XX (131345.1)

CINVESTAV

Centro de Investigación y de Estudios Avanzados del I.P.N.
Unidad Guadalajara

Coordinación de flujo de trabajo basado en sistemas de agentes móviles

Tesis que presenta:

Marina Flores Badillo

para obtener el grado de:
Maestro en Ciencias

en la especialidad de:
Ingeniería Eléctrica

Director de Tesis
Dr. Luis Ernesto López Mellado

Guadalajara, Jalisco, Agosto de 2006.

Coordinación de flujo de trabajo basado en sistemas de agentes móviles

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:

Marina Flores Badillo

Ingeniero en Sistemas Computacionales

Instituto Tecnológico de Morelia 1999-2004

Becario de conacyt, expediente no. 191083

Director de Tesis

Dr. Luis Ernesto López Mellado

CINVESTAV del IPN Unidad Guadalajara, Agosto de 2006.

COORDINACIÓN DE FLUJO DE TRABAJO BASADO EN SISTEMAS DE AGENTES MÓVILES

Resumen

Esta tesis trata sobre el ciclo de vida del desarrollo de sistemas para la coordinación de flujo de trabajo utilizando un enfoque basado en agentes móviles. Los problemas abordados son la especificación de los procesos de flujo de trabajo a través de un modelo formal multi-nivel y la implementación de sistemas multi agentes para la gestión del flujo de trabajo.

Para el modelado del flujo de trabajo se propone una metodología de construcción de modelos con nLNS, un formalismo basado en redes de Petri. Esta metodología conduce a descripciones modulares y jerárquicas, donde las componentes son redes de estructura simple, las cuales especifican el ambiente de trabajo, los agentes móviles, los recursos y los procesos, entre otras entidades. Los diversos modelos se interrelacionan a través de la sincronización, declarada no explícita, de transiciones.

Sobre el desarrollo de sistemas multi agentes se presenta una metodología que permite construir los componentes de software a partir de modelos expresados en nLNS; agentes estacionarios y móviles son implementados en Java utilizando las facilidades de JADE, un middleware que soporta el desarrollo de sistemas multi agentes, el cual cumple las especificaciones de FIPA. La metodología es demostrada mediante el desarrollo de un prototipo que se ejecuta en varias computadoras personales conectadas en red local.

Agradecimientos

A Dios, por haberme dado el ser, el entendimiento, la salud y por caminar a mi lado en los momentos más difíciles de mi vida.

A mi madre, por su ejemplo, confianza, lucha y constante entrega para que yo pueda lograr mis objetivos, por enseñarme a no darme por vencida cuando las cosas se ponen difíciles y sobre todo por su incondicional apoyo para lograr mis metas.

A mi familia, por confiar en mí y por estar siempre presentes en mi lucha por la superación.

A las personas allegadas a mí, por brindarme su confianza y ánimo para seguir siempre adelante.

A mi asesor el Dr. Luis Ernesto López Mellado por su gran comprensión y apoyo en los momentos difíciles y tristes, además por orientarme y aportarme sus conocimientos a través del desarrollo de esta tesis.

Y finalmente agradezco mucho a CONACYT por el apoyo brindado ya que sin su valiosa ayuda esto no habría sido posible.

Índice

Índice de Figuras

Introducción

La Coordinación de Flujo de Trabajo representa un aspecto crítico para lograr competitividad entre las empresas con las tecnologías actuales. Muchas compañías se han dado cuenta de que los procesos del negocio, dentro de la organización así como entre organizaciones, no han sido claramente descritos y por lo tanto no hay suficientes técnicas y métodos para controlar tales procesos.

Hoy en día, existe una organización no lucrativa llamada Workflow Management Coalition (WfMC), la cual define [WfC11] al término *flujo de trabajo* (en inglés *workflow*) como: la automatización de los procesos del negocio durante los cuales, documentos, información o tareas son transferidas de un participante a otro para realizar acciones.

La Gestión de Flujo de Trabajo es una tecnología que promete proveer una eficiente manera de modelar y controlar procesos del negocio complejos y cuyos beneficios incluyen la definición explícita de los procesos y un fácil seguimiento de las operaciones.

En el diseño de Sistemas de Gestión de Flujos de Trabajo (en inglés Workflow Management Systems), el marco de modelado es un aspecto muy importante y ha sido abordado por varios autores. En [DMH01], por ejemplo, se utilizan diagramas de actividad de UML para construir especificaciones de workflow, mientras que en [OrA05] se modelan utilizando formalismos de lógica temporal y transaccional.

También se han utilizado las Redes de Petri (RP) para modelar problemas de flujo de trabajo; tal es el caso de W. Van der Aalst quien ha publicado varios trabajos que abarcan la especificación y análisis de modelos de workflow. En [Aal98] se utilizan RP ordinarias como un lenguaje de diseño para la especificación de workflows complejos, además se argumenta que las razones para utilizar a las RP como marco de modelado para problemas de flujo de trabajo son que proveen una semántica formal, no presentan ambigüedades, provee varias técnicas de análisis, entre otras.

Adicionalmente, existen varias herramientas de desarrollo que están basadas en RP, tal es el caso de YAWL [AaH02] y COSA, aunque existen muchas otras que proponen sus propias estructuras de modelado [Bar03].

Aunque el uso de las RP proporciona modelos claros y sin ambigüedades, cuando los sistemas son grandes y complejos, el construir y manejar modelos con RP ordinarias se vuelve una tarea difícil. Por esta razón en esta tesis nos hemos propuesto como objetivo proponer una metodología para el modelado de problemas de flujo de trabajo utilizando el formalismo nLNS [San05] basado en RP y con el cual obtendremos modelos modulares y jerárquicos con redes más sencillas y fáciles de comprender.

Se presentará una técnica para construir un modelo multinivel de manera sistemática mediante un ejemplo el cual se modelará a 3 niveles de abstracción, donde el primer nivel nos representará a los departamentos involucrados en el problema de flujo de trabajo, el segundo nivel nos servirá para modelar el comportamiento general de un agente que se moverá entre los departamentos el cual irá guiando el proceso y, finalmente, en el nivel 3 detallaremos el comportamiento de dicho agente. Después se pasarán estos modelos a una implementación computacional distribuida basada en agentes en la cual se utiliza Java y el middleware JADE.

El presente documento está organizado como sigue: en el capítulo I introducimos algunos conceptos básicos del flujo de trabajo, se presentarán algunos enfoques existentes para el modelado de este tipo de problemas y finalmente se presentan algunas generalidades del enfoque a adoptar. El capítulo II contiene algunos conceptos básicos de Sistemas Multi agentes además de la definición del formalismo nLNS. En el capítulo III se presenta la metodología propuesta para el modelado de problemas de flujo de trabajo utilizando el formalismo nLNS y en el capítulo IV explicamos de manera general cómo pasar de los modelos creados a una implementación basada en agentes utilizando JADE. En el capítulo V se aplica la metodología presentada en los capítulos III y IV a un caso de estudio. Por ultimo se presentan las conclusiones del trabajo y sugerencias para un trabajo futuro.

CAPITULO I.

Especificación de los Sistemas de Flujo de Trabajo

Resumen: En este capítulo se presentan algunos conceptos básicos sobre flujo de trabajo. Hablaremos un poco sobre la organización que se encarga de las especificaciones apropiadas para su implementación en productos de flujo de trabajo y mencionaremos algunos de los enfoques de modelado de flujo de trabajo existentes que motivaron la metodología propuesta en esta tesis.

1.1 Conceptos Básicos

1.1.1 Definición de Flujo de Trabajo

Los sistemas de información fueron diseñados para soportar la ejecución de tareas individuales. Hoy en día también son utilizados para controlar, monitorear y soportar los aspectos de logística de los procesos del negocio, en otras palabras, necesitan manejar el Flujo de Trabajo a través de la organización. Muchas organizaciones, con complejos procesos de negocio, han identificado la necesidad de conceptos, técnicas y herramientas para soportar la Administración de los Flujos de Trabajo (en inglés **Workflow Management**).

La Administración de Flujo de Trabajo (**WFM** por sus siglas en inglés – Work Flow Management), es una tecnología dinámica la cual esta siendo explotada por los negocios de una gran variedad de industrias. Su característica principal es la automatización de los procesos que implican combinaciones de actividades humanas y basadas en máquinas. Su uso más extendido es dentro del ambiente de las oficinas en las operaciones intensivas del personal como seguros, bancos, hospitales, administración legal y general, etc., se usa también en algunas clases de aplicaciones de la industria y la manufactura [WfC03].

La WorkFlow Management Coalition (WFMC) publicó en 1996 un glosario de términos relacionados al Flujo de Trabajo [WfC11]; define al flujo de trabajo como:

"La automatización de los procesos del negocio, de manera total o parcial, durante los cuales documentos, información o tareas son pasados de un participante a otro para realizar alguna acción de acuerdo a un conjunto de reglas establecidas."

Así, una tarea o proceso define un trabajo a realizar y una entidad procesadora es la encargada de realizar el trabajo, dicha entidad puede ser una persona, una máquina, etc.

Esto permite utilizar la tecnología de workflow para modelar cualquier tipo de proceso, no solamente en el ámbito de los procesos de negocios sino en cualquier tipo de procesos que impliquen la colaboración entre personas o entre personas y máquinas.

Los Productos Software de Workflow han evolucionado desde distintos orígenes, mientras que algunas ofertas se desarrollaron como software workflow puro, muchas otras evolucionaron de otros sistemas por ejemplo: sistemas de manejo de imágenes, sistemas de control de documentos, sistemas de bases de datos relacionales o de objetos y sistemas de correo electrónico.

Los vendedores que desarrollaron ofertas de workflow puro inventaron sus propios términos mientras que los que evolucionaron de otros productos han adaptado la terminología y las interfaces de dichos sistemas.

Los beneficios principales del workflow son:

- Eficiencia Mejorada – la automatización de muchos procesos del negocio conlleva a la eliminación de varios pasos innecesarios.
- Mejor control del Proceso – Mejoramiento de la Administración de procesos del negocio lograda a través de la estandarización de métodos de trabajo.
- Flexibilidad – El software de control de procesos permite su rediseño en línea con necesidades del negocio cambiantes.
- Mejoramiento en los procesos del negocio – enfocarse en los procesos del negocio nos lleva a su simplificación.

En la figura 1.1 se muestran las relaciones entre la terminología básica de Workflow [WfC11].

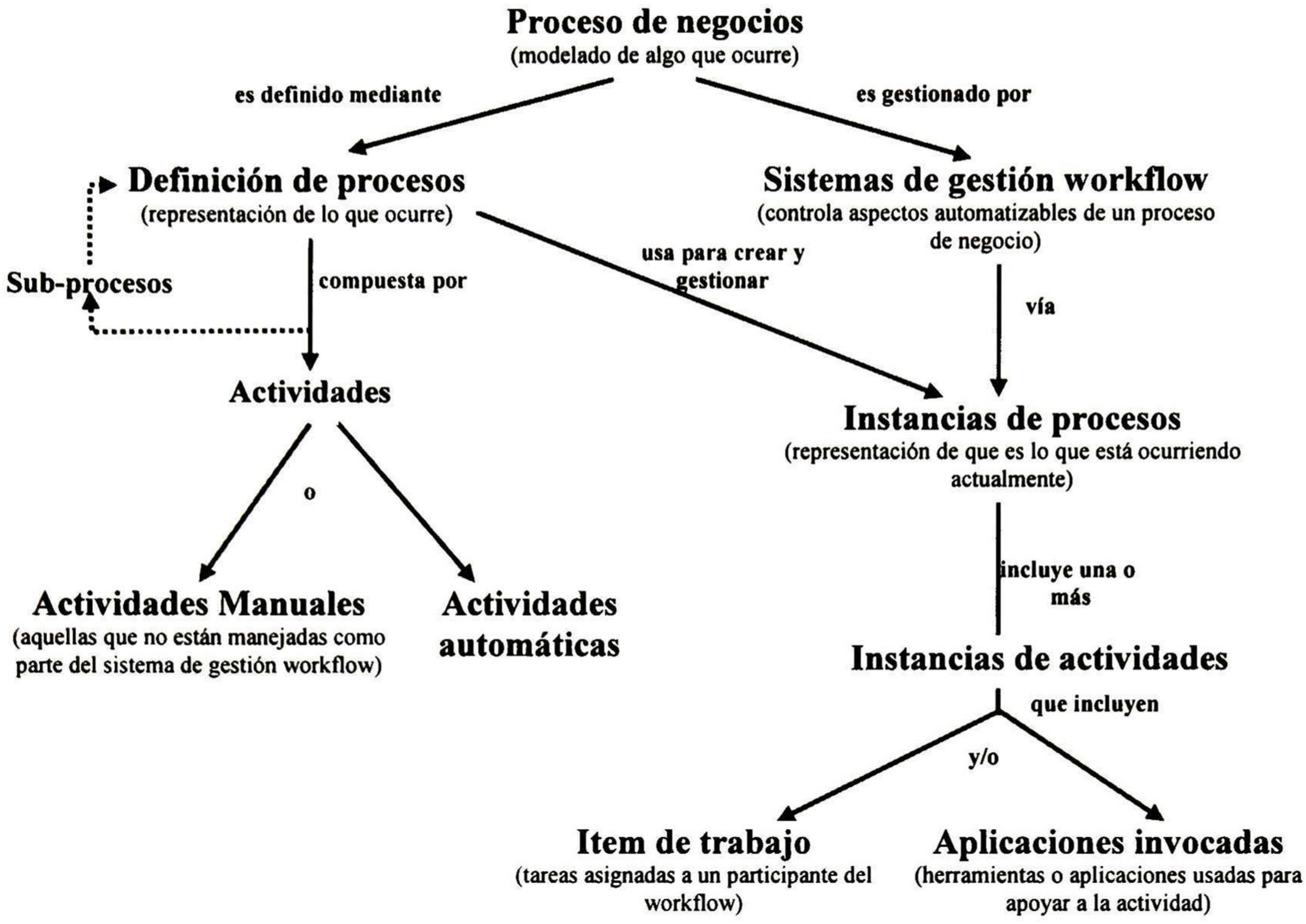


Figura 1.1: Relaciones entre la terminología Básica

1.1.2 Coalición para la Gestión de Flujos de Trabajo WFMC

Muchos vendedores tienen productos disponibles para la Administración de Flujos de Trabajo (WFM), pero tal disponibilidad y el amplio rango de productos dentro del mercado ha permitido a vendedores de productos individuales enfocarse en capacidades funcionales particulares y los usuarios han adoptado un producto en particular para satisfacer necesidades de aplicación específicas. Sin embargo, no hay estándares definidos para permitir que diferentes productos de WFM trabajen juntos, lo que da como resultado islas incompatibles de automatización de procesos.

En [WfC03] se encuentra que la Coalición para la Gestión de Flujos de Trabajo (WFMC por sus siglas en inglés – Workflow Management Coalition), fue fundada en 1993 y es un grupo de compañías que se han unido para tratar la situación descrita en el párrafo anterior. Estas compañías han estado estudiando que todos los productos para la administración de flujo de

trabajo tienen algunas características similares lo que les permite potencialmente lograr un nivel de interoperatividad a través del uso de estándares comunes para varias funciones.

La WFMC se ha establecido para identificar estas áreas funcionales y desarrollar las especificaciones apropiadas para su implementación en productos de Workflow. Se tiene la intención de que tales especificaciones permitirán la interoperabilidad entre productos workflow heterogéneos y la integración mejorada de aplicaciones workflow con otros servicios como correo electrónico y administración de documentos.

WFMC es una organización no lucrativa cuyo objetivo es incrementar las oportunidades para la explotación de la tecnología workflow a través del desarrollo de terminología común y estándares. Junto con esta organización un Comité Técnico es responsable por el desarrollo de las especificaciones técnicas apropiadas y documentos relacionados; dicho comité ha establecido un número de Grupos de Trabajo responsables de los diferentes aspectos del programa de estandarización. Especificaciones técnicas son desarrolladas por éstos Grupos de Trabajo individuales los cuales trabajan sobre el marco de trabajo del Modelo de Referencia (en inglés WFMC Reference Model). La figura 1.2 muestra los componentes e interfaces principales del Modelo de Referencia.

El número y nombre de cada uno de los documentos publicados por esta organización se encuentran en el documento número WFMC-TC-1002 [WfC02] llamado Document Index

- *Modelo de Referencia*

El [WfC03] dice que el Modelo de Referencia de Workflow ha sido desarrollado de estructuras de aplicación de workflow genéricos mediante la identificación de interfaces dentro de su estructura las cuales permiten a los productos interoperar en una variedad de niveles. Todos los sistemas workflow contienen un número de componentes genéricos que interactúan de manera definida; productos diferentes exhiben distintos niveles de capacidad.

Para lograr la interoperatividad entre los productos workflow son necesarios un conjunto estandarizado de interfaces y formatos de intercambio de datos entre los componentes. La figura 1.2 ilustra los componentes e interfaces principales dentro de la arquitectura workflow.

A continuación se presentarán de forma general los componentes principales del Modelo de Referencia.

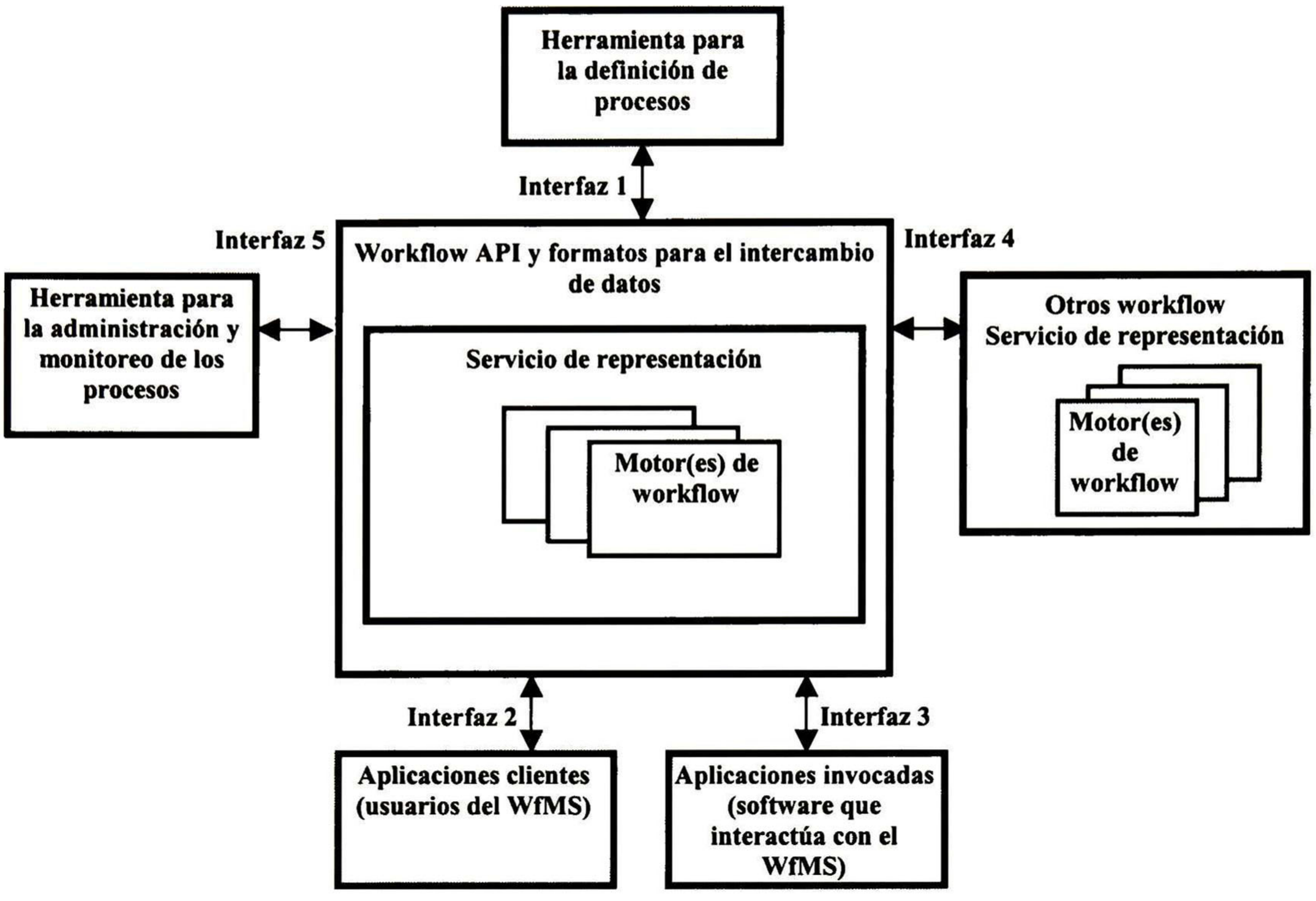


Figura 1.2: Modelo de Referencia de WorkFlow, Componentes e Interfaces

- Servicio de Representación

Este servicio provee un ambiente en tiempo de ejecución en el cual ocurren la inicialización y activación de procesos, utiliza uno o más motores de administración de workflow, es responsable de interpretar y activar parte, o el total, de la definición del proceso e interactuar con los recursos externos necesarios para procesar las distintas actividades.

Está definido como:

"Un servicio de software que puede consistir de uno o más motores workflow para crear, manejar y ejecutar instancias de flujo de trabajo. Las aplicaciones pueden interactuar con este servicio vía la interfaz programable de aplicaciones workflow (WAPI, del inglés Workflow Application Programming Interface)."

La Interacción con recursos externos accesibles ocurren vía una o dos interfaces:

- Interfaz de aplicaciones del cliente – Un motor workflow interactúa con un manipulador de la lista de trabajo responsable de la organización del trabajo; éste manipulador es responsable de seleccionar y avanzar work ítems individuales de la lista de trabajo.
- La interfaz de aplicaciones invocadas – Permite al motor workflow activar directamente una herramienta específica para emprender una actividad particular.

► Motor Workflow

Un motor workflow es responsable de parte (o todo) el ambiente de control de tiempo de ejecución dentro de un servicio de representación. Se define como:

"Un servicio de software o motor que provee un ambiente de ejecución de una instancia workflow."

Dicho software provee las facilidades para manipular:

- La interpretación de la definición del proceso
- El control de instancias de proceso – creación, activación, suspensión, terminación, etc.
- La navegación entre las actividades de proceso las cuales pueden implicar operaciones secuenciales o paralelas, programaciones de horarios, interpretación de datos de workflow relevantes, etc.
- El registro y salida de participantes específicos
- El mantenimiento de datos de control y datos relevantes de workflow

Un motor workflow puede controlar la ejecución de un conjunto de procesos o subprocesos, instancias con un alcance definido determinado por el rango de los tipos de objetos y sus atributos.

► Herramienta para la definición de Procesos

Una herramienta para la definición de procesos, permite modelar, describir y documentar un proceso de workflow, para esta tarea existen lenguajes y métodos de modelado para procesos de workflow, los cuales se basan en diferentes enfoques, como pueden ser: redes de Petri, UML (Unified Modeling Language), máquinas de estado finito, diagramas de flujo, entre otros más.

La salida de este proceso de modelado y diseño es una "definición de procesos" la cual puede ser interpretada vía la interfaz 1 en tiempo de ejecución por el(los) motor(es) de WorkFlow

a través del Servicio de representación. La interfaz 1 permite la comunicación entre éste y el componente de representación de workflow.

- Aplicaciones cliente

Este componente representa los programas de software utilizados por el usuario final del WfMS en las actividades que requieren participación humana. La interfaz 2 permite definir y manejar las listas de trabajo que se encuentran en los motores de workflow. Una lista de trabajo es una la lista asignada que contiene actividades por hacer, pendientes, recordatorios entre otras, las cuales deben ser ejecutadas por un usuario o grupo de usuarios.

- Aplicaciones invocadas

Este componente representa todo el software existente dentro de la organización, el cual es utilizado por el WfMS con el fin de interactuar y realizar ciertas actividades, estas aplicaciones de software pueden encontrarse en cualquier lugar dentro de la red de trabajo, un ejemplo es el software administrador de correos electrónicos. La interfaz 3 permite la comunicación entre éste componente y el de representación de workflow a nivel de invocación, transformación y representación de datos, de manera que éstos sean entendibles para el motor de workflow.

- Herramienta para administración y monitoreo

El componente Herramienta para administración y monitoreo y la interfaz 5 tienen como propósito permitir una vista completa del estado del workflow, además con este es posible realizar auditorias sobre los datos del sistema. Esta herramienta es utilizada por el administrador del WfMS y por los altos mandos de la organización los cuales tienen la necesidad de tener información exacta del estado de sus procesos de negocio.

1.2 Modelado de Workflow

1.2.1 Metodologías de Modelado

En general, una metodología de modelado workflow no sólo tiene que especificar cómo fluye el trabajo sino que debe abarcar tres perspectivas fundamentales:

- *Perspectiva funcional*. Donde se especifica cómo se constituyen los workflows.

- *Perspectiva de comportamiento*. En la que se especifica la forma en que se van a ejecutar las tareas, teniendo en cuenta dos aspectos fundamentales:
  - *Descriptivo*. Donde se indica cómo se ejecutan los procesos, es decir: en serie, paralelo o condicionales.
  - *Prescriptivo*. En la que se especifican las restricciones de ocurrencias de los workflows.
- *Perspectiva organizativa*. Donde se deben especificar las políticas de ejecución con respecto a los agentes y las aplicaciones.

Las principales Metodologías de modelado son:

- **Metodologías basadas en la actividad**. Estas metodologías se basan en la representación de las actividades que se van a producir en el desarrollo del workflow. En este caso se define la forma en que se van ejecutando los flujos de trabajo, identificándose principalmente tres formas de ejecutar las tareas: secuencial, paralela y condicional. En algunos casos se añade la ejecución repetitiva de tareas. Entre este tipo de teorías se encuentra el modelo de la WfMC [WfC11] en el que se establecen primitivas para la ejecución secuencial, la ramificación del flujo de trabajo en paralelo, la bifurcación alternativa del flujo de trabajo y la iteración (ver figura 1.3).
- **Metodologías basadas en la comunicación**. Las metodologías basadas en la comunicación tienen su origen en la teoría de Searle conocida como "*speech-act*" [CaJG99]. Entre este tipo de metodologías está la teoría "conversación para la acción".
- **Basadas en reglas y restricciones**. Son metodologías que definen los flujos de trabajo como formulaciones lógicas. Basan el modelado de procesos en el razonamiento lógico expresando los flujos de trabajo mediante de fórmulas.

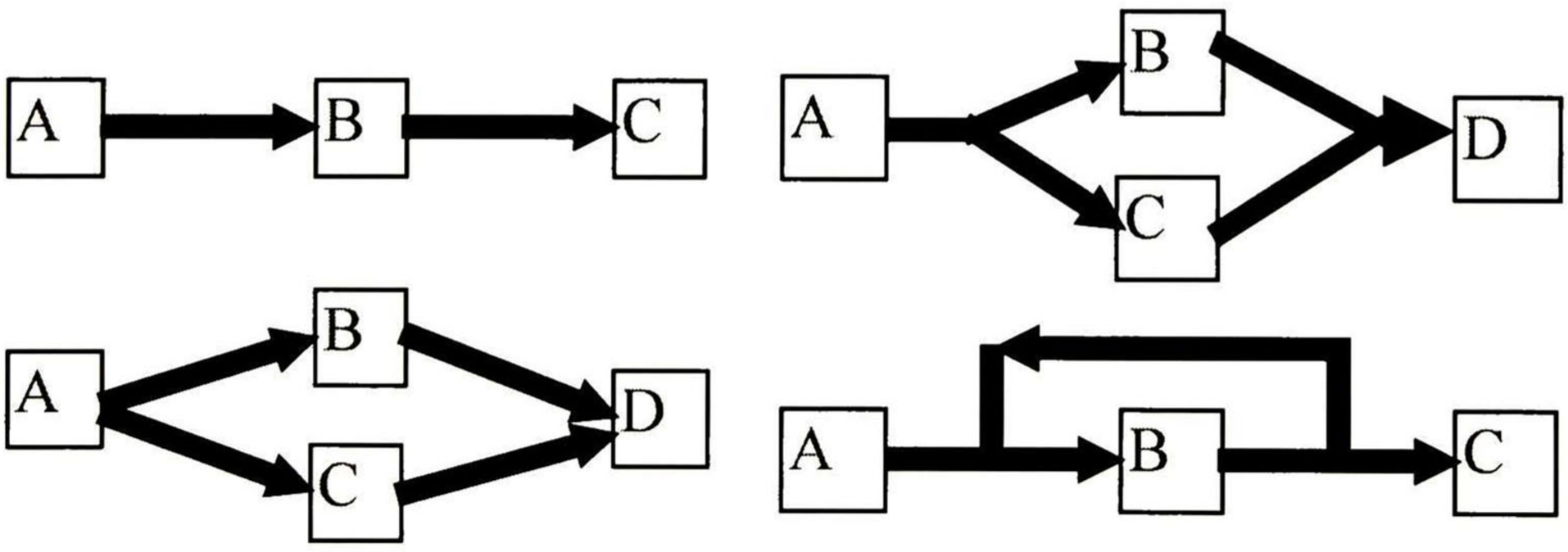


Figura 1.3: Algunas de las primitivas identificadas por la WfMC

1.2.2 Enfoques de Modelado

Existen varias técnicas de modelado de workflow, algunas son formales y otras no; pero es difícil saber cuál técnica es la apropiada ya que hay diferentes formas de resolver el mismo problema de modelado; el lenguaje adecuado debe de poseer suficiente poder de expresividad para poder modelar ciertos problemas. Muchas de estas técnicas de modelado se basan en constructores, algunos lenguajes permiten múltiples instancias de la misma actividad al mismo tiempo en el mismo contexto de workflow; algunos permiten estructuras repetitivas (ciclos) con un punto de entrada y uno de salida, algunos requieren actividades de terminación explícita mientras que otros la terminación es implícita. No es claro determinar cómo estas diferencias afectan al poder expresivo y conveniencia de un lenguaje de modelado dado.

- *Patrones de flujo de Trabajo*

Aalst en [ABH02] identifica requerimientos para lenguajes de workflow por medio de patrones (patterns) los cuales son definidos en [RZ96] como "una abstracción de una forma concreta que permanece recurrente en contextos no arbitrarios específicos"; éstos patrones de workflow tratan requerimientos de negocios en un estilo de expresión de workflow imperativo pero independiente de lenguajes workflow actuales. El artículo comienza con patrones simples y luego describe los complejos, además identifica los problemas de las rutinas de construcción complejas y posibles soluciones. Se proponen 20 patrones agrupados en 6 grupos y en [ABH00] reporta 4 de los patrones más complejos o de construcción avanzada.

- *UML*

Como se mencionó anteriormente, muchos lenguajes se han propuesto sin bases formales con lo que fácilmente se llega a especificaciones que son inherentemente ambiguas además de que no pueden ser validadas, verificadas o analizadas formalmente; uno de éstos lenguajes es UML (ver figura 1.4), en [DMH01] Marlon Dumas et al. examina la expresividad y utilidad de los diagramas de actividad para especificaciones de workflow evaluando su habilidad para capturar una colección de los patrones descritos en [ABH02], y aunque, los diagramas de actividad son capaces de describir situaciones que no pueden ser capturadas por muchos de los WFMS comerciales, no capturan algunas de las situaciones más útiles presentadas en el

modelado de procesos; además, por no estar soportadas por una base formal, estos diagramas suelen presentar ambigüedades.

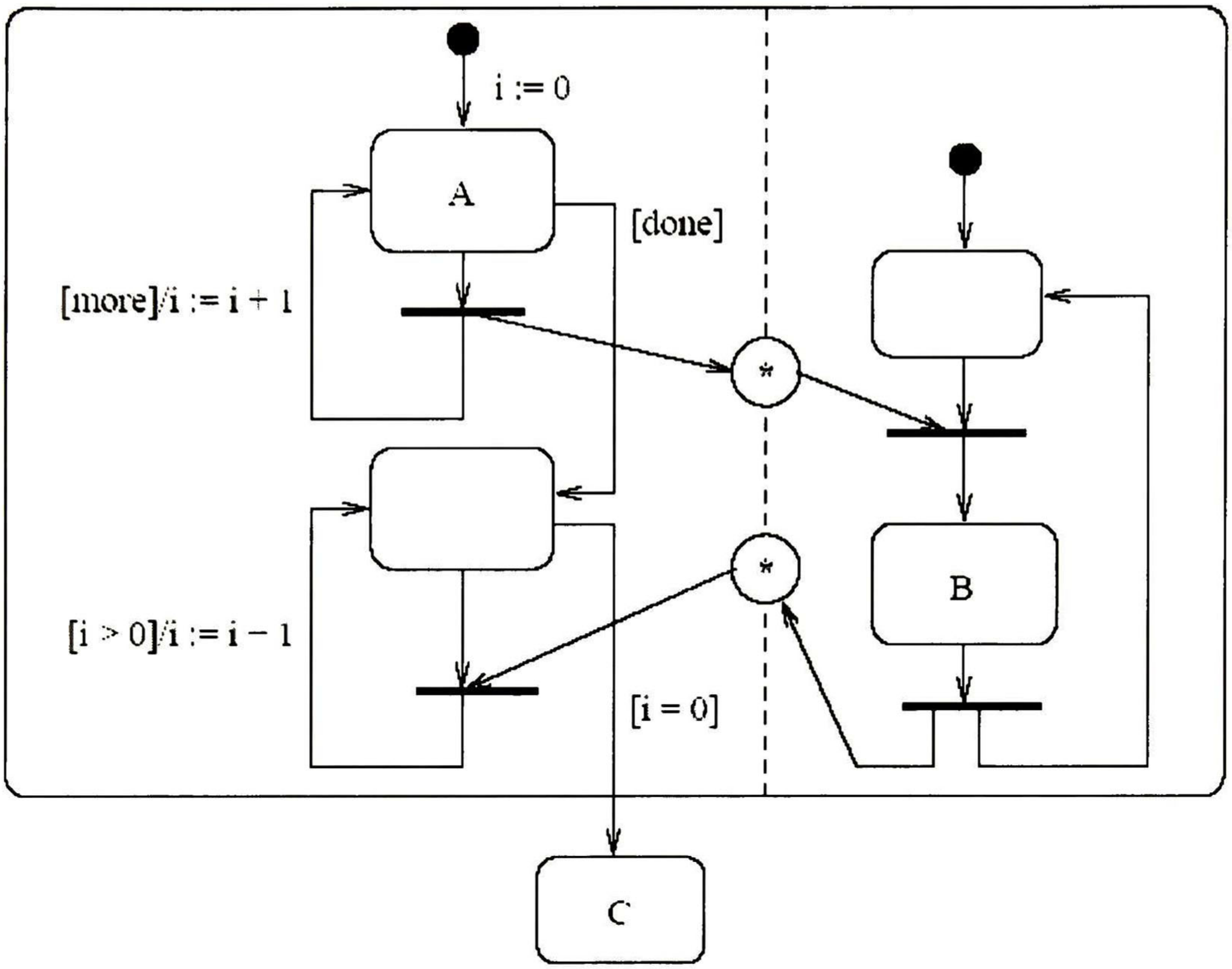


Figura 1.4: Diagrama de Actividad para el patrón productor-consumidor con terminación de actividad.

- *REDES DE PETRI*

Existen diversos trabajos de investigación en los que se han utilizado las Redes de Petri (RP) como base formal para el modelado de workflow; uno de estos trabajos es [Aal98], en el que se argumenta que existen varias razones para utilizar las RP para el modelado de Workflow algunas de éstas son: gracias a su semántica formal tienen una definición clara, por su naturaleza gráfica con intuitivas y fáciles de aprender; soportan todas las primitivas necesarias para modelar procesos workflow, muchas personas han investigado las propiedades básicas de las RP por lo que permiten un razonamiento de las mismas; son caracterizadas por la disponibilidad de muchas técnicas de análisis las cuales pueden usarse para probar las propiedades de vivacidad, bloqueos, etc; además, proporcionan un marco de trabajo independiente de la herramienta para el modelado y análisis de procesos y no están basadas en algún paquete específico de algún vendedor.

Muy pocas técnicas de modelado propuestas cuentan con una especificación formal de la semántica. En [Aal98] Aalst argumenta que al utilizar Redes de Petri como una base formal para workflows, proporciona al modelador de workflow varias técnicas de análisis que ya han sido utilizadas para Redes de Petri por lo que permiten una verificación formal de los modelos de workflow; para esto define y utiliza las WF-nets cuya característica principal es que son redes que poseen un lugar de inicio llamado fuente y un lugar final llamado pozo.

También en [AalW] Aalst describe cómo las redes de petri pueden utilizarse para construir modelos y cómo pueden aprovecharse las bases formales para realizar comprobaciones de solidez, además de argumentar que existe una técnica que soporta la validación la cual se llama herencia (en inglés inheritance) de comportamiento dinámico, la cual puede utilizarse como técnica para comparar procesos.

En [AaA99] Aalst presenta una aproximación (basada en las WF-nets introducidas en [AalW]) para el diseño de worflows interorganizacionales que soporta la cooperación de socios preservando la autonomía de las organizaciones; con lo que demuestra cómo el formalismo de las Redes de Petri puede ser utilizado para analizar la estructura de las definiciones de proceso de workflow (en [WfC11] se define como: "la representación de un proceso del negocio en una forma tal que soporte manipulación automatizada como modelado; una definición consiste de una red de trabajo de actividades y sus relaciones, criterios para indicar el inicio y la terminación de los procesos e información sobre actividades individuales").

Sin embargo, en [AaH02] Aalst también reconoce que las RP, como lenguaje de modelado de workflow, poseen limitaciones para representar: patrones que requieren múltiples instancias, patrones de sincronización avanzada y patrones de cancelación. Por lo que presenta una herramienta basada en RP llamada YAWL (del inglés, Yet Another Workflow Lenguaje), para superar las limitaciones impuestas por las RP, la figura 1.5 muestra que a pesar de estar basadas en estas redes, YAWL introduce sus propias estructuras de construcción de modelos con redes EWF-nets (del inglés Extended Workflow nets) y, aunque menciona que una EWF-net puede pasarse a una Red de Petri , no está clara la forma de hacerlo y no especifica las propiedades que pueden validarse en los modelos con YAWL.

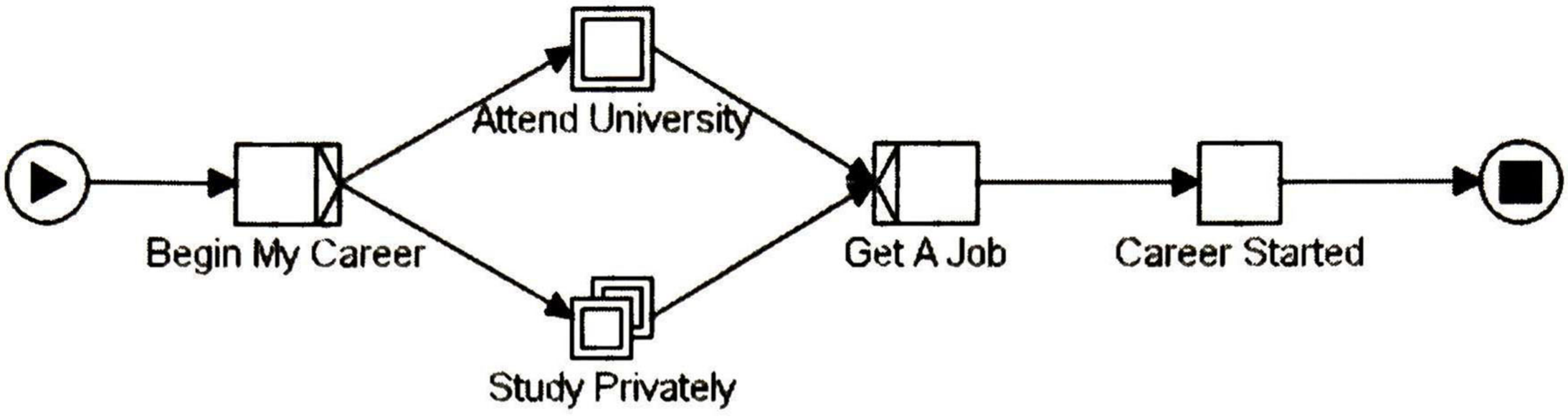


Figura 1.5: Ejemplo de una Red Modelada con YAWL

En [NVW98] Nabil R. Adam et al. presenta una técnica para modelar y analizar workflows utilizando RP, en donde resalta que las tareas poseen dependencias y que son conocidas como dependencias intra-workflow, las cuales pueden ser dinámicas o estáticas; la idea principal es construir los modelos de workflow basándose en las RP e identificando dependencias entre las tareas y probar la seguridad de los modelos. Además presenta que las RP:

- Facilitan la especificación de aplicaciones de workflow
- Sirven como una herramienta poderosa para el modelado de un sistema bajo el estudio en un nivel conceptual
- Permiten una transición suave entre el nivel conceptual a una implementación de prueba
- Permite el análisis, simulación y validación de los sistemas antes de proceder a su implementación.

◆ *Lógica Transaccional y lógica operacional*

Otra tentativa de proporcionar bases teóricas para especificaciones de workflow puede encontrarse en [OrA05] donde discute sobre los formalismos de Lógica Temporal y Lógica transaccional. En cuanto a lógica temporal argumenta que las dependencias de intertareas en el workflow pueden modelarse utilizando lógica computacional de árboles y donde un programador central es el responsable de manejar la ejecución de las tareas la cuales pueden estar en los estados de: ejecución, no ejecución, realizada, abortada y comprometida; además el workflow se especifica utilizando diagramas de Activad de UML y los requerimientos son especificados utilizando un lenguaje de requerimientos abstracto. Aunque no es posible representar muchas dependencias con esta técnica.

En cuanto a la Lógica Transaccional se argumenta que el workflow se especifica como un conjunto de fórmulas de esta lógica que describen las dependencia entre las tareas, las cuales se modelan por predicados; una teoría de prueba puede utilizarse para probar modelos de workflow y dichas pruebas describen ejecuciones de workflow; se concluye que esto es difícil de implementar.

Existen muchos productos comerciales disponibles, algunos se basan en un lenguaje formal (como las RP) y algunos otros definen su propio lenguaje de modelado.

1.3 Gestión de Flujo de Trabajo

1.3.1 Definición de un Sistema de Gestión de Flujo de Trabajo

Un Sistema de Gestión de Flujo de Trabajo (del Inglés WorkFlow Management System **WFMS**), es aquel que provee un procedimiento automatizado de procesos del negocio mediante la administración de la secuencia de actividades de trabajo y la invocación de el recurso apropiado.

La WFMC en [WfC11] define a un WFMS como:

"Un Sistema que define, crea y maneja la ejecución de flujos de trabajo a través del uso de software ejecutándose en uno o más motores Workflow, los cuales son capaces de interpretar la definición del proceso, la interacción con los participantes workflow y, cuando sea requerido, invocar el uso de los Sistemas de Información implicados en el trabajo."

Todos los WFMS exhiben ciertas características en común las cuales proporcionan las bases para desarrollar las capacidades de integración e interoperabilidad entre diferentes productos. Como ya se vió, el Modelo de Referencia describe un modelo para la construcción de sistemas workflow.

En [WfC03] Hollingsworth destaca que todos los WFMS pueden ser caracterizados como proveedores de servicio en tres áreas funcionales:

- Funciones en tiempo de diseño (Build-Time), relacionadas con la definición del proceso de negocio y las actividades que lo constituyen.

- Funciones de control en tiempo de ejecución (Run-Time), relacionadas con el manejo de procesos de workflow en un ambiente operacional y secuenciando las diversas actividades a ser manipuladas como parte de cada proceso.
- Interacciones en tiempo de ejecución (Run-Time interactions) con personas, aplicaciones y sistemas de información.

Su uso proporciona una serie de ventajas competitivas, entre las que cabe destacar:

- *Proporcionar un modelado global de todos los procesos*. Es de gran utilidad para los gestores del hotel poseer una visión global de todos los procesos, así como poseer un sistema que monitorice su ejecución.
- *Marco adecuado para la re-ingeniería de procesos (BPR)*. No es posible realizar reingeniería de los procesos sin un modelo de los mismos. Workflow proporciona un marco para desarrollar modelos formales.
- *Marco para la ejecución de procesos*. Esta tecnología abarca desde el modelado hasta la ejecución real mediante un sistema de gestión workflow.
- *Aumento de la satisfacción del cliente*. Puesto que los servicios proporcionados serán de mayor calidad.
- *Adaptabilidad ante los cambios organizativos*. El modelo workflow es adaptable según las necesidades del momento sin necesidad de modificar el WFMS.
- Mejora en el manejo de documentos e información
- Mejora en el trabajo del personal

1.3.2 Productos para la gestión de Flujos de Trabajo

Algunos de los productos disponibles de workflow soportan directamente todos (o casi todos) de los patrones identificados por Aalst, pero muchos otros solo soportan algunos y otros los soportan pero con construcciones complicadas y un tanto ambiguas. En [Bar03] Bartek realiza un estudio comparativo de ocho diferentes lenguajes de modelado soportados por productos de administración de workflow disponibles (Staffware, Visual WorkFlo, Forte Conductor, Changengine, Staffware, Fujitsu i-Flow, MQSeries Workflow, Verve y SAP R/3 Workflow) e indica qué patrones de workflow, de los reportados por Aalst, soportan y cómo, además, muestra un modelo de proceso de ejemplo y cómo es modelado por cada uno de los productos estudiados,

así también introduce las estructuras generales que define cada uno de estos sistemas para construir un modelo de flujo de trabajo.

De este sitio [AalWs] también pueden encontrarse algunos otros productos comerciales (por ejemplo: BizAgi, Pectra, COSA, FLOWer, etc.) disponibles con su respectivo análisis de los patrones soportados y la estructura que utilizan para hacerlo.

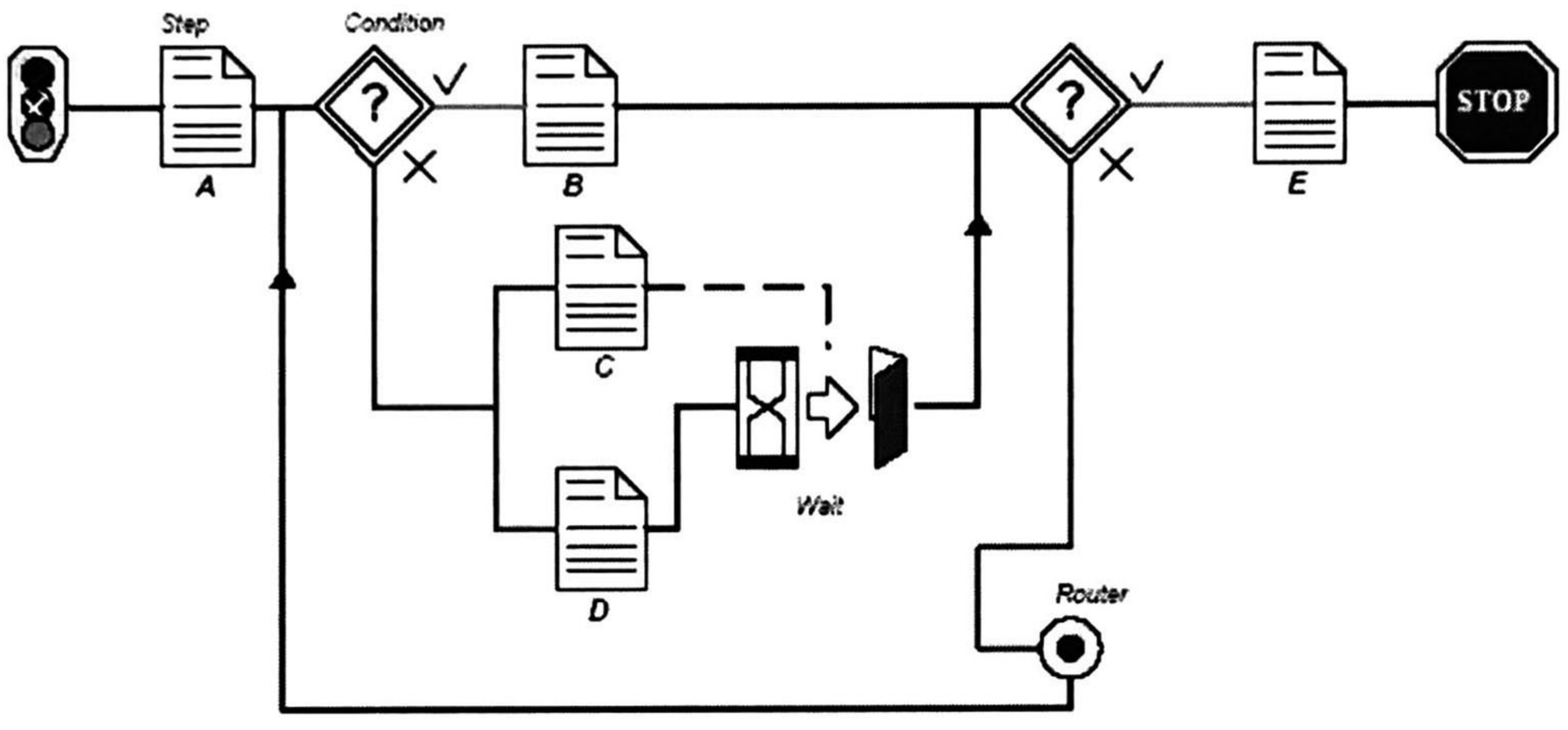


Figura 1.6: Ejemplo de una Red Implementada en Staffware

1.4 Enfoque a Adoptar

Para el desarrollo de esta tesis adoptaremos una extensión de las RP como formalismo de modelado. Se trata de nLNS (el cual se introducirá con más detalle en el capítulo II sección 2) en el cual se emplean redes como el marcado de otras redes permitiendo así un modelado multinivel. Emplearemos este formalismo para mostrar que es posible modelar problemas de Flujo de trabajo con esta herramienta con lo que se simplificará el modelado a redes más simples y fáciles de entender.

Para efectos prácticos modelaremos el problema utilizando solo 3 niveles de abstracción en un ejemplo de sencillo y después de aplicará la metodología descrita a un caso de estudio. Por lo cual tendremos que :

1. En la red de nivel 1 modelaremos la estructura general de la organización representando solo los departamentos involucrados en el problema de flujo de trabajo, en donde cada lugar nos representará un departamento de la empresa y las transiciones representarán el flujo de información de un departamento a otro dentro de la empresa; por lo que el disparo de una

transición representará que las tareas que se realizan dentro del departamento del lugar de entrada a la transición, ya terminaron su ejecución (se realizaron) y que la información se pasará al departamento representado por el lugar de salida de dicha transición. Esta red tendrá como marcado inicial a la red de nivel 2 la cual se describirá de manera general en el punto siguiente. Para la codificación, cada lugar de la red de nivel 1 se tomará como un host (sitio) dentro de un sistema en red, por lo que en términos de un sistema distribuido, ésta red modelará el ambiente en el cual el agente se moverá con objeto de realizar el proceso de flujo de trabajo modelado a partir de la definición del problema.

2. La red de nivel 2 la utilizaremos para modelar a el agente que irá guiando el proceso, por lo que cuando esta red agente este como marcado en un lugar de la red de nivel 1, significará que el agente se encuentra en el host del departamento (unidad organizacional) que representa del lugar de dicha red de nivel 1 y un disparo de una transición de esta red (red ambiente) nos representará la migración del agente de un host de la red a otro (movilidad de un agente modelada con el disparo de una transición) por lo que la red agente deberá tener una transición que modele la migración del agente y deberá estar sincronizada externamente con la red ambiente.
3. Para el nivel 3 modelaremos dos grupos diferentes de redes:
   a. Tipo 1: se empleará para modelar el proceso de flujo de trabajo, es decir, modelará las tareas involucradas en el problema de flujo de trabajo en el orden en que deberán realizarse en cada caso.
   b. Tipo 2: cada red de este tipo modelará las operaciones que conforman la ejecución de cada tarea, por lo que tendremos tantas redes de este tipo, como tareas involucradas en la definición del proceso del flujo de trabajo modelada en la red de nivel 3 tipo 1.

En términos generales, la red de tipo 1 modelará los comportamientos específicos del agente; y la red de tipo 2 modelará a más detalle las operaciones o pasos involucrados de cada comportamiento. Ambos tipos de redes se utilizarán para definir el marcado inicial de la red agente de nivel 2 con lo que tendrá así un modelado a 3 niveles del problema de flujo de trabajo.

Además se desarrollará una metodología para pasar del modelado multinivel del workflow a una implementación computacional basada en agentes; para ello se empleará la middleware JADE.

CAPITULO II.

Enfoque Multi-agente para la Coordinación de Flujo de Trabajo

Resumen. En este capítulo se presentan los conceptos básicos del paradigma de los Sistemas Multi-agente. Se muestra la definición del formalismo nLNS que se utilizará como marco de modelado para problemas de flujo de trabajo.

2.1 Sistemas multi-agentes

Un sistema multi-agente puede ser definido como un sistema compuesto de múltiples e interactivos elementos computacionales conocidos como agentes, los cuales son capaces de realizar acciones autónomas y de interactuar con otros agentes. Para lograr una interacción exitosa estos agentes requieren la habilidad de cooperar, coordinar y negociar unos con otros.

2.1.1 Conceptos sobre Agentes

- *Definición de Agente*

No existe una definición de agente universalmente aceptada, sin embargo existe el consenso de que la *autonomía* es una parte esencial de un agente. Wooldridge [WooM02] define a un agente como: "un sistema computacional que está situado en un ambiente y que es capaz de realizar acciones autónomas en esta ambiente con el propósito de lograr sus objetivos para los que fue diseñado".

El concepto de agente proviene de una poderosa forma de describir una entidad de software compleja que es capaz de actuar con cierto grado de autonomía con el propósito de lograr sus tareas. Un agente se define en términos de su comportamiento.

Nwana [NwH96] describe a un agente como: "un componente de software y/o hardware que es capaz de actuar rigurosamente con el fin de cumplir sus tareas"

Las definiciones propuestas por diferentes autores incluyen los conceptos de :

1. Persistencia: software que se ejecuta de manera continua y decide por si mismo cuando debe realizar alguna actividad.
2. Autonomía: agentes que tienen la capacidad de seleccionar tareas, prioridades, comportamiento orientado a metas, toma de decisiones sin intervención humana.
3. Habilidades sociales: agentes que son capaces de comunicarse y coordinarse con otros para colaborar en la realización de alguna tarea.
4. Reactividad: agentes que perciben el contexto en el cual operan y reaccionan a él apropiadamente.

- *Ventajas y desventajas de los Sistemas Multi-agentes*

Modelar los sistemas complejos como un SMA tiene las siguientes ventajas:

- Se evitan problemas de limitación de recursos o el riesgo de fallas en situaciones críticas que se podrían tener al encomendar un problema grande a un solo agente.
- Se permite la interconexión e interoperabilidad de sistemas existentes.
- Se obtiene una solución flexible a un problema grande entre una sociedad de agentes interactuando. Por ejemplo, una negociación multi-agente para la compra/venta de artículos o la búsqueda de información distribuída en el internet.
- Se tienen soluciones eficientes a problemas en los que se requiere el uso de información que está espacialmente distribuída.
- Se permite la extensibilidad: el número y la capacidad de los agentes trabajando en el sistema puede cambiar dinámicamente.

- *Aplicaciones de los sistemas Multi-agentes*

Algunos ejemplos de aplicaciones de los Sistemas Multi-agente son:

- Diagnóstico distribuido, donde los agentes con diferentes niveles de conocimiento y control (segmentos de red) comparten sus interpretaciones locales para llegar a explicaciones y respuestas consistentes (ejemplo: diagnóstico de redes, recolección de información en Internet, redes de sensores distribuidos).
- Planeación y programación de recursos distribuídas, donde los agentes (asociados a cada celda de trabajo) deben coordinar sus actividades para evitar y resolver conflictos

sobre recursos, y así maximizar el rendimiento del sistema (ejemplo: planificación de una fábrica, administración de redes, ambientes inteligentes).

- Sistemas expertos distribuidos, donde los agentes comparten información y negocian sobre soluciones colectivas en base a su especialización y criterio de solución (ejemplo: ingeniería concurrente, restauración del servicio de red).
- Simulación de modelos, en donde los agentes representan las entidades que se desenvuelven en un ambiente, que tienen un comportamiento y que interactúan con otras entidades.

♦ *Clasificación de Agentes*

Existen muchas formas de clasificar a los agentes, una de ellas es la vista en [NwH96], el cual clasifica a los agentes de acuerdo a 3 características principales (ver figura 2.1):

- Autonomía: se refiere al principio de que los agentes pueden operar por sí mismos sin la necesidad de guía humana. Un elemento principal de la autonomía es su habilidad para tomar la iniciativa, a diferencia de actuar simplemente en respuesta a su ambiente [WooM02].
- Aprendizaje: la habilidad de aprender de su ambiente.
- Cooperación: los agentes necesitan poseer habilidad social, la habilidad de interactuar con otros agentes y posiblemente humanos por medio de algún lenguaje de comunicación.

Otras clasificaciones se basan en diferentes criterios: movilidad (agentes estacionarios y móviles), su nivel de inteligencia, su benevolencia (agentes cooperantes), o su egoísmo.

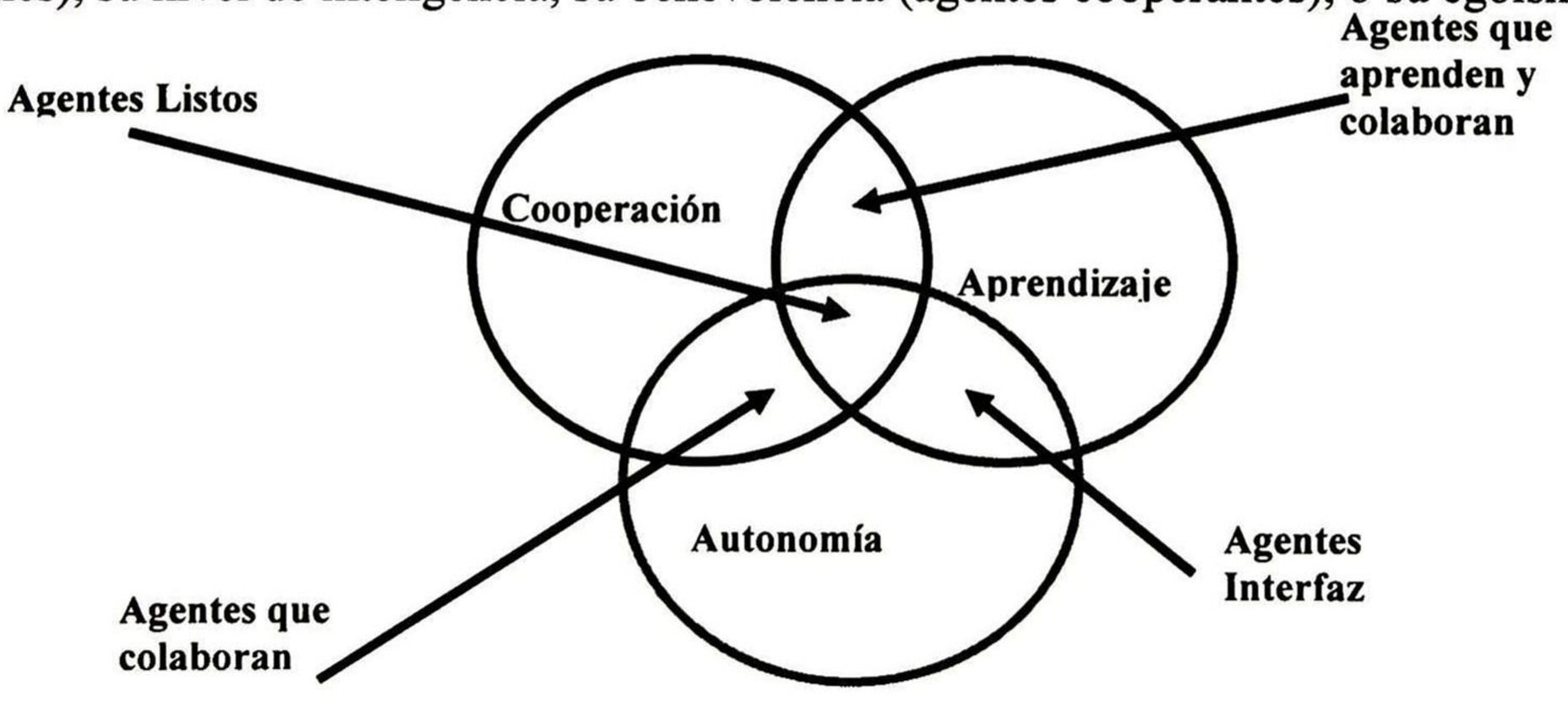


Figura 2.1: Categorías de Agentes Software

- *Conveniencia de usar Agentes*

Algunas de las razones más importantes para utilizar agentes son:

- Reducen el trabajo humano: muchas tareas pueden realizarse más rápido y a tiempo mediante agentes.
- Manejo de sobrecarga de información: los agentes pueden hacer una revisión minuciosa de información que se encuentra en la red de manera basta y sin estructurar.
- Proporcionan ayuda automatizada a un amplio número de usuarios.
- Proporcionan una metodología nueva y poderosa para desarrollar sistemas de software complejos.

2.1.2 Agentes Móviles

Los agentes móviles son programas autónomos que pueden viajar de una computadora a otra bajo su propio control. Pueden ofrecer un marco de trabajo robusto, conveniente y eficiente para implementar aplicaciones distribuídas incluyendo aplicaciones móviles.

Los agentes móviles han sido usados para el desarrollo de varias aplicaciones ([BGM99], [CLZ98], [MDW99]) permitiendo el uso de mejores funcionalidades.

- *Características de los Agentes Móviles*

Un agente estacionario se ejecuta únicamente en el sistema donde comenzó su ejecución y si necesita información de otro sistema o necesita interactuar con un agente de otro sistema, utiliza comunicación estándar de cliente-servidor (RMI, RPC, CORBA).

Un agente móvil no está atado al sistema donde comenzó su ejecución; es capaz de moverse entre los nodos de una red o del ambiente donde se desenvuelve, modificando eventualmente su ambiente de ejecución; transporta consigo su estado actual y su código.

Considerando el estado de un agente, podemos encontrar dos tipos de movilidad:

- Movilidad débil: El código y el valor de las variables del agente se transfieren al nuevo ambiente de ejecución.

- Movilidad fuerte: El código, valor de las variables y el estado de ejecución del agente se transfieren al nuevo ambiente de ejecución. Así, el agente puede comenzar su ejecución en el punto exacto donde se quedó antes de comenzar el movimiento.

Si el código del agente es interpretado, el acceso al estado de ejecución es difícil de obtener. Si el código del agente es compilado antes de la ejecución, el estado de ejecución es representado por una pila, transportar la pila y recompilarla en un distinto host no es una tarea trivial.

La movilidad también puede ser implícita o explícita. Cuando la movilidad es explícita el agente solicita explícitamente el cambio de ambiente de ejecución. Mientras que en la movilidad implícita el ambiente es quien decide cuándo mover al agente.

- *Ventajas*

Los agentes móviles poseen varias ventajas, mediante la migración a la localidad del recurso necesitado el agente puede interactuar con el recurso sin transmitir datos intermedios a través de la red conservando el ancho de banda y reduciendo la latencia. De igual forma, mediante la migración a la localidad del usuario, el agente pude responder a las acciones del usuario más rápidamente. En cualquiera de los casos mencionados el agente puede continuar su interacción con el recurso o con el usuario incluso si la conexión con la red se pierde temporalmente.

La mayoría de las aplicaciones distribuídas encajan naturalmente en el modelo de agentes móviles debido a que estos últimos pueden migrar secuencialmente a través de un conjunto de computadoras, envían a otros agentes-hijo a visitar máquinas de manera paralela, permanecen estacionarios e interactúan con recursos remotamente ó cualquier combinación de estas tres características.

Otras de las ventajas de trabajar con agentes móviles son :

* Superar las limitaciones de un cliente, como son memoria, poder de cómputo, retardos de comunicación, almacenamiento limitado, etc., enviando a un agente para que ejecute su código cerca de la fuente de datos.
* Adaptabilidad. Los agentes móviles pueden adaptarse fácilmente a las necesidades del usuario, y ser enviados al servidor donde las peticiones son ejecutadas, a diferencia del modelo cliente-servidor.

- Representación de un usuario desconectado. Se puede mandar un agente a realizar una tarea, desconectarse, y cuando se conecta de nuevo, se recupera el agente.
- Facilidad de desarrollo. Para muchos programadores, es más fácil programar cuando existe una analogía con el mundo real (por ejemplo, el problema del agente viajero).

◆ *Aplicaciones*

Estas son algunas de las aplicaciones que pueden tomar ventaja del uso de la tecnología de agentes móviles [CLZ98], [MDW99]:

- El comercio electrónico. La infraestructura para el comercio electrónico está emergiendo rápidamente. Protocolos estandarizados como la especificación de pagos por Internet *Secure Electronic Transaction* (SET) desarrollada por MasterCard y Visa, son usados para proveer seguridad a las transacciones.
- Distribución de software. Los agentes pueden automatizar el proceso de instalación y mantenimiento de software por medio de la instalación personalizada, verificación de versión on-site, e inventariado de paquetes de software.
- Recuperación de información. Un agente puede visitar varios stios buscando información relevante para los intereses del usuario, y regresar a su sitio original con los resultados de la búsqueda.
- Administración de sistemas. Los agentes móviles pueden desplazarse a través de los nodos de una red y realizar tareas administrativas (respaldos, identificación de archivos *core*, etc); también pueden realizar un diagnóstico inicial de problemas on-site antes de la intervención de un operador; además pueden usarse para automatizar tareas rutinarias, y pueden revisar periódicamente el estado de sistemas y de la red.
- Administración de la red. Algunas de las tareas de administración de red que pueden realizar los agentes son la instalación de actualizaciones de software, análisis del rendimiento, auditoria y comprobación de la red.
- Sistemas de cómputo móvil. Los agentes móviles pueden usarse para modelar aplicaciones que residen en un Asistente Digital Personal (PDA) o computadora portátil y usan recursos distribuidos en la red. El agente es enviado a la red, visita varios hosts, y regresa al PDA.

- Procesamiento de datos. Si se necesita realizar un procesamiento costoso sobre una base de datos remota, mejor se envía al agente al host donde está la base de datos, se realiza el procesamiento localmente, y el agente regresa con el resultado.

- *Esfuerzos de Estandarización en Agentes*

Existen algunas organizaciones que tiene el fin de establecer estándares en la programación de agentes con el fin de facilitar interoperabilidad entre agentes, algunas de éstas son:

- Object Management Group (OMG). Organización que recomienda estándares para la tecnología de agentes.
- FIPA (Foundation for Intelligent Physical Agents).Organización no lucrativa [FIPA] que promueve el desarrollo de especificaciones de tecnologías de agentes genéricas que maximicen la interoperabilidad entre agentes o aplicaciones basadas en agentes.
- DARPA (Defense Advanced Research Projects Agency). Agencia que tiene varios programas relacionados con agentes y control de Sistemas basados en agentes

2.1.3 WorkFlow y Tecnología de Agentes

La tecnología para la Administración de Flujo de trabajo promete proporcionar una forma eficiente para modelar y controlar procesos de negocio complejos dentro y entre las organizaciones. Los beneficios de esta tecnología incluyen:

- Definición de procesos explícita
- Rápida reacción en ambientes cambiantes
- Fácil seguimiento de operaciones

Como la administración de workflow se enfoca en manejar la lógica de los procesos, es necesario integrar otras tecnologías para controlar totalmente los procesos del negocio incluyendo la asignación de actividades y la distribución de recursos. La tecnología de agentes proporciona soluciones flexibles, distribuídas e inteligentes para la Administración de los procesos del negocio.

En [YZW01] se toma la definición de agente como "un sistema computacional situado en algún ambiente y que es capaz de acción autónoma en ese ambiente con el objetivo de lograr sus objetivos de diseño" Además enumera los beneficios de aplicar la tecnología de agentes a la administración de los proceso del negocio:

- Arquitectura Distribuída. Proporciona una estructura de para integrar sistemas administradores distribuidos de procesos del negocio.
- Automatización. La autonomía inherente de los agentes software pueden satisfacer actividades como substitución de una persona. Los agentes pueden iniciar workflow basados en eventos.
- Interacción. Los agentes software permiten organizaciones para interactuar unos con otros por medio de una semántica de intercambio de mensajes.
- Administración de Recursos. Los agentes pueden representar recursos.
- Reactividad. Los agentes pueden reaccionar a circunstancias cambiantes y tienen la habilidad de generar caminos alternativos de ejecución.
- Interoperación entre sistemas heterogéneos. Los agentes pueden ser heterogéneos, la interacción recae en una semántica de mensajes para intercambiar definiciones planes y servicios.
- Hacer decisiones inteligentes, algunas características de alto nivel de los agentes, como la capacidad de aprendizaje, son muy útiles en la administración de workflow.

♦ *Clasificación de las aplicaciones de agentes a sistemas de Flujo de Trabajo*

En [YZW01] se clasifican las aplicaciones de agentes a sistemas administradores de workflow, en dos formas:

- Administración de Workflow Agente-Aumentado, en este escenario los agentes son como servicios proporcionados por el sistema administrador de workflow, el objetivo de usar agentes es incrementar la automatización de los sistemas workflow. Desde el punto de vista del sistema, los agentes no necesariamente interactúan unos con otros, el motor de workflow controlar sus acciones.
- Administración de Workflow Basado en Agentes, en este escenario la lógica del proceso esta embebida en los agentes, los agentes son independientes entre ellos y responsables de la ejecución de los procesos; el proceso del negocio en su totalidad está formado por subredes en estos agentes.

En [RPB03], se toma esta clasificación y se describe una metodología para diseñar un sistema administrador de workflow basado en agentes. La metodología consiste de tres pasos. En el primer paso se modela el proceso de negocio con ayuda de diagramas de Actividad de UML identificando todos los recursos y actividades requeridas. En el segundo se agrupan estas actividades con sus roles en caminos paralelos. El tercer paso es tomar cada uno de estos grupos separados en paralelo y definir por cada uno un agente definiendo su comportamiento.

Hay algunos otros autores que toman la tecnología de agentes para mejorar aplicaciones de flujo de trabajo, por ejemplo, en [MaB05] se propone una arquitectura workflow de alto nivel basada en agentes, en la cual separan la ejecución del workflow y el control del flujo del proceso en pequeñas unidades de ejecución manejadas por agentes inteligentes; se propone que el proceso de workflow pueda ser manejado en una forma descentralizada y lo implementan en un sistema llamado JITIK, el cual es un sistema que entrega la información correcta a la gente adecuada en el tiempo correcto.

En [MHD05] se aplica la tecnología de agentes inteligentes al monitoreo de workflow con el fin de proveerle más flexibilidad e inteligencia; para esto se propone un Agente Usuario el cual ayuda a las peticiones de monitoreo de usuarios y un Agente Planeador que es capaz de generar planes individuales de monitoreo basados en las entradas de los usuarios; así los agentes inteligentes son diseñaros para proporcionar un plan de monitoreo flexible basado en peticiones individuales de monitoreo de usuarios.

2.2 Formalismo nLNS

El Formalismo nLNS se ha estado desarrollando desde trabajos anteriores, primero con [Alyd02] se introdujo el NS-3 para el modelado de agentes móviles con lo que se representaba la movilidad en los agentes por medio de movimiento de marcas, además de presentar las ventajas de modularidad (en donde un sistema agente se abstrae a 3 niveles de actividad: ambiente, agente y los elementos que guían su comportamiento) y compacidad (en el cual las redes del 3er nivel son similares a las Redes de Petri Coloreadas).

Después en [ViN03] se propone el nLNS, el cual se presenta como una extensión del trabajo presentado en [Alyd02] a *n* niveles de red con lo cual se eliminan algunas restricciones y además se mejoran los mecanismos de sincronización, también se presenta una aplicación de este modelado en los sistemas de manufactura por lotes. Este trabajo se extendería de nuevo en

[San04], en donde se perfeccionan la función de etiquetado y las reglas de habilitación, logrando así una notación más clara eliminando también algunas restricciones impuestas sobre las transiciones que no requieren sincronización. Así también se modificó la función para asignar pesos a los arcos y se agregó la cardinalidad de una transición, finalmente se utilizó este formalismo para modelar algunos protocolos de interacción entre agentes.

A continuación se presenta la definición del formalismo nLNS [San04] que es el que utilizaremos para modelar nuestros sistemas de Flujo de Trabajo.

2.2.1 El Formalismo nLNS

Un sistema n-LNS está formado por un número finito de niveles de red, donde el nivel 1 es el nivel más alto, y el nivel *n* es el más bajo. El número de niveles de red definido depende del grado de abstracción que se desea obtener en el modelo, y del tamaño y complejidad del sistema a modelar.

A cada nivel de red se le asigna un conjunto de redes. El nivel 1 consiste solamente de una sola red. Los niveles de red se relacionan de la siguiente manera: una red de nivel superior puede tener como marcas en sus lugares redes de nivel inferior, pero no al revés. Una red de nivel *i* consiste de una red tipo y un marcado.

- *Componentes y funcionamiento básico*

Una *red tipo* esta formada por:

- Una estructura de red de Petri ordinaria.
- Un conjunto finito de etiquetas que pueden ser asignadas a las transiciones de la red.
- Un conjunto finito de símbolos y redes tipo que son permitidos en los lugares de la red.
- Un conjunto finito de variables, que pueden ser asociadas a los arcos de la red.
- Una función de etiquetado, que asigna a las transiciones de la red un conjunto de etiquetas y atributos de sincronización.
- Una función de tipos, que asigna a un lugar las redes y/o símbolos que puede contener.
- Una función que asigna un peso a los arcos, respecto al conjunto de etiquetas.

El *marcado* es una función que asocia a cada lugar un multi-conjunto de redes y /o símbolos cuyas redes tipo pertenecen al conjunto de tipos permitidos en dicho lugar.

Ejemplo 1 La Figura 2.2 presenta un fragmento de un sistema de red de 4 niveles. El nivel 1 está formado por la red NET_1; el nivel 2 por las redes $NET_{2,1}$ y $NET_{2,2}$. Las redes $NET_{3,1}$, $NET_{3,2}$ y $NET_{3,3}$ integran el nivel 3, y el nivel 4 se compone de las redes $NET_{4,1}$, $NET_{4,2}$ y $NET_{4,3}$. La red NET_1 tiene como marcado redes de nivel 2 ($NET_{2,1}$, $NET_{2,2}$), de nivel 3 ($NET_{3,3}$), de nivel 4 ($NET_{4,2}$) y símbolos ($s3$, $s4$). La red $NET_{2,1}$ tiene como marcado solamente redes de nivel 3 ($NET_{3,1}$ y $NET_{3,2}$), mientras que $NET_{2,2}$ tiene una red de nivel 4 ($NET_{4,3}$). El marcado de $NET_{3,1}$ está compuesto por una red de nivel 4 ($NET_{4,1}$), y el resto de las redes tienen símbolos como marcas.

Los elementos del sistema interactúan por medio de la sincronización de transiciones. El etiquetado de las transiciones determina las interacciones entre los elementos; dos transiciones que requieren sincronizarse deben tener etiquetas iguales y su conjunto de atributos debe ser diferente de vacío; una etiqueta cuyo conjunto de atributos es vacío no requiere sincronización alguna.

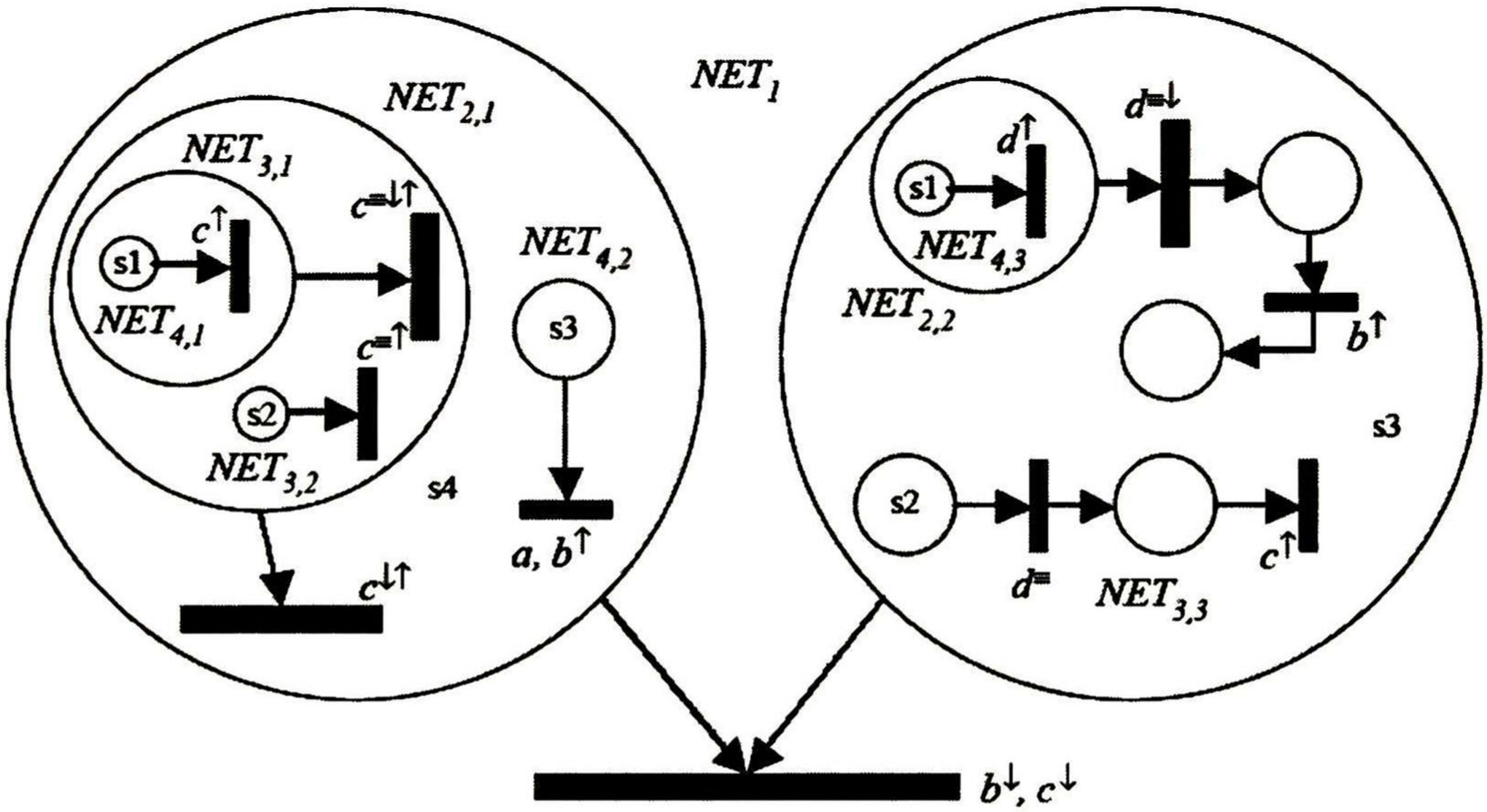


Figura 2.2: Fragmento de un Sistema de Red de 4 Niveles

El etiquetado juega un papel importante en las reglas de habilitación y disparo de las redes. Cuando una transición tiene una etiqueta con el conjunto vacío como su conjunto de atributos, su disparo es autónomo: su habilitación depende únicamente del marcado de sus

lugares de entrada. En caso de que dicho conjunto no sea vacío, la habilitación de esta transición depende no solamente del marcado de sus lugares, sino también de la habilitación de las transiciones de otras redes con la misma etiqueta, que deben dispararse simultáneamente, o *sincronizarse*.

Se definen tres tipos de sincronización: la *sincronización local*, representada por el atributo $\equiv$ que indica que la transición debe sincronizarse con transiciones de redes de nivel superior y/o inferior que se encuentran en el mismo lugar. En la *sincronización interna* ($\downarrow$), una transición debe sincronizarse con transiciones de redes marcando sus lugares de entrada. Finalmente, la *sincronización externa* comprende la sincronización de transiciones de una red con una transición de salida del lugar donde la red está contenida.

Una red de nivel 1 sólo puede utilizar sincronización interna. Una red de nivel intermedio (mayor que 1 y menor que n) puede usar los tres tipos, mientras que una red de nivel n sólo puede tener sincronización local y externa.

- *Definición formal*

Un sistema de red n-LNS consiste de un conjunto de redes de varios niveles. Cada una de estas redes consta de una red tipo con un marcado.

► Redes tipo

Definición 1 Una red tipo (k de nivel i) es una tupla $typenet_{i,k} = (G, TOKEN_{i,k}, LABEL_{i,k}, VAR_{i,k}, \tau, \lambda, \chi, \pi)$ para $1 \leq i \leq n$, donde:

– G es una estructura de RP ordinaria. $G = (P, T, F)$ donde :

- P es un conjunto finito no vacío de lugares
- T es un conjunto finito no vacío de transiciones
- F es una relación de flujo $P \times T \cup T \times P$, tal que $P \cap T = \varnothing$

– $TOKEN_{i,k}$ es un conjunto finito no vacío de redes tipo y símbolos permitidos dentro de los lugares de una red tipo k de nivel i:

$TOKEN_{i,k} \subseteq \{typenet_{j,k} \mid i < j \leq n, 1 \leq k \leq r\} \cup SYMBOLS$ donde

- n es el número de niveles de un sistema de red multi-nivel

- r es el número de redes tipo diferentes permitidas dentro de los lugares de una red tipo de nivel i
- *SYMBOLS* es un conjunto finito de símbolos.

Así, $TOKEN_{1,k} \subseteq \{typenet_{2,1}, \ldots, typenet_{n,k}\} \cup SYMBOLS$

$TOKEN_{2,\ldots k} \subseteq \{typenet_{3,1}, \ldots, typenet_{n,k}\} \cup SYMBOLS$

$\vdots$

$TOKEN_{n,k} \subseteq SYMBOLS$

- $LABEL_{i,k}$ es un conjunto finito de etiquetas definidas para la red tipo k de nivel i.
- $\tau : P \rightarrow 2^{TOKENi,k} - \varnothing$ es una función de asignación de redes tipo y símbolos a los lugares.
- $VAR_{i,k} = \{x, y, \ldots\}$ es un conjunto finito de variables definidas para la red tipo k de nivel i.

  Type: $VAR_{i,k} \rightarrow 2^{(TOKENi,k - SYMBOLS)}$ es una función de asignación de tipos a las variables

  $Type(x) = \{typeneti_{,k} \mid typeneti_{,k} \in \tau(p), p \in P_{i,k}\}$ es el conjunto de tipos asociados a la variable x.
- $\lambda : T \rightarrow 2^{LABi,k} - \varnothing$ es una función de asignación de etiquetas a las transiciones, donde:

  $LAB_{i,k} = (LABEL_{i,k} \times ATTRIB_i)$ y
  - Si $i = 1$, entonces $ATTRIB_i = 2^{\{\downarrow\}}$
  - Si $2 \leq i \leq n\text{-}1$ entonces $ATTRIB_i = 2^{\{\equiv, \downarrow, \uparrow\}}$
  - Si $i = n$ entonces $ATTRIB_i = 2^{\{\equiv, \uparrow\}}$

  Attrib: $LAB_{i,k} \rightarrow ATTRIB_i$ es una función que regresa el conjunto de atributos de sincronización de una etiqueta *label*.
- $\chi : \{(t, lab) \mid t \in T \wedge lab \in \lambda(t) \wedge \equiv \in Attrib(lab)\} \rightarrow N \times (N \cup \infty)\}$ es una función que asigna a cada transición respecto a una etiqueta el número de transiciones (expresado como un intervalo) con las que debe sincronizarse de manera local.
- π: $F_{i,k} \times LAB_{i,k} \rightarrow M_{VARi,k} \cup SYMBOLS \cup CONST$ es una función de asignación de pesos para cada arco respecto a las etiquetas de una transición; el peso es un multi-conjunto de variables y/o símbolos y constantes de red. Si $label \notin \lambda(t)$, entonces $\pi((p, t), label) = \pi((t, p), label) = \varnothing$. Además, si $i = n$ entonces $VAR_{i,k} = \varnothing$, y $\pi : F_{n,k} \times LABEL_{n,k} \rightarrow M_{SYMBOLS}$. *CONST* es un conjunto de redes de nivel inferior que pueden estar contenidas en P. Estas redes serán definidas más adelante.

Una red tipo $typenet_{i,k}$ es una estructura de RP ordinaria con información adicional que declara y manipula los datos definidos en $TOKEN_{i,k}$ de acuerdo a las pre- y postcondiciones establecidas por la función π, y al etiquetado simbólico de transiciones especificado por λ, para la interacción entre redes.

De acuerdo a la definición, las redes están organizadas en niveles: un nivel i puede contener una o más redes de diferente tipo. Cada red de tipo k y nivel i puede contener tipos definidos en el conjunto $TOKEN_{i,k}$. La función τ asigna un subconjunto de $TOKEN_{i,k}$ a cada lugar de G, indicando que el lugar puede contener como marcas únicamente elementos de dicho subconjunto (símbolos y/o redes tipo).

La función λ asigna a cada transición un conjunto de tuplas de la forma (*label*, *attrib*), donde *label* es un elemento de $LABEL_{i,k}$, y *attrib* es un subconjunto de $ATTRIB_i$, que representa los tipos de sincronización asociados a la etiqueta *label*. Por ejemplo, la tupla $(l,\varnothing)$ indica que la etiqueta l no requiere sincronización alguna (disparo autónomo), mientras que $(l,\{\uparrow\})$ denota que l requiere sincronización *externa*; $(l,\{\downarrow\})$ es para sincronización *interna*, y $(l,\{\equiv\})$ indica sincronización local. Los demás subconjuntos de *ATTRIB* representan combinaciones de tipos de sincronización. Así, la tupla $(l,\{\equiv, \downarrow, \uparrow\})$ indica que l requiere sincronización *local*, *interna* y *externa*. Por simplicidad, escribiremos de aquí en adelante $l^{\equiv\downarrow\uparrow}$ en lugar de utilizar la notación anterior, y cuando *attrib* es vacío, escribimos solo l.

Los elementos del conjunto $ATTRIB_i$ varían de acuerdo al nivel al que pertenece G. Si la red es de nivel 1, entonces sus transiciones sólo pueden sincronizarse internamente, por lo que $ATTRIB_i$ es igual al conjunto potencia de $\{\downarrow\}$; si la red pertenece al nivel n, entonces está permitido la sincronización local y externa de sus transiciones, por lo que $ATTRIB_i$ es igual al conjunto potencia de $\{\uparrow, \equiv\}$. Si la red es de un nivel intermedio (mayor que 1 y menor que n), entonces se permite los tres tipos de sincronizaciones, y $ATTRIB_i$ es el conjunto potencia de $\{\downarrow, \uparrow, \equiv\}$.

La función χ determina la cardinalidad de una transición t, esto es, el número de redes con las que t se debe sincronizar de manera local. χ asigna a cada transición, respecto a una etiqueta que tiene que sincronizar localmente, una pareja de enteros positivos; el primero representa el número mínimo de transiciones, y el segundo el máximo. El símbolo ∞ representa un número muy grande (infinito). Utilizaremos la siguiente notación: escribiremos [x, y] para

representar la pareja (x, y) y [x] para representar a (x, x); por convención, si en el modelo no se indica lo contrario, $\chi(t, lab) = (1, \infty)$ para toda transición t que deba sincronizarse localmente.

La función π establece las precondiciones y las poscondiciones de cada transición de la red. Asigna un peso a cada arco con respecto a una etiqueta. Determina la cantidad y el tipo de redes y/o símbolos que se necesitan dentro de los lugares de entrada para habilitar una transición, y la cantidad y el tipo de redes y/o símbolos que deben ser agregados a los lugares de salida. Cuando la red tipo es de nivel n, la función π sólo asigna un multiconjunto de símbolos al peso de los arcos, por lo tanto, la red tipo es similar a una Red de Petri Coloreada.

Una red tipo no tiene restricción alguna sobre el peso de los arcos. Es posible especificar en el peso de los arcos de salida de una transición, símbolos y/o variables del tipo no incluido en los lugares de entrada a dicha transición, e inclusive omitir uno o varios de ellos. De esta manera, una transición puede:

- Eliminar redes y/o símbolos, como resultado de consumirlos y no agregarlos a un lugar de salida.
- Crear redes y/o símbolos en un lugar de salida cuando no han sido removidos de sus lugares de entrada , como resultado del disparo de la transición.
- Clonar redes y/o símbolos que se obtienen de removerlos y sumar varias copias de estos.

El siguiente ejemplo muestra algunas redes tipo.

Ejemplo 2 En la Figura 2.3 se muestran dos redes tipo de nivel 4: $typenet_{4,1}$ y $typenet_{4,2}$. Para la primera red, sus elementos están definidos de la siguiente forma:

- $TOKEN_{4,1} = \{s1, s2, s3\}$, $LABEL_{4,1} = \{b, c, m\}$, $VAR_{4,1} = \varnothing$
- $\tau(p_1) = \tau(p_2) = \{s1\}$, $\tau(p_3) = \tau(p_4) = \{s2\}$, $\tau(p_5) = \{s3\}$
- $\lambda(t_1) = \{b^{\uparrow}\}$, $\lambda(t_2) = \{m\}$, $\lambda(t_3) = \{c^{\equiv\uparrow}\}$
- $\pi((p_1, t_1), b^{\uparrow}) = \pi((t_1, p_2), b^{\uparrow}) = \pi((p_2, t_2), m) = s1$, $\pi((t_1, p_4), b^{\uparrow}) = \pi((p_4, t_3), c^{\equiv\uparrow}) = \pi((t_2, p_3), m) = \pi((p_3, t_3), c^{\equiv\uparrow}) = s2$, $\pi((t_3, p_5), c^{\equiv\uparrow}) = s3$
- $\chi(t_3, c^{\equiv\uparrow}) = (1, \infty)$.

Para $typenet_{4,2}$, tenemos :

- $TOKEN_{4,2} = \{s1\}$, $LABEL_{4,2} = \{n, d\}$, $VAR_{4,2} = \varnothing$
- $\tau(p_1) = \tau(p_2) = \{s1\}$
- $\lambda(t_1) = \{n\}$, $\lambda(t_2) = \{d^{\uparrow}\}$

– $\pi((p_1, t_1), n) = s1$, $\pi((t_1, p_2), n) = \pi((p_2, t_2), d\uparrow) = 2s1$.

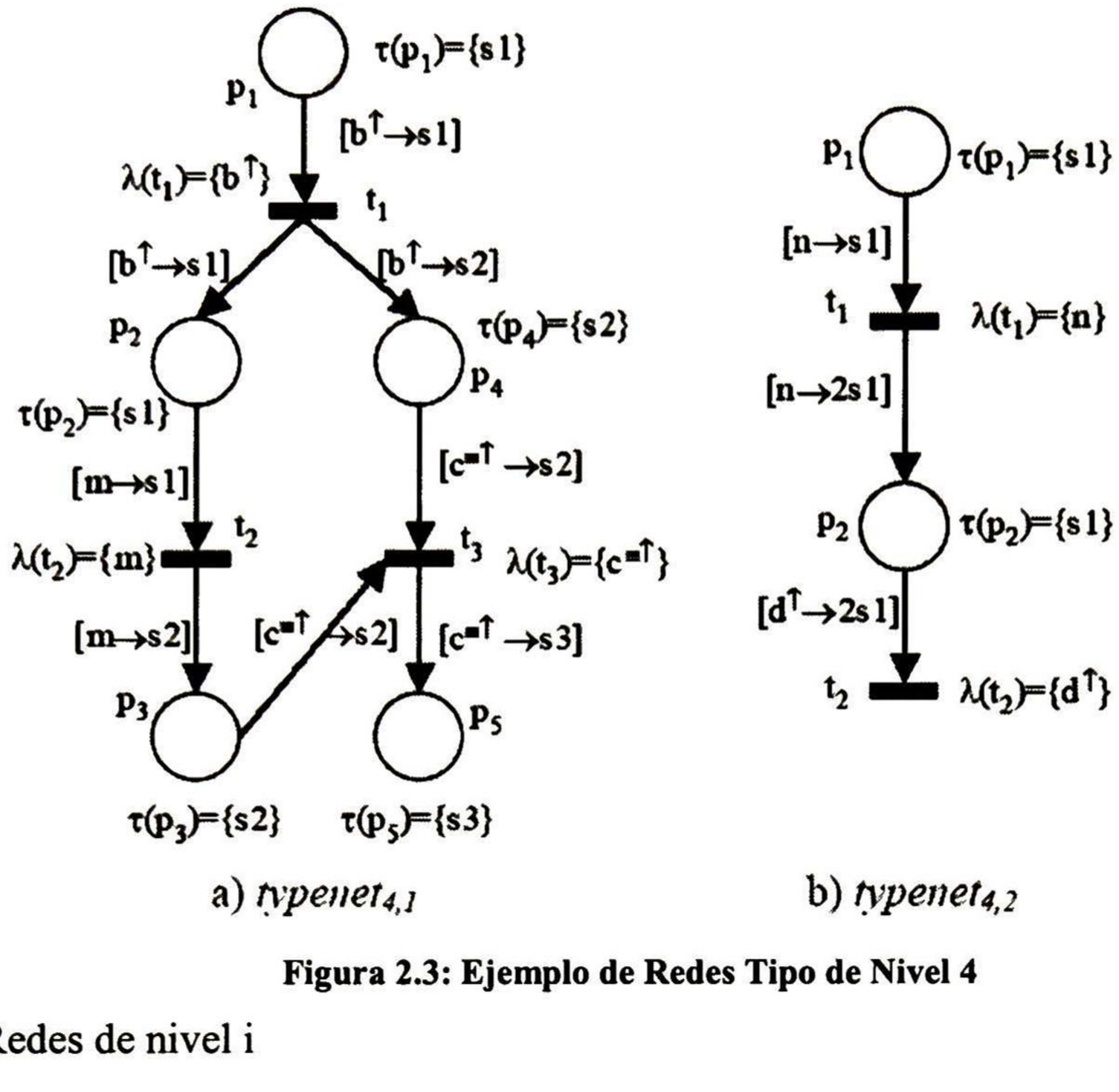


Figura 2.3: Ejemplo de Redes Tipo de Nivel 4

- Redes de nivel i

Las redes de nivel *i* están formadas por una red tipo y un marcado inicial.

Definición 2 Una red *k* de nivel *i* es una tupla $NET_{i,k} = (typenet_{i,k}, \mu_{i,k}) \mid 1 \leq i \leq n, k = 1, \ldots r$, donde:

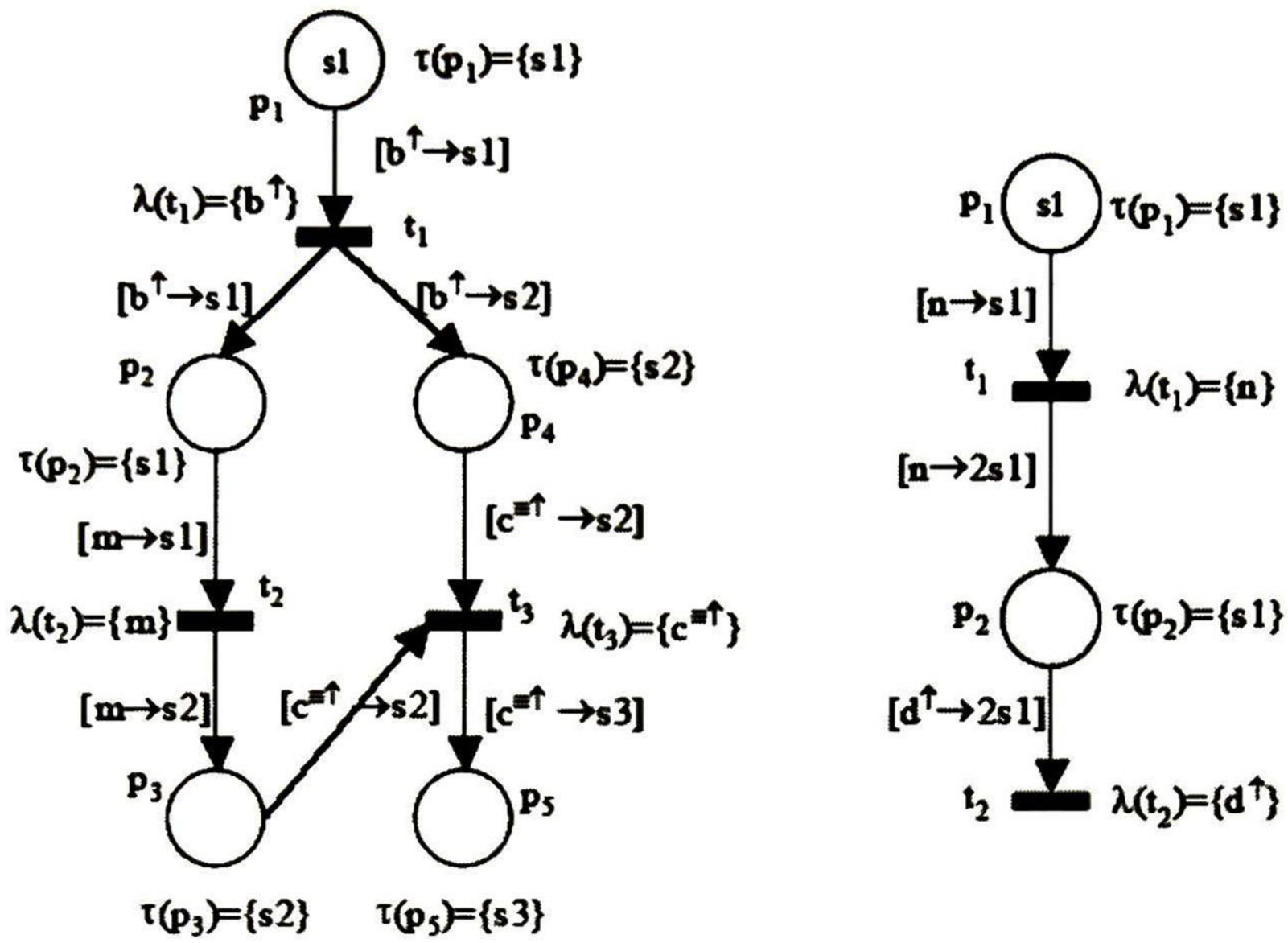
- $typenet_{i,k}$ es una red tipo *k* de nivel *i*.
- $\mu_{i,k} : P_{i,k} \rightarrow M_{NETS_{TOKENI,K} \cup SYMBOLS}$ es una función de marcado para la red *k* de nivel *i*.

$NETS_{TOKENi,k} \subseteq \{NET_{i+1,k}, NET_{i+2,k}, \quad , NET_{n,k}\}$ es el conjunto de todas o algunas redes de nivel inferior que están como marcas de la red $NET_{i,k}$.

La función de marcado $\mu_{i,k}$ le asigna a cada lugar de G un multi-conjunto de redes y de símbolos, cuyas redes tipos y símbolos se encuentran en el conjunto de tipos y símbolos asignados al lugar por la función τ.

El tipo de una red se determina mediante la función $TypeN(NET_{i,k}) = typenet_{i,k}$.

Ejemplo 3 La Figura 2.4 muestra dos redes de nivel 4, cuyas redes tipo son las mostradas en la Figura 2.3, y que se definen de la siguiente forma: $NET_{4,1} = (typenet_{4,1}, \mu_{4,1})$ y $NET_{4,2} = (typenet_{4,2}, \mu_{4,2})$, donde $\mu_{4,1}(p1) = \mu_{4,2}(p1) = \{s1\}$.



a) $NET_{4,1}$ b) $NET_{4,2}$

Figura 2.4: Ejemplo de Redes de nivel 4

- Sistema n-LNS

Un sistema n-LNS, o un sistema de red, es una colección de una o más redes instanciadas a partir de cada una de las redes tipo definidas en todos los niveles.

Definición 3 Un sistema de red a n niveles es una tupla $n-LNS = (NET_{i,k}, | \exists\, i = 1..n : n \in N, k = 1,.., r)$, donde:

- $NET_{1,1}$ es una red de nivel 1. Representa el nivel más alto del sistema de red.
- $NET_{i,k} = \{NET_{2,1}, \ldots, NET_{i,r}\}$ es el conjunto de r redes de nivel i. Representa las redes de nivel intermedio y nivel más bajo.

- Evolución del marcado.

Mientras que la parte estática de una red está representada por su estructura, la parte dinámica está caracterizada por la evolución de su marcado, mediante las reglas de habilitación y disparo de sus transiciones.

Antes de definir los estas reglas, debemos introducir algunos conceptos.

Definición 4 Una función de ligado *b* sobre un conjunto de variables *VARS* = {x, y, ...} es una función *b*: $VARS \rightarrow NETS_{TOKENi,k}$; para una variable $v \in VARS$, b(v) es una red de nivel inferior cuyo tipo pertenece a *Type*(v). $m<b>$ denota el multi-conjunto de redes que resulta de evaluar un multi-conjunto de variables *m* en un ligado *b*.

Definición 5 $b_t : \left\{ v \middle| v \in \bigcup_{p \in {}^*t} VAR\left(E_{\pi((p,t),label)}\right) \right\} \rightarrow \bigcup_{p \in {}^*t} NETS_{TOKEN_I}(p)$ es una función que asigna a cada variable definida en el peso de los arcos de entrada a la transición *t*, con respecto a la etiqueta *label*, redes del tipo asignado al conjunto de lugares de entrada a *t*, donde $NETS_{TOKENi(p)} = \{net \mid net \in NETS_{TOKEN_i} \wedge TypeN(net) \in \tau(p)\}$, $E_{\pi((p,t),label)}$ denota los elementos sin repetición del multi-conjunto $\pi((p, t), label)$ y VAR($E_{\pi((p,t),label)}$) el conjunto de variables en $E_{\pi((p,t),label)}$.

- Regla de habilitación.

Definición 6 Una transición *t* de una red *k* de nivel *i* $NET_{i,k}$ está habilitada con respecto a una etiqueta $lab \in \lambda(t)$ si se verifican las siguientes condiciones:

1. Existe un ligado b_t si $VAR_t \neq \varnothing$; si $VAR_t = \varnothing$, como en el caso de las redes de nivel *n* o cuando en el peso de los arcos incluye sólo símbolos y constantes de red, entonces el ligado es omitido; VAR_t es el conjunto de variables que aparecen en todos los arcos de entrada a *t*.
2. $\forall p \in {}^\bullet t$, $\pi((p, t), lab) <b_t> \subseteq \mu_{i,k}(p)$ si $VAR_t \neq \varnothing$, ó $\forall p \in {}^\bullet t$, $\pi((p, t), lab) \subseteq \mu_{i,k}(p)$ si $VAR_t = \varnothing$; en cada lugar de entrada a *t* hay tantas redes de nivel inferior y símbolos como lo especifica la función π evaluada en el ligado b_t. El ligado $<b_t>$ es omitido cuando el peso de los arcos no contiene variables.
3. Las condiciones de uno de los siguientes casos se cumplen:

Caso 1: Si lab = (l, ∅), el disparo de t no requiere sincronización, se realiza de manera autónoma.

Caso 2: Si *lab* ≠ (l, ∅) se debe considerar una o una combinación de las siguientes situaciones:

i. *lab* = (l ,{≡}) (Sincronización local). Se requiere la habilitación simultánea de las transiciones etiquetadas con $l^{\equiv}$ pertenecientes a otras redes que se encuentran en el mismo lugar p' de una red de nivel superior. Además, si $\chi(t, lab) = (x, y)$, el número de éstas redes debe ser mayor o igual a x y menor o igual a y. El disparo de estas transiciones es simultáneo, y todas las redes sincronizadas permanecen en p'.

ii. *lab* = (l, {↓}) (Sincronización interna). Se requiere la habilitación simultánea de las transiciones etiquetadas con $l^{\uparrow}$ pertenecientes a otras redes de nivel inferior que se encuentran en $\cdot t$. Estas transiciones se disparan simultáneamente y las redes de nivel inferior y símbolos especificados en $\pi((p, t), lab)\langle b_t\rangle$ son removidos.

iii. *lab* = (l, {↑}) (Sincronización externa). Se requiere la habilitación de al menos una de las transiciones $t' \in p'$ etiquetadas con $l^{\downarrow}$ de la red de nivel superior donde esta contenida la red $NET_{i,k}$. El disparo de t provoca la transferencia de $NET_{i,k}$ y los símbolos declarados en $\pi((p', t'), lab)\langle b_t\rangle$.

Una etiqueta puede tener más de un tipo de sincronización. Por ejemplo, una etiqueta *lab* = (l, {↑, ↓, ≡}), debe sincronizarse de manera local, interna y externa respecto al símbolo l.

Es importante señalar que la sincronización interna no tiene efecto cuando una red de nivel i se sincroniza internamente a través de una transición t, respecto a una misma etiqueta, con una transición t" de una red de nivel $i + 2$, la cual está contenida en una red de nivel $i + 1$, que se encuentra marcando $\bullet t$ de la red de nivel i, y cuyo símbolo no está asociado a la transición de salida t'. Esto mismo ocurre para el caso de sincronización externa.

- Regla de disparo.

La cantidad de redes y/o símbolos que una transición produce o consume al dispararse se indica en el peso de los arcos de entrada y salida a la transición.

Definición 7 El marcado de la red que se obtiene al disparar una transición t de una red k de nivel i $NET_{i,k}$ está dado por las siguientes expresiones:

$\forall p \in \bullet t \cup t\bullet$, $\mu'_{i,k}(p) = \mu_{i,k}(p) - \pi((p, t), label) \cup \pi((t, p), label)$ si $VAR_t = \varnothing$

$\forall p \in \bullet t \cup t\bullet$, $\mu'_{i,k}(p) = \mu_{i,k}(p) - \pi((p, t), label)\langle b_t \rangle \cup \pi((t, p), label)\langle b_t \rangle$ si $VAR_t \neq \varnothing$ $\wedge \downarrow \notin Attrib(label)$.

$\forall p \in \bullet t \cup t\bullet$, $\mu'_{i,k}(p) = \mu_{i,k}(p) - \pi((p, t), label)\langle b_t \rangle \cup \pi((t, p), label)\langle b_t\rightarrow \rangle$ si $VAR_t \neq \varnothing \wedge \downarrow \in Attrib(label)$.

donde:

- $\mu'_{i,k}(p)$ es el marcado obtenido después de disparar la transición t.
- $\mu_{i,k}(p)$ es el marcado actual antes de disparar t.
- b_t es el ligado que habilita a t.
- $b_t\rightarrow$ es un ligado con el dominio igual a b_t y cuyo rango son las redes en el rango de b_t pero con su marcado alcanzado al disparar sus transiciones que deben sincronizarse con t.
- VAR_t es el conjunto de variables que aparecen en todos los arcos de entrada a t.

Si t requiere algún tipo de sincronización, deben dispararse de manera simultánea junto con t todas las transiciones que estén habilitadas respecto a *label*.

El sistema de 4 niveles $4\text{-}LNS = \{NET_{1,1}, NET_{2,1}, NET_{3,1}, NET_{3,2}, NET_{4,1}, NET_{4,2}\}$ mostrado en la Figura 2.5 (la red $NET_{4,2}$ no es mostrada en la figura, debido a que no ha sido creada en ese momento) evoluciona de acuerdo a las reglas de habilitación y disparo antes mencionadas. A partir del marcado inicial de las redes, la transición t_1 de $NET_{1,1}$ está habilitada respecto a la etiqueta $a\downarrow$, y se sincroniza con la red que marca su lugar de entrada, $NET_{2,1}$ la cual tiene habilitada su transición t_1, que se sincroniza externamente con $NET_{1,1}$ e internamente con $NET_{3,1}$ y $NET_{3,2}$ respecto a $a\downarrow\uparrow$ Además, en $NET_{1,1}$ está habilitada la transición t_2 respecto a $b\downarrow$, sincronizándose con $NET_{4,1}$, cuya transición etiquetada con $b\uparrow$ está habilitada. En la Figura 2.6 podemos observar el resultado del disparo de estas transiciones, incluyendo el "movimiento" de redes, como en el caso de t_1 y t_2 de $NET_{1,1}$ y t_1 de $NET_{2,1}$,y la "creación" de redes con el disparo de t_1 en $NET_{3,1}$.

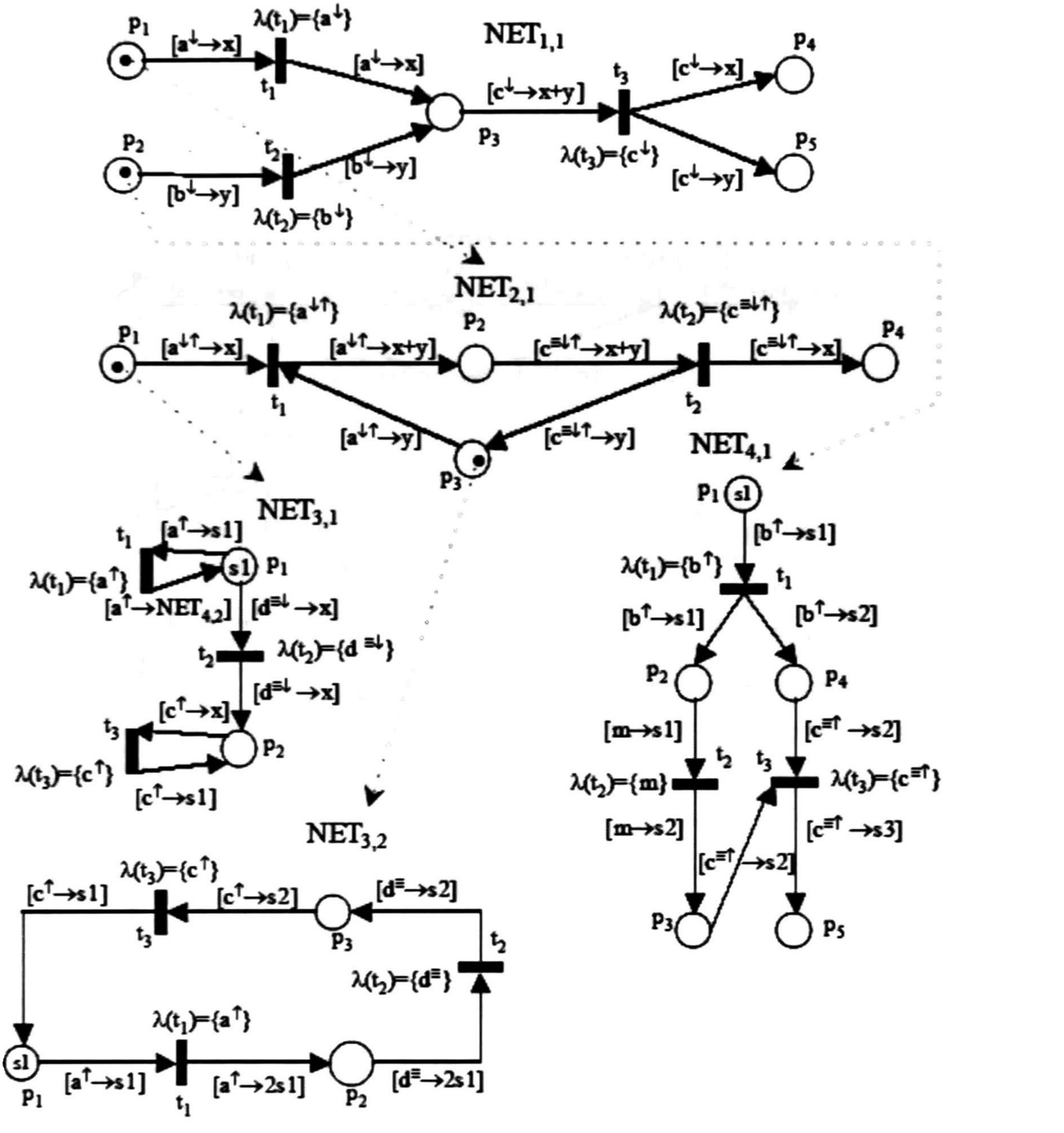


Figura 2.5: Sistema de red de 4 niveles

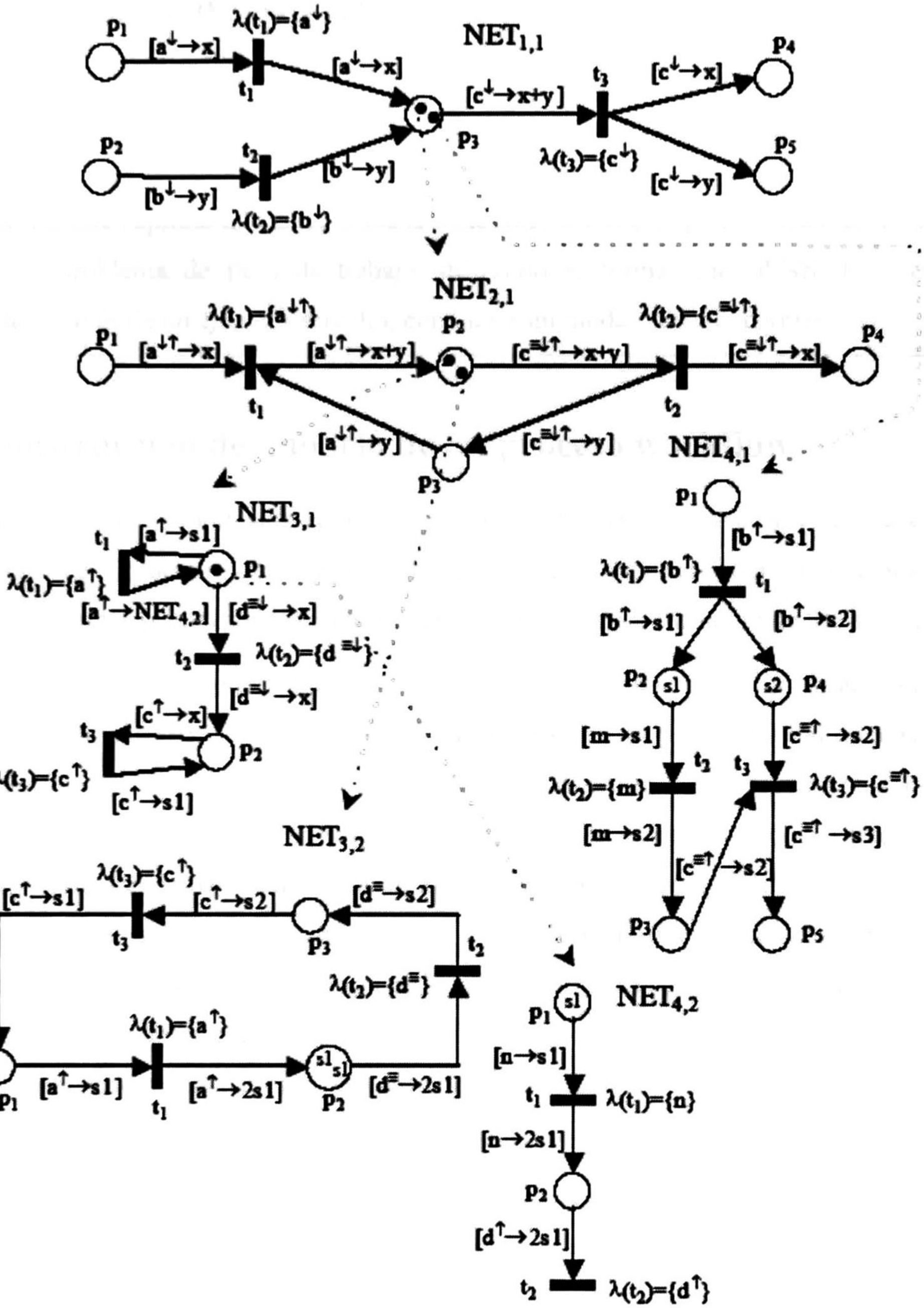


Figura 2.6: Evolución del marcado

CAPITULO III.

Metodología de Modelado de procesos WorkFlow

Resumen. En este capítulo introduciremos la metodología a seguir para obtener un modelo multi-nivel de un problema de flujo de trabajo utilizando el formalismo nLNS. La metodología, presentada a través de un ejemplo sencillo, conduce a un modelo de tres niveles.

3.1 Construcción del modelo de un proceso workflow

La estrategia de modelado consiste en definir los diferentes niveles de abstracción en las componentes del problema y construir las diferentes redes que describen el funcionamiento de dichas componentes para después relacionarlas mediante la sincronización de transiciones.

Considerando un modelo a tres niveles podemos enunciar sin mucho detalle por ahora, la estrategia para la construcción de un modelo en 3LNS (ver figura 3.1). La red de más alto nivel (1) describirá las diferentes unidades físicas o bien organizacionales (departamentos, oficinas, ..) en las que interesa especificar las actividades a realizar y la manera en la que estas unidades se relacionan y fluye el trabajo. Las redes de nivel 2 modelan el comportamiento general de una entidad que guía el caso. En las redes de nivel 3 se describen, por un lado, las operaciones que involucran cada tarea, y por otro lado la definición del proceso de flujo de trabajo, es decir, el plan a seguir enunciando qué tareas deben realizarse y en qué orden. La interacción entre los diferentes modelos se verifica para las transiciones previstas durante la creación de las redes o bien se detecta en el momento de hacer la integración y se modifican los modelos construidos.

A continuación detallaremos las etapas de modelado a través del siguiente ejemplo. *Considere un proceso de flujo de trabajo en el cual se realiza la compra de un producto; inicialmente un comprador se dirige a un vendedor quien le proporciona información general sobre el producto, se acuerda la compra y se pasa al proceso de facturación, posteriormente se procede a la entrega de dicho producto en el almacén.*

Paso 1. *Identificar los departamentos involucrados en el proceso y el sentido del flujo de la información entre ellos.*

a) Identificación de los departamentos involucrados

Para lograr esto se tiene que revisar la definición del problema y de ahí obtener toda la información posible involucrada con este punto. Si la información no es suficiente para englobar los departamentos involucrados, se deberá recurrir al organigrama de la empresa e identificar la parte del mismo donde fluirá la información del problema descrito.

Muchas veces, la información requerida, no se obtendrá completamente de la definición del problema, por lo que se deberá realizar una serie de entrevistas con el personal adecuado de la compañía para la cual se está diseñando el modelo. Para ello es necesario recabar la información posible acerca del flujo de tareas entre los distintos departamentos; para esto un organigrama detallado sirve de mucho ya que se pueden resaltar rápidamente los nombres de los departamentos involucrados y la dirección del flujo de tareas desde el inicio del caso hasta su terminación.

En este punto es importante fijar nuestra atención únicamente en la estructura organizacional en donde el flujo de trabajo toma parte y no tanto en las tareas involucradas; se trata de tener un modelo general de la estructura de la organización para ir siguiendo el flujo del trabajo entre los departamentos, definiendo así el departamento (oficina, unidad organizacional) en donde el caso inicia (o nace) hasta identificar el departamento en el que el caso llega a su completa terminación (o muere). El nivel de detalle de esta información deberá ser tan fina como se necesite para el modelo.

Supongamos que ya hemos identificado 3 departamentos involucrados en el problema de flujo de trabajo (ver figura 3.2).

DEPARTAMENTO 1 **DEPARTAMENTO 2** **DEPARTAMENTO 3**

Figura 3.2: Identificación general de Departamentos

Pero cuando el caso se encuentre en el Departamento 2 se desea saber cuándo el caso esta en el subdepartamento 1 o cuándo está en el subdepartamento 2 de dicho departamento, en un momento dado del flujo de trabajo. Por lo tanto, la figura 3.2 se puede transformar como se indica en la figura 3.3.

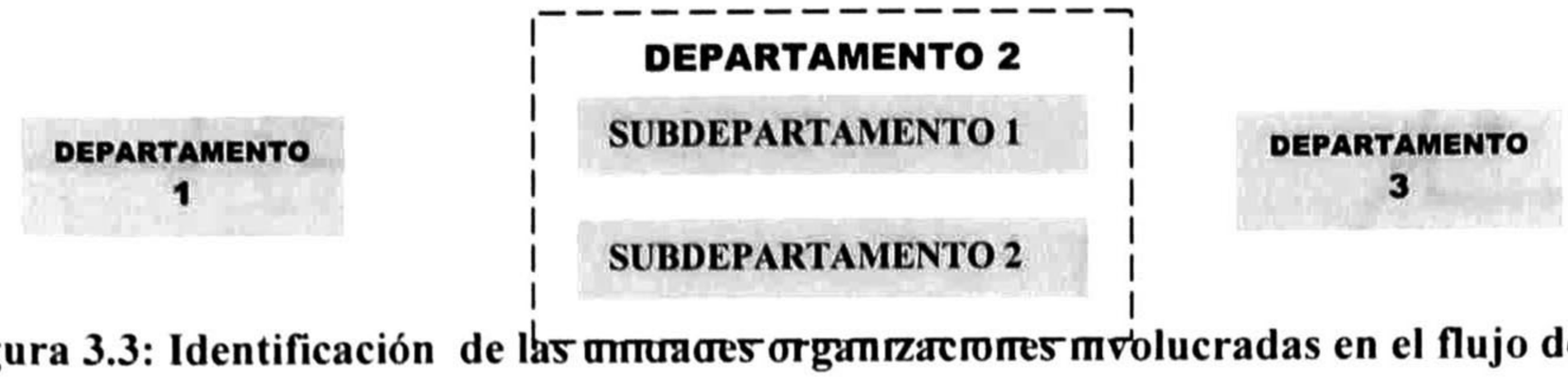


Figura 3.3: Identificación de las unidades organizacionales involucradas en el flujo de trabajo

Para el problema de flujo de trabajo de nuestro Ejemplo 4 observamos que se involucran principalmente 3 departamentos (ver figura 3.4):

- Ventas: A donde el comprador o cliente llega para solicitar información acerca de un producto y al cual le solicita (si es el caso) su adquisición.
- Facturación: En donde se realiza el cobro del producto al cliente, se establece el tipo de pago, se calcula el monto y se extiende la factura.
- Almacén: Lugar en donde se reclama el producto pagado por el comprador.

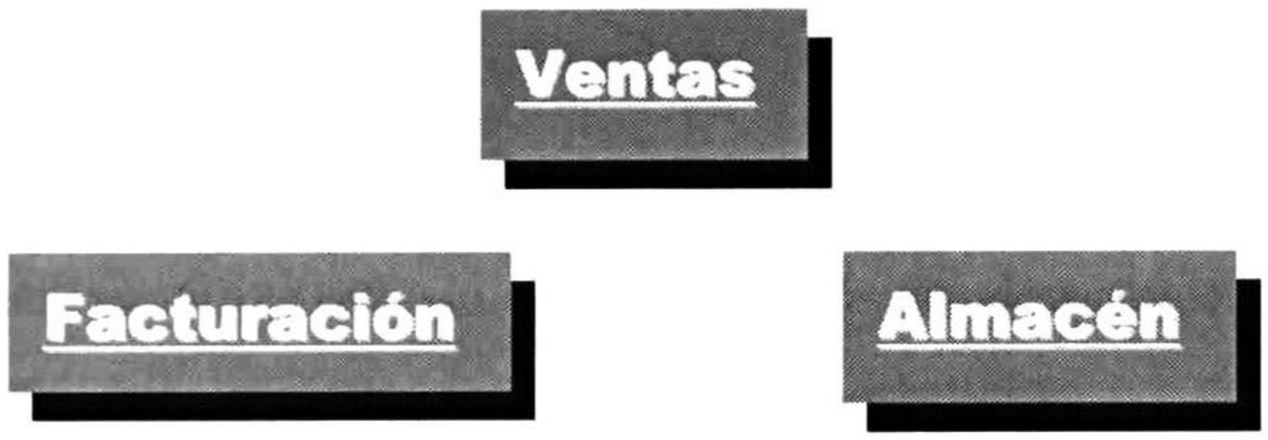


Figura 3.4: Departamentos involucrados en el problema de flujo de trabajo del ejemplo 4

b) Identificación de la dirección del flujo del caso entre los departamentos

Para la identificación de la dirección del flujo de la información entre departamentos debemos introducir primero algunas estructuras generales de unión las cuales se encuentran representadas en la figura 3.5 y se describen a continuación.

Flujo sencillo: se utiliza cuando después de la realización de las tareas correspondientes al caso en el departamento A, el caso se pasa al departamento B para realizar otra serie de tareas.

OR-Split: se utiliza cuando después de la realización de las tareas correspondientes al caso en el departamento A, el caso se pasa al departamento B ó bien al departamento C, según sea el resultado de las tareas en el departamento A. El departamento B y el departamento C, para este caso, son mutuamente excluyentes.

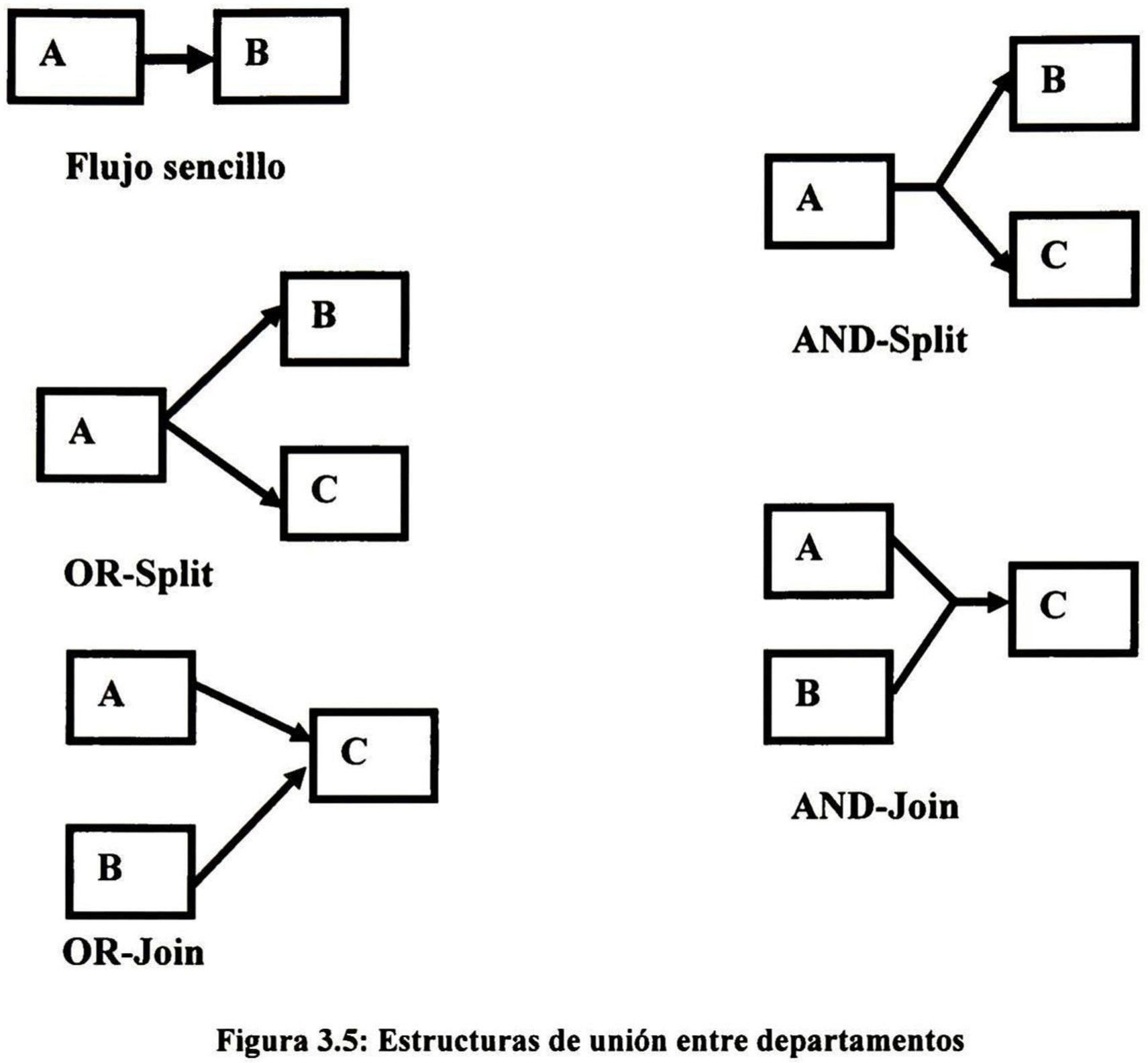


Figura 3.5: Estructuras de unión entre departamentos

AND-Split: se utiliza cuando el flujo después del departamento A, se bifurca en los departamentos B y C para que estos realicen tareas en paralelo.

OR-Join: esta estructura comúnmente se usa después de realizar la estructura OR-Split y significa que: 1) después de la realización de las tareas en el departamento A, el flujo sigue en el departamento C, ó 2) después de la realización de las tareas en el departamento B, el flujo sigue en el departamento C. En esta estructura se supone que solo se tiene un caso que fluye en A o bien en B, es decir, A y B son excluyentes.

AND-Join: esta estructura se utiliza después de haber utilizado un AND-Split y sirve para unir el flujo de información que viene desde dos departamentos distintos, en un solo flujo.

Para identificar el sentido del flujo de la información entre los departamentos se proponen algunas estrategias:

- Identificar el origen del flujo de trabajo, es decir, el departamento en donde se comienza el caso.

Para el ejemplo 4, de la definición del problema, tenemos que el departamento de origen del flujo sería el departamento de *Ventas.*

- Ya que se realizo el paso anterior, identificar a qué o a cuales departamentos se pasa la información del caso,
  - Si solamente le sigue un departamento, entonces se debe unir el departamento actual con el departamento siguiente utilizando la estructura de unión *flujo sencillo.*
  - Si la información puede fluir a un departamento A en un caso, o bien a otro departamento B en otro caso, entonces se unirá el departamento actual con el departamento A utilizando la estructura de unión *flujo sencillo* y también se unirá el departamento actual con el departamento B utilizando la misma estructura. Es importante resaltar que lo que logramos combinando estas uniones con *flujo sencillo* que comparten el mismo origen, resulta la estructura de unión *OR-Split,* por lo que se puede generalizar esta estructura a un OR-Split de dos o más salidas, es decir, un origen y múltiples salidas mutuamente excluyentes recordando que después se deberán unir los flujos con un OR-Join de tantas entradas como salidas requeridas en el OR-Split.
  - Si la información después del departamento actual, fluye tanto a un departamento A como a un departamento B, generando así los flujos en paralelo, se deberá utilizar la estructura de unión *AND-Split.* De la misma forma que hicimos con el *OR-Split*, podemos generalizar el *AND-Split* como bifurcaciones en paralelo que comparten el mismo origen, es decir un *AND-Split* de dos o más salidas, tantas como sea requerido. Recordemos que después de un *AND-Split* deberá utilizarse un *AND-Join* con mismo número de entradas que salidas tenga el *AND-Split*..

Para el ejemplo 4, tenemos que del departamento de *Ventas* se sigue el departamento de *Facturación* por lo que se unirán estos dos con un *flujo sencillo*:

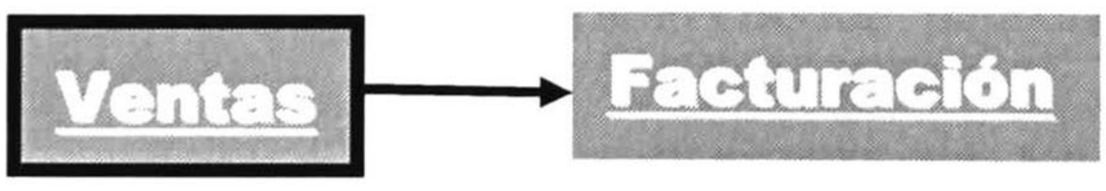


Figura 3.6: Unión de dos departamentos del ejemplo 4 con *flujo sencillo*

- De cada uno de los departamentos expandidos en el paso anterior, identificar su o sus departamentos subsecuentes y así sucesivamente hasta abarcar todos los departamentos involucrados.

  Para el ejemplo 4 tenemos que del único departamento expandido (*Facturación*), el flujo sigue hacia el departamento de *Almacén* (ver figura 3.7).

Figura 3.7: Expansión del flujo para el ejemplo 4

- Revisar el flujo y sentido del mismo con el objetivo de identificar si se emplearon las estructuras correctas.
  - Si se tiene un departamento con múltiples entradas, revisar si el flujo proveniente de dichas entradas es solo uno en un caso, es decir, son excluyentes ó si el flujo proviene paralelamente de todas las entradas.
    - Si son flujos excluyentes, revisar que se hayan utilizado solamente *flujos sencillos*, es decir, un *OR-Join* de múltiples entradas.
    - Si son flujos en paralelo, revisar que se haya utilizado la estructura de *AND-Join* de lo contrario reemplazar la estructura (estructuras) de unión utilizadas por un *AND-Join* de tantas entradas como así se requiera.

En el ejemplo 4, para la unión del flujo entre los departamentos solo se utilizaron *flujos sencillos* y como se mantiene un solo flujo, NO es necesaria la corrección de estructuras de unión.

Con la combinación de estas estructuras de unión se pueden formar bifurcaciones complejas, por ejemplo:

1. OR-Split sucesivos para dividir el flujo y un solo OR-Join para unir el flujo, o viceversa.

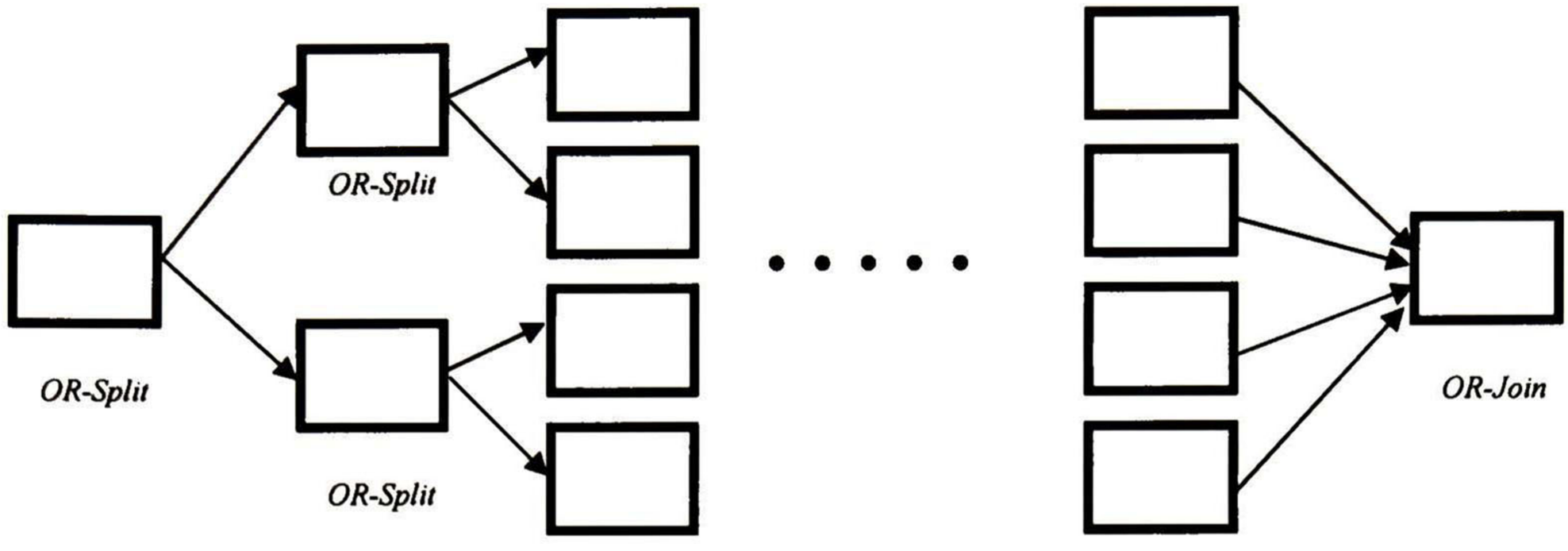


Figura 3.8: Estructuras OR-Split sucesivas unidas en un solo OR-Join

2. AND-Split sucesivos para bifurcar paralelamente el flujo y un solo AND-Join para unir el flujo, o viceversa.

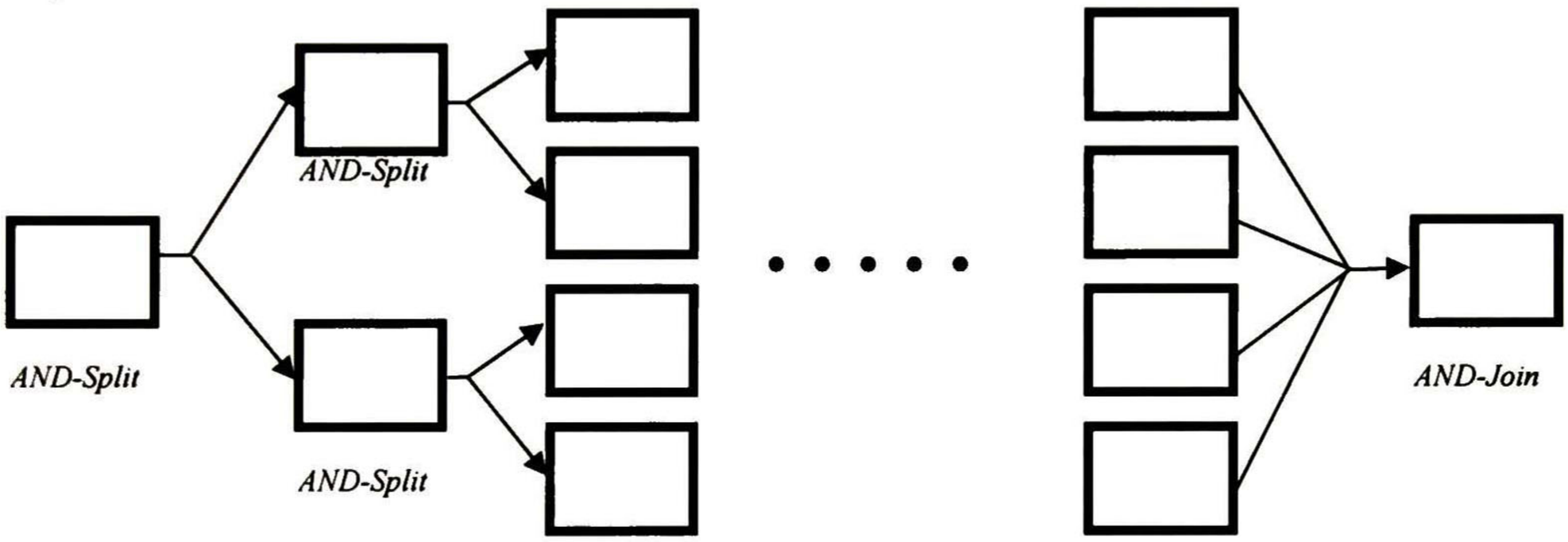


Figura 3.9: Estucturas AND-Split sucesivas unidas en un solo AND-Join

También podría darse una combinación de las anteriores, por lo que lo importante es conservar la siguiente regla:

"El caso deberá tener un solo lugar (departamento) de inicio, así como un solo lugar de terminación"

Si se cumple esta regla podremos decir que tenemos un flujo bien definido.

Una vez identificados los departamentos y haberlos unidos con las estructuras de unión de flujo, procederemos a la construcción de nuestra Red de Petri del Primer nivel.

Paso 2. *Construcción de la Red de Petri del primer nivel, a partir del modelo obtenido en el paso anterior.*

La construcción de la Red de Petri del primer nivel o del nivel 1, a partir del modelo anterior, se construye de una manera muy sencilla:

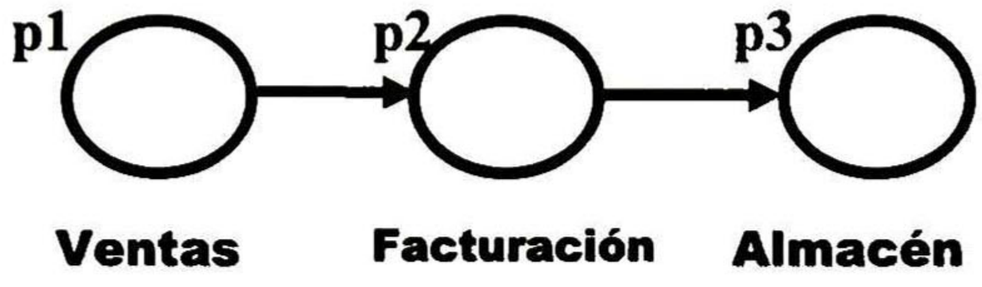
- Definir la Red tipo de nivel 1 como:

o $Net_{1,1}$ =(G, $TOKEN_{1,1}$,$LABEL_{1,1}$,$VAR_{1,1}$, τ, λ, χ, π)

o Construcción del grafo G= (P,T,F)

Lugares de la Red: Del modelo obtenido en apartado número 1, reemplazar cada departamento por un lugar de una Red de Petri y numerar dichos lugares. Al modelo resultante llamarlo *g1*.

Tomando el esquema que obtuvimos en el apartado 1 del ejemplo 4, tenemos el grafo de la figura 3.10.



g1

Figura 3.10: Obtención de *g1* del ejemplo 4

Por lo tanto se tiene que cada departamento de la estructura organizacional identificada se modelará con un lugar de una Red de Petri, por lo que el conjunto P de lugares para el ejemplo 4 sería: ***P = { p1, p2, p3 }***

Transiciones de la Red: De las estructuras de unión en *g1*, realizar las siguientes sustituciones expresadas en la figura 3.11.

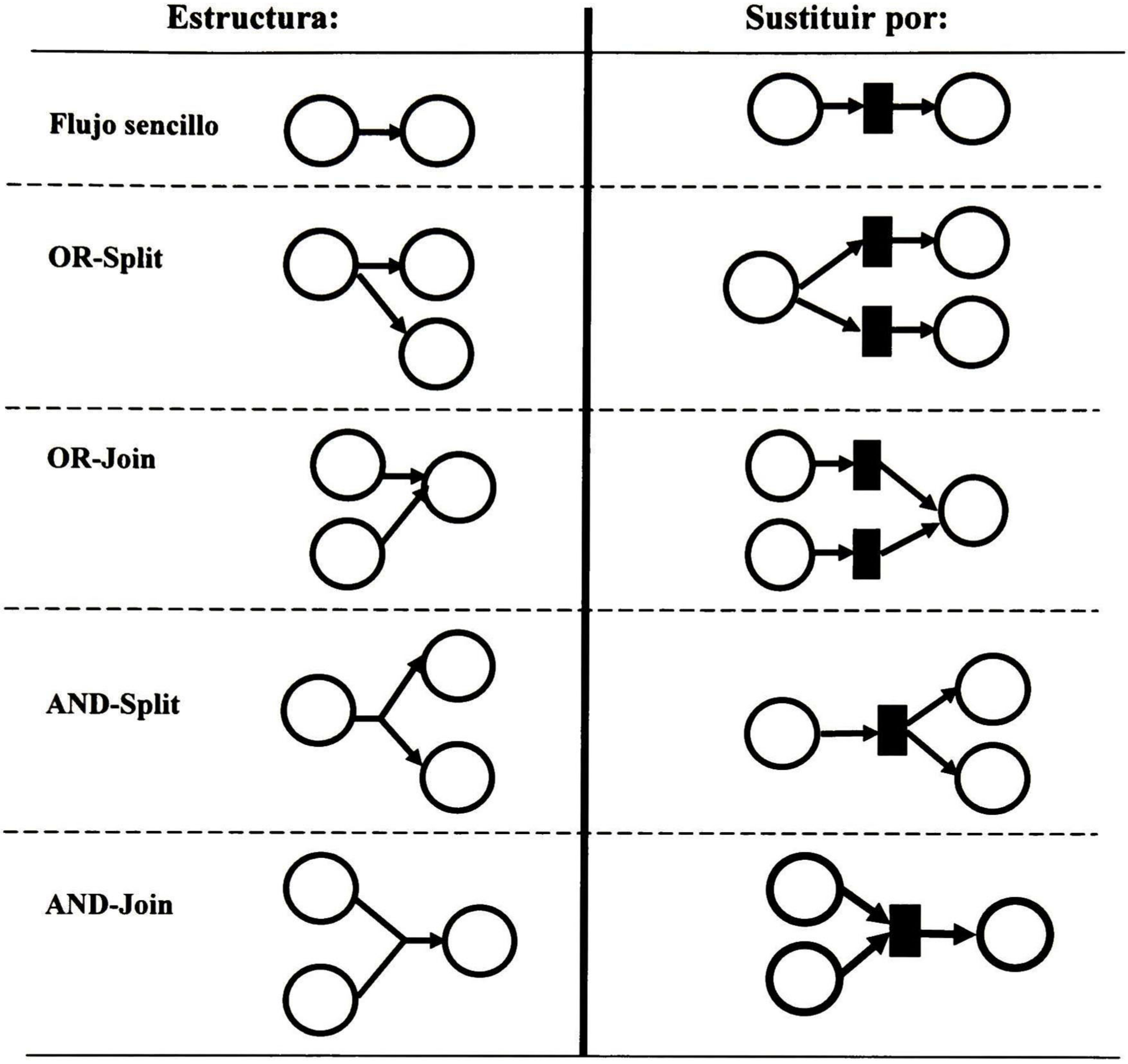


Figura 3.11: Definición de sustituciones para la transformación de estructuras de unión por transiciones de una red de petri

De las sustituciones descritas, cada barra (▮) representa una transición en una Red de Petri, por lo que, al realizar cada una de las sustituciones a *g1*, se deben numerar dichas transiciones. Esto significa que cada transición representa el cambio o flujo de la información de un lugar (departamento) a otro (departamento) en la Red de Petri.

Tomando el *g1* que obtuvimos del ejemplo 4, tenemos que solo debemos reemplazar los flujos sencillos por la estructura correspondiente (ver figura 3.12).

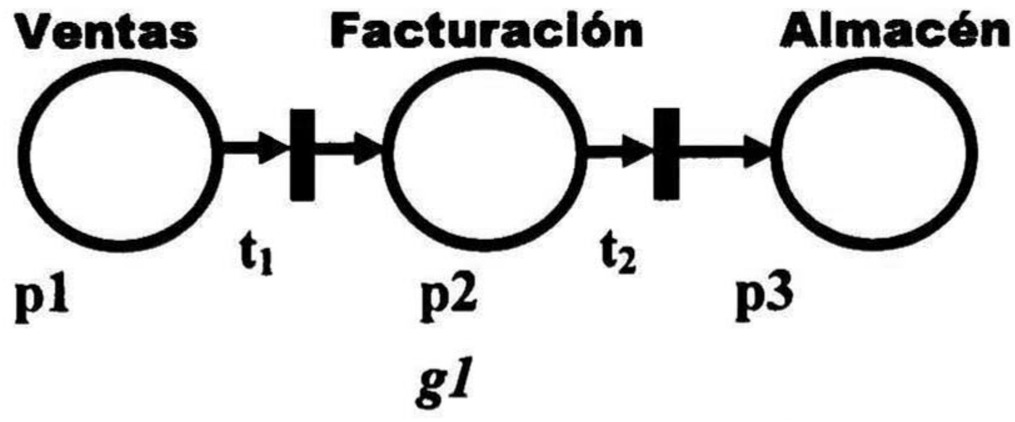


Figura 3.12: Sustitución de transiciones a *g1*

Una vez realizadas las sustituciones necesarias, introduciremos al grafo dos transiciones más:

- Una transición t_0 que no tendrá lugar de entrada y su lugar de salida será el lugar *p1*. Esta transición modelará el inicio del caso y por lo tanto la generación del agente que guiará el proceso (la cual estará modelada como la red de nivel inmediato inferior y que servirá de marcado para red de nivel 1) y comenzará a la ejecución de tareas en el lugar *p1*.

- Una transición t_{m+1} donde *m* es el número de transiciones del grafo *g1* antes de agregar t_0. Esta transición tendrá como lugar de entrada al último lugar del grafo *g1* (es decir, al lugar final donde termina el caso) y no tendrá lugares de salida. Con esta transición modelaremos la terminación del caso y la destrucción del agente que guía el proceso.

Por tanto, en el ejemplo 4, al introducir las nuevas transiciones tenemos la estructura de la figura 3.13

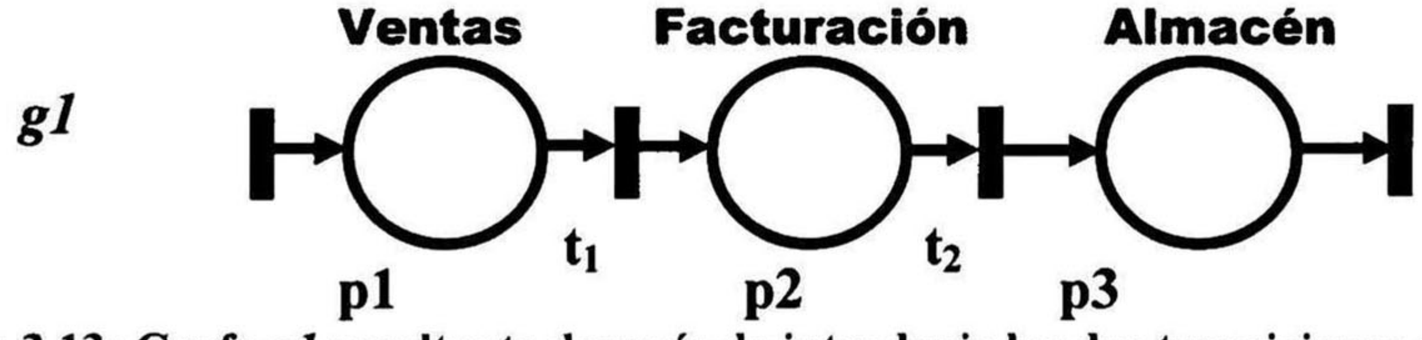


Figura 3.13: Grafo *g1* resultante después de introducir las dos transiciones especiales t_0 y t_3

El conjunto *T* de transiciones quedaría como : $T= \{ t_0, t_1, t_2, t_3\}$ y *F* quedaría como: $F=\{(t_0,p1), (p1, t_1), (t_1,p2), (p2, t_2), (t_2,p3),(p3, t_3)\}$, con esto queda definido el grafo G de nuestra red $Net_{1,1}$.

En esta última red podríamos suponer la existencia de un lugar p0 que fuera la entrada de la transición t_0 y un lugar p4 que fuera la salida de la transición t_3, si no se desea un modelo que comience con una transición y termine con una transición. El lugar p0 podría

modelar el estado de espera por inicio del caso (en cuyo caso podría representarse un marcado inicial como un token en dicho lugar) en donde al dispararse la transición t0 se representaría el inicio del caso y la generación de la red que representa al agente (el cambio de un token por una variable que representa una red de nivel interno). El lugar p4 podríamos modelar el lugar final del caso, es decir, un lugar en donde ya no se realizarán tareas en lugar de tirar la marca (ver figura3.14).

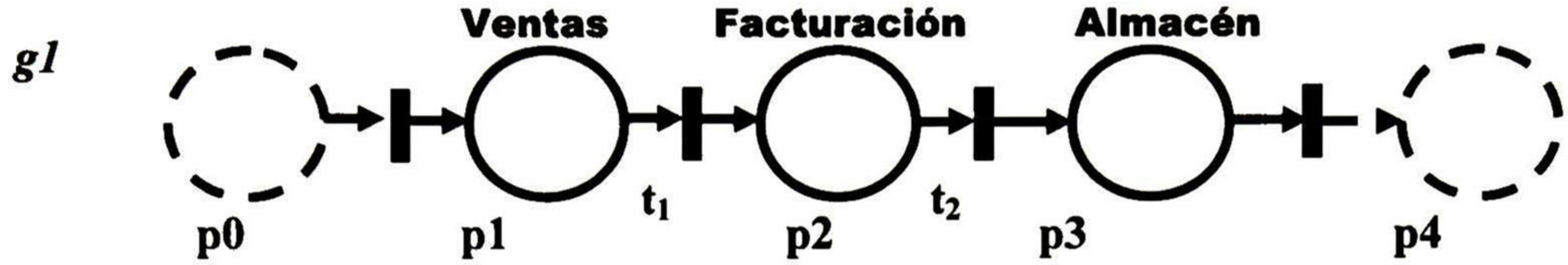


Figura 3.14: Estructura de la red resultante, si se agregan un lugar de inicio y un lugar final

- Definición de los conjuntos y funciones restantes de la red $Net_{1,1}$.

  - Asignación de etiquetas a cada transición. Asignaremos etiquetas a cada una de las transiciones de nuestro grafo G; el nombre asignado a cada etiqueta (para la mayor comprensión del modelo de RP) conviene que sea representativo a lo que queremos modelar con dicha transición. Como mencionamos anteriormente, en la red de nivel 1 cada transición nos representa el flujo de la información de un departamento (lugar) a otro de la empresa, por lo que se sugiere que etiquetemos a las transiciones de la siguiente forma: $MoverDep_i$ donde Dep_i es el nombre corto del departamento representado por el lugar que está a la salida de la transición a etiquetar, por lo que la etiqueta nos representaría *mover al departamento i*. En caso de que la transición posea más de una salida, deberá etiquetarse ésta con un nombre que simbolice la bifurcación necesaria.

Así para el ejemplo 4 tendríamos que:

- Para la transición t_0 podríamos utilizar la etiqueta *MoverVent* (Mover al departamento de Ventas). Esta transición al no tener lugar de entrada siempre está habilitada por lo que al seleccionar su disparo nos representaría el inicio del caso y la generación de un token que marcará a la red.
- Así podríamos definir las etiquetas *MoverFact, MoverAlmac, Mover* para las transiciones t1,t2,t3 respectivamente; el conjunto de etiquetas es:

$LABEL_{1,1}$ = *{MoverVent, MoverFact, MovAlmac, Mover}*

- Tokens de la red. En este punto se deben definir los tokens permitidos en cada lugar de la red. Para esta red de nivel 1 cada lugar red podrá ganar una marca que representará a la red del nivel inferior que nos modelará la red agente a la que llamaremos $Net_{2,1}$ (esta red de nivel 2 se definirá formalmente más adelante) pero para efectos de simplicidad del modelo, deberemos asociar a esta red con una variable definiendo así a el conjunto $VAR_{i,k}$.

  Para el ejemplo 4 definamos $TOKEN_{1,1}=\{Net_{2,1}\}$, ahora tomemos a *x* como la variable que se asociará a la red de nivel inferior $Net_{2,1}$, por lo tanto tendríamos a $VAR_{1,1}=\{x: Net_{2,1}\}$

- Función τ. Todo lugar de nuestra red de nivel 1 podrá admitir un token de la red de tipo $Net_{2,1}$ por lo que la esta función estaría definida como:

  $\tau(p1)= \tau(p2)=...= \tau(pn)=\{ Net_{2,1}\}$ donde *n* es igual al número de lugares de la red de nivel 1.

  Para el ejemplo 4 la función τ para cada lugar quedaría como:

  $\tau(p1)= \tau(p2)=\tau(p3)=\{ Net_{2,1}\}$

- Función λ

  Como se definió en la estrategia general descrita en el capítulo I, el disparo de cada una de las transiciones (exceptuando a t_0 por no poseer lugar de entrada por lo que no poseerá atributos de sincronización) deberá poseer un atributo de sincronización interna para sincronizar su disparo con la red de nivel inferior $Net_{2,1}$, por lo que la función λ estaría definida como:

  $\lambda(t_0)= \{MoverDep_1\}$

  $\lambda(t_1)= \{MoverDep_2^{\downarrow}\}$

  :

  $\lambda(t_m)= \{MoverDep_n^{\downarrow}\}$ donde *m* es el número de transiciones de la red G antes de agregar las dos transiciones especiales y *n* es el número de lugares de la red G.

  $\lambda(t_{m+1})= \{Mover^{\downarrow}\}$ para la transición agregada a la salida del departamento *i*

Para el ejemplo 4 se obtiene:

$\lambda(t_1)= \{MoverFact^{\downarrow}\}$

$\lambda(t_2)= \{MoverAlmac^{\downarrow}\}$

$\lambda(t_3)= \{Mover^{\downarrow}\}$

$\lambda(t_0)= \{MoverVent\}$

- Función π. Debido a que todos los lugares de la red admiten únicamente al mismo tipo de red (la red $Net_{2,1}$), esta función quedaría definida como:

$\pi((t_0,p1),MoverDep_1)=x$

$\pi((p1,t_1,),MoverDep_2^{\downarrow})= \pi((t_1,p2),MoverDep_2^{\downarrow})=x$

:

$\pi((p_{n-1},t_m,),MoverDep_n^{\downarrow})= \pi((t_m,p_n),MoverDep_n^{\downarrow})=x$

Para *n* igual al número de lugares en la red de nivel 1 (red $Net_{1,1}$) y *m* igual al número de transiciones de la red G antes de agregas las transiciones especiales.

$\pi((p_i,t_{j+1},),Mover^{\downarrow})=x$

Para el ejemplo 4 tendríamos:

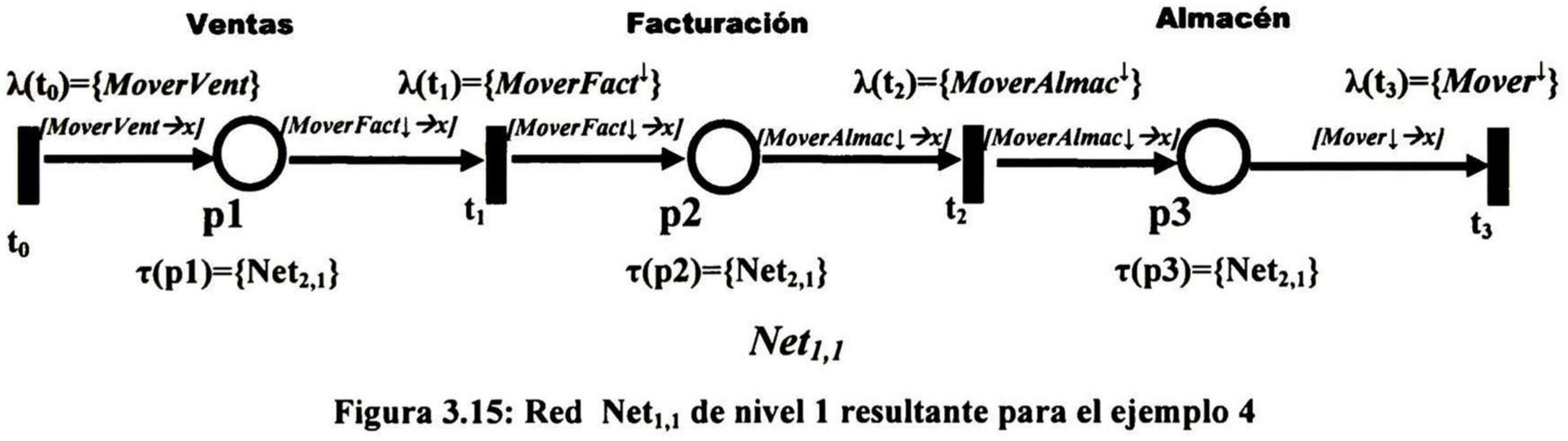
$\pi((t_0,p1),MoverVent)=x$

$\pi((p1,t_1,),MoverFact^{\downarrow})= \pi((t_1,p2),MoverFact^{\downarrow})=x$

$\pi((p2,t_2,),MoverAlmac^{\downarrow})= \pi((t_2,p3),MoverAlmac^{\downarrow})=x$

$\pi((p3,t_3,),Mover^{\downarrow})= x$

Para este punto ya tenemos definida nuestra red de nivel 1 como lo dice el formalismo nLNS la $Net_{1,1}$, para el ejemplo 4 es mostrada en la figura 3.15.



$Net_{1,1}$

Figura 3.15: Red $Net_{1,1}$ de nivel 1 resultante para el ejemplo 4

o Marcado de la red $Net_{1,1}$., por la manera en la que se construyó la red ambiente, o de nivel 1, el marcado inicial es vacío, debido a que el flujo de la red comienza con una transición, por lo que dicha transición de inicio siempre estará habilitada permitiendo así en cualquier tiempo el inicio del caso.

Para el ejemplo 4 se tiene: $M_0=\varnothing$

Una vez obtenida la red ambiente o red de nivel 1 siguiendo los pasos anteriormente descritos, pueden agregarse a este modelo más aspectos a modelar, por ejemplo, pueden modelarse recursos (como almacenes, recursos computacionales como pc o impresoras, etc). Si se deseara modelar en la red $Net_{1,1}$ obtenida del ejemplo 4 la asignación de recursos humanos (por ejemplo), es decir, los empleados de cada departamento que se involucran en cada caso, la red de la figura 3.15 se transforma quedando como la mostrada en la figura 3.16.

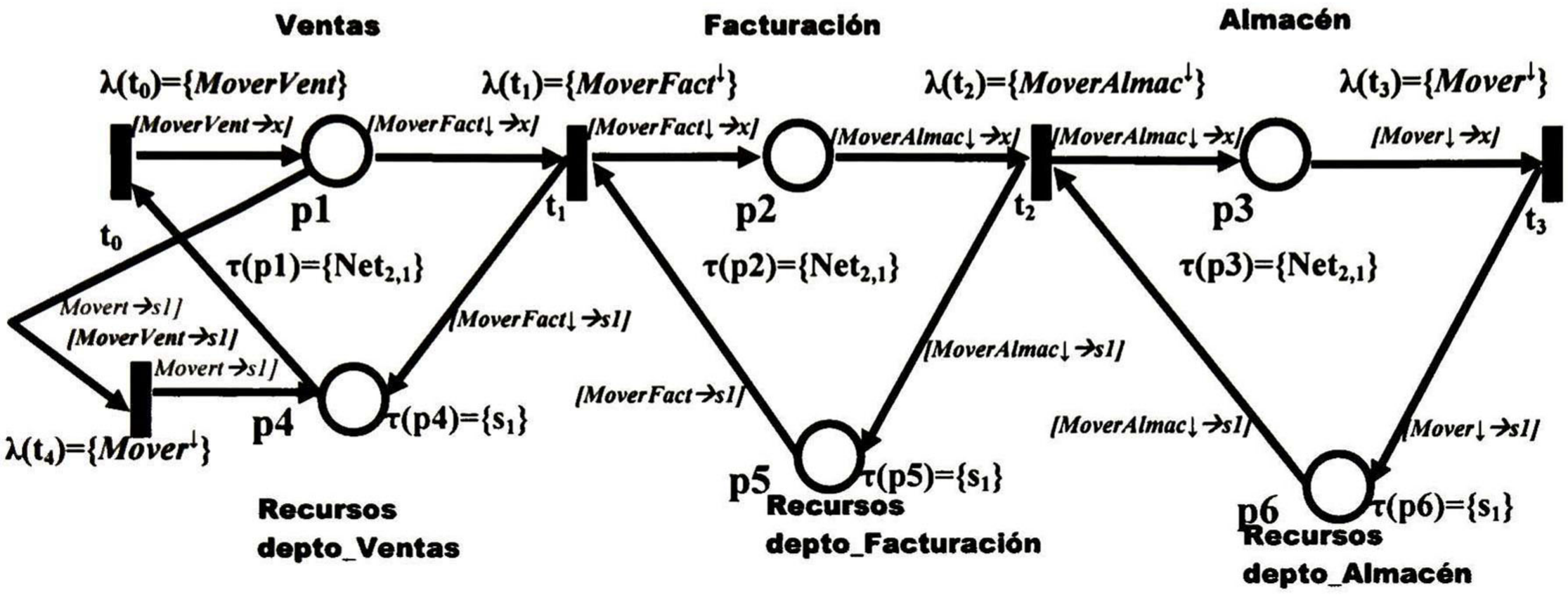


Figura 3.16: Red resultante al modelar recursos para el ejemplo 4

A la red de la figura anterior, se le ha agregado también una nueva transición (t4) la cual se etiquetó como *Mover*, esta transición se utilizará para cancelar el caso (si es necesario) y liberar el recurso.

A continuación describiremos la construcción de las redes que están contenidas en la red ambiente, pero para facilitar su explicación introduciremos primero la obtención de los modelos de las redes de nivel 3.

1.2.1 Construcción de las Redes de Tercer nivel

Tal como se mencionó en el capítulo I, utilizaremos redes de tercer nivel para modelar dos cosas diferentes: primero obtendremos el modelo de la definición del proceso del flujo de trabajo (es decir, la red que define el comportamiento específico del agente) y segundo las redes correspondientes a cada uno de los modelos de las tareas que el agente va a realizar. Ambos tipos de redes serán el marcado de la red agente de nivel 2.

- *Red para el modelado de la definición del proceso del flujo de trabajo*

En esta red modelaremos el comportamiento especifico del agente (el cual estará modelado de manera general mediante la red de nivel 2 que se escribirá más adelante); con lo que se modelarán las tareas que deberán ejecutarse en el problema de flujo de trabajo, así como el orden en las cuales éstas serán ejecutadas, por lo que ésta red deberá iniciar en un solo lugar y se recomienda que también tenga solo un lugar de salida.

Para construir esta red, necesitaremos analizar la definición del problema de flujo de trabajo con el objetivo de identificar la mayor cantidad posible de tareas involucradas. En caso de que de la definición no se obtengan todas las tareas que deben realizarse en el flujo de trabajo, se deberá obtener dicha información de la empresa (para la que se realiza el modelado), analizando qué tareas se realizan en cada departamento.

Por lo tanto, para realizar el modelado de esta red, definiremos los siguientes pasos:

1. *Identificación de las tareas involucradas.* Como ya se obtuvo el modelo general de los departamentos involucrados en el flujo de trabajo así como el sentido del flujo de la información (en la construcción de la red de nivel 1); por cada departamento que aparece en dicho modelo deberán identificarse todas las tareas que se deben realizar en el problema.

Con esta información define al flujo de trabajo como el conjunto W_i, tal que $W_i=\{tk_1, tk_2,....,tk_n\}$ donde n es el número de tareas involucrada en el flujo de trabajo *i*.

Para el ejemplo 4, que es el que hemos estado modelando, observamos que solo tenemos 3 departamentos involucrados: *Ventas, Facturación* y *Almacén*. Por lo que vamos a suponer que las tareas que se realizan en cada departamento, son las siguientes:

- ***Ventas***: *Recepción_de_solicitud* (Tarea para la recepción de la solicitud o petición del cliente de información sobre un producto), *Verificación_existencia_Producto* (tarea en la que, una vez obtenido el nombre o clave de producto, se verifica la existencia del producto en la base de datos), *Proporcionar_info* (esta tarea sirve para proporcionar los datos del producto que el cliente desea conocer), *Validación_compra* (tarea para verificar si el comprador desea adquirir el producto o no), *Generar_pedido* (significa que el comprador va a adquirir el producto por que esta tarea genera la orden de pago), *Registro* (en esta tarea se almacena o registra la información generada en este departamento).

- *Facturación*: *Facturar* (Tarea para generar la hoja de factura), *Cobro* (tarea para recibir el pago del cliente), *Generar_orden_Entrega* (Tarea para generar la orden de entrega al departamento de almacén).
- *Almacén*: *Entrega* (tarea para entregar dicho producto al comprador), *Registro* (en esta tarea se almacena o registra la información generada en este departamento).

Por lo que W_1 quedaría formada de la siguiente manera:

W_1 ={*Recepcion_de_Solicitud, Verificación_existencia_Producto, Proporcionar_info, Validación_compra, Generar_pedido, Registro, Facturar, Generar_orden_Entrega, Entrega*}

2. Analizar el orden de ejecución de las tareas en W_i. Para lograr esto debemos realizar la siguiente definición:

Asociado a W_i existe un orden de precedencia $tk_l \prec tk_m$ el cual implica que tk_l deberá ejecutarse antes que tk_m y que tk_m se inicia después de la ejecución de tk_l, siendo tk_l y tk_m cualquier par de elementos en W_i.

Supongamos que la tarea 1 (tk_1) de W_i debe ejecutarse antes que la tarea 2 (tk_2) de W_i, y tk_2 se inicia después de la ejecución de tk_1, entonces la relación de orden de precedencia quedaría como: $tk_1 \prec tk_2$.

Y si a su vez tenemos que la tarea 1 (tk_1) de W_i debe ejecutarse antes que la tarea 3 (tk_3) de W_i, y tk_3 se inicia después de la ejecución de tk_1, entonces la relación de orden de precedencia quedaría como: $tk_1 \prec tk_3$.

Ahora bien definamos W_i' como el conjunto de todos los ordenes de precedencia de las tareas en W_i

Por tanto, se debe encontrar este orden de precedencia de las tareas en W_i, con el objeto de identificar en qué orden deberán ejecutarse dichas tareas y así construir nuestra red.

Para el ejemplo 4, el la relación de orden de precedencia quedaría como:

W_1'={*Recepcion_de_Solicitud* $\prec$ *Verificación_existencia_Producto, Verificación_existencia_Producto* $\prec$ *Proporcionar_info, Verificación_existencia_Producto* $\prec$ *Registro, Proporcionar_info* $\prec$ *Validación_compra, Validación_compra* $\prec$ *Generar_pedido, Validación_compra* $\prec$ *Registro,*

Generar_pedido $\prec$ *Facturar,* *Facturar* $\prec$ *Generar_orden_Entrega,*
Generar_orden_Entrega $\prec$ *Entrega, Entrega* $\prec$ *Registro}*

3. Construcción de la Red. Definamos a red de nivel 3 como $Net_{3,1}$, tal que $Net_{3,1}$ =(G, $TOKEN_{3,1}$,$LABEL_{3,1}$,$VAR_{3,1}$, τ, λ, χ, π) donde todos sus elementos por el momento serán igual a $\varnothing$.

   Ahora encontremos el valor de cada uno de los elementos de esta red:

   a. En el paso 1 y 2 encontramos las tareas involucradas en el problema de flujo de trabajo y además analizamos su orden de precedencia, con ésta información vamos a construir nuestro grafo G.

      – El modelado del grafo G a partir de la información generada en el punto 2, puede realizarse de muchas maneras según la experiencia del modelador con las Redes de Petri. A continuación sugeriremos una manera de construir dicho grafo con la información disponible.

      Para el departamento 1 (por ser éste el departamento donde inicia el caso) separar las tareas involucradas, en dicho departamento, de W_i y sus respectivos ordenes de precedencia que aparecen en W_i'

      Para el ejemplo 4 esta información sería:

      Departamento de Ventas:

      Tareas:

      {*Recepcion_de_Solicitud, Verificación_existencia_Producto, Proporcionar_info, Validación_compra, Generar_pedido, Registro*}

      Ordenes de precedencia:

      {*Recepcion_de_Solicitud* $\prec$ *Verificación_existencia_Producto,*
      Verificación_existencia_Producto $\prec$ *Proporcionar_info,*
      Verificación_existencia_Producto $\prec$ *Registro,* *Proporcionar_info* $\prec$ *Validación_compra,*
      Validación_compra $\prec$ *Generar_pedido,* *Validación_compra* $\prec$ *Registro,*
      Generar_pedido $\prec$ *Recibir_orden_Pago* }

      – Una vez separadas las tareas y su orden de precedencia, modelaremos cada tarea como una red *transición*$_{l,1}$*-lugar-transición*$_{l,2}$, donde l es el número de tarea en *Wi* por lo que tendremos una red por cada tarea a realizar en el primer departamento.

La transición *transición*$_{k,1}$ se utilizará para enfatizar el inicio de la tarea por lo que la etiquetaremos como *Inicio_tk*$_l$, y a la transición *transición*$_{k,2}$ se utilizará para enfatizar la terminación de la ejecución de la tarea por lo que la etiquetaremos como *fin_tk*$_l$.

Por ejemplo, supongamos que tenemos el W_2 definido por $W_2 = \{tk_1, tk_2, tk_3\}$ por lo que se obtendrían las redes mostradas en la figura 3.17.

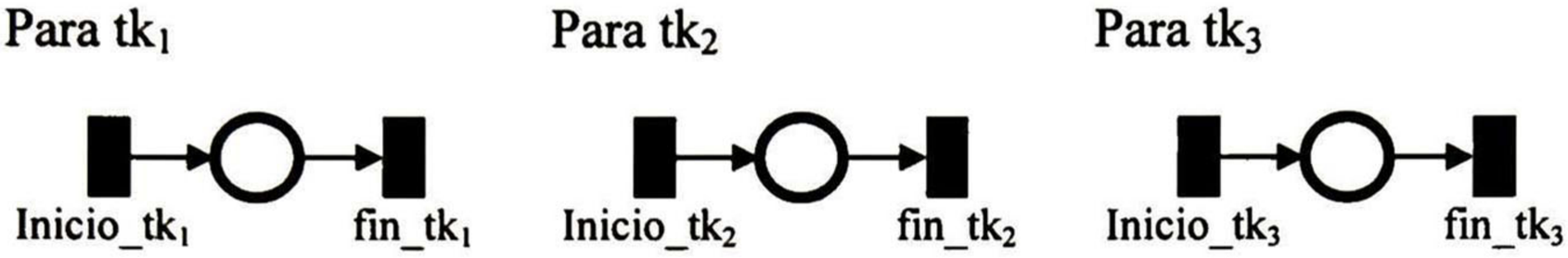


Figura 3.17: Modelado de Tareas por redes transicion_lugar_transicion

Para el ejemplo 4, las redes de las tareas del departamento 1 (*Ventas*) serían las mostradas en la figura 3.18.

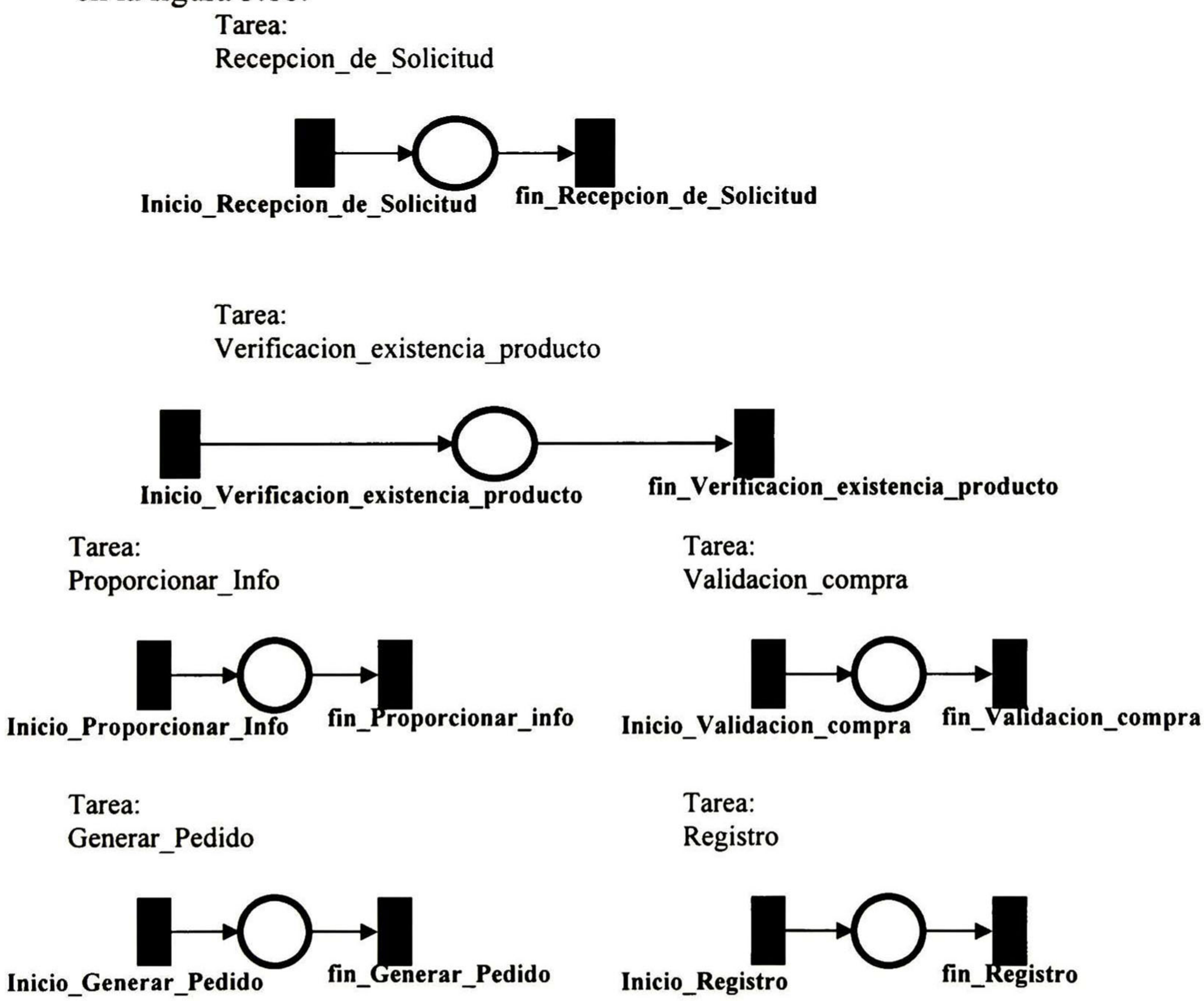


Figura 3.18: Construcción de las redes de las tareas involucradas en el departamento de ventas del ejemplo 4

– Ya que tenemos todas las tareas del primer departamento modeladas con redes, procederemos a unir dichas redes con la ayuda de la información proporcionada por los órdenes de precedencia.

Para lograrlo, deberemos analizar la información que representa el orden de precedencia. Como tenemos identificadas a las tareas involucradas en el departamento donde inicia el caso (departamento 1) entonces la tarea que, de acuerdo el orden de precedencia definido, NO necesita que otra tarea se haya ejecutado antes, será la *tarea inicial*. Por lo que iremos construyendo un grafo al que llamaremos $g_{3,1}$. Este grafo comenzará con un lugar al que uniremos por medio de un arco a la transición inicial de la red de la identificada como *tarea inicial*. Supongamos que la tarea inicial la hemos identificado como la tk_1 y cuya red ya hemos construido en el paso anterior (ver figura 3.19).

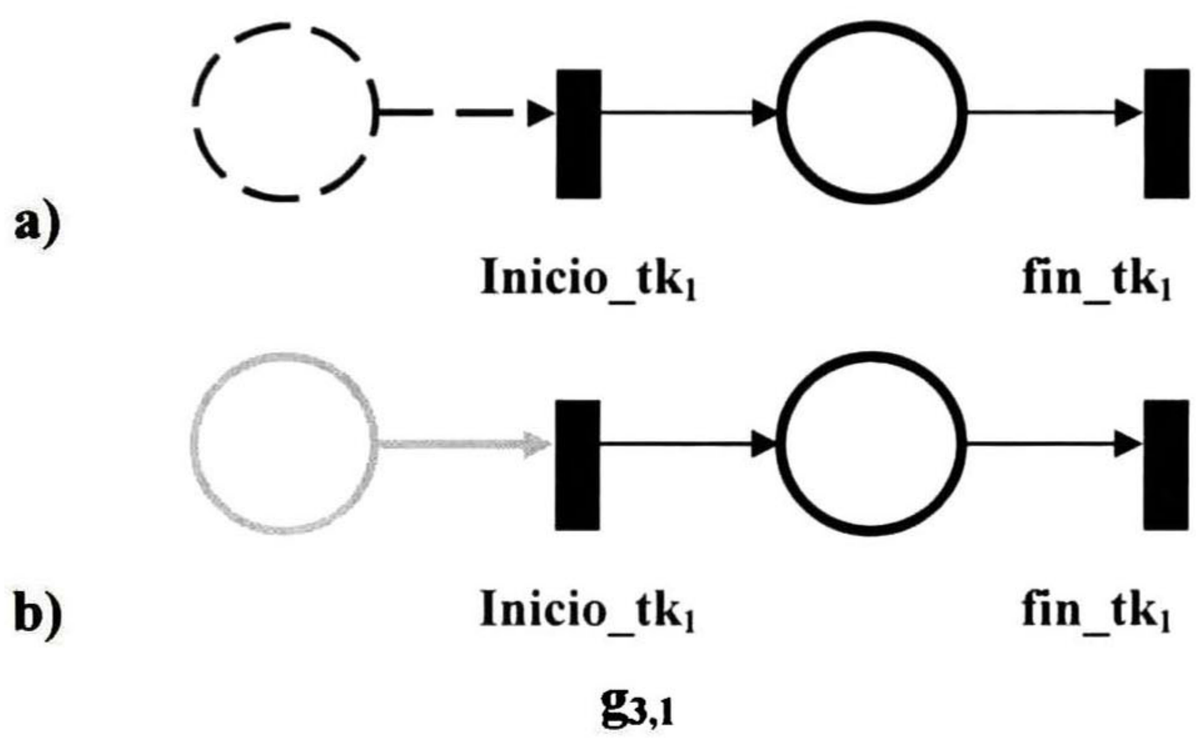


Figura 3.19: Obtención del grafo $g_{3,1}$. a) Red de la tarea tk_1 resaltando en donde se agregará el lugar, b) grafo resultante

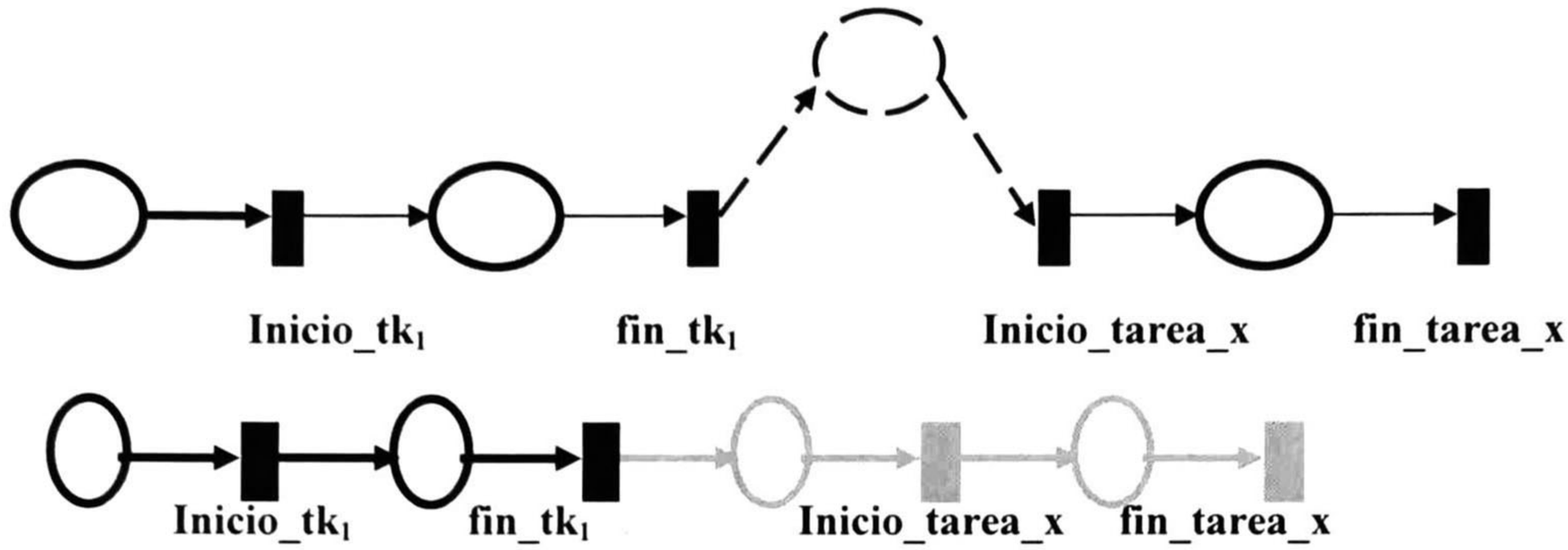
Ahora, debemos tomar las relaciones precedencia que inicien con la tarea identificada como *tarea inicial*. Aquí podemos encontrar los siguiente casos:

1. Un solo elemento del conjunto W_i', esto significa que después de la ejecución de la *tarea inicial* solo podrá iniciarse la tarea involucrada en la relación de dependencia.

Ya que sabemos que solo le puede seguir una tarea, supongamos la *tarea x*, ahora deberemos revisar entre los elementos de W_i' si existen otras relaciones de precedencia que **terminen** con la *tarea x*. Pueden ocurrir dos casos:

⇒ No se encuentran otros elementos en W_i', lo que significaría que para la ejecución de la *tarea x*, solamente debe de terminarse la ejecución de la tarea con la que se encuentra relacionada en W_i', por lo tanto se deberá unir la transición **inicial** de la *tarea x* con la

transición **final** de su predecesora en el grafo $g_{3,1}$ que llevamos construido agregando un lugar para poder unir ambas transiciones.



$g_{3,1}$

Figura 3.20: Construcción del grafo $g_{3,1}$

⇒ Se encuentran uno o más elementos en W_i', significa que para iniciar la ejecución de la *tarea x* se necesitan cumplir más condiciones, por lo que se deberá analizar de qué forma se une el flujo en la *tarea x*, puede ser en forma excluyente, es decir, que el flujo solo llegará en un caso dado por una de las precedencias de la *tarea x* (como la unión en un OR exclusivo), por todas (en paralelo como la unión en un AND) o por una combinación de éstas. Por lo que para esto el modelador deberá hacer uso de su capacidad para modelar y para identificar los distintos casos según se necesite. Por simplicidad nos centraremos en los casos en los que se trate de un OR exclusivo o un AND.

* Si se tratara de flujos excluyentes y suponiendo que se tienen las siguientes relaciones de precedencia:

$$\{tarea\_inicial \prec tarea\_x,\ tarea_i \prec tarea\_x,\ tarea_j \prec tarea\_x\} \in W_i'$$

Entonces la *tarea_inicial* se deberá unir a la *tarea_x* de la misma forma como se describió en la figura 3.20, es decir, agregando un nuevo lugar para poder unir ambas transiciones. Además se agregarán más arcos de entrada punteados a este nuevo lugar agregado, tantos como el resto de las relaciones faltantes en W_i', dichos arcos punteado luego serán reemplazado por un arco continuo, cuando se agreguen las tareas $tarea_i$ y $tarea_j$ al grafo $g_{3,1}$,(ver figura 3.21).

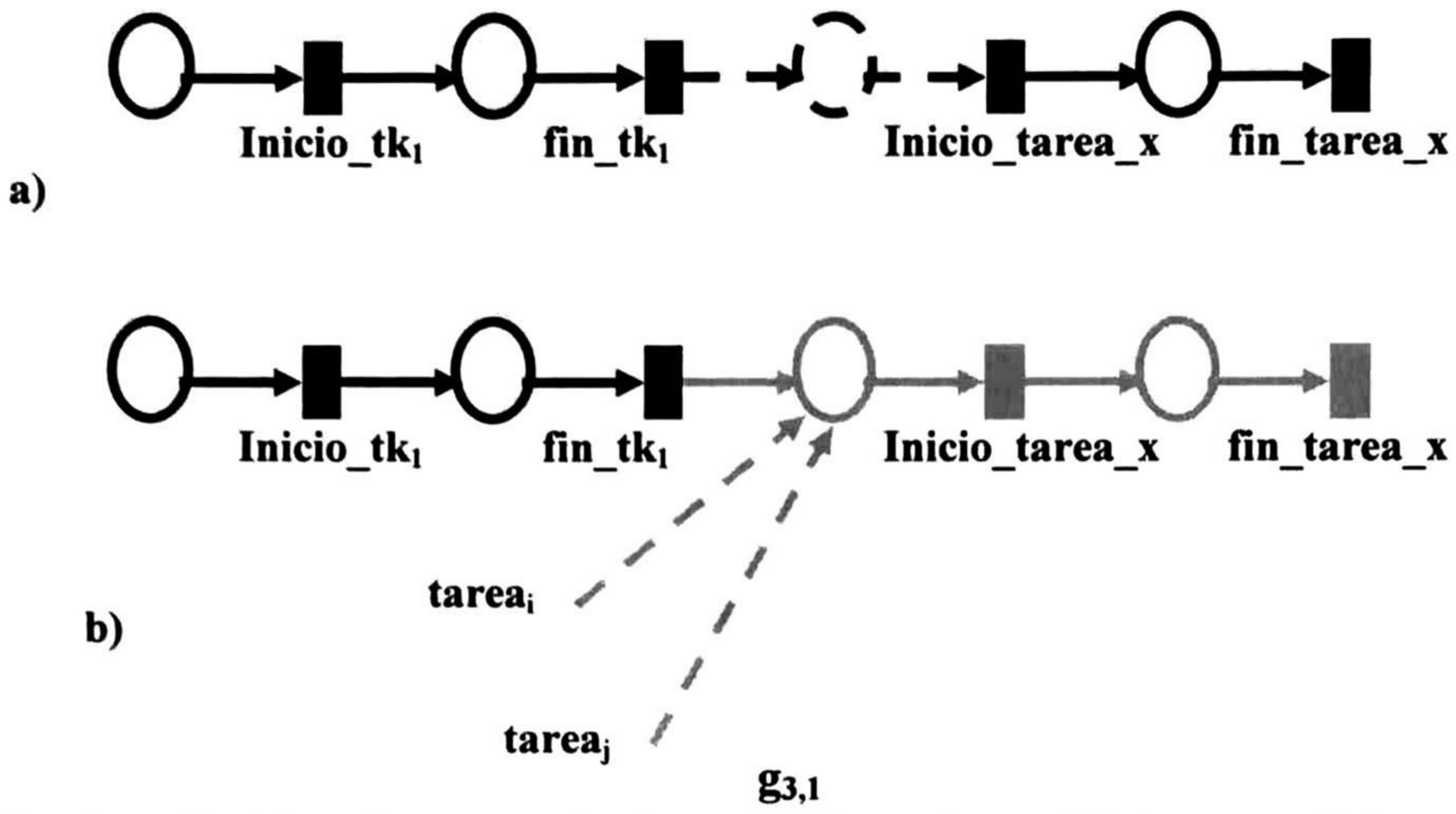


Figura 3.21: Construcción del grafo $g_{3,1}$ suponiendo que una tarea tiene múltiples precondiciones excluyentes, a)Unión de la tarea inicial tk1 con la tarea_x, b)adición de arcos punteados correspondientes a las tareas precondición de la tarea_x pero que aún no se agregan al grafo

* Si se tratara de flujos en paralelo y suponiendo que se tienen las siguientes relaciones de precedencia:

$$\{tarea\_inicial \prec tarea\_x,\ tarea_i \prec tarea\_x,\ tarea_j \prec tarea\_x\} \in W_i'$$

La construcción del grafo $g_{3,1}$ se hará de forma similar a la figura 3.21, solo que en lugar de agregar arcos punteados al lugar que se agrega para unir las transiciones de las redes de las tareas, se agregarán a la entrada de la transición que marca el inicio de la *tarea_x* para indicar con esto que se deberán haber completado todas las tareas identificadas en el orden de precedencia para poder iniciar la *tarea_x* (ver figura 3.22).

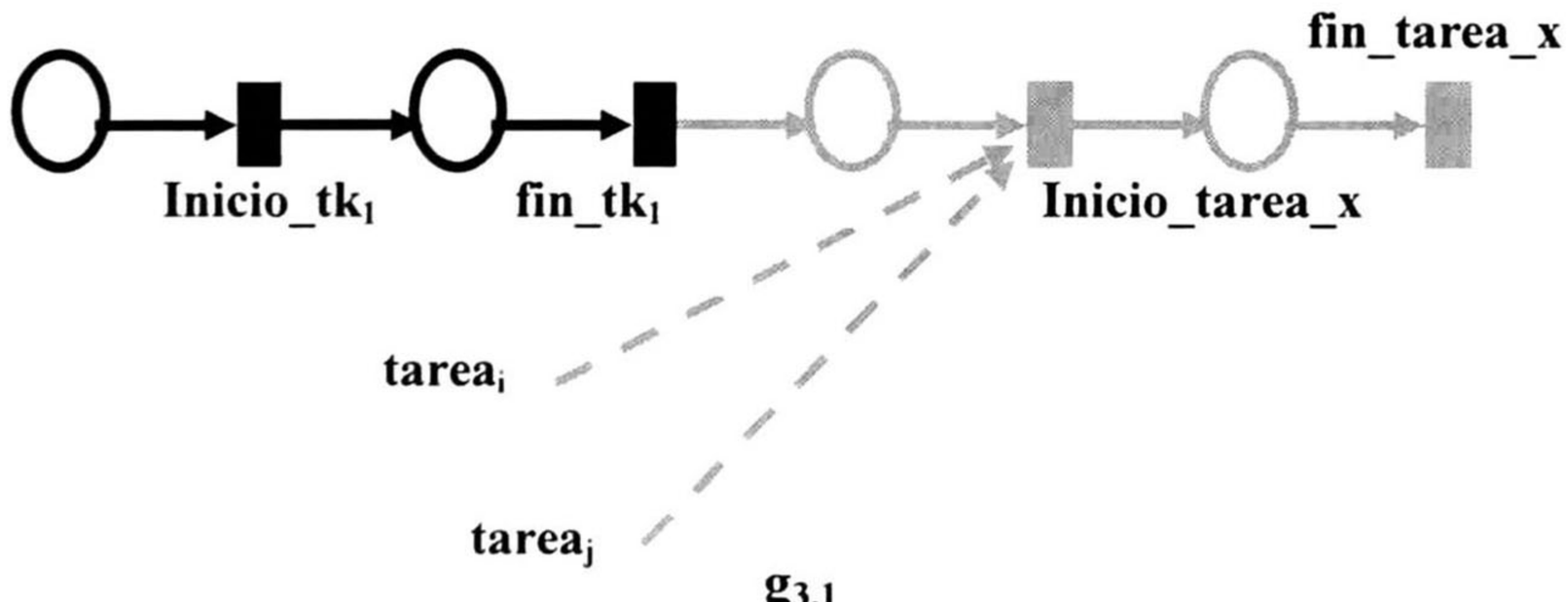


Figura 3.22: Construcción del grafo $g_{3,1}$ suponiendo que la tarea agregada posee más de una precondición necesaria

2. Varios elementos del conjunto W_i', esto significa que esta transición es precondición de varias tareas por lo que deberá analizarse el problema de flujo de trabajo para determinar si dichas tareas subsecuentes deben realizarse de forma paralela, de manera

excluyente (solo se elige una ruta a la vez) o una combinación de las anteriores. Para efectos de simplificación sólo nos enfocaremos en las primeras dos opciones.

- Tareas en forma paralela, por lo que significa que flujo de la red (el número de toquens en la red) va a aumentar. Supongamos que los elementos de Wi' encontrados son:

{tarea_inicial ≺ tarea_x, tarea_inicial ≺ tarea_y, tarea_inicial ≺ tarea_z}

Para unir las redes de las tareas recién identificadas, fusionaremos primero las primeras transiciones de las redes de las tareas, en una sola transición a la cual cambiaremos la etiqueta por un nombre que nos represente el inicio de las tareas a fusionar (ver figura 3.23).

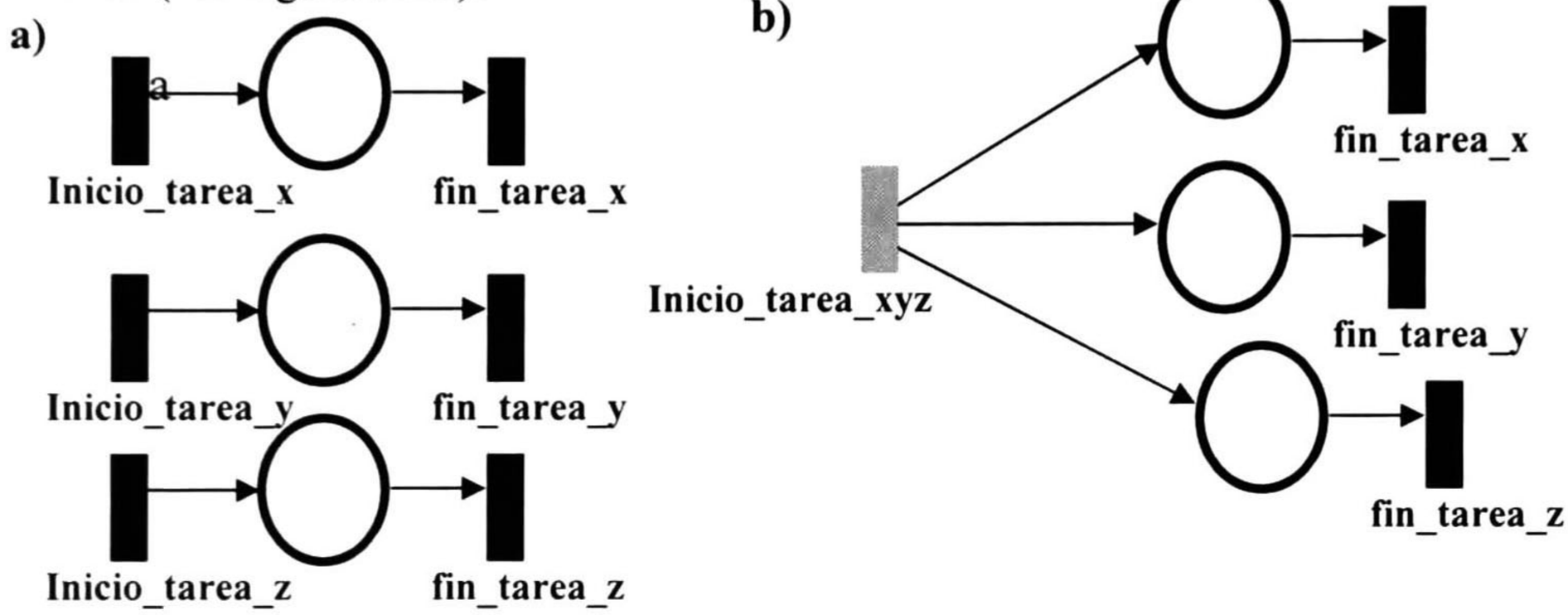


Figura 3.23: Fusión de las transiciones iniciales de las tareas que se ejecutan en paralelo

Ahora deberá unirse la nueva red de la figura **3.23 b)**, al grafo g3,1 agregando a la red un lugar auxiliar para la unión de las transiciones (ver fig. 3.24).

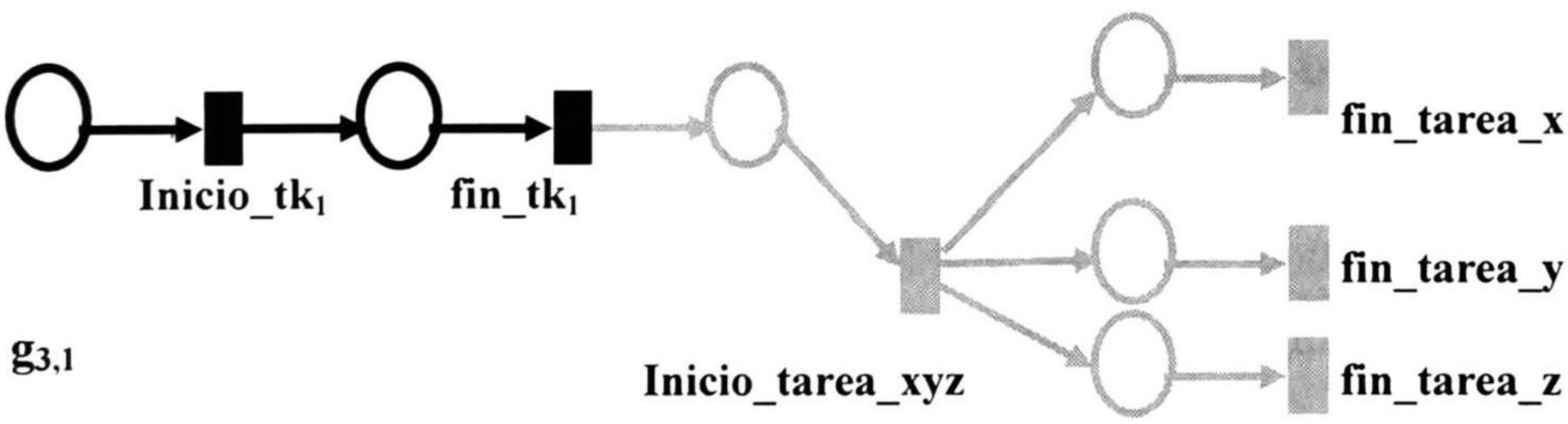


Figura 3.24: Grafo g3,1 al agregarle la estrura de la figura anterior inciso b)

- Para tareas en las que el flujo solo elige un camino a la vez, es decir, después de la ejecución de la *tarea_inicial* solo se inicia la ejecución de una sola tarea del conjunto posible, por lo que se dice que se realiza una elección. Supongamos que los elementos de Wi' encontrados son:

{tarea_inicial ≺ tarea_x, tarea_inicial ≺ tarea_y, tarea_inicial ≺ tarea_z}

Ahora solo resta aplicar el paso 1 para cada una de estas redes. Es importante notar que cada red de estas tareas se unirá al grafo $g_{3,1}$ utilizando un mismo y único lugar auxiliar.

El grafo podría resultar de la forma mostrada en la figura 3.25.

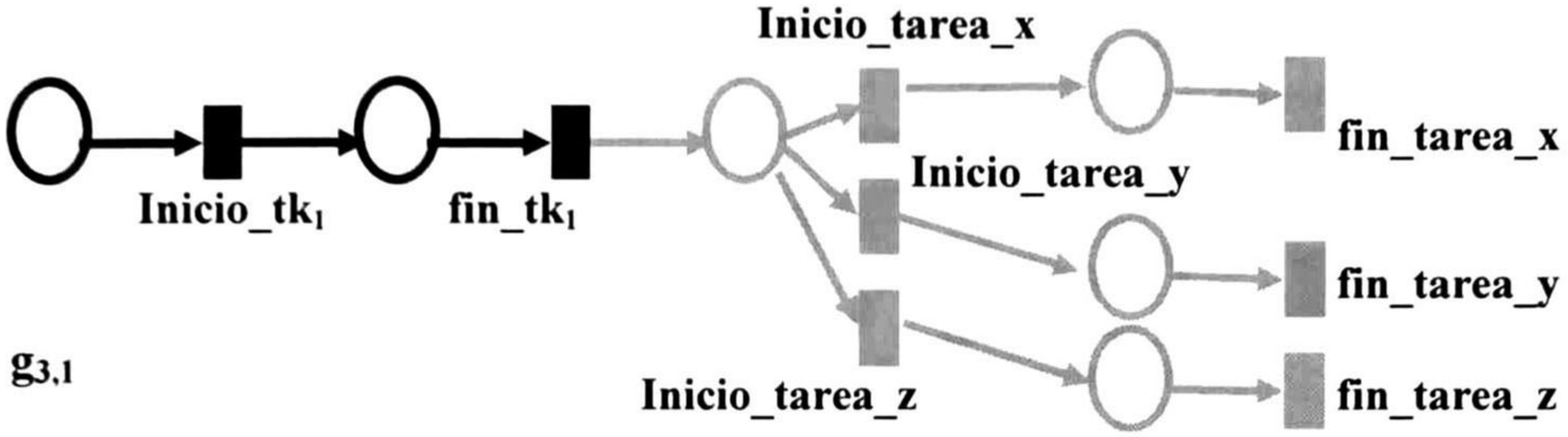


Figura 3.25: Grafo g3,1 resultante

En caso de que alguna (algunas) de las tareas (***tarea_x, tarea_y ó tarea_z***) ya se hayan agregado al grafo $g_{3,1}$ (supóngase la *tarea_z*) y al unir el lugar auxiliar con la transición de **Inicio_tarea_z** esta última resulte con dos lugares de entrada, se deberá realizar la siguiente sustitución: 1. Agregar un nuevo lugar auxiliar al grafo (supóngase lugar1), 2. Deberán unirse los lugares de entrada a la transición **Inicio_tarea_z** con el nuevo lugar1, se deberán agregar tantas transiciones como lugares de entrada a **Inicio_tarea_z** para conservar las reglas de unión. 3. Eliminar los arcos de entrada a la transición **Inicio_tarea_z.** 4. Unir el lugar1 con la transición **Inicio_tarea_z**

Para cada nueva tarea agregada, seleccionar de W_i' las relaciones de precedencia que comiencen con esta tarea y realizar los pasos 1 ó 2 según sea el caso. Se debe tener cuidado al unir los nuevos fragmentos de red al grafo, tomando en cuenta los arcos punteados agregados; por lo que al intentar unir un nuevo fragmento de red se deberá verificar si no existe un arco punteado que indique la ubicación de dicha unión en el grafo; de ser así se deberá reemplazar el arco punteado por un arco continuo de forma correspondiente.

Para el ejemplo 4 la red representada por la figura 3.26.

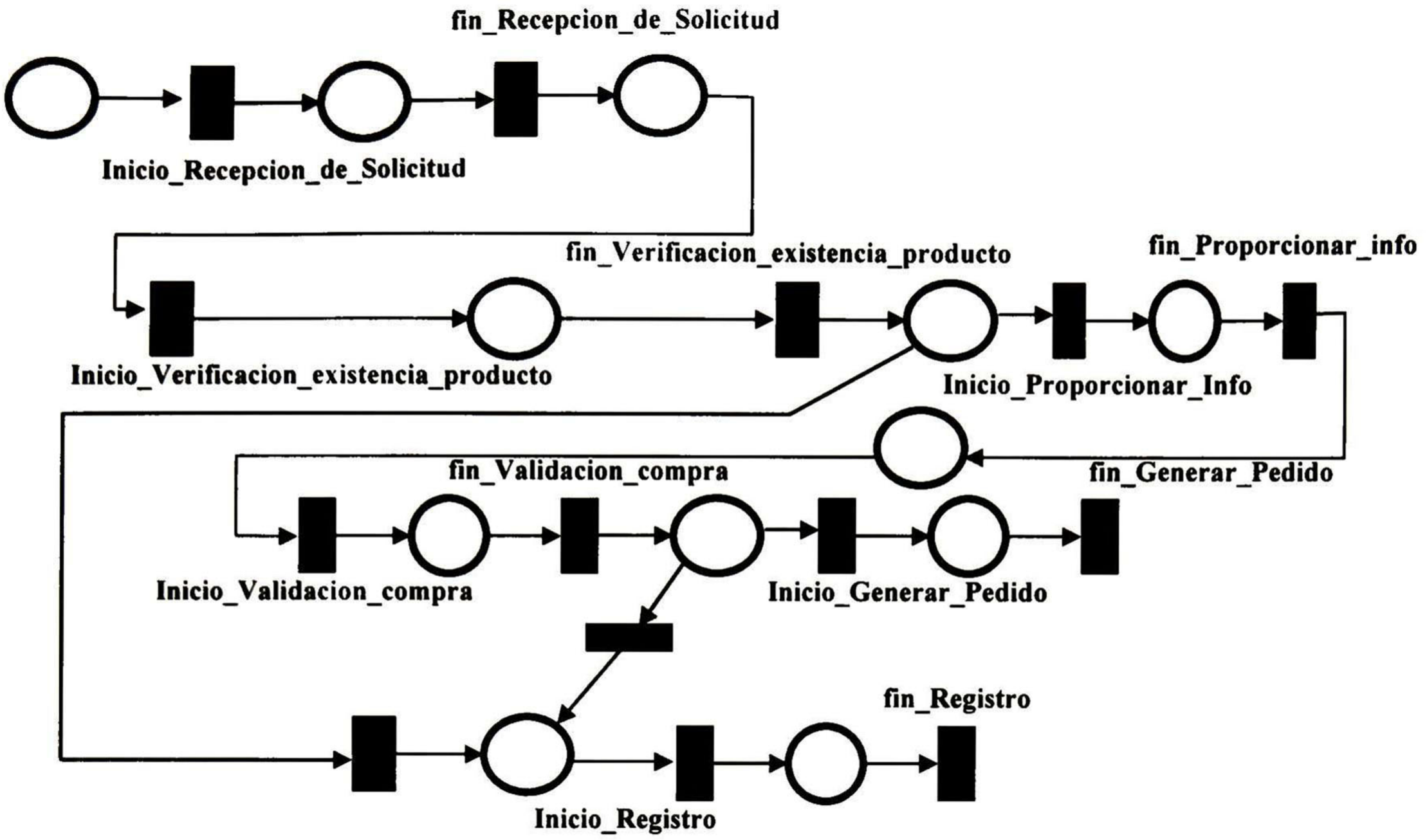


Figura 3.26: Grafo resultante al unir las tareas del departamento 1 para el ejemplo 4

Una vez agregadas todas las tareas del departamento 1, realizar los mismo para los departamentos siguientes, hasta unir todas las tareas al grafo $g_{3,1}$ con sus flujos correspondientes.

Ya que se agregaron todas las tareas agregar otro lugar al grafo $g_{3,1}$ al que llamaremos *lugar_final*, y deberá unirse la transición (transiciones) que no tienen lugar de salida en el grafo. Con lo anterior lograremos una red que tiene un único lugar de entrada y un único lugar de salida.

Para el ejemplo 4 se obtiene el grafo de la figura 3.27.

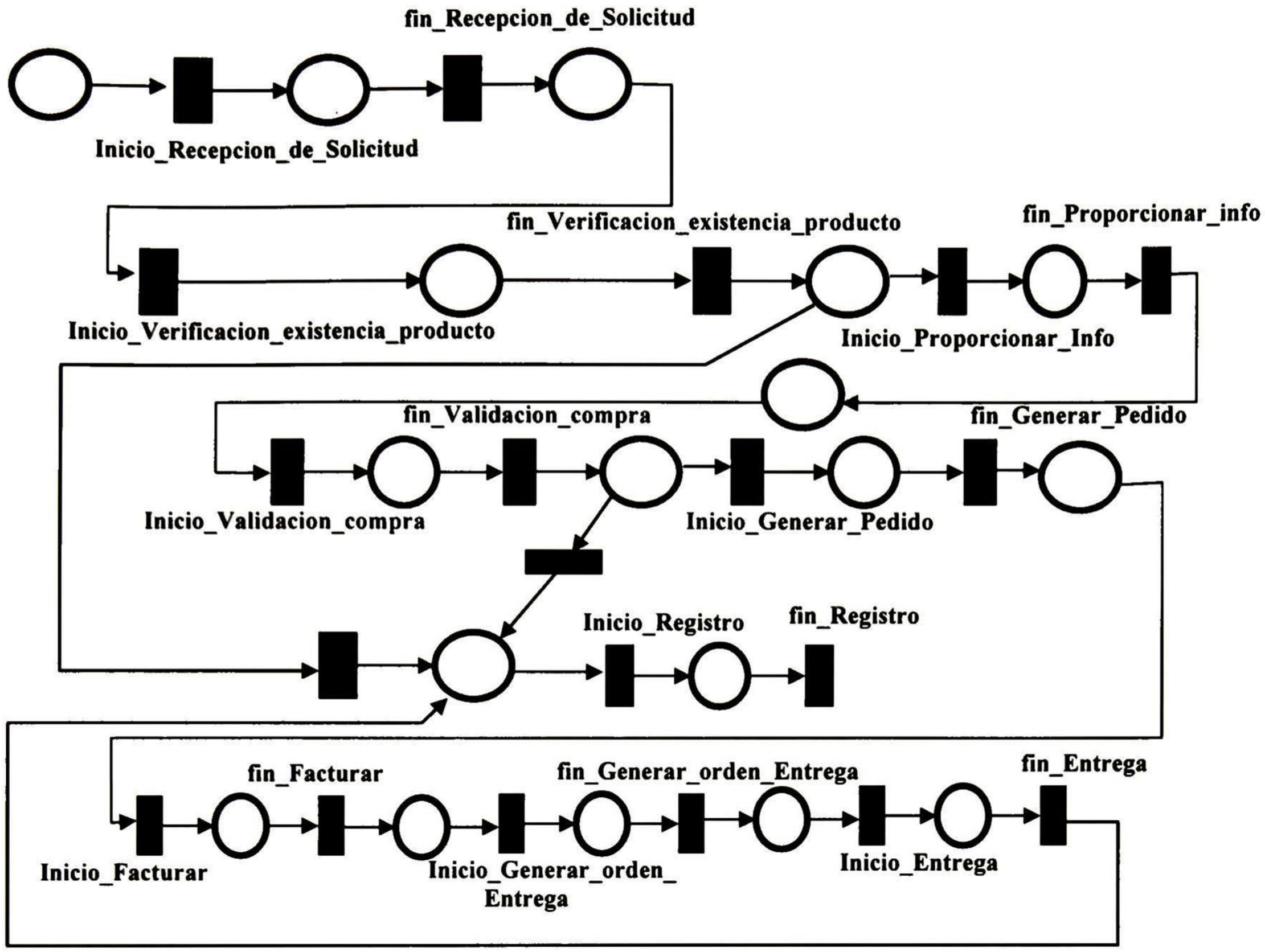


Figura 3.27: Grafo resultante para el ejemplo 4, al unir todas las tareas.

Finalmente agregaremos al grafo resultante $g_{3,1}$ unas **nuevas transiciones**, las cuales nos modelarán el paso del flujo o información de departamento de la organización a otro. Para esto deberemos observar los segmentos de red que modelan el final de una *tarea_y* y el inicio de otra *tarea_z*, si la *tarea_y* se ejecuta en un departamento distinto de la *tarea_z* entonces el lugar auxiliar que se usa para unir las dos redes se sustituirá por un segmento de red *lugar-transicion-lugar*, y a la transición la etiquetaremos como MoverDepartamento_ejecución_tarea_z (donde Departamento será el nombre del departamento donde se ejecuta la tarea_z, de hecho, deberá de ser el mismo nombre que se utilizó para etiquetar las transiciones de la red de nivel 1). Esto es mostrado en la figura 3.28.

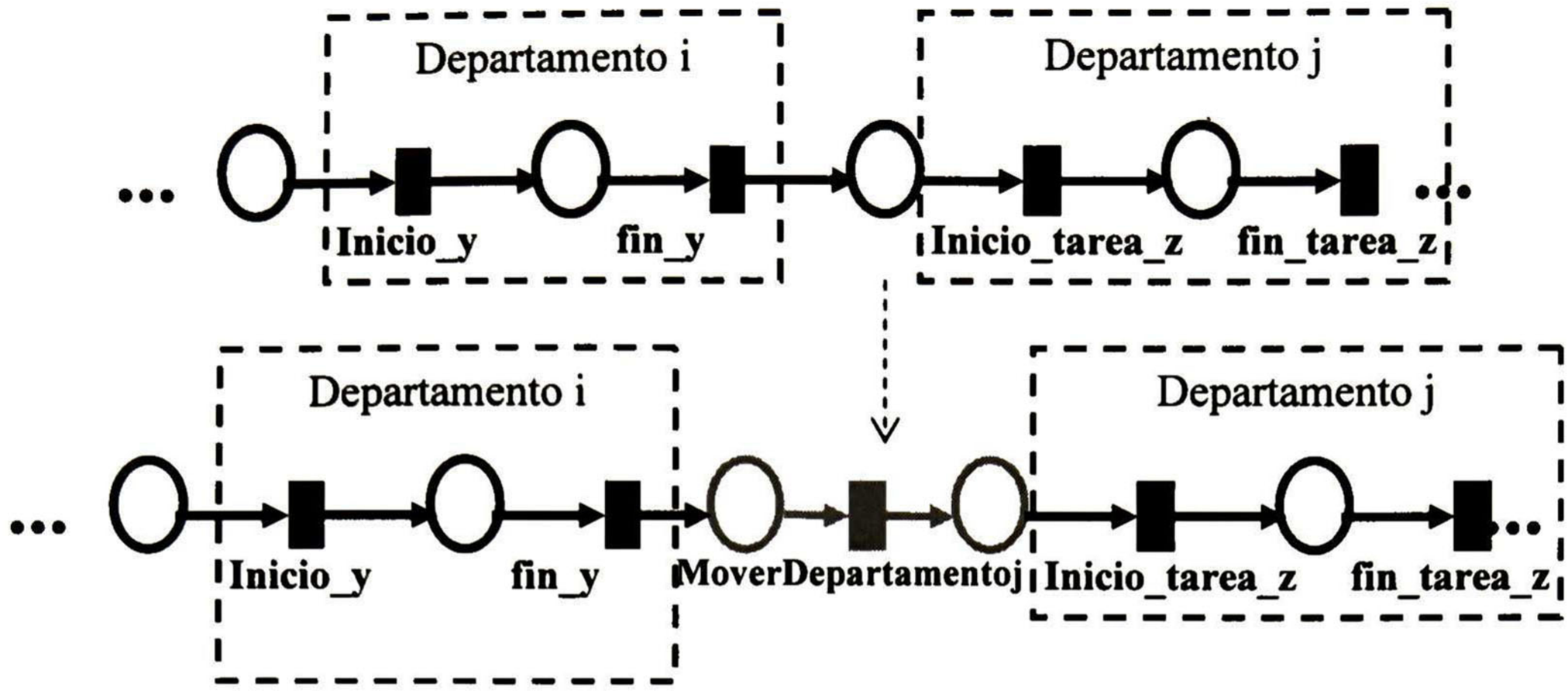


Figura 3.28: Sustitución de un lugar del grafo g3,1 por un lugar-transición-lugar

Para nuestro grafo del ejemplo 4 se obtiene el grafo mostrado en la figura 3.29.

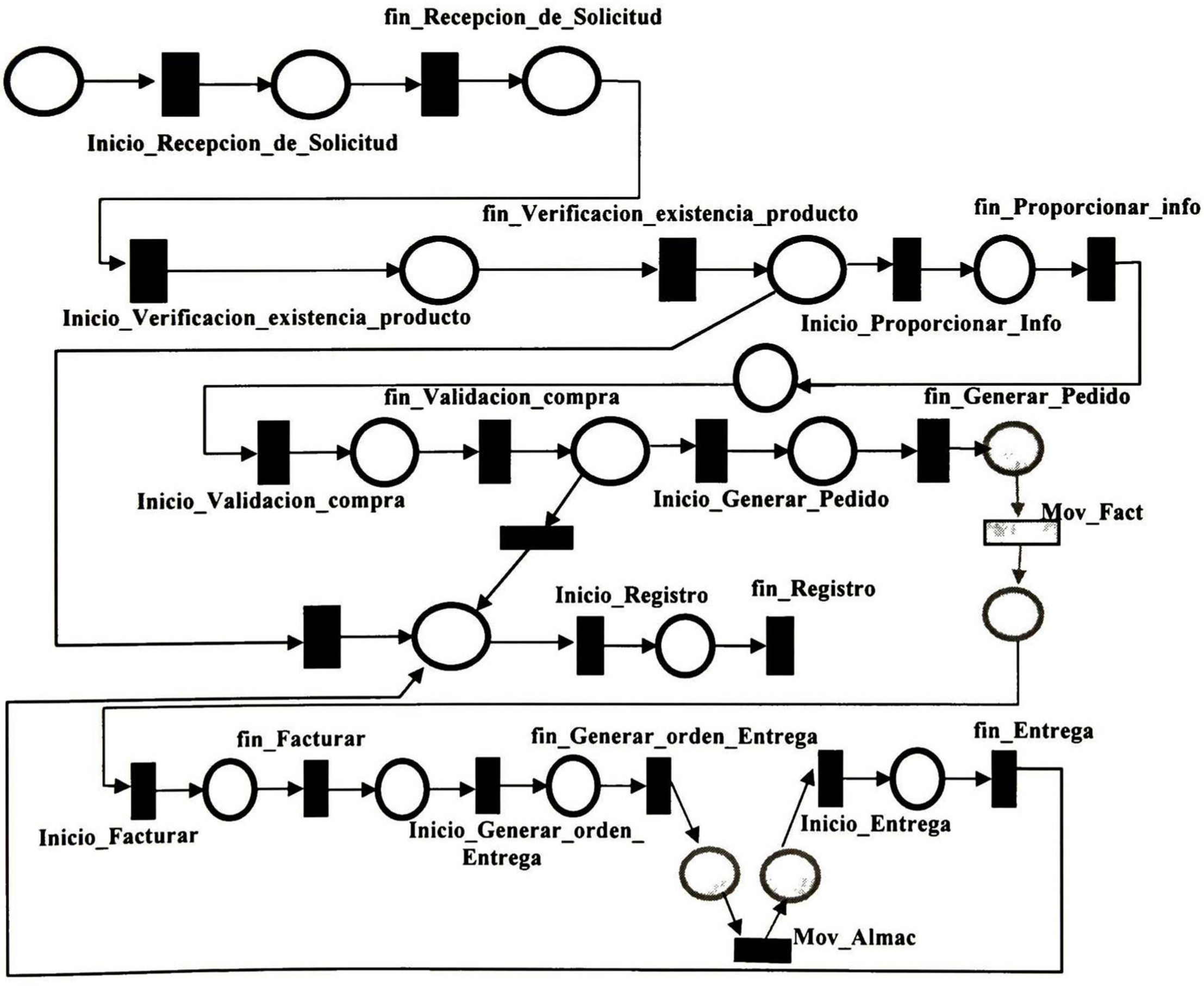


Figura 3.29: Grafo resultante, para el ejemplo 4, al agregar las transiciones para cambiar de departamentos

Por último, en el grafo obtenido se enumeran los lugares y las transiciones obteniendo el grafo G=(P,T,F) de la red $Net_{3,1}$. Donde P={pi | $0 < i \leq n$, n es el número de

lugares del grafo $g_{3,1}$ }, T={ti | 0 < i ≤ m, m es el número de transiciones del grafo $g_{3,1}$ }, y F se forma de las relaciones de flujo observadas en el grafo $g_{3,1}$.

b. Encontrar el valor del resto de los elementos de la $Net_{3,1}$ =(G, $TOKEN_{3,1}$,$LABEL_{3,1}$,$VAR_{3,1}$, τ, λ, χ, π)

o $TOKEN_{3,1}$. Como estamos en una red del nivel más interno, los lugares solo admitirán símbolos, por lo que agregaremos tantos símbolos a este conjunto como sean necesarios para nuestra red. Por lo que este conjunto se definiría como: $TOKEN_{3,1}$={s1, s2,..., s_n} donde n es el número de símbolos definidos.

Para el ejemplo 4 solo utilizaremos 2 símbolos, por lo tanto, $TOKEN_{3,1}$={s1, s2}

o $LABEL_{3,1}$, este conjunto se forma con el nombre de todas las etiquetas de la red, por lo que antes de definirlo será necesario revisar nuestro grafo G y verificar si todas las transiciones están etiquetadas (ya que en el inciso a) anterior definimos a la mayoría de las etiquetas), de lo contrario agregar las etiquetas correspondientes tratando siempre de poner un nombre de etiqueta representativo a lo que se quiere modelar. Por lo tanto:

$LABEL_{3,1}$={$Inicio\_tk_1$, fin_tk_1, $Inicio\_tk_2$, fin_tk_2,..., $Inicio\_tk_i$, fin_tk_i,.., $MovDepartamento_2$, $MovDepartamento_3$,..., $Mover\_Departamento_N$} ∪ {resto_de_las_etiquetas_definidas}

o $VAR_{3,1}$=∅ debido a que los tokens permitidos solo son símbolos.

o τ: si todas las transiciones de la red G, solo poseen un lugar de entrada y un lugar de salida, entonces esta función se definiría como: τ(p1)= τ(p2)=..= τ(pn)=s1, donde n es el número total de lugares de la red G y s1 es el símbolo definido en $TOKEN_{3,1}$.

Si en la red existen un lugar en el cual deberá de elegirse, de entre varios caminos, un solo camino de la red (es decir se tiene un lugar con múltiples salidas) se recomienda utilizar un símbolo diferente para cada bifurcación, por lo que para este lugar el τ(p)={s1,s2,....,s_n} según se necesite.

En general se recomienda que esta función se defina como mejor convenga al modelador.

o λ. Debido a que ya se agregaron la mayoría de las etiquetas básicas, deberemos mencionar el tipo de atributos que dichas etiquetas deberán tener y para esto separaremos las etiquetas por grupos:

- MovDepartamento$_i$, estas etiquetas deberán de tener el atributo de sincronización externa ya que esta red estará sincronizada con la red del nivel 2.
- Inicio_tarea_i, estas etiquetas deberán de tener sincronización externa. Debido a que esta red se sincronizará con la red de nivel 2.
- Fin_tarea_i, a estas etiquetas les asignaremos el atributo de sincronización externa y local. Debido a que para estas etiquetas, esta red se sincronizará eternamente con la red del nivel 2 y localmente con otra red de nivel 3.
- El resto de las etiquetas no deberá contar con elementos de sincronización.

Por lo tanto esta función se obtiene directamente del grafo G.

o χ(ti, fin_tarea_i≡)= (2,2) donde ti es la transición i donde está la etiqueta fin_tarea_i a la cual se le definió que tiene sincronización local. Por el valor dado a esta función, nos podremos dar cuenta de que esta red se sincronizará localmente solamente con otra red (que como veremos más adelante será una red del mismo nivel que nos modelará a las tareas individualmente y con esta etiqueta sincronizaremos la terminación de dicha tarea).

o π. Esta función la definirá el modelador según se haya construido la red del grafo G.

Para nuestro ejemplo 4 tenemos:

$LABEL_{3,1}$ ={Inicio_Recepcion_de_Solicitud, fin_Recepcion_de_Solicitud, Inicio_Verificación_existencia_producto, fin_Verificación_existencia_producto1, fin_Verificación_existencia_producto2 Inicio_Proporcionar_info, fin_Proporcionar_info, Inicio_Validación_compra, fin_Validación_compra1, fin_Validación_compra2, Inicio_Generar_Pedido, fin_Generar_Pedido, Mov_Fact, Inicio_Facturar, fin_Facturar, Inicio_Generar_orden_Entrega, fin_Generar_orden_Entrega, MovAlmac, Inicio_Entrega, fin_Entrega, Inicio_Registro, fin_Registro, No_valida, No_compra}

$VAR_{3,1}$ =∅

τ (p1)= τ (p2)= τ (p3)= τ (p4)= τ (p5)= τ (p6)= τ (p7)= τ (p8)= τ (p9)= τ (p10) = τ (p11)= τ (p12)= τ (p13)= τ (p14)= τ (p15)= τ (p16)= τ (p17)=

τ (p18)={s1} τ (p19)= τ (p20)= {s2}

λ(t1)={Inicio_Recepcion_de_Solicitud↑},

λ(t2)={fin_Recepcion_de_Solicitud↑≡},

λ(t3)={Inicio_Verificación_existencia_producto ↑},

λ(t4)={fin_Verificación_existencia_producto1↑≡}

λ(t4)={fin_Verificación_existencia_producto2↑≡}

λ(t5)={Inicio_Proporcionar_info ↑ },

λ(t6)={fin_Proporcionar_info ↑ ≡ },

λ(t7)={Inicio_Validación_compra ↑ },

λ(t8)={fin_Validación_compra1 ↑ ≡ },

λ(t8)={fin_Validación_compra2 ↑ ≡ },

λ(t9)={Inicio_Generar_Pedido ↑ },

λ(t10)={fin_Generar_Pedido ↑ ≡ },

λ(t11)={Mov_Fact ↑ },

λ(t12)={Inicio_Facturar ↑ },

λ(t13)={fin_Facturar ↑ ≡ },

λ(t14)={Inicio_Generar_orden_Entrega ↑ },

λ(t15)={fin_Generar_orden_Entrega ↑ ≡ },

λ(t16)={MovAlmac ↑ },

λ(t17)={Inicio_Entrega ↑ },

λ(t18)={fin_Entrega ↑ ≡ },

λ(t19)= {Inicio_Registro ↑ },

λ(t20)={fin_Registro ↑ ≡ },

λ(t21)= {No_valida},

λ(t22)= {No_compra}

χ(t2,fin_Recepcion_de_Solicitud↑≡)=

χ(t4,fin_Verificación_existencia_producto1↑≡)=
χ(t4,fin_Verificación_existencia_producto2↑≡)=

χ (t6,fin_Proporcionar_info ↑)=
χ(t8,fin_Validación_compra1 ↑ ≡)=
χ(t8,fin_Validación_compra2 ↑ ≡)=
χ(t10,fin_Generar_Pedido ↑ ≡)=
χ(t13,fin_Facturar ↑ ≡)=
χ(t15,fin_Generar_orden_Entrega ↑ ≡)=
χ(t18,fin_Entrega ↑ ≡)= χ(t20,fin_Registro ↑ ≡)= (2,2)

π((p1,t1), Inicio_Recepcion_de_Solicitud↑)={s1}
π((t1,p2),Inicio_Recepcion_de_Solicitud↑) = {s1}

π((p2,t2), fin_Recepcion_de_Solicitud↑≡)= {s1}

π((t2,p3), fin_Recepcion_de_Solicitud↑≡)= {s1}

π((p3,t3), Inicio_Verificación_existencia_producto ↑)= {s1}

π((t3,p4), Inicio_Verificación_existencia_producto ↑)= {s1}

π((p4,t4), fin_Verificación_existencia_producto1 ↑≡)={s1}

π((p4,t4), fin_Verificación_existencia_producto2 ↑≡)={s1}

π((t4,p5),fin_Verificación_existencia_producto1↑≡)={s1}

π((t4,p5),fin_Verificación_existencia_producto2↑≡)={s1}

π((p5,t5), Inicio_Proporcionar_info ↑)={s1}

π((t5,p6), Inicio_Proporcionar_info ↑)={s1}

π((p6,t6), fin_Proporcionar_info ↑ ≡)={s1}

π((t6,p7), fin_Proporcionar_info ↑ ≡)={s1}

π((p7,t7), Inicio_Validación_compra ↑)={s1}

π((t7,p8), Inicio_Validación_compra ↑)={s1}

π((p8,t8), fin_Validación_compra1 ↑ ≡)={s1}

π((t8,p9), fin_Validación_compra1 ↑ ≡)={s1}

π((p8,t8), fin_Validación_compra2↑ ≡)={s1}

π((t8,p9), fin_Validación_compra2↑ ≡)={s1}

π((p9,t9), Inicio_Generar_Pedido ↑)={s1}

π((t9,p10), Inicio_Generar_Pedido ↑)={s1}

π((p10,t10), fin_Generar_Pedido ↑ ≡)={s1}

π((t10,p11), fin_Generar_Pedido ↑ ≡)={s1}

π((p11,t11), Mov_Fact ↑)={s1}

π((t11,p12), Mov_Fact ↑)={s1}

π((p12,t12), Inicio_Facturar ↑)={s1}

π((t12,p13), Inicio_Facturar ↑)={s1}

π((p13,t13), fin_Facturar ↑ ≡)={s1}

π((t13,p14), fin_Facturar ↑ ≡)={s1}

π((p14,t14),Inicio_Generar_orden_Entrega↑)={s1}

π((t14,p15),Inicio_Generar_orden_Entrega↑)={s1}

π((p15,t15), fin_Generar_orden_Entrega↑≡)={s1}

π((t15,p16),fin_Generar_orden_Entrega↑ ≡)={s1}

π((p16,t16), MovAlmac ↑)={s1}

π((t16,p17), MovAlmac ↑)={s1}

π((p17,t17), Inicio_Entrega ↑)={s1}

π((t17,p18), Inicio_Entrega ↑)={s1}

π((p18,t18), fin_Entrega ↑ ≡)={s1}

π((t18,p19), fin_Entrega ↑ ≡)={s2}

π((p19,t19), Inicio_Registro ↑)={s2}

π((t19,p20), Inicio_Registro ↑)={s2}

π((p20,t20), fin_Registro ↑ ≡)={s2}

π((p9,t22), No_compra)={s2}

π((t22,p19), No_compra)={s2}

π((p5,t21), No_valida)={s2}

π((t21,p19), No_valida)={s2}

Se recomienda agregar un fragmento de red lugar1_transición_lugar2 a la red $Net_{3,1}$. Se deberán unir todas las transiciones finales de la red en un solo lugar, la transición la etiquetaremos como la última transición de la red de nivel 1 $Net_{1,1}$ (etiqueta *Mover*) y le pondremos atributos de sincronización externa con el objeto de sincronizar la terminación del caso con el ambiente.

- *Modelado de las redes de cada Tarea involucrada*

En esta sección se construirán las redes que modelarán las operaciones involucradas en la ejecución de cada una de las tareas que aparecen en Wi, es decir, definiremos las redes $Net_{3,1}$, $Net_{3,3}$,....,$Net_{3,n+1}$ donde n es el número total de tareas en el workflow.

Para el modelado de cada una de las tareas se deben seguir las siguientes recomendaciones.

1. La red deberá modelar cada una de las operaciones involucradas en la ejecución de la tarea, tan detallado como sea necesario, así como su orden de ejecución.
2. El flujo deberá regresarse siempre a su estado inicial
3. Deberán incluirse en el modelo transiciones que se etiquetarán como *Inicio_tarea*$_i$ y *fin_tarea*$_i$, es decir, se deberán utilizar dos transiciones para enfatizar el inicio de la ejecución de la tarea y terminación de la ejecución de la tarea, de la misma forma como se hizo para la construcción de la red $Net_{3,1}$.
4. Las transiciones *Inicio_tarea*$_i$, deberán tener el atributo de sincronización externa.
5. Las transiciones *fin_tarea*$_i$, deberán tener los atributos de sincronización tanto externa como local, por lo que la definición de la función χ para estas transiciones deberá ser: χ(ti, fin_tarea_i↑≡)= (2,2).
6. Etiquetar el resto de las transiciones modeladas utilizando una etiqueta representativa. Estas transiciones no poseerán atributos de sincronización.
7. Los lugares de la red solo admitirán símbolos por ser una red del nivel más interno, por lo que deberán utilizarse tantos símbolos como el modelador requiera.
8. Definir la red: $Net_{3,k}$ =(G, $TOKEN_{3,k}$,$LABEL_{3,k}$,$VAR_{3,k}$, τ, λ, χ, π), donde k= 2,3,...,n+1 siendo n el número de tareas contenidas en W_i,
9. Definir, según se haya modelado, los valores de las variables del punto anterior (ver figura 3.30).

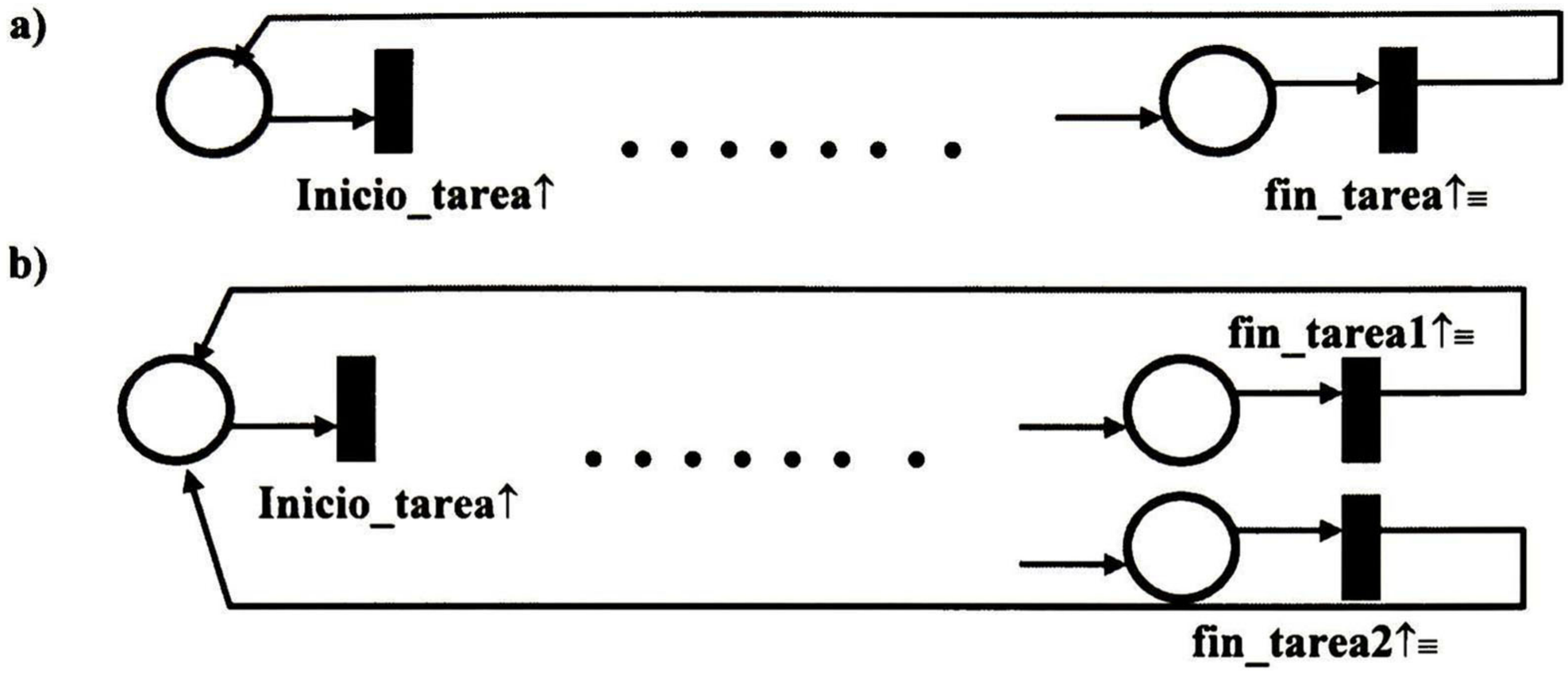
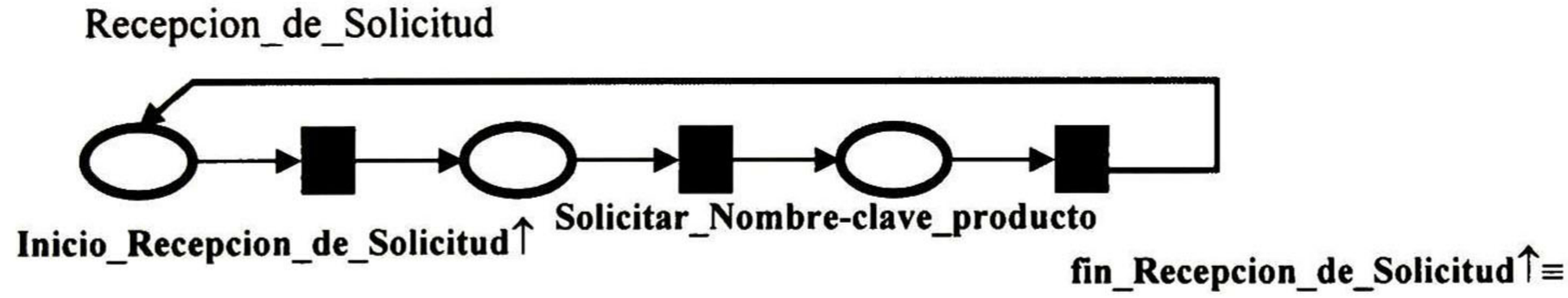


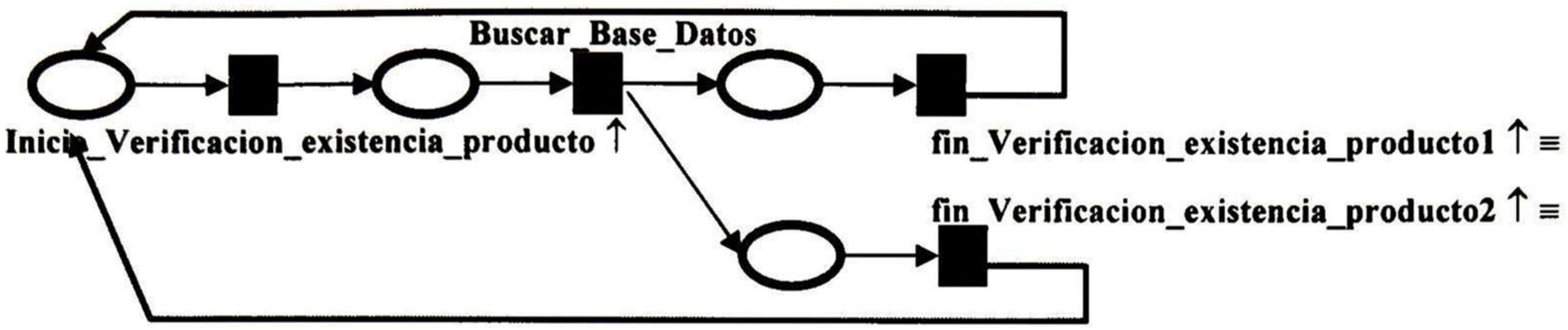
Figura 3.30: Partes principales del modelo de una tarea a) Si solo hay un lugar de salida, b) si se desea modelar dos condiciones diferentes de terminación de la tarea

Para el ejemplo 4 algunas de las redes serian $Net_{3,2}$ y $Net_{3,3}$ las cuales son mostradas en las figuras 3.31 y 3.32 respectivamente.



Para $Net_{3,2}$ tenemos:
$TOKEN_{3,2,}$={s1}
$LABEL_{3,2,}$={Inicio_Recepcion_de_Solicitud, Solicitar_Nombre-clave_producto, fin_Recepcion_de_Solicitud}
$VAR_{3,2,}$ =∅
τ(p1)= τ(p2)= τ(p3)={s1},
λ(t1)={ **Inicio_Recepcion_de_Solicitud↑**} λ(t2)={ **Solicitar_Nombre-clave_producto**}
λ(t3)={ **fin_Recepcion_de_Solicitud↑≡**}
χ,(t3, **fin_Recepcion_de_Solicitud↑≡**)=(2,2)
π((p1,t1), **Inicio_Recepcion_de_Solicitud↑**)= π((t1,p2), **Inicio_Recepcion_de_Solicitud↑**)= π((p2,t2), **Solicitar_Nombre-clave_producto**)= π((t2,p3), **Solicitar_Nombre-clave_producto**)= π((p3,t3), **fin_Recepcion_de_Solicitud↑≡**)= π((t3,p1), **fin_Recepcion_de_Solicitud↑≡**)=s1

Figura 3.31: $Net_{3,2}$ para el ejemplo 4



Para $Net_{3,3}$ tenemos:
$TOKEN_{3,3}$,={s1}
$LABEL_{3,3}$,={ **Inicio_Verificacion_existencia_producto , Buscar_Base_Datos, fin_Verificacion_existencia_producto1, fin_Verificacion_existencia_producto2** }
$VAR_{3,3}$, =∅
τ(p1)= τ(p2)= τ(p3)= τ(p4)={s1},
λ(t1)={ **Inicio_Verificacion_existencia_producto** ↑} λ(t2)={ **Buscar_Base_Datos**}
λ(t3)={ **fin_Verificacion_existencia_producto1** ↑ ≡} λ(t4)={ **fin_Verificacion_existencia_producto2**↑ ≡}
χ,(t3, **fin_Verificacion_existencia_producto1** ↑ ≡)= χ,(t4, **fin_Verificacion_existencia_producto2**↑ ≡)= (2,2)
π((p1,t1), **Inicio_Verificacion_existencia_producto**↑)= π((t1,p2),**Inicio_Verificacion_existencia_producto**↑)=
π((p2,t2), **Buscar_Base_Datos**)= π((t2,p3), **Buscar_Base_Datos**)= π((t2,p4), **Buscar_Base_Datos**)=π((p3,t3),
fin_Verificacion_existencia_producto1 ↑ ≡)= π((t3,p1), **fin_Verificacion_existencia_producto1**↑≡)= π((p4,t4),
fin_Verificacion_existencia_producto2 ↑ ≡)= π((t4,p1), **fin_Verificacion_existencia_producto1**↑≡)=s1

Figura 3.32: Net3,3 para el ejemplo 4

1.2.2 Construcción de la Red de Segundo nivel

Con esta red de segundo nivel modelaremos el comportamiento general del agente que se irá desplazando por los departamentos de la organización (mismos que fueron modelados por la red de nivel 1 o red ambiente), los tokens de esta red serán las redes tipo de nivel 3 definidas anteriormente por lo que se observan los siguientes puntos necesarios:

1. Es necesaria una transición para sincronizar el inicio de las tareas con las redes internas
2. Es necesaria una transición para sincronizar el desplazamiento entre los departamentos esta red tanto con las redes internas como con la red de nivel 1.
3. Una transición para sincronizar la terminación de las tareas con las redes internas.

La red propuesta se muestra en la figura 3.33.

- Los tokens permitidos en el lugar p1 son todas las redes tipo de nivel 3 que modelan a cada tarea (redes $Net_{3,2}$,$Net_{3,3}$,...$Net_{3,n+1}$).

- El tokens permitido en el lugar p2 es la red tipo de nivel 3 que modela el comportamiento del agente $Net_{3,1}$.

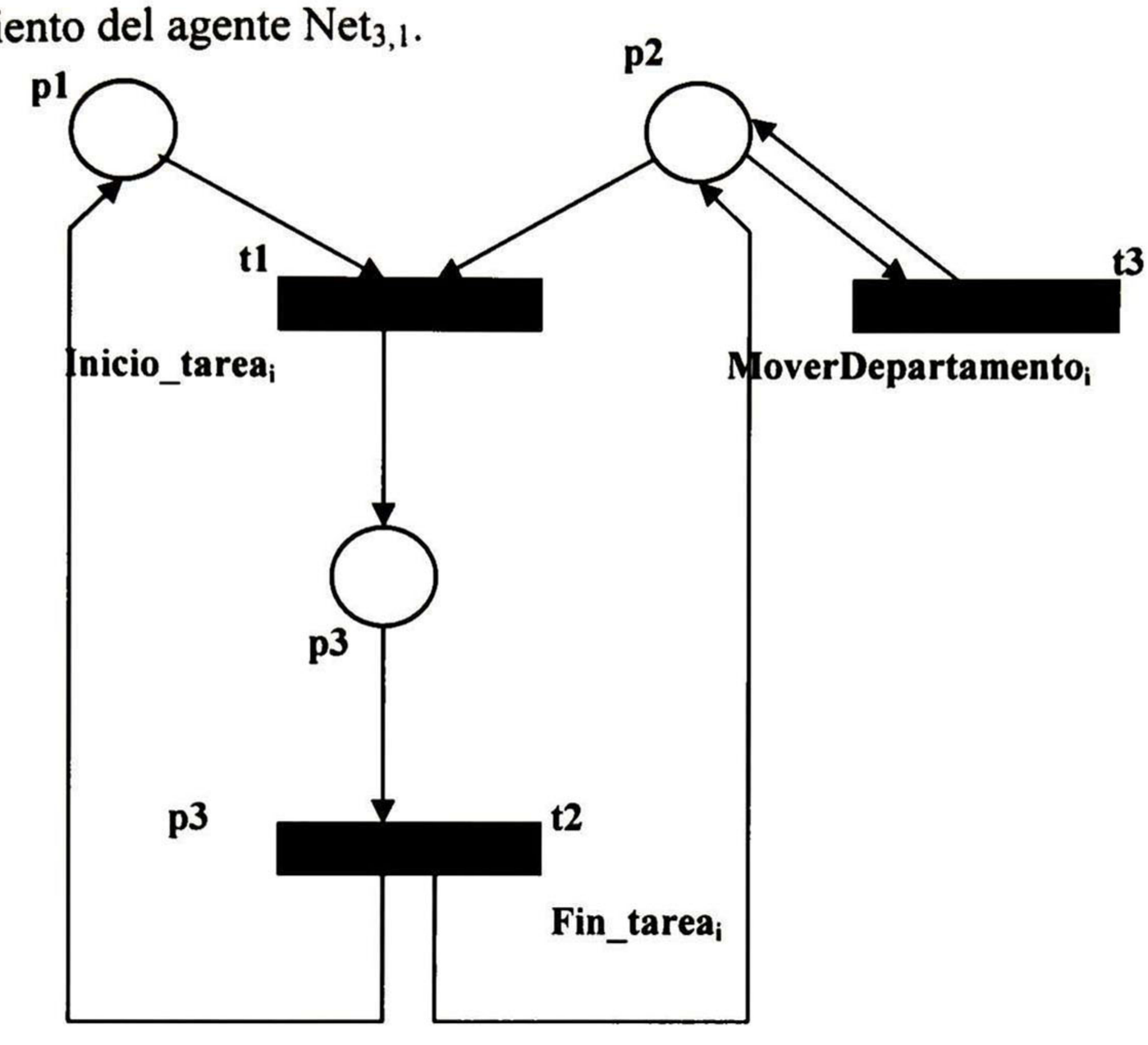


Figura 3.33: Red propuesta de nivel 2 (red Agente)

- Los token permitidos en el lugar p3 serán todas las redes tipo de nivel 3.
- La transición t1 deberá etiquetarse con las etiquetas de la red $Net_{3,1}$ que indican el inicio de cada tarea. Los atributos de sincronización de cada etiqueta serán ↑↓
- La transición t2 deberá etiquetarse con las etiquetas de la red $Net_{3,1}$ que indican el fin de cada tarea. Los atributos de sincronización de cada etiqueta serán ↑↓
- La transición t3 deberá etiquetarse con las etiquetas de la red $Net_{3,1}$ que indican el desplazamiento (cambio) de un departamento a otro.

Por lo que la red de nivel 2 para el ejemplo 4 se muestra en la figura 3.34.

1.2.3 Marcado de inicial de las redes

A continuación se explicará como van a marcarse inicialmente las redes de nuestro modelo:

1. $Net_{1,1}$: Esta red iniciará con un marcado vacío y al disparar la transición t0 se generará el token de tipo $Net_{2,1}$

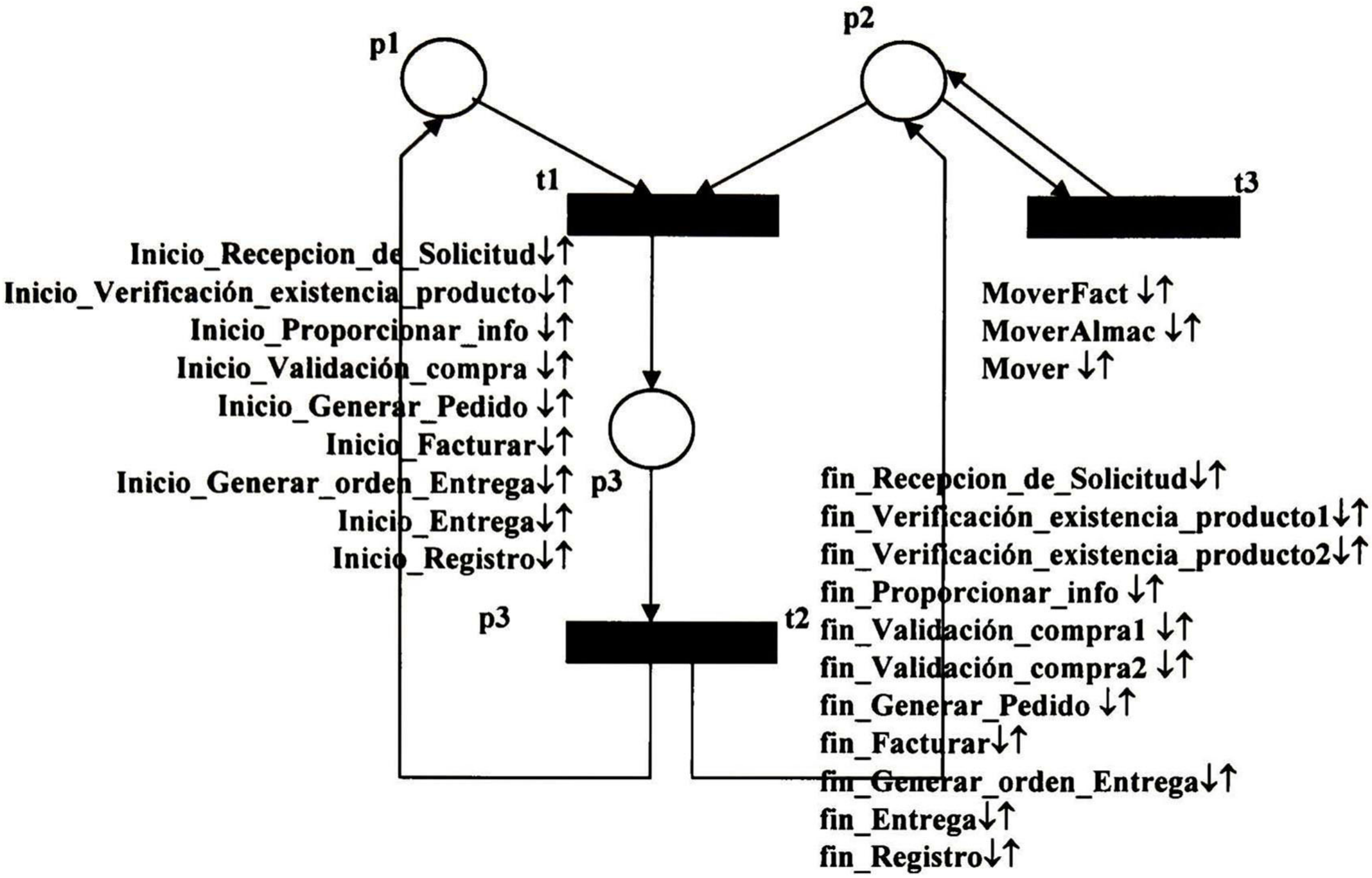


Figura 3.34: Net2,1 para el ejemplo 4

2. $Net_{3,1}$: Esta red tendrá el siguiente marcado μ(p1)={simbolo}

3. $Net_{3,k}$: Estas redes tendrán el siguiente marcado μ(p1)={simbolo}

4. $Net_{2,1}$: Esta red se marcará de la siguiente forma, μ(p1)={$Net_{3,2}$,$Net_{3,3}$,...,$Net_{3,n+1}$}donde n es el número total de tareas modeladas, es decir, el lugar 1 se marcará con los tokens correspondientes a las redes tipo de nivel 3 que modelan las tareas. μ(p2)={$Net_{3,1}$} es decir, el lugar 2 se marcará con el token correspondiente al modelo del comportamiento del agente.

Para el ejemplo 4, los marcados iniciales serán:

1. Para $Net_{3,1}$ el marcado inicial μ(p1)={s1}

2. Para las Redes $Net_{3,2}$, $Net_{3,3}$, $Net_{3,4}$, $Net_{3,5}$, $Net_{3,6}$, $Net_{3,7}$, $Net_{3,8}$, $Net_{3,9}$ y $Net_{3,10}$ el marcado inicial en todas ellas es μ(p1)={s1}

3. Para $Net_{2,1}$ el marcado inicial μ(p1)={ $Net_{3,2}$, $Net_{3,3}$, $Net_{3,4}$, $Net_{3,5}$, $Net_{3,6}$, $Net_{3,7}$, $Net_{3,8}$, $Net_{3,9}$, $Net_{3,10}$ }, μ(p2)={ $Net_{3,1}$}

CAPITULO IV.

Desarrollo basado en Agentes Móviles

Resumen. En este capítulo introduciremos la metodología para la definición sistemática de componentes de software para realizar una implementación de un sistema de gestión de flujo de trabajo utilizando las facilidades ofrecidas por JADE. La metodología, presentada utiliza los modelos obtenidos en el capítulo III.

4.1 Herramienta de programación JADE

JADE (Java Agent Development Framework) es un marco de trabajo de software completamente implementado en el lenguaje Java, el cual simplifica la implementación de sistemas Multi-agente a través del uso de un middleware que cumple con las especificaciones de FIPA. La plataforma agente puede ser distribuída a través de máquinas (las cuales no necesariamente necesitan compartir el mismo sistema operativo) y la configuración puede ser controlada por medio de una GUI remota. La configuración puede cambiar en tiempo de ejecución mediante el desplazamiento de agentes de una máquina a otra cuando esto sea requerido.

La arquitectura de comunicación ofrece un paso de mensajes flexible y eficiente, donde JADE crea y maneja la cola de mensajes ACL entrantes privados de cada agente. El modelo de comunicación FIPA completo ha sido implementado y sus componentes han sido claramente distinguidos e integra completamente: protocolos de interacción, ACL, ontologías, protocolos de transporte, entre otros. La mayoría de los protocolos definidos por FIPA están disponibles.

4.1.1 Generalidades

A continuación se listan un conjunto de características que ofrece JADE:

- Una plataforma de agentes distribuída. Esta plataforma puede dividirse en varios host (los cuales pueden conectarse vía RMI). Solamente una aplicación en Java y por lo tanto una sola máquina virtual es ejecutada en cada host.
- Interfaz gráfica de usuario para manejar varios agentes y contenedores desde un host remoto.
- Herramientas que ayudan en el desarrollo de aplicaciones multi-agentes basadas en JADE.
- Movilidad de agentes intra-plataforma, incluyendo transferencia de estado y código (cuando sea necesario) de cada agente.
- Soporte de ejecución de actividades múltiples, paralelas y concurrentes con la ayuda del modelo de comportamiento.
- Plataforma de Agente que cumple con FIPA, la cual incluye el AMS (Agent Management System), el DF (Directory Facilitator) y el ACC (Agent Communication Channel). Los cuales se activan una vez que la plataforma inicia.
- Transporte eficiente de mensajes ACL dentro de la misma plataforma, de hecho, los mensajes son transferidos codificados como objetos Java y no como cadenas.
- Librería de protocolos de interacción de FIPA.
- Servicio de nombres compatible con FIPA.
- Soporte de ontologías.

♦ *Contendores y Plataformas*

Cada instancia en ejecución del ambiente JADE se le conoce como *Container* (Contenedor)y éste puede contener a varios agentes. A el conjunto de contenedores activos se le llama *Plataform* (Plataforma). Un solo *Main container* (contenedor principal) debe estar siempre activo en una plataforma y todos los contenedores deben de registrarse con él tan pronto como inicien. El contenedor que inicie primero recibe el nombre de contenedor principal y el resto serán contenedores normales (no principales).

Si se inicia otro contenedor principal en algún lugar de la red, éste constituye una plataforma diferente. En la figura 4.1, se muestran estos conceptos.

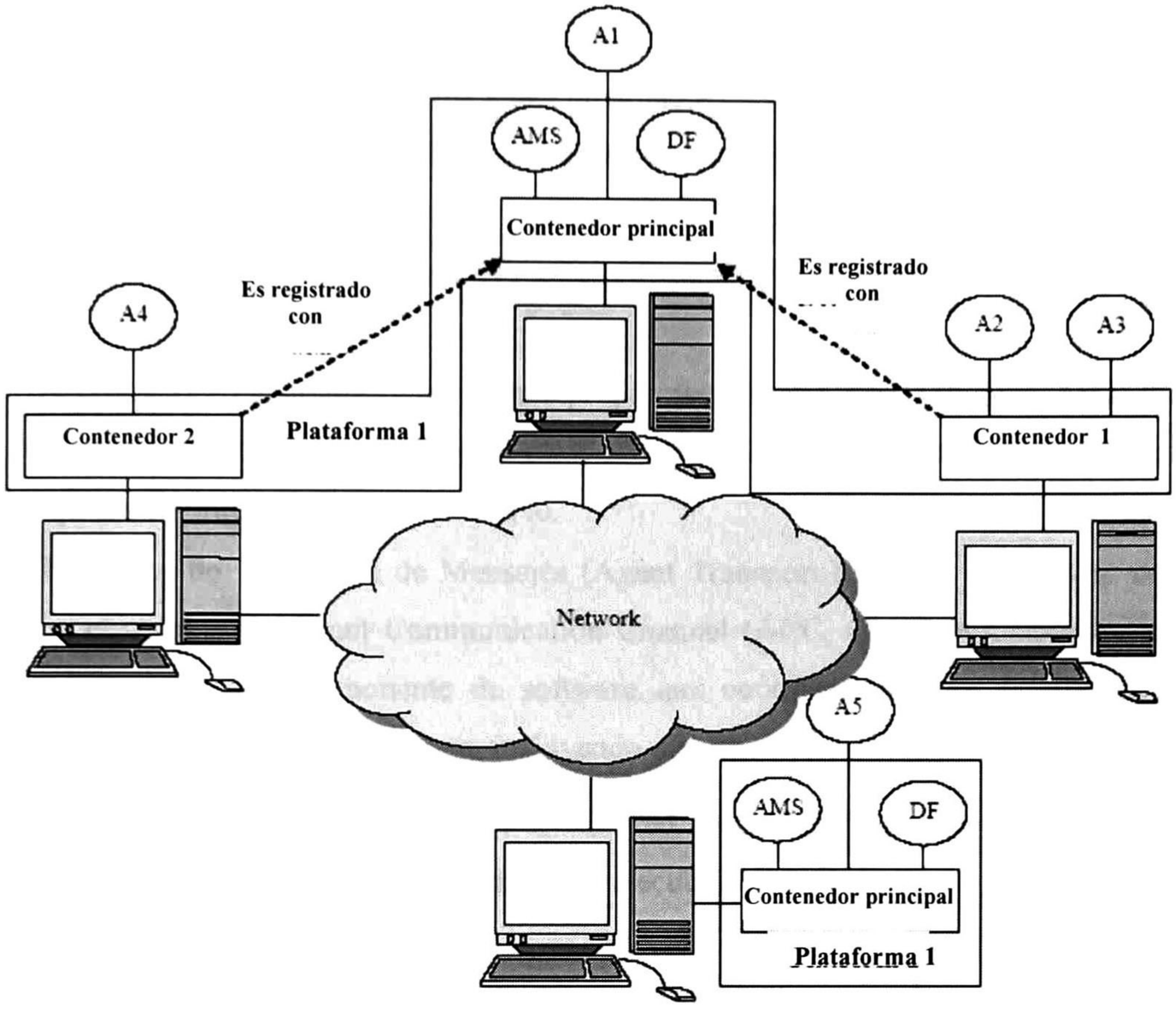


Figura 4.1: Contenedores y plataformas

- Plataforma Agente

El modelo estándar de una plataforma agente, como está definido en FIPA, se representa en la siguiente figura:

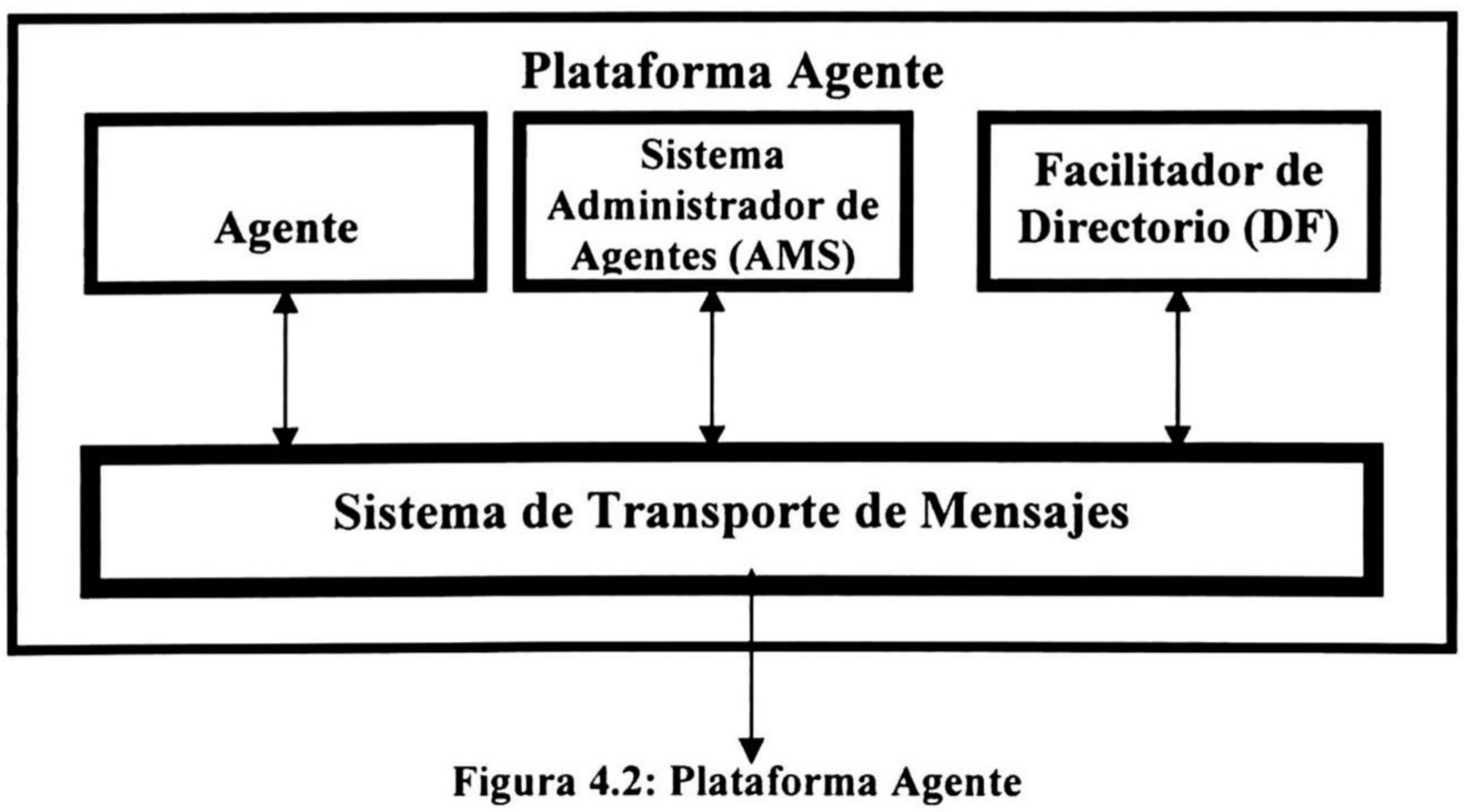


Figura 4.2: Plataforma Agente

- AMS (Agent Management System). Este agente provee el servicio de nombres (asegura que cada agente en la plataforma tenga un nombre único) y representa la autoridad en la plataforma (es posible crear/terminar agentes en contenedores remotos por medio de una petición a este agente). Este agente ejerce control sobre el acceso y uso de la plataforma Agente. Proporciona el servicio de páginas blancas y el servicio de ciclo de vida.
- DF (Directory Facilitator). Este agente proporciona el servicio de Páginas Amarillas en el cual un agente puede encontrar a otros agentes que proporcionen el servicio que éste requiera en orden de lograr su objetivo.
- El Sistema de Transporte de Mensajes (Agent Transport System), también se conoce con el nombre de Agent Communication Channel (ACC, Canal de Comunicación de Agentes). Es el componente de software que controla todos los intercambios de mensajes dentro de la plataforma, incluyendo mensajes de o desde plataformas remotas.

La plataforma Agente puede ser dividida en varios host, pero solo una aplicación en java y por lo tanto solo una máquina virtual es ejecutada en cada host.

- *Clase Agente*

La clase Agente representa una clase base común para agentes definidos por el usuario, en JADE, por otro lado, un agente es una simple instancia de una clase en Java definida por un usuario y que extiende la clase base *Agent*, esto implica herencia de características para lograr interacciones básicas con la plataforma Agente (registro, configuración, administración remota, etc.) y un conjunto básico de métodos que pueden ser llamados para implementar comportamientos de un agente (enviar o recibir mensajes, usar protocolos de interacción estándar, registro con varios dominios, etc.).

El modelo computacional de un agente es multitarea donde tareas (o comportamientos) son ejecutados concurrentemente. Cada servicio proporcionado por un agente debe ser implementado como uno o más comportamientos (behaviours).

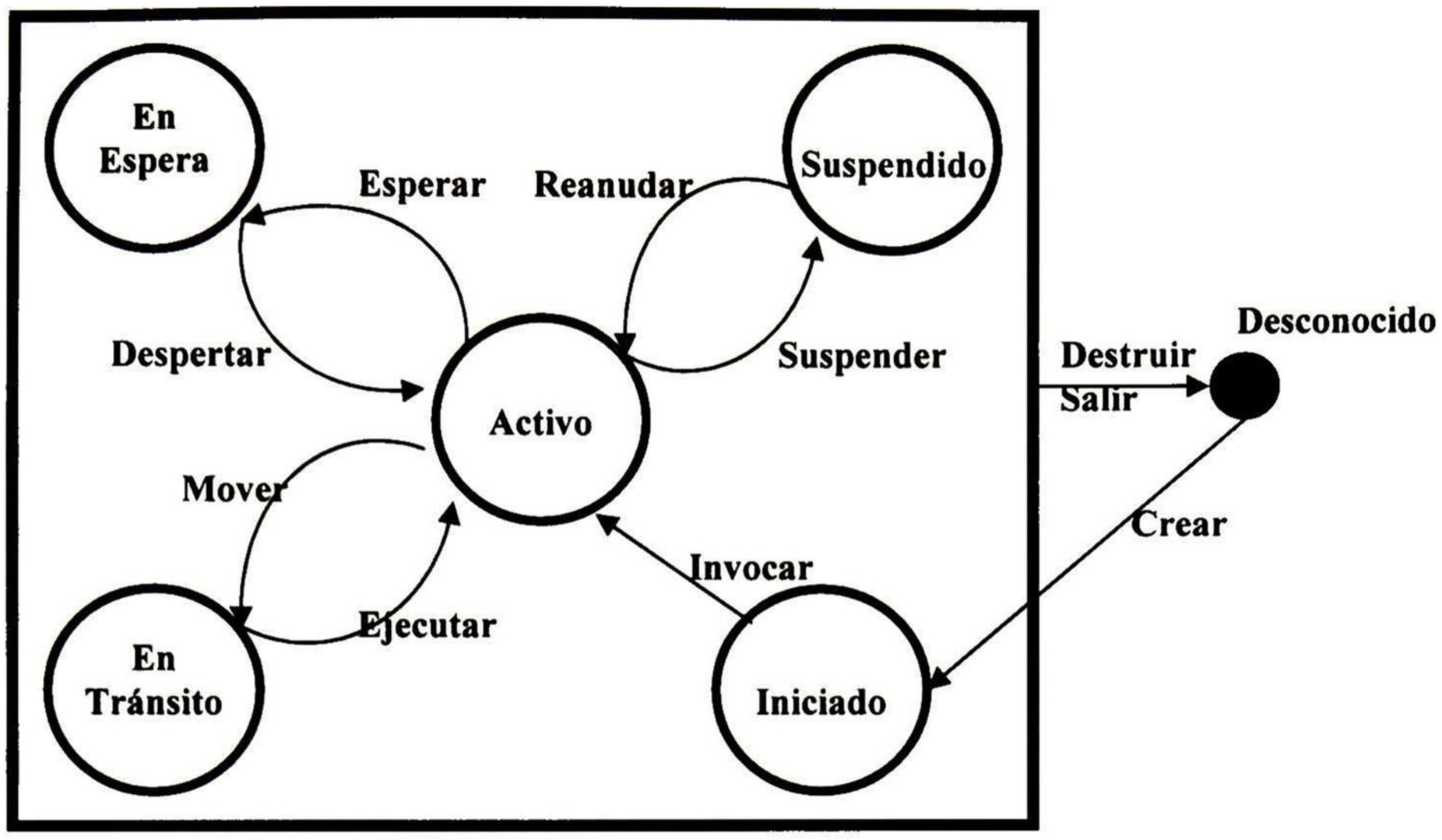


Figura 4.3: Ciclo de vida de un Agente como se definió en FIPA

INICIADO : El objeto agente es construido, pero aun no ha sido registrado consigo mismo al AMS, tampoco tiene un nombre o dirección y no puede comunicarse con otros agentes.

ACTIVO: El objeto agente es registrado con el AMS, tiene un nombre regular y dirección y puede acceder a todas las características de JADE

SUSPENDIDO : El objeto agente es detenido, su hilo interno es suspendido y ningún comportamiento del agente es ejecutada.

EN ESPERA : El objeto agente es bloqueado, y puesto en espera de un evento, su hilo interno se pone a dormir y es despertado cuando una condición conocida se cumple por ejemplo un mensaje llega.

ELIMIADO : Se elimina el agente. El hilo interno se termina y el agente ya no es registrado más con el AMS.

EN TRANSITO : Un agente móvil entra en este estado cuando esta migrando de a una nueva locacion.

El agente provee de varios métodos públicos los cuales permiten realizar la transición entre los diversos estados, estos métodos toman sus nombres de una transición conveniente en una maquina de estados finita; un ejemplo de esto es el método doWait() el

cual pone al agente en el estado WAITING del estado ACTIVE, para una explicación mas completa acerca de los métodos por refiérase a [JADE]. Cabe hacer mención que las tareas del agente solo son permitidas ser ejecutadas cuando se encuentra en el estado ACTIVE, por lo tanto si una tarea ejecuta el método doWait() bloqueara a todo el agente y no solamente la actividad que estaba realizando.

Para llevar acabo la ejecución de un agente se deben llevar a cabo una serie de pasos los cuales son:

Ejecutar el constructor del agente

Identificar el agente

Registrar el agente con el AMS

Poner el agente en el estado ACTIVO

De acuerdo con la especificación de FIPA, el identificador de un agente debe ser globalmente único y tiene la forma <nickname>@<platform-name> así por ejemplo un agente llamado *Peter* que se encuentra en la plataforma *P1* su nombre único seria *Peter@P1*. Cada nombre es representado como una instancia de la clase jade.core.AID y una manera para obtener el identificador del agente es haciendo uso del método getAID().

- Comunicación entre agentes

La comunicación entre agentes es una de las características mas importantes que JADE provee. El paradigma de comunicación adoptado es el de transmisión de mensaje asíncrono. Cada agente posee un buzón en el cual JADE coloca los mensajes enviados por otros agentes, siempre que un mensaje es colocado en el buzón el agente receptor es notificado de que tiene un mensaje nuevo.

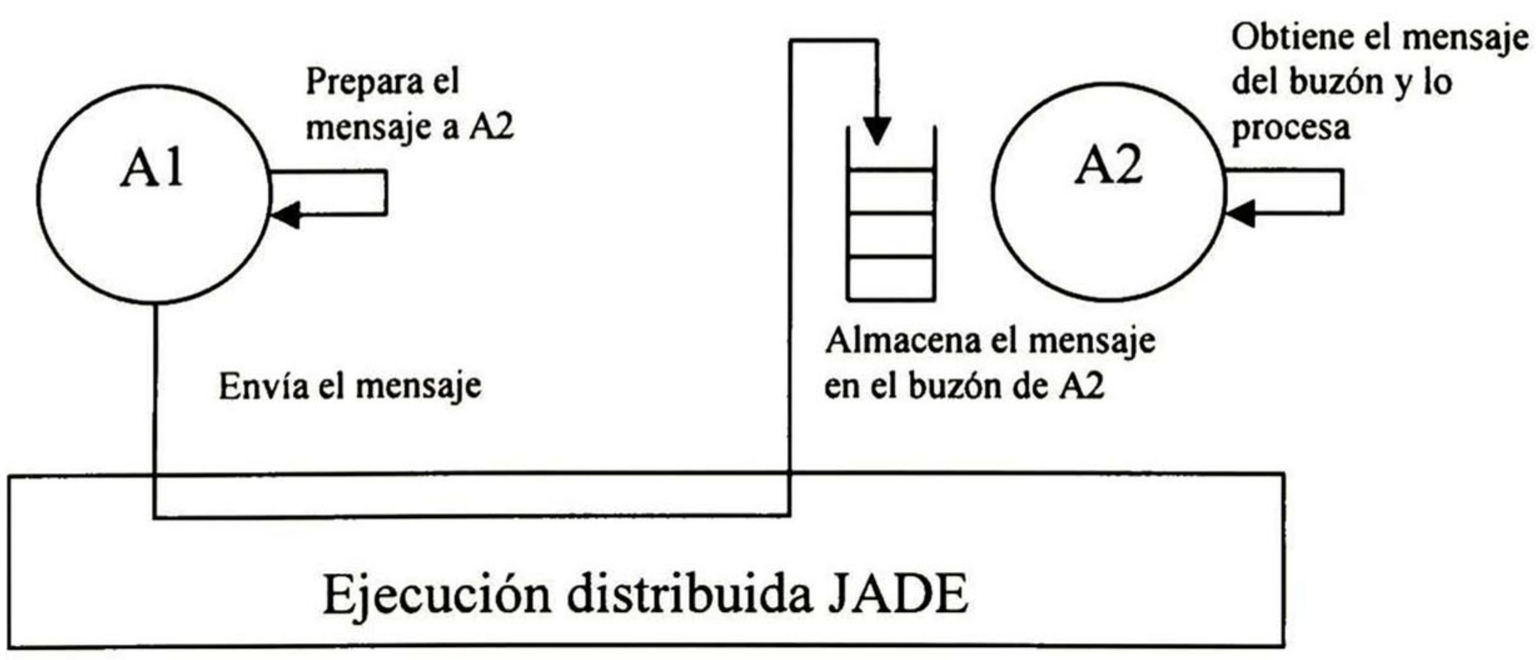


Figura 4.4: Esquema de comunicación entre agentes

Los mensajes intercambiados por los agentes tienen un formato especificado por el lenguaje ACL el cual esta definido por FIPA. Este formato comprende un número de campos y en particular los siguientes:

El remitente del mensaje

La lista de destinatarios

La intención de la comunicación, llamado también preformativa, indicando que pretende alcanzar el remitente al enviar el mensaje. Las preformativas pueden ser:

- REQUEST.- Si se desea que el destinatario realice una acción.
- INFORM.- Si se desea que el destinatario se de cuenta de algo.
- QUERY_IF.- Si el remitente quiere conocer si una condición se sigue manteniendo o no.
- CFP.- Si se desea realizar una propuesta.
- PROPOSE.- Propuesta.
- ACCEPT_PROPOSAL.- Aceptación de propuesta.
- REJECT_PROPOSAL.- Rechazo de propuesta.

El contenido, el cual es la información que se desea comunicar.

El lenguaje del contenido, el cual expresa la sintaxis utilizada para expresar el contenido.

La ontología, el cual es el vocabulario de símbolos usados en el contenido y su significado.

Campos de control, los cuales permiten controlar la conversación entre los agentes, i.e. tiempos de recepción y de replica.

A continuación se presenta el código necesario para enviar un mensaje, como ejemplo tomaremos que el destinatario tiene nickname *Peter* y le deseamos informar que *today it's raining.*

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-forecast-ontology");
msg.setContent("Today it's raining");
send(msg);
```

- *Comportamientos*

Un comportamiento representa una tarea que un agente puede realizar y es implementada como un objeto de una clase que extiende jade.core.behaviours.Behaviour. Para hacer que un agente ejecute una tarea implementada por un objeto de comportamiento es suficiente añadir el comportamiento al agente por medio del método addBehaviour de la clase Agente.

Los comportamientos pueden ser añadidos en cualquier tiempo. Un agente puede ejecutar varios comportamientos al mismo tiempo.

Jade utiliza una jerarquía de clases que representa los comportamientos que un agente puede realizar, esta jerarquía es mostrada en la figura 4.5. Estas clases son las siguientes:

OneShotBehaviour.- Puede ser ejecutada una sola vez y no puede ser bloqueada.

CyclicBehaviour.- Se ejecuta un número infinito de veces.

SimpleBehaviour.- Modela una tarea simple, i.e. una tarea que no esta compuesta de sub-tareas.

CompositeBehaviour.- Modela comportamientos que están hechos por la composición de otros comportamientos. Así las operaciones realizadas por este comportamiento no están definidas por este comportamiento en si, sino por los comportamientos de los cuales esta compuesto.

SequentialBehaviour.- Esta clase esta compuesta de comportamientos los cuales los ejecuta de manera secuencial siguiendo un orden definido por el programador.

ParallelBehaviour.- Esta clase esta compuesta también de comportamientos los cuales los ejecuta de manera concurrente pudiendo llagar a su terminación cuando un comportamiento alcanzo un valor establecido o todos los comportamientos hayan terminado.

FSMBehaviour.- Esta clase compuesta por comportamientos, los ejecuta de acuerdo a una maquina de estados finita definida por el usuario.

WakerBehaviour.- Esta clase implementa implementa una tarea ejecutada solo una vez la cual se realiza después de que un tiempo es alcanzado.

- TickerBehaviour.- Esta clase implementa una tarea cíclica que debe ser ejecutada periódicamente

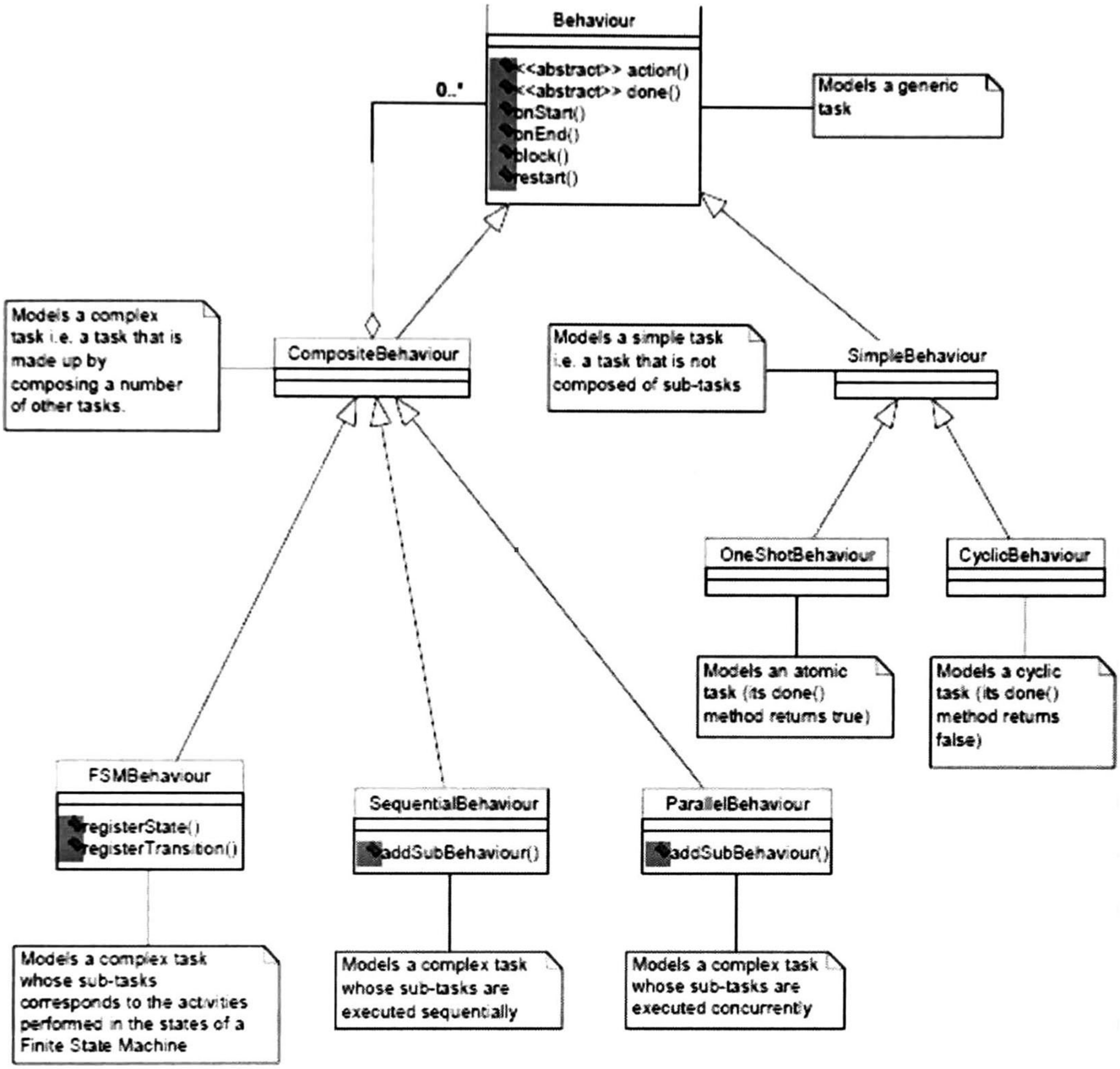


Figura 4.5: Diagrama de clases de los comportamientos soportados por JADE

Un ejemplo de la implementación de un comportamiento es mostrado en el siguiente código:

```
class my3StepBehaviour extends SimpleBehaviour {
        final int FIRST = 1;
        final int SECOND = 2;
        final int THIRD = 3;
        private int state = FIRST;
        private boolean finished = false;
public my3StepBehaviour(Agent a) {
        super(a);
}
public void action() {
        switch (state){
                case FIRST: {if (op1())
                                state = SECOND;
                                else
                                state= FIRST;
                                break;}
                case SECOND:{op2(); state = THIRD; break;}
                case THIRD:{op3(); state = FIRST; finished = true; break;}
                }
}
public boolean done() {
        return finished;
```

- *Protocolos*

Los protocolos son plantillas estándar especificadas por FIPA para establecer la conversación entre agentes. Para cada conversación entre agentes, JADE distingue entre el rol de *iniciador* (el agente que inicia la conversación) y el rol de *contestador* (el agente que contesta la conversación después de haber sido contactado por otro agente). JADE provee clases de comportamiento para ambos roles de la conversación para la mayoría de los protocolos de interacción. Algunos de los protocolos de interacción especificados por FIPA son:

FIPA-Request

FIPA-query

FIPA-Request-When

FIPA-recruiting

FIPA-brokering

FIPA-Contract-Net.- Este protocolo de interacción permite al *iniciador* enviar una llamada de oferta a un conjunto de *contestadores*, evalúa sus propuestas y entonces acepta la mejor o incluso puede rechazarlas todas.

FIPA-Propose.- Este protocolo permite al *iniciador* enviar un mensaje de propuesta a el participante, indicándole que desarrollara alguna acción si desea participar. El participante responde accediendo o rechazando la propuesta.

FIPA-Subscribe.- Este protocolo permite al *iniciador* enviar un mensaje de suscripción a los participantes, indicándoles si desean subscribirse. El participante procesa el mensaje de suscripción y responde aceptando o rechazando la suscripción.

- *Movilidad en Agentes*

Con JADE es posible crear agentes móviles, los cuales son capaces de migrar o copiarse a si mismos a través de múltiples hosts, solo es soportada la movilidad intra-plataforma, esto es una agente móvil de JADE puede navegar a través de diferentes contenedores de agentes pero es confinado a una sola plataforma de JADE. El movimiento o clonación es considerado un estado de transición en el ciclo de vida del agente, como cualquier otra operación del ciclo de vida de los agentes esta puede ser iniciada por el agente o por el AMS.

Los agentes móviles necesitan estar en una localidad conocida para poder decidir cuando y a donde moverse, para esto JADE provee una ontología, llamada *jade-mobility-ontology*, la cual contiene los conceptos y acciones necesarios. Jade provee la clase jade.domain.mobility.MobilityOntology, que trabajando como un singleton y dando acceso a uno solo, comparte instancias de la ontología de movilidad de JADE por medio del método getInstance().

Además JADE provee de dos métodos públicos doMove() y doClone() los cuales permite al agente migrar o copiarse en un lugar remoto bajo un nombre diferente. El método doMove() toma a jade.core.Location como su parámetro, el cual representa el destino a donde va a migrar el agente. El método doClone() también tomo jade.core.Location como parámetro, pero además añade un String el cual contiene el nombre del nuevo agente que será creado. Los agentes por si mismos no están permitidos para crear sus locaciones, en vez de eso ellos deben preguntar al AMS por la lista de locaciones y escoger una.

4.2 Técnica de codificación

En esta sección mostraremos cómo pasar de nuestro modelo multivel del problema de flujo de trabajo, a una aplicación basada en agentes móviles utilizando la herramienta de programación JADE, para esto definiremos una estrategia general:

- Red de nivel 1.- Con la ayuda de esta red definiremos la plataforma de agentes
- Red de nivel 2 y Red $Net_{3,1}$.- Nos servirán para programar a el agente
- Redes que modelan tareas.- Nos ayudarán a definir los comportamientos del agente.

4.2.1 Metodología

- *Plataforma de agentes*

Como ya se mencionó antes, para definir la plataforma de agentes utilizaremos nuestra red de nivel 1, esta red (en terminología de agentes), nos representa el ambiente en el cual se desplaza el agente y realiza sus tareas para lograr su fin, es decir, nos representa una plataforma de agentes en JADE. Cada lugar de esta red nos simboliza un contenedor de la plataforma, cada contenedor puede suponerse que se encuentra en un host distinto de una red por lo que cada lugar también simboliza un host (sitio) distinto en una red. Los host también pueden pertenecer a plataformas distintas aunque para efectos prácticos supondremos una misma plataforma

En esta red ambiente, al permitir la red agente como marcado en los lugares, cada transición de la red representará la migración de dicho agente de un lugar a otro (de un sitio a otro) dentro del ambiente.

Los pasos a seguir para definir un modelo de la plataforma son:

1. Sustituir cada lugar de la red Ambiente por un host (sitio).

Para la Red ambiente $Net_{1,1}$ obtenida para el ejemplo 4, se obtendría el modelo de la figura 4.6.

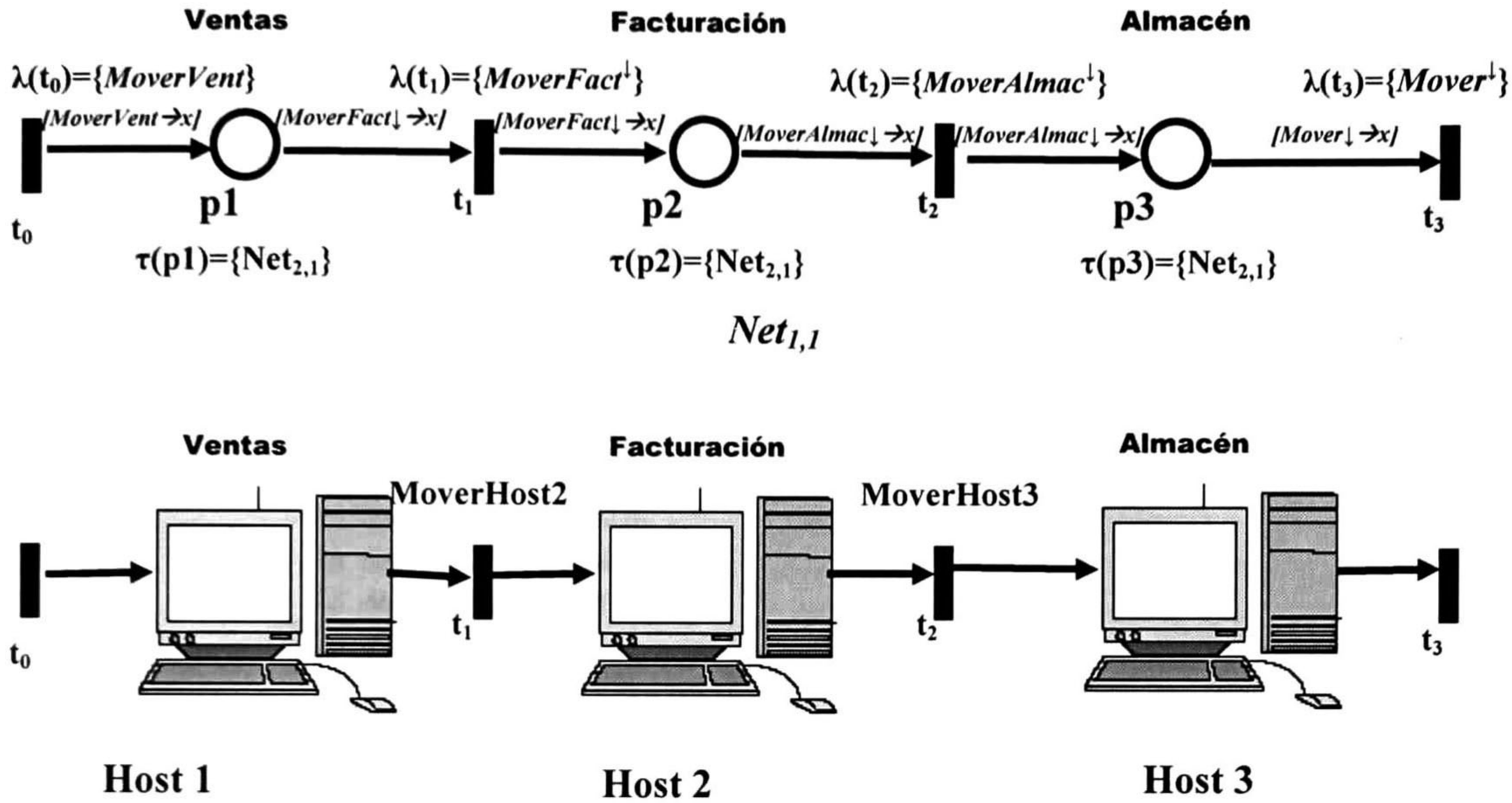


Figura 4.6: Analogía de los lugares de una Red a Sitios de un sistema en red

2. En cada host deberá de crearse un Contenedor, en donde se ejecutarán los agentes. Se deberá decidir cuál de los host contendrá al contenedor principal y por lo tanto se decidirá en cual lugar se registrarán los agentes.

   La creación de los contenedores se realiza de la siguiente manera:

   - Contenedor principal: en la línea de comandos del sitio donde se contendrá dicho contenedor, se puede escribir (de manera general):

     C:\ java jade.Boot -container-name Nombre_Host [-gui]

     Los corchetes [] significan que lo que está dentro de ellos e opcional; si se decide poner *–gui* (interfaz grafica de usuario), se abrirá una ventana de aplicación de jade en la que podremos monitorear, crear, eliminar ,etc, los agentes.

   - Resto de los contenedores: en la línea de comandos de cada sitio escribir (de manera general):

C:\ java jade.Boot -container-name Nombre_Host -container –host HostDelContenedorPrincipal

- Para la creación de un agente en un contenedor puede utilizarse la gui de Java como lo indica la siguiente figura:

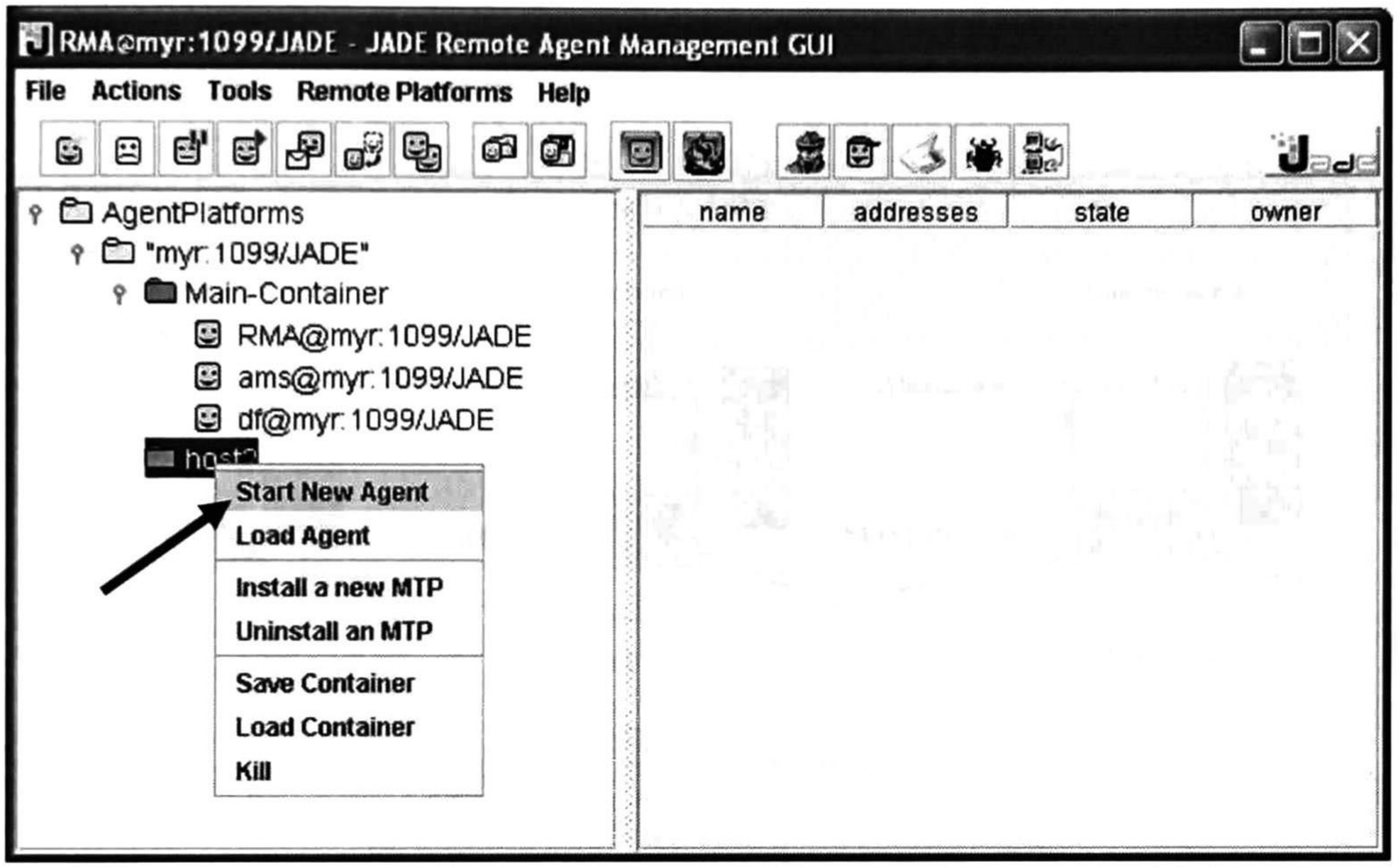


Figura 4.7: Interfaz Grafica de Jade

Con esto iniciaremos un nuevo agente en el host2 el cual debió crearse previamente.

Después se debe colocar el nombre del agente y nombre de la clase dentro de la ventana:

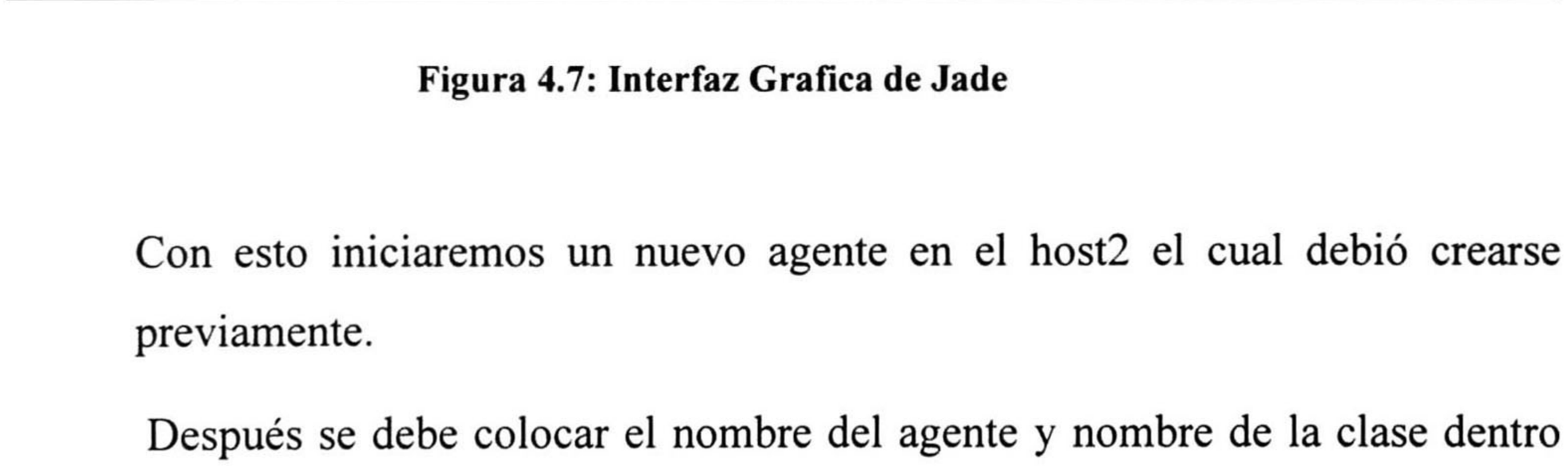


Figura 4.8: Ventana para crear un agente desde la GUI de jade

3. Cada transición de la red ambiente simboliza la migración del agente a un contenedor distinto de la plataforma, por lo que si un lugar en la red ambiente contiene una marca, significa que un agente está ejecutándose en el host que representa dicho lugar y si se dispara la transición que une al este host con el host 2 (por ejemplo), el agente se moverá del host actual al host 2.

Para el ejemplo, suponiendo que el host 1 contendrá al contenedor principal, seria:

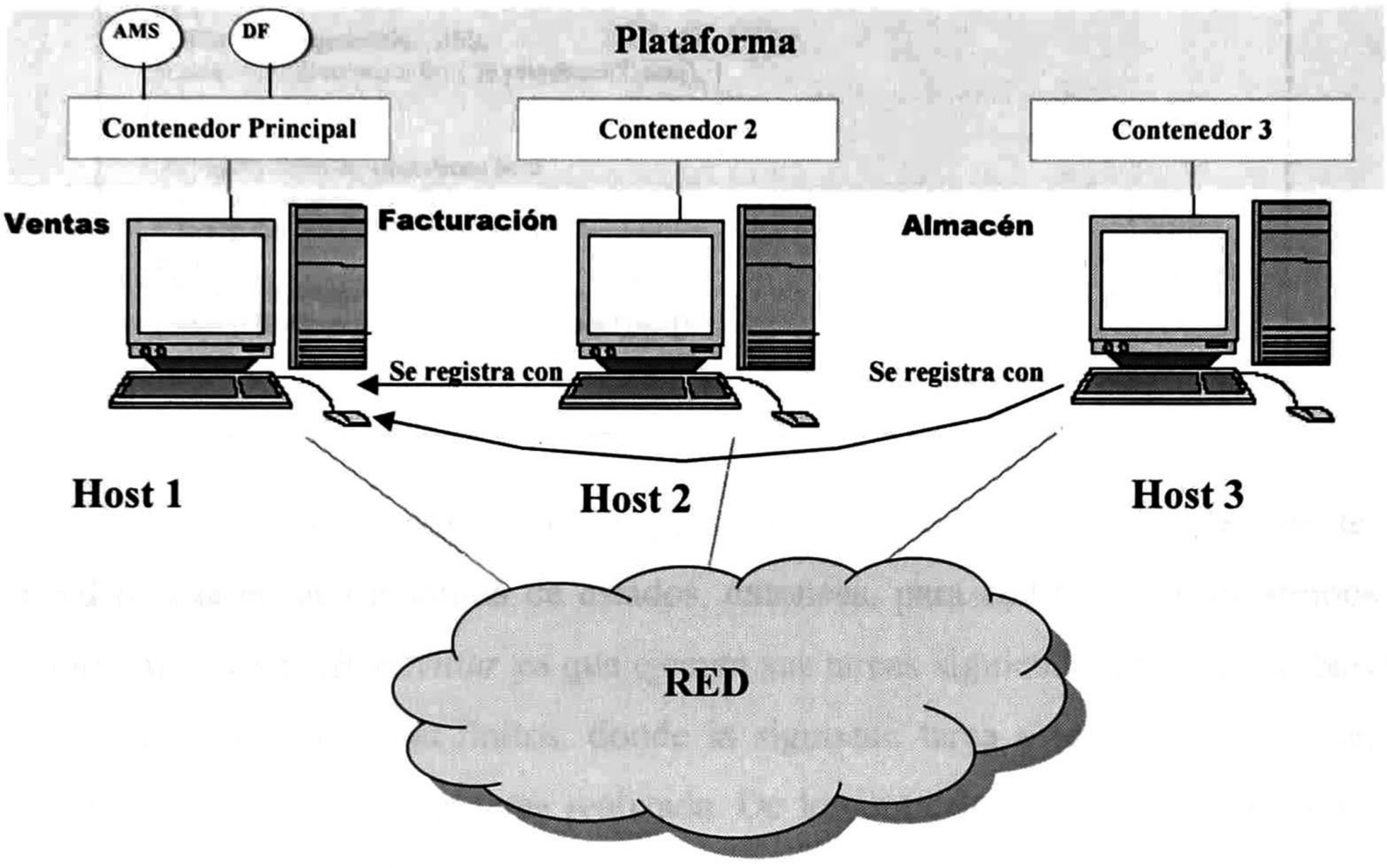


Figura 4.9: Plataforma de agentes para el ejemplo 4

- *Agente Móvil*

Para la codificación de este agente utilizaremos las redes de los niveles 2 y 3:

- La red de nivel 2, nos describe la estructura general del agente
- La red de nivel 3 $Net_{3,1}$, nos dice qué comportamientos tiene el agente
- El resto de las redes de nivel 3 nos detalla cada comportamiento.

A continuación se propondrán algunos pasos que pueden ser útiles para la codificación del agente:

1. Crear una subclase de la clase Agente y registrar su servicio con el DF de la siguiente manera:

```
public class MobileAgent extends Agent {
 // agent initializations here
 protected void setup() {
  // Register the service in the yellow pages
  DFAgentDescription dfd = new DFAgentDescription();
  dfd.setName(getAID());
  ServiceDescription sd = new ServiceDescription();
  sd.setType(Type);
  sd.setName(Name);
  dfd.addServices(sd);
  try {
   DFService.register(this, dfd);
  }catch (FIPAException fe) { fe.printStackTrace(); }
 }

 // Put agent clean-up operations here
 protected void takeDown() {
  // Deregister from the yellow pages
  try {
   DFService.deregister(this);
  }catch (FIPAException fe) {fe.printStackTrace(); }
 }
}
```

2. La red $Net_{3,1}$ del nivel 3 nos modela el plan o comportamiento a seguir del agente. Si ésta red resulta en una máquina de estados, entonces, para codificarlo utilizaremos el comportamiento *FSMBehaviour* ya que ejecuta sus tareas siguiendo una política basada en una máquina de estados finitos, donde la siguiente tarea a realizar se basa en el resultado que arroja la última tarea realizada. De lo contrario se deberá utilizar otro de los comportamientos facilitados por JADE (*SequentialBehaviour, ParallelBehaviour, SimpleBehaviour,* etc).

Para nuestro ejemplo (ejemplo 4) supongamos que el modelo de nuestra red resulta en una máquina de estados finitos, por lo que con ayuda de la red $Net_{3,1}$, modelada en el capítulo anterior, y el comportamiento *FSMBehaviour*, construiremos el código.

- Inicializar una variable con el comportamiento *FSMBehaviour*.

```
FSMBehaviour fsm = new FSMBehaviour(this) {
                    public int onEnd() {
                    System.out.println("FSM          behaviour
completed.");
                                        myAgent.doDelete();
```

- Registrar los estados. Cada segmento de la red *Inicio_tarea* –lugar– *fin_tarea*, nos representa el estado de ejecución de cada tarea, por lo que este segmento se representará como un estado del comportamiento y deberá registrarse como un estado del comportamiento.

  registerState(Funcion_Nombre_Tarea$_i$, Nombre_Estado$_i$);

  La tarea inicial del modelo se deberá registrar como el primer estado y la tarea final como el ultimo estado.

  registerFirstState(Funcion_Nombre_Tarea$_1$, Nombre_Estado$_1$);

  registerLastState(Funcion_Nombre_Tarea$_N$, Nombre_Estado$_N$);

  De igual forma, las transiciones marcadas en el modelo como *MoverDepartamentoi*, deberán de registrarse como estados.

  registerState(Funcion_MoverDepartmento$_1$, Nombre_Estado$_i$);

- Seguir el modelo y registrar cada una de las transiciones entre los estados del comportamiento.

  - Si después de la ejecución de una tarea, solo le sigue una sola tarea se registra una transición por default.

    registerDefaultTransition(Nombre_Estado$_1$, Nombre_Estado$_2$);

    Significa que después de la tarea1 deberá ejecutarse la tarea 2.

  - Si después de una tarea, por ejemplo la tarea 1, puede ejecutarse la tarea 2 ó la tarea 3 y la elección entre estas dos tareas depende de el resultado de la ejecución de la tarea 1, debe de utilizarse:

    registerTransition(Nombre_Estado$_1$, Nombre_Estado$_2$, 0);

    registerTransition(Nombre_Estado$_1$, Nombre_Estado$_3$, 1);

    donde 0 y 1, es el valor de retorno de la tarea 1. (*Nombre_Estado$_1$*).

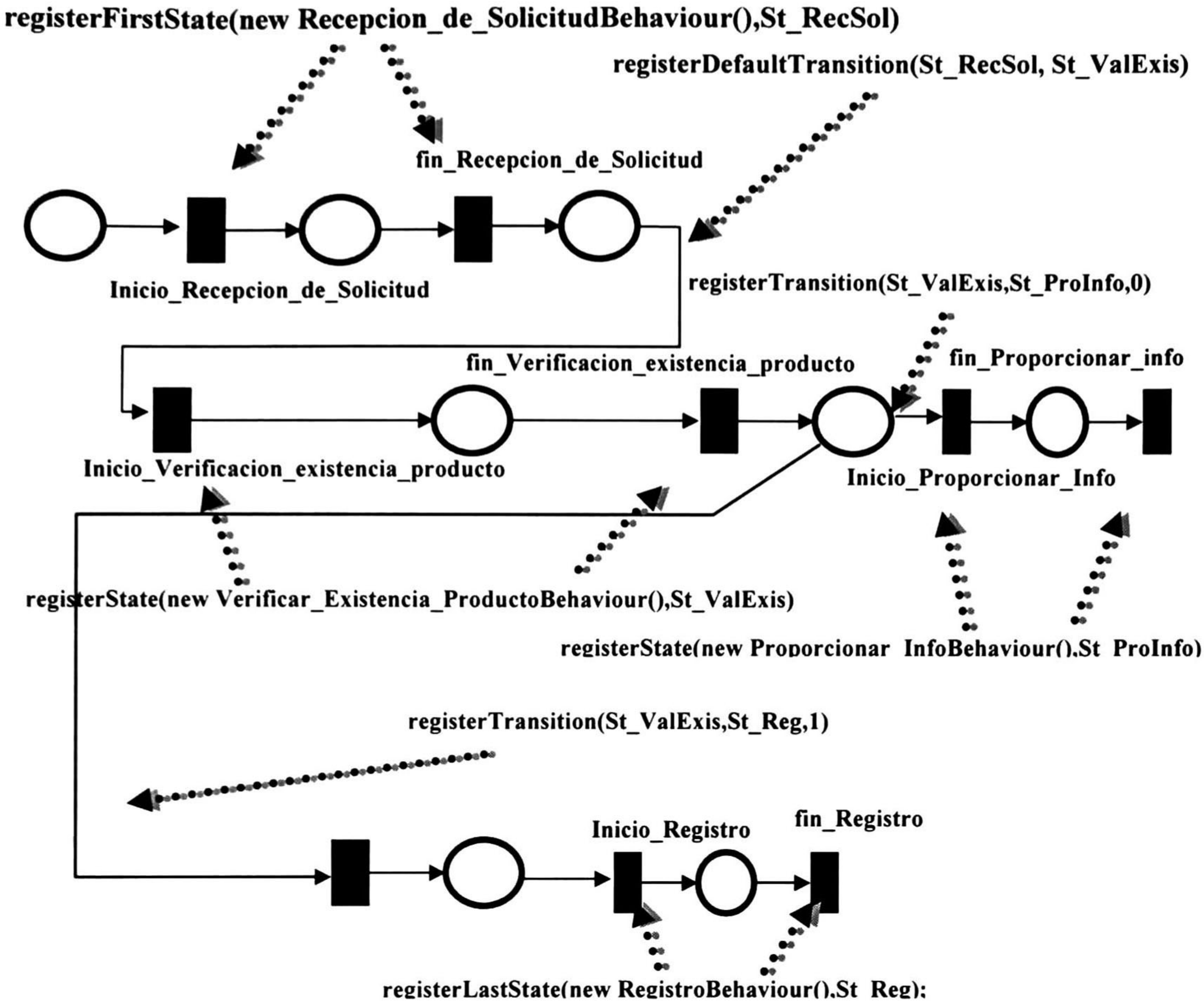


Figura 4.10: Fragmento de la red $Net_{3,1}$ del ejemplo 4 identificando los fragmentos para registrar los estados y las transiciones utilizando el FMSBehaviour

Como estamos programando un agente móvil debemos registrar la ontología de movilidad, lo cual puede hacerse desde el método *setup()*.

```
getContentManager().registerLanguage(new SLCodec(),
FIPANames.ContentLanguage.FIPA_SL0);
// register the mobility ontology
getContentManager().registerOntology(MobilityOntology.getInstance());
```

El código generado quedaría:

```
import jade.core.Agent;
import jade.core.behaviours.*;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.domain.FIPANames;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;

public class MobileAgent extends Agent {
 // states Name
  private static final String St_RecSol  ="1";
  private static final String St_ValExis ="2";
  private static final String St_ProInfo ="3";
  private static final String St_ValComp ="4";
  private static final String St_GenPed  ="5";
  private static final String St_Fact    ="7";
  private static final String St_GenEntr ="8";
  private static final String St_Entr    ="10";
  private static final String St_Reg     ="11";
  private static final String St_MovFact ="6";
  private static final String St_MovAlmac="9";
protected void setup() {
            getContentManager().registerLanguage(new SLCodec(), FIPANames.ContentLanguage.FIPA_SL0);
             getContentManager().registerOntology(MobilityOntology.getInstance());  //register ontology
  DFAgentDescription dfd = new DFAgentDescription();
  dfd.setName(getAID());
  ServiceDescription sd = new ServiceDescription();
  sd.setType("MobileAgent");
  sd.setName("JADE-product_buying");
  dfd.addServices(sd);
  try {   DFService.register(this, dfd);  } catch (FIPAException fe) {fe.printStackTrace();}

  FSMBehaviour fsm = new FSMBehaviour(this){
                        public int onEnd(){
                           myAgent.doDelete();
                        return super.onEnd();
                        } };
 //Register States
  fsm.registerFirstState(new Recepcion_SolicitudBehaviour(),St_RecSol);
  fsm.registerState(new Verificar_Existencia_ProductoBehaviour(),St_ValExis);
  fsm.registerState(new Proporcionar_InfoBehaviour(),St_ProInfo);
  fsm.registerState(new Validar_CompraBehaviour(),St_ValComp);
  fsm.registerState(new Generar_PedidoBehaviour(),St_GenPed);
  fsm.registerState(new FacturarBehaviour(),St_Fact);
  fsm.registerState(new Generar_Orden_EntregaBehaviour(),St_GenEntr);
  fsm.registerState(new EntregaBehaviour(),St_Entr);
  fsm.registerLastState(new RegistroBehaviour(),St_Reg);
  fsm.registerState(new MoverFacturacionBehaviour(),St_MovFact);
  fsm.registerState(new MoverAlmacenBehaviour(),St_MovAlmac);
//RegisterTransitions
  fsm.registerDefaultTransition(St_RecSol,St_ValExis);
  fsm.registerTransition(St_ValExis,St_ProInfo,0);
  fsm.registerTransition(St_ValExis,St_Reg,1);
  fsm.registerDefaultTransition(St_ProInfo,St_ValComp);
  fsm.registerTransition(St_ValComp,St_GenPed,0);
  fsm.registerTransition(St_ValComp,St_Reg,1);
  fsm.registerDefaultTransition(St_GenPed,St_MovFact);
  fsm.registerDefaultTransition(St_MovFact,St_Fact);
  fsm.registerDefaultTransition(St_Fact,St_GenEntr);
  fsm.registerDefaultTransition(St_GenEntr,St_MovAlmac);
  fsm.registerDefaultTransition(St_MovAlmac,St_Entr);
  fsm.registerDefaultTransition(St_Entr,St_Reg);
   addBehaviour(fsm);
}
```

```
  // Put agent clean-up operations here
  protected void takeDown() {
    // Deregister from the yellow pages
    try {
      DFService.deregister(this);
    }catch (FIPAException fe) {fe.printStackTrace();}
  }

private class Verificar_Existencia_ProductoBehaviour extends OneShotBehaviour{
  private int exitValue;
  public void action() {
    //Perform_TASK
     System.out.println("Executing behaviour "+getBehaviourName());
     }
   public int onEnd(){ return exitValue; }
}//behaviour

private class Proporcionar_InfoBehaviour extends OneShotBehaviour{
  private int exitValue;
  public void action() {
   //Perform_TASK
      System.out.println("Executing behaviour "+getBehaviourName());
     }
   public int onEnd(){ return exitValue; }
 }//behaviour

private class Validar_CompraBehaviour extends OneShotBehaviour{
  private int exitValue;
  public void action() {
  //Perform_TASK
      System.out.println("Executing behaviour "+getBehaviourName());
     }
   public int onEnd(){ return exitValue; }
}//behaviour


.
}//end Agent Class
```

En el código anterior observamos que las funciones definidas en el registro de estados del comportamiento FSMBehaviour deberán definirse como clases que extienden de la clase *Behaviour*. Pueden utilizarse también las especializaciones de esta clase según se necesite (para nuestro ejemplo utilizamos la especialización *OneShotBehaviour*). En la función *action()* de cada clase deberá agregarse el código necesario para la ejecución de cada tarea. Para realizar esto el programador deberá hacer uso de las redes del nivel 3 ($Net_{3,2}$, $Net_{3,3}$, $Net_{3,4}$, ..., $Net_{3,n+1}$, siendo n el número de tareas).

Las clases agregadas con el nombre de $MoveDepartamento_i$ deberán poseer el código necesario para lograr la migración del agente al host adecuado. Además deberán de agregarse las funciones *beforeMove()* donde se deberá poner lo que necesitemos que el agente realice ante de migrar a otro host, y *afterMove()* donde deberemos agregar el

código de lo que queremos que el agente realice una vez que llegue a su nuevo sitio, además es necesario registrar nuevamente la ontología de movilidad.

```
protected void afterMove() {
        System.out.println(getLocalName()+" is just Arrived to this location .");
        getContentManager().registerLanguage(new SLCodec(), FIPANames.ContentLanguage.FIPA_SL0);
        getContentManager().registerOntology(MobilityOntology.getInstance());
  }

protected void beforeMove(){
                System.out.println(getLocalName()+" is now moving .");
        }
private class MovFactBehaviour extends OneShotBehaviour{
        private int exitValue;
                public void action(){
                        doMove(Fact_Host_Location);
                }
         public int onEnd()
                {
                        return exitValue;
                }
       }//end Behaviour
```

Para compilar el código del agente se debe poner en la línea de comandos:

javac MobileAgent.java

CAPITULO V.

Desarrollo de un prototipo de un Sistema de Flujo de Trabajo

Resumen. En este capítulo se aplicará la metodología propuesta en los capítulos III y IV para modelar un problema de flujo de trabajo y obtener su implementación en un software basado en JADE. Con la ayuda de la metodología se obtiene un modelado a tres niveles el cual se simula su comportamiento en un sistema llamado nLNS Agents.

5.1 Definición del problema

En este capítulo aplicaremos la metodología descrita en los capítulos anteriores para modelar un problema de flujo de trabajo y luego realizar la codificación como un sistema basado en agentes móviles.

El problema que vamos a modelar es el de Procesar una Queja para una compañía aseguradora, el problema podría definirse como:

> *"Modelar el flujo de trabajo para el proceso de una queja en una compañía aseguradora, en donde un cliente reclama el seguro de un bien (seguro inmobiliario, auto, vida), la compañía deberá recibir la queja, solicitar los datos requeridos del quejoso (número de póliza, etc), validar la vigencia y pagos de la póliza de seguro, validar beneficiarios (si así se requiriera), valorar daños, evaluar el caso, calcular el monto a cubrir aplicando el deducible correcto y realizar el pago del seguro al quejoso, en caso de que la queja proceda, de lo contrario informar al usuario de las razones por las cuales la queja no procedió."*

5.2 Estructuración

Para modelar nuestro problema utilizaremos (como se explica en los capítulos III y IV) la metodología propuesta, por lo que primero desarrollaremos el modelado del problema utilizando el formalismo NLNs y después procederemos a su codificación utilizando la herramienta de programación JADE.

Antes de comenzar el modelado debemos obtener toda la información necesaria para la construcción de los mismos.

De la definición del problema podemos deducir:

- La empresa puede manejar tres tipos diferentes de seguros: seguro inmobiliario, seguro de vida y seguro de un automóvil.
- El quejoso deberá contactarse con la empresa realizando su solicitud y posteriormente la empresa solicitará al quejoso los datos necesarios para procesar la queja
- Se deberá validar la información obtenida del quejoso (se deberá contar con una base de datos que almacene dicha información)
- Se deberán validar los pagos de la póliza así como su vigencia (por lo que se deberá contar con una base de datos que almacene dicha información)
- Se deberán evaluar y registrar los daños y decidir si la queja procede o no
- Una vez evaluados los daños y procedida la queja, se deberá realizar una cotización del monto que la compañía deberá cubrir a dicho quejoso.
- Se deberán realizar los pagos correspondientes

Para términos prácticos modelaremos solamente el caso de una queja de un seguro inmobiliario. Vamos a suponer que la compañía tiene los siguientes departamentos:

- Recepción (Reception): este es el departamento a donde el quejoso se dirige en primera instancia para realizar su queja, por lo que el personal de este departamento le solicitará al quejoso la información necesaria para iniciar el proceso de la queja. A la queja se le asigna un número de queja y la información se pasa a otro departamento (Validación)

- Validación (Validation). Este departamento recibe información, proporcionada por el departamento de Recepción, sobre un número de queja por lo que aquí se utiliza el número de póliza para revisar en una base de datos si la información proporcionada es correcta y se verifica si la póliza aún es vigente y si los pagos de la misma se realizaron completos. Si existe información inconsistente o si la póliza no es vigente, el caso se desecha y se informa al quejoso que su queja no procede y se le notifican los motivos específicos, se registra la información necesaria sobre la queja procesada y se termina con la queja. De estar correcta toda la información proporcionada, se pasa la información al departamento de Valoración de los daños.

- Valoración de los daños (Adjustment): en este departamento se le solicita al quejoso que proporcione la información referente a los daños por los que está solicitando el pago del seguro, además se deberá evaluar cada daño informado y se deberá realizar una evaluación de la información obtenida confrontándola con la proporcionada por el quejoso, de esta manera un ajustador decide si la información es suficiente y necesaria para que la queja proceda o simplemente se rechace la queja. Si la queja procede el ajustador almacena todos los conceptos válidos para su reclamo de seguro para que sea calculado el valor final (en el departamento de Cotización) a pagar.

- Cotización (Assessment): en este departamento se deberá calcular el valor actual de cada daño registrado y se deberá evaluar el valor real a pagar así como el valor total al que ascienden los daños y por los que la compañía pagará.

- Pagos (Payment): en este departamento el quejoso cobra el monto aceptado por la compañía aseguradora y se termina con el proceso realizando el apropiado registro de los datos necesarios.

Suponiendo que las tareas generales involucradas en cada departamento son:

1. Registrar Queja (Claim_register)
    a. Generación de un número de queja
    b. Abrir expediente de la queja para su procesamiento
    c. Solicitar datos necesarios al quejoso

d. Registrar la información recibida

2. Validación de la información (Data_Validation)

a. Solicitar número de queja

b. Abrir registro de pagos de la base de datos

c. Verificar vigencia y pagos de la póliza

d. Decidir si la queja procede o no

3. Valoración de los daños (Adjustment)

a. Enviar formulario de daños al quejoso

b. Recibir formulario llenado por el quejoso

c. Valorar daños reales

d. Confrontar información obtenida con la proporcionada por el quejoso

e. Evaluar la situación

f. Decidir si la queja procede o no

4. Cotización de los daños (Assessment)

a. Abrir expediente de daños

b. De cada registro calcular el costo actual del daño

c. Estimar el costo del daño

d. Calcular la liquidación final

e. Aprobar liquidación

5. Pagos (Payment)

a. Abrir expediente de pagos

b. Aprobar pago

c. Verificar forma de pago

d. Generar formato de recibo de pago de seguro

e. Realizar pago

6. Registro (File)

   a. Recabar la información necesaria

   b. Llenar el expediente correcto

Donde cada nombre entre paréntesis nos representará el nombre corto por el que nos referiremos a cada tarea. Ahora procederemos a realizar el modelado

5.3 Modelado

Para lograr nuestro modelado a 3 niveles, construiremos el nivel 1, después las redes de nivel 3 (similarmente al capítulo anterior) y terminaremos con el nivel 2.

5.3.1 Red del primer nivel

A continuación modelaremos la red de primer nivel o red ambiente con ayuda de la información obtenida de la definición del problema y siguiendo la metodología propuesta en el capítulo III.

El modelo se lograría siguiendo:

a) Identificación de los departamentos involucrados

De la información descrita en el apartado anterior tenemos que los departamentos involucrados en el proceso son los mostrados en la figura 5.1.

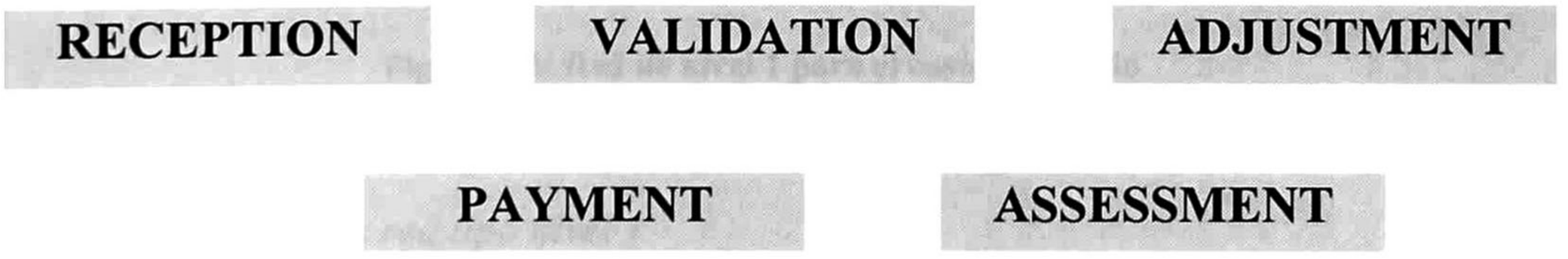


Figura 5.1: Identificación de los departamentos involucrados en el caso de estudio

b) Identificación de la dirección del flujo del caso entre los departamentos

La dirección del flujo de información quedaría como lo ilustra la figura 5.2.

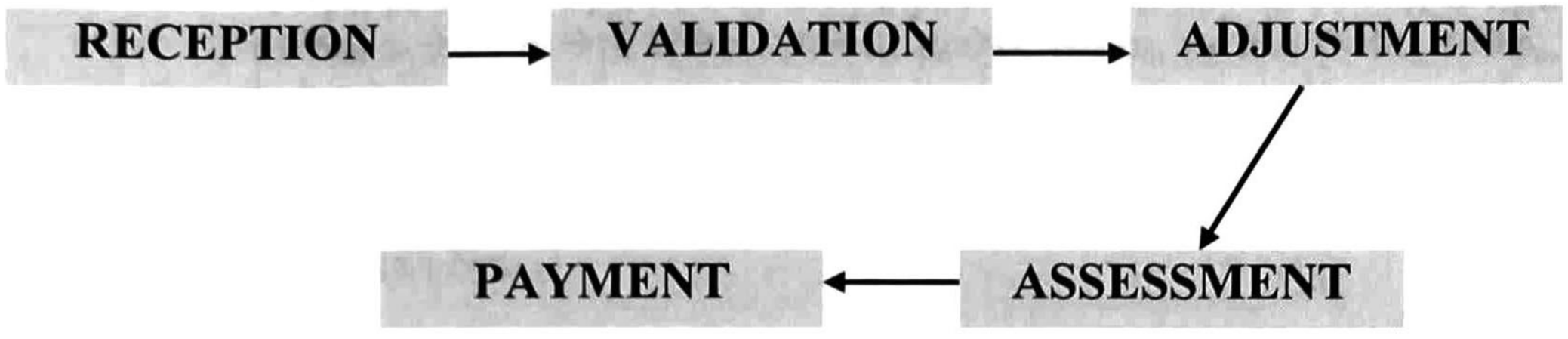


Figura 5.2: Identificación del sentido del flujo entre los departamentos

c) Construcción de la Red de Petri del primer nivel, a partir del modelo anterior

Si modelamos también los recursos que se requieren en cada departamento, la red quedaría la figura 5.3.

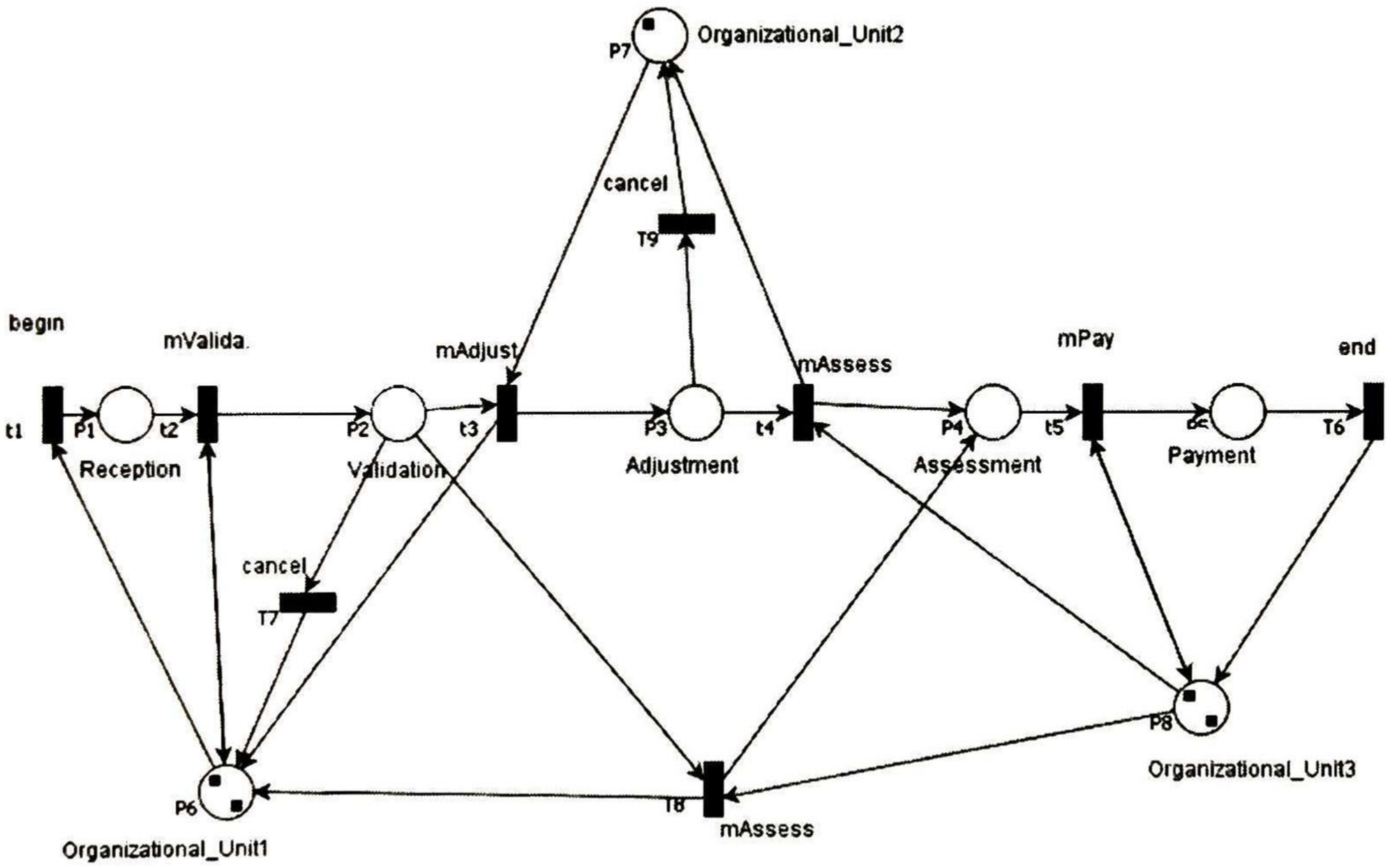


Figura 5.3: Red de nivel 1 para el caso de estudio

d) Definición de la red tipo nivel 1

a. $Net_{1,1}$ =(G, $TOKEN_{1,1}$,$LABEL_{1,1}$,$VAR_{1,1}$, τ, λ, χ, π)

donde:

G: Es el grafo obtenido en la figura anterior donde $P=\{p_1,p_2,p_3,\ldots, p_8\}$, $T=\{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9\}$ $F=\{t_1\rightarrow p_1,\ p_1\rightarrow t_2,\ t_2\rightarrow p_2,\ p_2\rightarrow t_3,\ t_3\rightarrow p_3,\ p_3\rightarrow t_4,$

$t_4 \rightarrow p_4$, $p_4 \rightarrow t_5$, $t_5 \rightarrow p_5$, $p_5 \rightarrow t_6$, $t_6 \rightarrow p_8$, $p_8 \rightarrow t_5$, $t_5 \rightarrow p_8$, $p_8 \rightarrow t_4$, $p_8 \rightarrow t_8$, $t_8 \rightarrow p_4$, $p_3 \rightarrow t_9$, $t_9 \rightarrow p_7$, $p_7 \rightarrow t_3$, $t_4 \rightarrow p_7$, $p_6 \rightarrow t_1$, $p_6 \rightarrow t_2$, $t_2 \rightarrow p_6$, $p_2 \rightarrow t_7$, $p_2 \rightarrow t_8$, $t_7 \rightarrow p_6$, $t_3 \rightarrow p_6$,}

$TOKEN_{1,1}$: se define como $TOKEN_{1,1}=\{Net_{2,1}, s1\}$

$VAR_{1,1}=\{x : Net_{2,1}\}$

$\tau(p_1)= \tau(p_2)=\tau(p_3)= \tau(p_4)= \tau(p_5)=\{Net_{2,1}\}$ $\tau(p_6)= \tau(p_7)= \tau(p_8)=\{s1\}$

$\lambda(t_1)=\{mRecep\downarrow\}$ $\lambda(t_2)=\{mValida\downarrow\}$ $\lambda(t_3)=\{mAdjust\downarrow\}$ $\lambda(t_4)=\{mAssess\downarrow\}$ $\lambda(t_5)=\{mPay\downarrow\}$ $\lambda(t_6)=\{Mov\downarrow\}$ $\lambda(t_7)=\{Cancel\downarrow\}$

$\pi((t_1,p_1),mRecep\downarrow)=$ $\pi((p_1,t_2),mValida\downarrow)=$ $\pi((t_2,p_2),mValida\downarrow)=$
$\pi((p_2,t_3),mAdjust\downarrow)=$ $\pi((t_3,p_3),mAdjust\downarrow)=$ $\pi((p_3,t_4),mAssess\downarrow)=$
$\pi((t_4,p_4),mAssess\downarrow)=$ $\pi((p_4,t_5),mPay\downarrow)=$ $\pi((t_5,p_5),mPay\downarrow)=$
$\pi((p_5,t_6),)=\{Net_{2,1}\}$

$\pi((t_2,p_6),$ $mValida\downarrow)=$ $\pi((t_3,p_6),$ $mAdjust\downarrow)=$ $\pi((t_4,p_7),$ $mAssess\downarrow)=$
$\pi((t_5,p_8),$ $mPay\downarrow)$ $=\pi((t_7,p_6),Cancel\downarrow)=$ $\pi((t_8,p_6),mAssess\downarrow)=$
$\pi((t_9,p_7),Cancel\downarrow)=$ $\pi((t_6,p_8),Mov\downarrow)=$ $\pi((p_6,t_2),mValida\downarrow)=$
$\pi((p_6,t_1),$ $mRecep\downarrow)=$ $\pi((p_8,t_5),$ $mPay\downarrow)=$ $\pi((p_7,t_3),$ $mAdjust\downarrow)=$
$\pi((p_8,t_4),$ $mAssess\downarrow)=$ $\pi((p_8,t_8),$ $mAssess\downarrow)=\{s1\}$

$\pi((t_8,p_4),$ $mAssess\downarrow)=$ $\pi((p_2,t_7),$ $Cancel\downarrow)=$ $\pi((p_2,t_8),$ $mAssess\downarrow)=$
$\pi((p_3,t_9),$ $Cancel\downarrow)=\{Net_{2,1}\}$

Las transiciones etiquetadas como *Cancel* nos sirven para cancelar en caso, cuando se disparan las transiciones con esta etiqueta se liberan recursos y se elimina el Token correspondiente a la red de nivel 2 (red Agente). Para estos casos podríamos suponer un lugar auxiliar a donde se manda el token que corresponde a la red de nivel 2 ($Net_{2,1}$) de manera que se enfatice el estado de caso cancelado, como lo ilustra la figura 5.4.

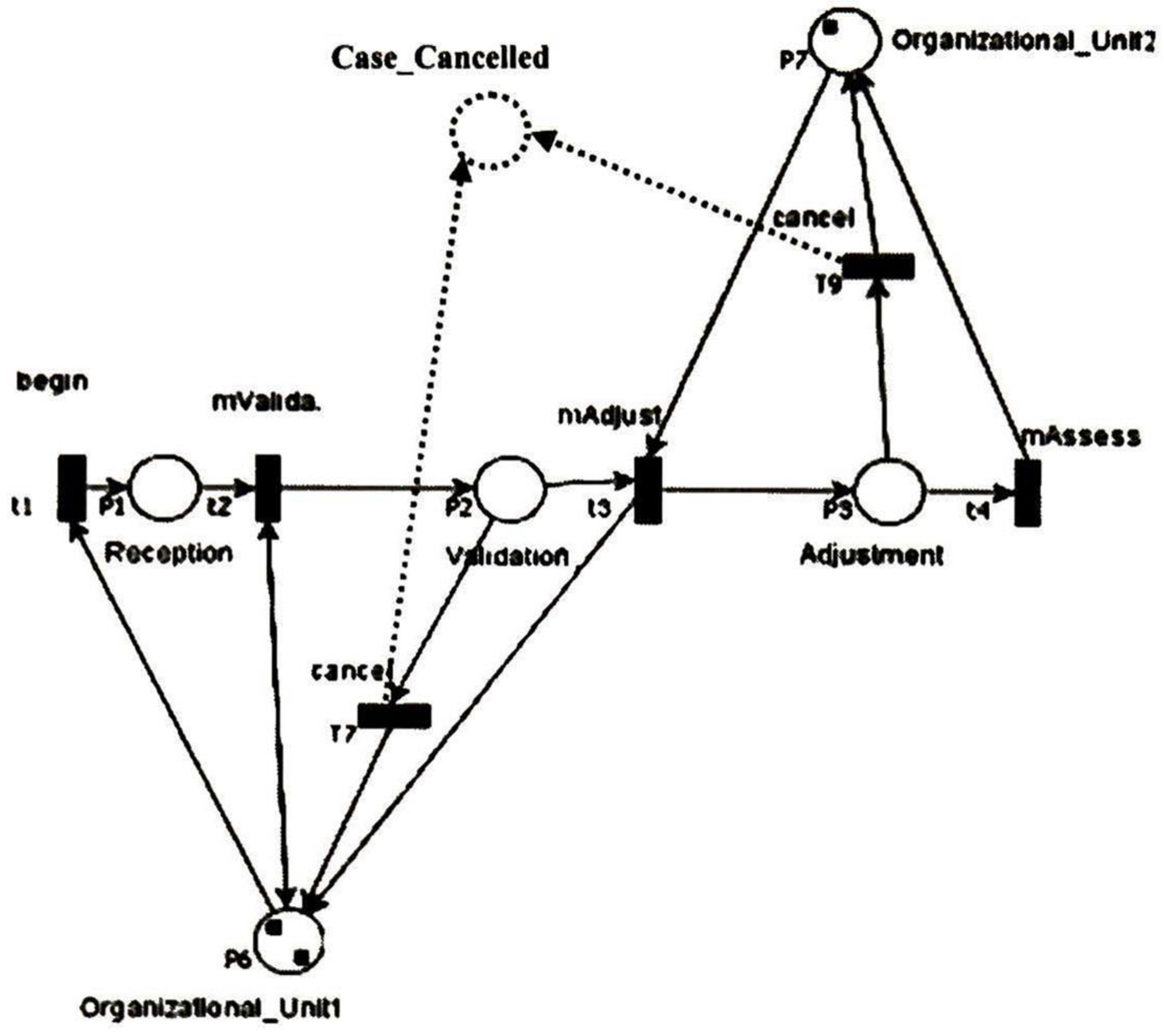


Figura 5.4: Fragmento de la red 1 enfatizando lugar de caso cancelado

5.3.2 Redes del tercer nivel

- *Red agente Net3,1*

Primero construiremos la red que modelara el comportamiento del agente (el cual se modelara su estructura general con la red de nivel 2 $Net_{2,1}$). Siguiendo los pasos de la metodología tenemos:

1. Definir la red tipo $Net_{3,1}$

$$Net_{3,1} = (G, TOKEN_{3,1}, LABEL_{3,1}, VAR_{3,1}, \tau, \lambda, \chi, \pi)$$

2. Construir el grafo G

a. Encontrar el valor de W_1 y W_1'

De la información obtenida en la definición del problema tenemos:

W_1={Claim_register, Data_Validation, Adjustment, Assessment, Payment, File}

W_1'=*{Claim_register ≺ Data_Validation, Data_Validation ≺ Adjustment, Data_Validation ≺ File, Adjustment ≺ Assessment, Adjustment ≺ File, Assessment ≺ Payment, Payment ≺ File}*

b. Tomar de W_1 y W_1' los valores correspondientes al departamento 1

El departamento inicial es el de *Recepción (Reception)* por lo que los subconjuntos serían:

Tareas:

{Claim_Register}

Relaciones de precedencia

{Claim_register ≺ Data_Validation}

c. Construir los fragmentos de red para enfatizar el inicio y fin de cada tarea.

Inicio de tarea: *B*. Fin de tarea: *F*.

Tarea: Claim_Register

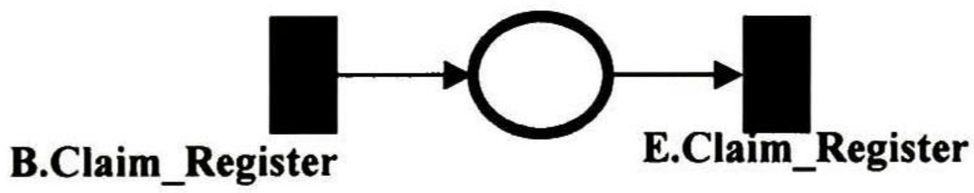


Figura 5.5: Segmento de red para la tarea Claim_register resaltando inicio y fin de tarea

d. Unir los fragmentos definiendo un lugar de inicio.

Tarea: Claim_Register

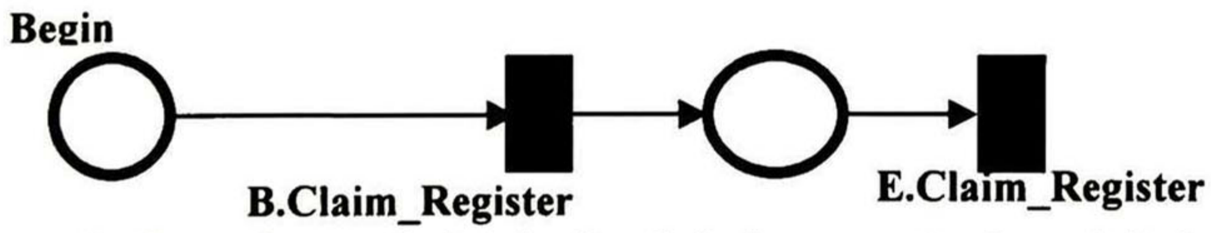


Figura 5.6: Agregado de un lugar de inicio (begin) al segmento de red de la tarea Claim_register

e. Realizar lo mismo con el resto de los departamentos

Para simplificación de pasos, aplicaremos todos los pasos a todos los departamentos al mismo tiempo.

Por lo que, para el departamento de *Validation:*

Tareas:

{Data_Validation}

Relaciones de precedencia

{Data_Validation ≺ Adjustment, Data_Validation ≺ File}

Para el departamento de *Adjustment:*

Tareas:

{Adjustment}

Relaciones de precedencia

{ Adjustment ≺ Assessment, Adjustment ≺ File}

Para el departamento de *Assessment*:

Tareas:

{Assessment}

Relaciones de precedencia

{Assessment ≺ Payment}

Para el departamento de *Payment:*

Tareas:

{Payment}

Relaciones de precedencia

{Payment ≺ File}

Construir los fragmentos de red para enfatizar el inicio y fin de cada tarea

Inicio de tarea: *B.* Fin de tarea: *F*

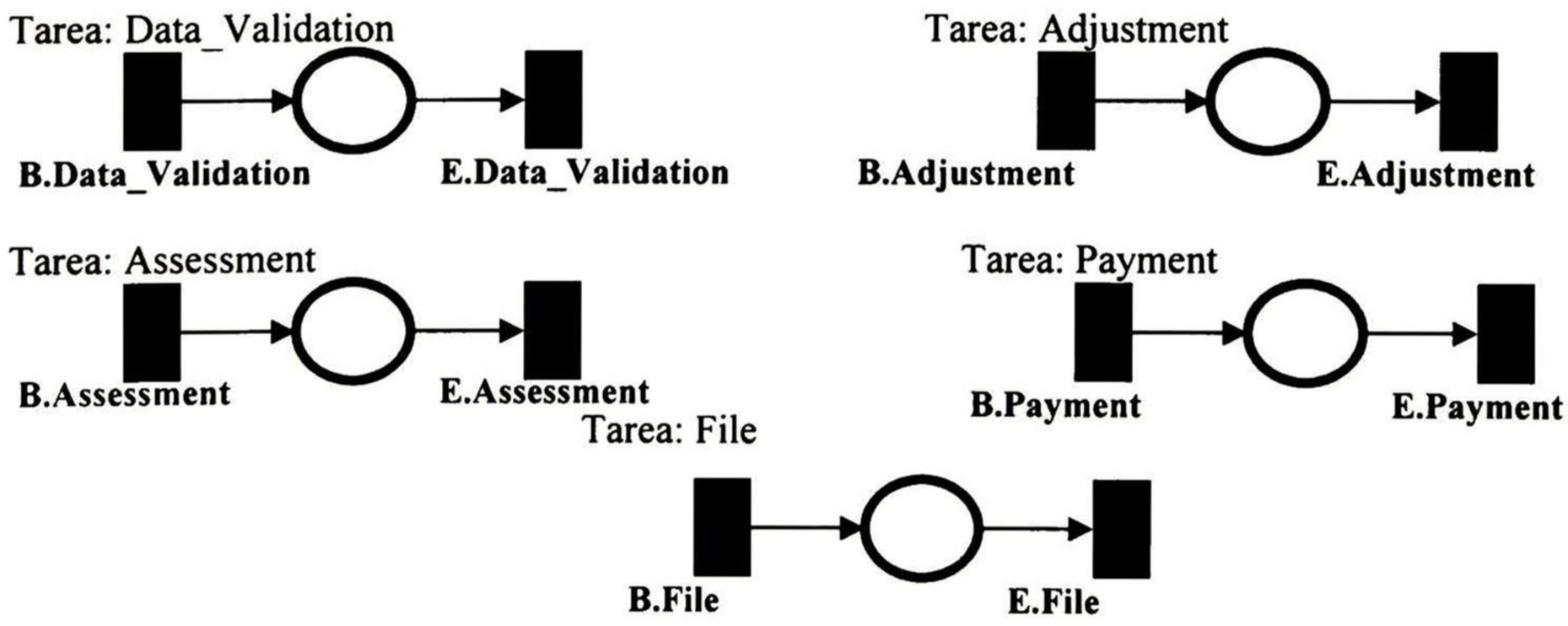


Figura 5.7: Modelado del resto de las tareas en segmentos de red, para la red Net3,1

Uniendo todos los fragmentos resulta la red de la figura 5.8.

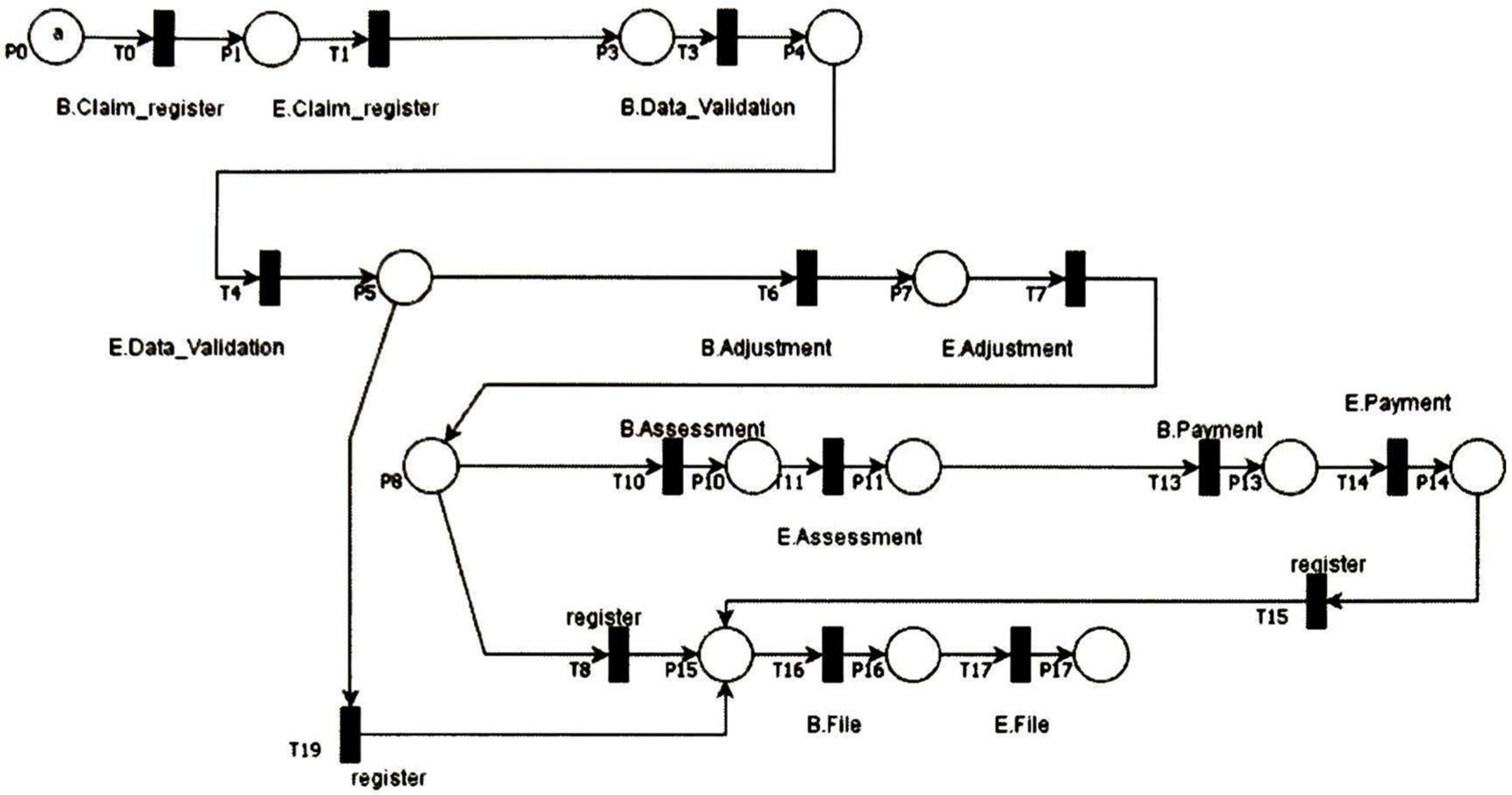


Figura 5.8: Unión de todos los segmentos de red. en la red Net3,1

Ahora, para enfatizar los cambios entre departamentos agregaremos las etiquetas identificadas como *mDepatamento*$_i$, en las que se modelará *mover al departamento i* como lo muestra la figura 5.9.

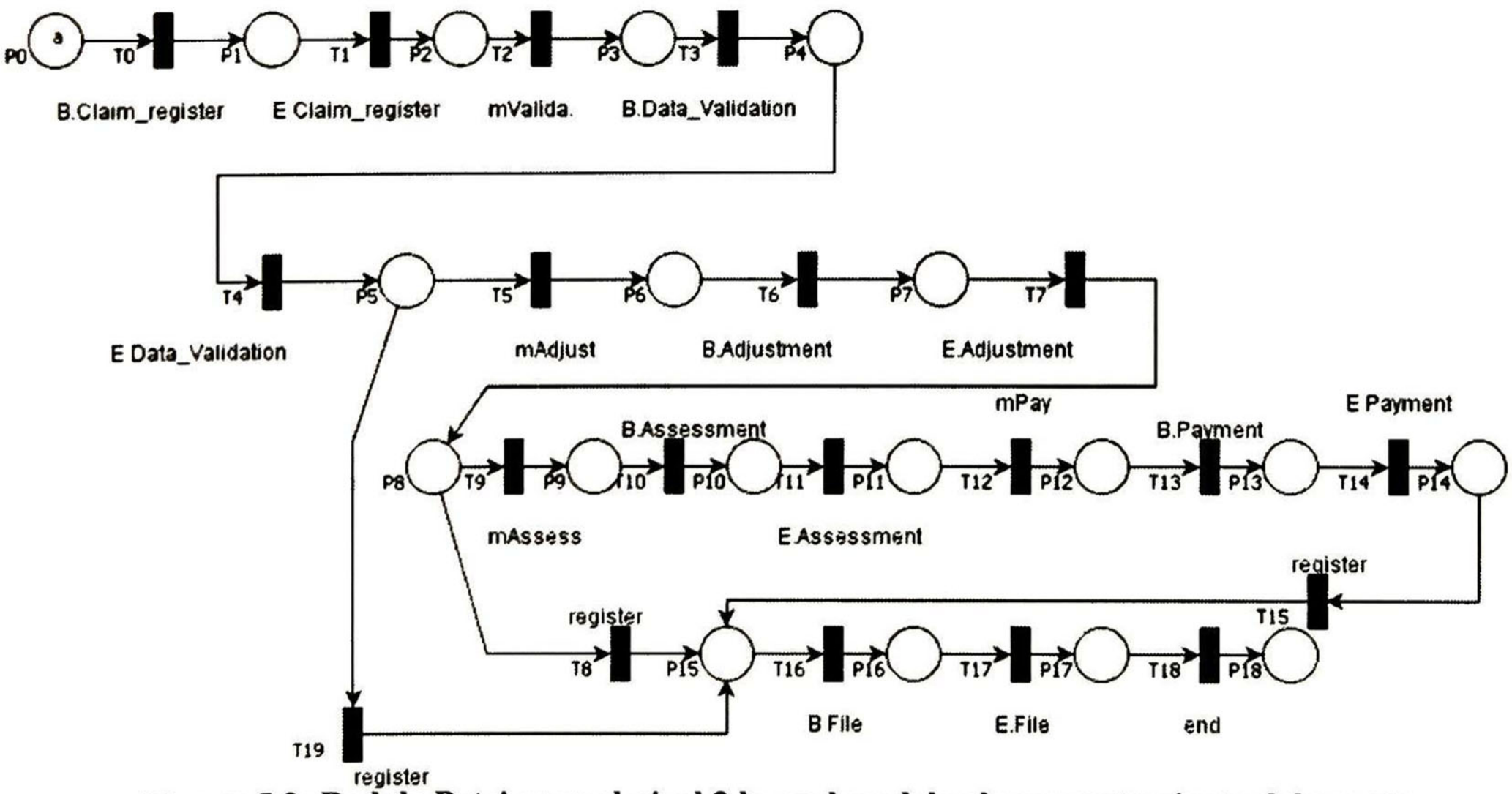


Figura 5.9: Red de Petri para el nivel 3 la cual modela el comportamiento del agente

3. Definir valor del resto de los conjuntos de la red

$Net_{3,1}$ =(G, $TOKEN_{3,1}$,$LABEL_{3,1}$,$VAR_{3,1}$, τ, λ, χ, π)

- $TOKEN_{3,1}$={a, s}

- $LABEL_{3,1}$= {B.Claim_register, E.Claim_register, mValida, B.Data_Validation, E.Data_Validation1, E.Data_Validation2, mAdjust, B.Adjustment, E.Adjustment$_1$, E.Adjustment$_2$, mAssess, B.Assessment, E.Assessment, mPay, B.Payment, E.Payment, register, B.File, E.File, Cancel, Mov}

- $VAR_{3,1}$= ∅

- $\tau(p_0)= \tau(p_1)= \tau(p_2)= \tau(p_3)= \tau(p_4)= \tau(p_6)= \tau(p_7)= \tau(p_9)= \tau(p_{10})= \tau(p_{11})= \tau(p_{12})= \tau(p_{13})= \tau(p_{14})=$ {a} $\tau(p_{15})= \tau(p_{16})= \tau(p_{17})= \tau(p_{18})=$={s} $\tau(p_5)= \tau(p_8)=$={a, s}

- $\lambda(t_0)$={B.Claim_register↑} $\lambda(t_1)$={E.Claim_register↑≡} $\lambda(t_2)$={mValida↑} $\lambda(t_3)$={B.Data_Validation↑} $\lambda(t_4)$={E.Data_Validation$_1$↑≡} $\lambda(t_4)$={E.Data_Validation$_2$↑≡} $\lambda(t_5)$={mAdjust↑} $\lambda(t_6)$={B.Adjustment↑} $\lambda(t_7)$={E.Adjustment$_1$↑≡} $\lambda(t_7)$={E.Adjustment$_2$↑≡} $\lambda(t_8)$={register} $\lambda(t_9)$={mAssess↑} $\lambda(t_{10})$={B.Assessment↑} $\lambda(t_{11})$={E.Assessment↑≡} $\lambda(t_{12})$={mPay↑} $\lambda(t_{13})$={B.Payment↑} $\lambda(t_{14})$={E.Payment↑≡} $\lambda(t_{15})$={register} $\lambda(t_{16})$={B.File↑} $\lambda(t_{17})$={E.File↑≡} $\lambda(t_{18})$={Mov↑} $\lambda(t_{18})$={Cancel↑} $\lambda(t_{19})$={register}

- $\chi(t_1$,E.Claim_register↑≡)=(2,2) $\chi(t_4$,E.Data_Validation$_1$↑≡)=(2,2) $\chi(t_4$,E.Data_Validation$_2$↑≡)=(2,2) $\chi(t_7$,E.Adjustment$_1$↑≡)=(2,2) $\chi(t_7$,E.Adjustment$_2$↑≡)=(2,2) $\chi(t_{11}$,E.Assessment↑≡)=(2,2) $\chi(t_{14}$,E.Payment↑≡)=(2,2) $\chi(t_{17}$,E.File↑≡)=(2,2)

- $\pi((p_0,t_0)$,B.Claim_register↑)= $\pi((t_0,p_1)$,B.Claim_register↑)= $\pi((p_0,t_1)$,E.Claim_register↑≡)= $\pi((t_1,p_2)$,E.Claim_register↑≡)= $\pi((p_2,t_2)$,mValida↑)= $\pi((t_2,p_3)$,mValida↑)= $\pi((p_3,t_3)$,B.Data_Validation↑)= $\pi((t_3,p_4)$,B.Data_Validation↑)= $\pi((p_4,t_4)$,E.Data_Validation$_1$↑≡)= $\pi((p_4,t_4)$,E.Data_Validation$_2$↑≡)= $\pi((p_5,t_5)$,mAdjust↑)= $\pi((t_5,p_6)$,mAdjust↑)= $\pi((p_6,t_6)$,B.Adjustment↑)= $\pi((t_6,p_7)$,B.Adjustment↑)= $\pi((p_7,t_7)$,E.Adjustment$_1$↑≡)= $\pi((p_7,t_7)$,E.Adjustment$_2$↑≡)= $\pi((t_7,p_8)$,E.Adjustment$_1$↑≡)= $\pi((p_8,t_9)$,mAssess↑)= $\pi((t_9,p_9)$,mAssess↑)= $\pi((p_9,t_{10})$,B.Assessment↑)= $\pi((t_{10},p_{10})$,B.Assessment↑)= $\pi((p_{10},t_{11})$,E.Assessment↑≡)= $\pi((t_{11},p_{11})$,E.Assessment↑≡)= $\pi((p_{11},t_{12})$,mPay↑)= $\pi((t_{12},p_{12})$,mPay↑)= $\pi((p_{12},t_{13})$,B.Payment↑)= $\pi((t_{13},p_{13})$,B.Payment↑)= $\pi((p_{13},t_{14})$,E.Payment↑≡)= $\pi((t_{14},p_{14})$,E.Payment↑≡)= $\pi((p_{14},t_{15})$,register)= {a}

$\pi((t_4,p_5)$,E.Data_Validation$_2$↑≡)= $\pi((t_7,p_8)$,E.Adjustment$_2$↑≡)= $\pi((t_{15},p_{15})$,register)= $\pi((p_{15},t_{16})$,B.File↑)= $\pi((t_{16},p_{16})$,B.File↑)=

$\pi((p_{16},t_{17},),E.File\uparrow\equiv)=$ $\pi((t_{17},p_{17}),E.File\uparrow\equiv)=$ $\pi((p_{17},t_{18}),Mov\uparrow)=$
$\pi((t_{18},p_{18}),Mov\uparrow)=$ $\pi((p_{17},t_{18}),Cancel\uparrow)=$ $\pi((t_{18},p_{18}),Cancel\uparrow)=$
$\pi((p_8,t_8),register)=$ $\pi((t_8,p_{15}),register)=$ $\pi((p_5,t_{19}),register)=$
$\pi((t_{19},p_{15}),register)=\{s\}$

- *Redes que modelan las tareas*

Las siguientes redes nos modelarán las operaciones involucradas en cada tarea. Cabe recordar que todas estas redes servirán como marcado de la red de nivel 2 que modelará el comportamiento general del agente y la cual modelaremos más adelante. Siguiendo los pasos de la metodología tenemos:

1. Para la tarea *Claim_Register* identificamos las siguientes operaciones:
    a. Generación de un número de queja
    b. Abrir expediente de la queja para su procesamiento
    c. Solicitar datos necesarios al quejoso
    d. Registrar la información recibida

El modelo del grafo G quedaría como lo ilustra la figura 5.10.

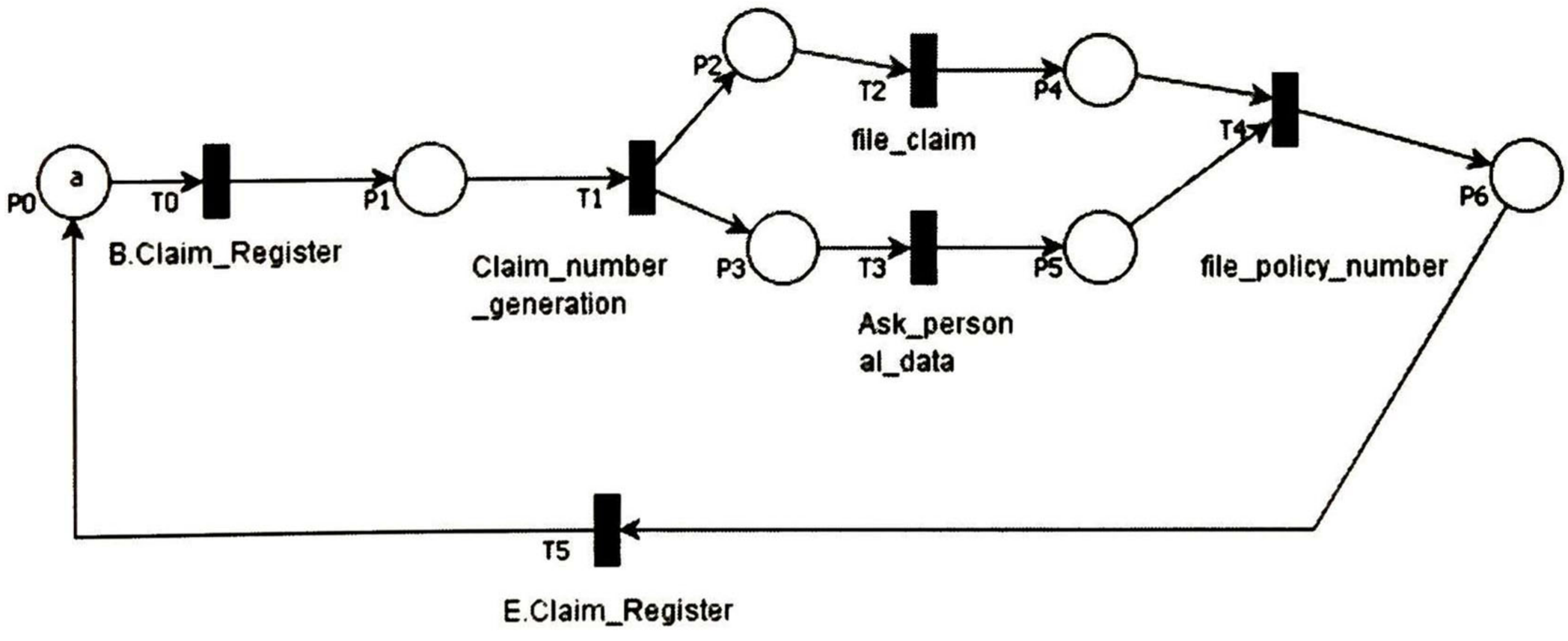


Figura 5.10: Red de Petri para el modelado de las operaciones de la tarea Claim_register (Net3,2)

$$Net_{3,2} = (G, TOKEN_{3,2}, LABEL_{3,2}, VAR_{3,2}, \tau, \lambda, \chi, \pi)$$

- $TOKEN_{3,2}$={a}
- $LABEL_{3,2}$={B.Claim_register, Claim_number_generation, file_Claim, Ask_personal_Data, file_policy_number, E.Claim_register}
- $VAR_{3,2}$=∅
- $\tau(p_0)= \tau(p_1)= \tau(p_2)= \tau(p_3)= \tau(p_4)= \tau(p_5)= \tau(p_6)=$ {a}
- $\lambda(t_0)$={B.Claim_register↑} $\lambda(t_1)$={Claim_number_generation} $\lambda(t_2)$={file_Claim} $\lambda(t_3)$={Ask_personal_Data} $\lambda(t_4)$={file_policy_number} $\lambda(t_5)$={E.Claim_register↑≡}
- $\chi(t_5$, E.Claim_register↑≡)=(2,2)
- $\pi((p_0,t_0)$,B.Claim_register↑)= $\pi((t_0,p_1)$,B.Claim_register↑)= $\pi((p_1,t_1)$,Claim_number_generation)= $\pi((t_1,p_2)$,Claim_number_generation)= $\pi((t_1,p_3)$,Claim_number_generation)= $\pi((p_2,t_2)$,file_Claim)= $\pi((t_2,p_4)$,file_Claim)= $\pi((p_3,t_3)$,Ask_personal_Data)= $\pi((t_3,p_5)$,Ask_personal_Data)= $\pi((p_4,t_4)$,file_policy_number)= $\pi((p_5,t_4)$,file_policy_number)= $\pi((t_4,p_6)$,file_policy_number)= $\pi((p_6,t_5)$,E.Claim_register↑≡)= $\pi((t_5,p_1)$, E.Claim_register↑≡)= {a}

2. Para la tarea *Data_Validation* identificamos las siguientes operaciones:

a. Solicitar número de queja

b. Abrir registro de pagos de la base de datos

c. Verificar vigencia y pagos de la póliza

d. Decidir si la queja procede o no

El modelo del grafo G se ilustra en la figura 5.11.

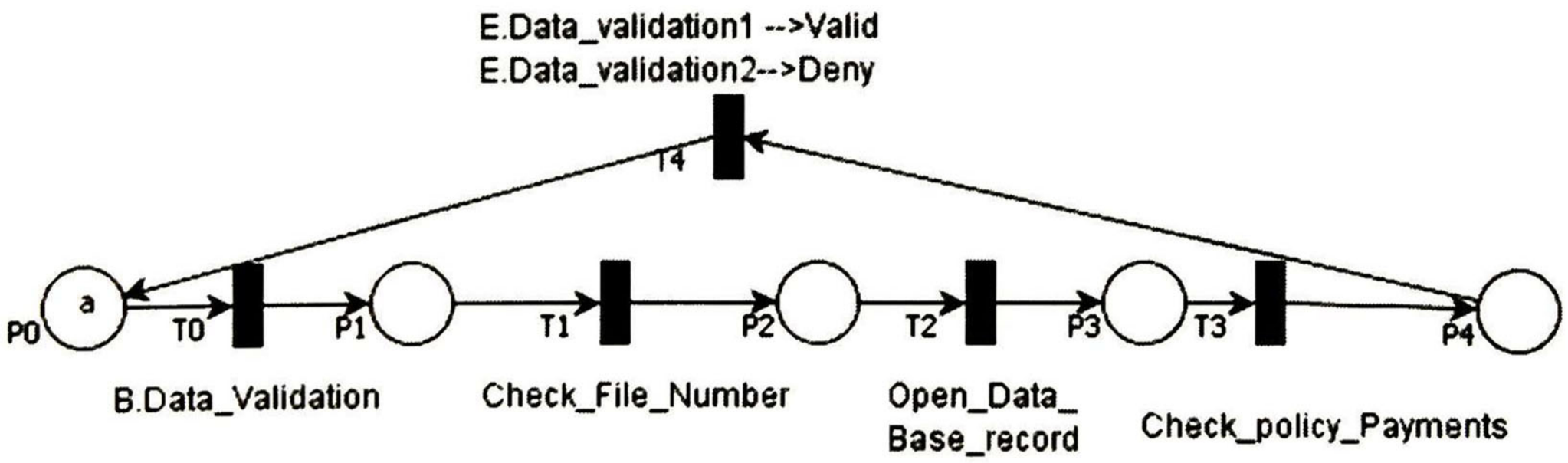


Figura 5.11:Red de Petri para el modelado de las operaciones de la tarea Data_Validation (Net3,3)

$Net_{3,3}$ =(G, $TOKEN_{3,3}$,$LABEL_{3,3}$,$VAR_{3,3}$, τ, λ, χ, π)

- $TOKEN_{3,3}$={a}

- $LABEL_{3,3}$={B.Data_Validation, Check_File_Number, Open_Data_Base_record, Check_policy_Payments, $E.Data\_Validation_1$, $E.Data\_Validation_2$,}

- $VAR_{3,3}$=∅

- $\tau(p_0)= \tau(p_1)= \tau(p_2)= \tau(p_3)= \tau(p_4)$={a}

- $\lambda(t_0)$={B.Data_Validation↑} $\lambda(t_1)$={Check_File_Number}
  $\lambda(t_2)$={Open_Data_Base_record} $\lambda(t_3)$={Check_policy_Payments}
  $\lambda(t_4)$={$E.Data\_Validation_1$↑≡} $\lambda(t_4)$={$E.Data\_Validation_2$↑≡}

- $\chi(t_4, E.Data\_Validation_1\uparrow\equiv)= \chi(t_4, E.Data\_Validation_2\uparrow\equiv)=(2,2)$

- $\pi((p_0,t_0)$,B.Data_Validation↑)= $\pi((t_0,p_1)$,B.Data_Validation↑)=
  $\pi((p_1,t_1)$,Check_File_Number)= $\pi((t_1,p_2)$,Check_File_Number)=
  $\pi((p_2,t_2)$,Open_Data_Base_record)= $\pi((t_2,p_3)$,Open_Data_Base_record)=
  $\pi((p_3,t_3)$,Check_policy_Payments)= $\pi((t_3,p_4)$,Check_policy_Payments)=
  $\pi((p_4,t_4)$,$E.Data\_Validation_1$↑≡)= $\pi((t_4,p_0)$,$E.Data\_Validation_1$↑≡)=
  $\pi((p_4,t_4)$,$E.Data\_Validation_2$↑≡)= $\pi((t_4,p_0)$,$E.Data\_Validation_2$↑≡)={a}

3. Para la tarea *Adjustment* identificamos las siguientes operaciones:

   a. Enviar formulario de daños al quejoso

   b. Recibir formulario llenado por el quejoso

   c. Tiempo agotado para recibir el formulario

   d. Valorar daños reales

   e. Confrontar información obtenida con la proporcionada por el quejoso

   f. Evaluar la situación

   g. Decidir si la queja procede o no

   El Modelo del grafo G quedaría como la figura 5.12.

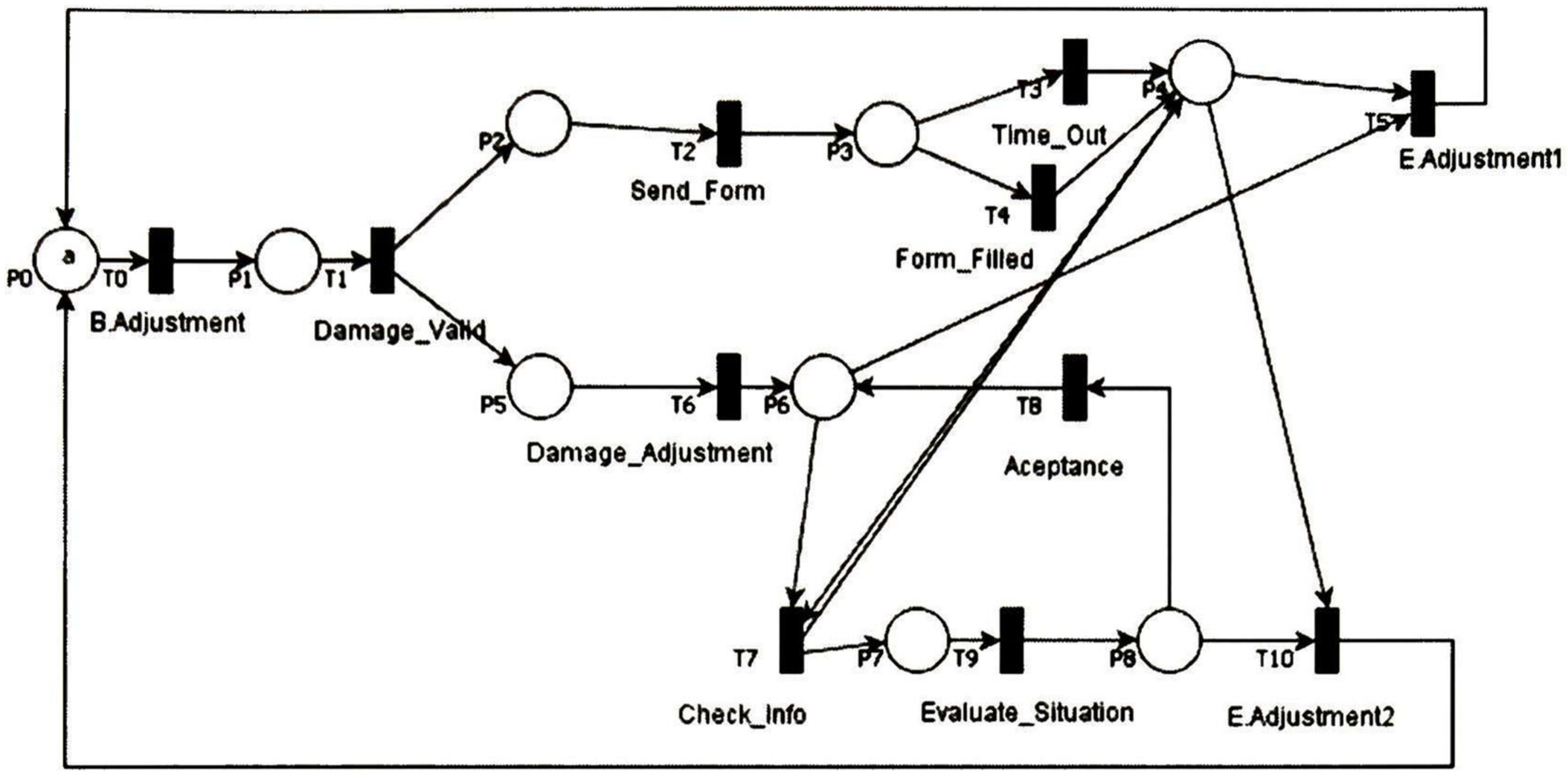


Figura 5.12:Red de Petri para el modelado de las operaciones de la tarea Adjustment (Net3,4)

$Net_{3,4}$ =(G, $TOKEN_{3,4}$,$LABEL_{3,4}$,$VAR_{3,4}$, τ, λ, χ, π)

- $TOKEN_{3,4}$={a}
- $LABEL_{3,4}$={B.Adjustment, Damage_Valid, Send_Form, Damage_Adjustment, Time_Out, Form_Filled, Check_info, Evaluate_Situation, Aceptance, $E.Adjustment_1$, $E.Adjustment_2$}
- $VAR_{3,4}$=∅
- $\tau(p_0)= \tau(p_1)= \tau(p_2)= \tau(p_3)= \tau(p_4)= \tau(p_5)= \tau(p_6)= \tau(p_7)= \tau(p_8)$={a}
- $\lambda(t_0)$={B.Adjustment↑} $\lambda(t_1)$={Damage_Valid} $\lambda(t_2)$={Send_Form} $\lambda(t_3)$={Time_Out} $\lambda(t_4)$={Form_Filled} $\lambda(t_5)$={$E.Adjustment_1$↑≡} $\lambda(t_6)$={Damage_Adjustment} $\lambda(t_7)$={Check_info} $\lambda(t_8)$={Aceptance} $\lambda(t_9)$={Evaluate_Situation} $\lambda(t_{10})$={$E.Adjustment_2$↑≡}
- $\chi(t_5, E.Adjustment_1\uparrow\equiv)= \chi(t_{10}, E.Adjustment_2\uparrow\equiv)$= (2,2)
- $\pi((p_0,t_0),B.Adjustment\uparrow)=$ $\pi((t_0,p_1),B.Adjustment\uparrow)=$ $\pi((p_1,t_1),Damage\_Valid)=$ $\pi((t_1,p_2),Damage\_Valid)=$ $\pi((t_1,p_5),Damage\_Valid)=$ $\pi((p_2,t_2),Send\_Form)=$ $\pi((t_2,p_3),Send\_Form)=$ $\pi((p_3,t_3),Time\_Out)=$ $\pi((t_3,p_4),Time\_Out)=$ $\pi((p_3,t_4),Form\_Filled)=$ $\pi((t_4,p_4),Form\_Filled)=$ $\pi((p_5,t_6),Damage\_Adjustment)=$ $\pi((t_6,p_6),Damage\_Adjustment)=$ $\pi((p_6,t_7),Check\_info)=$ $\pi((t_7,p_7),Check\_info)=$ $\pi((p_4,t_7),Check\_info)=$ $\pi((t_7,p_4),Check\_info)=$ $\pi((p_7,t_9),Evaluate\_Situation)=$ $\pi((t_9,p_8),Evaluate\_Situation)=$

$\pi((p_8,t_{10}),E.Adjustment_2\uparrow\equiv)=$ $\pi((t_{10},p_0),E.Adjustment_2\uparrow\equiv)=$
$\pi((p_8,t_8),Aceptance)=$ $\pi((t_8,p_6),Aceptance)=$ $\pi((p_6,t_5),E.Adjustment_1\uparrow\equiv)=$
$\pi((p_4,t_5),E.Adjustment_1\uparrow\equiv)=$ $\pi((p_4,t_{10}),E.Adjustment_2\uparrow\equiv)=$
$\pi((t_5,p_0),E.Adjustment_1\uparrow\equiv)=\{a\}$

4. Para la tarea *Assessment* identificamos las siguientes operaciones:

   a. Abrir expediente de daños

   b. De cada registro calcular el costo actual del daño

   c. Estimar el costo del daño

   d. Calcular la liquidación final

   e. Aprobar liquidación

   El Modelo del grafo G quedaría como la figura 5.13.

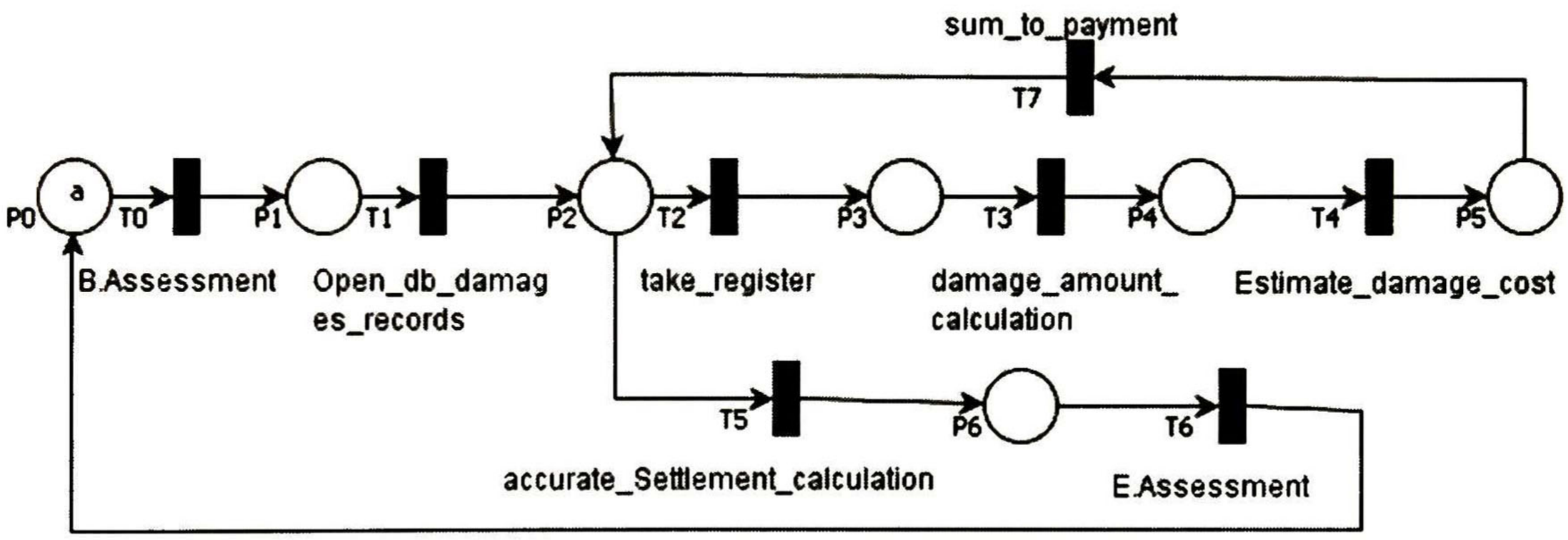


Figura 5.13: Red de Petri para el modelado de las operaciones de la tarea Assessment (Net3,5)

$Net_{3,5}$ =(G, $TOKEN_{3,5}$,$LABEL_{3,5}$,$VAR_{3,5}$, τ, λ, χ, π)

- $TOKEN_{3,5}$={a}
- $LABEL_{3,5}$={B.Assessment, Open_db_damage_records, take_register, damage_amount_calculation, Estimate_damage_cost, accurate_Settlement_calculation, sum_to_payment, E.Assessment$\uparrow\equiv$}
- $VAR_{3,5}=\varnothing$
- $\tau(p_0)=\tau(p_1)=\tau(p_2)=\tau(p_3)=\tau(p_4)=\tau(p_5)=\tau(p_6)=$ {a}
- $\lambda(t_0)$={B.Assessment$\uparrow$} $\lambda(t_1)$={Open_db_damage_records}
  $\lambda(t_2)$={take_register} $\lambda(t_3)$={damage_amount_calculation}

$\lambda(t_4)$={Estimate_damage_cost} $\lambda(t_5)$={accurate_Settlement_calculation}
$\lambda(t_6)$={E.Assessment↑≡} $\lambda(t_7)$={sum_to_payment}

- $\chi(t_6, E.Assessment\uparrow\equiv)$ (2,2)

- $\pi((p_0,t_0),B.Assessment\uparrow)=$ $\pi((t_0,p_1),B.Assessment\uparrow)=$
$\pi((p_1,t_1),Open\_db\_damage\_records)=$ $\pi((t_1,p_2),Open\_db\_damage\_records)=$
$\pi((p_2,t_2),take\_register)=$ $\pi((t_2,p_3),take\_register)=$
$\pi((p_3,t_3),damage\_amount\_calculation)=$
$\pi((t_3,p_4),damage\_amount\_calculation)=$ $\pi((p_4,t_4),Estimate\_damage\_cost)=$
$\pi((t_4,p_5),Estimate\_damage\_cost)=$ $\pi((p_5,t_7),sum\_to\_payment)=$
$\pi((t_7,p_2),sum\_to\_payment)=$ $\pi((p_2,t_5),accurate\_Settlement\_calculation)=$
$\pi((t_5,p_6),accurate\_Settlement\_calculation)=$ $\pi((p_6,t_6),E.Assessment\uparrow\equiv)=$
$\pi((t_6,p_0),E.Assessment\uparrow\equiv)=\{a\}$

5. Para la tarea *Payment* identificamos las siguientes operaciones:

   a. Abrir expediente de pagos

   b. Aprobar pago

   c. Verificar forma de pago

   d. Generar formato de recibo de pago de seguro

   e. Realizar pago

   El Modelo del grafo G quedaría como la figura 5.14.

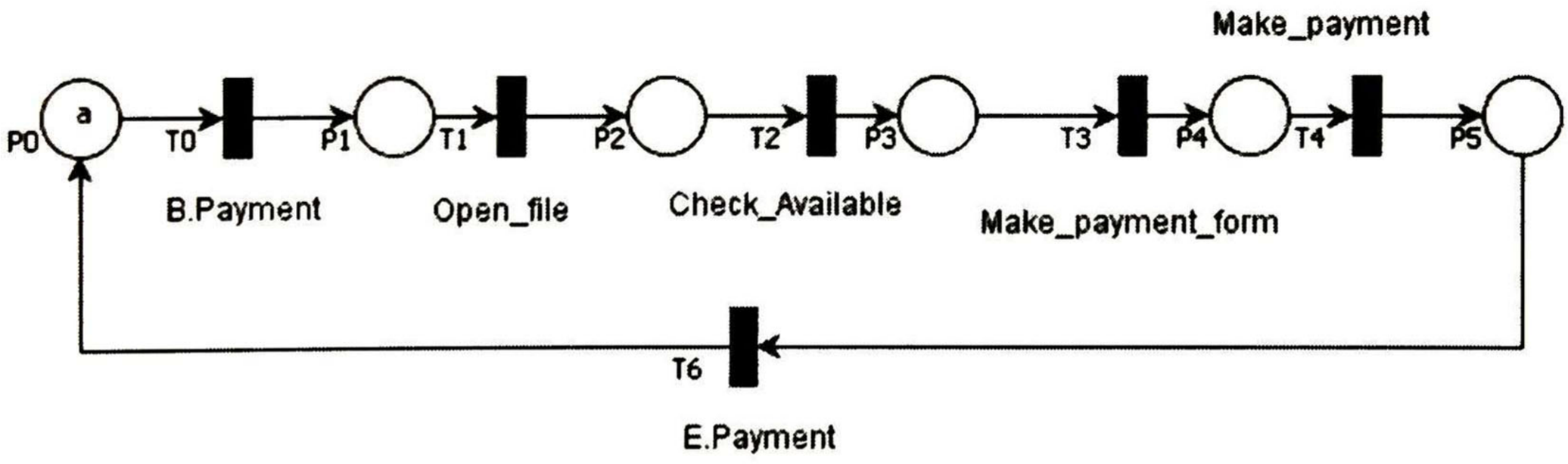


Figura 5.14:Red de Petri para el modelado de las operaciones de la tarea Payment (Net3,6)

$Net_{3,6}$ =(G, $TOKEN_{3,6}$,$LABEL_{3,6}$,$VAR_{3,6}$, τ, λ, χ, π)

- $TOKEN_{3,6}$={a}

- $LABEL_{3,6}$={B.Payment, Open_file, Check_Available, Make_payment_form, Make_payment, E.Payment}

- $VAR_{3,6}=\varnothing$
- $\tau(p_0)= \tau(p_1)= \tau(p_2)= \tau(p_3)= \tau(p_4)= \tau(p_5)= \{a\}$
- $\lambda(t_0)=\{B.Payment\uparrow\}$ $\lambda(t_1)=\{Open\_file\}$ $\lambda(t_2)=\{Check\_Available\}$ $\lambda(t_3)=\{Make\_payment\_form\}$ $\lambda(t_4)=\{Make\_payment\}$ $\lambda(t_6)=\{E.Payment\uparrow\equiv\}$
- $\chi(t_6,E.Payment\uparrow\equiv)=(2,2)$
- $\pi((p_0,t_0),B.Payment\uparrow\equiv)=$ $\pi((t_0,p_1),B.Payment\uparrow)=$ $\pi((p_1,t_1),Open\_file)=$ $\pi((t_1,p_2),Open\_file)=$ $\pi((p_2,t_2),Check\_Available)=$ $\pi((t_2,p_3),Check\_Available)=$ $\pi((p_3,t_3),Make\_payment\_form)=$ $\pi((t_3,p_4),Make\_payment\_form)=$ $\pi((p_4,t_4),Make\_payment)=$ $\pi((t_4,p_5),Make\_payment)=$ $\pi((p_5,t_6),E.Payment\uparrow\equiv)=$ $\pi((t_6,p_0),E.Payment\uparrow\equiv)=\{a\}$

6. Para la tarea *File*

   a. Recabar la información necesaria

   b. Llenar el expediente correcto

   El Modelo del grafo G quedaría la figura 5.15.

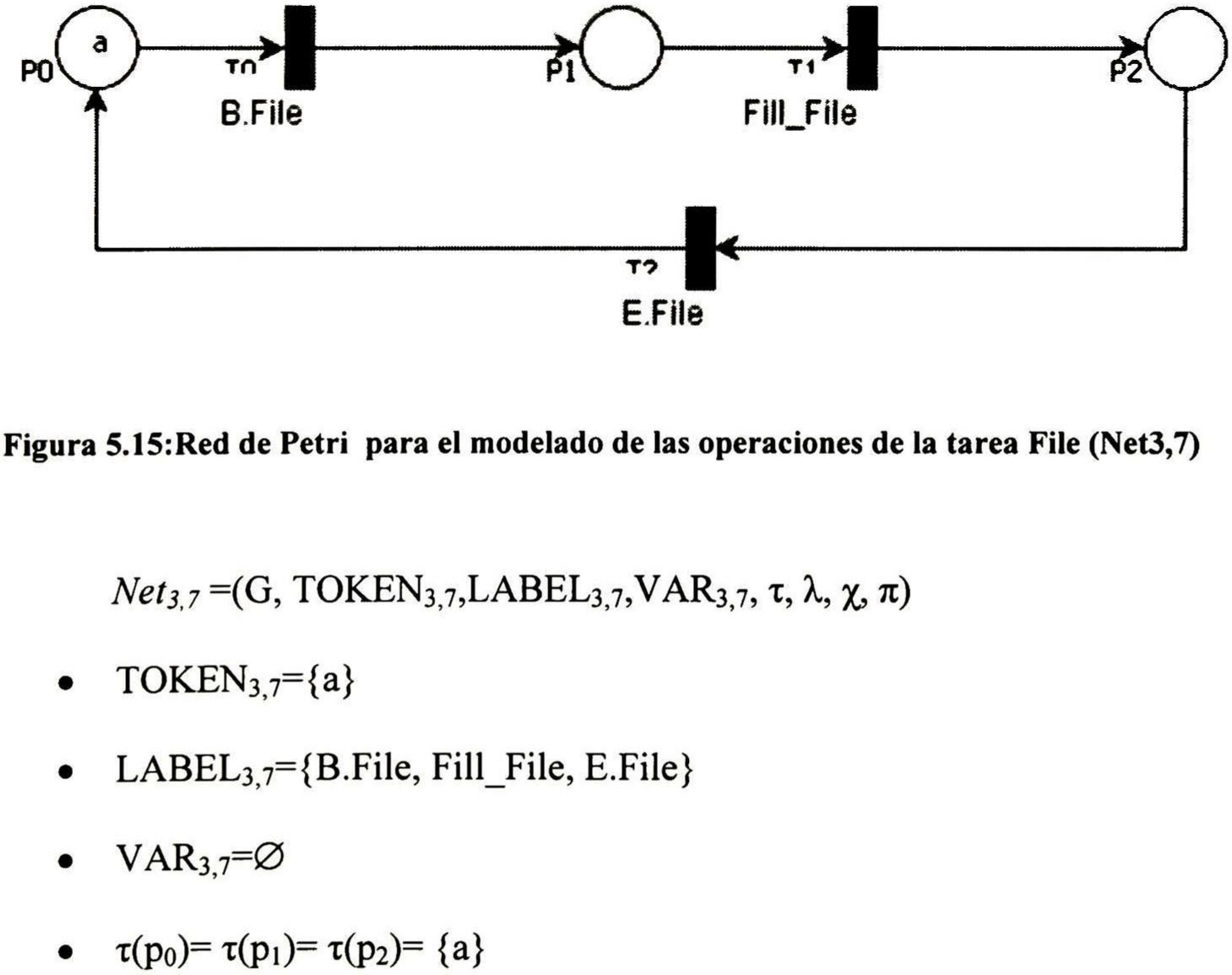


Figura 5.15:Red de Petri para el modelado de las operaciones de la tarea File (Net3,7)

$Net_{3,7}$ =(G, $TOKEN_{3,7}$,$LABEL_{3,7}$,$VAR_{3,7}$, τ, λ, χ, π)

- $TOKEN_{3,7}=\{a\}$
- $LABEL_{3,7}=\{B.File, Fill\_File, E.File\}$
- $VAR_{3,7}=\varnothing$
- $\tau(p_0)= \tau(p_1)= \tau(p_2)= \{a\}$
- $\lambda(t_0)=\{B.File\uparrow\}$ $\lambda(t_1)=\{Fill\_File\}$ $\lambda(t_2)=\{E.File\uparrow\equiv\}$

- $\chi(t_2,E.File\uparrow\equiv)= (2,2)$
- $\pi((p_0,t_0),B.File\uparrow)=$ $\pi((t_0,p_1),B.File\uparrow)=$ $\pi((p_1,t_1),Fill\_File)=$ $\pi((t_1,p_2),Fill\_File)=$ $\pi((p_2,t_2),E.File\uparrow\equiv)=$ $\pi((t_2,p_0),E.File\uparrow\equiv)=\{a\}$

5.3.3 Red del Segundo nivel

En este punto ya tenemos modelado el ambiente donde se moverá nuestro agente, el comportamiento del agente y cada una de las tareas que el agente debe realizar; así que ahora modelaremos el comportamiento general del agente, sin olvidar que ésta red estará contenida en la red del nivel ambiente ($Net_{1,1}$) y tendrá como marcado a las redes del nivel 3, agrupándose de la siguiente manera: en el lugar 1 (p_1) se colocarán inicialmente las redes que modelan a las tareas y en el lugar 2 (p_2) se colocará a la red que modela el comportamiento del agente ($Net_{3,1}$). Por lo que notamos que esta red deberá tener en sus transiciones a todas las etiquetas de sus redes internas que requieren sincronización externa en los lugares adecuados para lograr una sincronización en el inicio de cada tarea (para esto el plan lo lleva la red de nivel 3 $Net_{3,1}$) y el fin de cada tarea.

Así mismo deberá contener las etiquetas que nos representan el desplazamiento del agente de un lugar a otro, con sincronización tanto interna (pues estas mismas etiquetas las modelamos en la red interna $Net_{3,1}$) como externa (ya que se modelan en el ambiente del agente $Net_{1,1}$).

En el modelo además agregaremos una transición más por si el agente requiriera sincronización con otros agentes.

El modelo de esta red agente ($Net_{2,1}$) resultaría como lo indica la figura 5.16.

Definiendo cada uno de los valores de la red tenemos:

$Net_{2,1} =(G, TOKEN_{2,1}, LABEL_{2,1}, VAR_{2,1}, \tau, \lambda, \chi, \pi)$

- $TOKEN_{2,1}=\{ Net_{3,1}, Net_{3,2}, Net_{3,3}, Net_{3,4}, Net_{3,5}, Net_{3,6}, Net_{3,7}\}$
- $LABEL_{2,1}=\{$B.Claim_register, E.Claim_register, B.Data_Validation, E.Data_Validation$_1$, E.Data_Validation$_2$, B.Adjustment, E.Adjustment$_1$, E.Adjustment$_2$, B.Assessment, E.Assessment, B.Payment, E.Payment, B.File, E.File, mValida, mAdjust, mAssess, mPay, Mov, Cancel$\}$

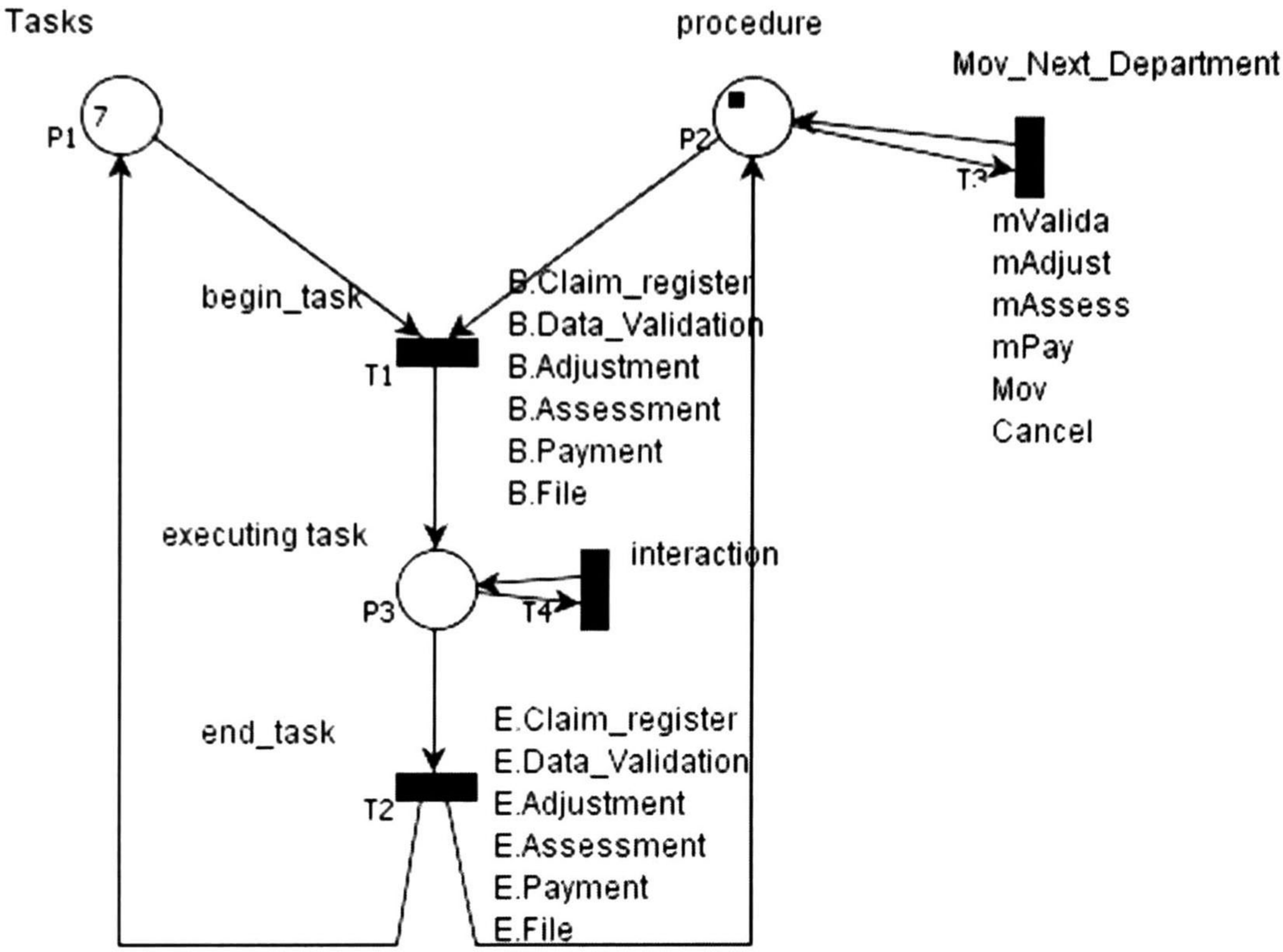


Figura 5.16: Red de Petri para la red de nivel 2 Net 2,1

- $VAR_{2,1}$={t:$Net_{3,1}$, u: $Net_{3,2}$, v: $Net_{3,3}$, w:$Net_{3,4}$, x: $Net_{3,5}$, y: $Net_{3,6}$, z: $Net_{3,7}$}

- $\tau(p_1)$= {$Net_{3,2}$, $Net_{3,3}$, $Net_{3,4}$, $Net_{3,5}$, $Net_{3,6}$, $Net_{3,7}$}

  $\tau(p_2)$= { $Net_{3,1}$}

  $\tau(p_3)$= {$Net_{3,2}$, $Net_{3,3}$, $Net_{3,4}$, $Net_{3,5}$, $Net_{3,6}$, $Net_{3,7}$, $Net_{3,1}$}

- $\lambda(t_1)$={B.Claim_register↓, B.Data_Validation↓, B.Adjustment↓, B.Assessment↓, B.Payment↓, B.File↓ }

  $\lambda(t_2)$={E.Claim_register↓, $E.Data\_Validation_1$↓, $E.Data\_Validation_2$↓, $E.Adjustment_1$↓, $E.Adjustment_2$↓, E.Assessment↓, E.Payment↓, E.File↓}

  $\lambda(t_3)$={mValida↑↓, mAdjust↑↓, mAssess↑↓, mPay↑↓, Mov↑↓, Cancel↑↓ }

- $\chi=\varnothing$

- $\pi((p_1,t_1)$,B.Claim_register↓)={$Net_{3,2}$}
  $\pi((p_1,t_1)$,B.Data_Validation↓)={$Net_{3,3}$} $\pi((p_1,t_1)$,B.Adjustment↓)={$Net_{3,4}$}

$\pi((p_1,t_1),B.Assessment\downarrow)=\{Net_{3,5}\}$ $\pi((p_1,t_1),B.Payment\downarrow)=\{Net_{3,6}\}$
$\pi((p_1,t_1),B.File\downarrow)=\{Net_{3,7}\}$

$\pi((p_2,t_1),B.Claim\_register\downarrow)=$ $\pi((p_2,t_1),B.Data\_Validation\downarrow)=$
$\pi((p_2,t_1),B.Adjustment\downarrow)=$ $\pi((p_2,t_1),B.Assessment\downarrow)=$
$\pi((p_2,t_1),B.Payment\downarrow)=$ $\pi((p_2,t_1),File\downarrow)=\{Net_{3,1}\}$

$\pi((t_1,p_3),B.Claim\_register\downarrow)=\{Net_{3,1}, Net_{3,2}\}$

$\pi((t_1,p_3),B.Data\_Validation\downarrow)=\{Net_{3,1}, Net_{3,3}\}$

$\pi((t_1,p_3), B.Adjustment\downarrow)=\{ Net_{3,1}, Net_{3,4} \}$

$\pi((t_1,p_3), B.Assessment\downarrow)=\{ Net_{3,1}, Net_{3,5}\}$

$\pi((t_1,p_3), B.Payment\downarrow)=\{ Net_{3,1}, Net_{3,6}\}$

$\pi((t_1,p_3), B.File\downarrow)=\{ Net_{3,1}, Net_{3,7} \}$

$\pi((p_3,t_2),E.Claim\_register\downarrow)=\{Net_{3,1}, Net_{3,2}\}$

$\pi((p_3,t_2),E.Data\_Validation\downarrow)=\{Net_{3,1}, Net_{3,3}\}$

$\pi((p_3,t_2), E.Adjustment\downarrow)=\{ Net_{3,1}, Net_{3,4} \}$

$\pi((p_3,t_2), E.Assessment\downarrow)=\{ Net_{3,1}, Net_{3,5}\}$

$\pi((p_3,t_2), E.Payment\downarrow)=\{ Net_{3,1}, Net_{3,6}\}$

$\pi((p_3,t_2), E.File\downarrow)=\{ Net_{3,1}, Net_{3,7} \}$

$\pi((t_2,p_1),E.Claim\_Register\downarrow)=\{ Net_{3,2}\}$

$\pi((t_2,p_1),E.Data\_Validation\downarrow)=\{ Net_{3,3}\}$

$\pi((t_2,p_1),E.Adjustment\downarrow)=\{ Net_{3,4}\}$

$\pi((t_2,p_1),E.Assessment\downarrow)=\{ Net_{3,5}\}$

$\pi((t_2,p_1),E.Payment\downarrow)=\{ Net_{3,6}\}$

$\pi((t_2,p_1),E.File\downarrow)=\{ Net_{3,7}\}$

$\pi((t_2,p_2),E.Claim\_Register\downarrow)=$ $\pi((t_2,p_2),E.Data\_Validation\downarrow)=$
$\pi((t_2,p_2),E.Adjustment\downarrow)=$ $\pi((t_2,p_2),E.Assessment\downarrow)=$ $\pi((t_2,p_2),E.Payment\downarrow)=$
$\pi((t_2,p_2),E.File\downarrow)=\{Net_{3,1}\}$

$\pi((p_2,t_3),Mov\uparrow\downarrow)=\pi((t_3,p_2),Mov\uparrow\downarrow)=$ $\pi((p_2,t_3),Cancel\uparrow\downarrow)=$
$\pi((t_3,p_2),Cancel\uparrow\downarrow)=$ $\pi((p_2,t_3),mValida\uparrow\downarrow)=$ $\pi((t_3,p_2),mValida\uparrow\downarrow)=$
$\pi((p_2,t_3),mAdjust\uparrow\downarrow)=$ $\pi((t_3,p_2),mAdjust\uparrow\downarrow)=$ $\pi((p_2,t_3),mAssess\uparrow\downarrow)=$
$\pi((t_3,p_2),mAssess\uparrow\downarrow)=$ $\pi((p_2,t_3),mPay\uparrow\downarrow)=\pi((t_3,p_2),mPay\uparrow\downarrow)=\{ Net_{3,1}\}$

5.3.4 Simulación

- *Sistema NLNs Agens*

El sistema NLNs Agents es un sistema que se encuentra en desarrollo como tesis de maestría, por la alumna del Cinvestav Guadalajara Mayra Padilla Duarte. Este sistema permite modelar redes en *n* niveles y realizar su simulación.

Nosotros utilizamos este sistema para modelar nuestro caso de estudio en 3 niveles con las redes mostradas anteriormente. Algunas de las pantallas de simulación son:

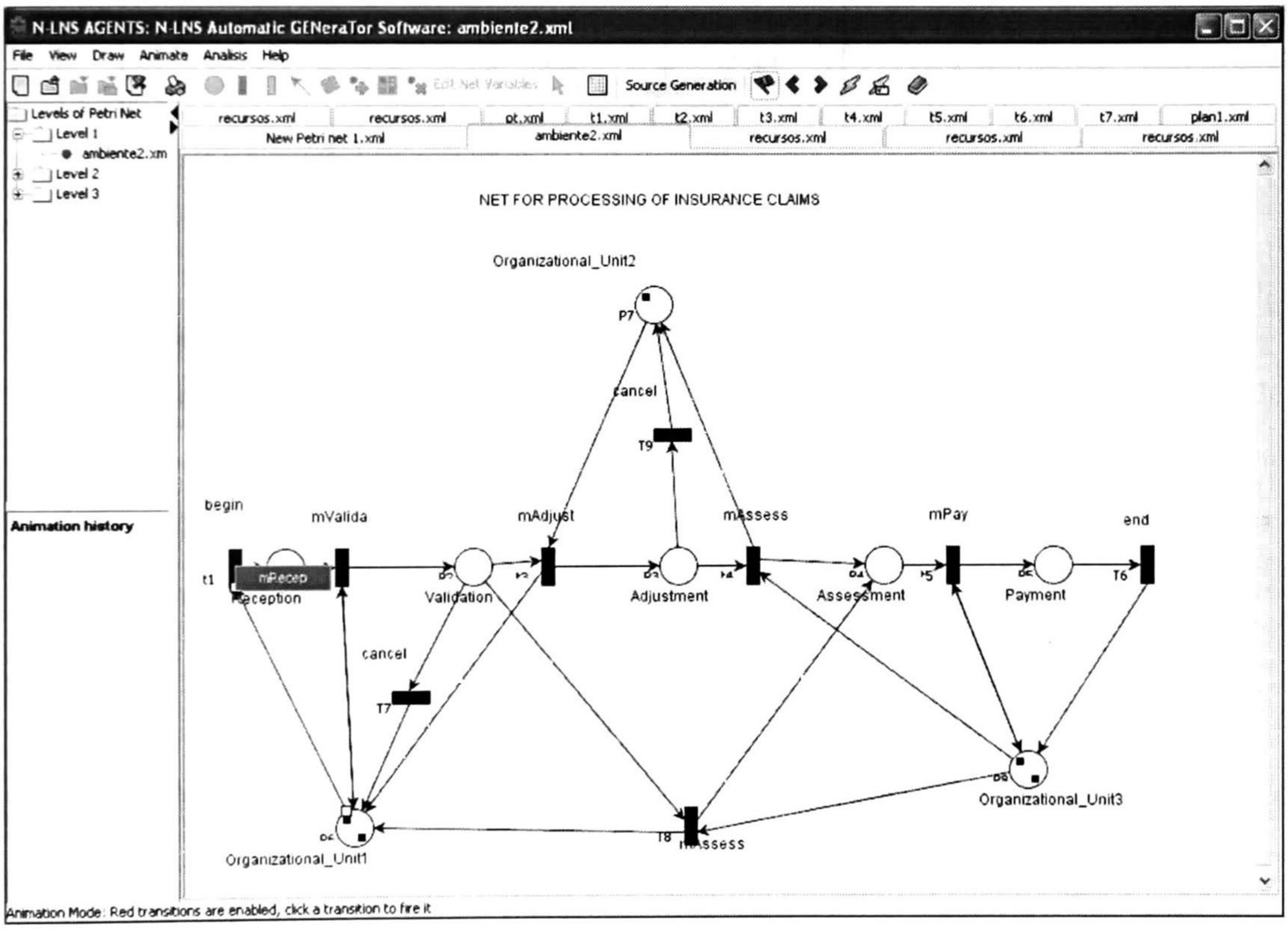


Figura 5.17: Ventana del Sistema NLNS AGENTS en la cual se está simulando la red de 3 niveles para nuestro caso de estudio. Red ambiente de nivel 1

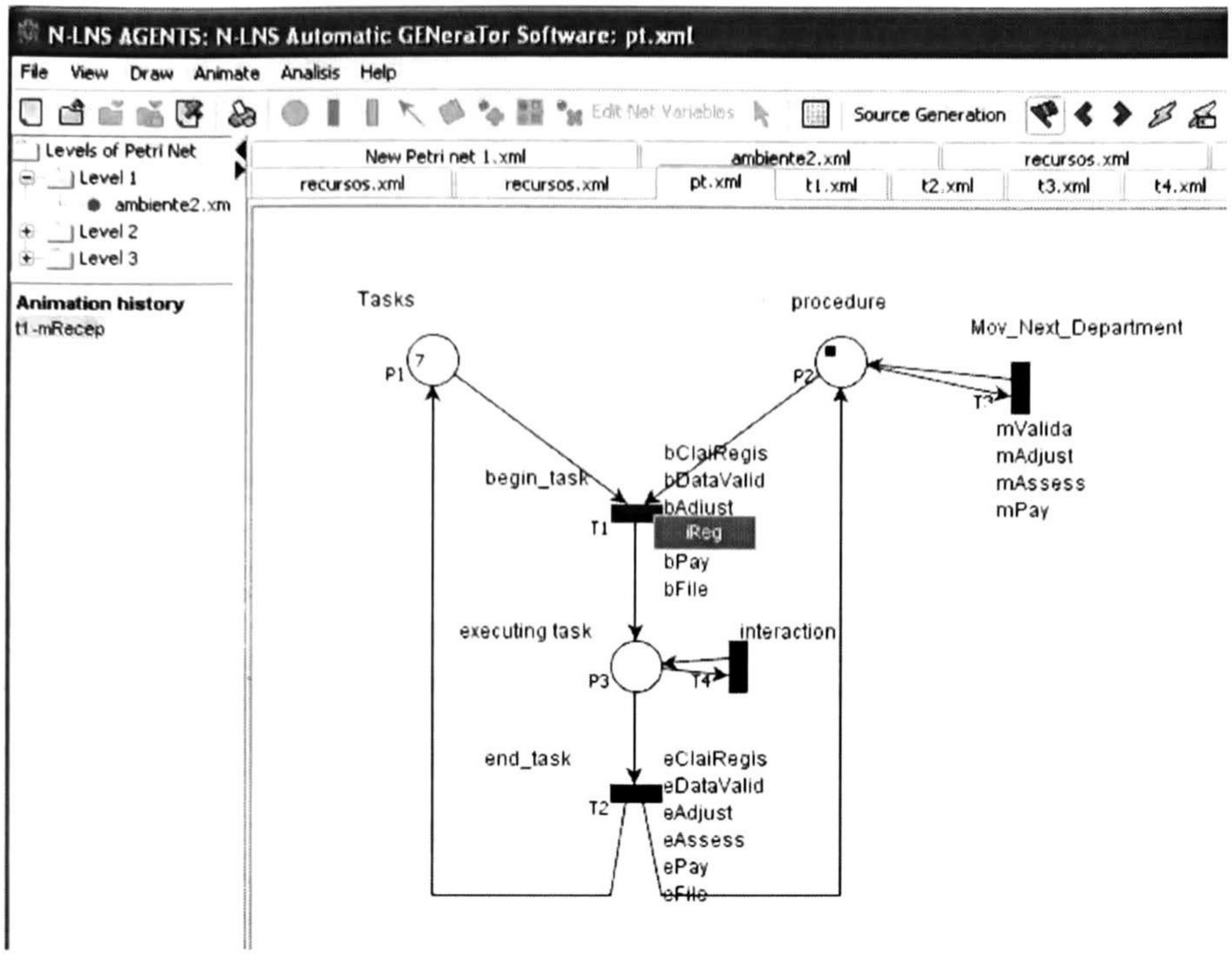


Figura 5.18: Estado inicial de la red de nivel 2 una vez iniciado el caso

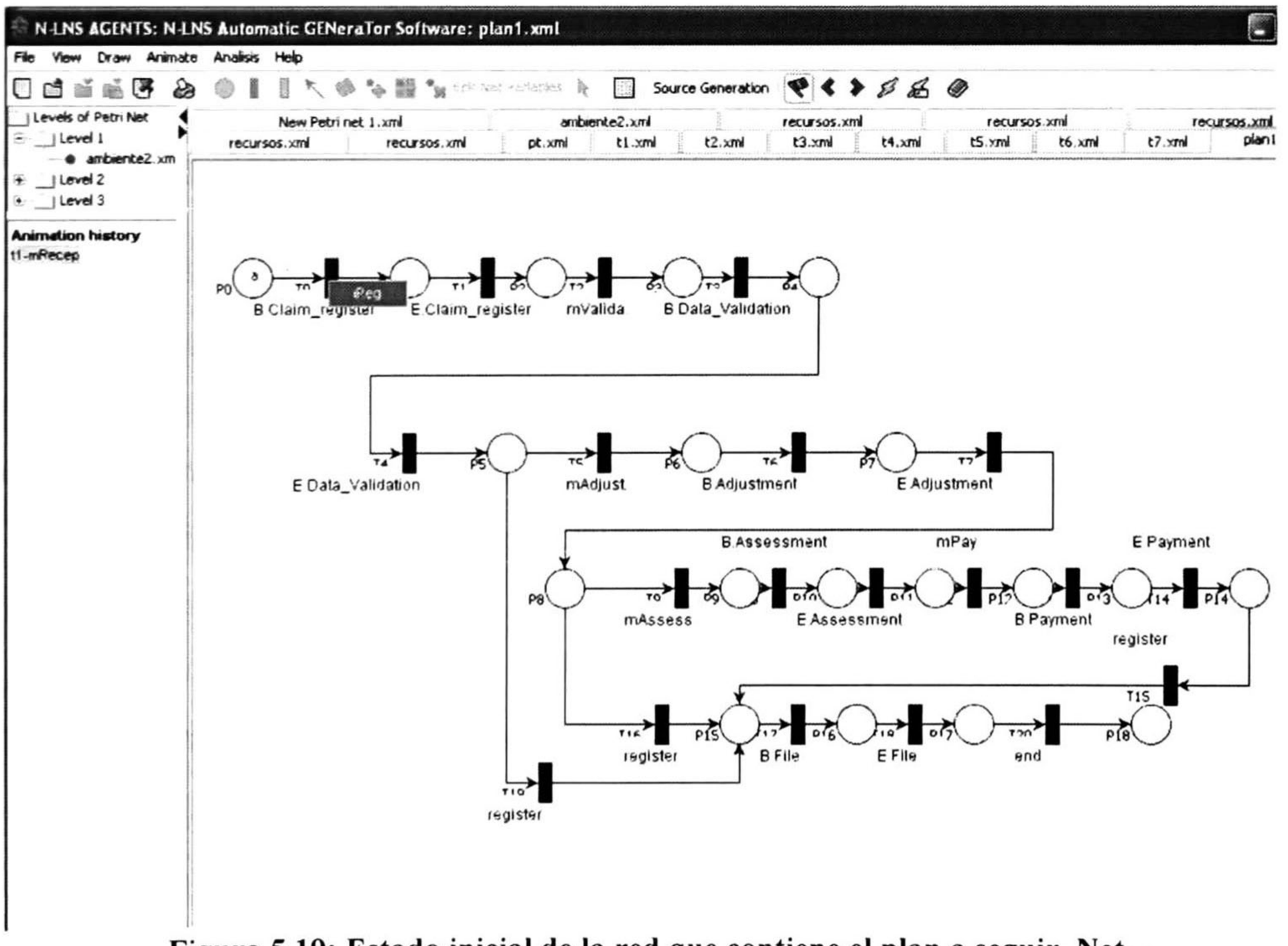


Figura 5.19: Estado inicial de la red que contiene el plan a seguir, $Net_{3,1}$

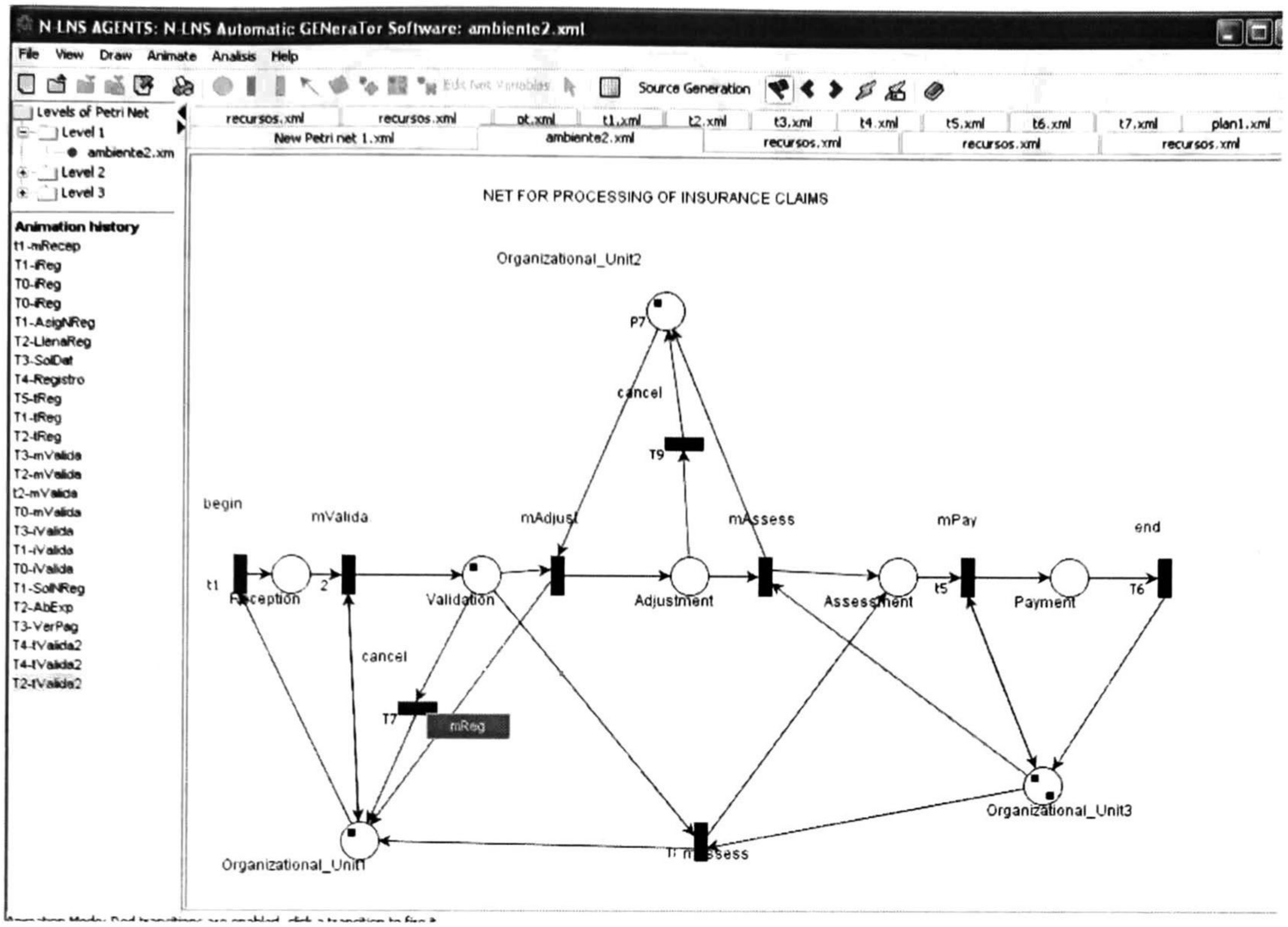


Figura 5.20: Red Ambiente una vez que se han disparado algunas transiciones

1.2 Codificación

Para la codificación de nuestro caso de estudio, además de utilizar la metodología propuesta, agregamos otros agentes (agentes estacionarios) y suponemos que se encuentran éstos en cada host de nuestro ambiente, de tal forma que el agente principal (modelado con la red $Net_{2,1}$) en lugar de realizar por sí mismo las tareas, pedirá a cada agente estacionario que ejecute la tarea correspondiente. Se supone que se cada agente estacionario solo conoce cómo realizar las tareas permitidas en el host (departamento) en el que está contenido, por lo que el agente móvil se moverá al host donde se encuentra el agente que sabe como realizar la tarea que necesita y le pide ayuda (para lo que utilizaremos el protocolo FIPARequest).

El modelo general quedaría según lo ilustrado en la figura 5.21.

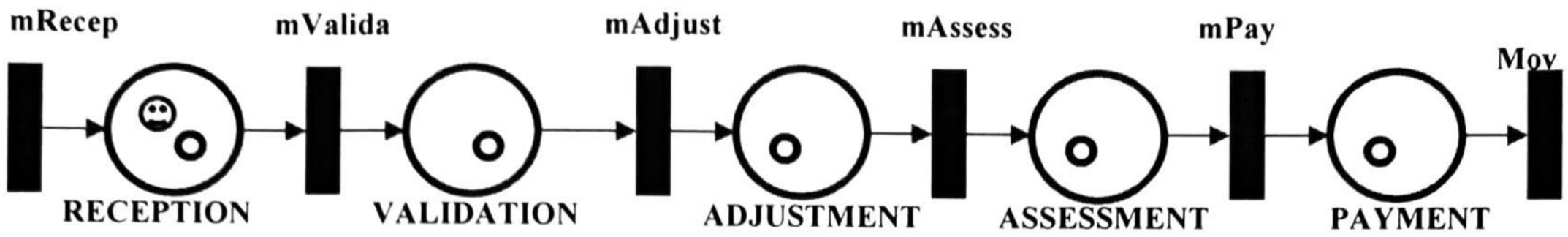


Figura 5.21: Modelo general del ambiente para la codificación del caso de estudio

Para nuestra aplicación supondremos la existencia de un agente usuario que es quien realiza la petición de inicio del proceso de la queja al agente estacionario del departamento de recepción, quien es el que va a crear al agente móvil adecuado para guiar al proceso de reclamo de queja correspondiente. Una vez creado el agente móvil, la comunicación con el usuario será a través de este agente. Por lo tanto, los agentes programaremos son:

- Agente estacionario Usuario: solicita el pago de un seguro al departamento de quejas de una compañía de seguros.
- Agente estacionario Recepción: recibe la petición del usuario (por medio de un mensaje) y crea al agente móvil que guiará el proceso de la queja recibido.
- Agente móvil: guiará el caso a través de los host desplazándose entre los mismos, se comunicará con los agentes estacionarios y con el agente usuario.
- Agentes estacionarios de cada departamento: recibirán peticiones por parte del agente móvil para que realicen una tarea, aceptarán la petición, ejecutarán la tarea y notificarán el éxito de la realización de la tarea solicitada.

Para cada agente programaremos una interfaz gráfica, por lo que cada clase programada deberá heredar de la clase *GuiAgent* en lugar de la clase *Agent*.

El fragmento de código para el agente **Usuario** quedaría:

```
protected void setup() {
            gui = new UserAgentGui(this);
            gui.setVisible(true);
    gui.disableSendButton();
    gui.displayOutPutText("Hallo!! User-Agent: " + getAID().getName()+ "is ready..");
    addBehaviour(new TickerBehaviour(this, 10000) {
                protected void onTick() {
                        if(!processEnabled){
                         gui.displayOutPutText("Trying to find Recepcionist Agents...");
    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("recepcionist");
    template.addServices(sd);
    try {
       DFAgentDescription[ ] result = DFService.search(myAgent, template);
     if(result.length>0)
      {
       gui.displayOutPutText("Recepcionist Agents already found...");
                    recepcionistAgents = new AID[result.length];
      for (int i = 0; i < result.length; ++i)
      recepcionistAgents[i] = result[i].getName();
            myAgent.addBehaviour(new RequestPerformer());
            myAgent.addBehaviour(new FailureMessages());
       }
      } catch (FIPAException fe) { fe.printStackTrace();  }
     }
            }
               }} );
}
```

En este código utilizamos un comportamiento llamado *TickerBehaviour* el cual se ejecutará con el objeto de buscar si hay agentes que tengan el servicio de *recepcionist* (a los que les enviaremos la queja), si no los hay lo intentará cada cierto tiempo hasta que se encuentren agentes que den este servicio. Además se agregan otros comportamientos, *FailureMessages* para procesar los mensajes que le envíen que sean de falla en la información enviada; *RequestPerformer* es el comportamiento principal del agente usuario y en él se programan los pasos de mensajes en los que se envía la información necesaria para el proceso de la queja.

El código de este último comportamiento podría quedar como:

```
private class RequestPerformer extends Behaviour {
                private MessageTemplate mt;
                private AID ContactAgent;
                ACLMessage request;
                public void action() {
                 switch (step) {
                 case 0:
                   //send complaint type to the recepcionist agent
                   ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
                   cfp.addReceiver(recepcionistAgents[0]);
                   cfp.setContent("complaint_type");
                   cfp.setConversationId("process-complaint");
                   cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
                   System.out.println("they must reply with: "  + cfp.getReplyWith());
                   myAgent.send(cfp);
                   // Prepare the template to get the reply
                   mt = MessageTemplate.and(MessageTemplate.MatchConversationId("process-complaint"),
                                MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
                   step = 1;
                   break;
               case 1:
                    ACLMessage reply = myAgent.receive(mt);
                   if (reply != null) {
                    gui.displayOutPutText("Receiving a Message from the Complaint Department...");
                    gui.displayOutPutText("Message Received..");
                    if (reply.getPerformative() == ACLMessage.INFORM) {
                    try{ //reading content message
                     Package1 p1=(Package1)reply.getContentObject();
                     ComplaintNm =p1.ComplaintNm; //complaint number
                     ContactAgent=p1.AgentAID;
                    }catch(UnreadableException e3){  System.err.println(getLocalName()+ " catched exception "+e3.getMessage()); }
                    step=2;        }
                   }
                   else { block();  }
                   break;
                case 2://receiving message
                    mt = MessageTemplate.and(MessageTemplate.MatchSender(ContactAgent),
                          MessageTemplate.MatchPerformative(ACLMessage.REQUEST));
                    request= myAgent.receive(mt);
                    if(request!=null){   step=3;  }   else      {block();}
                      break;
               case 3://sending message with personal data
                    try{
                    ACLMessage  reply2 = request.createReply();
                    reply2.setPerformative(ACLMessage.INFORM);
                    String  comNum=gui.getComplaintNumberField();
                    Person myData= new Person(gui.getNameField(), gui.getLastNameField(), Long.valueOf(comNum),new Date());
                    reply2.setContentObject(myData);
                    myAgent.send(reply2);
                    }catch (Exception e ){  System.err.println(getLocalName()+ " catched exception "+e.getMessage());  }
                    myAgent.addBehaviour(new InformMessages());
                    step=5;
                    break;
                }
                 }
                public boolean done() {
                 return step==5;
                }
               }  // End of inner class RequestPerformer
```

El agente **Recepcionista** deberá: registrarse y registrar su servicio con el ***DF***, deberá tener un comportamiento que reciba mensajes de los usuarios e inicie la ejecución de un agente móvil para que atienda y siga el proceso de la queja recibida. Un fragmento de este código quedaría como:

```
protected void setup() {
 gui = new RecepcionistAgentGui(this);
  gui.setVisible(true);
  cntComplaint=1;
  DFAgentDescription dfd = new DFAgentDescription();
  dfd.setName(getAID());
 gui.displayOutPutText("Hallo!! Recepcionist-Agent "+getAID().getName());
  ServiceDescription sd = new ServiceDescription();
  sd.setType("recepcionist");
  sd.setName("JADE-process-complaint");
  dfd.addServices(sd);
  try { DFService.register(this, dfd);}
  catch (FIPAException fe) {fe.printStackTrace();}
  addBehaviour(new OfferRequestsServer());
}

          private class OfferRequestsServer extends CyclicBehaviour {
           public void action() {
                    MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
                    ACLMessage msg = myAgent.receive(mt);
          if (msg != null) {
           gui.displayOutPutText("The content of his message is "+ msg.getContent().toString());
           ACLMessage reply = msg.createReply();
            reply.setPerformative(ACLMessage.INFORM);
       gui.displayOutPutText("**Creating the mobile agent according to the complaint type***");
            PlatformController container = getContainerController(); // get a container controller for creating new
agents
       try {
        String localName = "mobil_" + cntComplaint;
                          AgentController mobil = container.createNewAgent(localName, "MobileAgent", null);
                          mobil.start();
                          ACLMessage msg2 = new ACLMessage(ACLMessage.INFORM);
            msg2.addReceiver(new AID(localName,AID.ISLOCALNAME));
            msg2.setReplyWith(msg.getReplyWith());
            msg2.setConversationId("process-complaint");
            String sComplaint=""+cntComplaint;
            Package1 p1= new Package1(sComplaint, new AID(localName,AID.ISLOCALNAME));
            Package1 p2= new Package1(sComplaint,msg.getSender());
            reply.setContentObject(p1);
             msg2.setContentObject(p2);
              myAgent.send(msg2);//message to the mobile agent
              myAgent.send(reply); //message to the user
       cntComplaint++;
        }
        catch (Exception e) {System.err.println( "Exception while adding guests: " + e );
        e.printStackTrace();}
            }
                    else { block(); }
          }
        } // End of inner class OfferRequestsServer
```

Para el agente móvil utilizamos el comportamiento ***FSMBehaviour*** el cual nos ayuda a programar el comportamiento general del agente modelado en la red $Net_{3,1}$, el fragmento de código donde se declara este comportamiento es:

```
private static final String St_RegSol  ="1";
private static final String St_ValVig  ="2";
private static final String St_ValDama ="3";
private static final String St_Cotiz   ="4";
private static final String St_Pay     ="5";
private static final String St_Reg     ="10";
private static final String St_MovVal  ="6";
private static final String St_MovValo ="7";
private static final String St_MovCotiz="8";
private static final String St_MovPay  ="9";

 private final int VALID =0;
private final int NOVALID =1;
FSMBehaviour fsm = new FSMBehaviour(this){
                    public int onEnd(){
                                System.out.println("AGENT  " + getAID().getName()+ " is completed");
                                myAgent.doDelete();
                                return super.onEnd();
                    } };

fsm.registerFirstState(new RegSolBehaviour(),St_RegSol);
fsm.registerState(new ValVigBehaviour(),St_ValVig);
fsm.registerState(new ValDamaBehaviour(),St_ValDama);
fsm.registerState(new CotizBehaviour(),St_Cotiz);
fsm.registerState(new PayBehaviour(),St_Pay);
fsm.registerLastState(new RegBehaviour(),St_Reg);
fsm.registerState(new MovValBehaviour(),St_MovVal);
fsm.registerState(new MovValoBehaviour(),St_MovValo);
fsm.registerState(new MovCotizBehaviour(),St_MovCotiz);
fsm.registerState(new MovPayBehaviour(),St_MovPay);

fsm.registerDefaultTransition(St_RegSol,St_MovVal);
fsm.registerDefaultTransition(St_MovVal,St_ValVig);
fsm.registerTransition(St_ValVig,St_MovValo,VALID);
fsm.registerTransition(St_ValVig,St_Reg,NOVALID);
fsm.registerDefaultTransition(St_MovValo,St_ValDama);
fsm.registerTransition(St_ValDama,St_MovCotiz,VALID);
fsm.registerTransition(St_ValDama,St_Reg,NOVALID);
fsm.registerDefaultTransition(St_MovCotiz,St_Cotiz);
fsm.registerDefaultTransition(St_Cotiz,St_MovPay);
fsm.registerDefaultTransition(St_MovPay,St_Pay);
fsm.registerDefaultTransition(St_Pay,St_Reg);


addBehaviour(fsm);
```

En donde, como ya se explico en el capítulo anterior, cada estado registrado en el comportamiento, es una función que el agente debe realizar. Algunas de estas funciones podrían quedar como:

```
private class MovValBehaviour extends Behaviour{
        private int step = 1;
        private int exitValue;
        public void action(){
                switch(step){
                case 1:
                        gui.displayOutPutText("");
                        gui.displayOutPutText("Executing Behaviour: Move_to_Validation_Department...");
                        addBehaviour(new GetAvailableLocationsBehaviour((MobileAgent)myAgent));
                        listFull=false;
                        step=2;
                        break;
        case 2:   if(listFull==true)step=3;
                        break;
        case 3:
                        moving=false;
                        NextDepartment="validation";
                        addBehaviour(new MoveToNextDepartment1());
                       step=4;
                       break;
        case 4:
                       if(moving==true){
                       step=5;
                       System.out.println("the agent is already moved...");
                       moving=false;
                       exitValue=0;                        }
                        break;
       }}
public boolean done() {
      return step == 5;      }
            public int onEnd(){
            return exitValue;}
            }//end BehaviourMOVVALVIG
protected void onGuiEvent(GuiEvent ev){
            switch(ev.getType()) {
            case EXIT:
                              System.out.println("salir");
                                    gui.dispose();
                                    gui = null;
                                    doDelete();
                                    break;
            case MOVE_EVENT:
                        doMove(nextSite);
                                    gui.disableMoveButton();
                                    break;
            case CONTINUE:
                        fillinglist();
                        break;
                }}
protected void afterMove() {
    System.out.println(getLocalName()+" is just arrived to this location.");
    gui = new MobileAgentGui(this);
    if(nextSite != null)   {
     visitedLocations.addElement(nextSite);
     for (int i=0; i<visitedLocations.size(); i++)
       gui.addVisitedSite((Location)visitedLocations.elementAt(i));
      }
    gui.setVisible(true);
    gui.displayOutPutText("Thi Agent " + getLocalName() +"is just arrived to this location...");
    moving=true;
    getContentManager().registerLanguage(new SLCodec(), FIPANames.ContentLanguage.FIPA_SL0);
    getContentManager().registerOntology(MobilityOntology.getInstance());
  }
```

```
private class MoveToNextDepartment1 extends OneShotBehaviour {
 private MessageTemplate mt;
 private  AID[] Agents;
 int iaux;
public void action(){
        Location s;
        DFAgentDescription template;
       ServiceDescription sd;
       DFAgentDescription[] result;
       Boolean done=false;
       template = new DFAgentDescription();
       sd = new ServiceDescription();
       sd.setType(NextDepartment);
       template.addServices(sd);
       try {
            while(done==false){
          result = DFService.search(myAgent, template);
          if(result.length !=0){
            done=true;
     Agents = new AID[result.length];
  for (int i = 0; i < result.length; ++i)
     Agents[i] = result[i].getName();
}
  else {
       try{
       gui.displayOutPutText("I didn't find " + NextDepartment + " agents...");
       }catch(Exception e){e.printStackTrace();}
  } }
}catch (FIPAException fe) {  fe.printStackTrace(); }
 ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
cfp.addReceiver(Agents[0]);
cfp.setContent("request-container");
cfp.setConversationId("process-complaint");
cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
myAgent.send(cfp);
mt = MessageTemplate.and(MessageTemplate.MatchSender(Agents[0])/*.MatchConversationId("process-complaint")*/,
                         MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
String containerA;
ACLMessage replyC = myAgent.blockingReceive(mt);
if (replyC != null) {
             containerA=replyC.getContent();
            gui.displayOutPutText("The agent " + Agents[0].getName()+ " .. is in container....." + containerA);
             currentAgentName=replyC.getSender();
           if(availableSiteListModel.getRowCount()>0){
            s=availableSiteListModel.getElementAt(0);
                  for(int i=0;i<availableSiteListModel.getRowCount();i++)     {
                              s=availableSiteListModel.getElementAt(i);
                              if(containerA.compareToIgnoreCase(s.getName())==0)
                               break;
                  }
           nextSite=s;
          gui.displayOutPutText("This Agent is going to move to the direction: " +nextSite.getAddress());
    }
 else
 {  System.out.println(" I can´t move to he next department.....");
     doDelete();    }
        }   }
}//end of inner class movetonextdepartment
```

En la función *afterMove()* observamos que se tiene que volver a iniciar la *GUI* y se debe de registrar otra vez la ontología de movilidad. Utilizamos una función *MoveToNextDepartment1* en la cual solicitamos al agente que se encuentra en ese

departamento información sobre el contenedor en el que se encuentra y buscamos dicha información en nuestra lista de localidades disponibles (la cual contiene información acerca de las localidades que se encuentran registradas en la plataforma y que actualizamos con la función *GetAvailableLocations*)

En la función de *GetAvailableLocatios,* hacemos una petición al **AMS** para solicitarle información sobre las localidades disponibles, lo cual lo logramos con el siguiente código:

```
public class GetAvailableLocationsBehaviour extends SimpleAchieveREInitiator {
   private ACLMessage request;
   public GetAvailableLocationsBehaviour(MobileAgent a) {
    super(a, new ACLMessage(ACLMessage.REQUEST));
    request = (ACLMessage)getDataStore().get(REQUEST_KEY);
    request.clearAllReceiver();
    request.addReceiver(a.getAMS());
    request.setLanguage(FIPANames.ContentLanguage.FIPA_SL0);
    request.setOntology(MobilityOntology.NAME);
    request.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
     try {
     Action action = new Action();
     action.setActor(a.getAMS());
     action.setAction(new QueryPlatformLocationsAction());
     a.getContentManager().fillContent(request, action);
    }
    catch(Exception fe) {
                fe.printStackTrace();
     }
      reset(request);
}

protected void handleNotUnderstood(ACLMessage reply) {
 System.out.println(myAgent.getLocalName()+ " handleNotUnderstood : "+reply.toString());
}
protected void handleRefuse(ACLMessage reply) {
 System.out.println(myAgent.getLocalName()+ " handleRefuse : "+reply.toString());
}
protected void handleFailure(ACLMessage reply) {
 System.out.println(myAgent.getLocalName()+ " handleFailure : "+reply.toString());
}
protected void handleAgree(ACLMessage reply) {
}
protected void handleInform(ACLMessage inform) {
 String content = inform.getContent();
 try {
  final  Result results = (Result)myAgent.getContentManager().extractContent(inform);
                                                          updateLocations(results.getItems().iterator()),
 }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}//end of Get...
```

Para solicitar a un agente ayuda para que realice una tarea utilizamos el protocolo FIPA-Request, el cual puede implementarse con la siguiente función donde el parámetro *task* es el nombre de la tarea a solicitar.

```
protected void RequestIniciator(final String task)
{                   ACLMessage msgRequest = new ACLMessage(ACLMessage.REQUEST);
                    gui.displayOutPutText("Sending to the Agent : " + currentAgentName.getName());
                    msgRequest.addReceiver(currentAgentName);
                    msgRequest.setProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST);
                     msgRequest.setConversationId(sComplaint);
                     msgRequest.setReplyByDate(new Date(System.currentTimeMillis() + 10000));
                     msgRequest.setContent(task);
                     addBehaviour(new AchieveREInitiator(/*(MobileAgent)myAgent*/this, msgRequest) {
          protected void handleInform(ACLMessage inform) {
            gui.displayOutPutText("the Agent" +inform.getSender().getName()+" has already done succesfully the
task ---'' + task);
          }
          protected void handleRefuse(ACLMessage refuse) {
             gui.displayOutPutText("Agent "+refuse.getSender().getName()+" refused to perform the requested
action");
          }
          protected void handleFailure(ACLMessage failure) {
            if (failure.getSender().equals(myAgent.getAMS())) {
                    System.out.println("Responder does not exist");
             }
            else {
             gui.displayOutPutText("Agent "+failure.getSender().getName()+" found that the information is
invalid...");          }
          }
          protected void handleAllResultNotifications(Vector notifications) {
                    if (notifications.size() < 1) {  // Some responder didn't reply within the specified timeout
                    System.out.println("Timeout expired: missing  responses");
                    } } } );
          }
```

Cada agente estacionario se programó para completar el protocolo de FIPARequest, por ejemplo el agente PaymentAgent (el agente estacionario del departamento *Payment*) tendría que contener un comportamiento para completar el protocolo de la siguiente manera:

```
private class FIPARequestResponderBehaviour extends CyclicBehaviour {
            private int stepAux = 0;
  public void action() {
      if(stepAux==0){
          gui.displayOutPutText("Agent " + getLocalName() + "  free and waiting for request...");
          System.out.println("Agent "+getLocalName()+" waiting for requests...");
          MessageTemplate template = MessageTemplate.and(
          MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST),
          MessageTemplate.MatchPerformative(ACLMessage.REQUEST) );
           stepAux=1;
          addBehaviour(new AchieveREResponder((PaymentAgent)myAgent, template) {
          protected ACLMessage prepareResponse(ACLMessage request) throws NotUnderstoodException,
RefuseException {
                        gui.displayOutPutText("REQUEST receiven from : " +request.getSender().getName());
                        gui.displayOutPutText("The REQUESTED ACTION is: " + request.getContent());
                        System.out.println("Agent "+getLocalName()+": REQUEST received from
"+request.getSender().getName()+". Action is "+request.getContent());
                        if (checkAction()) {
                         // We agree to perform the action. Note that in the FIPA-Request
                           gui.displayOutPutText("This Agent is Agree with the requested Action....");
                           System.out.println("Agent "+getLocalName()+": Agree");
                          ACLMessage agree = request.createReply();
                          agree.setPerformative(ACLMessage.AGREE);
                         return agree;
                        }
                        else {
                                // We refuse to perform the action
                                System.out.println("Agent "+getLocalName()+": Refuse");
                                throw new RefuseException("check-failed");
                           }
                      }
        protected ACLMessage prepareResultNotification(ACLMessage request, ACLMessage response) throws
FailureException {
                        System.out.println("I have to do  ....." + request.getContent());
                        if (performAction(request.getContent())) {
                        System.out.println("Agent "+getLocalName()+": Action successfully performed, the information
is valid");
                           stepAux=0;
                        ACLMessage inform = request.createReply();
                        inform.setPerformative(ACLMessage.INFORM);
                        return inform;
                        }else {
                           stepAux=0;
                           System.out.println("Agent "+getLocalName()+": Action failed, information invalid");
                            throw new FailureException("unexpected-error");
                        }          }
                        } );
          } }

          }  // End
```

1.2.1 Prototipo

Algunas de las interfaces gráficas de los agentes descritos anteriormente son:

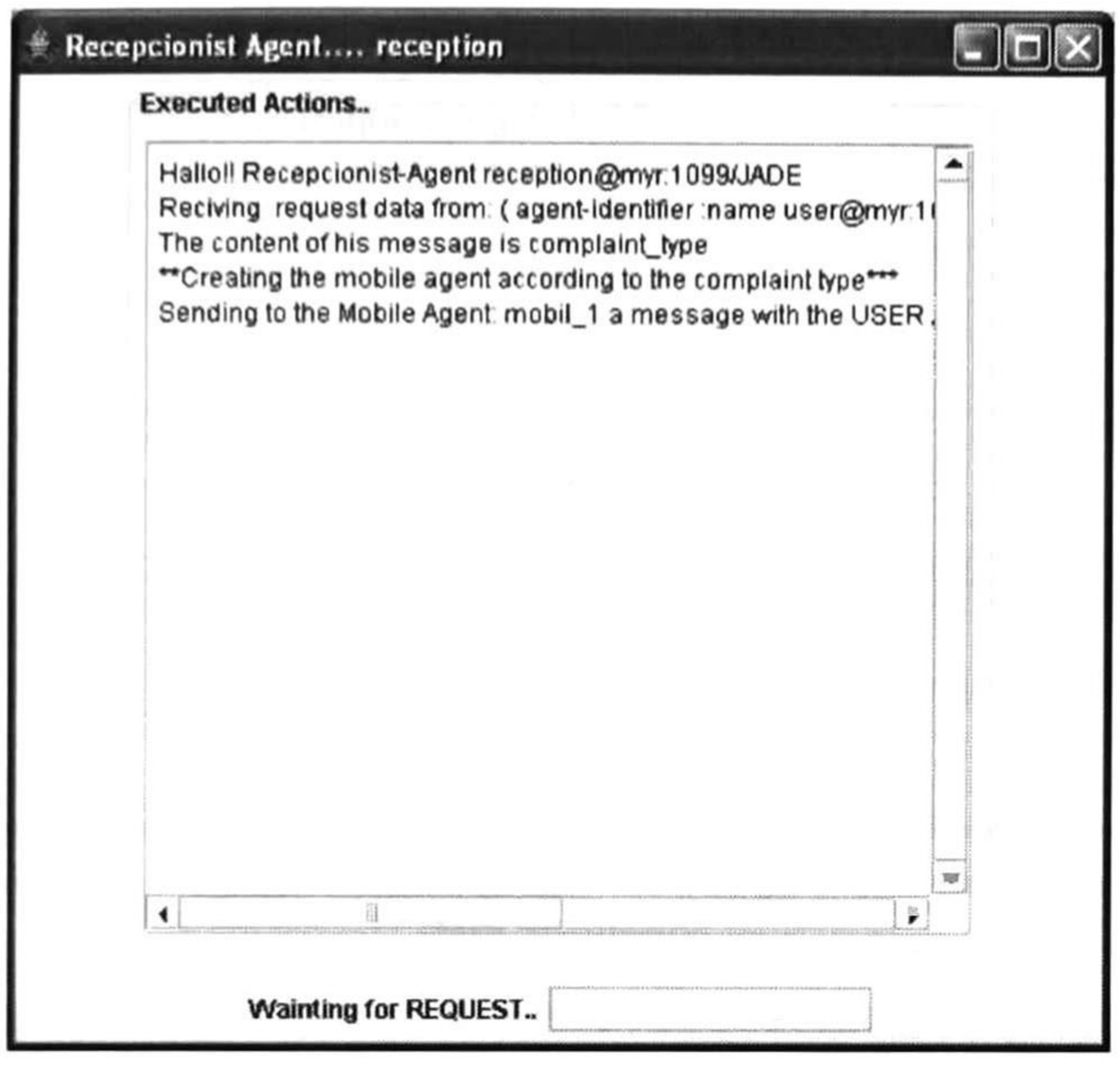


Figura 5.22: Interfaz gráfica del Agente Recepcionista

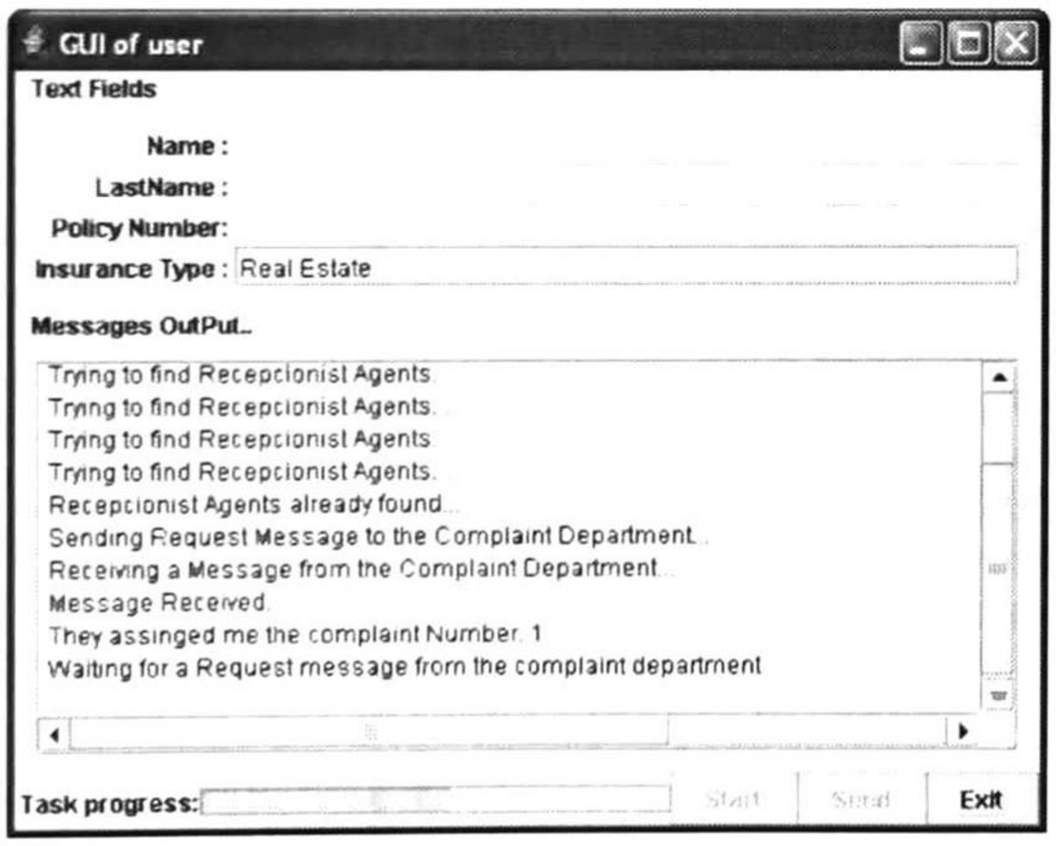


Figura 5.23: Interfaz del agente Usuario

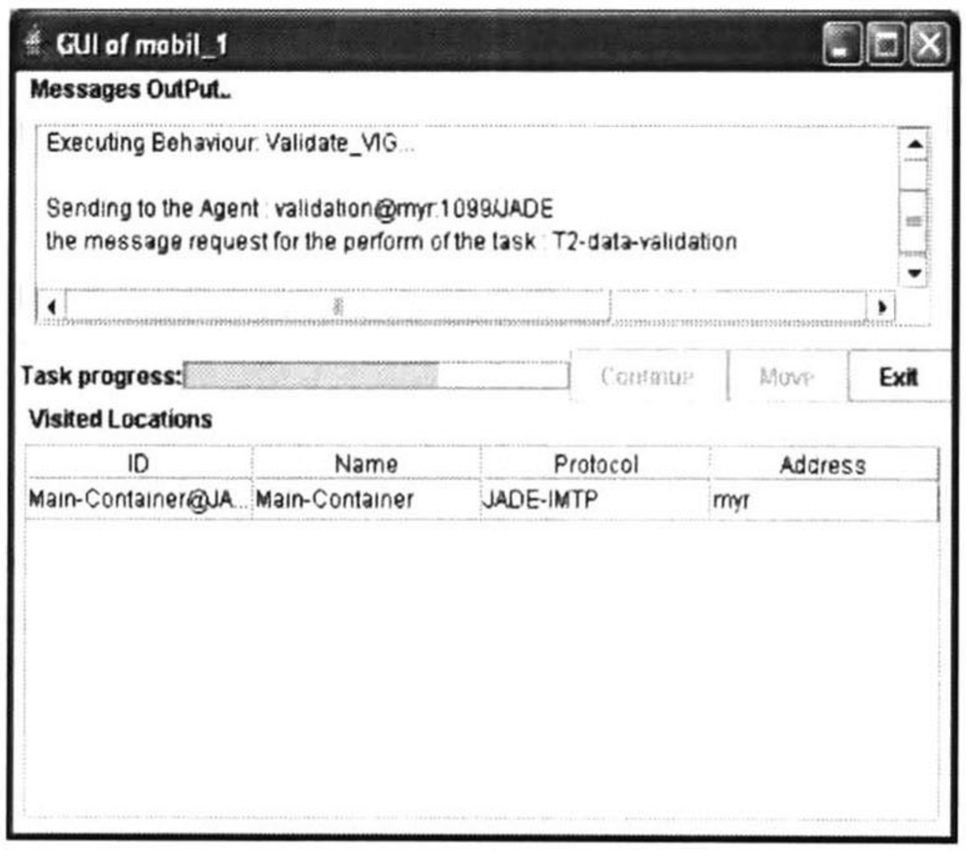


Figura 5.24: Interfaz del Agente Móvil una vez que ha movido al host2 e iniciado la petición por la tarea Data_validation

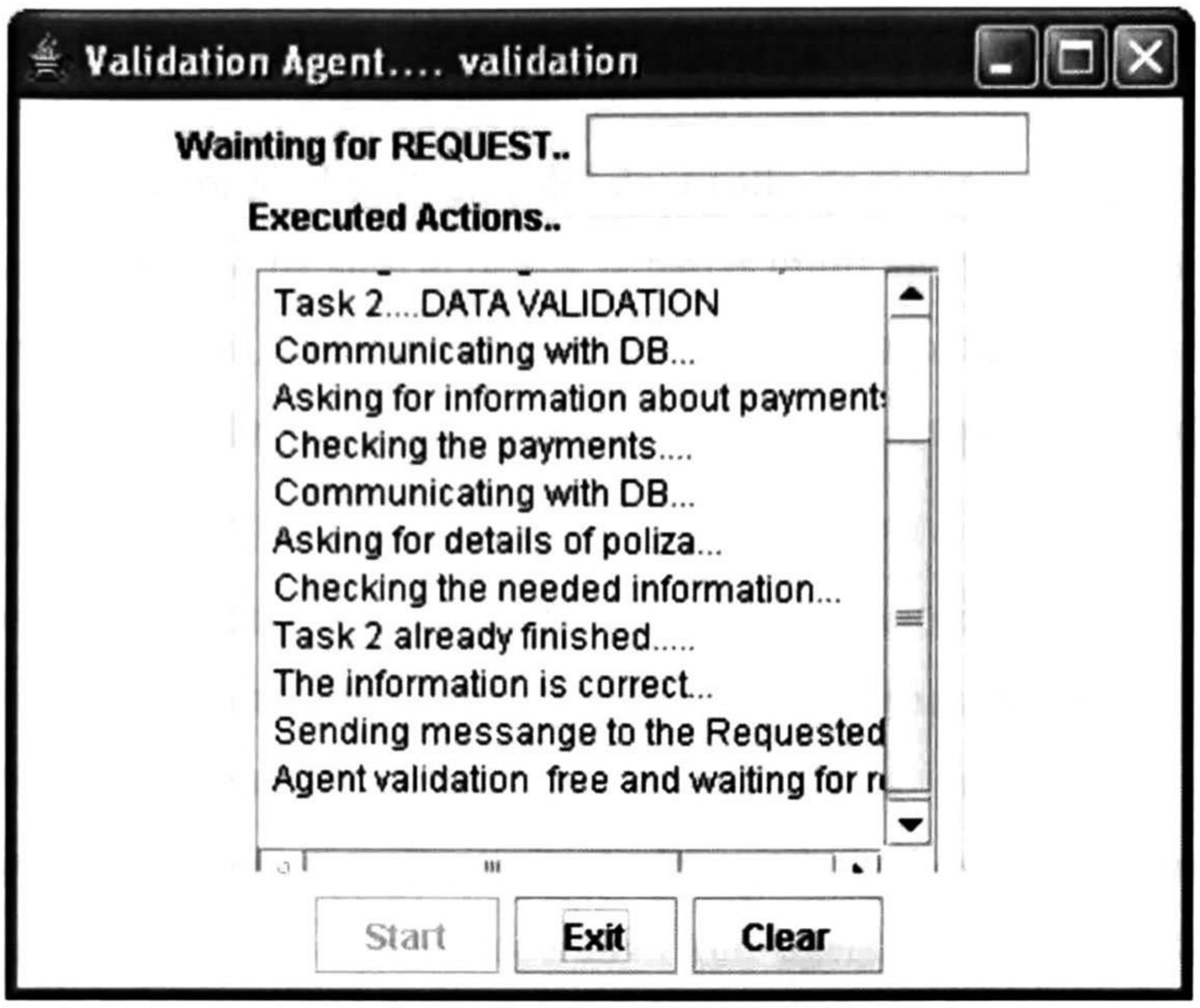


Figura 5.25: Interfaz del Agente Validation una vez que ha realizado la ejecución de la tarea Data_validation

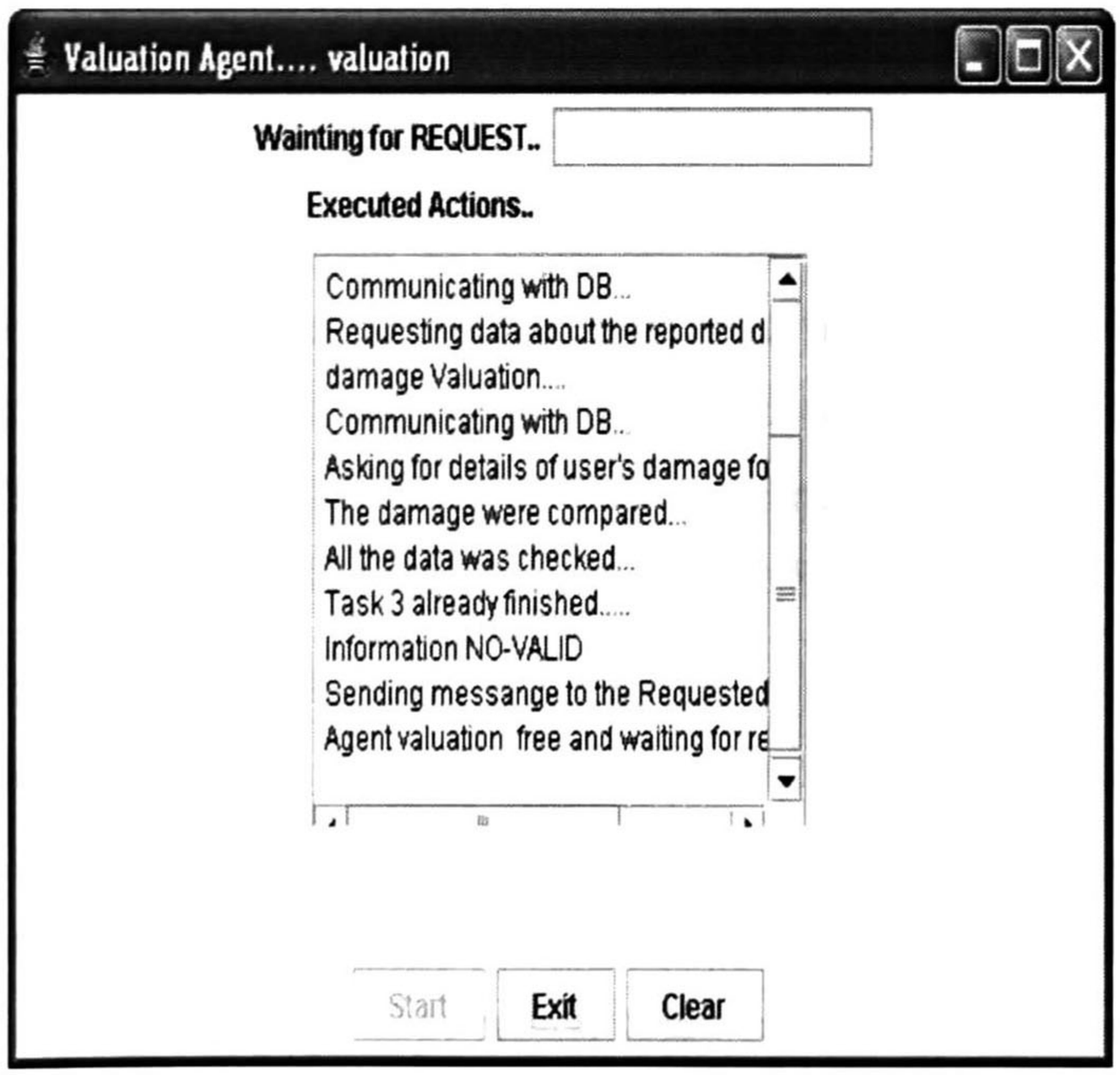


Figura 5.26: Interfaz del Agente Valuation, una vez que ha realizado la tarea de Adjustment

Conclusiones

En esta tesis se abordó el problema del desarrollo de sistemas de gestión de flujo de trabajo; el enfoque adoptado ha sido el de los sistemas multi agentes, en particular aquellos que utilizan agentes móviles. Las soluciones propuestas abarcan dos etapas importantes del ciclo de vida de desarrollo de este tipo de sistemas: la especificación formal y la implementación distribuida.

Para la etapa de especificación de los procesos de flujo de trabajo se propuso una metodología de modelado que utiliza el formalismo n-LNS (basado en Redes de Petri) el cual permite obtener un modelo jerárquico y modular. Los modelos de los diferentes niveles construyen de forma separada comenzando por el nivel organizacional (descripción del ambiente), enseguida por el nivel operativo (un agente que se desplaza entre las unidades organizacionales), y por último el nivel operativo, que escribe el plan del agente y el detalle de las tareas y operaciones a realizar por los agentes en el proceso de flujo de trabajo. Los modelos definidos se interrelacionan por medio de la sincronización de transiciones expresada a través de etiquetas.

Siguiendo esta metodología se llega de una manera sistemática a modelos modulares y jerárquicos en los que se distinguen claramente las partes descritas del sistema. Estos módulos, de estructura simple, pueden ser modificados sin afectar significativamente el resto del modelo. Los modelos obtenidos en los casos de estudio realizados fueron probados en simulación con ayuda de la herramienta nLNS Agents.

En la etapa de implementación del sistema de gestión de flujo de trabajo, la metodología presentada permite la definición sistemática de componentes del software soportada por las facilidades de JADE para definir plataformas, agentes, comportamientos, protocolos de comunicación y movilidad de los agentes. .El prototipo desarrollado nos ha permitido comprobar que se al aplicar la metodología se obtienen componentes integrables de manera sencilla en un prototipo; además de que cuenta con todas las ventajas que presenta un prototipo desarrollado en JADE, entre otros que es multiplataforma. De igual forma, en el prototipo del caso de estudio se pudo mostrar que JADE permite la comunicación e interacción entre agentes de una manera muy sencilla ya que soporta muchos de los protocolos definidos por FIPA.

Las metodologías propuestas en esta tesis son un primer paso hacia la estandarización del modelado de sistemas de gestión de flujo de trabajo, dado que a pesar de los esfuerzos de la Workflow Management Coalition aun no existe un estándar conceptual adoptado. La extensión natural de este trabajo es la concepción de una herramienta para el desarrollo de Sistemas de Gestión de Flujo de Trabajo, la cual permita la construcción (edición) de modelos nLNS, la simulación y el estudio de las propiedades de los modelos construidos; también podría generar el software para soportar la ejecución de flujos de trabajo mediante la interpretación de la definición del proceso, manejar la interacción con los participantes. Todo esto cumpliendo con los estándares propuestos por la WFMC en el Modelo de Referencia.

Referencias

[AaA99] Aalst, W. van der and Anyanwu, K.(1999). *Interorganizational Workflows to Enable Business-to-Business E-commerce.* In Proceedings of the Second International Conference on Telecommunications and Electronic Commerce, Nashville, Tennessee.

[AaH02] Aalst, W. van der y A.H.M. ter Hofstede(2002). *YAWL : Yet Another Workflow Language*

[Aal98] Aalst, W. van der(1998). *The Application of Petri Nets to Workflow Management.*The Journal of Circuits, Systems and Computers.

[AalW] Aalst, W. Van der. *Making Work Flow: On the Application of Petri nets to Business Process Management*

[AalWs] Workflow Patterns site, http://is.tm.tue.nl/research/patterns/patterns.htm

[ABH00] Aalst, W. van der, A.P. Barrors, A.H.M. ter Hofstede and B. Kiepuszewsk. *Advanced Workflow Patterns*. In O. Etzion and P. Scheuermann, editors, 7th International Conference on Cooperative Information Systems (CoopIS 2000). Volume 1901 of Lecture Notes in computer science, pages 18-29,Springer-Verlag, Berlin, 2000.

[ABH02] Aalst, W. van der, A.P Barros, A.H.M. ter Hofstede and B. Kiepuszewsk. *Workflow Patterns. QUT Technical report, FIT-TR-2002-02, Queensland University of Technology,Brisbane, 2002.*

[Alyd02] Almeida Canepa Hugo Isidro, *Un sistema de red a tres niveles para el modelado de agentes móviles*. Tesis de Maestría en Ciencias, Ingeniería Eléctrica. CINVESTAV del IPN Unidad Guadalajara, Septiembre de 2002

[Bar03] *Bartosz Kiepuszewski,* Expressiviness and Suitability of Languages for Control Flow Modelling in Workflows, *A dissertation presented to the Faculty of Information Technology Queensland University of Technology, Febrero 2003*

[BGM99] Brewington B., Gray R., Moizumi K., Kotz D., Cybenko G., Rus D., *Mobile Agents for distributed Information Retrieval*, Intelligent Information Agents, Berling: Springer-Verlag, 1999

[CaJG99] Caro, J.L., A. Guevara, A. Aguayo, S. Galvez, *Tecnología workflow aplicada a los sistemas informáticos de gestión hotelera*, Proc of Turitec'99, Málaga-Spain 145-158, (1999)

[CLZ98] Cabri G., Leonardi L., Zambonelli F., *Mobile Agent Technology: Current Trends and Perpectives,* Congresso annuale AICA'98, Napoli (I), November 1998

[DiY00] Dianxiang Xu, Yi Deng, *Modelling Mobile Agent Systems with High Level Petri Nets,*in Proc of the IEEE International Conference on Systems, Man and Cybernetics, pp 3177-3182, Nashville, octubre 2000

[DMH01] Dumas Marlon, Hofstede Arthur H.M.*UML Activity Diagrams as a Workflow Specification Language,* in Proceedings of the UML´2001 Conference

[FIPA] Sitio: http://www.fipa.org/

[FBLM06a] Flores Badillo Marina, López Mellado Luis Ernesto. *Multi Level Models for Workflow Process Automation*, In Proceedings of the 2006 International Symposium On Robotics and Automation ISRA'2006, August 2006.

[FBLM06b] Flores Badillo Marina, López Mellado Luis Ernesto. *Modeling and simulation of interactive mobile agents using a multi-level-net formalism*, In Proceedings of the 5th Mexican International Conference on Artificial Intelligence MICAI'2006, Springer LNAI, November 2006.

[FraWo] Frankie Wong, *"Prey and Predator: A Lesson in Stealth"*

[Hai03] Hai Zhuge, *Workflow- and agent-based cognitive flow Management for distributed team Cooperation, Information and Management Volume 40, No. 5, pp 419-429, mayo 2003*

[HolID] Hollingsworth, David. *The WorkFlow Reference Model 10 Years On*

[JADE] Sitio: http://jade.cselt.it/

[KamJi] Kam-Chuem Jim, *Talking Helps: Evolving Communicating Agents for the Predator-Prey Problem*

[MaB05] Marin Cesar A., Brena Ramon, *Multiagent Architecture for Decentralized Workflow Process Execution,* Technical Report, Center for Intelligent Systems, Tecnologico de Monterrey, marzo 2005

[MarDa] Marius Danca, *Detailed Analysis of a Nonlinear Prey-Predator Model*

[MDW99] Milojicic Dejan, Douglis Frederick, Wheeler Richard, *Mobility: Processes, Computers, and Agents*, Addison-Wesley, 1999

[MHD05] Minhong Wang, Huaiqing Wang, Dongming Xu, *The design of intelligent workflow monitoring with agent technology*, Knowledge-Based Systems, Volume 18, Issue 6, October 2005, pages 257-266.

[NVW98] Nabil R. Adam, Vijayalakshmi Atluri y Wei-Kuang Huang, *Modelling and Analysis of Workflows Using Petri Nets,* Journal of Intelligent Information Systems, Volume 10, Issue 2, pages 131-158, 1998

[NwH96] Nwana, Hyacinth. *Software Agents: an Overview,* Knowledge Engineering Review, Vol. 11, No 3, pp 1-40, September 1996.

[OrA05] Oren Eyal, Armin Haller, *Formal Frameworks for Workflow Modelling,*Digital Enterprise Research Institute Technical Report, Abril 2005

[RPB03] Repetto Marco, Paolucci Massimo y Boccalatte Antonio, *A design tool to Develop Agen-Based Workflow Management Systems,* Proc. Italian Workshop, from Objets to Agents: Intelligent Systems and Persuasive Computing (WOA2003), Villasimius I, 2003

[RZ96] D. Riehle and H. Züllighoven. *Understanding and Using Patterns in Software Development*. Theory and Practice of Object Systems, 2(1):3-13, 1996.

[Sam05] Samuel Garrido, Daniel, *Modelado de Workflow con Redes de Petri Coloreadas Condicionales,* Tesis para obtener el grado de Maestro en ciencias en el Cinvestav México, noviembre 2005

[San04] Sánchez Herrera, Roberto(2004). *Especificación Multinivel de Protocolos de Interacción en Sistemas de Agentes Móviles.* Tesis de Maestría

[Val98] Valk, Rüdiger(1998). *Petri Nets as Token Objets* And introduction to Elementary Object Nets

[ViN03] Villanueva Paredes Norma Isabel. *Modelado Multinivel de Sistemas de procesos por lotes*, Tesis de Maestría en Ciencias, Ingeniería Eléctrica, CINVESTAV del IPN Unidad Guadalajara, Diciembre de 2003

[Vkm04] V.K. Murthy, *Contract-Based Workflow Paradigm For Mobile E-Comerce,* Advanced Simulation, Technologies Conference 2004 (ASTC), Sponsored by the Society for Modelling and Simulation International, 2004

[WalCh] Walid Chainbi, *The Multi-agent Prey/Predator problem: a Petri Net Solution.*

[WfC02] WFMC-TC-1002, *Document Index* , sitio http://www.wfmc.org.

[WfC03] WFMC-TC-1003, Hollingsworth David, *The Workflow Reference Model,* junio 95, sitio http://www.wfmc.org.

[WfC11] WFMC-TC-1011, *Terminology & Glossary,* sitio http://www.wfmc.org.

[WfC12] WFMC-TC-1012, *Workflow Standard – Interoperability Abstract Specification* , sitio http://www.wfmc.org.

[WfC13] WFMC-TC-1013, *Workflow Client Application, Application Programming Interface Naming Conventions,* sitio http://www.wfmc.org.

[WooM02] Wooldridge Michael, *An Introduction to multiagent Systems*, John Wiley & Sons (Chichester, England) ISBN 0 47 149691X. 340 pp. February 2002

[YZW01] Yuhong Yan, Zakaria Maamar, Weiming Shen, *Integration of Workflow and Agent Technology for Business Process Management,* The sixth international Conference en CSCW in Desing, London, Ontario, Canada, julio 2001

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
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 de flujo de trabajo basado en sistemas de agentes móviles

del (la) C.

Marina FLORES BADILLO

el día 30 de Agosto de 2006.

Dr. Luis Ernesto López Mellado
Investigador CINVESTAV 3A
CINVESTAV Unidad Guadalajara

Dr. Félix Francisco Ramos Corchado
Investigador CINVESTAV 2B
CINVESTAV Unidad Guadalajara

Dr. Mario Angel Siller González
Pico
Investigador CINVESTAV
CINVESTAV Unidad Guadalajara