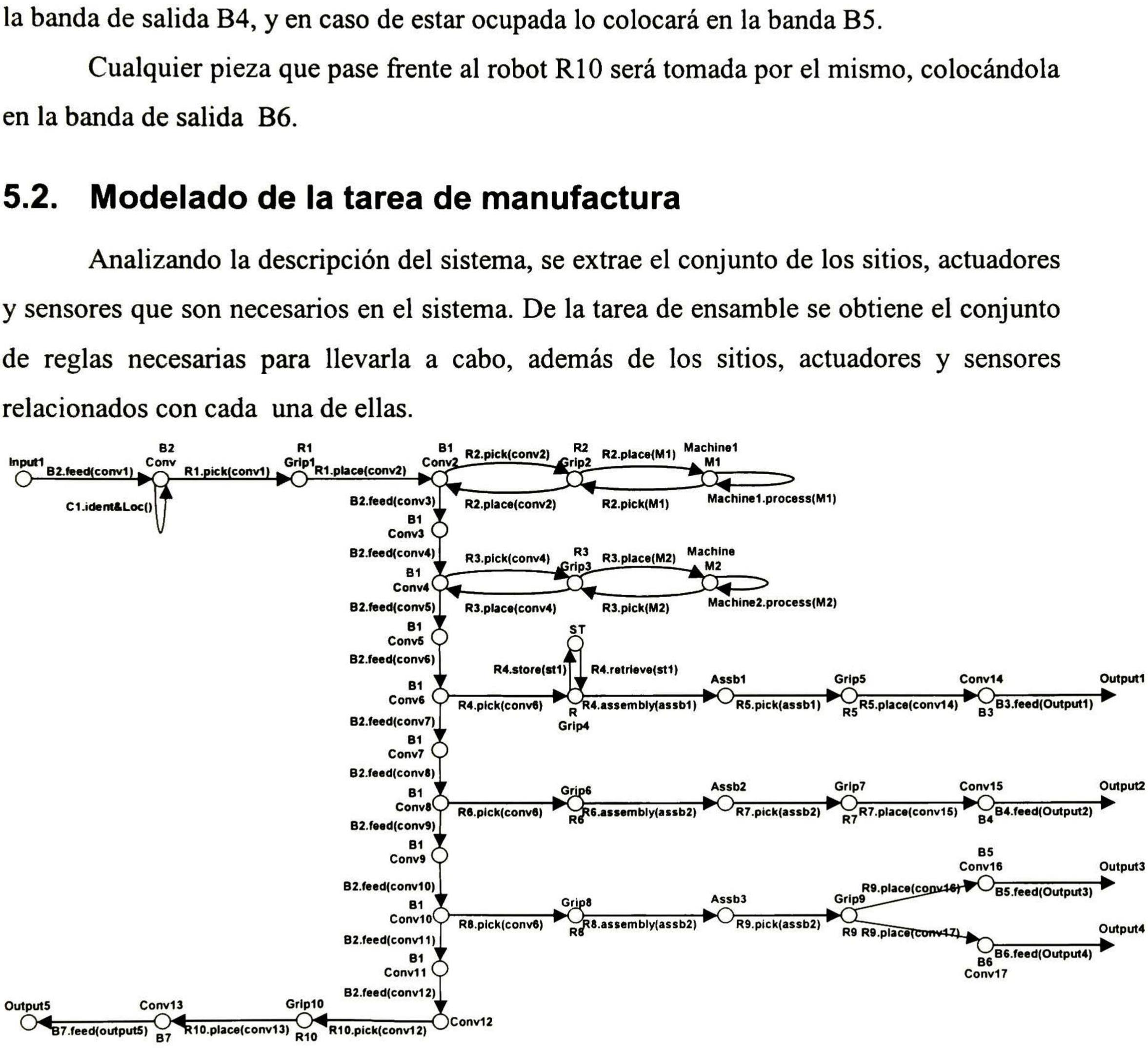

Figura 5-2. Grafo de flujo de piezas de la tarea del sistema de ensamble.

Los sitios identificados en el sistema de manufactura se muestran en la Tabla 5-2. Con estos sitios es posible construir el grafo de flujo de partes que modela la tarea de ensamble mostrando los sitios y las operaciones que transfieren partes entre ellos. En la Figura 5-2 se muestra el grafo de flujo de piezas de la tarea global.

Operación	**R2.pick(conv2)**
Descripción	La pieza es transferida del conveyor B1 al robot R2.
Sitios	Grip2, M1, Conv2.
Dispositivos	B1, R2.
Precondiciones	La pieza que contiene Conv2 es A, Grip2 y Máquina1 no contienen piezas.
Acciones	La pinza de R2 toma la pieza de Conv2.
Poscondiciones	El estado de los sitios interface modificados es comunicado a los demás agentes. El sitio Conv2 no contiene piezas y Grip2 contiene la pieza A.
Operación	**R2.place(M1)**
Descripción	La pieza es transferida del robot R2 a la Máquina M1.
Sitios	Grip2, M1, Conv2.
Dispositivos	B1, R2.
Precondiciones	La pieza que contiene Conv2 es A y Máquina1 no contiene pieza.
Acciones	La pinza de R2 pone la pieza a la entrada de Máquina1.
Poscondiciones	El estado de los sitios interface modificados es comunicado a los demás agentes. El sitio Grip2 no contiene piezas y Machine1 contiene la pieza A.
Operación	**Machine1.process(M1)**
Descripción	La pieza es procesada por la Máquina1 cambiando la identidad de la pieza del sitio M1.
Sitios	Grip2, M1, Conv2.
Dispositivos	B1, R2.
Precondiciones	La pieza que contiene Máquina1 es A.
Acciones	Máquina1 procesa la pieza A y cambia los atributos a M colocando la pieza a la entrada de la máquina.
Poscondiciones	El estado de los sitios interface modificados es comunicado a los demás agentes. El sitio Máquina1 contiene la pieza M.
Operación	**R2.pick(M1)**
Descripción	La pieza es transferida de la Máquina1 a R2.
Sitios	Grip2, M1, Conv2.
Dispositivos	B1, R2.
Precondiciones	La pieza que contiene Máquina1 es M y Grip2 no contiene pieza.
Acciones	La pinza de Robot1 toma la pieza de Machine1.
Poscondiciones	El estado de los sitios interface modificados es comunicado a los demás agentes. El sitio Machine1 no contiene piezas y Grip1 contiene la pieza M.
Operación	**R2.place(conv2)**
Descripción	La pieza es transferida del robot R2 al conveyor B1.
Sitios	Grip2, M1, Conv2.
Dispositivos	B1, R2.
Precondiciones	La pieza que contiene Grip2 es M y Grip2 no contiene pieza.
Acciones	La pinza de R2 coloca la pieza en Conv2.
Poscondiciones	El estado de los sitios interface modificados es comunicado a los demás agentes. El sitio Conv2 contiene la pieza M y Grip2 no contiene piezas.

Tabla 5-3. Reglas identificadas para la *Subtarea3*.

Las operaciones del agente son documentadas, identificando sus precondiciones, el conjunto de acciones a realizar si las precondiciones se cumplen y las postcondiciones que

se cumplen al terminar de procesar la regla. En la Tabla 5-3 se documentan brevemente algunas reglas del modelo.

5.3. Partición del modelo

La partición del modelo global se basa en los criterios establecidos en el capítulo 2; al aplicar estos criterios se obtiene el número máximo de particiones del modelo.

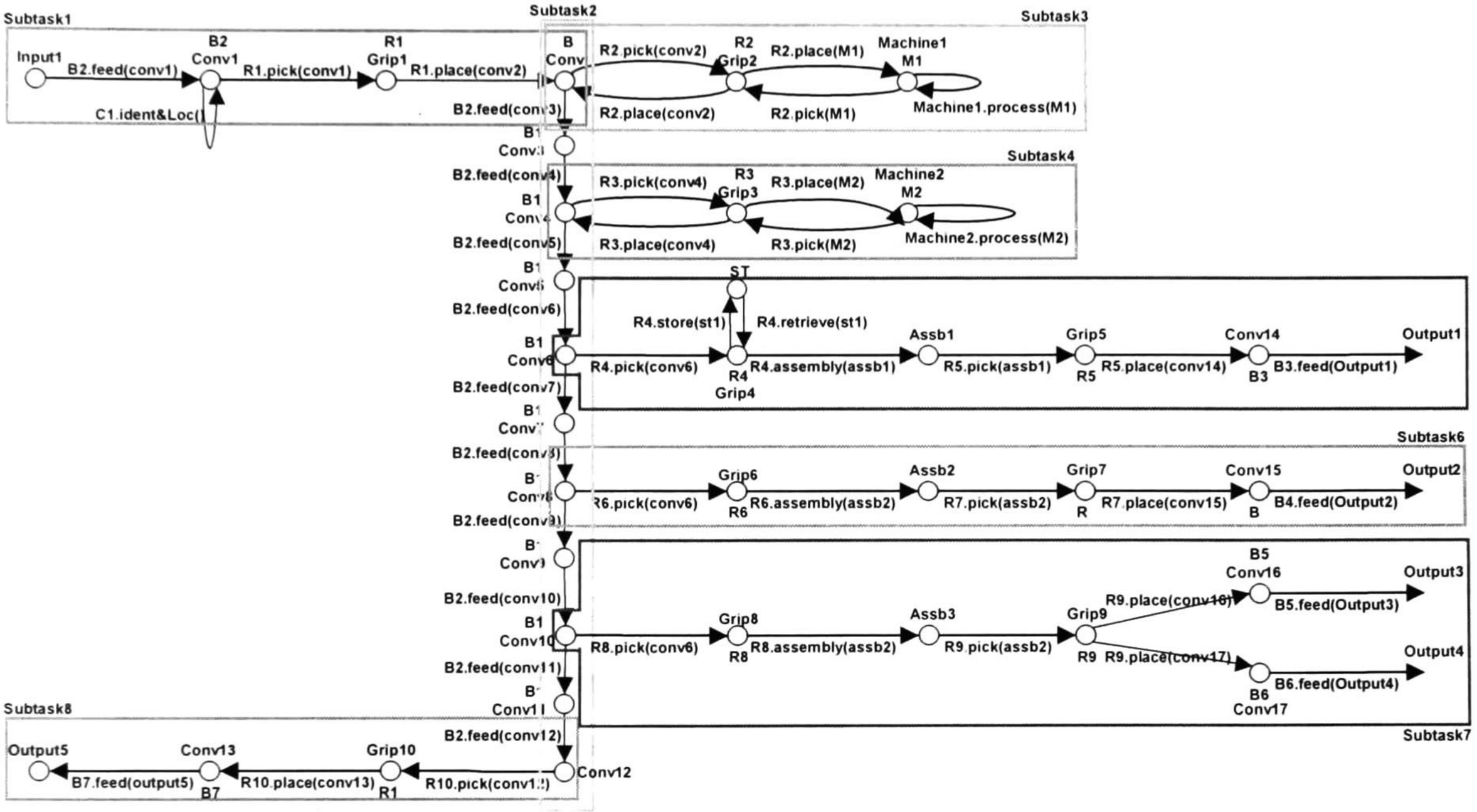

Figura 5-3. Partición del modelo en subtareas.

Para este problema, se decidió agrupar algunas particiones en subtareas más grandes aprovechando la semejanza de algunas particiones. Así, se decidió crear ocho particiones que se muestran en la Figura **5-3** denominadas como *SubTask1*, *SubTask2*, *SubTask3*, *SubTask4*, *SubTask5*, *SubTask6*, *SubTask7*y *SubTask8*.

5.4. Asignación de tareas a los agentes

Los agentes pueden tener asignadas una o más subtareas de la partición. Las subtareas asignadas ya no pueden ser asignadas a otro agente. Así, el conjunto de agentes depende del número de equipos de cómputo que estarán disponibles para el sistema y de la distribución natural de la tarea inducida por la distribución geográfica de los componentes.

La Figura 5-3 muestra las subtareas consideradas para la asignación a los agentes. Se determinó crear ocho agentes, donde cada uno controlará una de las subtareas ya definidas. Los nombres elegidos para los agentes son *MAgA, MAgB, MAgC, MAgD, MAgE, MAgF, MAgG, MAgH* y *MAgX*. En la Tabla 5-4 se muestra la asignación de subtareas.

Agente	Subtarea
MAgA	SubTask1
MAgB	SubTask3
MAgC	SubTask4
MAgD	SubTask5

Agente	Subtarea
MAgE	SubTask6
MAgF	SubTask7
MAgG	SubTask8
MAgX	SubTask2

Tabla 5-4. Asignación de subtareas a los agentes.

5.5. Codificación de los agentes

En el sistema multiagente, cada agente es programado como una clase de Java que hereda los datos y comportamientos de la clase AgentBase, que fue desarrollada en este trabajo como base para construir agentes basados en sitios. La clase AgentBase a su vez hereda el comportamiento de la clase Agent, clase base de los agentes de la plataforma JADE.

Los nombres dados a los agentes del sistema se declaran en constantes de la clase Agents. Lo anterior se muestra en el Listado 5-1. En la misma clase se declara el tamaño del reloj distribuido que manejará el sistema distribuido. A cada agente se le asigna el componente del reloj que le corresponde de manera exclusiva.

Como ejemplo se mostrará parte del código que lleva a cabo la subtarea 3 asignada al agente MAgB. Esta subtarea utiliza tres sitios, un robot, una banda y una máquina de procesamiento de partes.

La clase ManufacturingAgentB, que realizará el comportamiento del agente MAgB, hereda el comportamiento de la clase AgentBase desarrollada en este trabajo para sistemas de coordinación distribuida. En la clase se definen los sitios y actuadores necesarios para la tarea, tal como se muestra en el Listado 5-2.

El constructor de la clase del agente inicializa los sitios y actuadores del agente para que lleve a cabo la tarea. En el Listado 5-3 se muestra la construcción de las instancias para el reloj, los sitios y los actuadores del agente. Cuando se crean los sitios, éstos son añadidos a una tabla de sitios que mantiene el agente.

```
public final class Agents
{
   // declaring agent names of multiagent system.
   public static final String EX2MAA = "ex2maga";
   public static final String EX2MAB = "ex2magb";
   public static final String EX2MAC = "ex2magc";
   public static final String EX2MAD = "ex2magd";
   public static final String EX2MAE = "ex2mage";
   public static final String EX2MAF = "ex2magf";
   public static final String EX2MAG = "ex2magg";
   public static final String EX2MAH = "ex2magh";
   public static final String EX2MAI = "ex2magi";
   public static final String EX2MAX = "ex2magx";

   // asigned component of the global Vector Clock
   public static final int VC_EX2MAA = 0;
   public static final int VC_EX2MAB = 1;
   public static final int VC_EX2MAC = 2;
   public static final int VC_EX2MAD = 3;
   public static final int VC_EX2MAE = 4;
   public static final int VC_EX2MAF = 5;
   public static final int VC_EX2MAG = 6;
   public static final int VC_EX2MAH = 7;
   public static final int VC_EX2MAI = 8;
   public static final int VC_EX2MAX = 9;
   public static final int VC_EX2TOTAL = 10;
}
```

Listado 5-1. Clase Agents que contiene los nombres de los agentes del sistema.

```
import JGAgent.*;
import devices.effectors.*;
import devices.sensors.*;

public class ManufacturingAgentB extends AgentBase
{
   private Site conv2, grip2, m1;

   private Robot robot2;
   private Conveyor conveyor2;
   private Machine machine1;
   ..
}
```

Listado 5-2. Declaración de la clase del agente.

El sitio *conv2* además es configurado como sitio interface y se le asignan los nombres de los dos agentes que tienen representación de este mismo sitio.

Las cinco reglas que implementa el agente involucran los sitios y su contenido. Las reglas se programan en el método *rules( )* usando estructuras de decisión ***if-then***.

```
public ManufacturingAgentB()
{
   super();

   localVectorClock=new VectorClock(Agents.VC_EX2TOTALES,Agents.VC_EX2MAB);

   addSite( conv2 = new Site("Conveyor2") );
   addSite( grip2 = new Site("Grip2") );
   addSite( m1 = new Site("Machine1") );

   conv2.addAgent(Agents.EX2MAA);
   conv2.addAgent(Agents.EX2MAX);
   conv2.setInterface();

   robot2 = new devices.effectors.Robot("R2");
   conveyor2 = new devices.effectors.Conveyor("Conveyor2");
   machine1 = new devices.effectors.Machine("Machine1");
}
```

Listado 5-3. Constructor que inicializa los datos del agente.

Cuando se cumplen las condiciones de la regla se solicita el uso exclusivo de los sitios interface. Una vez que los agentes autorizan el uso exclusivo de esos sitios se ejecuta la regla. Antes de terminar la regla se deben enviar los mensajes de actualización a los agentes que mantienen representaciones del sitio. Finalmente se libera al sitio interface de la condición de exclusión mutua.

Las operaciones documentadas en la Tabla 5-3 son las operaciones que, luego de particionar el modelo, se asignaron al agente MAgB. La programación de estas operaciones se realiza en forma de reglas *if-then* en el método *rules()* de la clase *ManufacturingAgentB*. El Listado 5-4 muestra el código correspondiente.

```
public void rules() {
   // R2 pickup part from Conveyor2: Conv2 -> Grip2
   if(!conv2.isEmpty() && grip2.isEmpty() && conv2.isEqual("A")) {
      if( getMutex(grip2) && getMutex(conv2) ) {
         robot2.pick(conv2.getContent());
         grip2.setContent(conv2.getContent());
         conv2.removeContent();

         sendUpdateMessages(conv2);
         sendUpdateMessages(grip2);
         freeMutex(grip2);
         freeMutex(conv2);
      }
   }
   // R2 put part on Machine1: Grip2 -> m1
   if(!grip2.isEmpty() && m1.isEmpty()) {
      if( getMutex(grip2) && getMutex(m1) ) {
         robot2.place(grip2.getContent());
         m1.setContent(grip2.getContent());
         grip2.removeContent();

         sendUpdateMessages(m1);
         sendUpdateMessages(grip2);
```

```
            freeMutex(grip2);
            freeMutex(m1);
        }
    }
    // R2 pickup piece from Machine1: m1 -> Grip2
    if(!m1.isEmpty() && m1.isEqual("M") && grip2.isEmpty()) {
        if( getMutex(grip2) && getMutex(m1) ) {
            robot2.pick(m1.getContent());
            grip2.setContent(m1.getContent());
            m1.removeContent();

            sendUpdateMessages(m1);
            sendUpdateMessages(grip2);
            freeMutex(grip2);
            freeMutex(m1);
        }
    }
    // R2 put part on Conveyor2: Grip2 -> Conv2
    if(!grip2.isEmpty() && grip2.isEqual("M") && conv2.isEmpty()) {
        if( getMutex(grip2) && getMutex(conv2) ) {
            robot2.place(grip2.getContent());
            conv2.setContent(grip2.getContent());
            grip2.removeContent();

            sendUpdateMessages(grip2);
            sendUpdateMessages(conv2);
            freeMutex(grip2);
            freeMutex(conv2);
        }
    }
    // Process the part A and converts in part M
    if(!m1.isEmpty() && m1.isEqual("A")) {
        if( getMutex(m1) ) {
            machine1.process(m1.getContent());
            m1.setContent("M"); // processed part

            sendUpdateMessages(m1);
            freeMutex(m1);
        }
    }
} // end of rules method
```

Listado 5-4. Programación de las reglas del agente.

5.5.1. Exclusión mutua de sitios interface

La exclusión mutua de los sitios interface por los agentes es necesaria para garantizar la consistencia global de la información de dichos sitios. Los sitios interface tienen una representación local en cada agente que ejecuta la tarea. Así, cada representación del sitio interface dentro de cada agente debe contener la misma información. Cada cambio en el estado del mismo sitio debe ser comunicado a los demás agentes que comparten de manera lógica cada sitio. Para esto se necesita un reloj global para distinguir los mensajes en el tiempo. Como el sistema es distribuido se puede usar un reloj lógico global basado en los eventos que ocurren dentro de los agentes, tal como se propone en [Babaoglu95].

Para garantizar la exclusión mutua sobre un sitio interface los agentes deben cooperar. Cuando un agente necesita cambiar el estado de un sitio debe solicitar el uso exclusivo del sitio a los agentes que tienen representación del mismo. Estos agentes deben

autorizar el uso del sitio. Una vez obtenida la autorización de los demás agentes, el agente solicitante puede usarlo exclusivamente. Si dos agentes solicitan simultáneamente el uso exclusivo del mismo sitio se debe distinguir cuál de los agentes solicitó primero. Enseguida se describe el mecanismo usado para distinguir el orden en que han sido enviados los mensajes en el sistema distribuido.

Vector Clocks

Los *vector clocks* son descritos en [Babaoglu95] y se basan en el mecanismo para historias causales descrito en el mismo artículo. En este esquema cada proceso p_i posee un arreglo local VC de números naturales donde $VC(e_i)$ denota el valor del *vector clock* de p_i cuando ejecuta el evento e_i.

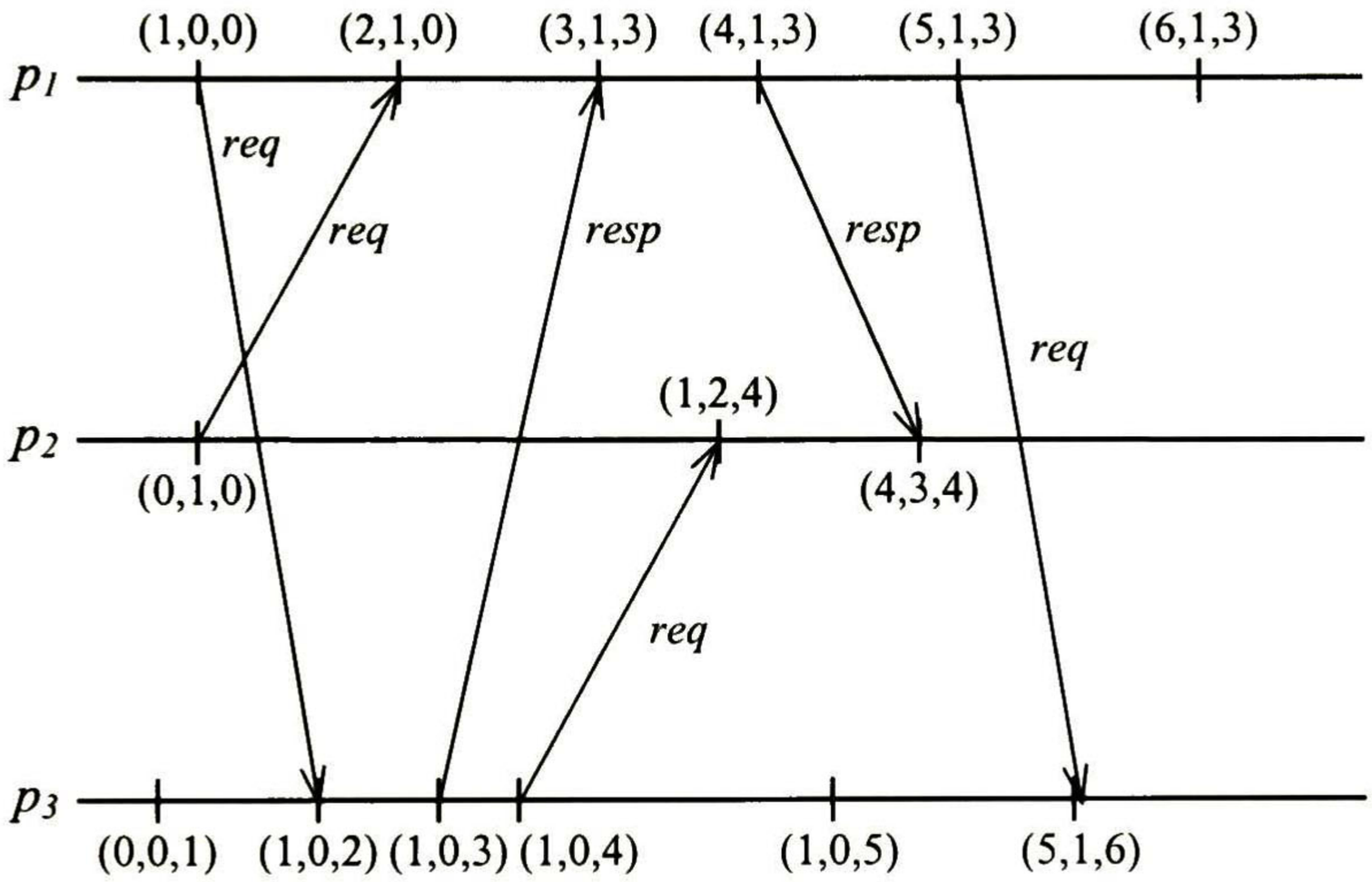

Figura 5-4. Vector Clocks.

Se usa VC para referirse al *vector clock* del proceso de que se trate. Cada proceso p_i inicializa VC para que contenga ceros. Cada mensaje m contiene un *timestamp* $TS(m)$, el cual es el valor del *vector clock* del propio evento que lo envió.

Las siguientes reglas definen cómo p_i modifica el *vector clock* con la ocurrencia de cada nuevo evento e_i:

$VC(e_i)[i] := VC[i] + 1$ si e_i es un evento interno o un envío de mensaje.

$VC(e_i) :=$ máximo$\{VC, TS(m)\}$ si $e_i = receive(m)$
$\text{VC}(e_i)[i] := VC[i] + 1$

Es decir, un evento interno o un evento de envío de mensaje simplemente incrementa la componente local del *vector clock*. En cambio, cuando ocurre un evento de

recepción de mensaje, primero actualiza cada componente del *vector clock* local para que tome el valor que sea mayor entre la componente del VC anterior al presente evento y la componente correspondiente del *timestamp* obtenido del mensaje entrante; luego de actualizar el *VC*, incrementa la componente local. La Figura 5-4 muestra un ejemplo de *vector clocks* asociados con algunos eventos.

Del ejemplo mostrado, el j-ésimo componente del *vector lógico* del proceso p_i tiene la siguiente interpretación para toda $j \neq i$:

$VC(e_i)[j] \equiv$ número de eventos de p_j que precede causalmente al evento e_i de p_i.

Se tiene también que $VC(e_i)[i]$ es el número de eventos p_i que han sido ejecutados hasta el momento, incluyendo e_i. De la misma manera, $VC(e_i)[i]$ es la posición ordenada del evento e_i en la enumeración de los eventos de p_i.

De la definición de *vector clocks*, se pueden derivar algunas propiedades útiles. Dados dos arreglos n-dimensionales **V** y **V'** de números naturales, se define la relación *menor que* (escrita como <) entre ellos de la siguiente manera:

$$\mathbf{V} < \mathbf{V'} \equiv (\mathbf{V} \neq \mathbf{V'}) \wedge (\forall k: 1 \leq k \leq n : \mathbf{V}[\mathrm{k}] \leq \mathbf{V'}[\mathrm{k}])$$

En [Babaoglu95] se describe formalmente el mecanismo para historias causales en sistemas distribuidos y se establecen las bases para usar los *vector clocks* como reloj lógico global.

Implementación de la exclusión mutua con vector clocks

Para saber cuál de los agentes que han solicitado el uso exclusivo del mismo sitio ha enviado Si dos agentes solicitan simultáneamente el uso exclusivo del mismo sitio se debe distinguir cuál de los agentes solicitó primero.

Para distinguir cuál de los agentes que solicitaron el uso exclusivo del sitio lo solicitó primero se emplea un reloj lógico distribuido basado en *vector clocks*. Dicho *vector clock* posee tantas componentes como agentes hay en el sistema. Cada agente tiene asignada una componente que puede cambiar. Cada mensaje intercambiado debe contener el momento en que partió según del reloj lógico del agente. Cuando el agente recibe un mensaje, incrementa su propia componente, actualiza la representación del reloj y procesa el mensaje. Al enviar un mensaje previamente se incrementa la componente del agente y se añade esta información.

Para sincronizar las peticiones de solicitud de sitios se ha implementado una clase *VectorClock*. Cada mensaje añade en el contenido información sobre el momento en que salió del agente. Esta clase permite al agente actualizar su propia componente de su reloj lógico. También debe actualizarse tomando en cuenta el momento en que salió cada mensaje del agente emisor.

5.6. Ejecución del sistema multiagente

Una vez programadas y compiladas las clases de los agentes se copian los archivos con extensión *.class* en las máquinas donde será ejecutado cada agente.

Una de las computadoras debe ejecutar la plataforma Jade. En la Figura 5-5 se muestra el ejemplo actual siendo ejecutado en la computadora llamado *gmorales*.

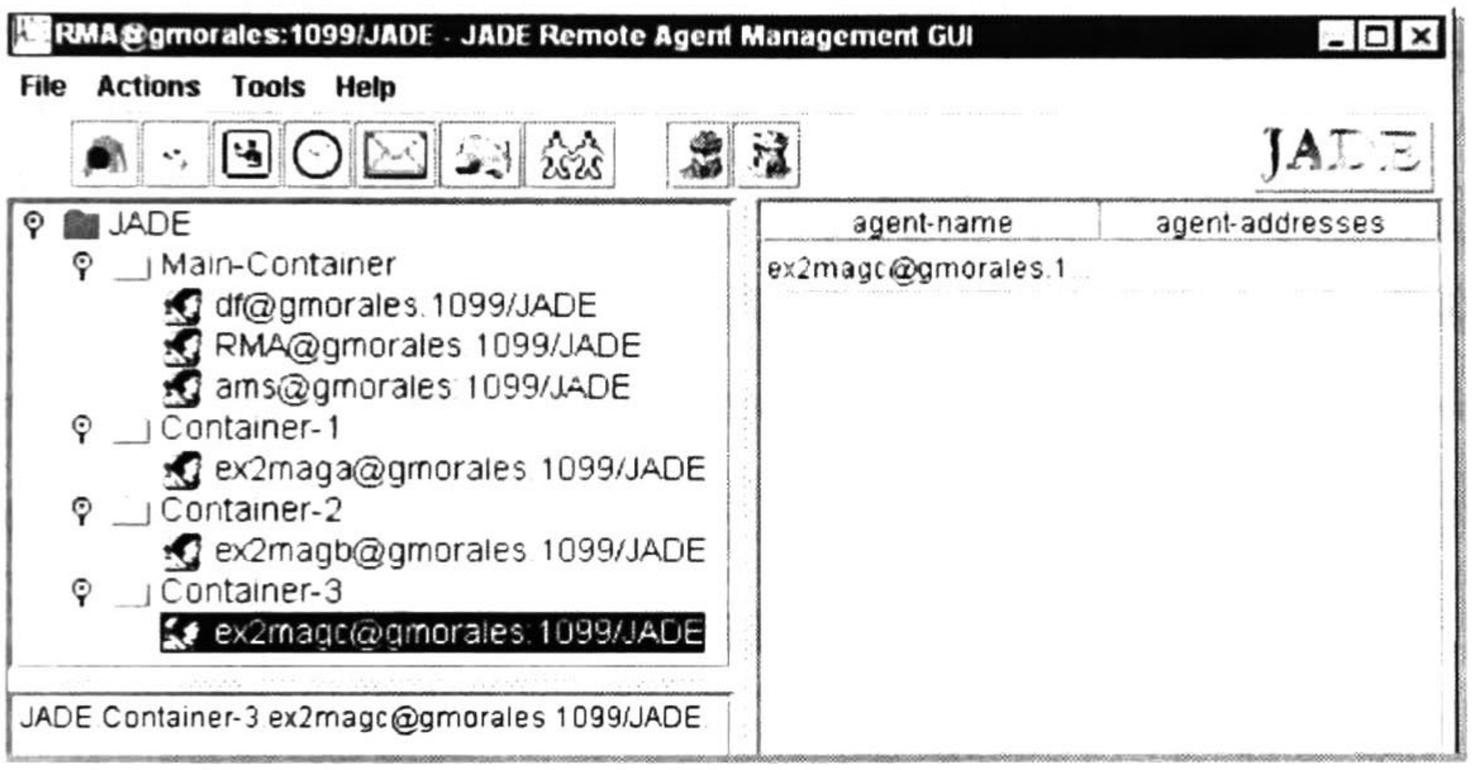

Figura 5-5. Ejecución de la plataforma Jade.

Una vez ejecutada la plataforma, se debe ejecutar cada agente desde una computadora que pueda interactuar con la máquina que está ejecutando la plataforma Jade.

Al comenzar la ejecución del agente, él mismo se registra en la plataforma y comienza a interactuar con el resto de los agentes del sistema.

En la Figura 5-6 se muestra la ejecución del agente cuyo identificador único es *MAgA@gmorales:1099/JADE*. Los agentes despliegan una ventana con el estado de los sitios y una ventana por cada dispositivo que maneja el agente. En la figura se muestra una ventana con el estado de los sitios del agente MAgA y otras ventanas que monitorean al conveyor *Conv1*, al robot *R1*, la cámara *Camera1* y al detector *Detector1*.

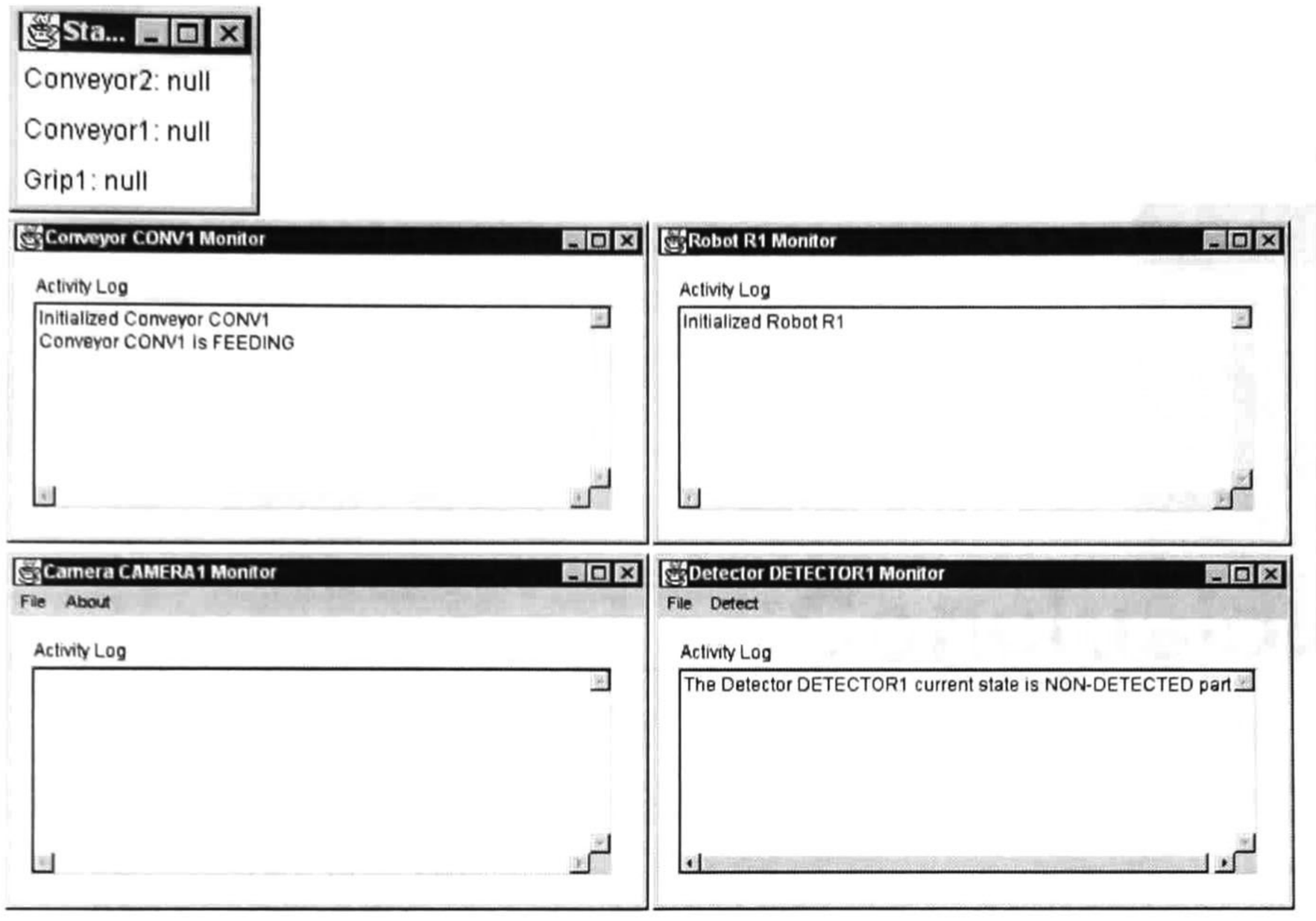

Figura 5-6. Ejecución del agente MAgA.

De esta manera se pueden observar los estados del agente y sus dispositivos. Cada ventana de dispositivo registra los eventos que ocurren y los cambios en su estado. En la ventana de estado de los sitios se muestra su contenido durante la ejecución de la tarea.

Conclusiones

En este trabajo se ha abordado el problema de la programación de sistemas de coordinación distribuida para Sistemas de Manufactura Flexible (FMS) utilizando el paradigma de los sistemas multiagente. Se presentó un método de desarrollo de software que parte de una especificación verbal del sistema y de la tarea a ejecutar y obtiene un conjunto de agentes que controlan, cada uno, parte del sistema logrando así la coordinación total del FMS. Los agentes pueden ejecutarse en equipos con distintas plataformas e interactuar a través del paso de mensajes.

Una parte importante de este método es el modelado. La especificación se traduce en elementos gráficos que permiten eliminar en gran medida enunciados ambiguos o poco claros de la especificación verbal. El modelo de la tarea conduce a la obtención del conocimiento necesario de la entidad responsable de la coordinación de la tarea, entidad que es distribuida entre un conjunto de agentes al realizar la partición del grafo de flujo de partes. La representación de conocimiento usando reglas permite modificar fácilmente las estrategias de ensamble o manufactura afectando el mínimo número de agentes en una fase de reprogramación.

La metodología de programación de los agentes coordinadores incluye una arquitectura basada en el estándar industrial FIPA y recurre a las facilidades de la plataforma JADE. Las pruebas efectuadas al software desarrollado para los casos de estudio

tratados consisten en la simulación de las tareas a través de las consolas de los equipos utilizados. Sin embargo no está lejana de una aplicación real: bastaría conocer las interfaces para comunicación de los dispositivos del FMS a controlar.

El presente trabajo constituye un área de oportunidad para proponer mejoras en el marco de una aplicación real: el método puede ser extendido para coordinar equipos de manufactura reales y manejar los eventos imprevistos de los equipos tales como fallas; la programación de la infraestructura de los agentes puede optimizarse para disminuir el tiempo en la transferencia de mensajes; finalmente, los agentes pueden usar interfaces con elementos gráficos para el seguimiento del estado de la tarea coordinada por el agente.

Partiendo de este trabajo puede desarrollarse una herramienta de software para la creación de sistemas de coordinación en los cuales se diseñe el grafo de flujo de partes, crear las particiones automáticamente y generar el código correspondiente para los agentes.

Referencias

[Babaoglu95] O. Babaoglu, K. Marzullo. Consistend Global States of Distributed Systems: Fundamental Concepts and Mechanisms. *Distributed Systems* second edition. Edited by Sape Mullender. Chapter 4. 1995.

[Bellifemine99] F. Bellifemine, A. Poggi, G. Rimassa. "JADE - A FIPA-Compliant Agent Framework". CSELT internal technical report. Part of this report has been also published in Proceedings of International Conference on Practical Applications of Intelligent Agents and Multi-Agent Technology (PAAM 99), London, April 1999, pags. 97-108. Available from http://sharon.cselt.it/projects/jade/papers.htm.

[Bellifemine00] F. Bellifemine, A. Poggi, G. Rimassa. P. Turci. An Object-Oriented Framework to Realize Agent Systems. Proceedings of WOA 2000 Workshop, Parma, May 2000, pags. 52-57. Available from http://sharon.cselt.it/projects/jade/papers.htm.

[Brazier95] F. Brazier, B. Dunin-Keplicz, N.R.Jennings, J. Treur. "Formal Specification of Multi-Agent Systems: a Real-World Case" Proceedings of International Conference on Multi-Agent Systems (ICMAS 95), AAAI Press/MIT Press, Menlo Park, USA, 1995. Pp. 25-32.

[Bussmann97] S. Bussmann, H. Baumgärtel, M. Klosterberg. "Multi-Agent Coordination of Material Flow in a Car Plant" Second International Conference on Practical Applications of Intelligent Agents and Multi-Agent Technology (PAAM 97). London, UK, 1997. Pp. 227-236.

[Bussmann98a] S. Bussmann, "Agent-Oriented Programming of Manufacturing Control Tasks", Proceedings of 3rd. International Conference on Multi-Agent Systems (ICMAS '98), París, France, 1998, 57-63.

[Bussmann98b] S. Bussmann, "An Agent-Oriented Architecture for Holonic Manufacturing Control", Proceedings of 1st International Workshop on Intelligent Manufacturing Systems, EPFL, Lausanne, Switzerland, 1998, pp. 1 12.

[Case01] S. Case, N. Azarmi, M. Thint, T. Ohtani. Enhancing e-Communities with Agent-Based Systems. *Computer*, vol. 34, no. 7, July 2001, pp. 64-69.

[Chochon87] Chochon, H. Alami, R. "A knowledge-based system for programming and execution control of multi-robot assembly cells" Int. Conf. on Advanced Robotics. Versailles, France, October 1987.

[DeLoach98] DeLoach, S.A. "Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems", Agent-Oriented Information Systems '99 (AOIS'99), Seattle WA, 1 May 1998.

[Durfee01] E. H. Durfee. Scaling Up Agent Coordination Strategies. *Computer*, vol. 34, no. 7, July 2001, pp. 39-46.

[FIPA97] Foundation for Intelligent Physical Agents, Specifications. 1997. Available from http://www.fipa.org.

[Franklin96] S. Franklin, A. Graesser. "Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents". In J. P. Müller, M. Wooldridge and N. R. Jennings editors. *Intelligent Agents III*. Lecture Notes in Artificial Intelligence. Springer-Verlag, Budapest, Hungary. p.23, 1996.

[Gershwin89] S. B. Gershwin. "Hierarchical Flow Control: A Framework for Scheduling and Planning Discrete Events in Manufacturing Systems" IEEE Proceedings, Special Issue on Discrete event Systems. Vol. 77, No. 1, January, pp. 195-209, 1989.

[Jennings98] N. R. Jennings, K. Sycara and M. Wooldridge (1998) "A Roadmap of Agent Research and Development" Int Journal of Autonomous Agents and Multi-Agent Systems 1 (1) 7-38.

[López-Mellado97]. E. López-Mellado, E. Medrano-Pérez. "Object-Based Design of FMS Controllers". Proceedings of the Fifth IASTED International Conference Robotics and Manufacturing, May 29-31, 1997, Cancún, México, pp. 342-345.

[Ouelhadj98] D. Ouelhadj, C. Hanachi, B. Bouzouia. "Multi-Agent System for Dynamic Scheduling and Control in Manufacturing Cells" Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'98), pp. 2128-2133, IEEE Computer Society, 16-20 May, 1998.

[Rodriguez00] Rodríguez B., J.S. "Optimización en sistemas de eventos discretos temporizados". Tesis de Maestría en Ciencias de la Ingeniería Eléctrica. Cinvestav Guadalajara. Agosto 2000.

[Shen99] W. Shen, D. H. Norrie. Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey. *Knowledge and Information Systems, an International Journal*. 1(2), 1999, pp. 129-156.

Centro de Investigación y de Estudios Avanzados del IPN

Unidad Guadalajara

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, aprobó la tesis: Coordinación Distribuida Basada En Agentes De Sistemas De Manufactura Flexible del(a) C. José Guadalupe MORALES MONTELONGO el día 22 de Noviembre de 2002.

DR. LUIS ERNESTO LÓPEZ MELLADO
INVESTIGADOR CINVESTAV 3A
CINVESTAV GDL
GUADALAJARA

DR. ANTONIO RAMIREZ TREVIÑO
INVESTIGADOR CINVESTAV 2A
CINVESTAV GDL
GUADALAJARA

DR. FÉLIX FRANCISCO RAMOS CORCHADO
INVESTIGADOR CINVESTAV 2A
CINVESTAV GDL
GUADALAJARA

DR. VICTOR MANUEL LARIOS ROSILLO
INVESTIGADOR TITULAR
UNIVERSIDAD DE GUADALAJARA
ZAPOPAN